SBGAMES 2008

VII Symposium on Computer Games and Digital Entertainment

Belo Horizonte, MG - Brazil
november 10th - 12th

Track:
Computing
Full Papers

www.sbc.org.br/sbgames


EM CONSTRUCAO

VII Brazilian Symposium on Computer Games and Digital Entertainment
November, 10-12, 2008
Belo Horizonte – MG – BRAZIL

PROCEEDINGS
Computing Track – Full Papers

Published by
Sociedade Brasileira de Computação – SBC

Edited by
Fernando S. Osório
Luiz Chaimowicz
Rosilane Mota
Zenilton Patrocínio

Computing Track - Full Papers Chairs
Fernando S. Osório
Zenilton Patrocínio

SBGames 2008 General Chairs
Luiz Chaimowicz
Rosilane Mota

Universidade Federal de Minas Gerais - UFMG
Pontifícia Universidade Católica de Minas Gerais
PUC Minas

Sponsored by
SBC - Sociedade Brasileira de Computação

Table of Contents

SBGames 2008

PREFACE

Welcome to the VII Edition of the Brazilian Symposium on Computer Games and Digital Entertainment, the SBGames 2008. SBGames is the yearly symposium of the Special Interest Group on Games and Digital Entertainment of the Brazilian Computer Society (SBC). SBGames 2008 is the most important event on game research and development to take place in Latin America, promoted by the Brazilian Computer Society (SBC) with the support of the Brazilian Electronic Game Development Companies Association (ABRAGAMES). This year the symposium brings together students, professors, artists, designers and professionals from several universities, research centers, graphical design centers and game industry.

This volume contains the 25 full papers accepted for the computing track, out of 57 submitted, with an acceptance ratio of 43%. Out of the 25 accepted papers, 16 are in English (64%). We hope this trend will continue, increasing the visibility of the research work being developed in Brazil by the gaming community. The selection process was double blind, and each paper was reviewed by at least 3 experts, enforcing the quality of the reviewing process. Also, the best papers of this conference were indicated to be published (an extended version) into a special edition of IJCGT - International Journal of Computer Games Technology and ACM CIE - Computers in Entertainment.

The SBGames 2008 is composed by 4 main tracks: Computing, Art & Design, Industry and Games & Culture, 2 festivals (Independent Games and Art Exhibition), Poster Exhibitions, Tutorials, Keynote presentations, and other satellite events. The papers from the different tracks, and also, the Posters (short papers) and complementary material from this Symposium have been included within the Proceedings of the Computing Track in the SBGames 2008 Conference CD-ROM.

We would like to thank all authors, whose work and dedication made possible to put together an exciting program. Next, we would like to thank all members of the technical program committee and reviewers, for their time helping us maintain the overall quality of the program.

We would like to wish all attendees an exciting symposium!

Belo Horizonte, November 2008,

Fernando S. Osório & Zenilton Patrocínio
Chairs of the Program Committee - Computing track

VII SBGames - ISBN: 85-766-9217-1

Program Committee

Adelailson Peixoto	UFAL - Universidade Federal de Alagoas
Alexandre Sztajnberg	UERJ – Univ. do Estado do Rio de Janeiro
André Campos	University of Utrecht
Bruno Feijó	PUC-Rio
Cesar Pozzer	UFSM – Univ. Federal de Santa Maria
Christian Hofsetz	Microsoft Corporation
Clauirton Siebra	UFPE - Universidade Federal de Pernambuco
Drew Davidson	CMU - Carnegie Mellon University
Edmond Prakash	Manchester Metropolitan University
Eduardo do Valle Simões	USP/Sao Carlos
Esteban Clua	UFF Universidade Federal Fluminense
Fernando Osório	USP/ICMC – Univ. de São Paulo
Fernando Trinta	UFPE - Universidade Federal de Pernambuco
Flávio Soares C.da Silva	USP- Universidade de São Paulo
Geber Ramalho	UFPE - Universidade Federal de Pernambuco
Jacques Brancher	URI - Campus de Erechim
Jim TerKeurst	University of Teesside
João Comba	UFRGS – Univ. Federal do Rio Grande do Sul
Jorge Barbosa	Unisinos
José Saito	UFSCar – Univ. Federal de São Carlos
Judith Kelner	UFPE - Universidade Federal de Pernambuco
Luiz Chaimowicz	UFMG - Univ. Federal de Minas Gerais
Luiz Gonzaga da Silveira Jr	UNISINOS
Manuel M. Oliveira Neto	UFRGS – Univ. Federal do Rio Grande do Sul
Marcelo Dreux	Puc-Rio
Marcelo Walter	UFPE - Universidade Federal de Pernambuco
Marcio Pinho	PUC-RS - Pontifícia Univ. Católica do RGS
Maria Andréia Rodrigues	UNIFOR - Universidade de Fortaleza
Martin Hanneghan	Liverpool John Moores University
Michael Youngblood	University of North Carolina at Charlotte
Patrícia Tedesco	Centro de Informática - UFPE
Paulo Pagliosa	UFMG – Univ. Federal de Mato Grosso do Sul
Paulo Rodacki Gomes	FURB - Universidade Regional de Blumenau
Romero Tori	Centro Universitário Senac / USP
Sérgio Scheer	Universidade Federal do Paraná
Sílvio Cazella	Unisinos
Soraia Musse	PUC-RS – PUC do Rio Grande do Sul
Waldemar Celes	Tecgraf / PUC-Rio
Zenilton Patrocínio Jr	PUC-Minas - Pontifícia Univ. Católica de Minas Gerais

VII SBGames - ISBN: 85-766-9217-1

Reviewers

Adelailson Peixoto
Alexandre Sztajnberg
André Campos
Bruno Feijó
Cesar Pozzer
Christian Hofsetz
Clauirton Siebra
Denison Tavares
Drew Davidson
Edmond Prakash
Eduardo do Valle Simoes
Esteban Clua
Fernando Osório
Fernando Trinta
Flávio Soares Corrêa da Silva
Fritz Heckel
Geber Ramalho
Gustavo Mello Machado
Jacques Brancher
Jezer Oliveira
Jim TerKeurst
Joao Bittencourt
João Bittencourt
José Saito
Judith Kelner
Kalinka Castelo Branco
Leandro Barros
Leandro Fernandes
Leonardo Schmitz
Luis Fernando M S Silva
Luiz Chaimowicz
Luiz Gonzaga da Silveira Jr
Marcelo Dreux
Marcelo Walter
Marcio Pinho
Marcos Kich
Maria Andréia Rodrigues
Martin Hanneghan
Michael Youngblood
Patrícia Tedesco
Paulo Pagliosa
Paulo Rodacki Gomes
Rafael Rieder
Rafael Torchelsen
Ricardo Nakamura
Roberto Cezar Bianchini
Rodrigo Hahn
Romero Tori
Samir Souza
Sérgio Scheer
Sílvio Cazella
Silvio Melo
Solon Rabello
Soraia Musse
Vitor Pamplona
Waldemar Celes
Wallace Lages
Zenilton Kleber Patrocínio Jr

VII SBGames - ISBN: 85-766-9217-1

SBGAMES 2008

COMPUTING TRACK

TECHNICAL PAPERS

Procedural Animation with Genetic Algorithms and Physics Simulation

Pedro Luchini Bruno Feijó Marco Aurélio Pacheco*

PUC-Rio, Dept. of Computer Science, Brazil
* PUC-Rio, Dept. of Electrical Engineering, Brazil

Abstract

This paper describes a method for automatically animating a virtual character based on the topology of its musculoskeletal system. Plausible and realistic movements can be computed for any kind of character; no assumptions are made about its body structure. Arbitrarily-shaped creatures or robots, thus, learn how to move in the same way real-life animals did: through evolution.

Keywords: artificial intelligence, biologically-inspired computing, procedural animation, genetic algorithms, physics simulation

Authors' contact:
{pluchini,bruno}@inf.puc-rio.br
*marco@ele.puc-rio.br

1. Introduction

Procedurally-generated content has become important in the past years as a means of reducing designer workload and producing games with smaller footprints. Additionally, these techniques are useful when dealing with unpredictable input from the player, such as user-generated content.

This paper proposes a method of animating characters whose bodies are represented by a series of constraints (bones, joints) and force actuators (muscles) driven by physics simulation. The character's "brain," then, will send a sequence of commands to each muscle, causing the body to move. This way, the animation can be modeled as a list of instructions (a program) that is executed by its body.

Clearly, characters with different body structures will need different programs to animate properly. It is possible, nevertheless, to establish very simple and broad goals for a class of animations regardless of the character's body structure, such as "maximize horizontal speed" to describe a running animation.

This approach to character animation has the following characteristics:

- The animation is simple to represent;
- Displaying and evaluating the animation is straightforward;
- Creating the animation itself, however, is not.

These three traits suggest the employment of *genetic algorithms*, an optimization technique well-suited for finding approximate solutions based on heuristics.

2. Related Work

There is an extensive list of publications related to character physics. Particularly worthy of note are [Jakobsen 2001] and [Witkin and Kass 1988] who focus on constraints capable of simulating bones, joints, and other structural elements of a live character.

Genetic algorithms have been used by [Machado and Cardoso 2002] and [Cope 2005] in the field of artificial creativity with remarkable success. They have shown that the pseudo-random nature of G.A.s can create aesthetically-pleasing works in a manner that traditional algorithms cannot, with surprising results that often startle the developers themselves.

3. Modeling the Character

Since we are trying to simulate real-life biologic systems, it makes sense that we use the same components that are found in real-life animals to represent the virtual character: bones, muscles, joints, etc. These components can be divided in two categories: *passive* and *active*.

3.1 Passive Components

Passive components are those that cannot be controlled by the character's brain. Just like we are unable to change our bones' length, the character should not be allowed to change his. Some examples of passive components are:

- **Bones** can be represented by a simple bilateral constraint between two particles, which is easy and fast to implement with Verlet integration [Jakobsen 2001]. If a more robust physics simulation engine is available (and needed), more complex bone structures can be represented by rigid bodies.
- **Joints** keep bones connected to each other. [Baltman and Radeztsky Jr 2004] propose a method for constraining the angle between bones, as well as their spatial orientation.

3.2 Active Components

Active components are those that can be controlled by the brain.

- A **spring** applies linear force when contracted or stretched past its natural length (Figure 1). To achieve locomotion, the brain can send commands to change the spring's natural length (Figure 2).

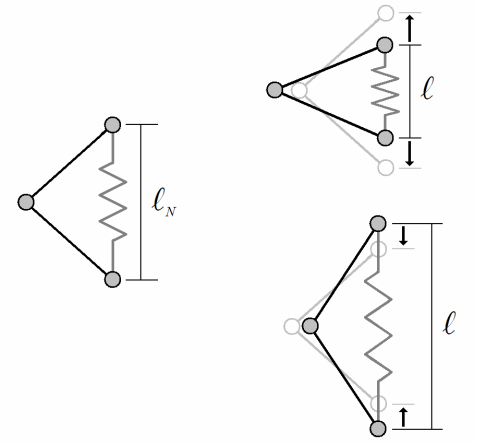


Figure 1: A spring exerts a force whenever it is stretched or compressed. As per Hooke's law, the force is proportional to the spring's deformation (ℓ_N - ℓ) and its force constant (k).

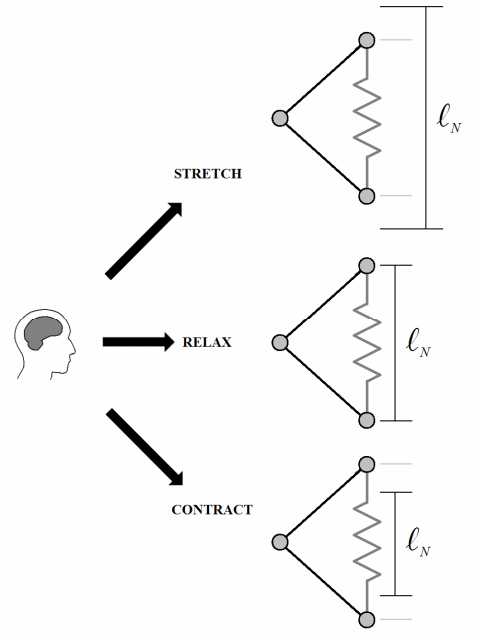


Figure 2: The brain sends commands to the spring, telling it to change its natural length. This pulls the bones together (by decreasing ℓ_N) or pushes them apart (by increasing ℓ_N).

Springs are a rather crude approximation of how muscles really work. However, they are useful as a simulation of abduction/adduction movements (such as pulling the knees together), which would otherwise require very complex modeling.

- A **torsion spring** is the angular equivalent of a spring. It connects to a joint (the junction of two bones) and applies a torque when it is contracted or stretched past its natural angle (Figure 3). The brain can send commands to change the torsion spring's natural angle.

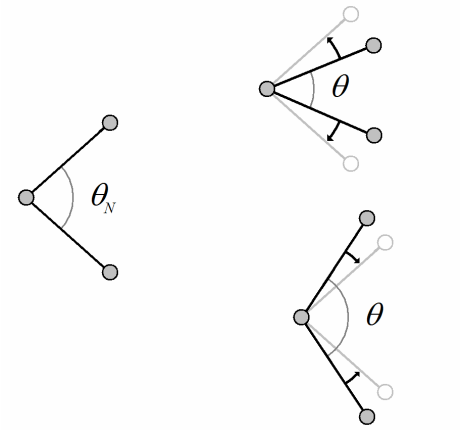


Figure 3: A torsion spring exerts a torque whenever it is stretched or compressed. As per the angular version of Hooke's law, the torque is proportional to the torsion spring's angular deformation (θ_N - θ) and its force constant (k).

- A useful (if somewhat unrealistic) component to have in a character is a **claw**, that is, an extremity that is capable of gripping walls and floors to anchor the body's movement. When the brain sends a *grip* command to the claw, it will "stick" to whatever surface it is currently touching (and thus become unable to be moved from that spot). When the brain sends the claw a *release* command, it will cease being "sticky." A claw that is in a "sticky" state but is not touching any surface is unaffected.

Springs and torsion springs have a theoretically infinite number of possible states – any real number can be set as their natural length or angle. Our test application simplified it down to only three states: The *relaxed* state will set the natural length to the value it had at the beginning of the simulation, the *contracted* state will set the natural length to half that value, and the *stretched* state will set it to twice that value. It was a somewhat arbitrary choice; depending on the application, this set-up might take a lot of tweaking to achieve good results.

4. Representing the Animation

Because the character moves as a result of the commands issued by its brain, the animation is represented by a *sequence of commands* that are issued one after the other. This is very similar to imperative programming languages (such as assembly, or machine

code), so we will refer to these commands as *op-codes*. We can think of the animation as a program that is executed one op-code at a time.

There is no reason to pre-define the size of the program – let the genetic algorithm figure that out on its own –, so the sequence should be allowed to have a variable length. Depending on the animation being generated, it may also be necessary to loop the program – restarting from the first op-code when the end of the program is reached – to simulate an endless cycle (such as a walking animation).

Figure 4: A short animation with ten op-codes.

5. Playing Back the Animation

Physics simulation systems are notoriously fickle when it comes to accuracy and consistency. It is often the case that running the same simulation on different machines will result in different outcomes.

Numeric methods (such as those used in real-time physics simulation) are inherently inaccurate, but we can at least ensure they are *consistent* by employing what [Valente et al. 2005] describe as the "fixed-frequency deterministic game loop." This technique is based on the fact that if we always update the game state at a fixed number of times per second (with the same delta-time, or *dt*), then the simulation will consistently produce the same result:

```
accumDT ← 0
function GameTick()
   dt ← time elapsed since last frame
   accumDT ← accumDT + dt
   while accumDT ≥ FIXED_DT do
      GameStateStep(FIXED_DT)
      accumDT ← accumDT – FIXED_DT
   end while
end function
```

To play back the animation, one must define a time interval between op-code executions, and then run the animation program side-by-side with the physics simulation:

```
accumOpDT ← 0
function GameStateStep(dt)
   // Run animation program:
   accumOpDT ← accumOpDT + dt
   while accumOpDT ≥ OPCODE_DT do
      Take next op-code from the program.
      Interpret its command.
      Modify the state of springs/claws.
      accumOpDT ← accumOpDT – OPCODE_DT
   end while

   // Run physics simulation:
   PhysicsStep(dt)
end function
```

The exact values of `FIXED_DT` and `OPCODE_DT` should be adjusted to suit the application. Our test program used `FIXED_DT` = 1/500 and `OPCODE_DT` = 1/200 (in seconds), which ensured a high degree of accuracy in the physics simulation.

6. Creating the Animation

Now that the animation can be represented, stored, and played back, all that is left to do is understand how it is created. This is where genetic algorithms come in.

6.1 An Introduction to Genetic Algorithms

A genetic algorithm is a search technique used to find optimal (or approximate) solutions to a problem based on a heuristic measure of the solution's quality.

Each solution is represented as an individual in a population of candidate solutions. Each individual holds a *chromosome*, a representation of its solution; the chromosome can be evaluated by an *objective function* to determine its *fitness* (a real number).

At first, a population of individuals is created with random chromosomes, and each of them is evaluated by the objective function to discover its fitness. Then, some individuals of the population are selected to be modified and included in a new population of individuals, which forms the next *generation* of solutions. These steps are repeated until either a pre-set number of generations have been evolved, or a desired fitness score has been reached.

The fitness of an individual determines the probability that it will be selected for modification and inclusion in the next generation. Thus, individuals with high fitness are more likely have their genes included in the final solution.

Individuals can be modified by either *crossover* or *mutation*.

- **Crossover** is an operation where the chromosomes from two individuals (parents) are combined. This results in two new individuals (children) with genetic material that is related, but not identical, to the originals'. The children then replace their parents in the new population.
- **Mutation** is an operation where a single individual has its chromosome modified in some random manner.

Crossover is typically set to occur with a high probability, usually 80% or more. Mutation, on the other hand, has a deleterious effect on the evolution if it happens too frequently; it is usually set to occur with a low probability, rarely exceeding 10%.

There is an entire field of research dedicated to genetic algorithms, with several existing techniques to improve their performance. A detailed look into that field is beyond the scope of this paper; for a more thorough introduction we recommend [Davis 1991].

6.2 The Animation Defined in G.A. Terms

The solution we are trying to optimize is the animation program. In our G.A., then, the chromosome is represented by a sequence of op-codes. Each individual in the population can be evaluated by playing back the animation; the fitness score depends on what type of animation is being sought. Some examples:

- **Walking cycle:** Run the animation program for a few seconds, looping it every time it reaches the last op-code. The fitness score is the total horizontal displacement of the character.
- **Jumping animation:** Run the animation program for a few seconds, without looping it. The fitness score is the maximum height achieved by the character.

The crossover modifier of choice is the *one-point crossover* (see Figure 5). It is useful because it recombines the two animations without completely destroying the parents' sequencing of op-codes.

Mutation modifiers can be as random as the application requires. Here are some suggestions:

- **Constructive mutator:** Inserts random op-codes at random points in the chromosome.
- **Destructive mutator:** Removes op-codes from random points in the chromosome.
- **Replacing mutator:** Takes an op-code at a random point and replaces it with another random op-code.
- **Swapping mutator:** Takes two op-codes at random points and swaps their positions.

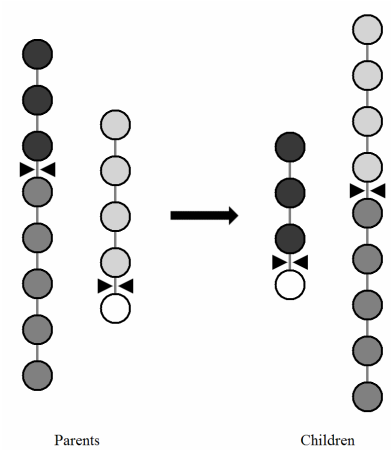


Figure 5: One-point crossover. Each of the parent chromosomes is "cut" at a random point to create two sub-sequences. By swapping the sub-sequences, two new chromosomes are created.

6.3 Encoding Op-Codes in the Chromosome

An op-code can be represented by something as simple as a single integer number. Let us imagine a character whose body is composed of 2 springs and 3 claws. The op-codes can be represented thusly:

1. Set spring #1 state: CONTRACTED
2. Set spring #1 state: RELAXED
3. Set spring #1 state: STRETCHED
4. Set spring #2 state: CONTRACTED
5. Set spring #2 state: RELAXED
6. Set spring #2 state: STRETCHED
7. Set claw #1 state: GRIPPING
8. Set claw #1 state: RELEASED
9. Set claw #2 state: GRIPPING
10. Set claw #2 state: RELEASED
11. Set claw #3 state: GRIPPING
12. Set claw #3 state: RELEASED

That is, there are twelve valid op-codes for that character. Generalizing for a character with N_S springs and N_C claws, there will be $3N_S + 2N_C$ valid op-codes. (Naturally, this quantity will be different if the character includes other active components, such as torsion springs.) Any integer number greater than the number of valid op-codes can be interpreted as a "nothing" op-code.

"Nothing" op-codes are extremely important during the evolution of an animation program. A sequence of op-codes that is excessively dense with commands will cause the character to thrash its limbs in every direction. At worst, this will prevent the character from actually moving; at best, the result will be unsightly.

We suggest that one third of the op-codes in the programs be "nothing" op-codes; this ratio produced the best results in our test application. Every time a random op-code had to be created (in the initialization of the first generation, in the constructive mutator, and

in the replacing mutator), our application called the following function:

```
function MakeRandomOpCode()
   return rand() * 1.5 * (3*N_S + 2*N_C)
end function
```

where `rand()` is a function that returns a random number between zero and one. The return from `MakeRandomOpCode()` should then be cast to an integer type to be stored in the chromosome.

7. Conclusion

We have shown that animations can be represented by a sequence of commands sent from a character's brain to its body. There aren't any traditional algorithms capable of deriving this sequence of commands, so we suggest the use of a genetic algorithm instead. As is the hallmark of artificial intelligence, the results are somewhat unpredictable, startling, and require significant tweaking before they look "just right." On the other hand, this approach is extremely versatile, being capable of animating any kind of character with very little input from a human operator.

One clear disadvantage of the method proposed is that the generated animations will only apply to the environment where they were evolved – a walking animation that evolved on a flat floor will be useless if the character is confronted with a staircase. As a direction for future work, the authors propose incorporating *senses* into the model, turning it into an implementation of the genetic programming paradigm [Koza 1992]. Specifically, *touch* (whether a part of the character is touching a surface) and *proprioception* (the angles of the character's joints) are simple to implement and can be used as inputs for the animation program.

Acknowledgements

The authors would like to thank Waldemar Celes for his assistance with implementing the physics simulation system we used in the test application.

The software for this work used the GAlib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology.

References

JAKOBSEN, T., 2001. Advanced Character Physics. *Proceedings of the 2001 Game Developers Conference*.

WITKIN, A. AND KASS, M., 1988. Spacetime constraints. *Proceedings of the 15th International Conference on Computer Graphics and Interactive Techniques*, 159-168.

MACHADO, P. AND CARDOSO, A., 2002. All the Truth About NevAr. *Applied Intelligence*, 16 (2), 101-118.

COPE, D., 2005. Computer Models of Musical Creativity. MIT Press.

BALTMAN, R. AND RADEZTSKY JR, R., 2004. Verlet integration and constraints in a six degree of freedom rigid body physics simulation. *Game Developers Conference 2004*.

VALENTE, L., CONCI, A. AND FEIJÓ, B., 2005. Real Time Game Loop Models for Single-Player Computer Games. *Conference Proceedings of the 4th Brazilian Workshop on Computer Games and Digital Entertainment*, 89-99.

DAVIS, L., 1991. *Handbook of Genetic Algorithms*. New York: Van Norstrand Reinhold.

KOZA, J., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

SDK *GAMEPLAY* - FERRAMENTA VOLTADA PARA EDIÇÃO DE *GAMEPLAY*

Leandro B. Motta[1] Julio A.A. Contreras*[1] Fernando S. Osório[2]

[1]UNISINOS – Univ. do Vale do Rio dos Sinos, Ciências Exatas e Tecnológicas, RS - Brasil.
[2]USP – ICMC – Instituto de Ciências Matemáticas e de Computação, SP – Brasil.

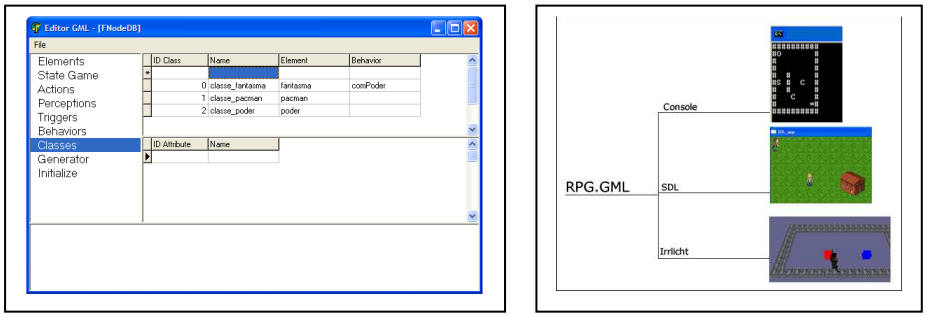


Figure 1: Projeto SDK Gameplay. (a) Editor de gameplay. (b) Visualização da portabilidade do formato GML.

Resumo

Este trabalho descreve o projeto e o desenvolvimento de um conjunto de ferramentas para facilitar a implementação de qualquer tipo de *gameplay* em jogos digitais. Dentre essas ferramentas destaca-se a criação de uma biblioteca denominada *GameplayLib*, que auxilia na importação das regras no formato XML para estruturação dos dados (e.g. regras, agentes, eventos) e um editor de *gameplay* com suporte visual para poder simular um jogo em tempo real. Este projeto visa criar uma nova ferramenta RAD (*Rapid Application Development*) para jogos.

Palavras-chave: RAD; inteligência artificial; *gameplay*.

Contatos com Autores:
{fosorio, lmbarros}@gmail.com
*juliocontreras@gmail.com

1. Introdução

Este trabalho tem como função principal ajudar a prototipar, automatizar e simplificar a criação das regras de um jogo, acelerando a produtividade no desenvolvimento de jogos. O projeto foi desenvolvido de forma que pessoas com menos conhecimento tecnológico possam também usufruir das ferramentas por ele oferecidas. O desenvolvimento rápido de protótipos de jogos permite que decisões de continuação, alteração ou descontinuação de um projeto possam ser tomadas antes de terminado o projeto por completo.

Sempre que se desenvolve um jogo é necessário pensar em vários elementos e passos, como o ponto de início, meio e fim, o número de fases, a derrota, as vitórias, os inimigos, o cenário, os obstáculos, etc. Todos estes elementos demandam um investimento de tempo considerável na produção de um jogo. Algumas vezes, somente no final do processo de desenvolvimento se descobre que o jogo não é tão divertido quanto foi planejado, o que exige um tempo adicional em desenvolvimento, adaptações e testes.

Uma pessoa encarregada de criar as regras de um jogo não tem a obrigação de ter um conhecimento tecnológico profundo, pois um jogo é constituído de regras e de suas interações, cuja definição independe das tecnologias usadas na implementação. Na verdade, muitas vezes, as regras são adaptadas ou aproveitadas diretamente de jogos não-eletrônicos. Por exemplo, as regras de um jogo de futebol de simulação são as mesmas do jogo no mundo real.

Existem pessoas com conhecimento de jogos e de suas interatividades, mas que não dominam o conhecimento tecnológico, e vice-versa. As ferramentas propostas por este trabalho se encarregam de realizar a junção entre ambos para a realização de um jogo. Por exemplo, um desenvolvedor (programador) poderia ter acesso a essas regras e utilizá-las em um jogo, mesmo não conhecendo o autor delas. Esse fator ajudaria a ambos, pois a necessidade de uma pessoa pode ser o negócio de outra.

Depois de construídas as regras, o próximo passo é a prototipação de um jogo, utilizando-se um artefato que executa as regras e suas interações, em conjunto com as tecnologias e seus recursos. As tecnologias utilizadas no protótipo podem ser as mesmas do jogo final, ou podem ser mais simples (por exemplo, um jogo baseado em gráficos 2D poderia ter um protótipo apenas em caracteres, no console). O objetivo de prototipar é realizar testes, incluindo a diversão do jogo, antes de concluído o desenvolvimento, para decidir o andamento do projeto.

Como as ferramentas criadas neste trabalho não são dependentes das tecnologias do jogo, elas favorecem a prototipação, permitindo que diferentes artefatos possam ser utilizados na realização do jogo.

Visando aos benefícios para aquele que vai construir o jogo e não as regras, as ferramentas desenvolvidas permitem a automatização no gerenciamento das regras do jogo, a separação das tecnologias externas com as regras e o aumento da reusabilidade para futuros projetos.

Duas das principais ferramentas desenvolvidas são um editor de regras e uma biblioteca que permite utilizar as regras criadas com este editor. O gerenciamento das regras do jogo é proporcionado pela biblioteca e suas respectivas classes, que irão importar os arquivos gerados pelo editor.

As ferramentas apresentadas são capazes de promover uma separação das tecnologias externas com as regras de forma transparente, pois os arquivos gerados pelo editor são interpretados pela biblioteca como uma caixa preta, somente se preocupando com as entradas e saídas, que no caso são as *Actions* e *Perceptions* do jogo.

O aumento da reusabilidade de código para futuros projetos também pode ocorrer, pois, uma vez desenvolvido o jogo com a biblioteca, na construção do próximo jogo muitas classes poderão ser reusadas. Pode-se inclusive, caso necessário, realizar a construção de um conjunto de código de programação para a reusabilidade em outros jogos, denominado de *framework*. No caso deste trabalho isso acabou ocorrendo, pois há dois *engines* (um com gráficos 2D, outro em console) para o mesmo jogo, usando o mesmo *framework*. Caso se desejasse portar o jogo para terceira dimensão, seria preciso usar ou construir um *engine* para esta, mas as regras permaneceriam as mesmas; caso o *framework* fosse compatível, poderia ser usado também.

Considerando os problemas descritos anteriormente e considerando que as regras e a mecânica do jogo são elementos fundamentais, pretende-se, por meio deste trabalho, apresentar uma proposta de solução: parte do peso e da complexidade da programação pode ser retirada, trocando-se por uma ferramenta que seja mais intuitiva e que ajude a compor e editar a jogabilidade.

2. Embasamento Teórico

2.1 Gameplay

Todas as experiências de um usuário durante a interação com um jogo são denominadas de *gameplay*. Geralmente, este termo na terminologia dos videogames é usado para descrever a experiência total de jogar, ou seja, a jogabilidade, que exclui fatores como gráficos, som e *storyline* (linha de história). Quando se está pulando, se esquivando dos ataques do inimigo ou acelerando um carro de corrida, todas essas ações e experiências estão incluídas na jogabilidade, ou *gameplay* [Rollings 2004].

Para incorporar o anteriormente dito, neste trabalho criou-se uma estrutura de linguagem, de forma que qualquer tecnologia pudesse interpretar as regras dentro de um jogo. Esta estrutura se armazena em arquivo no formato XML, que foi dividido em áreas para oferecer ao criador das regras uma facilidade na organização das idéias de forma simples, definindo seus elementos, ações, percepções, estados de jogo e interações.

O *gameplay* participa desta estrutura ajudando nas interações por meio de eventos sendo ativados por *actions* (como pular, atirar, correr, etc.) e *perceptions* (como acertar, pegar item quando colidir, etc.), modificando *attributes* (força, pontos de vida, etc.) e *objects* (personagens, carro de corrida, etc.).

2.2 Mecânica de jogo

A mecânica de jogo é uma construção de regras pré-determinadas para produzir um entretenimento agradável ou *gameplay*. Todos os jogos usam uma mecânica; entretanto, as interações e os estilos os tornam diferentes [Rollings 2004].

Um exemplo é o jogo de Xadrez, que tem a alternância dos jogadores em turnos como um importante elemento de mecânica de jogo. Isso quer dizer que um jogador deve esperar a jogada de seu oponente para jogar. Essa metodologia se usa em alguns jogos famosos do gênero de estratégia, como Civilization. Esta é uma mecânica diferente dos jogos eletrônicos de combate em primeira pessoa, em que as regras são executadas em tempo real, podendo os dois ou mais jogadores realizar tarefas ao mesmo tempo.

Em geral, a mecânica do jogo é um projeto que visa permitir que pessoas joguem, tendo uma diversão e/ou ganhando uma experiência.

2.3 Engine

Engine ou motor de jogo é um conjunto de bibliotecas que auxiliam no desenvolvimento de jogos, para videogames e/ou computadores rodando dentro de sistemas operacionais. Este conjunto de tecnologias pode incluir gráficos 2D e/ou 3D, detecção de colisão, linguagem de *script*, sons, física, inteligência artificial e redes [MOTOR DE JOGO 2008].

Um motor de jogo pode ser denominado também de *game engine*, ou *engine.*

2.4 XML

XML (*eXtensible Markup Language*) é um formato para representação e armazenagem de conteúdo de forma hierárquica, conhecida pela sua criação de nodos sem limitação, validação de estrutura, fácil entendimento humano e portabilidade [Harold 1999].

Esse formato permite que uma aplicação possa criar um arquivo XML e que outro aplicativo distinto possa ler esses mesmos dados sem restrição. Por essas razões foi escolhida esta estrutura para o desenvolvimento da linguagem das regras.

2.5 SDK

A sigla SDK (*Software Development Kit)* significa kit de desenvolvimento de *software*. Estes kits normalmente contêm documentação, códigos, bibliotecas e ferramentas para o auxílio no desenvolvimento de *softwares* [SDK 2008].

Empresas de grande porte desenvolvem SDKs para que programadores externos, que não têm uma ligação direta com a empresa, possam usá-las para o de desenvolvimento de *softwares*.

No trabalho desenvolvido, houve a necessidade de criar uma biblioteca de gerenciamento das regras, um editor para a edição das regras e um *engine* para a demonstração dos dois jogos, atribuindo a esse conjunto de ferramentas o nome de SDK *Gameplay*.

3. Trabalhos Relacionados

Existem outros trabalhos relacionados com ferramentas RAD ou editores para um tipo de mecânica específico, mas usualmente essas ferramentas não são genéricas. Este capítulo apresenta alguns desses trabalhos e ferramentas, destacando que, neste trabalho, ao contrário dos outros, buscou-se a criação de uma ferramenta genérica, tipo RAD, voltada para a edição de *gameplay*.

3.1 Regras de jogo em XML

Na literatura foi encontrada uma proposta de modelagem de dados para a criação das regras de jogo, por meio de arquivos em formato de XML [CAMOLESI; MARTIN 2005], em artigo apresentado no SBGames 2005. Esse trabalho reforça a importância do tema aqui abordado, e também serviu de referência para os estudos e desenvolvimentos relacionados a este trabalho de conclusão.

Analisando esse modelo estudado, o método de estruturação do arquivo em relação a outros arquivos XML, foi encontrado um modelo intuitivo que contém elementos e interações para criação das regras de um jogo. Isso foi de grande importância para a construção da nossa estrutura, visando uma simplicidade visual, sendo que uma pessoa com menos conhecimento tecnológico pudesse entender o documento, não perdendo a complexidade das regras e suas interações.

3.2 No mercado de games

No mercado existem diversas ferramentas RAD [Martin 1991] ou ferramentas de desenvolvimento rápido usadas para o desenvolvimento de *softwares*. Dentro desse nicho devem ser destacadas as que incluem a mecânica de jogo no seu sistema, como os aplicativos Game Factory [CLICK 2007] e Game Maker [YOYO 2007]. Com esses *softwares* RAD podem-se criar jogos por meio de uma interface simples e intuitiva, praticamente sem a necessidade de programação. Entretanto, essas ferramentas são limitadas, pois algumas desenvolvem somente jogos em duas dimensões, além de serem desenvolvidas como pacotes fechados, o que dificulta a combinação destes com outras tecnologias.

No mercado de jogos em terceira dimensão, pelo que foi constatado, existe a ferramenta Dark Basic, que necessita de um conhecimento de programação fundamentada em Basic. Há também o Game Maker, com sua interatividade para pessoas que têm menos conhecimento; porém, pela sua proposta comercial acaba criando uma dependência da tecnologia com as regras, tornando o usuário dependente da tecnologia desenvolvida pela empresa criadora desta ferramenta, diferente da tecnologia proposta.

Existe, portanto, uma carência de ferramentas RAD modulares e abertas (para 2D ou 3D) que permitam programar e testar os processos, ou uma ferramenta que permita automatizar amplamente o desenvolvimento de qualquer jogo.

3.3 Game Maker

Game Maker (que significa “criador de jogos”) é um *engine* proprietário. Com suporte a uma linguagem de *script*, renderização 2D por *tiles* isométricos e retangulares e suporte limitado a 3D. Todos os recursos dos jogos são organizados em pastas dentro do programa, que inclui pequenos programas para criar seus recursos, como editores de imagens, sons, *scripts* e fases [YOYO 2007].

O Game Maker permite ainda salvar os recursos criados para que possam ser usados em outros jogos ou

fora do programa e importar ações adicionais para estender as funções do programa [YOYO 2007].

Foi analisado o funcionamento desta ferramenta em comparação com outras descritas; verificou-se que pessoas que não têm um conhecimento técnico profundo podem desenvolver os jogos simples. Porém, caso se queira criar alguma idéia mais complexa, ela precisará ser programada em *scripts*.

Este *engine* contém entrelaçadas as regras de jogo. Esse fator é negativo, pois o código é fechado, não permitindo acesso a modificações e alterações. Essa característica cria uma dependência da tecnologia para o usuário, que não pode acompanhar as novas tecnologias emergentes. Se o usuário criou um jogo no Game Maker e quiser portar para outras tecnologias terá que desenvolver a lógica do início.

3.4 Dark Basic

Dark Basic é um *software* proprietário para o desenvolvimento de jogos para computador. Tem uma linguagem própria baseada em BASIC com instruções próprias. Possui suporte para Pixel Shader e Vertex Shader, suporta modelos animados de muitos formatos e usa os recursos do DirectX 9 para renderização gráfica [HARBOUR; SMITH 2003].

Neste trabalho foi analisado o funcionamento da ferramenta Dark Basic em comparação com outras descritas. Nesta ferramenta, o usuário precisa ter um conhecimento na área de programação, descartando pessoas que não dominam a área técnica, mas têm o desejo e o conhecimento de criar as regras e suas interatividades.

Este *engine* contém os mesmo problemas de entrelaçamento com as regras de jogo que o Game Maker, sendo um fator negativo. O Dark Basic contém técnicas avançadas de renderização, podendo criar jogos com melhor aparência; por outro lado, tem menos recursos automáticos de regras e interatividades que o Game Maker, deixando-o menos interativo.

4. Metodologia

Este trabalho se divide em três projetos: uma biblioteca que controla as regras, um *engine* que controla as tecnologias externas e um editor de jogos.

A construção do SDK – *Software Development Kit* (Figura 1) está dividida no editor de *gameplay* e na biblioteca *GameplayLib*. Por meio do editor pode-se criar e alterar a mecânica de jogo, gerando um arquivo em XML. Uma vez exportado, este deve ser conectado junto ao jogo com a biblioteca de controle de *gameplay*, podendo assim controlar os elementos do jogo e jogar (executando a simulação, visto que o jogo é considerado uma aplicação de simulação em tempo real).

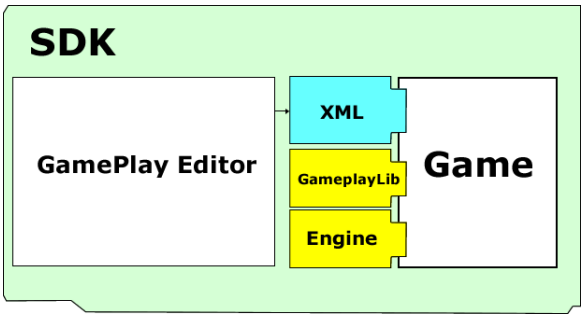


Figura 1 – Organização do SDK e comunicação do editor com o jogo; pode-se ver que o editor está exportando o arquivo XML para o jogo, que o utiliza em conjunto com a biblioteca *GameplayLib* e o *engine*.

4.1 GameplayLib

A biblioteca *GameplayLib* (Figura 2) contém uma série de componentes que ajudam a desenvolver regras dinâmicas, sendo usada para a criação das regras de forma genérica. Dessa forma pode-se importar um arquivo em formato XML, que é traduzido para este conjunto de classes (representado na cor verde na Figura 2). Depois disso, as tecnologias externas podem fazer uso de seus recursos por meio da classe *gameplay*, podendo escutar as execuções (ativações) das regras, as criações e destruições dos *objects*, tudo isto por meio da classe *listener*. Com isso, o desenvolvedor tem um controle completo do jogo, abrindo uma série de possibilidades; cada tecnologia diferente, inclusive, poderá ter um *listener* diferente para organizar melhor seu código.

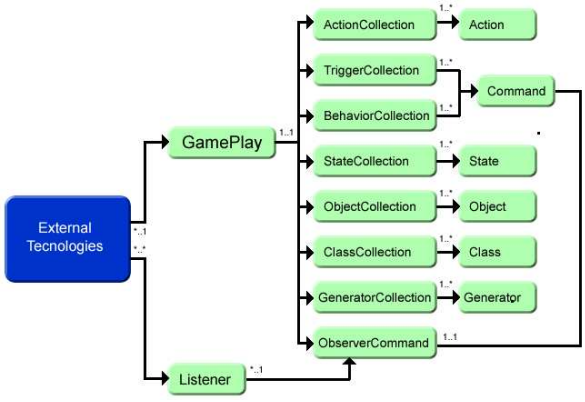


Figura 2 – Organização da biblioteca *GameplayLib*; a classe *gameplay* é a responsável pelo gerenciamento da biblioteca, podendo usar as classes conectadas a ele; *listener* é a classe que interage com as tecnologias externas.

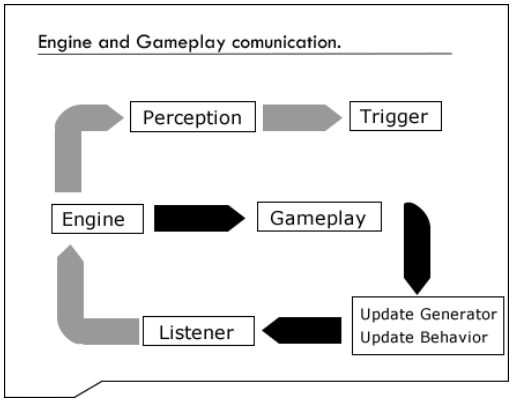


Figura 3 – Ciclo de comunicação do *engine* com a classe *gameplay*. Inicia com o *engine* ativando *gameplay*, que por sua vez atualiza o *generator* e o *behavior*, que por sua vez escuta os *commands* executados, gerando o *engine perceptions* de volta para o *trigger*.

Como exemplo, pode-se tomar o jogo do Pacman. O criador das regras deve se preocupar em criar elementos de interatividade, como Pacman, Fantasma, Ponto, Vitamina e Parede, que são aqui denominados de *elements*. Cada *element* pode ou não se ligar com uma *class*, caso haja necessidade de criação de variáveis auxiliares. No caso, o Pacman necessita das variáveis auxiliares para realizar a contagem de pontos e verificar se ele está em um estado em que consegue matar os fantasmas (após pegar a vitamina). Cada *element* pode conter ações que são usadas na interação com o jogo. No exemplo, o jogador pode realizar ações como movimentos para cima, para baixo, para a esquerda e para a direita, denominadas de *actions*. As *actions* normalmente são ativadas pela entrada de dados (teclado, mouse, *joystick*, etc.), que oferece o *engine* e enviada para a biblioteca que contém as regras do jogo denominada *GameplayLib*. Esta, por sua vez, executa as regras que se referem à ativação da *action* e ao *element* que a ativou (no exemplo, o Pacman é o *element* e a *action* é andar para a frente). A biblioteca avisa ao *engine* que o Pacman andou, e este move o personagem na tela. Caso existia uma parede na frente e ele não possa andar, o *engine* envia uma *perception* para a biblioteca avisando que o Pacman colidiu com a parede, podendo gerar ou não uma *action* de volta para o *engine*.

Este é o elo de comunicação entre o *engine* e a *GameplayLib*. Todas as *perceptions* são enviadas mediante eventos do tipo *trigger* (evento instantâneo). Exemplo de *perception* é o Pacman colidindo com a parede, fantasma, o ponto ou a vitamina. Dentro dos *triggers*, pode-se modificar o *state game*, as variáveis das *classes* e a ativação de *behaviors*. Caso o Pacman pegue uma vitamina, ativa-se um *trigger*, que por sua vez ativa um *behavior* (evento não-instantâneo) de 30 segundos durante os quais ele pode matar os fantasmas. A cada ciclo de jogo existe um gerador de agentes denominado de *generator*, que percebe que quando morrem quatro fantasmas deve nascer um número randômico entre um e quatro, mas nunca deve-se ultrapassar quatro fantasmas.

4.1.1 Trigger

Triggers são eventos ativados, de modo automático ou manual, quando houver uma ação (*action*) em relação a um *state game*, ou ao ocorrer uma interação com o mundo virtual, ou vindo de um *object* secundário com uma *perception*. Uma vez ativado, executa os *commands*, que posteriormente desativam-se automaticamente.(Figura 4).

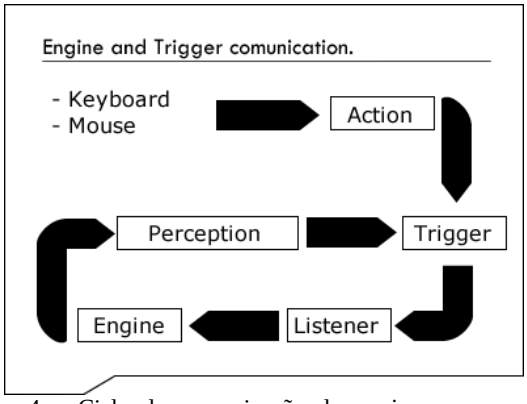


Figura 4 – Ciclo de comunicação do *engine* com o evento *trigger*. Inicia com a entrada de dados ativando uma *action*, sendo enviada para um *trigger*. O *engine*, por sua vez, escuta os *commands* executados gerando *perceptions*, que voltam para o *trigger*.

Dentro deste evento podem existir n *commands*, que contêm as regras do jogo. Estas regras serão explicadas melhor na sessão (4.1.3 *Command*). Tanto as *actions* como as *perceptions* podem ser executadas com um ou com n *objects*, podendo realizar múltiplas interações. Cabe ainda destacar que cada *trigger* pode ainda acionar outro *trigger*, ativando múltiplos eventos.

Os eventos podem ser filtrados por um determinado *state game*, *perception*, *action* e *object*. Assim, o desenvolvedor tem controle para ativar eventos somente para um determinado estado. Por exemplo, em um jogo de RPG combate, a *perception* só se ativa quando recebe um ataque e se este ataque é de um *element* do tipo inimigo. Isso vale para os eventos do tipo *trigger* e *behavior*.

4.1.2 Behavior

Behaviors são eventos ativados automaticamente quando um *object* tem escolhido seu *behavior*. Assim, cada *object* pode ativar somente um *behavior* por vez. Uma vez ativado, executa os *commands*, não se desativando logo a seguir. É necessário ordenar a desativação por meio do *object*, *behavior* ou *trigger*.

Dentro desse evento podem existir n *comamnds*, que contêm as regras do jogo. Essas regras serão explicadas melhor na sessão 4.1.3 *Command*.

Por meio de um *behavior* podem-se ativar n *triggers*, dando uma maior flexibilidade das regras. Um exemplo de *behavior* é o planejamento e execução de trajetórias (e.g. A* - *A Star*).

4.1.3 Command

Commands são regras de estados que interagem com um *object* primário ou secundário. *Object* primário é aquele que executa o evento, e o secundário é usado somente no evento *trigger*, quando for executada uma *action* ou *perception*.

Todas estas regras interagem com os *attributes* de um *object*, podendo modificar o jogo. Um exemplo é ganhar pontos em um jogo, o personagem deve ter um *attribute* deniminado ponto. Ele só pode alterar esses *attributes* através dos operadores.

Existem operadores do tipo somadores, multiplicadores e divisores. Esses operadores são usados junto aos *attributes* de um *object* primário ou secundário, ou mesmo o *command* interagindo com dois tipos de *objects*, seja alterando ou comparando seus *attributes*. Exemplo: O símbolo "@+@S" significa que "@" é *attribute* primário, "+" é igual a mais e "@S" é igual à *attribute* secundário, traduzindo para a forma humana dessa forma: *attribute* do *object* primário mais *attribute* do *object* secundário (Figura 5).

GML Operators

@S + @S	@ / @	@S #- V	@S == @
@ + @	@ / @S	@ #+ @	
@ + @S	@S / @	@S #+ @S	
@S + @	@ / V	@ #+ @S	
@ + V	@S / V	@ #+ V	
@S + V	@S = @S	@S #+ V	
@S - @S	@ = @	@ != @	
@ - @	@ = @S	@S != @S	
@ - @S	@S = @	@ != @S	
@S - @	@ = V	@ != V	
@ - V	@S = V	@S != V	
@S - V	@S ? @S@S	@ == @	
@S * @S	@ ? @@	@S == @S	
@ * @	@ ? VV	@ == @S	
@ * @S	@S ? VV	@ == V	
@S * @	@ #- @	@S == V	
@ * V	@S #- @S	@S #- @	
@S * V	@ #- @S	@S #+ @	
@S / @S	@ #- V	@S != @	

Figura 5 – Exemplo de operadores no padrão GML (*Gameplay Language*).

A cada execução de um *command*, são enviados eventos para os *listeners*, que posteriormente repassarão essas mensagens para as tecnologias externas, criando uma ponte de conexão.

4.1.4 Element

No componente *element* estão representados os elementos que compõem a interatividade do jogo, pois todo jogo é constituído de um ou mais jogadores, inimigos e obstáculos – tudo isso são *elements*. Existem jogos em que um único jogador pode controlar três personagens. Isso pode ser representado, no contexto deste trabalho, pela criação de três *elements*: Personagem A, B e C. A mesma coisa vale para inimigos: existem diferentes tipos de inimigos, podendo ser classificados por sua raça, dificuldade, ou pelo fato de ser um chefão. O mesmo ocorre com obstáculos como parede, chão escorregadio, etc.

Cada *element* pode ser ligado com um componente do tipo *class*. Essa ligação é realizada quando se necessita atribuir variáveis a um *element*.

No caso do conhecido jogo do videogame da Atari denominado Pacman, os *elements* são Fantasma, Parede, Ponto, Vitamina e o próprio Pacman.

4.1.5 Action

Action representa as ações que os *objects* podem realizar. No caso do Pacman, pode ser exemplificado como andar para a esquerda, para a direita, para cima e para baixo, no caso do jogador; para o Fantasma, as suas *actions* são perseguir o Pacman ou fugir dele.

As *actions* podem ser ativadas pelo correspondente *engine* comunicado com a *GameplayLib*. No exemplo, o jogador, por meio de, por exemplo, um teclado ou *mouse*, pode mover o Pacman. Já no caso do Fantasma não é o jogador que manipula, então a entrada de dados é substituída pela inteligência artificial para encontrar o menor caminho até o Pacman para capturá-lo.

4.1.6 Perception

Perception representa as percepções que interagem com um *object*, ou mundo externo. No caso do Pacman pode ser exemplificado como colidir com um fantasma, com um ponto, com uma parede ou com uma vitamina.

Aprofundando o assunto, o *engine* é responsável por enviar essas percepções para a biblioteca por meio de eventos do tipo *trigger*. Nesse momento, a biblioteca pode modificar as variáveis das *classes*, dos *behaviors* e do *state game*. O *engine* escuta esses eventos e os executa, gerando processos e respondendo com *perceptions*. Assim ele continua até o estado de fim de jogo, podendo se tornar uma vitória ou uma derrota.

4.1.7 Gameplay

O *gameplay* gerencia as classes e une todos os elementos. Por meio dele pode-se usar todos os componentes para usufruir as regras do jogo. Existe uma função denominada *updateStep* que serve para atualizar os *behaviors* (comportamentos dos *objects*) e os *generators* (geradores de *objects*).

Este gerenciador *gameplay* foi construído para ter somente uma instância, não possibilitando ao

desenvolvedor criá-lo duas vezes. Em padrões de projeto isto é denominado de *singleton*. Esta decisão foi tomada para simplificar o entendimento dessa biblioteca e diminuir a margem de erros, como criar componentes desvinculados. Isso facilitou no próprio desenvolvimento da biblioteca, diminuindo a quantidade de testes.

4.1.8 Class

Por meio das *class* se criam os *objects*. Elas foram criadas para não perder a referência ao gerar ou inicializar as regras. A *class* é constituída de *element*, *behavior* e *attributes*. Caso seja desejado, o desenvolvedor tem acesso para modificar as atribuições iniciais.

Essas atribuições iniciais referem-se aos *attributes* da *class*. Por exemplo, quando se criam *objects* da *class* carro, pode-se definir que é uma Ferrari ou um Fusca, mas ele sempre nasce Fusca, pois na *class* Carro esta selecionado ao *attribute* "tipo de carro" para Fusca. Pode-se tanto modificar o *attribute* inicial da *class* como do *object*. Isso dá um poder de customização, realizando modificações em cima dos *attributes* e aumentando as possibilidades de criação das regras.

4.1.9 State Game

State Game representa um estado do jogo, sendo que um jogo pode ter n estados (implementando de certa forma uma FSM – *Finite State Machine*). Os jogos mais simples contêm início, meio e fim. Outros jogos mais complexos podem ter n estados de início, meio e fim. Em alguns casos, dependendo do *state game* atual, mudam as regras do jogo. Esta ferramenta também proporciona funcionalidades como gerência de contexto (*state game* atual) e mudança de *state game*.

Um exemplo de aplicação dos *state game* é o caso de jogos de RPG. O agente inicia em um estado de "navegar", em que pode andar por um vilarejo. Dependendo de onde ele caminha, é ativado um modo de combate aleatório, mudando o estado do jogo para "em combate". Caso o agente perca a batalha, o jogo não termina, voltando ao estado de "navegar", em que pode andar pelo mundo até encontrar um ferreiro e mudar de estado para "comprar itens". Sendo assim, a cada mudança de estado mudam as regras do jogo. Quando se está navegando só interessa andar; quando se está em combate só interessa ver os atributos de força, destreza e magia para decidir quem ataca primeiro, a chance de acertar, a quantidade de dano causada no oponente, etc. E quando acaba a batalha, ainda se pode navegar até uma taverna, entrando no estado de comprar itens e escolhendo os objetos que deseja comprar para ganhar vida, força, magia e destreza. Sem contar que ainda podem existir muitos finais diferentes dependendo do caminho que o jogo tomar em seu desenrolar, por exemplo, acabando de uma maneira se matou o oponente ou de outra caso não tenha conseguido matá-lo.

4.1.10Object

Object guarda os *attributes*, o *element* e o *behavior* a ser usado. Por exemplo, com *object* denominado Herói João pode se ligar a vários *attributes*, como pontos de vida, força, magias, entre outros itens do *element* do tipo Herói, ou ainda inserir velocidade, marchas, turbo representados junto a um *element* do tipo carro. Resumindo, esses *attributes* podem significar qualquer coisa, de acordo com a necessidade do desenvolvedor, para a criação de seu jogo.

4.1.11Attribute

Attribute é um componente constituído de um nome e um valor do tipo inteiro, existente dentro de *object* e *class*. Com isso o *object* pode conter uma série de *attributes* que vão interagir com as regras, abrangendo as relações de interatividades de um jogo.

Um exemplo são os jogos de RPG, que necessitam de uma grande quantidade de *atributtes* para a customização do personagem, criando um maior número de possibilidades de interação com as regras de um jogo.

4.1.12Generator

Generator é um gerador de *objects* em que se pode escolher o número de gerações a serem geradas, pois a cada ciclo podem morrer *objects*, sendo criados novamente em um determinado intervalo de tempo. Pode-se escolher, por exemplo, que a cada 5 segundos nasça um número pré-determinado de *objects* ou que uma quantidade indeterminada (aleatória) de *objects* seja gerada. Sendo assim podem-se criar gerações de *objects* no *state game* que inicializa o jogo, ou mesmo durante a execução do jogo, caso se deseje estar sempre criando novos *objects*.

No jogo do Pacman, por exemplo, quando inicializa já existem três fantasmas, nunca mais, isso porque foi estipulado assim pelo criador das regras. Quando morre um ou mais fantasmas a cada ciclo podem nascer de um até três fantasmas.

4.2 Formato GML

A formatação GML (*Gameplay Language*) foi baseada no artigo denominado "Um Modelo de Interações para Definição de Regras de Jogos" [CAMOLESI; MARTIN 2005], sendo escolhido o formato XML, envolvendo componentes com as seguintes especificações: ator, objeto, atividades, espaço e tempo. Estes elementos foram controlados por um livro de regras.

A teoria foi fundamental para dar início à criação das regras do formato GML, tendo como definições *Elements*, *Objects*, *Classes*, *State Game*, *Generators*, *Triggers*, *Behaviors*, *Actions* e *Perceptions*.

Não se optou aqui pela mesma especificação do artigo citado, pois foi encontrada dificuldade na criação de uma aplicação digital; existem muitas relações entre os nodos, não proporcionando uma arquitetura adequada para uma biblioteca.

Com o objetivo de criar regras genéricas, o formato GML teve de ser testado com diferentes tipos de jogos, compostos por diferentes regras, gêneros e tecnologias. Para isso foram analisados quatro jogos de tipos diferentes – Super Mario World, Pacman, Chrono Trigger e Need For Speed –, a fim de encontrar um padrão na formatação de regras que pudesse comportar tanto um jogo simples como um complexo.

Depois de analisados os jogos, descobriu-se que existe um padrão entre todos eles, possibilitando a criação de regras genéricas. Foi constatado que essas regras podem suportar qualquer tipo de jogo, mas se devem seguir algumas recomendações para criar as regras no item "Como desenvolver um *gameplay*".

4.2.1 Projeto

Foi criada uma biblioteca para importação e exportação em XML no modelo GML, para interpretar a estrutura do arquivo e inserir na biblioteca *GameplayLib*. Exemplos de arquivos com a descrição XML adotada no *GamePlayXML* podem ser encontrados no *site* da internet desenvolvido para este projeto (http://tinyurl.com/4uvbam) e nos anexos deste trabalho.

O modelo GML é representado por nodos, propriedades e valores. Existem nodos organizados por *elements*, *actions*, *perceptions*, *state game*, *initializes*, *triggers*, *comands*, *behaviors*, *generators*, *objects* e *classes*. Propriedades específicas para cada nodo permitem criar, modificar e destruir estados conforme as regras do jogo. Cada propriedade contém valores padrões, usados em operadores e identificadores dos componentes (*objects*, *elements*, etc.), e os valores inseridos pelo desenvolvedor do jogo para criar as regras.

Cada evento (*trigger* ou *behavior*) necessita cumprir alguns requisitos para poder ser executada. Como se pode ver na, em todos os eventos existem propriedades (como *idStateGame*, *idPerception*, *idElement* e *idAction*). Antes de ativar as regras, a biblioteca verifica se o estado de jogo é o indicado em *idStateGame*, se o *object* primário ativado pelo evento é o mesmo que o presente em *idElement*, se a *perception* ativada confere com *idPerception* e se a *action* ativada é a mesma que a indicada em *idAction*. Caso se deseje ativar a *trigger* para qualquer valor de uma determinada propriedade, usa-se o número "-1". Para que a *trigger* nunca seja ativada, usa-se "-2". Um exemplo da opção "-2" é criar uma *trigger* somente para o tratamento de *actions*; nesse caso o *perception* fica com a opção "-2".

Um fator que diferencia os *behaviors* e os *triggers* é que os *behaviors* são sempre executados a cada ciclo de jogo e têm uma restrição a mais, que é o tempo estipulado pelo criador das regras. Outro diferencial é que eles estão ligados aos *objects*. Cada *object* contém uma possível ligação para um *behavior*; sendo assim, cada *object* pode ter um *behavior* diferente.

4.3 Como desenvolver um *gameplay*

Ao longo do trabalho, passou-se por diversas experiências no desenvolvimento das regras, até conseguir deixá-las totalmente genéricas. Por esse motivo devem ser observados alguns tópicos, como simplicidade, personalização, g*ameplay* e tecnologia externa.

4.3.1 Simplicidade

Manter a simplicidade é saber entender as regras na sua essência, precisando entendê-las de uma visão abstrata, esquecendo os algoritmos que fazem o jogo funcionar. A pessoa responsável pelo desenvolvimento do *gameplay* não deve detalhar o tipo de algoritmo de colisão ou de renderização, mas de como os elementos devem interagir entre si resultando em uma vitória, derrota ou empate.
Um exemplo é o jogo do Pacman. O criador das regras deve se preocupar em criar *elements* de interatividade, como Pacman, Fantasma, Ponto, Vitamina e Parede; *actions* que o jogador pode realizar, como movimento para cima, para baixo, para a equerda e para a direita; *perceptions* que o jogo responde referentes ao *element* e suas *actions*, como o Pacman pode colidir com a parede, com o fantasma ou com o ponto, e como irá afetá-lo trocando de *state game* como vitória ou derrota.

4.3.2 Personalização

À medida que se desenvolvem as regras, pode-se perceber que alguns elementos ficariam mais bem incorporados dentro do *engine*, seja porque facilita na programação ou por questões de performance. Quando for necessária essa decisão, deve-se perguntar se fará diferença na hora de personalizar depois de terminado o jogo, pois em caso afirmativo haverá dificuldade de personalização, tendo que abrir o código-fonte e compilar e entrar em ciclo de *bugs*, perdendo tempo de uma mão-de-obra de alto custo.

4.3.3 *Gameplay* e tecnologia externa

As regras devem ser criadas separadamente das tecnologias, pois do contrário haverá dependência entre ambos, impedindo migração para outras tecnologias.

É um erro comum incorporar as regras com a posição do jogador com duas dimensões x e y. Nesse momento está havendo ligação da tecnologia com as regras, e isso acarreta a dependência de uma biblioteca de duas dimensões. No momento em que se trocar para uma que tiver três dimensões terá de haver mudança nas regras.

Em vez de dizer "o botão A", diga "acelera o carro"; em vez de dizer "X + 1". diga "para frente". Isso facilitará o entendimento dos programadores para a implementação do jogo.

4.4 Resultados

O principal resultado que permite validar e avaliar este trabalho é o desenvolvimento de dois jogos. Cada jogo teve suas regras analisadas, pensadas e criadas no formato GML. Posteriormente, as regras foram ligadas à biblioteca com o *engine*, para o funcionamento do jogo.

Os dois jogos desenvolvidos foram uma versão do Pacman (rodando em console) e um jogo estilo RPG (que possui duas versões, uma com gráficos em duas dimensões e outra em console).

4.4.1 Jogo estilo RPG

O jogo de RPG foi desenvolvido com base nos jogos de Super Nintendo, como Chrono Trigger. Este jogo de referência foi importante para o entendimento da mecânica de jogo, pois em um RPG, dependendo do estado de jogo, as regras mudam.

Em uma visão geral a mecânica de jogo funciona da seguinte forma: no momento inicial, o personagem pode andar pelo mundo, mas quando chega perto do inimigo, entra-se em estado de combate, no qual as regras do jogo são modificadas. Durante o combate o personagem não pode mais andar pelo mundo até terminar a batalha. Se o jogo estiver no estado de navegação, é possível ainda entrar em uma loja, que faz o jogo entrar em modo de compra, permitindo comprar itens que vão ajudar no combate.

Os eventos permitem a realização de atribuições randômicas em cima de *attributes*, ajudando para a criação das regras referentes à chance de atingir o adversário durante o combate, pois nos jogos de RPG existe o fator sorte: existe a chance de um herói mais fraco vencer um inimigo mais forte e vice-versa, tornando o jogo mais divertido.
Isto é feito em cima dos *attributes* do *object* primário e secundário em conjunto com uma randomização nesses valores.

Para a execução de cada evento, um *object* primário deve ser referenciado para a execução das regras e, caso se deseje, podem ser referenciados *objects* secundários. No caso do evento "atacar_forca", necessita-se do *object* primário e secundário para a realização de um ataque, pois nas interações destas usam-se *attributes* dos *objects* como força, vida e chance de acertar.

Um ataque pode ser usado de um *object* primário para um *object* secundário ou de um *object* primário para muitos *objects* secundários, permitindo a realização de ataques simultâneos. Isso pode ocorrer de um herói para inimigo, ou vice-versa, tendo em vista que, para realizar este feito, necessita-se de uma organização das ordens dos atributos. Se os atributos desejados para a interação forem os mesmos e na mesma ordem, independentemente dos *objects*, o evento executará as regras sem problema algum.

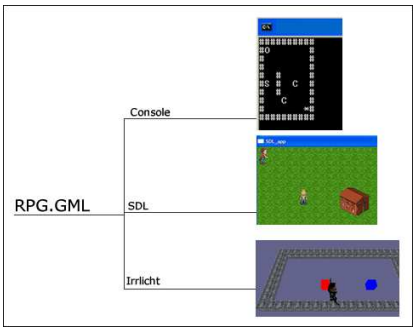


Figura 6 – Jogo de RPG de console em estado de navegação; um fator importante é que os três *engines* estão usando o mesmo arquivo GML.

5. Considerações finais

A principal motivação deste trabalho era criar um conjunto de ferramentas para o auxílio na criação de regras de jogos digitais de uma forma genérica, focada para as pessoas que não têm um profundo conhecimento na área tecnológica. Para isso, foi necessária a criação de uma estrutura usando o formato XML, proporcionando a separação do *gameplay* e do *engine* por meio de uma biblioteca que gerencia as regras. Esta separação ajudou na prototipação de jogos podendo usar motores mais simples para testes de jogabilidade.

Visando à criação do editor, foram estudadas três ferramentas no mercado de jogos: Game Maker, Dark Basic e RPG Maker. Concluiu-se que, para obter uma não-dependência das tecnologias, era preciso se desligar dos recursos de um jogo, como imagens, sons e modelos 2D ou 3D, focando o pensamento somente nas regras.

No desenvolvimento do *engine*, o desenvolvedor ganhou benefícios utilizando a ferramenta como: automatização das regras, separação das tecnologias. Resultando em um código robusto e organizado.

Depois de desenvolvido o editor e de exportadas as regras para o *engine*, era necessário criar os resultados, que no caso são os jogos para verificar a mecânica de jogo. Foram desenvolvidos dois jogos: Pacman, usando um *engine* de console, e um jogo estilo RPG com dois *engines* (um em 2D e outro mais simples para prototipação em console). É importante destacar que o arquivo em XML com as regras do jogo estilo RPG era o mesmo em ambos *engines*. Isso demonstra que foi atingido o objetivo de independência de tecnologia.

Todo o processo de desenvolvimento do trabalho está disponível na internet, no *blog* http://sdkgameplay.blogspot.com. No mesmo endereço também se pode realizar o *download* da ferramenta sob licença LGPL.

Este trabalho tem potencial não só de ajudar as pessoas que tenham menos conhecimento técnico, mas também de integrá-las com as pessoas que dominam a tecnologia. Isso permite que essas pessoas possam realizar parcerias pra a construção de jogos, criando um elo entre os conhecimentos. Isso antes não era possível porque as ferramentas estudadas não permitiam que pessoas com mais conhecimento tecnológico tivessem a oportunidade de usar, modificar, alterar ou adicionar as tecnologias existentes.

Este trabalho vai ajudar a comunidade de jogos, pois, parametrizando as regras de uma forma genérica, possibilita-se o compartilhamento no mundo digital usando um formato popular denominado de XML. Une-se, assim, profissionais técnicos e desenvolvedores de *gameplay*.

O próximo projeto seria explorar o *gameplay* utilizando as tecnologias construídas para a realização de experiências científicas referentes aos tipos de mecânicas de jogos existentes, resultando na diversão, para futuramente o programa poder ajudar na construção desses jogos.

Referências Bibliográficas

CAMOLESI JR., Luiz; MARTIN, Luiz Eduardo Galvão. *Um modelo de Interações para Definição de Regras de Jogo*. São Paulo: SBGames, 2005. Sociedade Brasileira de Computação.

Click Team - The Games Factory, 2007. Disponível em: <http://www.clickteam.com/eng/>. Acesso em: 9 ago. 2007.

HARBOUR, Jonathan S.; SMITH, Joshua R. *Beginner's Guide To Dark Basic Game* Programming. Portland: Premier Press, 2003.

HAROLD, Elliotte Rusty. *XML Bible*. IDG Books Worldwide, 1999.

MARTIN, James. *Rapid Application Development*. Indiana: Macmillan Coll Div Publisher, 1991.

MOTOR DE JOGO (Definição) - Wikipedia. Disponível em: <http://pt.wikipedia.org/wiki/Motor_de_jogo>. Acesso em: 20 maio 2008.

SDK (Definição) - Wikipedia. Disponível em: <http://pt.wikipedia.org/wiki/SDK>. Acesso em: 20 maio 2008.

YoYo Games – Gamemaker. Disponível em: <http://www.yoyogames.com/gamemaker/>.

Parallel Culling and Sorting based on Adaptive Static Balancing

Lucas Machado Bruno Feijó
VisionLab/ICAD/IGames, Dept. of Informatics, PUC-Rio
{lmachado, bfeijo}@inf.puc-rio.br

Abstract

This paper presents a new and effective method for parallel octree culling and sorting for multicore systems using counting sort and based on a new balancing algorithm, called adaptive delayed static balancing. The adaptive nature of the method is governed by a dynamic split level that can adjust the algorithm to new camera positions keeping a well-balanced workload amongst the processors. Also this paper introduces the concept of n-dimensional resource space as a discrete Euclidean space. This work presents a simple and effective thread management system called MinTMS.

1. Introduction

Octree culling is a classical algorithm for reducing the amount of data sent to the GPU for rendering. The technique consists of dividing the 3D space into eight cubes and repeating the process for each cube until a certain level of the octree is reached (usually, the leaves) and objects are stored. Rendering is done by testing the intersection of the view frustum with the octree nodes and sending to the GPU only the visible objects. In this case, if a certain node cannot be seen its entire subtree is pruned from the octree. This process can be easily parallelized, but the balance of the workload is not trivial. Another aspect of the rendering process is resource sorting (e.g. Textures, meshes, and pixel shading techniques). There is always a cost associated to resource changes. Therefore, these changes should be reduced by sorting and grouping objects with common resources. Most of the methods for parallel rendering is concentrated on PC clusters and grids, while the literature on parallel culling for multicore systems is scarce. This paper presents a new and effective method for parallel octree culling and sorting for multicore systems using counting sort (which is O(n) time) and based on a new balancing algorithm called adaptive delayed static balancing. Tests revealed a performance improvement of the culling process between 3 and 4 times in relation to the classical single threaded octree culling process. However, the most important performance analysis concerns the capability that the proposed method has to adapt itself to new camera positions, which are continuously changing over time.

This paper is organized as follows. In the next section, previous works are analyzed. In section 3, we present the concept of the adaptive delayed static balancing for parallel culling. The algorithm for node rendering is presented in section 4. Section 5 presents the adaptive nature of the proposed method. In section 6, we have the entire algorithm. Also this paper proposes a simple API for thread management called MinTMS in section 7. Section 8 presents the parallel sorting method that handles the sorting of multiple resources of the objects. Finally, some results are described in section 9 and section 10 presents some final remarks.

2. Related Work

A lot of research on parallel search and sorting algorithms has already been done and many techniques exist in the literature [Grama et al. 2003] [Wilkinson and Allen]. Also several works have been carried out in the area of parallel rendering using sorting techniques. Molnar et al. [1994] proposed a classification of parallel rendering system based on in which stage of the rendering pipeline the sorting is carried out (sort-last, sort-middle, and sort-first). Humphreys et al. [2002] present a sort-first method for distributed rendering using a cluster of common PCs. Abraham et al. [2004] propose a load-balancing strategy for sort-first distributed rendering using PC-based clusters. Baxter et al. [2002] present a parallel rendering architecture using two graphics pipeline and one processor, including occlusion culling, LOD and scene graph hierarchy. However, these works concentrate on distributed rendering using PC clusters

and/or on global aspects of parallel rendering. The literature has few works concentrated on algorithms for parallel culling and sorting using multicore systems.

Octree is a classic data structure used in many computer graphics applications [Foley et al. 1995], [Dalmau 2003]. However, parallel occlusion algorithms using octrees are not usual. Greene et al. [1993] are the first authors to propose an octree hierarchy for visibility computation with some potential to parallelize. Their work had a great influence on graphics hardware design. Xiong et al. [2006] present an algorithm for parallel occlusion culling on GPUs clusters using the occlusion query function provided by current GPUs. As far as the authors of the present papers are aware, there is no previous work on parallel octree occlusion and sorting for multicore systems based on simple and efficient static balancing and $O(n)$ time sorting algorithm.

3. Initial Concepts

One of the main problems in parallel culling using octrees is how to balance the workload amongst the processors. The simple strategy of equally distributing the top level nodes between processors (called static balancing) may result in long idle times in some processors at certain camera positions. An alternative solution to the problems of static balancing is the use of a dynamic balancing strategy, where a processor asks another one for working when it becomes idle. The drawback of this solution is the addition of increasing communication overheads. In this paper, we propose a new and effective strategy called "adaptive delayed static balancing" that has the following characteristics:

1. Instead of distributing the nodes equally amongst the processors at the start of the processing, the algorithm waits until a certain level d (called "split level") in the octree is reached and only then it distributes the work as a static balancing procedure. This characterizes a "delayed" static balancing strategy.
2. Irrelevant nodes are pruned from the tree before the work is distributed amongst the processors.
3. The split level d is dynamically adapted to changes in the virtual environment. This characterizes an "adaptive" strategy.

The reason for the implementation of the above-mentioned delay is that the frustum usually interacts with the nodes in lower levels of the octree. In such lower levels there is a better chance for a more balanced distribution of work. In the proposed algorithm, before the split level d is reached, a sequence of nodes is visited in a breath-first way and a list of nodes (node_list) is prepared for the distribution stage of the algorithm. Irrelevant nodes (*i.e.* branches of the tree with no intersection with the frustum) are automatically pruned from node_list. The nodes from node_list are distributed amongst the processors by creating a list of nodes for each processor and storing it in a vector called working_list. The implementation of this strategy requires the following main tasks:

- To visit the nodes until the split-level is reached
- To set up the list of nodes to be distributed (pruning the octree adequately): node_list
- To expand nodes in node_list
- To render the leaf nodes of the octree that are intersected by the frustum

4. Rendering Nodes

The tasks presented in the previous section can be accomplished by the function RenderNodes(idx, n, node_list, **frustum**). In this paper, "to render nodes" from an octree means to add the objects from a leaf node to a data structure that should be processed by the processor idx considering resource optimizations and GPU communications. The function RenderNodes can transverse the octree completely or stop after n nodes (n = 0 means no limit to transverse the tree). In the case of having a limit (n > 0), this function returns a non-empty node_list containing the nodes to be distributed amongst the processors (including the main processor that is currently setting node_list up). Figure 1 presents the pseudo-code of the function RenderNodes, where frustum is a structure containing the coordinates and orientation of the frustum (which are constantly moving at each frame in time).

```
RenderNodes(idx, n, node_list, frustum)
  set node_count to 0
  while node_list is not empty
      node = first node of node_list
      eliminate first node of node_list
      if frustum intersects node
          if node is a leaf
              add all objects of node to idx data
          else
              put children of node at the end of node_list
      if n > 0          // i.e.: render must stop after n nodes
          increment node_count by 1
          if node_count is equal to n
              break the loop of while
  return node_list
```

Figure 1 The function to render nodes

The function RenderNodes can be executed by the main processor (*e.g.* P1) or one of the secondary processors (*e.g.* P2, P3, or P4). In the case of secondary

processors, RenderNodes is executed by another function that is controlled by a thread management system. This later function is RenderNodesProcessorTask(idx, working_list[idx]), where idx is the processor index and working_list is the list of nodes to be processed, as shown in Figure 2. The global vectors startTime[idx] and endTime[idx] are used to calculate the idle time of the processors. The task RenderNodesProcessorTask is controlled by using a new and simple API for thread management proposed in the present paper.

```
RenderNodesProcessorTask (idx,working_list[idx])
  get current time and save it as startTime[idx]
  RenderNodes(idx,0,working_list[idx],frustum
  Get current time and save it as endTime[idx]
```

Figure 2 Calling RenderNodes for processor idx

5. Dynamic Adaptation of the Split Level

Before presenting the complete algorithm proposed in this paper, we should consider the dynamic adaptation of the split level. The best split level (d=0, d=1, d=2, …) is the one that minimizes the sum of the idle time of all processors. Our algorithm employs an adaptive strategy that constantly changes the split level. This strategy is based on the fact that deeper levels tend to reduce the total idle time. Therefore, we expect that each new time frame should increment the split level d. However, depending on the movement of the camera through the virtual environment, the tendency for decaying idle time is broken and decrements in the value of the split level should be tried until the normal trend is recovered (*i.e.* the increase of d causes the decay of total idle time). This is a process that searches

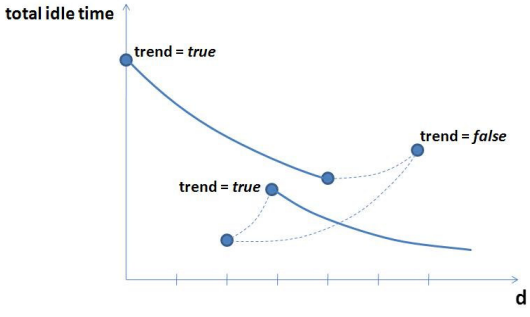


Figure 3 Trends (decaying idle time with increasing d) and adjustment periods

for the optimum value of d. There is no way of deducing a function relating d and total idle time. Experiments have suggested us that trends (solid lines in Figure 3) can be eventually disturbed by adjustment periods (dashed lines in Figure 3). This behavior inspires us to propose the function split_level to dynamically adapt d to changes in the virtual environment based on trends, as shown in Figure 4.

```
Last sum of the idle time of all processors as a global
variable: last_total_idle_time = 0
Current trend as a global variable: trend = false
Number of levels of the octree as a global variable (already
calculated): num_levels

split_Level(d, total_idle_time)
  if total_idle_time is greater than last_total_idle_time
       reverse trend       // e.g. trend = not(trend)
  if trend is true
       if  d  is less than (num_levels – 1)
            increment d by 1
  else
       if  d  is greater than zero
            decrement d by 1
  last_total_idle_time = total_idle_time
  return d
```

Figure 4 Function split_Level to dynamically adapt d to changes in the virtual environment

6. The Proposed Algorithm

Considering the explanation presented so far, the adaptive delayed static balancing algorithm can be described by the pseudo-code of the function ADStBalancingRender(d, octree, frustum) in Appendix A, where d is the split level (initially zero), octree is a structure containing the octree of the scene, and frustum is a structure containing the coordinates and orientation of the frustum (which are constantly moving at each frame in time). The function ADStBalancingRender is called by the main program at each time frame. This function calls CalculateTotalIdleTime() that is presented in Figure 5.

7. MinTMS

The proposed algorithm considers that the threads are initialized by the main program. This initialization procedure together with three other procedures (used in ADStBalancingRender, Appendix A) are proposed as a simple API for thread management that hides the difficulties of using low level system functions. This API, called MinTMS (for Minimum Thread Management System) (see Appendix A), is described as follows:

Init(n)
  This method creates n threads that remain blocked until StartWorking() is called.
SetTask(idx,task,data)

This method sets a task and the data to be processed by the thread idx. The task is executed when StartWorking is called.

StartWorking(idx)

This method unblocks the thread idx. The thread idx returns to a blocked state after processing its task.

WaitUntilWorkFinished()

This method implements a barrier and blocks the main processor until all threads have finished their tasks.

The starting processing time of each processor as a global vector: **startTime[]**
The ending processing time of each processor as a global vector: **endTime[]**
Total number of processors as a global variable: **num_processors**

```
CalculateTotalIdleTime()
    min = stratTime[0]
    max = endTime[0]
    for i = 0 to (num_processors -1) stepping by 1
        if startTime is less than min
            min = startTime[i]
        if endTime[i] is greater than max
            max = endTime[i]
    idle = 0
    for i = 0 to (num_processors -1) stepping by 1
        increment idle by (startTime[i] – min)
        increment idle by (max – endTime[i])
    return idle
```

Figure 5 Function to calculate the total idle time

8. The Proposed Counting Sort Method

8.1 Sorting Algorithms

The main feature of a sorting algorithm [Cormen et al. 2001] is the amount of time required to reorder *n* given numbers into increasing order. However, there are other features to be considered. A sorting algorithm is called *in-place* if it uses no additional array storage (buffer) and is called *stable* if duplicate elements remain in the same relative position after sorting. Mergesort is a stable $O(n \log n)$ sorting algorithm but it is not in-place. Heapsort is an in-place $O(n \log n)$ sorting algorithm, but it is not stable. Quicksort is regarded as one of the fastest sorting algorithm, but it is not stable and, stickling speaking, it not in-place.

It is a well-known theorem that is not possible to sort faster than $O(n \log n)$ time for algorithms based on 2-way comparisons. Sorting numbers faster than this lower bound must be done without the use of comparisons, what is only possible under certain very restrictive circumstances. Under these special conditions, an entire class of linear time sorting algorithm arises. For instance, counting sort is a stable O(n) sorting algorithm, but not in-place, which can only be used in applications that sort small integers. In

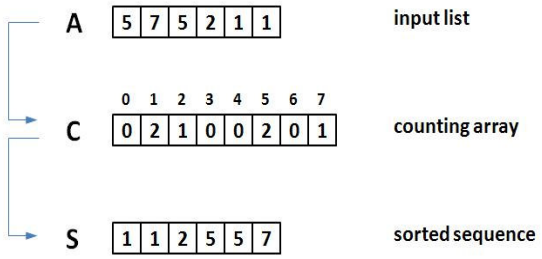


Figure 6 Example of counting sort

this algorithm, for each integer k found in the input list A, we increment the value of C[k] by 1 (the size of C is determined by the largest integer in A), as shown in Figure 6. C[k] is called *counting array*. In the next section, counting sort is presented as the best algorithm for resource sorting in parallel rendering.

8.2 Resource Sorting

Resources are data, properties, components, techniques, and programs used by the 3D objects in order to be rendered properly. Textures, meshes, and pixel shading techniques are common resources used in the rendering processes of real-time applications. Each type of resource defines a discrete axis (*i.e.* an axis with integer coordinate values) called *dimension* (*e.g.* textures are identified by the integer values 0, 1, 2, … in the texture axis). *Resource space* is a discrete Euclidean space defined by one or more dimensions. Therefore, the *texture* dimension and the *mesh* dimension form a two-dimensional resource space. An efficient rendering strategy is the one that groups objects sharing the same resources (*i.e.* it groups the objects in the same point of the resource space). This strategy minimizes the costs associated with every resource change during the rendering process (there is always a great cost associated to jumps within the resource space). In this paper, for each point (i,j,k,…) of the resource space, we define the *n-dimensional resource data array* R[i,j,k,…] containing the following data:

- The number ***c*** of objects sharing the same set of resources i,j,k, …;
- A list ***L*** of these objects.

We use the following notation to present this n-dimensional array:

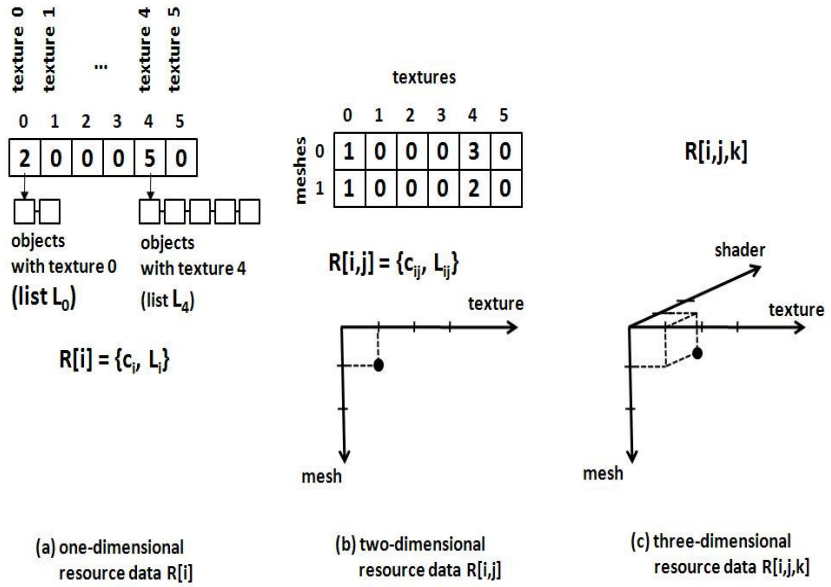


Figure 7 Simple cases of the n-dimensional resource data array R, where *c* is the number of objects and *L* is the list of the objects. The discrete resource spaces are also illustrated.

$$R[i, j, k, ...] = \{c_{i,j,k,...}, L_{i,j,k,...}\} \qquad \text{(Eq.01)}$$

Figure 7 illustrates the simplest cases for R[i,j,k,…]: one, two, and three-dimensional resource data arrays. In Figure 7(b) the two dimensions are texture and mesh. In this 2-dimension example, the rendering process can fix a mesh and render objects per texture (*e.g.* it fixes mesh 0 and renders 1 object with texture 0 and then 3 objects with texture 4).

In the case of one dimension represented by textures (Figure 7(a)), we can easily identify R[i] as being an extended version of the *counting array* C[k] in the counting sort algorithm (Figure 6). The main job of the function RenderNodes (Figure1) is to add objects to the resource data array R of each processor. Therefore, this job is a counting sort process. As resources can be represented by small integer numbers (complex 3D scenes hardly go beyond 300 different textures), the most appropriate sort algorithm for parallel rendering is counting sort. In this way, we have the fastest and convenient option: a *stable O*(*n*) sorting algorithm. We should notice that the *in-place* nature of counting sort (presented in the previous section) is not relevant in the present application, because we need a storage array to distribute work amongst the processors anyway.

8.3 The Sorting Process

The function RenderNodes (Figure1) builds the resource data array R of each processor Pi, in such a way that the objects are distributed amongst the processors and grouped according to the resources they use. In this paper, the proposed algorithm merges the

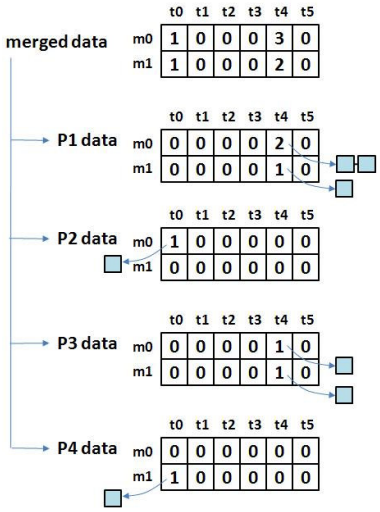


Figure 8 The merged resource data array M

arrays R into a single n-dimensional array M, called *merged resource data array*, by performing the sum of the corresponding $c_{i,j,k,...}$ and transferring the references to the lists $L_{i,j,k,...}$. Figure 8 illustrates the entire merging process for the two dimensional case and four

processors. We should notice that Pi data is not inside each processor (in fact the sets of Pi data are in a common structure that each processor can freely access).

Once the merged data array M is completed we can scan it and whenever *c* is greater than zero the list of sorted objects *L* can be rendered using the resources identified by the integer coordinates (i,j,k,…).

9. Some Results

The computer used for tests is a quadcore machine (Intel Core 2 Extreme Q6850 3.00GHz). The GPU rendering performance should be isolated from the performance analysis of the proposed parallel culling method. Therefore, no fps figures are presented.

The first test compares the proposed method with a classical single thread octree culling for an octree with 8 levels (2.396.745 nodes). The result in Table 1 shows an improvement of 3.16.

Table 1 The proposed method (4 threads) *vs* standard Single Thread for an 8-level octree

ADStBalancing (4 Threads)	Single Thread
milliseconds	milliseconds
19	60

The second type of test analyses the adaptive nature of the proposed method by investigating its performance at several values of the split level (d) and the number of nodes processed by each processor. The tests use a camera with FOVy = 30 degrees with a 9-

Table 2 The proposed method with the camera at the centre of a 9-level octree and FOV=30 (intersecting all main nodes right below the root). GPU time is not included.

Split Level	number of nodes				Average Culling Time
d	P1	P2	P3	P4	microseconds
0	2019	482	482	2018	360
1	2025	480	480	2016	362
2	2073	464	464	2000	365
3	897	1880	1880	344	202
4	4761	80	80	80	304

Table 3 The proposed method with the camera at the corner of a 9-level octree and FOV=30 (intersecting only one main nodes right below the root). GPU time is not included.

Split Level	number of nodes				Average Culling Time
d	P1	P2	P3	P4	microseconds
0	4435	2	2	2	288
1	4435	2	2	2	288
2	1945	1888	304	304	214
3	1033	2512	448	448	260
4	4441	0	0	0	292

(a) Camera at center (b) Camera at corner

Figure 9 Two different camera positions used in the tests of Table 2 and Table 3

level octree, 4 processors, and 5 values of split levels.

Tables 2 and 3 show that the split level scheme adapts the algorithm for different camera positions. In Table 3, d = 0 is a bad start for both time (a culling time higher than the one for the camera at the center) and workload balancing (number of nodes). In both cases the system stabilizes around d=3 for case 1 (Table 2) and d = 2 in case 2 (Table 3). Figure 9 shows the final rendering for each camera position.

10. Final Remarks

This paper presents a new and effective method for parallel octree culling and sorting for multicore systems using counting sort (which is $O(n)$ time) and based on a new balancing algorithm called *adaptive delayed static balancing*. Tests revealed a time performance improvement of the culling process between 3 and 4 times in relation to the classical single threaded octree culling process. However, the most important result is the effectiveness of the adaptive mechanism based on the dynamic split level that can adjust the algorithm to new camera positions and keep a well-balanced workload amongst the processors. The tests do not consider GPU time and also avoid any connection with the number of resources (*i.e.* number of textures, meshes, …).

Also this paper introduces the concept of *n-dimensional resource space* as a discrete Euclidean space, in which the resource array is identified with the counting array of the counting sort algorithm. No other sorting algorithm can be faster than this $O(n)$ time algorithm for the culling process. The proposed *adaptive delayed static balancing* method naturally generates points in the n-dimensional resource space in a counting sort way.

Another important result is the proposed thread management system MinTMS, which reveals itself as a simple and effective API.

Future works should cover extensive statistics and comparisons, including plots of time *vs* number of nodes, time *vs* number of resources, total idle time *vs* split level, more complex scenes, and more points in the camera path. The comparison with related work is difficult because the literature is scarce on parallel octree culling for multicore machines and we have no access to the code of other authors to reproduce the same test situation. Another future work should be the investigation of other heuristics and statistics that can improve the adaptive performance of the method. Further work should also consider a parallel merging process (*i.e.* to mount the array M in parallel, Figure 8).

Acknowledgements

We would like to thank CNPq and FINEP for the financial support of scholarships and research projects.

References

Cormen, T. H. Leiserson and Charles, E. R. and Ronald, L., 2001. *Introduction to Algorithms Second Edition*. Massachusetts: The MIT Press.

Dalmau, D. S.-C., 2003. *Core Techniques and Algorithms in Game Programming*. Indiana: New Riders.

Foley, J. D. V. D. and Andries, F. and Steven, K., 1995. *Computer Graphics: Principles and Practice in C*. New York: Addison Wesley.

Grama, A. and Gupta, A. and Karypis, G. and Kumar, V., 2003. *Introduction to Parallel Computing*, New York: Addison Wesley.

Wilkinson, B. and Allen, M., 2004. *Parallel Programming Techniques and Applications using Networked Workstations and Parallel Computers*, New Jersey: Prentice Hall.

Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H., 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics & Applications*, 14(4), 1994, pp. 23-32.

Humphreys, G., Houston, M., Ng, R., Frank, R. Ahern, S., Kirchner, P.D., and Klosowski, J.T., 2002. Chromium: a stream-processing framework for interactive rendering on clusters. *Proceedings of ACM SIGGRAPH 2002, acm Transactions on Graphics*, 21(3), 2002, pp. 693-702.

Abraham, F., Celes, W., Cerqueira, R., and Campos, J.L., 2004. A load-balancing strategy for sort-first distributed rendering. *XVII Brazilian Symposium on Computer Graphics and Image Processing, Proceedings SIBGRAPI 2004*, 17-20 Oct 2004, Curitiba, PR, Brazil, IEEE Computer Society, 2004, pp. 292-299.

Baxter, W.V, Sud, A., Govindaraju, N.K., and Manocha D., 2002. Gigawalk: Interactive walkthrough of complex environments. *Proceedings of 13th Eurographics workshop on Rendering*, 2002, pp. 203-214.

Greene, N., Kass, M., and Miller, G., 1993. Hierarchical Z-buffer visibility. *Proceedings of ACM SIGGRAPH 1993, acm Transactions on Graphics*, 1993, pp. 231-238.

Xiong, H., Peng, H., Qin, a., and Shi, J., 2006. Parallel occlusion culling on GPUs cluster. *Proceedings of 2006 ACM International Conference on Virtual Reality Continuum and its Applications* (VRCIA 2006), Hong Kong, China, 14-17 June 2006, pp. 19-26.

APPENDIX A – Algorithm and MinTMS

The index of the main processor as a global constant: **MAIN_PROCESSOR_IDX** = 0
Last sum of the idle time of all processors as a global variable: **last_total_idle_time** = 0
Current trend as a global variable: **trend** = *false*
Number of levels of the octree as a global variable: **num_levels**
Total number of processors and processor id vector as global variables: **num_processors** and **p_idx[ ]**

ADStBalancingRender (d, **octree**, **frustum**)
get current time and save it as startTime[MAIN_PROCESSOR_IDX]
calculate[1] the number of nodes up to the current split level d: n = $(8^{d+1} - 1)/7$
node_count = 0
clear node_list
put the root node of **octree** in node_list
node_list = RenderNodes(MAIN_PROCESSOR_IDX, n, node_list, **frustum**) // n is greater than zero
set work_size to the size of node_list divided by num_processors
for i = 0 to (num_processors-2) stepping by 1 // i is the secondary processor executing the rendering task
transfer work_size nodes from node_list to working_list[i]
eliminate the transferred nodes from node_list
set the task RenderNodesProcessorTask(p_idx[i],working_list[i]) // done by the MinTMS method: SetTask
make processor i to start the task RenderNodesProcessorTask // this is done by calling StartWorking(i)[1]
RenderNodes(MAIN_PROCESSOR_IDX,0,node_list,frustum) // remaining nodes in the main processor
get current time and save it as startTime[MAIN_PROCESSOR_IDX]
If num_processors is greater than 1
wait for the other processors finish working // this is done by waitUntilWorkFinished()[1]
total_idle_time = CalculateTotalIdleTime()
d = split_Level(d, total_idle_time)
return d

```
#include "ThreadManager.h"
#include <process.h>
ThreadManager* ThreadManager::instance;
ThreadManager::ThreadManager() {
	instance = NULL;
}
ThreadManager* ThreadManager::GetInstance() {
	if(instance == NULL)
	   instance = new ThreadManager;
	return instance;
}
void ThreadManager::Init(int thread_count) {
	num_threads = thread_count;
	task_list = new ThreadTask[num_threads];
	data_list = new void*[num_threads];
	start_work = new HANDLE[num_threads];
	work_finished = new HANDLE[num_threads];
	for(int i = 0 ; i < num_threads ; i++)
	{
	   start_work[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
	   work_finished[i] = CreateEvent(NULL,TRUE,FALSE,NULL);
	   int* id = new int;
	   *id = i;
	  _beginthread(ThreadManager::ProcessorThread,1024,id);
	}
}
void ThreadManager::ProcessorThread(void *data) {
	int processor_idx = *((int*) data);
	ThreadManager* manager = ThreadManager::GetInstance();
	while(true) {
	   WaitForSingleObject(manager->start_work[processor_idx],
INFINITE);
	   ResetEvent(manager->start_work[processor_idx]);
	   manager->task_list[processor_idx](manager-
>data_list[processor_idx]);
	   SetEvent(manager->work_finished[processor_idx]);
	}
	delete data;
}
```

```
void ThreadManager::SetTask(int thread_idx, ThreadTask task, void
*data) {
	task_list[thread_idx] = task;
	data_list[thread_idx] = data;
}
void ThreadManager::StartWorking(int thread_idx) {
	SetEvent(start_work[thread_idx]);
}
void ThreadManager::WaitUntilWorkFinished() {
	WaitForMultipleObjects(num_threads, work_finished, TRUE,
INFINITE);
	for(int i = 0 ; i < num_threads ; i++)
	   ResetEvent(work_finished[i]);
}
```

The ThreadManager.h file is presented below:

```
#ifndef THREAD_MANAGER_H
#define THREAD_MANAGER_H
#include <windows.h>
typedef void (*ThreadTask)(void*);
class ThreadManager {
public:
	static ThreadManager* GetInstance();
	void Init(int num_threads);
	void SetTask(int thread_idx, ThreadTask task, void* data);
	void StartWorking(int thread_idx);
	void WaitUntilWorkFinished();
private:
	ThreadManager();
	static ThreadManager* instance;
	int num_threads;
	ThreadTask* task_list;
	void** data_list;
	HANDLE* start_work;
	HANDLE* work_finished;
	static void ProcessorThread(void* data);
};
#endif
```

Proposta de uma heurística para o jogo de dominó de 4 pontas

Nirvana da S. Antônio* Cicero F.F. Costa Filho Marly G. F. Costa

Universidade Federal do Amazonas, Centro de Tecnologia Eletrônica e da Informação, Brasil

Abstract

This paper presents a new methodology for the choice of the best move in the four extremity domino game. The proposed methodology can be divided in two parts. First of all is defined the game state. This definition comprises the use of seven vectors. These vectors map some aspects of the game such as the number of pieces played, the number of pieces in hand, etc. Second is proposed a heuristic evaluation function to the choice of the best move, comprised of two terms. The first term takes into account the number of points marked in a move. The second one incorporates the game strategy. In the results section are shown some simulation results to games of four persons, grouped in two groups. The group that used the heuristic function proposed in this paper wins the game in 66% of the simulations. Future works are proposed that makes use of a new heuristic function associated with genetic algorithm as an optimization tool.

Keywords: domino game, heuristic function, game state.

Authors' contact:
{ccosta,mcosta}@ufam.edu.br
* nirvana.sa@gmail.com

1. Introdução

O dominó é composto de 28 peças (pedras) chatas, retangulares As 28 pedras têm duas metades, cada uma dessas metades contém uma numeração que varia de zero (vazio) a seis pontos, formando várias combinações. Na forma clássica do jogo, são sete números (de zero a seis), combinados entre si. Matematicamente: C(7,2) + 7 = C(8,2) = 28. O dominó pode ser jogado em duplas adversárias onde cada jogador recebe 7 peças ou alternativamente pode ser jogado por apenas 2 jogadores com 7 pedras cada um e 14 pedras para comprar no caso do oponente não ter a pedra da vez. Existem várias formas de se jogar dominó, a mais comum é o dominó de 2 pontas. No estado do Amazonas joga-se outro tipo pouco difundido, o dominó de 4 pontas, objeto desse trabalho.

No dominó de 2 pontas, o objetivo do jogo é tão somente conseguir colocar a última peça no tabuleiro do jogo. Durante a partida, são adotadas algumas estratégias para o jogador "bater", isto é, ser o primeiro a desfazer-se de todas as suas pedras. As jogadas visam encaixar alguma peça nas peças que estão nas pontas do jogo, uma por vez e minimizar a possibilidade do adversário encaixar uma pedra em uma das pontas e passe. Caso algum jogador tenha batido o jogo, sua dupla leva todos os pontos das peças que estão nas mãos dos adversários. A partida pode terminar em duas circunstâncias: quando um jogador consegue bater o jogo, ou quando o jogo fica trancado. Caso o jogo fique trancado, contam-se todos os pontos conseguidos por cada dupla. A dupla que possuir menos pontos é a vencedora, e leva todos os pontos da dupla adversária.

No dominó de 4 pontas, a disputa ocorre, em geral, entre duplas. Cada jogador deve ter 7 pedras em mãos no início de uma rodada. Na primeira rodada da competição, a primeira pedra a ser jogada deverá ser a de numeração 6-6, chamada de "carroça de sena" por ter suas duas metades iguais a seis. Os jogadores têm a possibilidade de abrir até quatro pontas de jogo a partir da carroça 6-6. Na figura 1 mostra-se um jogo onde já foram abertas 3 pontas. O objetivo principal dessa versão do jogo é alcançar uma pontuação igual ou superior a 200 pontos, que pode ser alcançada em uma ou mais rodadas. Em cada jogada existe a possibilidade de se pontuar 5, 10, 15,..., 50 pontos. Pode-se marcar pontos (um múltiplo de 5) em cinco situações distintas:

P_1) A soma dos pontos das pontas da mesa é múltiplo de 5, sendo a pontuação igual a essa soma. Na figura 1 mostra-se um exemplo de jogada que rende 15 pontos equivalente a soma dos pontos das pontas abertas (2+5+8). Na figura 2 mostra-se um exemplo de jogada onde não se pontua, pois a soma dos pontos nas pontas abertas é 14 (1+5+8).

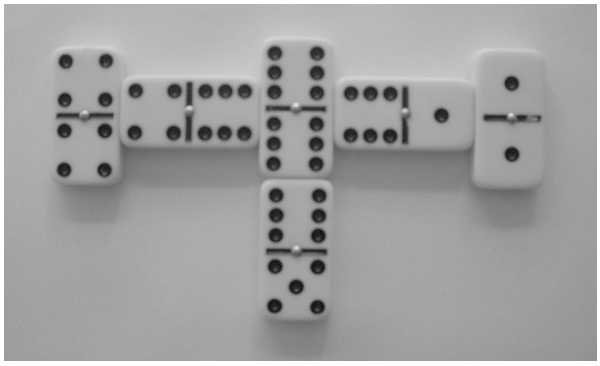


Figure 1: Exemplo de jogada do dominó de 4 pontas onde pontua-se 15 pontos.

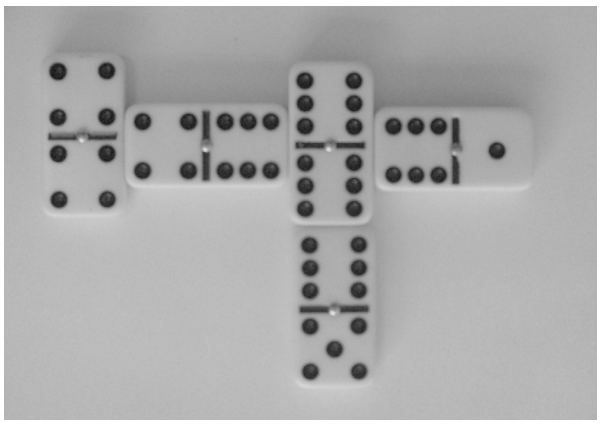


Figure 2: Exemplo de jogada do domino de 4 pontas onde não se pontua, pois a soma dos pontos nas pontas abertas é igual a 14.

P_2) O jogador adversário "passa" em sua vez de jogar, ou seja, não possui pedra que se encaixe em nenhuma das 4 pontas. O impedimento de jogada do adversário rende a dupla que provocou o passe 20 pontos.

P_3) O jogador provoca o passe de todos os demais jogadores (inclusive o parceiro) e a vez volta para o jogador ele. Esse passe geral, chamado de "galo", equivale a uma pontuação de 50 pontos.

P_4) Quando um jogador "bate" (consegue encaixar todas as pedras que dispunha). Nesse caso, somam-se todos os pontos das peças que estão nas mãos dos adversários. Essa soma é denominada de "garagem". A pontuação da jogada corresponde ao maior múltiplo de 5 menor ou igual a essa soma. Na figura 3 mostra-se um exemplo em que a soma dos pontos dos adversários é igual a 12, sendo a pontuação da jogada igual a 10 pontos;

P_5) A última peça descartada pelo jogador que bateu é uma "carroça", ou seja, uma pedra que tem os dois lados com numerações iguais, sendo a pontuação da jogada igual a 20 pontos.

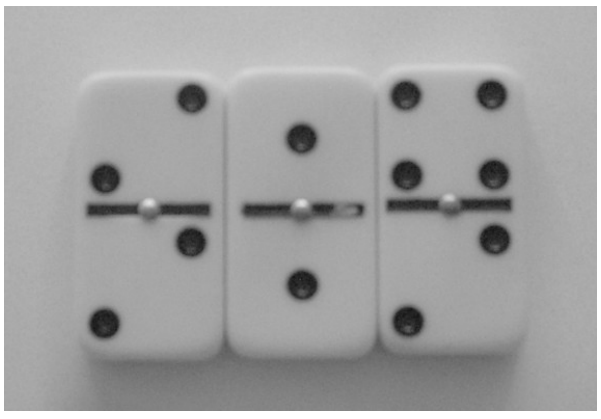


Figure 3: Exemplo de "garagem" onde a soma de pontos da dupla adversária é igual a 12. A pontuação da jogada é igual a 10 pontos.

Nas rodadas seguintes a primeira, o jogador que "bateu" na rodada anterior deverá iniciar com a carroça de sua escolha.

No dominó de quatro pontas, devido ao fato do objetivo e das regras serem mais elaboradas, as estratégias de jogo são mais complexas que às utilizadas no dominó de duas pontas.

As estratégias utilizadas durante o jogo do dominó de 4 pontas, além do objetivo de pontuar, almejam também facilitar as ações futuras de quem joga (ou de seu parceiro) e dificultar a ação dos jogadores da dupla adversária. Quando no início de uma rodada, um jogador tem em mãos quatro ou mais peças que apresentem a mesma numeração em um das metades (vide exemplo na figura 4), a estratégia recomendada é fazer com que essa numeração esteja presente no maior número possível de pontas, com o intuito tanto de fazer seus adversários "passarem" como de facilitar suas jogadas futuras. Para o conjunto de pedras mostrado na figura 4 é interessante buscar uma configuração de jogo onde exista um maior número de pontas com a numeração 5.

O jogo de domino de 4 pontas caracteriza-se por ser um problema multiagente [Sycara 1998]. Trata-se de um jogo de vários jogadores, de soma não zero, de informações imperfeitas. Na sua modalidade mais jogada, duas duplas, caracteriza-se como sendo de dois jogadores, pois só existem duas pontuações. Por ser um jogo com informações imperfeitas, um algoritmo de busca para selecionar a melhor jogada explora um espaço de estados de crença (crenças sobre quem tem determinadas peças e com que probabilidade) [Russel e Norvig 2004]. Tais algoritmos utilizam raciocínios probabilísticos complexos.

Nesse trabalho propõe-se a pesquisa de um heurística que possibilite o desenvolvimento de um agente inteligente para o jogo do dominó de 4 pontas. Procurou-se uma alternativa mais simples para a escolha da melhor jogada, baseada na utilização de uma função de avaliação que combina nos seus termos informações sobre o estado presente do jogo. Tal função incorpora informações estratégicas que visam facilitar as ações futuras da dupla do jogador que joga e dificultar as ações futuras da dupla adversária.

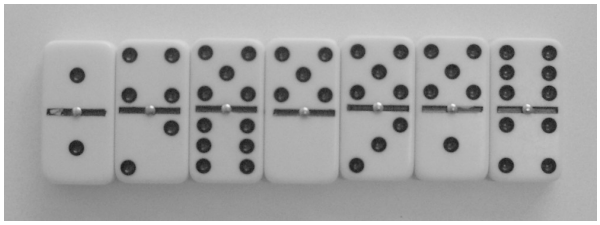


Figure 4: Exemplo de um conjunto de 7 peças iniciais com 4 peças contendo a numeração 5.

Na seção trabalhos relacionados analisa-se alguns trabalhos relacionados com a aplicação de funções de avaliação a jogos. Na seção metodologia propõe-se um conceito de estado para o jogo de dominó de 4 pontas e propõe-se uma função heurística para o mesmo em função do estado proposto. Na seção de resultados mostra-se o desempenho de um jogo de duas duplas em função de algumas escolhas feitas para os parâmetros da função heurística. Na seção de discussão propõe-se a continuidade do trabalho através da exploração de outros valores para os parâmetros da função heurística e através da utilização de algoritmos genéticos.

2. Trabalhos Relacionados

A utilização de funções de avaliação associadas ao algoritmo alfa-beta para a escolha da melhor opção de jogada em jogos de informações perfeitas foi inicialmente proposta por Shannon [Shannon 1950]. Tal proposta foi implementada para os jogos de xadrez [Campbell et al. 2002] e de damas [Schaeffer, 1997].

Para os jogos de informação imperfeita a utilização da proposta de Shannon envolve raciocínios complexos sobre as estratégias do jogo. Cita-se como exemplo o programa Bridge Baron [Smith et al. 1998], desenvolvido para o jogo de Bridge. Para o jogo de dominó de 2 pontas, encontram-se na literatura alguns trabalhos que buscam a melhor estratégia do jogo, como Yen e Chlebus [Yen 1992, Chlebus, 1986]. Não encontrou-se, porém trabalhos relacionados ao dominó de 4 pontas.

3. Metodologia

A função de avaliação proposta para a escolha da melhor jogada é constituída pela soma de dois termos, conforme mostra a expressão (1). A variável n identifica uma opção de jogada para a qual a função de avaliação é calculada. O termo T_1 corresponde aos pontos obtidos ao se efetuar a opção de jogada n. O termo T_2 incorpora a estratégia do jogo. O mesmo corresponde a valores inteiros positivos que procuram retratar como a escolha pela jogada n pode facilitar ações futuras do jogador que efetua a jogada e dificultar ações futuras dos jogadores adversários.

$$f(n) = T_1 + T_2 \quad (1)$$

Em que:
n – opção de jogada a ser avaliada;
T_1 – soma dos pontos obtidos com a jogada n;
T_2 – termo que incorpora a estratégia do jogo.

Antes que seja feito o detalhamento de como se calculam os termos T_1 e T_2 da função f(n) é necessário definir-se um estado para o jogo de dominó de 4 pontas. O estado do jogo é definido nesse trabalho através de um conjunto de vetores, V_i , onde $0 \leq i \leq 6$. Esses vetores expressam estatísticas que são atualizadas em cada jogada. Os quatro primeiros vetores contêm valores inteiros enquanto que os três últimos contêm valores binários. A definição dos mesmos é feita a seguir.

1) V_0: Vetor contendo a quantidade de peças em jogo para cada numeração. Esse vetor é expresso por: $V_0 = \{a_0, b_0, c_0, d_0, e_0, f_0, g_0\}$. Em que: a_0 corresponde ao número de peças já jogadas com a numeração 0; b_0 corresponde ao número de peças já jogadas com a numeração 1, e assim sucessivamente;

2) V_1: Vetor contendo a quantidade de peças na mão para cada numeração. Esse vetor é expresso por: $V_1 = \{a_1, b_1, c_1, d_1, e_1, f_1, g_1\}$. Em que: a_1 corresponde ao número de peças na mão com a numeração 0; b_1 corresponde ao número de peças na mão com a numeração 1, e assim sucessivamente;

3) V_2: Vetor contendo a quantidade de peças nas pontas para cada numeração. Esse vetor é expresso por: $V_2 = \{a_2, b_2, c_2, d_2, e_2, f_2, g_2\}$. Em que: a_2 corresponde ao número de peças nas pontas com a numeração 0; b_2 corresponde ao número de peças nas pontas com a numeração 1, e assim sucessivamente;

4) V_3: Vetor contendo a quantidade de peças já jogadas pelo parceiro de dupla para cada numeração . Esse vetor é expresso por: $V_3 = \{a_3, b_3, c_3, d_3, e_3, f_3, g_3\}$. Em que: a_3 corresponde ao número de peças já jogadas pelo parceiro com a numeração 0; b_3 corresponde ao número de peças já jogadas pelo parceiro com a numeração 1, e assim sucessivamente;

5) V_4: Vetor que indica as numerações onde o adversário seguinte já passou. Esse vetor é expresso por: $V_4 = \{a_4, b_4, c_4, d_4, e_4, f_4, g_4\}$. Em que: a_4 indica se o adversário seguinte já passou ou não para a numeração 0. Se a_4 for igual a 1 o adversário seguinte já passou para a numeração 0. Se a_4 for igual a 0 o adversário seguinte ainda não passou para a numeração 0; b_4 indica se o adversário seguinte já passou ou não para a numeração 1. Se b_4 for igual a 1 o adversário seguinte já passou para a numeração 1. Se b_4 for igual a 0 o adversário seguinte ainda não passou para a numeração 0, e assim sucessivamente;

6) V_5: Vetor que indica as numerações onde o adversário anterior já passou. Esse vetor é expresso por: $V_5 = \{a_5, b_5, c_5, d_5, e_5, f_5, g_5\}$ Em que: a_5 indica se o adversário anterior já passou ou não para a numeração 0. Se a_5 for igual a 1 o adversário anterior já passou para a numeração 0. Se a_5 for igual a 0 o adversário anterior ainda não passou para a numeração 0; b_5 indica se o adversário anterior já passou ou não para a numeração 1. Se b_5 for igual a 1 o adversário anterior já passou para a numeração 1. Se b_5 for igual a 0 o adversário anterior ainda não passou para a numeração 0, e assim sucessivamente;

7) V_6: Vetor que indica as numerações onde o parceiro de dupla já passou. Esse vetor é expresso por: $V_6 = \{a_6, b_6, c_6, d_6, e_6, f_6, g_6\}$. Em que: a_6 indica se o parceiro de dupla já passou ou não para a numeração 0. Se a_6 for igual a 1 o parceiro de dupla já passou para a numeração 0. Se a_6 for igual a 0 o parceiro de dupla ainda não passou para a numeração 0; b_6 indica se o parceiro de dupla já passou ou não para a numeração 1. Se b_6 for igual a 1 o parceiro de dupla já passou para a numeração 1. Se b_6 for igual a 0 o parceiro de dupla ainda não passou para a numeração 0, e assim sucessivamente;

A seguir analisar-se-á como são calculados os termos T_1 e T_2. O termo T_1 incorpora no cálculo da função de avaliação os pontos obtidos ao se realizar a opção de jogada *n*. Na seção de introdução foram listadas 5 situações distintas em que se pode pontuar numa jogada. O termo T_1 incorpora no seu cálculo os pontos que podem ser obtidos através das situações P_1, P_2, P_3 e P_5. Assim T_1 é dado por (2):

$$T_1 = P_1 + P_2 + P_3 + P_5 \quad (2)$$

Observar que T_1 não incorpora a situação P_4, que corresponde aos pontos de garagem, pois os mesmos, na medida em que se encontram nas mãos da dupla adversária, não podem ser contabilizados. Os termos P_2 e P_3 são calculados através de um único algoritmo. Para o cálculo dos mesmos são utilizados os vetores V_4, V_5 e V_6. Na figura 5 mostra-se um fluxograma desse algoritmo. Para o entendimento desse fluxograma e do texto que se segue as seguintes definições são necessárias:

Np_1, Np_2, Np_3 e Np_4 – valores inteiros correspondentes as numerações existentes nas pontas 1, 2 ,3 e 4 do jogo, respectivamente. A numeração das pontas não obedece a nenhuma ordem pré-definida;
L – peça a ser inserida no jogo na opção de jogada *n*.
L_1 - valor inteiro correspondente a numeração da metade da peça *L* que coincide com um ou mais dos valores Np_1, Np_2, Np_3 ou Np_4.
L_2 – valor inteiro correspondente a numeração da metade da peça *L* que não coincide com um ou mais dos valores Np_1, Np_2, Np_3 ou Np_4.

Para o fluxograma mostrado na figura 5 supõe-se que $L_1=Np_4$.

O termo T_2 incorpora no seu cálculo três parcelas distintas relacionadas com a estratégia do jogo. A expressão para T_2 é mostrada em (3).

$$T_2 = \alpha .\left(- E_1 + E_2 + \delta E_3\right) \quad (3)$$

O valor de α permite modelar a importância da estratégia (T_2) em relação aos pontos obtidos com uma jogada (T_1).

A parcela E_1 considera a possibilidade de se fazer o adversário seguinte passar na sua vez de jogar. Para o entendimento da parcela E_1 será utilizada a seguinte situação hipotética: considere que em duas das pontas do jogo exista uma numeração n_1 e que o jogador da vez tenha em mãos mais três peças com essa mesma numeração. Então, 5 das 7 peças possíveis com a numeração n_1 não pertencem ao adversário seguinte, sendo grande a possibilidade do mesmo passar se todas as pontas estiverem com essa numeração. Tendo em vista a possibilidade de fazer o adversário seguinte passar, e dessa forma obter 20 pontos, é desejável que não seja descartada nenhuma das peças em que $L_1=n_1$. Assim sendo a opção de jogada de uma peça *L* em que $L_1=n_1$ não é desejada do ponto de vista estratégico e daí o sinal negativo para E_1 na expressão (3). A determinação de E_1 é realizada conforme mostrado na expressão (4). Quanto maior o número de peças com a numeração L_1 nas pontas e nas mãos de quem joga maior será o valor de E_1. Por outro lado, o valor de E_1 deve ser tanto maior quanto maior for o número de peças com a numeração L_1 já jogadas. O valor de K_1 controla a importância da parcela E_1 frente às outras parcelas utilizadas no cálculo de T_2.

$$E_1 = K_1.\left(V_1(L_1) + V_2(L_1) + V_0(L_1)\right) \quad (4)$$

A parcela E_2 tem por objetivo facilitar as ações futuras de quem joga. A determinação de E_2 é realizada conforme mostrado na expressão (5). Essa expressão é muito semelhante àquela proposta para E_1, sendo a única diferença que ao invés de L_1 utiliza-se L_2 como argumento independente. O valor de K_2 controla a importância da parcela E_2 frente às outras parcelas utilizadas no cálculo de T_2.

$$E_2 = K_2.\left(V_1(L_2) + V_2(L_2) + V_0(L_2)\right) \quad (5)$$

A parcela E_3 reforça o objetivo da parcela E_2, visando facilitar as ações futuras de quem joga quando o número de peças na mão é menor ou igual a 3. No fluxograma da figura 6 mostra-se como é realizado o cálculo de E_3. O valor de K_3 controla a importância da parcela E_3 frente as outras parcelas utilizadas para o cálculo de T_2.

4. Resultados

Para se testar a heurística proposta foram realizadas simulações de um jogo com duas duplas de jogadores. A primeira dupla, denominada dupla 1, realizou a escolha da melhor jogada utilizando a função f_1 dada pela expressão (6).

$$f_1 = T_1 \quad (6)$$

A segunda dupla, denominada dupla 2, realizou a escolha da melhor jogada utilizando a função f_2 dada pela expressão (7):

$$f_2 = T_1 + T_2 \quad (7)$$

Os valores de α na expressão para f_2 foram variados entre 0 e 10, em incrementos de 0,1 entre 0 e 1 e em incrementos de 0,25 entre 1 e 10. Aos valores de K_1, K_2 e K_3 foram atribuídos o mesmo valor, 2, fazendo com que E_1, E_2 e E_3 assumissem igual importância no cálculo de T_2.

Para cada valor de α realizou-se um total de 100.000 partidas e anotou-se o número de vitórias da dupla 1 e da dupla 2. Nas tabelas 1 e 2 mostram-se esses resultados.

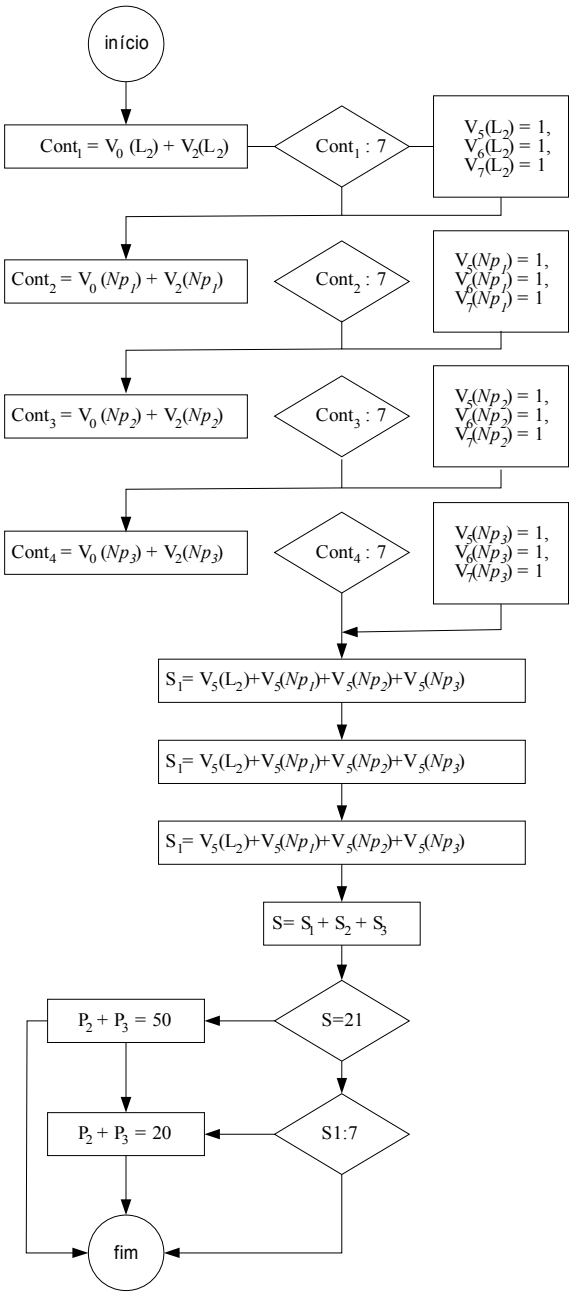


Figure 5: Fluxograma para o cálculo de $P_2 + P_3$.

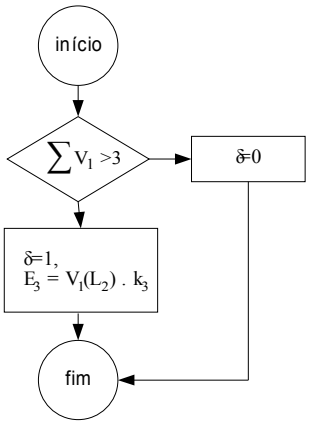


Figure 6: Fluxograma para o cálculo de E_3.

Tabela 1: Resultados da simulação com α variando no intervalo entre 0 e 5.

Coef. α	Número de Vitórias em 100.000 partidas	
	Dupla 1	Dupla 2
0	50248	49752
0,01	36541	63459
0,02	36541	63459
0,03	36541	63459
0,04	36541	63459
0,05	36504	63496
0,10	36504	63496
0,15	36504	63496
0,20	36504	63496
0,25	36579	63421
0,50	36278	63722
0,75	35565	64435
1,00	35052	64948
1,25	34659	65341
1,50	34233	65767
1,75	33998	66002
2,00	34076	65924
2,25	33981	66019
2,50	34300	65700
2,75	34383	65617
3,00	34507	65493
3,25	34574	65426
3,50	35018	64982
3,75	35267	64733
4,00	35510	64490
4,25	35571	64429
4,50	35574	64426
4,75	35574	64426
5,00	36378	63622

Tabela 2: Resultados da simulação com α variando entre 5,25 e 10.

Coeficiente α	Número de Vitórias em 100.000 partidas	
	Dupla 1	Dupla 2
5,25	36907	63093
5,50	36907	63093
5,75	36907	63093
6,00	36909	63091
6,25	36937	63063
6,50	36948	63052
6,75	37162	62838
7,00	37162	62838
7,25	37163	62837
7,50	37734	62266
7,75	38236	61764
8,00	38236	61764
8,25	38236	61764
8,50	38265	61735
8,75	38265	61735
9,00	38265	61735
9,25	38270	61730
9,50	38270	61730
9,75	38273	61727
10,00	38839	61161

4. Discussão

Através da observação das tabelas 1 e 2 observa-se que: A dupla 2, a exceção de α=0, sempre obteve um número de vitórias maior que a dupla 1; O maior número de vitórias obtidas pela dupla 2 foi 66002, tendo ocorrido para um valor de $\alpha = 2.25$; Para $\alpha = 0$, quando as duas funções tornam-se iguais, $f_1=f_2$, o número de vitórias das duas duplas é praticamente igual; Quando α varia de 0 para 0,01, o número de vitórias da dupla 2 cresce de forma abrupta, de 49752 para 63459. As outras variações observadas para o número de vitórias da dupla 2 em função de α são mais suaves.

As observações anteriores permitem concluir que a dupla que escolheu a melhor jogada utilizando a função de avaliação f_2 teve um desempenho superior a dupla que escolheu a melhor jogada utilizando a função de avaliação f_1. A responsabilidade por esse melhor desempenho cabe ao termo de estratégia T_2 presente na função f_2 e ausente na função f_1. O melhor resultado alcançado foi de 66% de vitórias para a dupla que escolheu a melhor jogada utilizando a função f_2.

Para obtenção desses resultados utilizou-se um mesmo valor para K_1, K_2 e K_3 igual a 2, escolhido de forma empírica.

Em trabalhos futuros pretende-se: Realizar outras simulações com a função de avaliação f_2 associando α a diferentes valores de K_1, K_2 e K_3; Utilizar algoritmo genético para otimização de uma outra função de avaliação onde o termo T_2 seja dado pela expressão (8). Essa expressão reúne as parcelas presentes nos termos E_1, E_2 e E_3. Através do algoritmo genético serão obtidos os valores de α_1, α_2, α_3, α_4, α_5 e α_6 que otimizarão o número de vitórias da dupla que utilizar a função f_2 para a escolha da melhor jogada.

$$T_2 = \alpha_1.V_1(L_1) + \alpha_2.V_2(L_1) + \alpha_3.V_0(L_1) + \alpha_4.V_1(L_2) + \alpha_5.V_2(L_2) + \alpha_6.V_0(L_2) \quad (8)$$

5. Conclusão

Propôs-se nesse trabalho uma nova metodologia para a escolha da melhor jogada no dominó de 4 pontas, utilizando uma função heurística que, para o seu cálculo, utiliza informações provenientes do "estado do jogo", definido como um conjunto de 7 vetores atualizados a cada jogada. Os parâmetros utilizados na função heurística não foram otimizados nesse trabalho. Propõe-se isso seja feito em trabalho futuro. Os resultados foram avaliados através de simulações com jogos de duplas. O melhor resultado alcançado em 100.000 partidas foi de 66% de vitórias para a dupla que utilizou a função de avaliação proposta nesse trabalho. Espera-se que o processo de otimização proposto para trabalhos futuros resulte em uma estatística de vitórias mais elevada. Acredita-se que a metodologia apresentada nesse trabalho para a escolha da melhor jogada seja original. Preetende-se, em trabalhos futuros, explorar a aplicação dessa metodologia em outros tipos de jogos.

Agradecimentos

Esse trabalho teve o apoio financeiro da SUFRAMA (convênios 068/2001 e 069/2001) e do CNPq (bolsa de IC).

Referências

CAMPBELL, M.S., HOANE, A.J., HUS, F.H. 2002, Deep Blue, Artificial Intelligence, 134(1-2), 57-83.

CHLEUBS, B.S., 1986, Domino-tiling games, Journal of Computer and System Sciences, v. 32, n.3, 374-392.

RUSSEL, S., NORVIG, P., 2004, Inteligência artificial, Elsevier, 2ª edição;

SHAEFFER, J., 1997, One Jump Ahead: Challenging Human Supremacy in Checkers, Springer-Verlag, Berlin.

SHANNON, C.E. , 1950, Programming a computer for playing chess. Philosophical Magazine, 41(4), 256-275.

SMITH, S.J.J, NAU, D.S., THROOP, T.A., 1998, Success in spades: Using ai planning techniques to win the world championship of computer bridge. In Proceedings of the Fifteenth National Conference on Artificial Intelligence, 1079-1086, Madison, Wisconsin, AAAI Press.

SYCARA, K.P., 1998. Multiagent systems, AI Magazine, v. 19, n. 2, 79-92.

YEN, H.C., 2002. A multiparameter analysis of domino tiling with an application to concurrent systems, Theoretical Computer Science, v. 98, n. 2, 263-287.

Event Relations in Plan-Based Plot Composition

Angelo E. M. Ciarlini[1] Simone D. J. Barbosa[2] Marco A. Casanova[2] Antonio L. Furtado[2]
[1]UNIRIO, Depto. de Informática Aplicada, Brasil [2]PUC-Rio, Depto. de Informática, Brasil

Abstract
The process of plot composition in the context of interactive storytelling is considered under a fourfold perspective, in view of syntagmatic, paradigmatic, antithetic and meronymic relations between the constituent events. These relations are shown to be associated with the four major tropes of semiotic research. A conceptual model and set of facilities for interactive plot composition and adaptation dealing with the four relations is described. To accommodate antithetic relations, corresponding to the irony trope, our plan-based approach leaves room for the unplanned. A simple storyboarding prototype tool has been implemented to conduct experiments.
Keywords: Interactive Storytelling, Plots, Planning, Narratology, Tropes.
Authors' contact:
angelo.ciarlini@uniriotec.br
{simone,casanova,furtado}@inf.puc-rio.br

1. Introduction
The role of storytelling in games has long been the subject of lively debates [Wardrip-Fruin & Harrigan]. Although some believe that story and game are in direct opposition [Costikyan], most agree that successful narrative in games is possible, and a few argue for the importance of story creation as part of gameplay [Wallis]. However, a different sort of narrative is required: it must be non-linear and play-centric, that is, it must revolve around the player's experience [Pearce]. The player is no longer a mere consumer of the narrative, but both a consumer and a (co-) producer of the plot. The game designer typically selects a genre. In game playing, interactive storytelling emerges, but care must be taken to ensure that the basic rules of the genre, as well as corresponding tropes and narrative structures, are understood by the co-authors of the story [Wallis].

A few computational systems and approaches have been proposed to support interactive storytelling. Some of them focus on the interaction among characters [Cavazza et al.], whereas others focus on plot structure and coherence [Grasbon & Braun], and a few others attempt to combine both [Mateas & Stern]. What kind of system would be suitable for assisting users in creating stories within games or other interactive storytelling contexts? *Planning algorithms* have proven to be a useful alternative to help create narratives by exploring different chains of events to achieve the characters' or the storytellers' goals [Ciarlini et al.; Riedl & Young]. In game playing, planning algorithms make it practical to create non-linear narratives that are both coherent and diverse, by allowing players to proceed in different courses of actions with varying results, and yet respecting the game structure, rules and constraints.

To support the production of stories, we have drawn on what semiotic research has singled out as the *four major tropes* [Burke], namely: metaphor, metonymy, synecdoche, and irony. By offering mechanisms derived from these tropes, we intend both to augment the expressiveness of narrative models and to provide better support to authors who are less familiar with or confident in creating and telling stories.

In this paper, we associate those tropes with four types of relations between narrative events: *syntagmatic*, *paradigmatic*, *meronymic* and *antithetic*. They play a basic role in an interactive plan generating system that creates plots within a predefined genre.

Narratology studies distinguish three levels in literary composition: fabula, story and text [Bal]. In the present work, we stay at the *fabula* level, where the characters acting in the narrative are introduced, as well as the narrative plot, consisting of a partially-ordered set of events. We focus on plots whose constituent events happen as a consequence of a predefined repertoire of actions, which we shall call *operations*, deliberately performed by the characters. Plot composition will be treated here as a plan generation process, and hence the terms plot and plan will be used interchangeably. Yet, since narratives are often more attractive when unplanned shifts can occur, the user shall retain the power to issue certain *directives* when interventions are needed or desired.

Starting from such considerations, this paper proposes a fourfold way to characterize plot composition at the fabula level. Section 2 describes the relations between events in correspondence with the four major tropes. Section 3 outlines how we model an intended genre, to whose conventions the plots must conform. Section 4 sketches, over a simple example, the main features of our plan-based prototype tool. Concluding remarks are presented in section 5.

2. From Tropes to Event Relations
It has been suggested that the four major rhetorical tropes provide models for remarkably comprehensive analyses in different areas [Burke; Chandler; White]. They all involve *relations between pairs of words*, thanks to which, given two related words w1 and w2, a person can meaningfully use w1 to refer to w2.

They are not defined in a uniform way by linguists, there being much disagreement, especially on the distinction between metonym and synecdoche. A useful discussion is found in [Chandler], where many practical applications of Burke's four tropes theory are surveyed.

Metaphor [Lakoff & Johnson; Ortony] and synecdoche [Chandler] have to do with hierarchical structures such as those represented in ontologies [Breitman, Casanova & Truszkowski]. If one concept C_1 can be metaphorically used to denote another concept C_2, the two concepts are said to be similar or analogous, and are placed under a more general concept $\hat{C}$ that subsumes both of them. C_1 and C_2 would be represented in the network with **is-a** links connecting them to $\hat{C}$. Also, one could add an **is-like** link from C_1 to C_2 [Breitman et al.]. Clearly, metaphor is a displacement along the verbal

paradigmatic axis [Saussure], from which we took the suggestion of a *paradigmatic relation* between events.

In synecdoche, concept C_1 is used to denote concept C_2, if C_1 is a part of C_2 (which calls for another link, C_1 **part-of** C_2); the converse substitution, from whole to part, is also usual in common parlance. The corresponding association between events is called *meronymic relation* in the present paper.

According to [Chandler], metonyms are based on various indexical relationships between concepts, notably the substitution of effect for cause, and convey an idea of contiguity. Borrowing again from [Saussure], we require the presence of s*yntagmatic relations* between events, to justify their being meaningfully placed in sequence.

Irony is the most intriguing of the four tropes. In verbal communication, it reflects the opposite of the thoughts or feelings of the speaker or writer (as when you say 'I love it' when you hate it) or the opposite of the truth about external reality (as in 'there's a crowd here' when it is deserted). It also takes the form of substitution by *dissimilarity* or *disjunction*. Variations such as understatement and overstatement can also be regarded as ironic. At some point, exaggeration may slide into irony [Chandler]. Disclosing paradoxes and hidden agendas in literary texts, in sharp contrast between the declared intentions and the real ones, is another source of irony, constituting a trend in critical studies known as *deconstruction* [Culler].

Not only mental attitudes, feelings and statements can be ironic – actions can also be ironic, but always in an unplanned, non-deliberate fashion. Irony is in fact a characteristic of certain intrigue situations that are often referred to as *dramatic irony* [Booth].

Consequently any kind of irony induces an *antithetic relation* between events that look, in principle, incompatible with each other, given their dependence on contexts characterized by radically opposite properties. Mediating two such events, the until then well-behaved world must suffer a disruptive shift, whereby the truth value of certain facts or beliefs is inverted, or certain properties move from one extreme to the other within the ascribed value range (e.g. from helplessly weak to heroically strong).

To illustrate the event relations derived from the major tropes, we shall employ a simple example to be referenced along the paper. Consider four types of events, all having one woman and two men as protagonists: *abduction*, *elopement*, *rescue*, and *capture*. As demonstrated in folktale studies [Propp], many plots mainly consist of an act of villainy, i.e. of a violent action that breaks the initially stable and peaceful state of affairs, followed ultimately by an action of retaliation, which may or may not lead to a happy outcome.

Propp distinguished seven character roles (*dramatis personae*) according to the events assigned to each one's initiative: hero, villain, victim, dispatcher, donor, helper, false hero. Curiously, in literary texts involving the four events above, this distribution is not unique: we called the violent initial act "villainy", but the perpetrator of abduction, and more often of elopement, can be the hero of the narrative, and in such cases the woman's original guardian (husband, father) is regarded as the villain.

2.1 Syntagmatic Relations

To declare that it is legitimate to continue a plot containing abduction by placing rescue next to it, we say that these two events are connected by a *syntagmatic relation*. More precisely, we can define the semantics of the two events in a way that indicates that the occurrence of the first leaves the world in a state wherein the occurrence of the second is coherent. Similarly, a plot involving elopement followed by capture looks natural, and hence these two events are likewise related.

The syntagmatic relation between events induces a weak form of causality or enablement, which justifies their *sequential ordering* inside the plot.

2.2 Paradigmatic Relations

The events of abduction and elopement can be seen as *alternative* ways to accomplish a similar kind of villainy. Both achieve approximately – though not quite – the same effect: one man takes away a woman from where she is and starts to live in her company at some other place. There are differences, of course, since the woman's behaviour is usually said to be coerced in the case of abduction, but quite voluntary in the case of elopement. In fact, it is usual to assume that a sentence such as "Helen elopes with Paris", implies that Helen had fallen in love with Paris.

To express that abduction and elopement play a similar function, we say that there is a *paradigmatic relation* between the two events. Likewise, this type of relation is perceived to hold between the events of rescue and capture, which are alternative forms of retaliation. And, again, there is a difference between the woman's assumed attitude, associated as before with her feelings. An abducted woman expects to be rescued from the villain's captivity by the man she loves. On the contrary, she will only return through forceful capture if she freely eloped with the seducer.

As the present example suggests, the syntagmatic and the paradigmatic axes identified by Saussure are really *not* orthogonal in that the two relations cannot be considered independently when composing a plot. Thus, in principle, the two pairs enumerated in the previous section (abduction-rescue and elopement-capture) are the only normal combinations, the former illustrated by the Sanskrit *Ramayana* [Valmiki] and the similarly structured Arthurian romance of *Lancelot* [Chrétien; Furtado & Veloso], and the latter by the Irish *Story of Deirdre* [McGarry]. Yet the next section shows that such limitations can, and even should, be waived occasionally.

2.3 Antithetic relations

While normal plots, whose outcome is fully determined, can be composed exclusively on the basis of the two preceding relations, the possibility to introduce unexpected turns is often desirable in order to make the plots more attractive – and this requires the construct that we chose to call *antithetic relation*. A context where a woman suffers abduction by a ravisher whom she does not love would seem incompatible with a capture event, since there should be no need to employ force to bring back the victim. So, in this sense, abduction and capture are in antithetic relation.

The mythical *Rape of the Sabines* shows what can happen as a consequence of a drastic reversal of the circumstances. King Romulus is facing a problem at the newly founded city of Rome: the population is entirely male at first. To remedy the lack, he leads his men to break into the dwellings of the Sabines and abduct their women. Sometime afterwards the Sabine warriors march against the Romans, but the women have no wish to be taken back, leaving to their countrymen no option except

their capture. King Romulus's men had lawfully married them and made them bear children. A Roman chronicle [Titus Livius] reports the radical change in the women's feelings, and tells how the seemingly inevitable confrontation ended with the reconciliation of the two parties.

In contrast, modern history provides some distinctly regrettable examples of abduction actually followed by capture, categorized by psychiatrist Nils Bejerot as the *Stockholm syndrome*. One case in point is the abduction by a group of terrorists of the daughter of a millionaire, who ended up joining her tormentors in the practice of crimes, and was captured by the police in San Francisco [Hearst & Moscow].

The occurrence of elopement followed by rescue provides a much stronger case of antithetic relation. Indeed, elopement only makes sense if the victim loves the seducer, whereas, for this very motive, she would resist to any attempt to rescue her, leaving forceful capture as the only viable alternative. Even so the legendary story of *Helen of Troy*, in spite of various discordant interpretations, seems to offer a counter-example. Married to king Menelaus of Sparta, Helen fled to Troy in the company of Paris, out of her free will according to a number of versions (e.g. the *Heroides* [Ovid]). But, after their escapade to Troy where they married, her love feelings started to wane while the Trojan War followed its bloody course and she kept recalling the far manlier Menelaus. The *Iliad* [Homer] signals repeatedly this critical change of sentiment. At the end her recovery turned from capture into rescue, as registered in the *Aeneid* [Virgil]. Paris was dead, and she had been delivered to Paris's brother Deiphobus. When the Greeks came out of the wooden horse and stormed the Trojan palaces, Helen herself made sure that Menelaus should win – and know that she was helping him in atonement for her previous misconduct. The shadow of Deiphobus tells the episode to Aeneas; and what better example of irony could we find than his calling Helen "this peerless wife"?

One more example appears in the story of *Tristan and Isolde*, in several versions [Marchello-Nizia]. The knight had eloped with the queen; they were living in harsh conditions in a forest. The dramatic change of their love feelings, which allowed Isolde's rescue by king Mark to be achieved through a simple invitation, with no need to fight, had a very curious cause – the timely expiry date of the love potion they had drunk before, when sailing from Ireland to Cornwall [Béroul].

Generally speaking, if some *binary opposition* – the "to love or not to love" dilemma, in the present case – is allowed to be manipulated via some agency external to the predefined events, then one can have plots that no longer look conventional. A sort of discontinuity is produced by such radical shifts in the context. Intervening between abduction and capture, or between elopement and rescue, a sudden change of feelings can give rise to these surprising sequences. Also, both in fiction and in reality, things not always proceed according to planned events. Natural phenomena and disasters, the mere passage of time, the intervention of agents empowered to change the rules, supernatural or magic manifestations, etc., cannot be discounted.

Specifically for the tragedy genre, the *Poetics* [Aristotle] distinguishes between simple and complex plots, characterizing the latter by the occurrence of *recognition* (αναγνορισισ) and *reversal* (περιπετεια). Differently from reversal, recognition does not imply that the world itself has changed, but rather the *beliefs* of one or more characters about the actual facts. Because of a change of beliefs, a reason to be added to those enumerated in the previous paragraph, a reversal in the course of actions can take place, usually in a direction totally opposite to what was going on so far. Yet another possible external cause of both recognition and reversal in the tragic scene was the intervention of a god, who was lowered onto the stage using a crane – known, accordingly, as *deus ex machina*.

Aristotle's remarks are clearly relevant to the present discussion of plots in general. Following his lead, we shall admit state changes outside the regular regime of predefined events by allowing the user – literally acting *ex machina* (via the computer...) – to impose variations to the context (both in terms of *facts* and of *beliefs*), and thereby deviate the action from its predicted path.

This extreme device will be necessary to allow the elopement-rescue sequence. We decided, however, not to make it indispensable for abduction-capture, in order to have a chance to present a good example of *erroneous* beliefs, contradicting the actual facts. Criminal records everywhere are full of simulated abduction pacts for drawing a ransom from a deluded family. Conversely, a man can unnecessarily decide that capture is the only way to bring back a woman, if he mistakenly believes her to love the ravisher.

Figure 1 shows the relations thus far discussed.

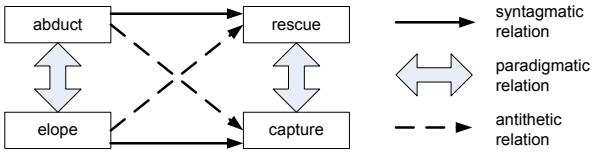


Figure 1: Syntagmatic, paradigmatic, and antithetic relations.

2.4 Meronymic relations

Meronymy is a word of Greek origin, used in linguistics to refer to the decomposition of a whole into its constituent parts. Forming an adjective from this noun, we shall call *meronymic relations* those that hold between an event and a lower-level set of events, with whose help it is possible to provide a more detailed account of the action on hand.

Thus, we could describe the abduction of a woman called Sita by a man called Ravana (characters taken from the *Ramayana* [Valmiki]) as: "Ravana rides from Lanka to forest. Ravana seizes Sita. Ravana carries Sita to Lanka." And her rescue by Rama could take the form: "Rama rides from palace to Lanka. Rama defeats Ravana. Rama entreats Sita. Rama carries Sita to palace." But notice that such decompositions are not fixed, since the lower-level events are selected as required by the current state. For instance, with respect to the rescue event, the hero may already be present at the ravisher's dwelling, or perhaps the victim is not held in captivity, respectively obviating the need for the voyage or for fighting the enemy (Figure 2).

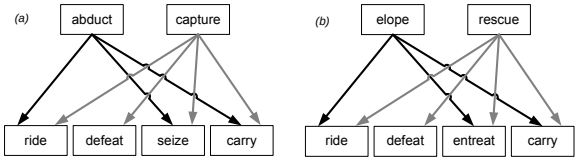


Figure 2: Meronymic relations:
(a) the forceful actions and (b) the gentle actions.

Detailing is most useful to pass from a somewhat abstract view of the plot to one, at a more concrete physical level, that is amenable (possibly after further decomposition stages) to the production of a computer graphics animation [Ciarlini et al.]. Mixed plots, combining events of different levels, do also make sense, satisfying the option to represent some events more compactly while showing the others in detail.

The intuitive notions behind figures 1 and 2 are partly derivable from a context-sensitive grammar:

```
PLOT ::= VILLAINY•RETALIATION
VILLAINY ::= ABDUCT | ELOPE
RETALIATION ::= RESCUE | CAPTURE
ABDUCT•RESCUE ::= abduct, rescue
ELOPE•CAPTURE ::= elope, capture
ABDUCT•CAPTURE ::= *(abduct, capture)belief
ELOPE•RESCUE ::= *(elope, rescue)fact
ABDUCT•RESCUE ::= ABDUCT2•RESCUE2
ELOPE•CAPTURE ::= ELOPE2•CAPTURE2
ABDUCT•CAPTURE ::= *(ABDUCT2•CAPTURE2)belief
ELOPE•RESCUE ::= *(ELOPE2•RESCUE2)fact
ABDUCT2 ::= ride, seize, carry
RESCUE2 ::= ride, defeat, entreat, carry
ELOPE2 ::= ride, entreat, carry
CAPTURE2 ::= ride, defeat, seize, carry
```

3. A Plan-based Modelling Approach

To model a chosen genre, to which the plots to be composed should belong, we must specify at least (to be the object of section 3.1):

a. what can exist at some state of the underlying mini-world,
b. how states can be changed, and
c. the factors driving the characters to act.

In our model, we equate the notion of event with the state change resulting from the execution of a predefined operation. Being defined in terms of their pre-conditions and post-conditions, operations can be readily chained together by a *plan-generating algorithm* [Ciarlini et al.; Barros & Musse] in order to achieve a given goal of some character. As a consequence, it becomes natural to equate plots (sequences of events) with plans (sequences of operations able to bring about the events). Also, to confer a degree of autonomy [Riedl & Young] to the characters performing the operations, it is convenient to make their goals emerge from appropriately motivating situations.

Viewing plots as plans suggests an obvious plot composition strategy, having a plan-generator as its main engine. This and the fact that our conceptual model is expressed in Prolog make the genre specification executable. In sections 3.2, 3.3 and 3.5, we will argue that, duly complemented by auxiliary routines, the planning strategy deals effectively with narrative plots in view of three out of the four event relations. To accommodate antithetic relations, however, it will be necessary to leave room for the *unplanned*, as proposed in section 3.4, leading to plots that may to a limited extent break the conventions of the adopted genre.

3.1 Conceptual schemas

We start with a conceptual design method involving three schemas – static, dynamic and behavioural – which has been developed for modelling literary genres encompassing narratives with a high degree of regularity, such as fairy tales, and application domains of business information systems, such as banking, which are obviously constrained by providing a basically inflexible set of operations and, generally, by following strict and explicitly formulated rules [Furtado et al. 2008]. For brevity, the detailed logic programming notation is omitted; the full specification is shown in Appendix A of our technical report[1].

The *static schema* specifies, in terms of the *Entity-Relationship* model [Batini et al.], the entity and relationship classes and their attributes. In our simple example, `character` and `place` are entities. The attributes of characters are `name`, which serves as identifier, and `gender`. Places have only one identifying attribute, `pname`. Characters are pair-wise related by relationships `loves`, `held_by` and `consents_with`. The last two can only hold between a female and a male character; thus `held_by(Sita,Ravana)` is a *fact* meaning that Sita is forcefully constrained by Ravana, whereas `consents_with(Sita,Ravana)` would indicate that Sita has voluntarily accepted Ravana's proposals. Two relationships associate characters with places: `home` and `current_place`. A state of the world consists of all facts about the existing entity instances and their properties holding at some instant.

The *dynamic schema* defines a fixed repertoire of operations for consistently performing state changes. The *STRIPS* [Fikes & Nilsson] model is used. Each operation is defined in terms of pre-conditions, which consist of conjunctions of positive and/or negative literals, and any number of post-conditions, consisting of facts to be asserted or retracted as the effect of executing the operation. Instances of facts such as `home` and `gender`, are fixed, not being affected by any operation. Of special interest are the *user-controlled* facts which, although also immune to operations, can be manipulated through arbitrary *directives* (cf. section 3.4). In our example, `loves` is user-controlled.

Again for the present example, we have provided operations at two levels. The four main events are performed by level-1 operations: `abduct`, `elope`, `rescue` and `capture`. Operations at level-2 are actions of smaller granularity, in terms of which the level-1 operations can be detailed: `ride`, `entreat`, `seize`, `defeat`, and `carry`.

Our provisional version of the *behavioural schema* consists of goal-inference (a.k.a. situation-objective) rules, belief rules, and emotional condition rules.

For the example, three goal-inference rules are supplied. The first one refers to the ravisher. In words, in a situation where the princess is not at her home and the hero is not in her company – and hence she is unprotected – the ravisher will want to do whatever is adequate to bring her to his home. The other goal-inference rules refer to the hero, in two different situations having in common the fact that the ravisher has the woman in his home: either the hero believes that she does not love the other man, or he believes that she does. In both situations, he will want to bring her back, freely in the first case and constrained in the second.

Informally speaking, beliefs correspond to the partial view, not necessarily correct, that a character currently forms about the factual context (for a formal characterization, cf. the BDI model [Cohen & Levesque; Rao & Georgeff]). The belief rules that we formulated for our example look rational, but notice that they are treated as

[1] ftp://ftp.inf.puc-rio.br/pub/docs/techreports/08_30_barbosa.pdf

defaults, which can be overruled as will be described in section 3.4. A man (the hero or the ravisher) believes that the woman does *not* love his rival if the latter has her confined, but if she has ever been observed in his company and in no occasion (state) was physically constrained, the conclusion will be that she is consenting (an attitude seemingly too subjective to be ascertained directly in a real context).

The emotional condition rules refer to the three characters. A man (or woman) is happy if currently in the company of his (or her) beloved, and bored otherwise. A special condition applies to the woman: she will be *absolutely* happy if, in addition to the first motive for contentment, she has never been constrained by any of the two adversaries.

3.2 Coherent sequences

Moving along the *syntagmatic* axis is primarily the task of the plan-generator, as it composes a coherent plot by aligning events in view of the pre- and post-conditions of the appropriate predefined operations.

For plot composition, it is convenient to proceed in a step-wise fashion, starting from a given initial state. At each state, the goal-inference rules are used to induce opportunistic short term goals from which successive plot sequences will originate.

In an interactive environment, at any step, the user, henceforward called the *Author*, should be allowed to intervene, reducing thereby the characters' autonomy, but relying on the plan-generator to enforce consistency within the genre. To this purpose, the Author may indicate a goal, to be tried by the plan-generator, or even a specific operation, which the plan-generator may or may not find applicable.

A more complex request is to indicate a sparse list of operations, to be filled-up until a valid plot sequence containing all operations in the list, possibly interspersed with others, is formed. The Author may optionally also indicate the desired goal, which would otherwise be assumed to coincide with the effects of the last operation in the list.

After the step-wise process terminates, it should still be possible to perform various kinds of adaptation. Those that have to do with the syntagmatic relations include adding or deleting operations and changing the sequence, if the partial order requirements imposed by the interplay of pre- and post-conditions permit. For instance, consider plot P below:

P = start => ride(Ravana, Lanka, forest) => entreat(Ravana, Sita) => seize(Ravana, Sita) => carry(Ravana,Sita,Lanka)

which can be re-ordered, to meet the Author's preferences, to produce:

Ps = start => ride(Ravana, Lanka, forest) => entreat(Ravana, Sita) => carry(Ravana,Sita,Lanka) => seize(Ravana, Sita)

Curiously, both the original plan P and the reordered plan Ps suggest stories that may well happen in reality or fiction. In P, a voluntary elopement is disguised as an abduction, whereas in Ps elopement is cruelly followed by the woman's confinement.

Also, a plot can be extended with more operations if the Author supplies an additional goal in an attempt to provide a continuation.

3.3 Alternative choices

Moving along the *paradigmatic* axis gives ampler opportunity to obtain different plots than simply changing the sequence of events within the partial order requirements.

Alternatives may result, first of all, from starting from a different initial state, so that different goal-inference rules may be triggered. Notice also that more than one such rule may be ready for activation. In any case, the standard plan-generator's ability to *backtrack* is an expedient mechanism to engender alternative plots.

To resort to violence, as in abduction or capture, can be certainly regarded as excessive and unnecessary when the patient of the action loves the agent, even though our specification does not invalidate their occurrence. Accordingly, if the goal-inference rules are in control and the context is not tampered with (but see section 3.4), they will not figure in any generated plot. And yet the Author can have them as valid alternatives, simply by using the option to directly indicate a goal to the plan-generator. Such goal can be relatively non-specific, such as current_place(Sita,palace), or else more restrictive, such as (current_place(Sita,palace), held_by(Sita,Rama)) – in which case only the forceful capture event will result.

At the adaptation phase, the ability to replace one or more operations is a way to produce alternatives. One must bear in mind that a replacement may require another, if the Author is concerned with preserving consistency; so, replacing `abduct` by `elope` normally implies the replacement of `rescue` by `capture`.

A particularly convenient way to deal with entire plots, rather than with individual operations, is to take advantage of the similarity or analogy among situations, inherent in the notion of paradigms. Previously existing plots, no matter if composed manually or automatically, can be converted into *plot patterns* to be kept in a Library of Typical Plots [Furtado & Ciarlini 2001]. Plot patterns can then be *reused* to originate new plots, essentially by instantiating their variables in view of a new situation.

3.4 Shifts along the way

Until this point we restricted ourselves to planned and hence well-behaved plots. It is time now to introduce a measure of transgression, disrupting the context in order to obtain plots with events in *antithetic* relation.

The Author, as *deus ex machina*, can interfere with the plan generation discipline by issuing two kinds of *directives*, which can be applied both during composition and adaptation. One directive is `make_believe`, arbitrarily assigning a belief B to a character C, which overrules any previous belief on the same facts, either specified through the belief rules of the behavioural schema or stated by a previous application of the `make_believe` directive itself. If Sita was violently abducted by Ravana, Rama will believe (as a consequence of a belief rule) that she does not love the villain, and therefore that she will gladly consent to be rescued. However, the Author is allowed to induce Rama to falsely believe the contrary, which activates a goal-inference rule leading to a forceful `capture` event.

Another directive is `vary`, which manipulates user-controlled facts, instead of mere beliefs. In our example, the only facts declared to be user-controlled are the instances of the `loves` relationship, whose Boolean value will be inverted if the directive is applied. Sita can be eloped if she currently loves Ravana, and then be willingly rescued by Rama if between these two events the Author issues the directive so as to change her feelings. But `vary` does not have to be explicitly called for. A helpful feature in the course of plan-generation can detect

failures involving user-controlled facts, in which case the Author is asked whether or not the context should be tampered with accordingly.

In other example mini-worlds, one might have different kinds of user-controlled properties, e.g. with numerical values inside a range, such as degree of strength, which the `vary` directive could change in some radical proportion. Such representation is also appropriate for emotions in general, including love itself, enabling finely graded nuances of expression, obviously unattainable with simple two-valued Boolean alternatives.

We began to investigate another line, in an attempt to offer clues to an Author intent on finding ways to, at a later stage, replace the external *deus ex machina* directives by some internal narrative device with a flavour of irony, almost crossing the borderline of plausibility. Folktales, myths, and popular culture have pooled together through time rich repertoires of *motifs* [Aarne & Thompson], often containing ingenious solutions to dilemmas arising from antithetic situations.

Authors have always felt free to borrow from all kinds of sources, and one can easily discover occurrences of certain motifs in the literature of different countries, modified as required by cultural differences. For our example, we found three convenient motifs:

a. life token: an object whose aspect changes if the owner is in distress,
b. love potion: stimulates romantic/ erotic feelings,
c. ordeal: to vindicate a discredited or accused person.

where (a) (indexed as E761 in [Aarne & Thompson]) allows to do without the unrealistic assumption that characters are omniscient, e.g. explaining how Rama learned that Sita suffered abduction in the forest, (b) provides an excuse for sudden variations in amorous attachments, and (c) serves to restore the man's belief in his beloved's faithfulness. Curiously, both (b) and (c) occur in the *Tristan* romance, wherein the ordeal takes the especially ironic form of an ambiguous oath [Béroul], while in the *Ramayana* Sita has to walk through the fire [Valmiki]. In our example, we treat these motifs as black boxes, merely associating to their names a <*situation*, *goal*> specification. Thus, if the Author wants to insert motifs (simply through the mention of their names) at the positions in a generated plot where the respective situation holds, this can be asked for at the adaptation phase.

Such insertions are therefore to be regarded as provisional annotations only, which the Author should later have to unravel by mapping the events in the motifs into analogous events congenial to the genre adopted in the plot. The mappings should preserve the <situation, goal> of the motif and might require the definition of additional operations, such as communicative acts for instance. The persistence of motifs is a remarkable phenomenon, with relatively modern versions: microchip implants for (a), aphrodisiac drugs like the LSD hallucinogen for (b), and lie detectors and truth serums for (c), all of so dubious or controversial value as their primitive counterparts, but equally acceptable to the general public.

3.5 Down to details

As stated before, between level-1 and level-2 operations there may be *meronymic* relations. Creating plots in hierarchic fashion is a most common practice, starting with a broad view of the events, which in the case of our example corresponds to the level-1 operators. At later stages, one would gradually decompose each event into finer grain actions, possibly along more than just two levels, to the point of coordinated physical movements, as required for displaying animated scenes [Ciarlini et al.].

When composing a plot, the plan-generator is free to mix operations of the two levels, a reasonable default option considering that the Author may wish to treat some events more succinctly than others. But the Author may, on the contrary, settle for a uniform style by indicating that only one of the two levels will be used. This choice can be altered at any time, in composition or adaptation.

Once a plot is composed, it can be adapted either by detailing or summarizing its constituent operations. Detailing each level-1 operation `Op` in a plot into level-2 operations is treated as one more plan generation task, taking as *situation* the instantiated pre-conditions of `Op`, and as *goal* the effects of `Op`, and using exclusively the operations in the level-2 repertoire. More than one decomposition may be possible, depending on the initial state and on the changes effected by the preceding operations.

The inverse of detailing, summarizing, is also useful. We are currently restricted to a rather limited version, which only works if the detailed plan is divisible into subsequences that can be exactly subsumed by level-1 operations. This means that the process fails if other extraneous operations intervene. In other words, `summarize(P1,P2)` succeeds if and only if `detail(P2,P1)` also does.

Figures 2(a) and 2(b) are suggestive in that they illustrate a curious symmetry in how they map the example level-1 operations into level-2 operations. The decompositions in the two figures are the same, except for the substitution of `entreat` for `seize`. This is not surprising, since a similar decomposition comes as a consequence of the paradigmatic relation between the two villainy and the two retaliation events. Notice too that, in both figures, the event corresponding to villainy only differs from the retaliation event by the possible presence of `defeat` – reflecting our observation, after surveying a number of traditional narratives, that the villain almost always resorts to some trick, avoiding a confrontation that often (though not necessarily) occurs as part of retaliation.

The decompositions suggested by the two figures are typical but not unique, since the correspondence induced by the meronymic relations is not rigidly determined, i.e. it is, so to speak, context-sensitive, depending on the current state. For instance, `abduct` can be expressed by `seize` followed by `carry` if both the victim and the ravisher are currently at the same place, but will need a preliminary `ride` if the former is in the forest and the latter still in his home.

All this suggests that it may be difficult to interpret what is happening by looking at a sequence of level-2 operations without examining the context. In this regard, the ability to fill-up (cf. section 3.2) a sparse list of observed level-2 operations and then performing summarization, identifying what level-1 operation is taking place at some point, constitutes a not so trivial form of *plan-recognition* [Kautz]. Plan generation is more directly relevant to the composition and adaptation of plots than the recognition of plans and objectives. But the latter task is an asset in interactive plan-supported game-playing environments, since each player might employ it as an aid to discover what the opponents are trying to do.

4. A Prototype Implementation

A very simple prototype, **PlotBoard**, was designed to experiment with the notions discussed here. Dealing with *storyboarding* [Truong et al.] – exclusively at the fabula level – it serves to compose plots interactively with the help of an extended version of the early *Warplan* algorithm [Warren]. Written in SWI-Prolog[2], it interfaces with Java to show events in image format.

4.1 Some Features of the Plan-generator

The plan generator follows a backward chaining strategy. For a fact `F` (or `not F`) that is part of a given goal, it checks whether it is already true (or false) at the current state. If it is not, it looks for an operation `Op` declared to add (or delete) the fact as part of its effects. Having found such operation, it then checks whether the pre-condition `Pr` of `Op` currently holds – if not, it tries, recursively, to satisfy `Pr`. Moreover, the plan generator must consider the so-called frame problem [Lloyd], by establishing (in second-order logic notation) that the facts holding just before `Op` is executed stay valid unless explicitly declared to be altered as part of the effects of `Op`.

Like goals, pre-conditions are denoted by conjunctions of literals and arbitrary logical expressions. We distinguish, and treat differently, three cases for the involved positive or negative facts:

a. facts which, in case of failure, should be treated as goals to be tried recursively by the plan generator;
b. facts to be tested immediately before the execution of the operation, but which will not be treated as goals in case of failure: if they fail the operation simply cannot be applied;
c. facts that are not declared as added or deleted by any of the predefined operations.

Note that the general format of a pre-condition clause is `precond(Op, Pr) :- B`. In cases (a) and (b), a fact `F` (or `not F`) must figure in `Pr`, with the distinction that the barred notation `/F` (or `/(not F)`) will be used in case (b). Case (c) is handled in a particularly efficient way. Since it refers to facts that are invariant with respect to the operations, such facts are included in the body `B` of the clause, being simply tested against the current state when the clause is selected.

An example is the precondition clause of operation `seize(M,W)`, where `M` is the agent and `W` the patient of the action. Clearly the two characters should be together at the same place, and, accordingly, the `Pr` argument shows two terms containing the same variable `P` to express this requirement, but the term for `W` is barred: `/current_place(W, P)`, which does not happen in `M`'s case. The difference has an intuitive justification: the prospective agent has to go to the place where the patient is, but the latter will just happen to be there for some other reason.

The proper treatment of (a) and (b) is somewhat tricky. Suppose the pre-condition `Pr` of operation `Op` is tested at a state S_1. If it fails, the terms belonging to case (a) will cause a recursive call whereby one or more additional operations will be inserted so as to move from S_1 to a state S_2 where `Op` itself can be included. It is only at S_2, not at S_1, that the barred terms in case (b) ought to be tested, and so the test must be *delayed* until the return from the recursive call, when the plan sequence reaching S_2 will be fully instantiated.

Operations can admit more than one precondition clause, so as to cope with different circumstances. This happens with the `carry(M,W,P2)` operation, whereby `W` will either freely consent to be transported to `P2` by `M`, or will have to be forcefully held by him.

With respect to the added and deleted clauses declaring effects of operations, the plan generator also employs a barred notation, to distinguish between two cases: (a) primary effects, and (b) secondary unessential effects. In case (a), if any fact `F` to be added by `Op` already holds, or already does not hold if it should be deleted, then `Op` is considered *non-productive* and fails to be included in the plan. In contrast, in case (b), such lack of effect would be admitted and cause no failure.

As an example, consider the clause of operation `capture(M1,W)` that declares as deleted the fact `held_by(W,M2)`, as a result of `M1`'s action to take away `W` from `M2`. Notice that the fact may or may not hold prior to capture; it will hold if `W` was abducted by `M2`, but will not hold if an elopement occurred instead – and that is why the barred notation is used for this particular deleted clause. On the contrary, the fact `current_place(W,P2)`, where `P2` is the home of `M2`, must necessarily be deleted by an effective execution of the operation, and so does not figure as barred.

The execution of plans is done through `assert` or `retract` commands on the facts to be, respectively, added or deleted. The plan's pre- and post-conditions are checked during the process, there being no effect in case of failure. A `log(L)` literal, initiated with `L=start`, is extended with each successful plan execution and can be usefully retrieved for a variety of purposes. On the basis of the log and of the initial state, which is saved when a session begins, it is possible to query about facts at any intermediate state. It is also possible to save and restore any previous state S (initial or intermediate), which enables simulation runs.

User interventions, necessary to achieve unplanned situations, are permitted in a limited scale through *directives* that can be either intermixed with the operations in a plan or called separately. Two of these are used in our example, one for changing `loves` facts, immune to the predefined operations, and the characters' beliefs, which may not correspond to actual facts.

To finish this partial review of the plan features, we remark that the planning algorithm `plans(G,P)` is called in more than one way. More frequently `G` is given, as the goal, and `P` is a variable to which a generated plan will be assigned as output. However an inverse usage has been provided, wherein `P` is given and `G` is a variable; in this case, the algorithm will check whether `P` is valid and, if so, assign its net effects (a conjunction of `F` and `not F` terms) to `G`.

4.2 The PlotBoard Tool

We shall briefly describe how **PlotBoard** works, after the controlling user, here called the Author, enters the `plot` command. The diagram of figure 3 will serve to guide the description.

The main option is to compose the plot from scratch, in a step-wise fashion. Ideally, the Author should leave a measure of autonomy to the characters (branching into the

[2] http://www.swi-prolog.org/

planner node of the diagram). At each step (cf. the plan step node), one subsequence of the plot will be generated. As if emerging from the mind of a character C, a short-term opportunistic goal G is instantiated by some goal-inference rule (C,S,G), if the situation S of the rule currently holds. More than one rule may be simultaneously ready for activation, and the planning algorithm may find more than one alternative subsequence able to achieve the corresponding goals (whenever the planning algorithm backtracks), as indicates the self-loop around the plan step node. While a subsequence is presented, the Author is prompted to either issue an ok reply or call for an alternative, possibly after inspecting what effects it would have. An ok reply is followed by a return to the planner node.

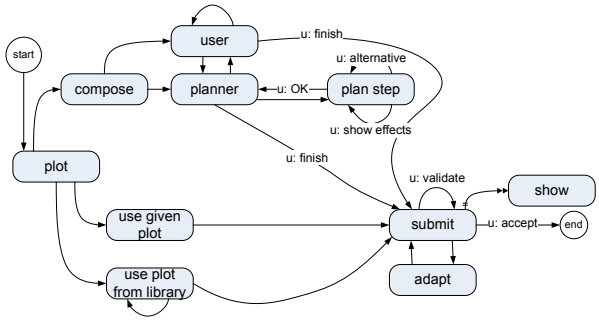


Figure 3: Flow of control of the **PlotBoard** prototype

The subsequence thus selected is then executed in a simulated mode, and the Author is asked whether the plan step iterations should continue, producing further subsequences to be appended to the plot so far obtained, or whether the composition process is finished for the time being (passing to the submit node), though still subject to possible adjustments.

If the Author is more inclined towards a closer arbitrary control than to the character autonomy policy described above, several options are available to determine the goals that the planning algorithm should try to achieve (cf. the the first 3 items of the menu for the user node below). Again, the self-loops around the user node represent the possibility of alternative plot subsequences being offered to the Author's choice. These options permit step-wise composition, which can be entirely commanded from the user node, but can also alternate with the activation of goal-inference rules, by intercalating transfers of control to the planner node.

An additional purpose of the user node is to prepare and support the composition process, by allowing to pose queries about the database state at each step, to change the operation level, and to issue directives to alter the characters' beliefs and the value of user-controlled properties.

```
1: goal
2: operation
3: list of operations
4: query
5: operation level
6: directive
7: planner
8: finish
```

Whatever composition policy is preferred – autonomous, arbitrary, or mixed – the finished plot is passed to the submit node. At this point, the Author can either accept the plot, which terminates the process, or can go through one or more rounds of adaptation, using the options offered at the adapt node below.

```
1: detail
2: summarize
3: change sequence
4: add operation
5: delete operation
6: replace operation
7: extend
8: queries or directives
9: insert motif
10: back to the submit options
11: stop
```

To help decide whether to accept the current plot or perform other adaptations, the submit menu allows to validate the plot (again through the planning algorithm). This may be in order if the Author directly introduces specific changes (items 4-6 of the adapt menu), noting that in all other forms of adaptation the planning algorithm intervenes to prevent integrity violations.

Another feature available at the submit node deserves attention, since what it produces, together with the menu-based dialogues, constitutes the intended output of **PlotBoard**. If selected, via the show option, it provides a visual display that can be repeated for the successive versions. For each operation in the current plot, the event it denotes appears as a rough drawing, side by side with a short template-driven natural language sentence.

We refer again to the diagram in figure 3, to consider two ways to obtain a plot without requiring step-wise composition from scratch. In both cases, a full plot is used to start with, and in both cases the process converges afterwards to the submit node.

Branching into use given plot, the Author can either enter the intended plot or retrieve a previously composed one. The planning algorithm is automatically called to inspect the plot, operation by operation, to check whether each of them can be applied in view of the pre- and post-conditions interplay. If an operation is found that can only be applied if a user-controlled property is tampered with, the possibility of changing the value of the property is indicated to the Author, who may or may not permit the execution of the necessary vary directive. If the Author denies permission, or if the offending property is not user-controlled, the plot is rejected.

In case the node use plot from library is chosen, the Library of Typical Plots (LTP) will be searched for items (S,G,P), such that situation S currently holds, thereby propagating the instantiation of the parameter variables figuring in S to goal G and plot P. If more than one such item is found, the Author will have once more an opportunity to select the preferred P among the alternatives presented.

4.3 An example run

At the initial state, both Rama and Ravana are in their homes, respectively the royal palace and the city of Lanka, whereas Sita is alone in the forest. The two men love Sita, who only loves Rama. Starting to compose the plot, the Author invokes the planner in two stages, always selecting the detailed (level 2) alternatives. At this point the plot is, in natural language format:

> Ravana rides from Lanka to forest. Ravana seizes Sita. Ravana carries Sita to Lanka. Rama rides from palace to Lanka. Rama defeats Ravana. Rama entreats Sita. Rama carries Sita to palace.

Wishing to try different versions, the Author looks at the adapt menu, shown in the previous section. The first change selected is the deletion of the two events that close the narrative. The next step is to issue directives to change

the emotional attachments and certain of the characters' beliefs: now Sita loves Ravana and Rama believes this fact. This justifies adding `entreat(Ravana,Sita)` as second event (after Ravana approaches the princess):

> Ravana rides from Lanka to forest. Ravana entreats Sita. Ravana seizes Sita. Ravana carries Sita to Lanka. Rama rides from palace to Lanka. Rama defeats Ravana.

The plot now suggests the fake abduction pattern, wherein the villain seizes his pretended victim only to simulate a violent action. The Author wonders then if the same events could be arranged in some different sequence, and a dialogue ensues:

> [f1:entreat(Ravana, Sita), f2:seize(Ravana, Sita)]
> choose one of the fi tags: f1
> [f1:seize(Ravana, Sita), f2:carry(Ravana, Sita, Lanka)]
> choose one of the fi tags: f2
> [f1:seize(Ravana, Sita), f2:ride(Rama, palace, Lanka)]
> choose one of the fi tags: f1
> Ravana rides from Lanka to forest. Ravana entreats Sita. Ravana carries Sita to Lanka. Ravana seizes Sita. Rama rides from palace to Lanka. Rama defeats Ravana.

This sounds as overt elopement after which the seducer restricts the woman's freedom. What can happen next?

Selecting the extend option of the `adapt` menu, the Author proposes: `current_place(Sita,palace)` as a goal, and the planner responds (figure 4) with: Rama captures Sita. Is this a satisfactory way to end the narrative? The Author selects option 8 and poses queries, to learn what the characters think and how they feel:

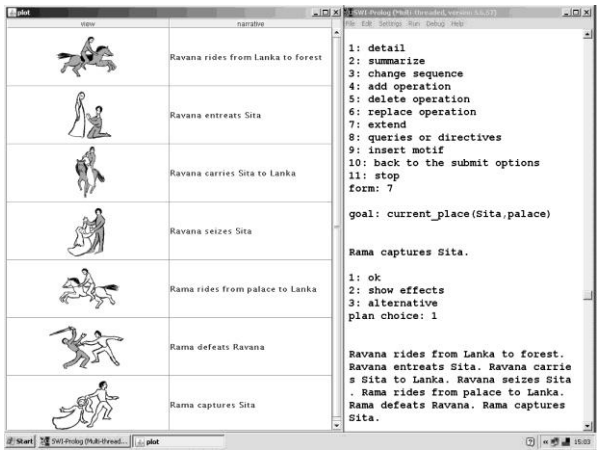


Figure 4: A **PlotBoard** screen.

> query: beliefs
> Rama believes that Sita loves Ravana
> Ravana does not believe that Sita loves Rama
> more queries?(yes/no): yes
> query: emotional_condition
> Sita is bored. Rama is happy. Ravana is bored.

Sympathizing with the princess, the Author decides to revert the situation. Perhaps her love for the hero could revive (as happened with Helen of Troy), and the last event is replaced according to this expectation: `capture(Rama,Sita)` turns into `rescue(Rama, Sita)`. How does it look now? Back at the `submit` menu, the Author asks to visualize the scenes and accepts this result, a happy end for Sita as well as for the Author, who receives a grateful acknowledgement from the **PlotBoard** tool (Figure 5).

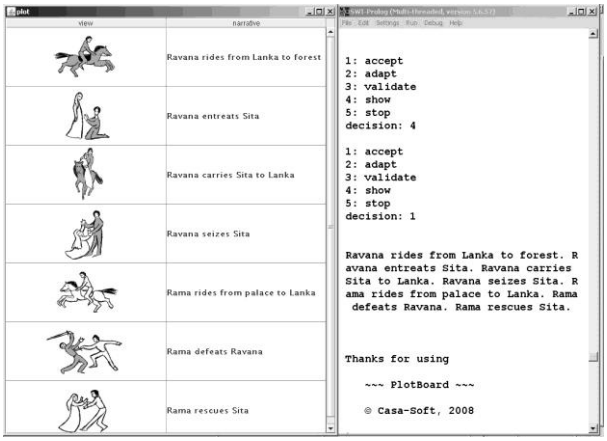


Figure 5: The accepted plot.

But much remains to be done. The *deus ex machina* directives should be replaced eventually by something internal to the narrative. Also, how to explain that Rama knew without being told that Sita had become Ravana's prisoner? To gather suggestions, to be possibly (re)used after due modifications appropriate to the genre, the Author might have inspected (figure 6) the applicable motifs, before issuing the final accept response, in which case the *life token*, the *love potion* (twice) and the *ordeal* motifs would be indicated at one or more positions in the plot wherein the respective motivating situation holds.

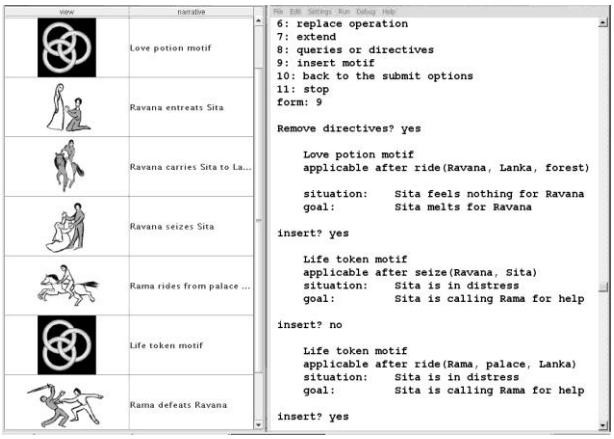


Figure 6: Insertion of motifs (partial view).

5. Concluding Remarks

Although the process of plot composition could surely be enriched far beyond what was presented here, the suggested fourfold approach seems to provide a sound initial basis. The conjecture that the interplay of the syntagmatic, paradigmatic, antithetic and meronymic relations already permits an ample coverage is reinforced by the connection between these relations and the four major tropes. Other concepts may be adduced to extend the model. If we see a disruption not as a discontinuity in one context, but as an attempt to put together two originally incompatible contexts, the notion of *blending* [Fauconnier & Turner; Casanova et al.] comes to mind, as the technique or artisanship of conciliating the pending conflicts, often requiring a great deal of creativity.

The facilities associated with the four relations are adequate for other tasks, besides storyboarding, under suitable user interfaces. In interactive storytelling systems designed for entertainment, as well as in games, they might prove instrumental to support the production of

coherent stories with an ability to cause surprise. Further research might investigate ways to adjust the generation of alternatives to users' satisfaction models, so that there would be no longer a need to explicitly interfere to obtain varied and interesting outcomes.

Finally, let us recall that we have addressed the *fabula* level only, where one simply indicates *which* events should be included in the plots. A complex problem to be faced at the next level – the *story* level, where the concern is *how to tell* the events – is to find a plausible justification for the contextual disruptions introduced *ex machina* via user interaction. As said, such elaborations may be suggested by some fanciful motif annotated in the plot. Moreover a plurality of narrative objectives must be satisfied [Crawford; Turner; Montfort].

At the third and last level – the *text* level – the narrative is *represented* in some medium, not necessarily printed pages. Natural language text-generation from plots of log-registered business transactions is covered in [Furtado & Ciarlini 2000]. In the realm of literary genres, an ongoing project applies computer graphic animation to display narrative plots [Ciarlini et al.; Camanho et al.].

References

AARNE, A. AND THOMPSON, S. 1987. *The Types of the Folktale*. Suomalainen Tiedeakatemia.

ARISTOTLE. "Poetics". 2000. In *Classical Literary Criticism*. Penelope Murray et al. (trans.). Penguin.

BAL., M. 2002. *Narratology*. U. of Toronto Press.

BARROS, L. AND MUSSE, S. 2007. "Planning algorithms for interactive storytelling".In *ACM Computers in Entertainment*, ACM 5.1.

BATINI, C., CERI, S. AND NAVATHE, S. 1992. Conceptual Design – an Entity-Relationship Approach. Benjamin Cummings.

BÉROUL. 1970. *The Romance of Tristan*. A.S. Fedrick (trans.). Penguin.

BREITMAN, K.K., BARBOSA, S.D.J., CASANOVA, M.A. AND FURTADO, A.L. 2007. "Using analogy to promote conceptual modeling reuse". In Proc. of *Workshop on Leveraging Applications of Formal Methods, Verification and Validation* .

BREITMAN, K.K., CASANOVA, M.A. AND TRUSZKOWSKI, W. 2007. *Semantic Web*. Springer.

BOOTH, W. 1974. *A rhetoric of Tropes*. U. of Chicago Press.

BURKE, K. 1969. *A Grammar of Motives*. U. of California Press.

CAMANHO, M., CIARLINI, A.E.M., FURTADO, A.L., POZZER, C.T. FEIJÓ, B. 2008. "Conciliating coherence and high responsiveness in interactive storytelling". In *Proc. 3rd. ACM International Conference on Digital Interactive Media in Entertainment and Arts (DIMEA 2008)*.

CASANOVA, M.A., BARBOSA, S.D.J., BREITMAN, K.K. AND FURTADO, A.L. 2008. "Generalization and blending in the generation of entity-relationship schemas by analogy". In Proc. of the *Tenth International Conference on Enterprise Information Systems* (ICEIS).

CAVAZZA, M., CHARLES, F. AND MEAD, S. 2002. "Character-based interactive storytelling". *IEEE Intelligent Systems*, special issue on AI in Interactive Entertainment, 17(4).

CHANDLER, D. 2007. *Semiotics: The Basics*. Routledge.

CHRÉTIEN DE TROYES. 1983. *Le Chevalier de la Charrete*. M. Rocques (ed.). Honoré Champion.

CIARLINI, A.E.M., POZZER, C.T., FURTADO, A.L. AND FEIJÓ, B., 2005. "A logic-based tool for interactive generation and dramatization of stories". In *Proc. of Advances in Computer Entertainment Technology*.

COHEN, P.R. AND LEVESQUE, H.J., 1990. "Intention is Choice with Commitment". *Artificial Intelligence* 42.

COSTIKYAN, G. 2002. "I have no words and I must design: Toward a Critical Vocabulary for Games". In *Proc. of Computer Games and Digital Cultures*.

CRAWFORD, C., 1984. *The Art of Computer Game Design*. Osborne-McGraw-Hill.

CULLER, J. 1983. *On Deconstruction: Theory and Criticism after Structuralism*. Cornell U. Press.

FAUCONNIER, G. AND TURNER, M. 2002. *The Way We Think*. Basic Books.

FIKES, R.E. AND NILSSON, N.J. 1971. "STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence* 2.

FURTADO, A.L., CASANOVA, M.A., BARBOSA, S.D.J. AND BREITMAN, K.K. 2008. "Analysis and Reuse of Plots using Similarity and Analogy". In *Proc. of 27th International Conference on Conceptual Modeling (ER)*.

FURTADO, A.L. and CIARLINI, A.E.M. 2000. "Generating Narratives from Plots using Schema Information". In *Proc. of 5th International Conference on Applications of Natural Language to Information Systems*.

FURTADO, A.L. AND CIARLINI, A.E.M. 2001. "Constructing Libraries of Typical Plans". In Proc. of *13th International Conference on Advanced Information Systems Engineering*.

FURTADO, A.L. AND VELOSO, P.A.S. 1996. "Folklore and Myth in The Knight of the Cart". In *Arthuriana*, vol 6, 2.

GRASBON, D. AND BRAUN, N. "A morphological approach to interactive storytelling". 2001. *In Proc. CAST01, Living in Mixed Realities*. Special issue of Netzspannung.org/ journal, the Magazine for Media Production and Inter-media Research.

HEARST, P.C. AND MOSCOW, A. 1988. *Patty Hearst: her own Story*. Avon.

HOMER. *The Iliad*. 1950. E.V. Rieu (trans.). Penguin.

KAUTZ, H. A. 1991. "A Formal Theory of Plan Recognition and its Implementation". In *Reasoning about Plans*. J. F. Allen et al. (eds.). Morgan-Kaufmann.

LAKOFF, G. AND JOHNSON, M. 1980. *Metaphors We Live By*. U. of Chicago Press.

LLOYD, W. 1987. *Foundations of Logic Programming*. Springer.

MARCHELLO-NIZIA, C. (org.). 1995. *Tristan et Yseut*. Gallimard.

MATEAS, M., AND STERN, A. 2000. "Towards integrating plot and character for interactive drama". In *Socially Intelligent Agents: the Human in the Loop*, AAAI Fall Symposium.

MCGARRY, M. (ed.). 1979. "The Story of Deirdre". In *Great Folk Tales of Ireland*. Frederick Muller.

MONTFORT, N. 2006. "Natural Language Generation and Narrative Variation in Interactive Fiction". In *AAAI Workshop* on *Computational Aesthetics*.

ORTONY, A. (ed.). 1996. *Metaphor and Thought*. Cambridge: Cambridge U. Press.

OVID, 1986. *Heroides and Amores*. G. Showerman (trans.). Harvard U. Press.

PEARCE, C. (2002) "Emergent authorship: the next interactive revolution". In *Computers & Graphics* 26.

PROPP, V. 1968. *Morphology of the Folktale*. S. Laurence (trans.). U. of Texas Press .

RAO, A.S. AND GEORGEFF, M.P. 1991. "Modeling rational agents within a BDI-architecture". In Proc. of *Int'l Conf. on Principles of Knowledge Representation and Reasoning*.

RIEDL, M. and YOUNG, R. M. 2004. "An intent-driven planner for multi-agent story generation". In *Proc. of 3rd Int'l. Conf. on Autonomous Agents and Multi Agent Systems*.

SAUSSURE, F. 2006. *Cours de Linguistique Générale*. C. Bally, A. and A. Riedlinger (eds.). Payot.

TITUS LIVIUS. 1919. *History of Rome*, vol I. B. O. Fster (trans.). Harvard U. Press. Book I, Chapter I: 13.

TRUONG, K. N. HAYES, G. R. AND ABOWD, G. D. 2006. "Storyboarding: An Empirical Determination of Best Practices and Effective Guidelines". In *Proc. of 6th conference on Designing Interactive systems*.

TURNER, S.R. 1992. *Minstrel: A computer model of creativity and storytelling*. T. R. UCLA-AI-92-04, Computer Science Dept.

VALMIKI. 1999. *Le Ramayana*. Philippe Benoît et al. (trans.). Gallimard,.

VIRGIL. 1994. *Eclogues, Georgics, Aeneid*. H.R. Fairclough (trans.). Harvard U. Press.

WALLIS, J. 2008. *Making Games that Make Stories*. Electronic book review [www.electronicbookreview.com].

WARDRIP-FRUIN, N., HARRIGAN, P. (eds.). 2004. *First Person: New Media as Story, Performance, and Game*. The MIT Press.

WARREN, D.H.D. 1974. *WARPLAN: a System for Generating Plans*. Edinburgh: University of Edinburgh, Department of Computational Logic, memo 76.

WHITE, H. 1973. *Metahistory: The Historical Imagination in Nineteenth-Century Europe*. The Johns Hopkins U. Press.

Improving Boids Algorithm in GPU using Estimated Self Occlusion

Alessandro Ribeiro da Silva[1]
Universidade Federal de Minas Gerais

Wallace Santos Lages[2]
Universidade Federal de Minas Gerais

Luiz Chaimowicz[3]
Universidade Federal de Minas Gerais

Abstract

Behavioral models are used in games and computer graphics for realistic simulation of massive crowds. In this paper, we present a GPU based implementation of Reynolds [1987] algorithm for simulating flocks of birds and propose an extension to consider environment self occlusion. We performed several experiments and the results showed that the proposed approach runs up to three times faster than the original algorithm when simulating high density crowds, without compromising significantly the original crowd behavior.

Keywords:: Boid simulation, GPGPU

Author's Contact:

[1]alessandrosilva@ufmg.br, [1]www.alessandrosilva.com
[2]wlages@ufmg.br
[3]chaimo@dcc.ufmg.br

1 Introduction

The simulation of a large number of individuals has applications in many different games, whether to compose the background scene in games (GTA, Rockstar Games, 1997) or as part of the gameplay itself (Pikimin, Nintendo, 2001). As members of the crowd meet each other, they interact by coordinating their motion accordingly to the goal of each individual. As examples we may cite the motion of flocks of birds, banks of fishes, herds of land animals, or even groups of human characters.

The first behavioral models appeared as extensions of particle systems used to model water, fire, grass and atmospheric effects [Reeves 1983]. Other extensions soon followed. In 1987, Reynolds presented a distributed model for controlling flocks of birds that considered interactions between agents [Reynolds 1987]. Although every agent, or *boid*, takes decisions considering only its local perception of the world, the sum of the behaviors enable the flock to present a very real-like motion.

However, to simulate local perception, one must be able to identify neighbors among all existing agents. The naive option (comparing each boid to the other) leads to a $O(n^2)$ behavior that becomes prohibitive for a large number of boids. So, to obtain an interactive system, we must have both fast implementations and low complexity algorithms.

To speedup the process of finding neighbors, researchers have used different spatial structures [Shao and Terzopoulos 2005], [Reynolds 2006]. Instead of searching in the whole population, this enables a local search in a pre-sorted structure, thus lowering the asymptotical complexity. Another approach used is to avoid the computation when neighbors do not change much [Chiara et al. 2004] or even update only a small percentage of the population per frame [Reynolds 2006].

On the other hand, current graphics architectures exhibit a large degree of parallelism which can be used for a highly efficient boid computation and display. GPU implementations are presented by Court and Musse [2005] and Chiara et al. [2004]. A fast implementation on the Playstation3 hardware was presented by Reynolds [2006].

In this work we present an implementation of the model proposed by Reynolds [1987], [1999] for a Geforce 8800 GPU. We also present an extension to estimate self occlusion in the neighbor computation and show how it can be used to improve the simulation performance. Our idea is to estimate the number of boids occluding the view cone of each boid and avoid considering *invisible* boids in the behavior calculation. This technique is orthogonal to the one mentioned above and specially useful for very dense populations.

The remainder of this paper begins with a review of the original boid model and other related work. We then present the graphics hardware mappings and algorithms we used to estimate density and behavior. Finally, we present results and conclusions.

2 Related Work

Agent simulation for large crowds is very computing expensive. Some techniques used to alleviate the problem are: parallelization, use of spatial structures, and heuristics to reduce the update rate of the crowd.

Quinn et al. [2003] presented a parallel pedestrian movement model running over 11 processors Linux-based multicomputer with MPI. They were able to simulate and render the motion of tens of thousands of pedestrians in real time using a manager/worker organization. Other researchers used the powerful parallelism of graphics processors to speedup the processing and display of large crowds. Chiara et al. [2004] present a massive simulation and rendering of a behavioral model using graphics hardware. They rendered a 3D scene with a flock of 8000 animated bird models at 20 fps. They describe the use of vector fields to manage obstacle avoidance and a heuristic that avoids recomputing the behaviors when the list of neighbors does not change. Courty and Musse [2005] used the GPU to compute a physics-based animation model which considers the influence of gaseous phenomena in the behavior of the crowd. This system, called FastCrowd, ran a crowd of 10,000 individuals at 50 fps without visualization and at 35 fps using impostors. The behavior model is very complex and include new psycho-physical forces. In 2006, Reynolds published an implementation for the PLAYSTATION3 hardware [Reynolds 2006]. He was able to concurrently simulate and display simple crowds of 15,000 individuals at 60 frames per second.

Since the number of individuals is large and the global behavior changes slowly, many researchers decoupled simulation update from rendering [Reynolds 2006], [Treuille et al. 2006]. As long as the position is properly updated, errors are very difficult to observe. On Reynolds implementation, [Reynolds 2006] only 1/8 of the individuals are updated at each frame. We preferred not to take this approach. Every simulation is fully computed for every individual on every frame.

Another way to improve speed is to use spatial hierarchies to quickly exclude individuals too far to influence the one being computed. For GPU computation, the most common data structure is the regular grid [Shao and Terzopoulos 2005],[Reynolds 2006]. More sophisticated data structures are more complex to navigate and therefore, slower. Some works do not use spatial structures at all and rely solely on brute force [Drone 2007].

As mentioned before, the main contribution of this work is a GPU implementation of the original algorithm by Reynolds, that considers visibility into the behavior of each boid. The visibility is estimated by computing the density of boids in the field of view.

3 Background Information

Visibility Culling

The goal of visibility culling is to quickly reject parts of the scene that are not visible for a given viewpoint. In computer graphics, occlusion culling is used to avoid processing or drawing such parts

of the scene. In our work visibility is used to avoid computing the influence of occluded boids. From the taxonomy proposed by Cohen-Or et al. [2000] the more relevant classifications for this work are:

- Point vs. Region. Point algorithms performs the computation with respect to the location of the current viewpoint only whereas from-region performs a computation valid for a region of the space. From-region visibility has its cost amortized over time but usually requires a longer processing.
- Image precision vs. Object precision. Object precision methods use the raw objects in their visibility computations. Image precision methods, on the other hand, operate on the discrete representation of the objects after they are rendered into images.
- Conservative vs. approximate. Conservative techniques overestimate the visible set. Approximate techniques may fail to include the entire visible set as a trade off for speed.

The technique employed in this work can be described as an approximate from-point visibility algorithm. In particular, we approximate visibility based on the volumetric representation of the scene. Instead of performing geometric visibility computations, we compute for each voxel the density of boids and approximate the visibility between regions by computing the volume opacity between them. This idea was first proposed by Sillion [1995] in the context of a radiosity system. Volumetric visibility was also independently developed by Klosowski and Silva [2000].

Behavioral Models

In this paper we implement the original model proposed by Reynolds [1987], [1999], where each boid steering behavior is computed independently based on its field of view. The field of view can be described by a maximum viewing distance and a angle of view (Figure 1).

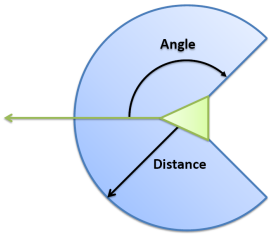


Figure 1: *Agent visibility (Adapted from Reynolds [1987]).*

Based on the visibility of each element, we consider three basic steering behaviors. The first, *separation*, is the behavior that prevents the boids from colliding. The steering force is computed as the average of the difference vector between current boid position and every neighbor. *Alignment* is the behavior that tends to align the boid with the average group direction. The *Cohesion* behavior moves the boid toward the center of his local neighborhood. The steering force is computed as the average position of the neighbors. An intuitive description of the behaviors is shown on Figure 2.

At each simulation step, the steering force is applied to the current position of every boid, and a new position is computed according to the resultant velocities.

4 GPU Mapping and Implementation

The graphics processing unit (GPU) was developed to transform, light and texture map triangles. For this reason, in GPGPU algorithms, data is usually encoded into textures or geometry before being processed by the GPU. The GPU mapping used in this work was based in general GPU programming techniques [Owens et al. 2007]. It uses texture maps to encode vectors of speed, position etc. This same technique has been used in other similar works [Kolb et al. 2004], [Chiara et al. 2004], [Courty and Musse 2005]. The execution flow can be divided in three distinct steps, two of them are performed by the CPU.

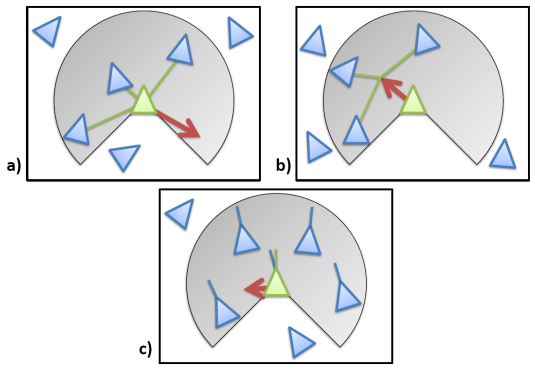


Figure 2: *The steering behaviors: a) Separation; b) Cohesion; c) Alignment (Adapted from Reynolds [1987]).*

Before delving into the algorithm, we shall first explain the data mapping and the data structure used.

4.1 Data Mapping

To describe the state of each boid, we need to store the following data: one translation vector, two orientation vectors (z and y axis) and a 3D force vector. This can be mapped into four RGBA textures. Since the graphics pipeline cannot be used to read and write at the same time, we used two copies of each texture. After each simulation step, input and output textures are switched (ping-pong buffering).

Since the maximum a 1D texture length allowed in our GPU is 8162, we used a function to map 1D address to a 2D address. The mapping function is defined by the Equation 1. Figure 3 shows the same mapping in a graphical form. This mapping was applied to every data addressed by an one dimensional coordinate. It will be referred as a virtual index since it does not represent the real address sent to the graphics API.

$$Tex2D = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1D\ index\ \mathbf{mod}\ TexWidth \\ \lfloor 1D\ index / TexWidth \rfloor \end{bmatrix}. \quad (1)$$

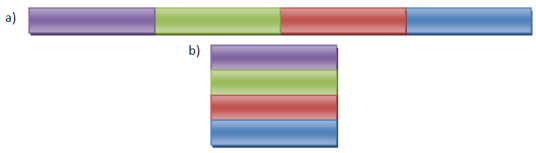


Figure 3: *Mapping of a 1D virtual index (a) to a 2D texture (b).*

4.2 Data Structure

The cost of neighbor search can be accelerated by using spatial indexing structures. We used a uniform grid since it has a constant cost to build and it is easy to evaluate inside the GPU. Other recursive structures, although more efficient, require a costly maintenance cost and are more complex to construct.

We encoded the grid structure as a 3D texture. Each position contains a virtual index to one boid. This index can be used to retrieve information about position, orientation or force in another texture. To be able to store more than one boid per cell, we used a linked list. This was implemented using the fourth coordinate (w) as a index to the next boid in the same cell. A value of -1 means the list has reached the end.

Figure 4 shows the mapping from the world space to the grid space, the mapping of the grid to a virtual index and the list of elements inside the position array.

In the virtual index implementation, all indexes are stored with a $+1$ increment. In this way, it is possible to use a *memset* function

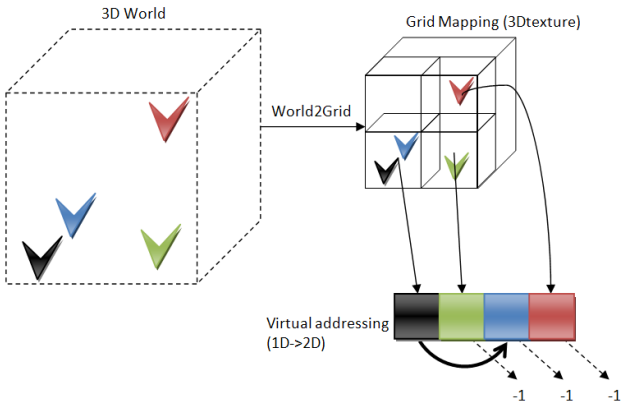


Figure 4: *Mapping of the 3D world to grid space and linked list indexing.*

to clear the grid content to zero. Inside the shader, all access and index conversions must subtract 1 from the elements to be accessed. A value of -1 after this operation means an invalid or null index.

4.3 Algorithm

As mentioned before, the execution flow can be divided in three distinct steps. The first step is the update of the grid structure (1). This is done on the CPU. Following, boids simulation is computed in the GPU (2) and finally they are rendered on the screen (3).

Step One: Grid update The first step objective is to associate each boid to a grid cell. This is necessary since at each step boids move among them. The application downloads the texture containing the position from the GPU and uses the values to update the internal grid indexes and the indexes of the position list. The grid construction is done in $O(m^3 + n)$, where m is the grid dimension and n the number of boids. After the construction, the application uploads the updated textures back to the GPU(Algorithm 1).

Algorithm 1 Grid structure construction algorithm.

```
1:  Pos ← download positions from GPU
2:  Grid ← clear grid content
3:  for i ← 0 to n do
4:      Pos[i].w ← 0
5:      GridIndex ← Compute cell of pos[i]
6:      if Grid[GridIndex]exists then
7:          Pos[i].w ← next address pos + 1
8:      else
9:          Grid[GridIndex]← n + 1
10:     end if
11: end for
12: GPU_Positions ← upload Pos from CPU
13: GPU_Grid ← upload Grid from CPU
```

Step Two: Simulation The simulation step is done entirely inside the GPU, including the search for neighbors and the calculation of the vector for each behavior. The grid is used to estimate the visibility for each boid (Algorithm 2).

Step Three: Rendering To render the position of each boid, we used a static 3D model of a bird without texture. The model has 268 triangles and normals. The geometry was compiled in a *display list* [Opengl et al. 2005]. Since an OpenGL display list is static, we added a parameter to index the information of each boid in the position array. Using this method, it is possible to render all the static models with only one API call, reducing considerably the overhead due to matrix calls.

4.4 Grid cell visibility

The estimative of the visibility uses three levels of tests to avoid unnecessary processing of grid cells and individuals inside them.

Algorithm 2 Simulation algorithm.

```
1:  GridPos ← Calculate the boid grid position
2:  for i ← all the neighbor grid cell do
3:      GridIndex ← Compute cell of pos[i]
4:      if i is visible then
5:          for j ← all the neighbor in grid cell i do
6:              if j is visible then
7:                  Update Cohesion,
                    Alignment and Separation
8:              end if
9:          end for
10:     end if
11: end for
12: Desired force ← force based on vectors
13: lerp ← linear interpolation factor
14: FinalForce ← PreviousForce + (DesiredForce −
    PreviousForce) ∗ lerp
15: FinalTranslation ← Translation + FinalForce
16: Update axis 'y' and 'z'
17: Store FinalTranslation, FinalForce, 'y' and 'z'
```

First level: Maximum grid level

In the first test we select potential grid cells based on the max vision distance parameter. The output is a cube of cells with the local maximum cell count in all grid's direction (x,y,z) according to the Equation 2 (Figure 5a).

$$CellCount = \left\lceil \frac{visionDistance \times (GridSize - 1)}{WorldGridDimention} \right\rceil + 1. \quad (2)$$

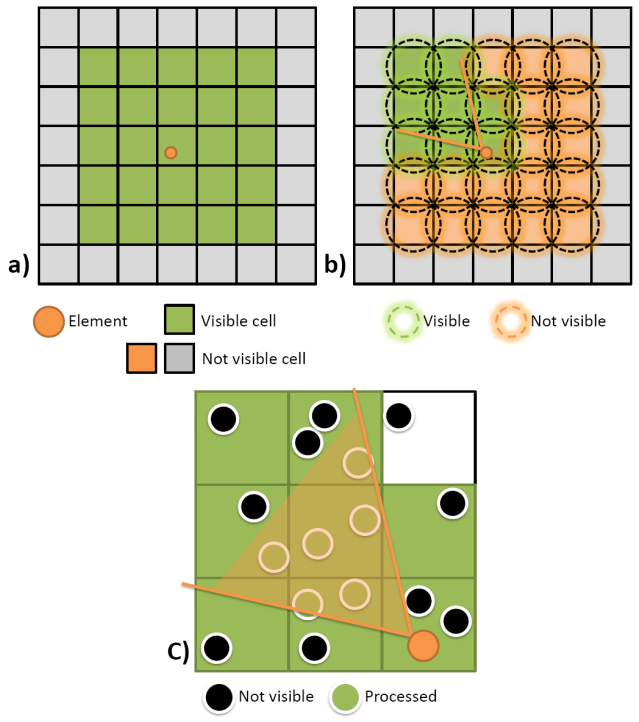


Figure 5: *Visibility tests: a) Maximum grid range; b) Sphere-cone test; c) Element test.*

Second level: Sphere-cone test

This test filters the grid cells that are not visible using sphere-cone collision. First we define spheres for each cell using the grid center as the sphere center and half of cell diagonal as the sphere radius. From the element orientation we construct a inverse rotation matrix that puts each grid cell sphere in the element local space. The cone-sphere test executes as a 2D test using the length of the sphere from local Z axis.

Figure 6 shows the result of each calculation. In a) we have the grid-sphere in grid space, b) shows the sphere after the inverse transformation, c) shows the 2D test space. Notice that the scale is not important to determine the cone visibility test since the ratio between the cone and the sphere will be the same.

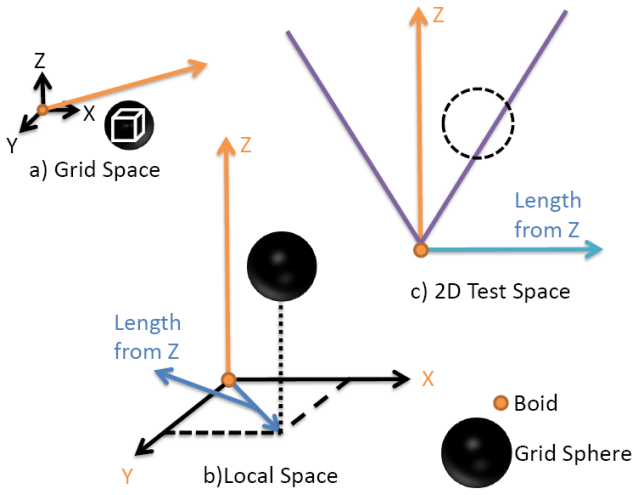


Figure 6: *a) The grid cell sphere in grid space; b) The grid cell sphere in the element local space; c) The 2D space used for the cone test.*

The output of this test is a grid cell filtered accordingly the element view (Figure 5 b).

Third level: Element test

After we know all visible cells, we iterate over all lists inside of them and test each element against the boid field of view. As output we have a list of the boids inside the field of view. This is the list used in the behavior calculation (Figure 5c).

4.5 Estimating self occlusion

Note that although many individuals are inside the field of view, not all of them would be really seen in a real situation. To estimate the visibility we iterate over the grid cells from inside to outside. Figure 7 shows one iteration example. As the grid becomes lighter the layer is increased.

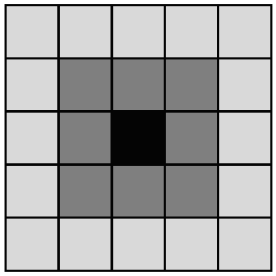


Figure 7: *Layer iteration. As the blocks become lighter the layer is increased.*

We stop the search when the number of processed neighbors reach the minimum visibility density or the maximum vision range is exceeded. In Figure 8, the minimum visibility density of four neighbors is reached before the maximum vision range and there are many neighbors discarded from the processing.

Grid size influence and estimation errors

Since the number of boids is only an estimative of the real occlusion, sometimes we may not consider boids that otherwise would be included in the behavior computation. This situation is depicted in Figure 9. This may change the behavior of the flock, specially if the number of boids being simulated is small.

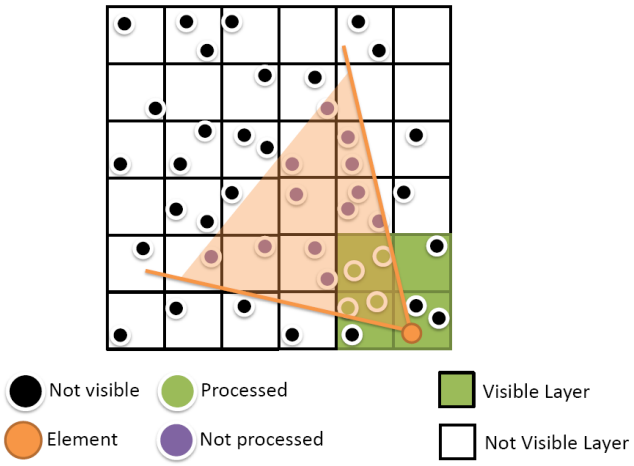


Figure 8: *Using visibility to cull boids.*

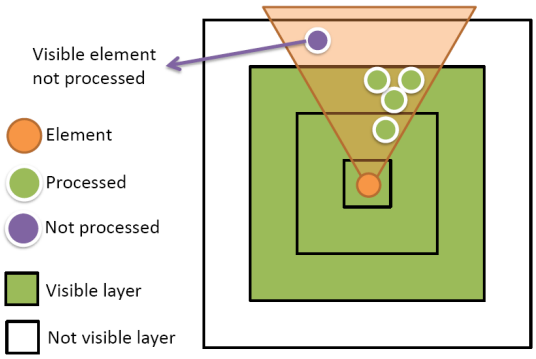


Figure 9: *Problem when the neighbors do not occlude another visible neighbor.*

Another important issue regarding the simulation is the grid density (elements per grid cell). When it uses on fixed number, if the density is too low the algorithm will discard a lot of empty cells and if the density is too high the algorithm will iterate over a lot of elements even if the most part of them are not visible.

We try to create a grid with a more uniform distribution by changing the size of the grid along the number of boids being simulated. The size of the grid is given by Equation 3, that considers the total number of boids and the desired density.

$$gridSize = \left\lceil \sqrt[3]{\frac{TotalBoids}{DesiredDensity}} \right\rceil + 1. \quad (3)$$

5 Results and Discussion

We performed a series of experiments in order to determine the effectiveness of the occlusion algorithm. The tests were executed in a PC with the following configuration:

- **Processor**: Athlon64x2 4200+
- **Memory**: 2Gb DDR1 - 400 Mhz
- **Graphics Processor**: GeForce8800GT - 512Mb DDR3 - PCI-e 16x

The shaders were written in Cg[Mark et al. 2003] and the application in C++.

We measured the time spent in the three steps described before. In the first step, we also measured the time spent transferring textures to and from the GPU and the time spent in grid construction. All times are in milliseconds.

We evaluated the influence of visibility estimation on rendering time. The number of boids changed from 256 to 207936 and the grid size changed according to the Equation 3. The vision angle was fixed in 45 degrees and the vision distance in 100 units. Our virtual environment extends from -500 to 500 units in the three axis. The model used for each boid has 268 triangles. Data was collected over a range of 45 different population sizes. The time for each one was obtained as the average of 30 frames.

The grid was an important factor for reducing the complexity of neighbor search in the algorithm, which would otherwise be $O(n^2)$ where n is the number of the boids.

Figure 10 shows the processing times of the different operations in each simulation step. Notice that only the grid construct time does not exhibit a linear increase. Since the grid construction needs to clear the entire grid in main memory, and the grid increases according the Equation 3, the total time increases accordingly to the grid storage complexity ($O(n^3)$).

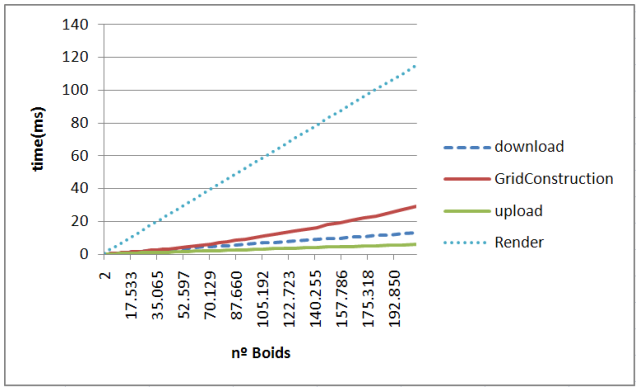


Figure 10: *Time to download, grid construction, upload and render in milliseconds.*

The upload time would have the same behavior, since it uploads the contents of the grid to the GPU, however it remains below all the other times as the number of boids increases. The render time and download time increases linearly with the number of boids. The grid construction time very low compared to the render time.

The simulation time is shown separately in Figure 11 because it is the only metric affected by the visibility estimation algorithm. Notice that the time near 70k and 192k boids increases abruptly, probably because these values affects some GPU internals, like cache. As expected, the simulation with visibility estimation consume less time than the algorithm without it.

The steep jumps on Figure 11 are similar to the ones observed on the NV4X Nvidia GPU architecture when resolving dynamic branches [Harris and Buck 2005]. However we are not sure if this also applies to the newer Tesla architecture (G8X, G9X). More tests would be required to settle down this point.

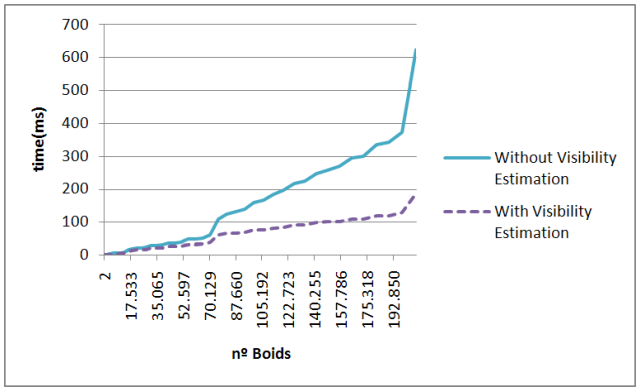


Figure 11: *Simulation time in milliseconds with and without the visibility estimation.*

To evaluate the time improvement of our approach, we computed the ratio (speedup) between the time taken by our implementation of Reynolds algorithm divided by the time taken by same algorithm with visibility estimation. Figure 12 shows the speedup obtained for an increasing number of boids. Notice that for few boids the speed up is smaller than 1, but for 10k boids or more, the speed up increases in almost a constant rate.

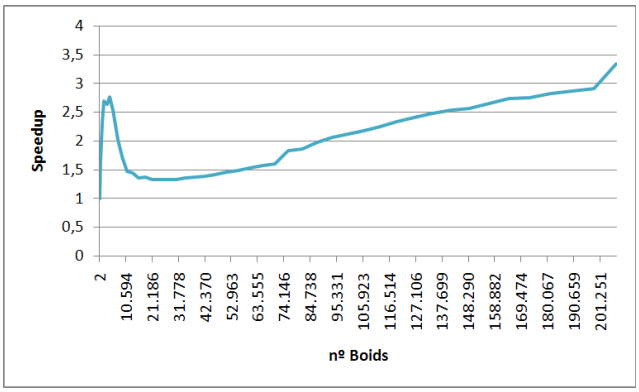


Figure 12: *Simulation with visibility speedup.*

When simulating boids, the expected result is a believable group simulation. However, after we turned on the visibility estimation, the simulation diverged from the behavior of the Reynolds original algorithm. This was expected since we change the dynamical parameters of the system. However very similar results were obtained by adjusting the steering constants. With the new parameters we could achieve a reasonable simulation with an speedup of more than 3 times of update rate. Since most crowd models usually require a lot of tweaking we do not see this as a real concern.

Figure 13 shows one example of a simulation with 20164 boids.

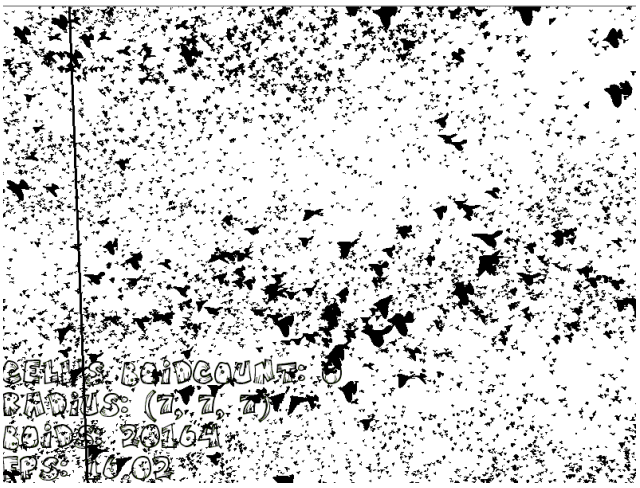


Figure 13: *Simulation example of 20164 boids.*

6 Conclusion and Future Work

We presented a mapping of a behavioral model to run in a GPU and proposed one extension to optimize the neighbor search by using a visibility estimation. The experimental results showed that this technique can be very effective in reducing the total complexity of crowd simulation. Some of the future extensions for this work include:

- Evaluate other types of occlusion estimation.
- Optimization of the linked list structure for cache access.
- Use a 2D texture for encoding the grid structure instead of a 3D
- Evaluate other spatial subdivision structures

- Include environment obstacle avoidance.

References

CHIARA, R. D., ERRA, U., SCARANO, V., AND TATAFIORE, M. 2004. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In *VMV*, 233–240.

COHEN-OR, D., CHRYSANTHOU, Y., AND SILVA, C. 2000. A survey of visibility for walkthrough applications. *Proc. of EUROGRAPHICS'00, course notes*.

COURTY, N., AND MUSSE, S. R. 2005. Simulation of large crowds in emergency situations including gaseous phenomena. In *CGI '05: Proceedings of the Computer Graphics International 2005*, IEEE Computer Society, Washington, DC, USA, 206–212.

DRONE, S. 2007. Real-time particle systems on the gpu in dynamic environments. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, 80–96.

HARRIS, M., AND BUCK, I. 2005. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, ch. GPU Flow-Control Idioms, 547–555.

KLOSOWSKI, J. T., AND SILVA, C. T. 2000. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics 6*, 2, 108–123.

KOLB, A., LATTA, L., AND REZK-SALAMA, C. 2004. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, New York, NY, USA, 123–131.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, ACM Press, New York, NY, USA, 896–907.

OPENGL, SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2005. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1, 80–113.

QUINN, M. J., METOYER, R., AND HUNTER-ZAWORSKI, K. 2003. Parallel implementation of the social forces model. In *in Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics*, 63–74.

REEVES, W. T. 1983. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph. 2*, 2, 91–108.

REYNOLDS, C. W. 1987. Flocks, herds and schools: A distributed behavioral model. *Proceedings of the 14th Annual Conference on Computer Graphics and interactive Techniques*, 25–34.

REYNOLDS, C. 1999. Steering behaviors for autonomous characters. In *Proceedings of Game Developers Conference 1999*, Miller Freeman Game Group, San Francisco, California, 763–782.

REYNOLDS, C. 2006. Big fast crowds on ps3. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, ACM, New York, NY, USA, 113–121.

SHAO, W., AND TERZOPOULOS, D. 2005. Autonomous pedestrians. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, 19–28.

SILLION, F. X. 1995. A unified hierarchical algorithm for global illumination with scattering volumes and object clusters. *IEEE Transactions on Visualization and Computer Graphics 1*, 3, 240–254.

TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum crowds. *ACM Trans. Graph. 25*, 3, 1160–1168.

P2PSE - A Peer-to-Peer Support for Multiplayer Games

Felipe J. Vilanova, Carlos Eduardo B. Bezerra , Marcos R. Crippa, Fábio R. Cecin, Cláudio F. R. Geyer
Universidade Federal do Rio Grande do Sul
Bento Gonçalves, 9500
Porto Alegre, RS, Brazil
+55 (51) 3316 0000

Abstract

The successful commercial MMOGs, at the moment, are implemented using some variant of the client-server model. This model offers simple design, good security and fast detection and treatment of cheating. However, it lacks scalability. The cost of maintaining the server becomes excessive with the increased number of customers. One approach to deal with this situation is distributing the MMOG simulation. The changes in the virtual world of the game would be processed by the machines of the clients, without interference from the server. This distribution can be achieved using the peer-to-peer model. In this paper, we describe a network engine, called P2PSE, which supports partially decentralized MMOGs, ensuring scalability and flexibility, while guaranteeing the basic properties of a client-server model, such as security and consistency. We first examine more closely the existing distributed models. After that, we discuss the P2PSE project and its characteristics. Finally we present a simulation of the architecture in comparison with the traditional client-server model.

Keywords: Peer-to-peer, Descentralization, Scalability, Multiplayer Games, Hybrid Topologies, Instanced Game, Distributed Systems

Author's Contact:

{fjvilanova, carlos.bezerra, mrcrippa, fcecin, geyer}@inf.ufrgs.br

1 Introduction

Multiplayer is a very popular game genre, due to its highly interactive nature. Among those games, MMOGs (Massively Multiplayer Online Games) are becoming increasingly popular. MMOGs are real-time multiplayer games played through the Internet, where a large number of players (usually thousands) play within a persistent-state virtual world. Successful examples include EverQuest [Sony Entertainment 1999] and World of Warcraft [Blizzard Entertainment 2004].

The most common network model used in MMOGs is client-server. In this model, the client only sends its data to the server (which can operate on a single machine, a cluster or a grid) and receives frequent updates from the server. The server process all the data from the clients, and broadcasts all the results that occur on the virtual world back to them. Some advantages of this model are: being simple in design; cheating can be efficiently detected and stopped; retaining control over access to the game; and being a predictable model.

The main disadvantage of the client-server model is the lack of scalability. The cost of maintaining the server-side becomes excessive with the increase in the number of clients. When it comes to commercial games, the usual approach is to continuously expand the number of machines on the server-side. That is a reasonable solution, however it is not suitable for projects on a limited budget such as those from small companies or research groups.

One way of dealing with the above problem is to fully or partially distribute the MMOG simulation. All the changes on the game world would be registered and dealt within the client-side, with no interference of the server-side. Security and world state consistence are issues on that model, because there isn't a point where the changes in the virtual world can be evaluated and considered legal or illegal. The main challenge of our project is thus to provide a partially decentralized support model for MMOGs, while retaining the basic properties of the client-server model such as security, consistency and scalability [Schiele et al. 2007].

In this paper a hybrid model is proposed, combining the security and consistency of the client-server model with the scalability and flexibility of the distributed model. In sections 2 and 3 we discuss the client-server and peer-to-peer models. The P2PSE (acronym in portuguese for "P2P for Entertainment Software") architecture is explained in section 4. Section 5 presents the simulations we made to compare client-server and peer-to-peer models. Section 6 presents conclusions on the topic.

2 Client-server model

The client-server paradigm is undoubtedly the most used one in the implementation of Internet real-time multiplayer games. Commercial examples of these games include Doom [id Software 1993], Quake [id Software 1996] and Counter-Strike [Valve 2000], all of them 3D real time action games. In any client-server game, massively multiplayer or not, the players interact with the environment and with other players through a client program. Each client has a network connection only with the server. The server is responsible for receiving the information from each client and passing updates to the other clients. There are several different ways the communication protocol between the client and the server can be implemented: some of them provide more security, while others provide efficiency.

Regardless of the technique used, the server is responsible for keeping the game state up-to-date. Since the game is a real time simulation, it is necessary for the state of the game to be frequently recalculated. In general, the server is configured to update its state in a fixed and relatively small frequency, and to periodically send update messages to all clients. Due to that, the server must have great processing power (CPU and, indirectly, memory) and enough bandwidth (network).

A distributed game can use the centralized simulation to accommodate thousands of simultaneous clients. In practice, however, it can be difficult to have all the processing power required to perform the simulation in real time in a single machine. One solution to this problem, commonly used in commercial MMOGs such as EverQuest and PlanetSide [Sony Entertainment 2003], is to divide the virtual world and simulate each piece of it on an individual machine such as a particular computer in a cluster. But even using the clusters to mitigate the problem of the server processing cost, the problem of network consumption still remains. The MMOG server, a single machine or a cluster, will need an Internet connection with low latency and very high bandwidth.

3 Peer-to-peer model

There is a growing interest in research and development of peer-to-peer architectures for MMOGs. The proposed models seek decentralization as a way of increasing scalability and reducing dependency on nodes in trusted areas. Other benefits include the elimination of central points of failure, as well as the increase of responsiveness.

The Communications Architecture for Massive Multiplayer Games, proposed by [Fiedler et al. 2002], was the first architecture to address the scalability problem by proposing a solution based on the publisher-subscriber paradigm. In this paradigm, the virtual world is divided into smaller pieces, usually called "cells", and each par-

ticipant can choose to sign (or participate) in only a few cells. Thus, each subscriber of a cell only needs to exchange update messages with subscribers of the same cell. These architectures also deal with other fundamental problems, such as the lack of responsiveness to commands generated by each participant problem.

There are other proposals that do not address the cheating problem, such as Hydra [Chan et al. 2007], that focuses on guaranteeing the consistency in the messages committed when nodes fail; and Mediator [Fan et al. 2007], which addresses the scalability and performance problems by proposing a solution based on a super-peer network and a reward scheme: peers that contribute more can use more resources.

It can be noted that recent proposals for MMOG support based on peer-to-peer overlays are becoming more aware of issues in actually deploying a peer-to-peer MMOG on the Internet, as they show more concern about peer-to-peer problems, like hostile service providers and bandwidth limitations, for instance. There are many pure peer-to-peer and hybrid model proposals. However, some of these proposals don't offer any kind of deterrent mechanism against cheating players, which is an essential feature for implementing an online massively distributed and persistent-state game. In the next section we will show how our model offers scalability and also cheat-resistance so that its commercial application is viable.

4 P2PSE project

The P2PSE project is a distributed simulation model, and also a library (in reality, a stack of C/C++ libraries) that implements the instanced game model, which will be described in section 4.1. The instanced game model is the price to pay for a simple approach in unifying security, consistency and scalability in a decentralized MMOG. The objective is to drastically reduce the processing and communication cost in the server-side.

4.1 The Instanced Game Model

Several MMOGs, like PlanetSide and World of Warcraft, offer to the player the illusion of one or several large and contiguous virtual spaces where all the gaming takes place. Those spaces are divided in segments, and each server machine or group of server machines is responsible for one segment. A warping system is necessary to tie all segments together, allowing the player to move from one space to the other transparently.

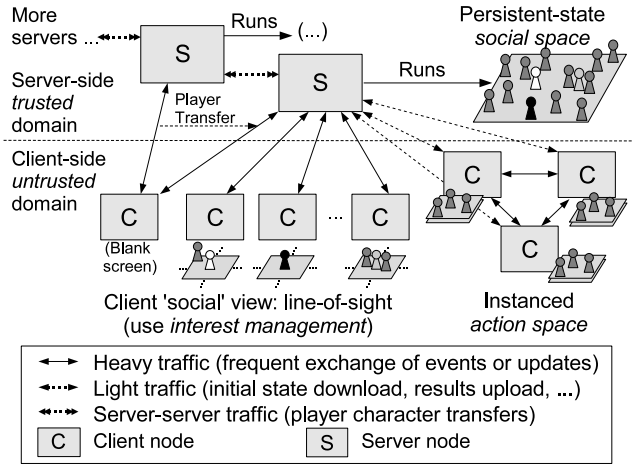


Figure 1: *The instanced 'social vs. action' MMOG*

The game Guild Wars introduced a new game model, which will be referenced to as the instanced game model [Cecin et al. 2006]. There are two virtual spaces on Guild Wars: the social space and the action space. The first kind is a medium-sized virtual space where a significant number of players can socialize, trade virtual goods and organize game sessions. Because of the nature of this space, low consistency requirements are necessary. Game sessions happen on the action space, which is a small-sized virtual space where a small number of players gather to play a game session. Higher consistency requirements are necessary on that space because of its fast and dynamic nature.

The relationship between the two spaces is as follows. When a game session ends, the players involved in it return to the social space. All the traits of the characters (e.g. virtual world money, experience points and statistics) are updated with new information from the session. The social space is designed as a contiguous world, as mentioned in the beginning of this section. The action spaces are created dynamically to support temporary game sessions with a small number of players. This space is destroyed after the session ends.

Guild Wars follows the client-server model. Our proposal, which is illustrated in Figure 1, is to use the instanced game model with a hybrid network architecture model. Since the interactions on the social space (chatting, trading) don't require high consistency, and there is a necessity for validation of who can play (game accounts, passwords), the server-side will be responsible for coordinating this part of the game. So even a server-side with modest processing capacity and network bandwidth could manage the social space if game quality is scaled down accordingly, for instance by sending less frequent updates to clients. The game sessions would be processed only on the client-side machines, avoiding unnecessary processing and communication cost on the server-side. The clients would form groups, which would operate within a peer-to-peer dynamics.

4.2 P2PSE architecture

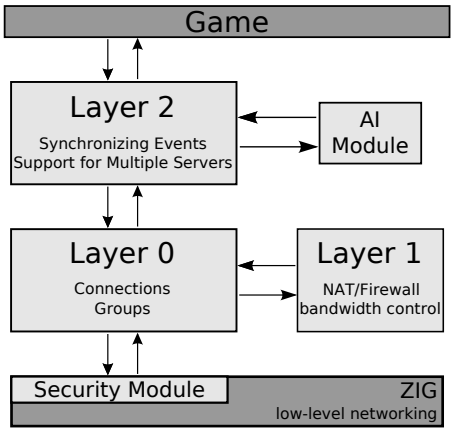


Figure 2: *P2PSE Architecture*

The functionality promised by our network engine is delivered by a set of libraries. Those libraries are organized in layers, as shown in Figure 2. The next sections describe the implementation of the three layers, followed by an overview of security and artificial intelligence (AI) techniques which are employed.

4.2.1 Layer 0

The first layer in the P2PSE architecture is responsible for forming the mesh of point-to-point connections. The clients' connections to the servers and to other clients are established and managed in this layer. The basic network functions for connection establishment, such as socket creation, connection and packet sending and receiving, are given by a low-level API provided by the ZIG library [Cecin 2007] which basically implements a protocol that resembles SCTP [Stewart et al. 2000] over UDP. The term "connection", when used in the context of the P2PSE library, refers to this ZIG-layer level protocol connection, which, again, is implemented over UDP, adding some features, such as reliability, timeout detection and utilization of only one single socket in every host to handle all communications.

The Layer 0 server is responsible for managing connections with all clients. Naturally, it maintains a list of all clients that are connected

to it along with a logical communication endpoint object for each client. When started, the server spawns a socialization space called the social space object. The clients that are not in an action space are bound to the social space - or to the "limbo", when in transit from one space to the other. All communication between clients that are in the social space passes through the server.

The server can create new empty groups in order to instantiate new action spaces, or destroy existing groups and their corresponding spaces. In the action spaces all the clients communicate directly, which requires them to already have the ability to send UDP packets to one another. The entry of a client into a group is managed by the server in the following manner:

1. The server sends a synchronization message to the new member and to all the other members informing them to which clients they must be connected in order to enter (or stay) in the group. The server sends a list with all the members of the group;
2. All members try then to establish connections to every other member in the recently received member list, or just keep the connection, if it has already been established (the ZIG library supports simultaneous bidirectional connections through the same socket);
3. When a client is entering a group, but still has not finished the process, it belongs to the "limbo";
4. As soon as the new member establishes connections to the other clients, the server excludes it from the limbo and inserts it in the member list of the group;
5. When a client is removed from the group, all of its connections to the other clients are interrupted and it returns to the social space;
6. If a client, for any reason, loses its connection to the server, it is removed from the group;
7. When a client enters or leaves a group, the server sends an up-to-date member list to the others.

The groups provide to the clients message exchanging functionalities, such as broadcast and unicast. The connections between the group members are represented by peers. The clients, when in groups, will keep a list of peers, what is equivalent to the client list of the server. With this list, the clients in that group manage their connections between one another.

Layer 0 is also responsible for the timeout control of the connections. Keep-alive dummy messages are sent regularly in order to maintain all connections active. If, after an interval of time specified by the application, no messages from a host have arrived, the connection to that host is closed.

The communication between the layer 0 and the ZIG library is done with a listener. This listener is implemented in layer 0 and receives ZIG callbacks to indicate the occurrence of events such as connections, disconnections and receiving of messages. The communication between the upper layers of the P2PSE library and layer 0 is made through service calls. Layer 0 returns callbacks to the upper layers in response to these service calls.

4.2.2 Layer 1

Layer 1 of the P2PSE architecture is responsible for handling connection-related problems between the peers within a group. This layer will ensure that all peers are able to send and receive messages from each other.

We see two main problems that can cause a connection failure between two peers: NAT/Firewall blocking and bandwidth insufficiency. When the Layer 0 indicates that a message must be sent from one peer to another, Layer 1 must verify if there is a direct connection between the peers. If this connection doesn't exist, an alternative path must be discovered, using intermediate peers. After this route is created, the messages will be sent through it.

Among all peers within a group there are different bandwidth capabilities. It is very likely to occur a situation where a peer doesn't have enough upload bandwidth to deal with the number of messages that it has to send. Layer 1 will identify such situation and, instead of sending the messages directly to the destinations, send them to a peer with good upload bandwidth capability, holding him responsible to send the message.

4.2.3 Layer 2

The upper layer of the P2PSE implementation is responsible for guaranteeing the simulation consistency on action spaces, providing support for multiple servers and exposing all of the implementation's functions to the game programmer. Due to the simulation of action spaces being executed asynchronously in each peer, there is no intrinsic guarantee of consistency. To partially address that, Layer 2 introduces the role of the super-peer, which is a special peer responsible for receiving, ordering and redistributing some of the game events such as updating player scores and picking up the flag in a capture-the-flag action game, for instance. To avoid overloading the super-peer, position updates or player movement requests (depending on the game's protocol design) are to be sent directly from each peer to every other peer using the unicast full-mesh provided by the lower layers. Not only position updates, but any other frequent, delay-sensitive and weakly-coupled message type should also be sent peer-to-peer.

Depending on the game, the resulting conflicts from a lack of centralized timestamping of movement and similar events, whenever they ensue, can be either handled by each peer individually or left for the super-peer to detect it and issue a special correction message later, as the BZFlag protocol does [Pellegrino and Dovrolis 2003]. Local corrections are an option if each peer is authoritative over its own avatar's position and if it broadcasts position updates for its own avatar. For instance, if a peer's local avatar overlaps with a remote peer's, the local avatar can push its own avatar out of that position. The remote avatar will do the same. The only issue is ensuring that the correction step won't make both peers try to resolve the situation by unstucking their avatars to the same spot again. This local conflict resolution is what we are currently employing in our capture-the-flag action prototype, Hoverkill [Singular Studios 2006].

Besides ordering and distributing some of the events, the super-peer is responsible for the communication of the peer-to-peer action group with the server. The super-peer is the one that reports gameplay results back to the server, whenever necessary.

Layer 2 also introduces support for multiple game servers (Figure 3) in the API. The idea is that players can choose their server from a list of servers. Each server will typically host a single social space and when that is full (server capacity is reached) players can connect to other social spaces in other servers. To tie all of the players scattered on different servers together, there is a master server to which all servers are connected. Whenever a player connects to a server and authenticates, his player account is downloaded from the master server to that game server. When the client drops from the game server, his account is uploaded back to the master server. There is a simple Yellow Pages Server proxy that can be used by clients to query the master server for the currently available game servers.

The social space support provided by Layer 2 is a simple client-server API. Any interest management [Morse et al. 1996] in the social space is currently left for the application to implement. We are planning to add something akin to OpenTNL ghosting [Garage Games 2004] into the API which would allow the game programmers to optimize the social space more easily. Another idea is to implement a peer-to-peer mesh for social spaces, but that one would have to be scalable (no full-mesh possible). Since we are aiming at a simple solution, the use of probabilistic broadcast for position updates on the social space is being considered, leaving timestamping and distribution of more infrequent events for the server.

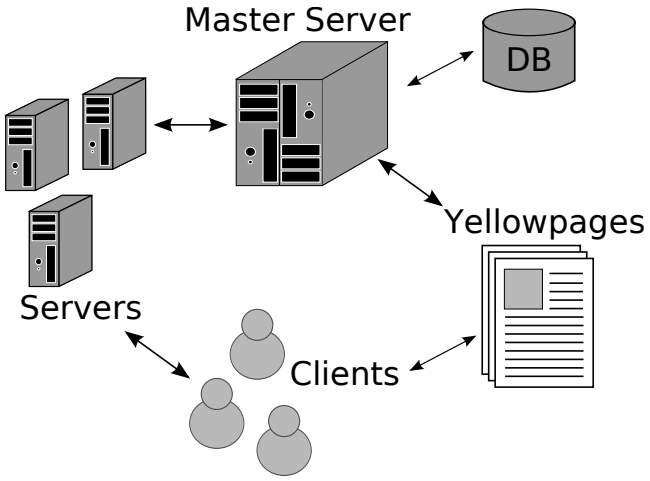


Figure 3: *Multiple Servers Support*

4.2.4 Artificial Intelligence Module

In the previous section, we revealed that we encourage peers in an action space to send authoritative position updates, among other events that might have low-latency requirements and a low probability of having their order of execution affecting the resulting game states too much (weakly-consistent events). This opens up the possibility of cheating.

In this project we tried a somewhat unique approach [Gaspareto et al. 2008]. Since most events would be still centralized at the super-peer and most action games have one-per-client avatar position updates as their latency, consistency and bandwidth bottlenecks, we just left peers with total control over their positions and with the responsibility for sending them directly to other peers in a broadcast fashion. This optimizes for consistency and latency, and doesn't create a bandwidth bottleneck at the super-peer. On top of that, we tried AI-based detection of unusual patterns over streams of position updates coming from each peer. The approach used for cheat detection was focused on the data packets being transferred during the game sessions - both in the transmitters and receptors - and how they are used during the functioning of the system. Only packets relevant to the cheat being investigated are considered.

The cheat detection module is implemented in an independent way and works as an attachment to P2PSE, which, with a parameter passage protocol, informs the set of data received from a player and returns a correspondent "fraud index", that tries to quantify the cheating that the player is assumed to have performed in the analyzed period.

The adopted approach was to consider only movement data as input to the cheat detection module. Although this might seem to be too few information, it is based on movement that malicious players perform great part of their cheating. For example, one could try to move at a speed higher than the maximum speed allowed in the game (most common kind of cheating in MMOGs) or teleport instantaneously to strategic spots, which is also known as speed-cheating.

This way, the point is to analyze a set of movements of the player, in order to detect anomalous sequences. Each player will be evaluated individually, based on his position history, using Artificial Neural Networks (ANN). ANNs, because of their great generalization and pattern recognition capacity, present satisfactory results in tasks similar to the speed-cheating problem. The idea is to identify, in the set of analyzed data, at least one movement that could be considered a fraud, which characterizes the player where this set of data came from as a cheater. The ANN then identifies and classifies the movement patterns as acceptable and non-acceptable.

During the modeling and design of the cheat detection module, the architectures of the ANNs, the set of training data and the set of tests have been defined in order to allows them to solve this kind of problem. Considering that the task consists of the classification of behavior patterns, it has been decided to use the full-connected multilayer perceptron ANN with supervised learning based on back-propagation. The reason to choose it was that it is the most commonly used ANN model in many classification applications, due to it simplicity of understanding and implementation.

The projected ANN was simulated using MatLab, to evaluate its efficiency. MatLab allows the creation of complex ANN structures only by informing the neural network parameters, such as the number of neurons in each layer, training algorithm, rates, values that configure this training etc. MatLab was also used to train the ANN to be integrated to the game. This way, the implemented module only performs propagation, but not training of the ANN.

After the efficiency has been proven, the module was implemented and tested in Hoverkill, obtaining a 21.14% cheat detection, and 0.98% false positives. The results were satisfactory because, even though the cheat detection percentage was only 21.14%, the rate of false positives was less than 1%. It is preferable not to detect a cheat than considering a fair player as a cheater. Other parameters, besides movement data, may be added in the future, in order to detect other kinds of cheating.

4.2.5 Security Module

The main purpose of the security module is to provide secure communication between servers and clients, and between peers in an action space group. Our security module is almost completely transparent to the application, both in terms of API and in network and CPU overhead. The module does many tasks behind the scenes, such as cryptographic key distribution and secure channel handshakes, tending to each application's individual needs for message confidentiality, integrity and authenticity.

The implementation works mainly by intercepting any engine UDP packets before they are written to the UDP socket for sending. In each packet, a small header is inserted with control information and, upon first contact between two parties (being them clients, servers or both), that message is delayed while another quick message exchange is performed for setting up the secure channel on-the-fly. The root Certificate Authority (CA) of the cryptographic system is the operator of the master server.

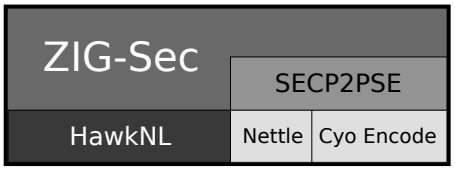


Figure 4: *Security module architecture*

The module has been implemented by extending the ZIG library functionalities, providing secure communication, transparently to the application, being responsible for distributing cryptogaphic keys. It runs over HawkNL [Hawk Software 2006] and over SECP2PSE. HawkNL is a library that, among other functionalities, provides a simpler and portable API for sockets. SECP2PSE is our library that provides C++ abstractions for cryptographic algorithms available in Nettle [Möller 2006] and CyoEncode [CyoTec 2004]. SECP2PSE also implements certification and serialization functionalities meant for security related transmissions. Figure 4 illustrates the security module architecture.

In our secure communication protocol, all ZIG-generated packets sent, both by clients and by servers, are intercepted right before being transmitted through the network. In each one of them, a small header is introduced with control information used to guarantee confidentiality, integrity and authenticity of data when such attributes are required. The size of this header is variable, for it depends on what characteristics are desired by the application, and on the intended level of security (key length). For a real-time application, such as distributed games, usually there is no need for a very long key, because the hacker would have too little time to break the algorithm by brute force. For example, using AES and HMAC-MD5, it would be necessary only 21 bytes. In general, the overhead

size of each packet is somewhere between 1 and 37 bytes. These packets containing user data and control header are then called **data packets**.

There is also another type of packet, called **control packet**, which is utilized to negotiate keys between peers who intend to communicate with each other. They are usually longer, for they need to attach a copy of the users certificates. However, these packets are sent only once in the beginning of the communication with a peer which has been unknown until that moment. Once they have negotiated a key, they only need to negotiate a new one when one of the peers loses it or if the key is already being used for a time long enough for the key to become breakable by brute force.

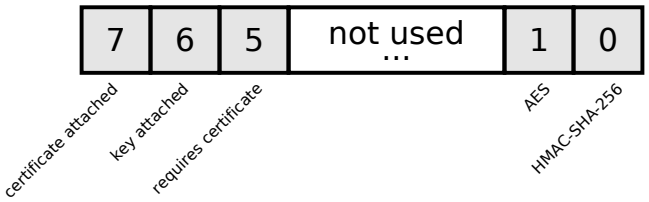


Figure 5: *Control header*

The control header consists of the packet descriptor (1 byte). As shown in Figure 5, the bit 0 indicates whether the data are protected by HMAC-SHA-256. The bit 1 indicates whether the data are encrypted with AES. Bit 5 indicates that the message source needs the destination host certificate. Bit 6 indicates that a key has been generated and attached to the packet in control data and bit 7 indicates that the certificate of the origin has been attached to the packet in control data. Bits 2, 3 and 4 are not used.

When the application requires the secure transmission of a packet to the security library, it must inform: data being transmitted, socket used for sending, address and port of the destination, type of desired security (confidentiality, authenticity and integrity) and the identification of the user which will receive the message at the provided address.

To guarantee confidentiality of the messages, it can be used both symmetric and asymmetric cryptographic algorithms. Both are considered secure, depending on the length of the key used. The secure communication protocol adopts the strategy of initially using an asymmetric cryptography algorithm to negotiate a key to be used then by each pair of computers to communicate securely with symmetric cryptography. This approach allows to reduce the processing time of encrypting and decrypting data, for symmetric cryptography requires less CPU time. Along with the confidentiality provided by this mechanism, authenticity may also be achieved by using hash functions with keys, also known as Message Authentication Code (MAC), to verify the authenticity and integrity of the messages.

Every time that a peer needs to communicate with another which was unknown until that time, they first need to negotiate a key. Therefore, before sending the data packet required by the application, the library sends a requisition for the certificate of the destination through a control packet, which already carries the source's own certificate. When the packet arrives in the destination peer, the security library reads the attached certificate and verifies its authenticity with the specified certification authority responsible for the game. If the verification fails, the packet is discarded. If it succeeds, the process continues: the peer who received the packet allocates a reply packet, where it attaches its own certificate, as required, and a randomly generated key, which is encrypted using the public key present in the certificate of the peer that initiated the whole process. This way, it can be assured that only the receiver of this reply packet will be able to decrypt it and then obtain the key. To guarantee that the sender of this packet is not an attacker who intercepted the communication, the already encrypted key is re-encrypted, this time with the private key of the sender. This way, both authenticity and integrity are provided.

After receiving the reply packet containing the double-encrypted key, the first peer decrypts it using the public key of the other peer, verifying the authenticity of the message, and then decrypts it again, using its own private key. When this whole process is successfully completed, the peers will be able to securely communicate with a fast symmetric cryptographic algorithm using the generated key.

5 Simulations

Since one of the main objectives of the P2PSE engine is to reduce the bandwidth utilization by the server, a numerical comparison between the load on a traditional server and the load on a P2PSE server was required. In order to do that, it was necessary to simulate two game servers, each one with players connected to it. One of them was a traditional game server, to which the clients kept connected and trading packets the whole time. The other one was a P2PSE server, where the players needed to trade packets with the server only while they were in the social space. In the action space, the players formed a peer-to-peer group, communicating to one another directly.

The ns-2 simulator [McCanne et al. 1995] was used to create the simulated environment and perform the simulation itself. The servers and clients - both traditional and P2PSE - were programmed extending the Application base class of the ns-2 API. The P2PSE clients connect to the P2PSE server, sending packets to it at a fixed rate while in the social space. The server, in turn, sends to each client in the social space an update containing information about all other clients in the same space. When a player leaves the social space and joins a peer-to-peer group of an action space, it stops exchanging packets with the server, due to its direct communication with the other players. The non-P2PSE server and clients behave simply as follows: each client sends a packet to the server at a given fixed rate, while the server broadcasts an update to all clients, also at a fixed rate. Although this approach does not take into consideration possible optimizations on the server, such as interest management [Morse et al. 1996], this is not relevant, since the same optimizations could be used in the social space of the P2PSE server, resulting in a similar percentage of bandwidth usage reduction, when comparing the two server models.

We based our simulation on some works, such as [Breu 2007], [Park et al. 2005] and [Feng et al. 2005], which analyse the network traffic generated by action games, since this is the genre targeted by the P2PSE engine. As in most first-person shooters, the transport layer protocol used was UDP. Each packet received by the server from the clients is 100 bytes long. The update sent to each client, containing information about all the players, is proportional to the number of players present in the game, for the traditional server. For the P2PSE server, the update length is based only on the number of players in the social space. In both cases, the length of this update is 100 bytes multiplied by the number of players who will be updated. The interval between two consecutive packets received from each client is 150 ms, while the interval between two consecutive updates from the server to each client is 100 ms. Each player keeps alternating between the social space and the action space. The time he stays in each one of them is chosen randomly, ranging from 0 to 20 minutes, for each space change. The whole game session lasts for 1 hour.

The results were collected as follows: to measure the average upload bandwidth usage by the server, the sizes of all packets sent during the whole session were added up and then divided by the session time; to determine the maximum usage, it was measured how many bytes had been sent each second and the highest value was selected. In Figure 6, it is shown the maximum and average upload bandwidth utilization results found for a traditional server and a P2PSE server. In Figure 7, the download bandwidth usage is also depicted. Table 1 and Table 2 show the numerical values of the average bandwidth utilization found in the simulation.

As we can see, the average download bandwidth utilization is decreased by, approximately, one half, and the average upload bandwidth utilization is reduced to one quarter. This happens because the periods during which a client stays on the social and action spaces have approximately the same duration. In consequence of that, there are roughly half the clients exchanging packets with the server for each instant, in the average. As each client sends fixed

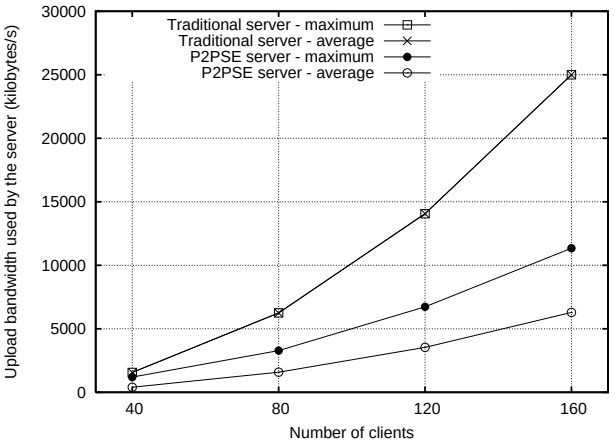


Figure 6: *Upload bandwidth utilization by the server*

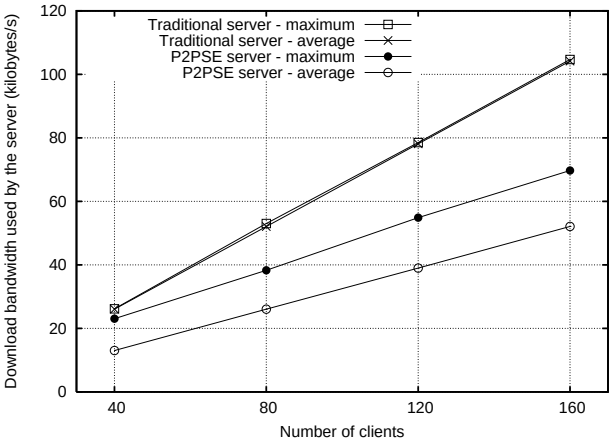


Figure 7: *Download bandwidth utilization by the server*

size packets periodically to the server, the bandwidth occupation grows linearly, and so, as the number of players halves, so does the server's download bandwidth utilization. However, as the update sent by the server to each client is proportional to the number of clients being updated, the upload bandwidth utilization by the server grows quadratically. Therefore, when the number of players is divided by two, the upload bandwidth utilization of the server is divided by the square of two, that is, four.

It is important to notice, however, that the time a player usually spends playing action games, such as first-person shooters, is much longer than the time they keep talking in the game chat room. Therefore, the average time spent by each player in the social space should be much shorter than the time spent on the action spaces in the simulation. Also, the social space will most likely have much less need for frequent updates, allowing further decrease of the bandwidth utilization by the P2PSE servers. Anyway, those parameters were chosen considering a pessimistic scenario, where the social space is as interactive as the action space, and the players spend more time socializing than it usually occurs in this kind of games.

6 Conclusion

We proposed a different approach to MMOGs, that uses P2P groups and transfers the simulation processing to them. In order to provide security and scalability, the game model was restricted to the instanced game model. The existence of a server-side guarantees that, if deemed necessary, the server can act as a final arbiter. The performed simulations demonstrate that, even in a pessimistic scenario, the reduction of the average bandwidth usage by the game server can be significant. The download bandwidth utilization was reduced by one half, while the upload bandwidth utilization was the most benefited one, decreasing by 75%, allowing to reduce the maintenance cost of such kind of server.

Table 1: *Average upload bandwidth utilization (kB/s)*

Clients	Traditional Server	P2PSE Server
40	1562.50	400.76
80	6250.00	1581.38
120	14062.49	3533.67
160	24999.98	6290.14

Table 2: *Average download bandwidth utilization (kB/s)*

Clients	Traditional Server	P2PSE Server
40	26.04	13.02
80	52.08	26.04
120	78.12	39.00
160	104.17	52.09

We are currently integrating the P2PSE architecture with our game prototype, Hoverkill. This game is a client-server capture-the-flag tank game, and its network system is being replaced by our architecture. The final goal is to be able to perform tests under a real environment, in order to base our conclusions in real results rather than simulations.

Acknowledgements

This work was supported by the Funding for Studies and Projects (FINEP), through the P2PSE project (P2PSE-5849-1), and by the National Research Council (CNPq). We also would like to thank Andrius Batalauskas (Hochschule Bremen) for the final review of the text.

References

BLIZZARD ENTERTAINMENT, 2004. World of warcraft. http://www.worldofwarcraft.com/.

BREU, L., 2007. Online-Games: Traffic Analysis of Popular Game Servers (Counter Strike: Source).

CECIN, F. R., GEYER, C. F. R., RABELLO, S., AND BARBOSA, J. L. V. 2006. A peer-to-peer simulation technique for instanced massively multiplayer games. *Proceedings of the Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'06)-Volume 00*, 43–50.

CECIN, F. R., 2007. Zig game engine. http://zige.sourceforge.net/.

CHAN, L., YONG, J., BAI, J., LEONG, B., AND TAN, R. 2007. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 37–42.

CYOTEC, 2004. CyoEncode. http://www.cyotec.com/resources/cyoencode/.

FAN, L., TAYLOR, H., AND TRINDER, P. 2007. Mediator: a design framework for P2P MMOGs. *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 43–48.

FENG, W., CHANG, F., FENG, W., AND WALPOLE, J. 2005. A traffic characterization of popular on-line games. *Networking, IEEE/ACM Transactions on 13*, 3, 488–500.

FIEDLER, S., WALLNER, M., AND WEBER, M. 2002. A communication architecture for massive multiplayer games. *Proceedings of the 1st workshop on Network and system support for games*, 14–22.

GARAGE GAMES, 2004. Open tnl - torque network library. http://www.opentnl.org/.

GASPARETO, O., BARONE, D., AND SCHNEIDER, A. 2008. Neural Networks Applied to Speed Cheating Detection in Online

Computer Games. *To be published in the 4 th International Conference on Natural Computation (ICNC'08), Shandong University, Jinan, China*.

HAWK SOFTWARE, 2006. Hawknl. http://www.hawksoft.com/hawknl/.

ID SOFTWARE, 1993. Doom. http://www.idsoftware.com/games/doom/.

ID SOFTWARE, 1996. Quake. http://www.idsoftware.com/games/quake/.

MCCANNE, S., FLOYD, S., ET AL. 1995. Network simulator ns-2. *The Vint project, available for download at http://www.isi.edu/nsnam/ns*.

MORSE, K., ET AL. 1996. Interest management in large-scale distributed simulations. *Technical Report ICS-TR-96-27, University of California, Irvine*.

MÖLLER, N., 2006. Nettle. http://www.lysator.liu.se/~nisse/nettle/.

PARK, H., KIM, T., AND KIM, S. 2005. Network Traffic Analysis and Modeling for Games. *LECTURE NOTES IN COMPUTER SCIENCE 3828*, 1056.

PELLEGRINO, J., AND DOVROLIS, C. 2003. Bandwidth requirement and state consistency in three multiplayer game architectures. *Proceedings of the 2nd workshop on Network and system support for games*, 52–59.

SCHIELE, G., SUSELBECK, R., WACKER, A., HAHNER, J., BECKER, C., AND WEIS, T. 2007. Requirements of Peer-to-Peer-based Massively Multiplayer Online Gaming. *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 773–782.

SINGULAR STUDIOS, 2006. Hoverkill. http://www.singularstudios.com/sitehoverkill/.

SONY ENTERTAINMENT, 1999. Everquest. http://www.everquest.com/.

SONY ENTERTAINMENT, 2003. Planetside. http://planetside.station.sony.com/.

STEWART, R., XIE, Q., MORNEAULT, K., SHARP, C., SCHWARZBAUER, H., TAYLOR, T., RYTINA, I., KALLA, M., ZHANG, L., AND PAXSON, V. 2000. Stream Control Transmission Protocol.

VALVE, 2000. Counter-strike. http://www.counter-strike.net/.

A Feature Model Proposal for Computer Games Design

Victor T. Sarinho Antônio L. Apolinário*

State University of Feira de Santana, Brazil

Abstract

An increasingly important attribute of modern software development is that of variability. Software variability is the ability to control the software change in a particular context. To manage the game development variability, due to the game domain diversity, the most successful approach was first achieved by game engines. Nowadays, software variability can be initially identified based on the concept of feature, a logical unit of behavior that corresponds to a set of functional and quality requirements in a system. This paper presents a new way to describe a game, based on a feature modeling. The NESI model is based on four main features: Narrative, Entertainment, Simulation and Interaction. The main contribution of this paper is to propose a model to design the variability aspects of computer games, simplifying the effort of game design projects.

Keywords: software variability, feature modeling, game design.

Authors' contact:
vsarinho@gmail.com
*apolinario@ecomp.uefs.br

1. Introduction

According Svahnberg et al. [2005], "software variability is the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context".

Software variability achive many issues, "ranging from the development process itself to the various artifacts created during the development process, such as requirements specifications, design documents, source code, and executable binaries" [Svahnberg et al. 2005].

According Gurp et al. [2001], "a variability can be initially identified based on the concept of feature". They define a feature as "a logical unit of behavior that corresponds to a set of functional and quality requirements". Indeed, "variability management is concerned with the management of the differences between products throughout the Product Line lifecycle" [Junior 2005].

In computer games, the variability is a direct consequence of the game domain diversity, working with simulations (sports, adventure, fighting), hardware technologies (mobile games, web games), human interactions (immersion, multiplayer) and complex stories (games based on movies, RPG series) [Kent 2001].

To manage the game development variability, the most success approach was first achieved by Quake game engine [Lewis 2002], a platform based product line strategy [Bosch 2002] for game creation.

Others interesting attempts to manage the game variability, based on SPL (Software Product Line) concepts, were developed [Zhang and Jarzabek 2005; Alves 2007], but only for specific games design, and without commitment to the existing works about computer games domain [Björk and Holopainen 2005; Davidsson et al. 2004; Zagal et al. 2005; Furtado 2006; Bates and Bates 2004; Rollings and Morris 2004; Hunicke et al. 2004; Järvinen 2007; Lemay 2007].

In this way, the objective of this paper is present a feature modeling proposal to design the variability aspects of computer games, based on the analysis of the main computer games domain engineering works.

This paper is organized as follows: Section 2 presents some important game domain engineering works. Section 3 describes the feature modeling technique used and the proposed computer game feature model. Finally, Section 4 presents the conclusions.

2. Related Work

How to make a computer game using existent components? Can a set of components be integrated or composed to produce a new computer game? What kinds of game components should exist to build a computer game? What features a software should contain to be considered a computer game? Unfortunately, there is not a direct response for these questions, but just "hints" described by game domain engineering works.

For Hunicke et al. [2004], the Mechanics, Dynamics, and Aesthetics (MDA) framework described a game as a collection of Mechanics to compose Dynamics, and a collection of Dynamics to compose Aesthetics characteristics of a game. It was a interesting conception of game views, but without resolve the gap between game design and game development.

Björk and Holopainen [2005] described a model to support the game design using a set of game design patterns. It has been a good approach to describe the components of games and its patterns of interaction, but due to the very high level patterns description, and the repetition of semantic concepts between them, the gap between game design and game development persists.

Zagal et al. [2005] proposed the Game Ontology Project (GOP), a framework that define a hierarchy of ontology concepts: interface, rules, goals, entities, and

entity manipulation. In fact, a good representation of game rules was described, but more ontology concepts are necessary to represent game design and game development aspects.

Furtado [2006] proposed the use of a game ontology to set game implementations. Low level game design aspects were described, but focused on a specific game domain, becoming insufficient to represent the game diversity.

Järvinen [2007] presented a design methodology with psychological theories of cognition and emotions for game design, but with a great distance of game implementation aspects.

Zhang and Jarzabek [2005] proposed an RPG product line architecture (RPG-PLA), grouping similarities and differences among four RPGs (probably the first use of features on game design). As result, a specific RPG feature model, and insufficient solution to represent the game diversity, was described.

The works described above are focused on generic game domain descriptions and specific game project implementations. Therefore, the definition of an intermediate modeling technique, covering domain and design game aspects, reducing the gap between them, to describe and analyze generic game components and concepts, is necessary.

3. Computer Game Feature Model

Next subsections will present the used feature modeling technique (section 3.1), and the proposed feature models (sections 3.2, 3.3, 3.4, 3.5 and 3.6) to design computer game variability aspects.

3.1 Feature Modeling

The feature model concept was first introduced by Kang et al [1990], in the Feature Oriented Domain Analysis (FODA) method, to help the identification of important or domain special properties during analysis phase.

According Antkiewicz and Czarnecki [2004], a feature is "a system property that is relevant to some stakeholder", and is "used to capture commonalities or discriminate among desired products".

A feature model consists of one or more feature diagrams. These diagrams organize features into hierarchies, as a decorated tree, containing features identified in a system.

Others notations have been proposed to expand the feature model representativeness and to provide support to different types of structural relationships. For example, Czarnecki and Eisenecker [2000] uses XOR and OR relationships to represent alternative and mutually exclusive features, and Czarnecki et al. [2005] proposes a feature cardinality-based notation.

This paper will use both feature notations to represent the feature model proposal for computer games design.

3.2 A Feature Model for Computer Games

The basis of NESI model is to define a game as a combination of four main standard features: Narrative, Entertainment, Simulation and Interaction, which was called NESI model (Figure 1).

The NESI model was based on the game conceptual definitions proposed by Esposito [2005], where a definition of videogame, combining existing researchs about game, play, interactivity, and narrative concepts, was described.

The structure of the NESI model was derived from the analysis of different domain engineering [Björk and Holopainen 2005; Davidsson et al. 2004; Zagal et al. 2005; Furtado 2006; Bates and Bates 2004; Rollings and Morris 2004; Hunicke et al. 2004; Järvinen 2007; Lemay 2007] and implementation works [Zhang and Jarzabek 2005; Alves 2007; Trinta 2007] of computer games.

A *Narrative* is a Flow, a dynamic script trying to achive Goals following some defined game Rules. *Entertainment* is represented by the player Immersion during the game, accompanied by a Theme proposal, to characterize the player and the game in the proposed reality. *Simulation* is a combination of Elements (resources for play) and their Relationships, that occurs in a defined Environment (spaces to play). The human *Interaction* is represented by the Control (game input) and the Presentation (game output) features, consolidated by the GamePlay feature that manages the game state and execution as a whole.

To illustrate the proposed model, just imagine a Chess game. Chess is a *Narrative*, trying to execute some steps in a Flow (prepare the game, start the game, wait for player movement), according defined Rules

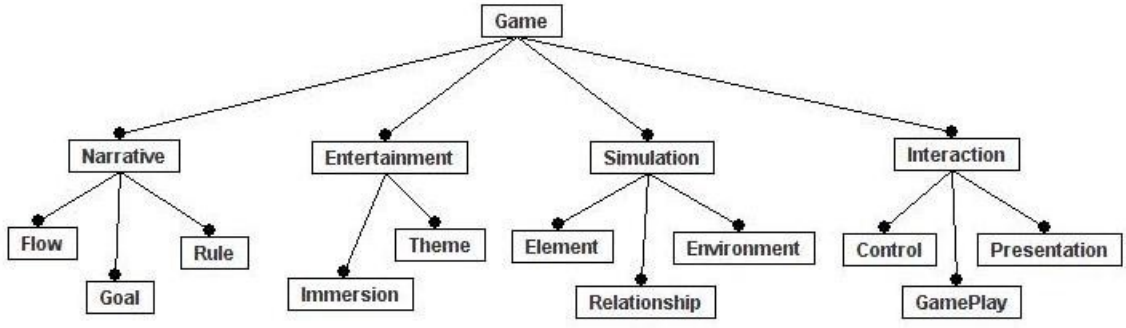


Figure 1: NESI model for computer games representation.

(types of movement) to complete these Goals (take the queen, kill the king). Chess is a kind of *Entertainment* because it offers a combination of Immersion feelings (Emotion, Challenge), following a Theme (war of kingdoms) according a Story or not (kill the bad white king for example) and using some Avatars (queen, bishop, king). Chess is a *Simulation*, by the representation of Environment (game board), Elements (white and black pieces), and their Relationships (Alliances in a multiplayer challenge, piece Consumption). Finally, Chess is a *Interaction*, because it offers Presentation opportunities by Visual (3D and 2D Cameras to show the Chess board) and Aural (special SoundEffects for each player movement) features. To Control the Chess game, an IndirectControl (PointAndClick using the mouse to move any piece) is more used, and for the GamePlay, a Persistent Context (storing the game state) with TurnBased Policy Execution (one player movement each time) and PreGame Customization (config the game before play it) are the most common used features.

3.3 Narrative Features

As illustrated in Figure 1, the Narrative feature is composed by three subfeatures: Flow, Goal and Rule.

The Flow feature, which represents a sequence of operations [Taylor 2006; Brown 2006], is basically a collection of Events and Nodes (Figure 2).

Each Node contains: a collection of ActionList, which contains Actions to be performed; and a collection of Transitions, to define the new execution Node in the Flow when one Transition is performed.

During the game execution Events are used to distinguish what ActionLists or Transitions will be performed. As result, for each Event in the execution Node, new Actions or Transitions can be performed, triggering new Events again.

Different types of Nodes could be used: Fork, Join, Start, and others. When these nodes are combined they compose a Flow in the game.

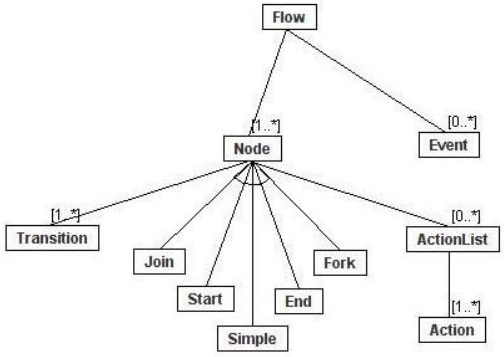


Figure 2: Flow feature diagram.

According Zagal et al. [2005], "goals are the in-game objectives or conditions that the player must meet if he expects to succeed at the game".

To complete a game, goals must be attained. The Goal feature (Figure 3) is composed by many SubGoals, which contains the following features: Change (the Goal is Static, Dynamic or can be Customized by the player), Constraint (regular expression composed by and, or, not and sequence

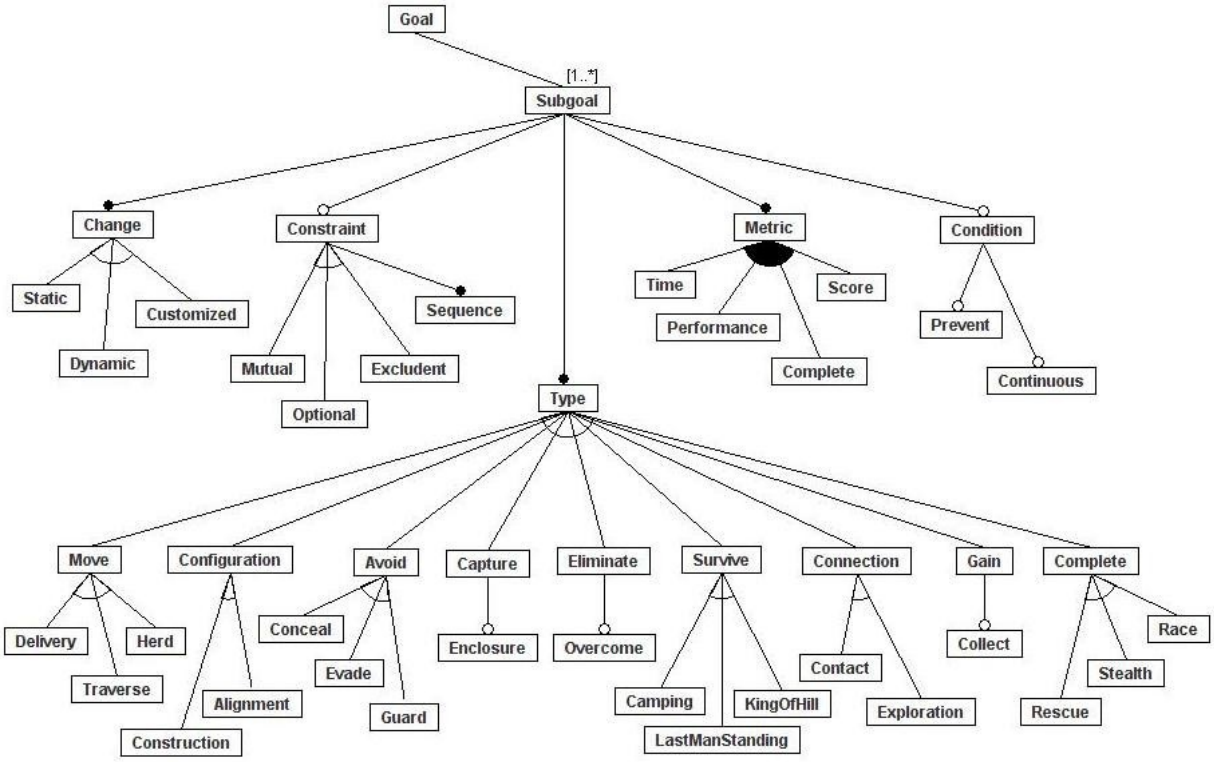


Figure 3: Goal feature diagram.

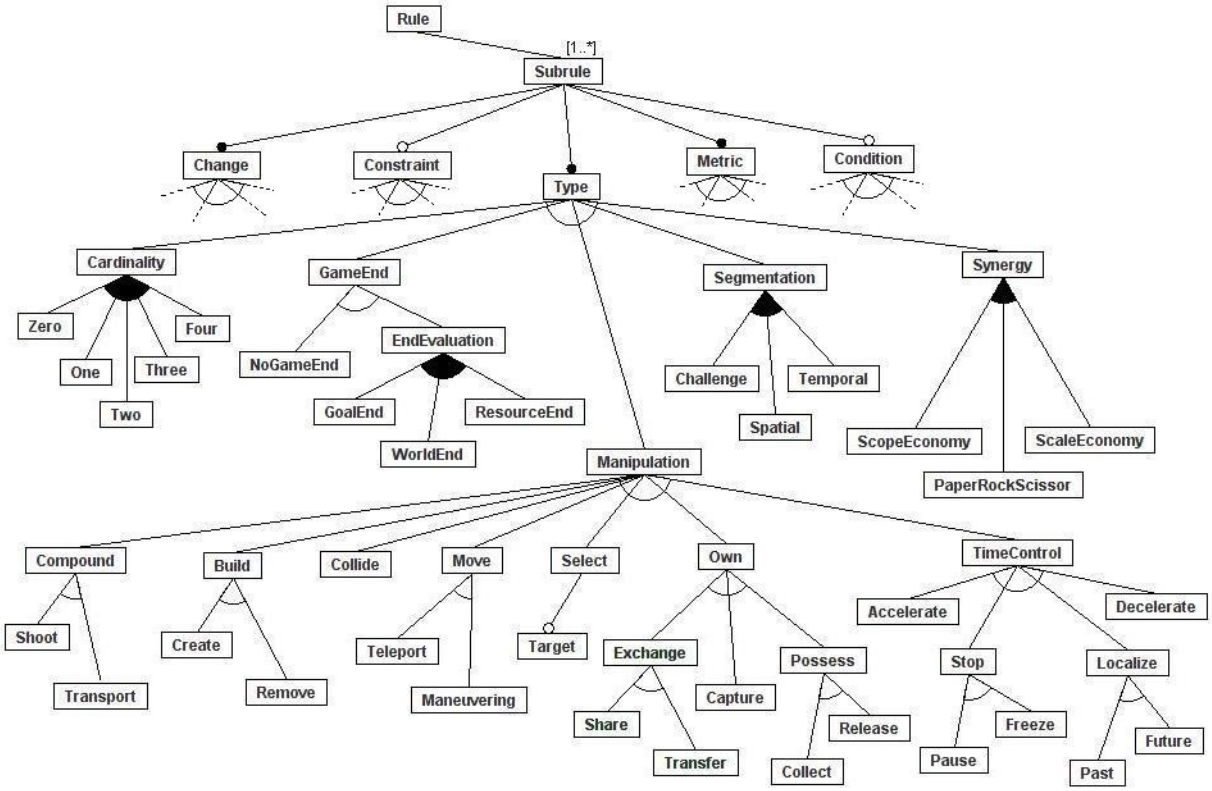


Figure 4: Rule feature diagram.

rules), Type (different kinds of goals in a game), Metric (modes of evaluation of a goal progress) and Condition (the goal is Prevent a goal completude, maintain a Continuous state for a period to conclude the goal [Björk and Holopainen 2005]).

Similar to Goal, a Rule can be considered as a game condition, a game constraint which must be validated in the game execution to perform game entity manipulations.

Figure 4 shows the different types of rules in a game, which are subdivided in: Cardinality, GameEnd, Manipulation, Segmentation and Synergy. SubRule, Change, Constraint, Metric and Condition, similar to Goal feature (with the same behaviour), are subfeatures of the Rule feature too.

According Zagal et al. [2005], the Cardinality of gameplay refers to the "degrees of freedom the player has with respects to movement (or the control of movement) in a certain game". Zero, One, Two, Three and Four cardinalities are proposed in the feature model, that can be found, respectively, in the following game examples: Guitar Hero, Space Invaders, Pac-Man, Quake and Neko.

GameEnd describes some ways to determine if a game ended or not. GoalEnd (goal completed), ResourceEnd (consuption of all resources), WorldEnd (end of the world was achieved), and NoGameEnd (Sims, Second Life, for example) are shown in the game feature model, acting as initial proposals to finish the game play, and allowing new different possibilities to determine the game end.

Manipulation validates attempts to change the context of entities in the game. Create, Share, Capture, Target, Shoot, Teleport, and others can be considered as a Manipulation feature.

According Zagal et al. [2005], the Segmentation of gameplay describes "how a game is broken down into smaller/shorter elements or chunks of gameplay". One Segmentation rule may present simultaneously: Challenge (missions, boss stage), Spatial (sectors in a build) and Temporal (time limit to explore a sector) features.

Synergy defines the relationships between rules, describing strategic options available to players [Zagal et al. 2005]. ScaleEconomy, ScopeEconomy and PaperRockScissor describes rules constraints for quantity, quality and transitive effects between entities, respectively.

3.4 Entertainment Features

Composed by Immersion and Theme features, the Entertainment feature define entities to response the following questions: What makes a game fun? What features are necessary to build a funny game?

For Immersion, Björk and Holopainen [2005] describe that "games require the player attention, and as such can make players focus on gameplay to the extent that they feel immersed in the games". This Immersion can take many forms, depending on "what type of activity the players are performing, putting them deeply focused on the interaction within the game".

The Immersion feature (Figure 5) in the NESI model proposes four types of immersion features: Cognitive, Emotional, Spatial and Challenge.

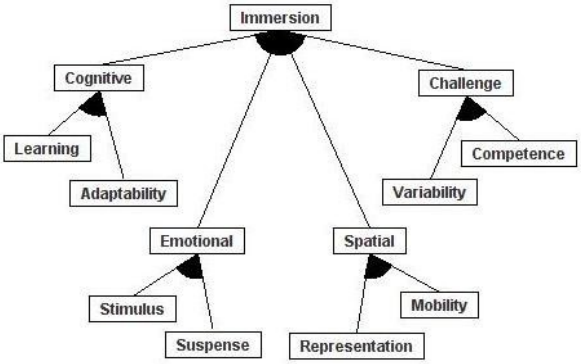


Figure 5: Immersion feature diagram.

Cognitive presents Learning opportunities to promote memorizing in a constructive play for puzzle solving, and Adaptability to reach the right level of complexity in a game [Björk and Holopainen 2005].

Emotional describes "Suspense as modulation of hopes and fears through elements of uncertainty", and Stimulus because "games privilege so-called prospect-based emotions that are always focusing on events and their outcomes" [Järvinen 2007].

Spatial presents a real and contextualized game entity Representation followed by the player Mobility inside them [Björk and Holopainen 2005].

In *Challenge*, a game must explore the player Competence, offering a game Variability by new and continuously player adventures [Lemay 2007].

According Järvinen [2007], a Theme is the subject matter of the game, is a collection of functions to represent a metaphor for the system and the proposed rule set.

In the NESI model, a Theme feature (Figure 6) is basically composed by Avatars and a Story (which is optional). A Story is a collection of Acts with a time Momentum (story Beginning, Middle or End) for each one [Bates and Bates 2004]. Each Act is formed by one or more Types, which can be a ScriptEvent (a dynamic Act based on the player choices during the Story course), a CutScene, a BackStory, a Dialog or a Surprise (as result of Secrets or SideEffects during the game).

An Avatar is an actor in the game, and is subdivided in Agent ("game entities with intentional behavior" [Zagal et al. 2005]) and Player ("those who play, in various formations and with various motivations, by performing game mechanics in order to attain goals" [Järvinen 2007]). Each Avatar has many Skills (Luck, Dextery and others) independent if it is a Player or an Agent. According Björk and Holopainen [2005], Skills are the "numerical representation of how likely an avatar is to succeed with an action, and what possible consequences the action has".

An Agent can also represent a Mule ("character that is set, typically by using scripts, to perform long, monotonous and specialized sets of actions" [Björk and Holopainen 2005]) or an Enemy ("character that hinder the players trying to complete the goals" [Björk and Holopainen 2005]), and an Enemy could be a Boss ("a more powerful enemy the players have to overcome to reach certain goals in the game" [Björk and Holopainen 2005]).

3.5 Simulation Features

According Narayanasamy et al. [2006], "the advanced computational capabilities in modern personal computers have made it possible for consumers to experience simulations with a high degree of verisimilitude through simulation games (a.k.a. Sims)".

A good example is the serious games category, a result of applying simulation technology to nonentertainment purposes (mostly training), contributing to the technology exchange between simulation games and simulators [Narayanasamy et al. 2006].

Other game categories uses simulation concepts too, when they abstract real life entities and concepts to gain more veracity in these game contexts.

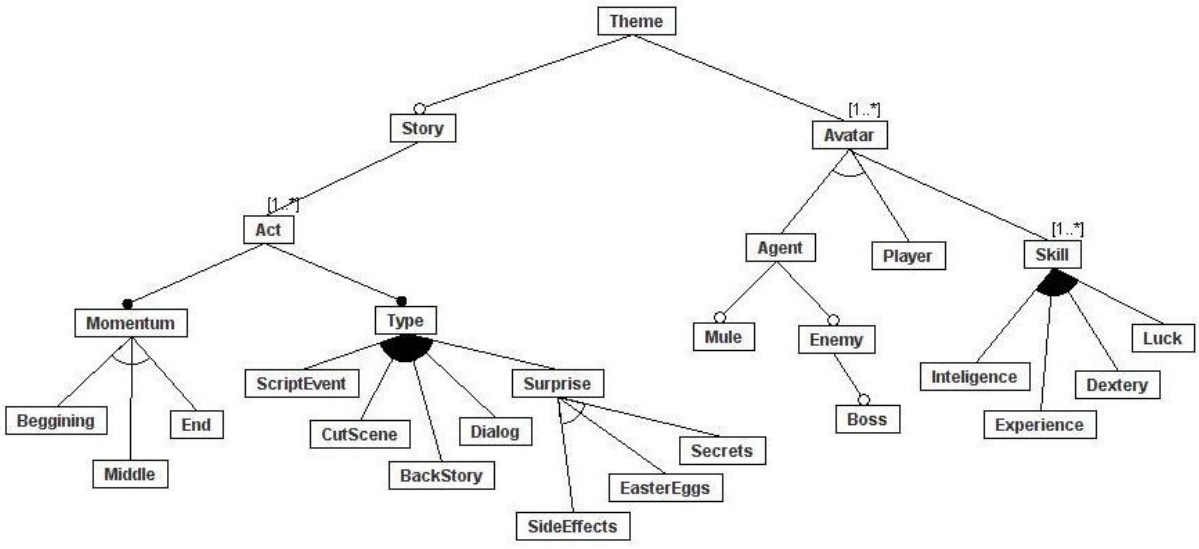


Figure 6: Theme feature diagram.

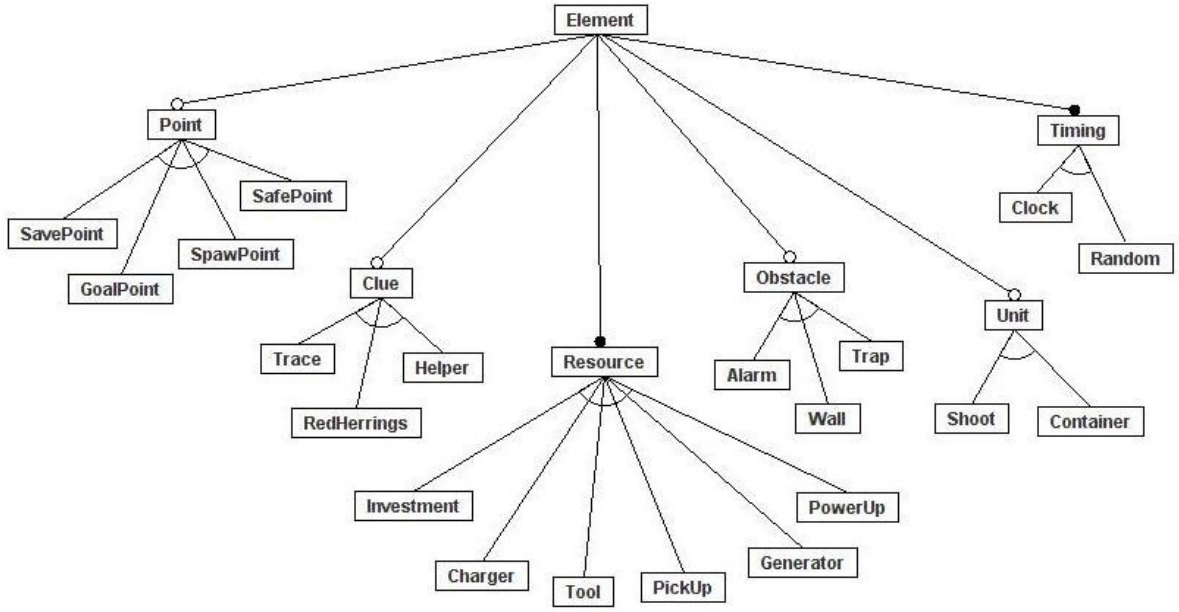


Figure 7: Element feature diagram.

So, is impossible to think in a game without simulation features. For this purpose, the NESI model presents a Simulation feature, composed by Element, Environment and Relationship main features.

The Element feature (Figure 7) represents the structural game aspects, and contains the following features: Point, Clue, Resource, Obstacle, Unit and Timing.

Points are "specific places or moments in a game" [Zagal et al. 2005]. The NESI model proposes some Points to save the game (SavePoint), to stay in secure mode (SafePoint), to complete a game goal (GoalPoint), and to generate new game entities (SpawPoint).

Clues are "game elements that give the players information about how the goals of the game can be reached" [Björk and Holopainen 2005]. For this, Traces (soldiers traces on a terrain, for example), RedHerrings and Helpers (villagers and their stories, for example) features are proposed. Bates and Bates [2004] disagrees with the inclusion of RedHerrings in a game, because they could harm in some moment the player Entertainment, and the player already has enough problems to worry about during the game. However, it is a characteristic that can be implemented

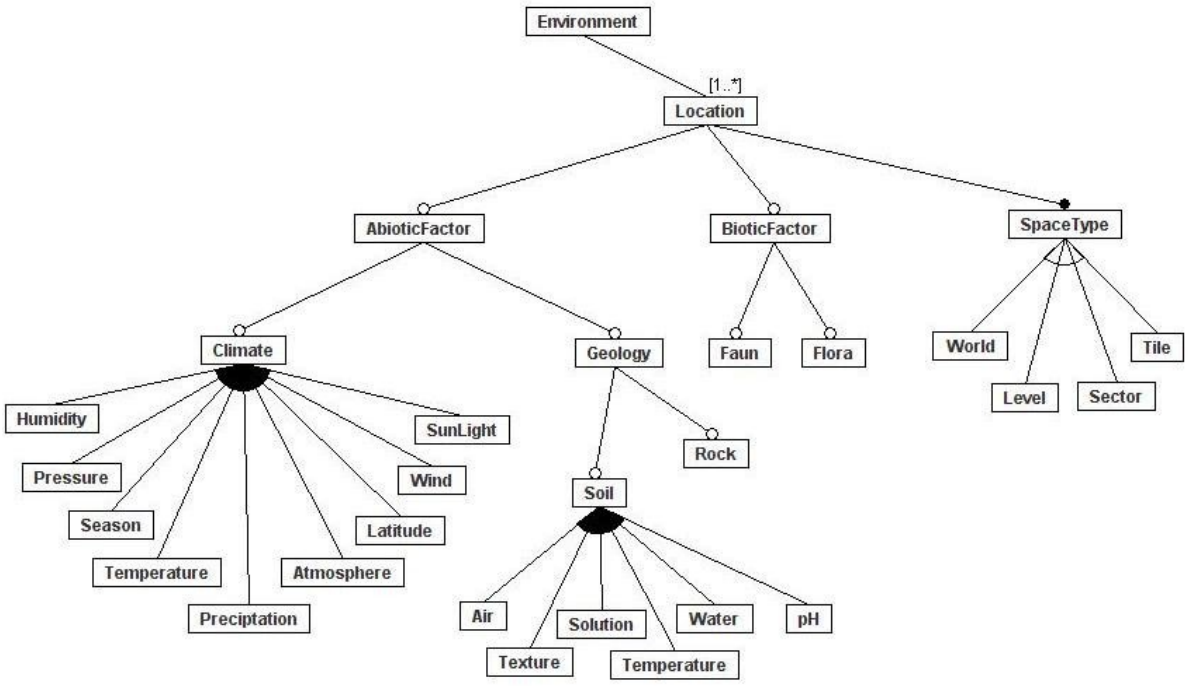


Figure 8: Environment feature diagram.

in the games, so it should be described in the proposed model.

Resources are "game elements that are used by players to enable actions in a game" [Björk and Holopainen 2005]. Some proposed resources are: Investiment (in order to reap the rewards later), Charger (speed boosters for example), Tools (weapons, armor, gadgets for example), PickUp (elements that can be collected by players), Generator (mechanisms that bring game elements to the game) and PowerUp (power pill in Pac-Man for example).

Obstacles are "game elements that hinder the players from taking the shortest route between two places" [Björk and Holopainen 2005]. Alarms, Walls and Traps are some possible and proposed Obstacle features.

Units are "groups of game elements under the player's control that let the player perform actions to influence the Game World" [Björk and Holopainen 2005]. Shoot (performs multiple actions during the release and collision of the projectile object) and Container (game element that can store other game elements) features are defined.

Timing are game elements to manage time events which can or cannot be exactly predicted in the game. Clock (synchronous events) and Random (asynchronous events) features are proposed.

The Environment feature (Figure 8), representing the game space for play [Järvinen 2007], is composed by many Locations. Each Location can take a different SpaceType (World, Level, Sector, Tile and others), and could represent different Biotic and Abiotic factors available on specific game Locations.

For AbioticFactors, Climate (with Humidity, Pressure, Temperature, Precipitation and others) and Geology (with Soil and your parameters: pH, Air, Water, Texture, etc.; and Rocks) aspects are the most outstanding features.

For BioticFactors, Faun (types of species) and Flora (types of vegetation) features in a location, with its rich diversity, can be represented.

Relationship feature (Figure 9) defines the interaction type between Element-Element, Element-Environment and Environment-Element. For that, Biological (essential for life support) and Social (essential for communities support) interactions are expected.

Biological features describes: Consumption (Parasitism, Predation, Herbivory and others), Competition (struggle to achieve a certain goal) and Cooperation (in a biological level, like Commensalism and Mutualism) features.

Social features describes: Communication (traditional chat between players), Collaboration ("players cooperate to reach goals or subgoals of the game" [Björk and Holopainen 2005]), Solidarity (players share Penalties and Rewards between them) and Negotiation ("a situation where the players confer with each other in order to reach an agreement or settlement" [Björk and Holopainen 2005]).

It is important to reinforce the game designer responsibility to choice the Relationship type between Elements and Environment, due to conceptual Relationship similarities that could confuse the designer (Cooperation and Collaboration for example).

3.6 Interaction Features

According Zagal et al. [2005], to create an interactive experience in games, it must provide interesting input and output resources.

Trinta [2007] describes interaction as "how the player performs its actions in the virtual world, which is also dependent on the context of implementation and the perception of the player at any given time".

The NESI model proposes three features to achive the desired game Interaction: Presentation (managing sensory outputs for the player), Control (managing players actions) and GamePlay (managing all created GameSessions necessary to start the GamePlay).

Zagal et al. [2005] described that games present themselves to the player through the senses, through multiple sensory channels or focused on only one.

In this way, Presentation feature is composed by three sensory features: Visual (most important Presentation game characteristic), Aural (representing every sound that can be reproduced during the game) and Tactil (with few game domain studies).

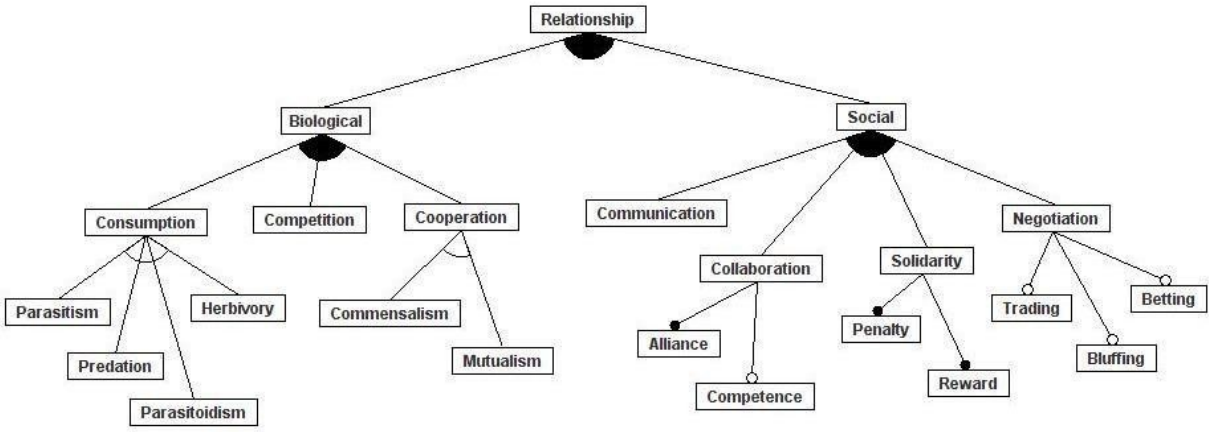


Figure 9: Relationship feature diagram.

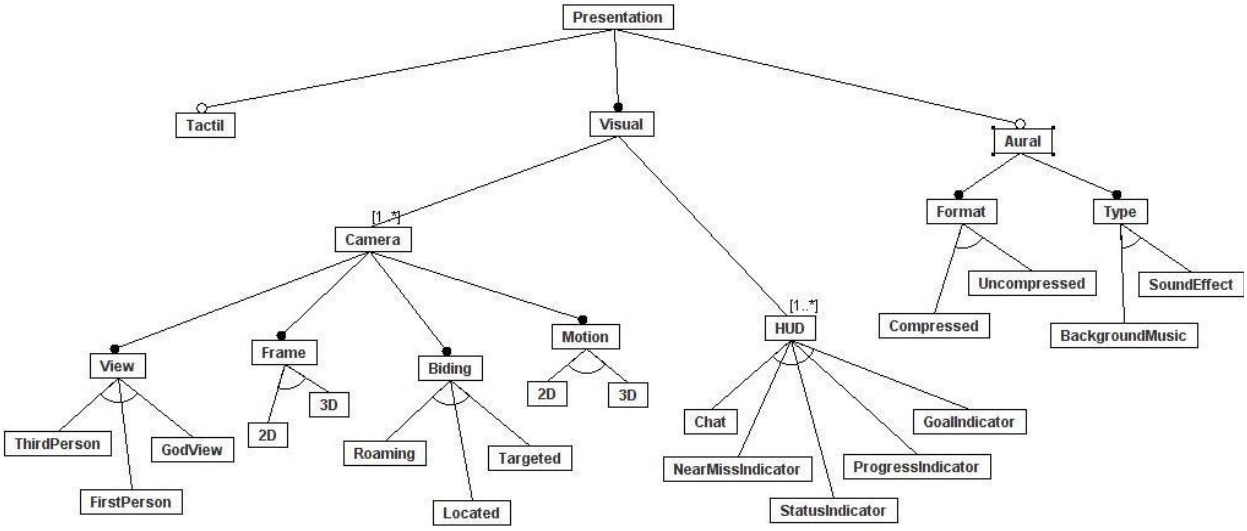


Figure 10: Presentation feature diagram.

For the NESI model, the Visual game aspect is formed by a combination of Cameras and HUDs (Head-Up Displays) features.

According Zagal et al. [2005], a Camera is "a metaphor to describe a graphical representation of the game world in which the world, the entities within it and their spatial relationships are depicted visually".

To represent this metaphor, Camera feature is composed by the following features: View (perspective from which the player perceive and interact with the space), Frame (how the camera represents the space), Biding (maintain continuity in the view), and Motion (how the camera moves within or in relation to the represented space) [Zagal et al. 2005].

The HUD objective is provide to the game player some information about the game in general, such as the lives indicator, health indicator, time indicator and others [Furtado 2006]. In summary, your goal is to indicate progress (ProgressIndicator), status (StatusIndicator), communication (Chat), and proximity for a goal (GoalIndicator) or to lose it (NearMissIndicator).

According Zagal et al. [2005], an Aural output varies over its constituent wavelengths, like a BackgroundMusic or SoundEffect to reflect the dynamical game state. To represent an Aural feature, a sound Type (SoundEffect or BackgroundMusic) and Format (Compressed or Uncompressed) are expected.

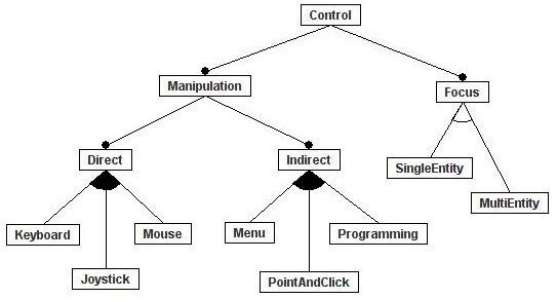


Figure 11: Control feature diagram.

For game control objectives, "a game can use multiple approaches to its control mapping as well as multiple approaches for the same control" [Zagal et al. 2005]. Indeed, some games provides to the player a level of Direct control over his Avatars actions, and other games take an Indirect control approach, using either menus or point and click interfaces [Zagal et al. 2005].

In this way, Figure 11 shows the proposed Control feature, composed by Direct and Indirect player commands, and the Focus of manipulation (Single or Multiple game entities). As shown, at least one type of command to manage the game play is expected in a game.

According Björk and Holopainen [2005], a game instance, represented as GamePlay feature (Figure 12), is the "whole lifetime of the game", and a GameSession is the "whole activity of a player playing one game".

Therefore, GamePlay contains a collection of GameSessions, which are composed by collections of Context, Execution and Customization features.

Game Context is defined as "where, when, and why the gaming encounter takes place" [Järvinen 2007]. For Trinta [2007], "all relevant information to represent the situation of an entity that composes or interacts with the application" is called Context.

In the NESI model, the game Context is a collection of entity Instances, with a Persistent status capability. Some games could include the ClosurePoints feature, "points where most of the information about game elements and actions performed become irrelevant and are discarded" [Björk and Holopainen 2005], a game point without return.

As an Instance can be of any game entity type, this feature is composed by NarrativeInstance, EntertainmentInstance, SimulationInstance and InteractionInstance features.

An Execution is equivalent to a game thread, necessary to execute actions, generated from players and multiplayers (agents or not), in a parallel way.

Each Execution has a game Policy that determines the player command execution at a given time [Trinta 2007]. TurnBased ("only one player or agent may

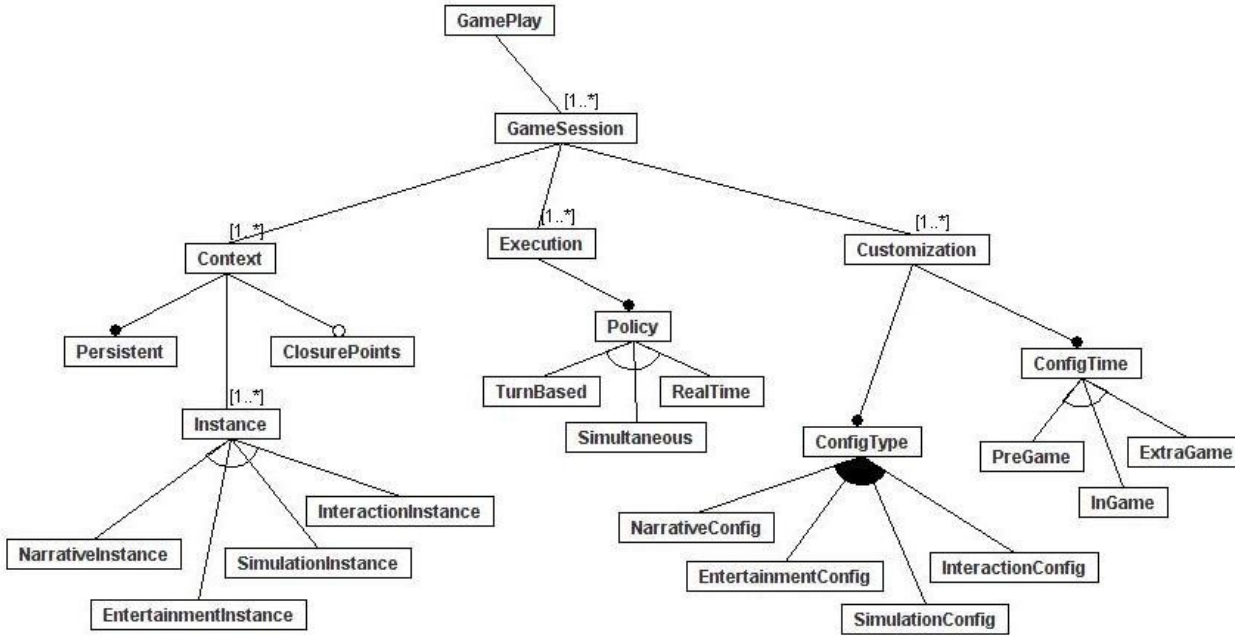


Figure 12: GamePlay feature diagram.

perform activities in the game at a given moment" [Zagal et al. 2005]), RealTime ("do not require player actions to change the game state" [Björk and Holopainen 2005]), and Simultaneous (after receive all next player commands, the game state will change [Trinta 2007]) policies are the most common used.

According Zagal et al. [2005], customize an entity means basically "the ability of the player to create or modify it according to his/her taste". This customization can be performed in a number of ways and involve several aspects of the affected entity, such as graphics, sounds, name, performance, and others.

In the NESI model, the Customization feature is composed by two features: ConfigType and ConfigTime.

ConfigType represents the customization of all game entity features. Therefore, the NarrativeConfig, EntertainmentConfig, SimulationConfig and InteractionConfig features represent all possible game entity configurations.

According Zagal et al. [2005], ConfigTime can be classified as: PreGame (not performed where the game action takes place), InGame (performed by the player where the game action actually takes place) and ExtraGame (achieved through the use of software or tools external to the game itself).

4. Conclusion

This paper presents the NESI feature model which describes game modeling aspects based on Narrative, Entertainment, Simulation and Interaction features.

As the first collection of feature models for generic games design, the great importance of this work is reduce the gap between generic game domain descriptions and specific game project implementations. This goal is achieved organizing different game domais studies in a more formal, and domain level, design approach.

Using the NESI model, the diversity and variability involved in the game modeling, and treated by the game designer, can be better controlled.

This is the result of the simplification of the game domain to four main feature aspects, allowing a more objective decision about what feature aspects a final game project will contain.

A more systematic and simplified game reuse approach is obtained too, because the NESI model can be used to guide the decomposition of existing games (creating new core assets for future game projects), and to develop new games from the ground up (configuring available core assets that implement game features).

The NESI model has been thought to be independent of graphics, audio, physics, artificial inteligence, or other type of game engine. It focus on the game domain, on the g-factor (game factor) inside the game project [BinSubaih and Maddock 2008].

For future works, a game product line and a game domain engine, based on the NESI model, and the integration of these techniques to some existent game engines, using a game software bus proposal, will be developed. Case studies showing how the model could be used in the decomposition of existing games and in the development of new game projects will be described too.

References

ALVES, V. R., 2007. *Implementing Software Product Line Adoption Strategies*. PhD thesis, Federal University of Pernambuco.

ANTKIEWICZ, M. AND CZARNECKI, K., 2004. FeaturePlugin: feature modeling plug-in for Eclipse. *In: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, October 24th to 28th, 2004, Vancouver, Canada, pp 67-72.* New York, USA, ACM Press.

BATES, R. AND BATES, B., 2004. Game Design, Second Edition. *Thomson Course Technology*, ISBN: 1592004938, 9781592004935.

BINSUBAIH, A. AND MADDOCK, S., 2008. Game Portability Using a Service-Oriented Approach. *International Journal of Computer Games Technology, Volume 2008, Article ID 378485, 7 pages*. Hindawi Publishing Corporation, doi:10.1155/2008/378485.

BJÖRK, S. AND HOLOPAINEN, J., 2005. Patterns in Game Design. Charles River Media, ISBN: 1-58450-354-8.

BOSCH, J., 2002. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. *In: Proceedings of the Second Conference Software Product Line Conference (SPLC2), pp. 257-271.*

BROWN, R., LIM, A., WONG, Y., HENG, S. AND WALLACE, D., 2006. Gameplay Workflow: a Distributed Game Control Approach. *In: CyberGames '06: Proceedings of the 2006 international conference on Game research and development, pp. 207-214.* Australia, Perth, Murdoch University, ISBN:86905-901-7.

CZARNECKI, K. AND EISENECKER, U. W., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley.

CZARNECKI, K., HELSEN, S. AND EISENECKER, U. W., 2005. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice, 10(1):7–29, jan/mar.*

DAVIDSSON, O., PEITZ, J. AND BJÖRK, S., 2004. Game Design Patterns for Mobile Games. *Project report to Nokia Research Center, Finland.* Available from: http://procyon.lunarpages.com/~gamed3/docs/Game_Design_Patterns_for_Mobile_Games.pdf [Accessed 05 August 2008].

ESPOSITO, N., 2005. A Short and Simple Definition of What a Videogame Is. *In: Proceedings of the Digital Interactive Games Research Association Conference (DiGRA 2005), Vancouver B.C., June, 2005, pp. ?-?.*

FURTADO, A. W. B., 2006. Defining and Using Ontologies as Input for Game Software Factories. *In: Proceedings of the 3rd Brazilian Symposium on Computer Games and Digital Entertainment.*

GURP, J. V., SVAHNBERG, M.. AND BOSCH, J., 2001. On the Notion of Variability in Software Product Lines. *In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), August 2001, pp. 45-55.*

HUNICKE, R., LEBLANC, M., AND ZUBECK, R., 2004. MDA: A formal approach to game design and game research. *In: Proceedings of the AAAI-04 Workshop on Challenges in Game AI, July 2004., pp. 1-5.*

JÄRVINEN, A., 2007. Introducing Applied Ludology: Hands-on Methods for Game Studies. *In: Proceedings of the DiGRA 2007 Situated Play. International Conference of the Digital Games Research Association, September 24th to 28th, 2007, Tokyo, Japan, pp 134-144.*

JUNIOR, E. A. O., GIMENES, I. M. S., HUZITA, E. H. M. AND MALDONADO, J. C., 2005. A Variability Management Process for Software Product Lines. *In: CASCON '05: Proceedings of the conference of the Centre for Advanced Studies on Collaborative research, Toranto, Ontario, Canada, pp 225-241.* IBM Press.

KANG, K., COHEN, S., HESS, J., NOVAK, W. AND PETERSON, S., 1990. Feature-Oriented Domain Analysis (FODA): Feasibility Study. *CMU/SEI-90-TR-21, SEI, USA.*

KENT, S. L., 2001. The Ultimate History of Video Games. *Tree Rivers Press*, ISBN:0761536434.

LEMAY, P., 2007. Developing a pattern language for flow experiences in video games. *In: Proceedings of the DiGRA 2007 Situated Play. International Conference of the Digital Games Research Association, September 24th to 28th, 2007, Tokyo, Japan.*

LEWIS, M. AND JACOBSON, J., 2002. Games engines in scientific research. *Communications of the ACM, vol. 45, no. 1, pp. 27–31.*

NARAYANASAMY, V., WONG, K. W., FUNG, C. C. AND RAI, S., 2006. Distinguishing Games and Simulation Games from Simulators. *ACM Computers in Entertainment, Vol. 4, No. 2, April-June 2006, pp. 1-18*. New York: ACM Press, ISSN:1544-3574.

ROLLLINGS, A. AND MORRIS, D., 2004. Game Architecture and Design : A New Edition. *New Riders Publishing*, ISBN: 0735713634.

SVAHNBERG, M., GURP, J. V. AND BOSCH, J., 2005. A Taxonomy of Variability Realization Techniques. *Software—Practice & Experience, Volume 35, Issue 8 (July 2005), 705-754*. New York: John Wiley & Sons, ISSN:0038-0644.

TAYLOR, M. J. GRESTY, D. AND BASKETT, M., 2006. Computer game-flow design. *ACM Computers in Entertainment, Vol. 4, No. 1, January 2006, pp. 5*. New York: ACM Press, ISSN:1544-3574.

TRINTA, F. A. M., 2007. *Definindo e Provendo Serviços de Suporte à Jogos Multiusuário e Multiplataforma: Rumo à Pervasividade*. PhD thesis, Federal University of Pernambuco.

ZAGAL, J., MATEAS, M., FERNANDEZ-VARA, C., HOCHHALTER, B. AND LICHTI, N., 2005. Towards an Ontological Language for Game Analysis. *In: Proceedings of the Digital Interactive Games Research Association Conference (DiGRA 2005), Vancouver B.C., June, 2005, pp. 3-14.*

ZHANG, W. AND JARZABEK, S., 2005. Reuse without Compromising Performance: Experience from RPG Software Product Line for Mobile Devices. *In: Proceedings of the 9th Int. Software Product Line Conf., SPLC'05, Sept. 2005, Rennes, France, pp. 57-69.*

Fast and Safe Prototyping of Game Objects with Dependency Injection

Erick B. Passos
Media Lab - UFF

Jonhnny Weslley S. Sousa
LCD - UFCG

Giancarlo Nascimento
Media Lab - UFF

Esteban Walter Gonzales Clua
Media Lab - UFF

Lauro Kozovits
UERJ

Abstract

Most game engines are based on game objects inheritance and/or componentization of behaviors. While this approach enables a clear visualization of the system architecture, good code reuse and fast prototyping, it brings some issues, mostly related to the high dependency between game objects/components instances. This dependency often leads to static casts and null pointer references that are difficult to debug. In this paper we propose the use of the Dependency Injection design pattern to safely initialize game objects and alleviate the role of the programmer in the handling of these issues both during prototyping and production fases. Since these dependencies are attributes in game objects and the injection occurs only at the initialization pass, there is no performance penalty at the game loop.

Keywords:: game engine architecture, dependency injection, object composition

Author's Contact:

{epassos,esteban}@ic.uff.br
jonhnny@lsd.ufcg.edu.br
giancarlotaveira@gmail.com
lauro@jogos.etc.br

1 Introduction

Few domains in computer science map to a programming paradigm so directly as computer games to object orientation. A *Game* class matches the concept of a virtual world where several different instances of a *GameObject* class reside. A *GameObject* instance can be anything such as the player character, a house or an invisible trigger object. The game execution usually consists of a loop inside the *Game* class with three main goals:

1. Collect user and network input;
2. Update each GameObject instance based on input and/or physics simulation, animation and Artificial Intelligence step;
3. Draw visible game objects on the output device.

There are alternative forms of this loop proposed to better exploit parallelism [de Moraes Zamith et al. 2007], but we consider that this basic model serves well to express the purposes of our work.

To represent different types of simulated objects, the programmer usually creates subclasses of *GameObject*, each one specifying new and more specialized content and behavior. The problems with the inheritance approach are well known. A good example of this is when two objects from different hierarchies sometimes have common functionality and characteristics, leading to redundancies in code. This is a common issue in object oriented software design and replacing inheritance by composition in game engines is already recognized as a good practice [Folmer 2007; Stoy 2006; Ponder 2004; Billas 2002].

In its componentized alternative, the GameObject class holds only common attributes that all game objects should have such as name, position and orientation. Most important, however, is that it acts as a container for reusable components. Each class extending *AbstractComponent* represents a different aspect/behavior of a *GameObject* such as physics, AI or health, depending on the needs of the game object being (now) composed. These components should be highly flexible and easier to maintain while also maximize code reuse as many of them are useful in several different game genres.

Both approaches have a common issue, related to high coupling between components (or game objects). For instance, an AI component commonly depends on the existence of a health component to decide actions to take and also on a physics component to apply movements to. These dependencies are normally resolved directly by the programmer, as shown in the following Java code, part of an AIComponent class, adapted from the original C++ example [Stoy 2006]:

```
1 public void update(float interpolation) {
2  final GameObject o = getOwner();
3  Health h = (Health) o.getComponent("health");
4  if (h != null) {
5    // take AI actions based on health
6  }
7 }
```

Code 1: *Traditional dependency handling*

Its easy to see that this implementation has an implicit dependency on the existence of a *Health* component (line 3), which is expected to be registered at the same game object under the label "health". If every game object containing an instance of this *AIComponent* class is properly initialized with an accompanying *Health* component, everything will work as planned by the programmer. Apparently there is nothing wrong with this code, but a closer inspection shows us that:

- While necessary for solving the implicit dependency, lines 2-4 have nothing to do with the expected game logic of an AI update, being just boilerplate code;
- The explicit cast in line 3, or the absence of strong typing in the case of some script languages, is a common source of runtime problems in dependent component/objects;
- If no *Health* component is initialized for this particular game object, no proper AI action (code inside if clause) will ever be taken, making it harder to debug while doing level design (unless the component logs the unfound dependencies, something the programmer would have to code directly).

An equivalent code in UnrealScript [EpicGames 1998] would be even more difficult to debug because the language hides all null pointer references by making them equivalent to a void execution line. This is such an issue that Tim Sweeney have recently said that around fifty percent (50%) of the bugs found in Unreal were related to this kind of dependencies and the lack of stronger typing in its programming languages [Sweeney 2006]. He also points out that a typical game object update usually touches five to ten other objects, which shows how relevant and frequent the problem is.

In this paper we propose the use of the Dependency Injection design pattern [Jin 2007; Fowler 2004] to solve part of this problem by freeing the programmer from the responsibility of manually checking for these dependencies. As will be shown in the next sections, our framework takes care of the safe initialization of game objects and components, which can be coded in a much cleaner and maintainable fashion. The rest of the paper is organized as follows: section 2 discusses related work, section 3 presents the concepts and design patterns implemented by the GCore framework while section 4 explains the use of Dependency Injection and the advantages of our framework over previous research. Finally, section 5 concludes the paper and outlines future work.

2 Related Work

Successful Game engines from the industry are strongly based on the game object inheritance model such as CryEngine [CryTek 2008] with its entities and entity-items being equivalent to game objects and components respectively. The same architecture is found in other commercially available engines such as UnrealEngine [EpicGames 1998] and Torque [GarageGames]. At the same time there are more consistent componentized architectures [UnityTechnologies 2008; Spinor ; 3DVia ; Billas 2002]. The problems related to high dependency between objects are a common issue of these tools, making them potential targets to our proposal.

In a talk presented by Tim Sweeney [Sweeney 2006], he exposed in detail two problems with current game programming languages and tools: poor concurrency handling and weak typing. He proposes some features that a new programming language should have that could solve most of the runtime bugs programmers deal with when implementing game object scripts. While his arguments are somehow similar to ours, he proposes the creation of new languages with the suggested features, which is not a simple task. In this paper we are proposing the use of a currently available technique that can be adapted in most existing tools. Our Dependency Injection implementation is based on Java reflection features, being trivial to port to C# based platforms such as XNA [Microsoft]. C++ and some script languages doesn't have the exact reflection features used by our framework but it is possible to implement the same idea with some workarounds [Pocomatic 2007].

Dungeon Siege was one of the first games to include a fully componentized game object system. In two different years at the Game Developers Conference [Billas 2002; Billas 2003], Scott Billas showed how this architecture and some other features helped the development of the game and its "continuous world". In a more recent talk [Billas 2007], he exposed ideas on how to improve a game production pipeline, several of them being related to sanity checks during game objects/components initialization such as attribute requirements and dependencies. He proposes that these assertions and error messages should be implemented directly by the component programmer/engineer on the behalf of the level designer. We recognize that these ideas are very important indeed, but also aim to provide tools to automatically perform these sanity checks and also dependency solving/injection, this time on the behalf of the programmer/engineer and consequently improving the whole production pipeline.

Haller et. al. [Haller et al. 2002] proposes a new architecture for game objects and components using communication slots and a message manager to remove the strongly typed dependencies between them. The solution enables easy composition and connection of components but requires a strong commitment to a more complex architecture. With our approach the programmer doesn't have to learn any new language or communication architecture, making it more suitable to fast prototyping.

The Unity3D game engine [UnityTechnologies 2008] is a relatively recent product that has gained attention from developers given its well designed game object component system and scene editor, which uses a visual approach to composition. In its latest version (2.1, released in late July, 2008), a simplified form of dependency sanity checking is provided for the script programmers, who can specify that a component has a dependency on the existence of another, which is checked at runtime and also inside the scene editor. However, although automaticaly instantiated by the scene editor, this instance is not automaticaly injected in the dependent component, leaving this responsibility to the programmer, who still has to explicitely call a *getComponent(Type)* method to obtain the reference. Our system does both the sanity checking and the automatic injection (reference solving) of dependent components.

To our knowledge, all previous research and/or products only go so far in the development of game object component systems. We propose the full adoption of Dependency Injection to handle the coupling of components in a game engine. In the following sections, our framework, named GCore, will have its architecture explained together with the application of Dependency Injection in game objects composition.

3 GCore Architecture

GCore, abbreviation for Game Core, is a data-driven game framework aimed at high productivity that abstracts the use of JMonkeyEngine [JMonkeyEngine], a Java scene-graph engine implemented over hardware-accelerated OpenGL for graphics and OpenAL for audio. GCore also includes add-ons such as a physics engine [JMEPhysics] and a network subsystem [Imagination] to deliver an extensible and easy to use tool. Since its core concepts are feasible to implement in other platforms and languages such as C++ or C#, we focus our attention on its data-driven approach and Dependency Injection features, which enable game designers, level designers, programmers and artists to cooperate seamlessly in a game production pipeline. The role of the programmer is to code reusable components implementing the game design ideas while the level designer integrate these components with the assets created by artists. In this paper we show GCore tools and techniques to assist both programmers and level designers.

3.1 Main Concepts

From a software engineering perspective, GCore defines four main concepts/classes to represent a game: GameManager, GameState, GameObject and AbstractComponent. GameManager is a Facade [Gamma et al. 1995] for the whole system while a GameState is similar to a use-case scenario, each one isolating a game scene (or other user interaction concept) such as the 3D playing environment, a menu or a heads up display (HUD). During execution, the player alternates through these scenarios as the game switches between the different instances of GameState. A runnable GameState is defined with an unique name and containing a collection of GameObject instances, each one composed by a collection of AbstractComponent implementations. In Figure 1 it is possible to see a simple class diagram for GCore architecture while in Figure 2 one can see the sequence diagram that explains the game loop concept used in GCore by specifying the order of method calls at each discrete step.

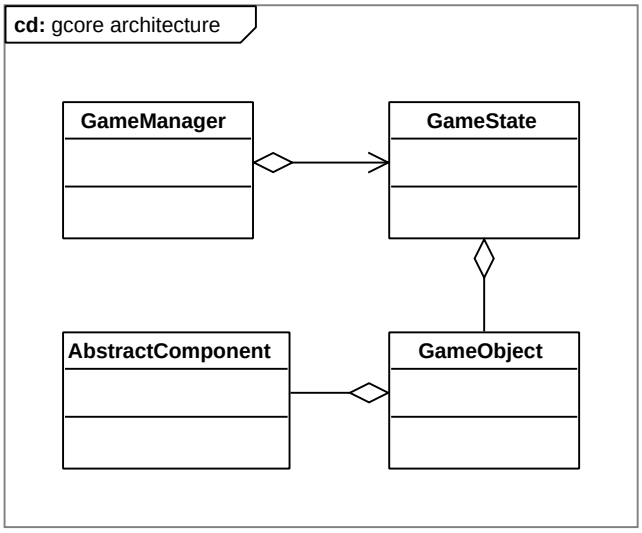


Figure 1: *GCore componentized architecture*

As can be seen from the diagrams, each component is updated at every frame. There is no render method in the AbstractComponent class because most of them do not represent graphical properties of the object. Instead, the GameObject class keeps a scene-graph node where any graphical component can attach a drawable geometry if needed. This node is rendered at the end of each frame step which leads to the rendering of the geometries of graphical components attached to the game object.

3.2 Data-driven Composition of Game Objects

The main reason for GCore's productivity is the possibility of creating a game entirely in declarative form. An example of compo-

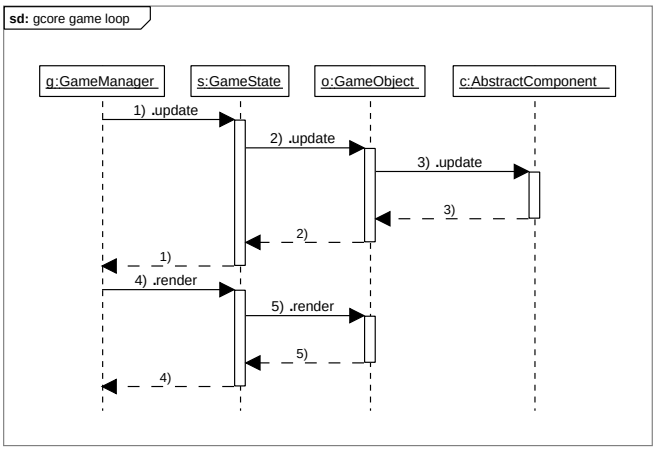


Figure 2: *GCore game loop*

sition is shown in Figure 3, an object diagram where a GameState is composed of two GameObject instances, each one with different components representing their characteristics and behaviors.

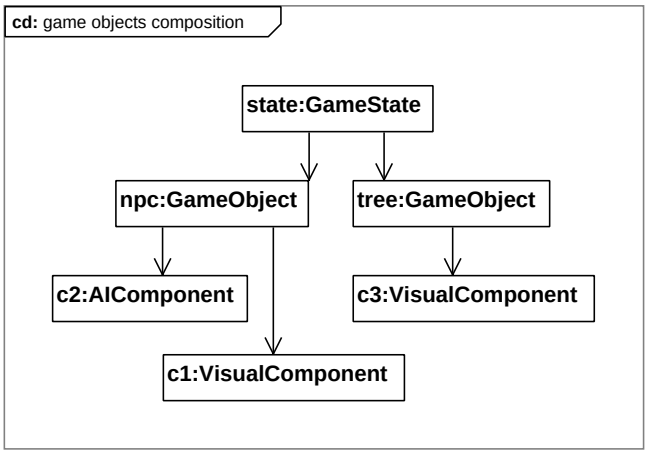


Figure 3: *Composition example*

In the above example, there are two active objects in the game state named ”npc” and ”tree” respectively. The ”npc” object is composed of an instance of VisualComponent carrying its graphical representation and an AIComponent, responsible for controlling its actions during game execution. Notice that since the update method is called at every frame step, both components are updated. In the AIComponent this method contains the actual implementation of the AI step, while in the VisualComponent this method is empty. The other game object, named ”tree”, is static and consists only of a graphical geometry, represented by a single VisualComponent.

Game states and their companion game objects are all stored in XML files, which are very easy to maintain and also suitable for an integration tool such as a level editor. In Code 2 one can see the main XML configuration file of a simple game. The root element of the configuration is the ”game” element, which has two attributes: a name and the first game state to be loaded. The ”game” element must be composed of type declarations and game state compositions, being possible to keep them in as many separate files as desired, feature exemplified by the use of the ”include” tag.

Type declarations provide for a way to compose reusable types that can be further specialized and instantiated into game objects inside the game states. Types are composed of components which can have their attributes values also specified inside the XML files. In Code 3 one can see a type declaration showing the concepts of composition, inheritance and attribute specification. The first type, named ”basic”, defines that all derived types and objects will have a VisualComponent attached. The second one, named ”npc”, exemplifies inheritance by extending ”basic”, from which it brings the VisualComponent and also attaches a new AIComponent. The ”tree” type shows the specification capabilities of GCore by inheriting from ”basic” and modifying an attribute in the VisualComponent, in this case defining an external 3D model to be loaded as geometry for any ”tree” typed game object.

```
<game name="example" init="menu">

    <!-- type definitions -->
    <include file="types.xml" />

    <!-- game states -->
    <include file="menu.xml" />
    <include file="farm.xml" />

</game>
```

Code 2: *Main configuration file (game.xml)*

```
<type name="basic">
    <component class="VisualComponent" />
</type>

<type name="npc-type" extends="basic">
    <component class="AIComponent" />
</type>

<type name="tree-type" extends="basic">
    <component class="VisualComponent">
        <model value="tree.3ds" />
    </component>
</type>
```

Code 3: *Sample type declarations (types.xml)*

Game states are composed of game objects, which can inherit from pre-defined types and specify or include any component as needed. One can even define objects without a supertype, but this practice minimizes reuse and is not recommended. In Code 4 one can see the XML for the game state shown in Figure 3 in Page 3. The ”npc” game object inherits from the previously declared ”npc-type” and specifies a 3D model for its VisualComponent. The ”tree” object specifies a new position vector for its VisualComponent. One can easily notice the flexibility of this data-driven approach for game objects composition and it is even possible to have multiple named components of the same class in the same type. The classes GameObject and AbstractComponent also implement the Composite design pattern [Gamma et al. 1995], being possible to have chain of nested components if needed.

```
<gamestate name="farm">

    <!-- game object1: npc -->
    <object name="npc" type="npc-type">
        <component class="VisualComponent">
            <model value="zombie.3ds" />
        </component>
    </object>

    <!-- game object2: tree -->
    <object name="tree" type="tree-type">
        <component class="VisualComponent">
            <position x="10" z="15" />
        </component>
    </object>

</gamestate>
```

Code 4: *Game state and objects composition (farm.xml)*

3.3 XML parsing and game execution

From the last section one can see that each game is completely specified by a meta-data configuration stored in XML files. This XML specification is parsed during initialization and loaded into a set of configuration objects. This light-weight meta-data structure is used at runtime to instantiate game states and game objects as needed. GCore is capable of parsing attribute values of all Java primitive types and also 3-valued vectors, quaternions and assets (files) such as textures, models and audio clips. It is also possible to create your own parser for any user-defined structured type (class) by extending the PropertyParser utility class.

There has to be at least one default game state in a game, which will be the first (or only) to be loaded. Since a game is composed by a collection of such game states, there should be a way to instantiate, destruct or switch between them at runtime. To provide for an elegant implementation of these features, the GameManager class implements the Mediator design pattern [Gamma et al. 1995] with public methods to (re)activate, pause or destroy any declared game state by its name. The destroying of previously enabled game states is optional (prior the activation of another one) since more than one can coexist in memory at the same time.

The GameManager also take care of the correct initialization of all game objects and its components. The Builder design pattern [Gamma et al. 1995] was applied in the initialization code for game states, objects and components, taking care of attribute and dependency injection. However, for the sake of readability, in the next section we will show a simplified version of this process instead. The reason is that we will focus on the explanation of the dependency injection usage, its advantages and implementation details.

4 Dependency Injection in GCore

Dependency Injection, sometimes referred to as inversion of control, is a design pattern that provides a flexible way to indirectly assembly dependent software components together [Jin 2007; Fowler 2004]. In a complex componentized system, such as a game, it is common to find classes that depend on others to perform their tasks. The programmer usually implements this dependency as an attribute declared with its type being the target class. Dependency solving normally occurs at runtime by getting an instance using an available pull-like API.

With Dependency Injection the underlying framework is responsible for automatically look for this instance and setting it under the dependent component. There are two gains associated with the use of this pattern: smaller and cleaner code at the components, since it's not necessary to manually look for the dependency; and safer initialization, because it's a responsibility of the framework to check for the existence of the dependencies, leading to less runtime errors.

To implement this technique we make use of the runtime reflection features of the Java language, especially annotations, which are a special kind of meta-data available both at compile and runtime. We created a custom annotation type, named @Inject, to be used to mark dependencies between components in the source code. These dependencies will be solved at loading time by the game object initializer every time an attribute identified with @Inject is found. In the following section we explain two examples of Dependency Injection in our framework: the composition of a player game object showing the safe initialization of dependent components in the first example and the fast prototyping of new game mechanics in the second one.

4.1 Example 1: Safe Initialization of Components

In GCore, all direct dependencies between components can be automatically solved by the framework. First lets consider how very simple game object features can be implemented as reusable components: external 3D model loading, player input and a chase camera. It's recomended, for the sake of reusability, to implement each one as a separate component class, named VisualComponent, PlayerInput and ChaseCamera. VisualComponent can be used independent of the other two for any static object that needs a visual representation such as a house or a tree. None of these need to be followed by a camera or controlled by the player so there will be no need to include the other respective components. However, when used to compose a player object, ChaseCamera and PlayerInput are used and both have a dependency on the existence of an oriented geometry. They use this geometry respectively as a target to look at or to update based on player commands. VisualComponent already defines a geometry attribute (its loaded model) that fits this need. In Figure 4 this relation is expressed in the form of an object diagram. The player game object has a collection of components, all being concrete subclasses of AbstractComponent and having dependencies between them.

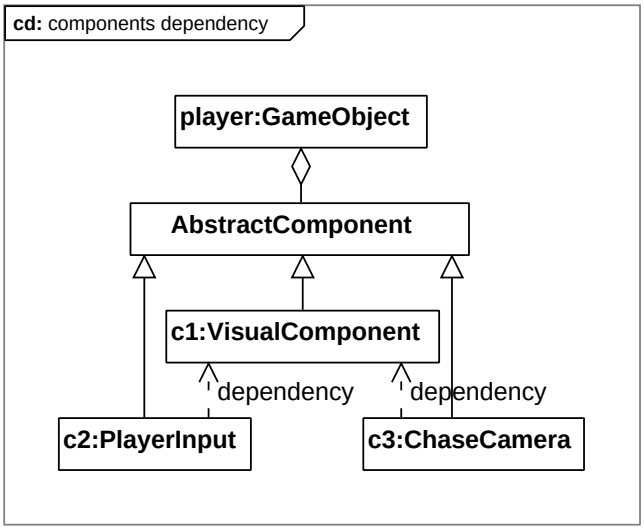


Figure 4: *Dependent components example*

Components c2 and c3 from Figure 4 show a dependency to component c1. Now it's clear that both ChaseCamera and PlayerComponent have a attribute of type VisualComponent, so they can use it at their respective update methods. Instead of manually looking for this object at each update method, the component programmer only has to include the runtime-available custom annotation @Inject to the attribute declaration as show in Code 5. When initializing each component, as will be explained next, if the annotation @Inject is found before any declared attribute, our framework looks for an instance of this component type in the same game object and sets it into the attribute.

```
class ChaseCamera extends AbstractComponent {

  @Inject
  VisualComponent vc;

  public void update(float interpolation){
    camera.lookAt(vc.getWorldTranslation());
  }
}
```

Code 5: *@Inject in ChaseCamera source code*

One can see that there is no line of code looking for the VisualComponent instance or checking its nullability. It's also easy to notice that, compared to the example in Code 1 in Page 1, the above code is smaller, considerably cleaner and also safer since the framework will stop initialization and show an error log if there is no VisualComponent declared and already initialized for the related GameObject. In Code 6 one can see a correct XML for the player object composition.

Since this declaration includes a VisualComponent, the other two components, which depend on the existence of the previous, are initialized correctly. Game object initializations occurs as shown in the sequence diagram presented in Figure 5. As shown in the

```
<object name="player">

    <component class="VisualComponent">
        <model value="knight.md5" />
    </component>

    <component class="PlayerInput" />

    <component class="ChaseCamera" />

</object>
```

Code 6: *Correct player object composition*

diagram, when creating a game state, all game objects and their components are first instantiated and their declared attributes set. After this first step (messages 2, 3 and 4 in the diagram), the game objects are properly initialized and components dependencies are injected (messages 5 and 6). By running the injection last, all declared components are already attached to the game object leading to a complete dependency solving.

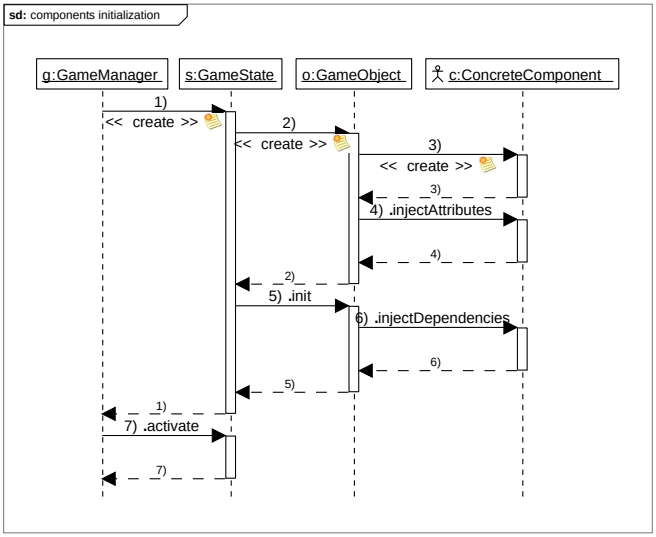


Figure 5: *Game objects initialization*

In Code 7 an incorrect player object composition is presented. Let's imagine for instance that the level designer made a mistake and thought the "character" supertype already had a VisualComponent declared and included only the (in his mind) two missing ones. Instead of leading to an unpredicted runtime error, this specification is not valid and our framework will show a message at initialization such as seen in Code 8.

```
<object name="player" type="character">

    <component class="PlayerInput" />

    <component class="ChaseCamera" />

</object>
```

Code 7: *Incorrect player object composition*

```
"Incomplete composition of object: 'player'.
    Missing required 'VisualComponent'
    needed by included 'ChaseCamera'."
```

Code 8: *Unsolved dependency initialization error*

By having automatic handling of the coupling between components and a safer initialization of game objects, the programmer will not need to manually check for explicit dependencies or their nullability. From this example one can see that the level design step in the production pipeline is improved with less dependency on code debugging tools.

4.2 Example 2: Fast Prototyping of New Game Mechanics

During the paper introduction we pointed that one of the problems caused by the dependency between game objects and components was the high number of lines of code dealing with issues not related to the core game logic being implemented. Boilerplate code like this needs a lot of concentration from the programmer and, usually, also debugging. In this example we show how Dependency Injection improves fast prototyping of components by making the programmer focus only on the core mechanics implementation.

Lets assume we are going to implement a "lunar cargo" game, where the core mechanics consists of controlling a rocket-powered heavy weight lift vehicle. The goal of the implementation is to expose the core mechanics to early tests as fast as possible. The following features are essential to the prototype:

1. A moon-like terrain;
2. Gravity and collision physics;
3. 3D model loading;
4. Player-controlled thrust to the lunar module (most important).

It is easy to notice that features 1-3 are also needed by several different game genres and are already available as reusable GCore components. The only lasting, and important, feature is the player-controlled thrust. In Code 9 one can see the complete XML used for this prototype, declaring a single game state, composed of two game objects: the terrain and the lunar module. The implementation of the missing Thrust class is explained next.

```
<game name="lunarCargo" init="moon">

  <gamestate name="moon">

    <!-- prototype object1: terrain -->
    <object name="terrain">
      <component class="TerrainComponent">
        <heightmap value="moon.png" />
      </component>
      <component class="TerrainPhysics" />
    </object>

    <!-- prototype object2: lunar module -->
    <object name="module">
      <component class="VisualComponent">
        <model value="cargo-ship.3ds" />
      </component>
      <component class="DynamicPhysics" />
      <component class="Thrust" />
    </object>

  <gamestate>

<game>
```

Code 9: *Lunar Cargo prototype XML*

GCore's DynamicPhysics component has a dependency to VisualComponent that the former uses as collision geometry and also to move as the simulation goes. The Thrust being implemented clearly has a dependency to the DynamicPhysics, which is going to have forces applied to as the user presses the "thrust" key. In Code 10 one can see the implementation of the Thrust class. The dependency to DynamicPhysics is exposed to the framework by the attribute annotated with @Inject and the update method just takes for granted that this attribute will not be null at runtime.

It is clear that the programmer could concentrate on the core mechanics implementation: checking for the user input and imposing a thrust to the lunar module physics. With this approach we believe

```
class Thrust extends AbstractComponent {

  @Inject
  DynamicPhysics physics;

  public CargoController() {
    KeyManager.set("thrust", KEY_SPACE);
  }

  public void update(float tpf) {
    if (KeyManager.isValidCommand("thrust"))
      physics.addForce(Vector3f.UNIT_Y);
  }
}
```

Code 10: *Thrust component source code*

one can write better code and achieve faster prototyping in a game production pipeline without compromising a good design.

5 Conclusion

Data driven is a proved approach to manage risk during a game production pipeline. Putting this together with a well-designed game engine, a powerful architecture is achieved, as can be shown by several successful engines and frameworks. However, object orientation, especially components composition, has some issues with maintainability of highly dependent instances. In this paper we have presented a technique based on the Dependency Injection design pattern that safely removes this task from the programmer duties.

There is a know myth in game production circles that it is impossible to fast prototype and write well-designed code at the same time. We firmly believe that by using the GCore framework one can achieve very fast prototyping without giving up most good programming practices. By using Dependency Injection we completely removed the hard-coded implicit dependencies, which are so common in game objects scripting, from GCore components library, providing a powerful, extensible and safer tool.

GCore is a constant work in progress and we are now investigating how to apply the technique to other scenarios in game design such as dependencies between a component and any attribute from other game objects and components. We plan to extend the use of annotation to enable the programmer to apply constraints, such as not-null, min/max values/lenght, to any primitive type attribute (plus files, strings, vectors and quaternions). The short-term roadmap also includes a level-editor, which provides for the visual composition of games freeing the level designer from the need to write XML specifications.

Acknowledgements

We would like to thank Scott Bilas for the meaningful discussions about game object components dependency and other relevant issues related to software engineering and current trends in game engines development.

We are also very thankful to everybody over the JMonkeyEngine discussion forums, specially the developers, always willing to help with the accurate solutions on rendering, audio and physics that made the GCore framework possible.

References

3DVIA. Virtools. http://www.virtools.com/.

BILLAS, S., 2002. A data-driven game object system. Talk at the Game Developers Conference '02.

BILLAS, S. 2003. The continuous world of dungeon siege. In *Proceedings of the Game Developers Conference '03*.

BILLAS, S., 2007. Optimizing the development pipeline - tools, technology, process. Lecture at the CGA Casual Connect Europe: West 2007.

CRYTEK, 2008. Cryengine sandbox 2 manual. http://doc.crymod.com/SandboxManual/.

DE MORAES ZAMITH, M. P., CLUA, E. W., CONCI, A., MONTENEGRO, A., PAGLIOSA, P. A., AND VALENTE, L., 2007. Parallel processing between gpu and cpu: Concepts in a game architecture.

EPICGAMES, 1998. Unrealscript language reference. http://unreal.epicgames.com/UnrealScript.htm.

FOLMER, E. 2007. Component based game development - a solution to escalating costs and expanding deadlines? In *Component Based Software Engineering*, Springer, vol. 4608 of *Lecture Notes in Computer Science*, 66–73.

FOWLER, M., 2004. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

GARAGEGAMES. Torque game engine. http://www.garagegames.com/.

HALLER, M., ZAUNER, J., AND HARTMAN, W. 2002. A generic framework for game development. In *In Proceedings of the ACM SIGGRAPH and Eurographics Campfire '02*, ACM.

IMAGINATION, C. Jgn: Java game networking. http://forum.captiveimagination.com/index.php/board,4.0.html.

JIN, K., 2007. Why and what of inversion of control. http://www.pocomatic.com/docs/whitepapers/ioc/.

JMEPHYSICS. Jmephysics: interface between jme and physics engines. https://jmephysics.dev.java.net/.

JMONKEYENGINE. Jmonkey engine 1.0 online documentation. http://www.jmonkeyengine.com/.

MICROSOFT. Microsoft xna. http://www.xna.com/.

POCOMATIC, 2007. Pococapsule/c++ ioc and dsm framework. http://www.pocomatic.com/docs/whitepapers/pococapsule-cpp/.

PONDER, M. 2004. *Component-Based Methodology and Development Framework for Virtual and Augmented Reality Systems*. PhD thesis, Ecole Polytechnique Federeale de Lausanne.

SPINOR. Shark 3d real time 3d software. http://www.shark3d.com/.

STOY, C. 2006. Game object component system. In *Game Programming Gems 6*, Charles River Media, M. Dickheiser, Ed., 393–403.

SWEENEY, T. 2006. The next mainstream programming language: a game developer's perspective. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, New York, NY, USA, 269–269.

UNITYTECHNOLOGIES, 2008. Unity3d game engine 2.1. http://unity3d.com/unity.

Supermassive Crowd Simulation on GPU based on Emergent Behavior

Erick Baptista Passos
UFF, Medialab

Mark Joselli
UFF, Medialab

Marcelo Zamith
UFF, Medialab

Jack Rocha
UFF, Medialab

Esteban Walter Gonzalez Clua
UFF, Medialab

Anselmo Montenegro
UFF, Medialab

Aura Conci
UFF, Medialab

Bruno Feijo
PUC-RIO, ICAD Games

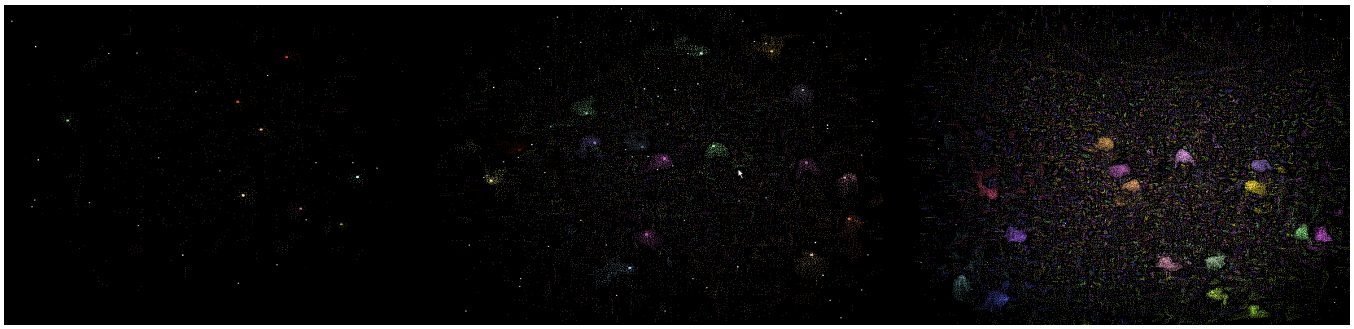


Figure 1: *Screenshots of a boids simulation with variable number of entities*

Abstract

Computing and presenting emergent crowd simulations in real-time is a computationally intensive task. This intensity mostly comes from the $O(n^2)$ complexity of the traversal algorithm needed for the interactions of all elements against each other based on a proximity query. By using special data structures such as grids, and the parallel nature of graphics hardware, relevant previous works reduces this complexity by considerably factors, making it possible to achieve interactive frame rates. However, existent proposals tend to be heavily bound by the maximum density of such grids, which is usually high, yet leading to arguably inefficient algorithms. In this paper we propose the use of a fine grained grid and accompanying data manipulation that leads to scalable algorithmic complexity. We also implement a representative flocking boids case-study from which we run benchmarks with more than 1 million simulated and rendered boids at nearly 30fps. We remark that previous works achived not more than 15,000 boids with interactive frame rates.

Keywords:: GPGPU, CUDA, Crowd Simulation, Cellular Automata, Flocking Boids

Author's Contact:

epassos,mjoselli,mzamith,esteban,anselmo,aconci@ic.uff.br
jack.f.rocha@gmail.com
bruno@inf.puc-rio.br

1 Introduction

Visual simulations and 3D games are growing fast in terms of content and visual accuracy due to the increasing power of graphics hardware and computers architecture. One consequence of this evolution is that users expectations are now much more sensible when it comes to judging if a simulated entity behavior is overall believable. At the same time, visual applications that were only available at not real time applications regarding to the technology limitations are now becoming possible to run in real-time systems. For this reason the seek for more detailed graphics and more realistic appearance have been a easy to grasp trend, but there has also been a growing interest in more complex animation and Artificial Intelligence algorithms.

While the real world environments can be composed by thousands, or even millions of moving entities, a simulated one is usually constrained by a limited number of them. In a typical environment in the nature one can find a huge number of different animals or even cells interacting between them and with physics elements. This can happen in very different situations, like in a sports arena that is full of autonomous individuals, ants or bee communities or even cells in a blood system. When simulating a similar virtual and real time scene, bound by available computational power, it is common to find a very limited number of independent entities, most of them behaving very predictably. There are several approaches that aim to include more unpredictable behavioral models in simulated environments, such as [Reynolds 1987; Musse and Thalmann 1997; Shao and Terzopoulos 2005; Pelechano et al. 2007; Treuille et al. 2006].

On the other hand, given its computational requirements, visual improvements have been made possible by the clever use of evolving parallel graphics hardware technology. Even being graphics content driven, many today 3D applications have bottlenecks stipulated by the CPU, which is responsible for non-graphics calculations. The use of graphics processing units (GPUs) for general purpose computing has become a new and interesting research area that continuously gains attention from industry and also academia, in order to resolve in a huge parallel architecture problems that are not related to graphics. Behavioral Artificial Intelligence algorithms, although traditionally sequential and executed on the CPU are sometimes suitable, but not easily, to parallel execution. In dynamic emergent crowd simulation, algorithms are driven by the need to avoid the $O(n^2)$ complexity of the proximity queries between active entities. Several approaches have been proposed to cope with this issue [Reynolds 2000; Chiara et al. 2004; Courty and Musse 2005] but none of them has reached an ideal level of scalability. It is also important to notice that no work until the present date has proposed the real time simulation of more than just a few thousands of complex entities that interact with each other. Applications for such technique ranges from crowd behavior prediction in case of a stadium fire or street traffic simulation, interactions between different blood cells and enrichment of computer game worlds.

GPUs are specialized devices for graphics computation, often comprising a set of SIMD (Single Instruction, Multiple Data) processing units due to the parallel nature of this task. The constant development of these devices, pushed mainly by the computer games industry, turned them fast enough to be appropriate for solving other computational intensive problems. The broad adopt of the term GPGPU (General Purpose computation on GPUs) to name this new field of research shows its importance. However, the first applications of GPUs to do general purpose computing had to rely on the adaptation of graphics rendering APIs to different concepts, representing a difficult learning curve to developers. The CUDA technology [NVidia 2007] aims to provide a new abstraction layer on top of (former) graphics hardware to facilitate its use for non graphical processing. While being an important tool to enable a faster modeling of problems to a more general parallel machine concept, CUDA abstracted hardware still is very specialized, carrying a dif-

ferent memory model, with huge impacts on the performance of the developed application, depending on its memory access patterns.

GPUs are SIMD processors that take advantage of the streamed nature of graphics computation, where the processing of each subsequent pixel would require localized data from textures. This specialized hardware uses a great deal of read-ahead and caching techniques to accelerate this computation based in this localization. There are rules of thumb to create efficient streamed applications, the most important being to structure the data in a way to maximize memory reads based on locality and avoiding random writes. These rules enable an efficient use of available cache memory and read ahead mechanisms of these devices.

In this paper we propose a novel simulation architecture for crowds of autonomous geometric entities that are based on a partially sorted matrix. The entities are simulated as Cellular Automata with Extended Moore Neighborhood [Sarkar 2000] over this matrix, which is ideal for the memory model of GPUs. The high performance and scalability are achieved by a very low parallel complexity and independence of the sorting and simulation algorithms. We call this architecture Supermassive Crowd Simulation. To model the data structures and the simulation technique we use a traditional emergent behavior model of flocking boids [Reynolds 1987] but the architecture can be further extended to other simulation models that rely dynamic autonomous entities.

The rest of the paper is organized as follows: Section 2 discusses related work on general purpose computation on GPU and emergent behavior models. Section 3 explains the data structures and simulation steps while section 4 describes the particular behavior model used to validate the proposed architecture. Section 5 brings benchmark results and performance analysis compared to the same simulation on a CPU. Finally, section 6 concludes the paper with a discussion on future work.

2 Related work

This work has two different goals and contributions in crowd behavior simulation. The first one is to provide data structures and an architecture that are suitable for believable modeling of observable behavior in the real world. The second goal is to extract the best performance possible of current hardware. These two goals are shared among the majority of the cited research. Hence, this section is organized in approximate chronological order.

The first known agent-based simulation for groups of interacting animals is the work proposed by Craig Reynolds [Reynolds 1987], in which he presented a distributed behavioral model to perform this task. His model is similar to a particle system where each individual is independently simulated and acts accordantly to its observation of the environment, including physical rules such as gravity, and influences by the other individuals perceived in the surroundings. The main drawback of the proposed approach is the $O(n^2)$ complexity of the traversal algorithm needed to perform the proximity tests for each pair of individuals. At the time, this was such an issue that the simulation had to be run as an offline batch process, even for a limited number of individuals. In order to cope with this limitation, the author suggested the use of spatial hashing. This work also introduced the term *boid* (abbreviation for birdoid) that has been used to designate an individual in animal crowd simulations ever since.

Musse and Thalmman [Musse and Thalmann 1997] propose a more complex modeling of human motion based on internal goal-oriented parameters and the group interactions that emerge from the simulation, taking into account sociological aspects of human relations. Others include psychological effects [Pelechano et al. 2007], social forces [Cordeiro et al. 2005] or even knowledge and learning aspects [Funge et al. 1999]. Shao and Terzopoulos [Shao and Terzopoulos 2005] extend the latest including path planning and visibility for pedestrians. It is important to mention that these proposals are mainly focused on the correctness aspects of behavior modeling. While serving as foundations on the subject, these structures and algorithms are not suitable for real-time simulation of very large crowds, which is one of the goals of this papers.

Reynolds further developed his behavioral model to include more complex rules and to achieve the desired interactive performance by the use of spatial hashing [Reynolds 2000; Reynolds 1999]. This implementation could simulate up to 280 boids at 60fps in a playstation 2 hardware. By using the spatial hash to distribute the boids inside a grid, the proximity query algorithm could be performed against a reduced number of pairs. For each boid, only those inside the same grid cell and at adjacent ones, depending on its position were considered. This strategy led to a complexity that is close to $O(n)$. This complexity value, however, is highly dependent on the maximum density of each grid cell, which can be very high if the simulated environment is large and dense. We remark that the complexity of our data structure is not affected by the size of the environment or the distribution of the boids over it.

Quinn et al. [Quinn et al. 2003] used distributed multiprocessors to simulate evacuation scenarios with up to 10000 individuals at 45fps on a cluster connected by a gigabit switch. More recently, a similar spatial hashing data-structure was used by Reynolds [Reynolds 2006] to render up to 15000 boids in playstation 3 hardware at interactive framerates, with reduced simulation rates of around 10fps. Due to the distributed memory of both architectures, it is necessary to copy compact versions of the buckets/grids of boids to the individual parallel processors before the simulation step and copying them back at the end of it, leading to a potential performance bottleneck for larger sets of boids. This issue is evidenced in [Steed and Abou-Haidar 2003], where the authors span the crowd simulation over several network servers and conclude that moving individuals between servers is an expensive operation.

The use of the parallel power of GPUs to this problem is very promising but brings another issue, related to its intrinsic dependency on data-locality to achieve high performance. For agent-based simulation that relies on spatial hashing, it is desired that the individuals should be sorted through the containing data-structure based on their cell indexes. The work by Chiara et. al. [Chiara et al. 2004] makes use of the CPU to perform this sorting. To avoid the performance penalty, this sorting is triggered only when a boid departs from its group, which is detected by the use of a scattering matrix. This system could simulate 1600 boids at 60fps including the rendering of animated 2D models. The FastCrowd system [Courty and Musse 2005] was also implemented with a mix of CPU and GPU computation that could simulate and render a crowd of 10000 individuals at 20fps as simple 2D discs. Using this simple rendering primitive, the GPU was capable of simultaneously computing the flow of gases on an evacuation scenario. We also make use of the fact that groups tend to move as blocks in crowd simulations and, as will be explained in next sections, use a parallel partial sorting algorithm on the GPU to achieve even higher performance.

The simulation architecture and data-structures of [Treuille et al. 2006] depart from the agent-based models presented so far. It uses a 2D dynamic field to represent both the crowd density and the obstacles of the environment. The individuals navigate through and according to this continuum field. It is argued that locally controlled agents, while providing for complex emergent behavior, are not an appropriate model for goal-driven individuals, such as human pedestrians. The implemented system could simulate up to 10000 humans at 5fps (without graphics) even with the inclusion of a dynamic environment such as traffic lights. The continuum field is an interesting approach but limits the environment to a predetermined size.

Our architecture stores the entities/boids data over a matrix, with an individual cell corresponding to exactly one boid and the simulation occurring in parallel on the GPU. In the implemented system, each boid is modeled as an agent that, based on an Extended Moore Neighborhood [Sarkar 2000], perceives a constant number of other surrounding boids. This cellular automaton model matches perfectly with the data-locality dependency of graphics hardware but imposes that boids data have to be kept spatially sorted over the matrix during simulation. Our proposal, such as most of the above work, is based on distributed agents to yield emergent behavior, but the novel data-strutures is prepared for unlimited environment size and better scalability.

3 Simulation Architecture

Individual entities in crowd behavior simulations depend on observations of their surrounding neighbors in order to decide actions to take. The straightforward implementation of the neighborhood finding algorithm has a complexity of $O(n^2)$, for *n* entities, since it depends at least on distance tests performed for all entity pairs in the crowd. Individuals are autonomous and move during each frame, which leads to a very computationally expensive task.

Techniques of spatial subdivision have been used to group and sort these entities to accelerate the neighborhood finding task. Current implementations are usually based on variations of relatively coarse Voronoi subdivisions, such as a grid. After each update, all entities have their grid cell index calculated. For GPU based solutions, some kind of sorting has to be performed, so geometric space neighbors are grouped together in the used data structure. However, static Voronoi structures have some limitations related to the simulation of very large geometric spaces, where each cell node may have a large number of entities inside. This issue makes the neighborhood finding problem again limited by a hidden $O(n^2)$ complexity factor.

Our architecture is built around a fine grained dynamic data structure that is used to store information about all entities and is sorting a simulation agnostic, meaning one can use it with different sorting strategies and simulation algorithms. The following subsections describe these data structures, the role of sorting and the types of simulation algorithms suitable to the proposed architecture.

3.1 Neighborhood Matrix

The proposed architecture was developed with the CUDA technology [NVidia 2007], in order to keep the process entirely at the GPU. To assure the desired high performance, all information about entities is organized in matrixes that are mapped as textures. Follows the minimum information required for each entity: Position - a 2D vector, representing the position of the entity; Speed - a vector for storing the direction and velocity in a single structure; Type - an integer that can be used to differentiate entity classes.

These entities information are stored in matrixes, where each index contains values for an individual entity. Since is possible to be required a variable number of data about the entities, it may be necessary to use more than one matrix. However it is a good practice to avoid wasting GPU memory by sharing vectors to store more than one single piece of information in each index.

The matrix containing the position vector for the entities is then used as a sorting structure. In Figure 2, one can see an example of such matrix where information about the position of 36 individual entities. To reduce the cost of proximity queries, each entity will have access to the ones surrounding its cells based on a given radius. In the example, the radius is 2, so the entity represented at cell (2,2) would have access to its 24 surrounding entities only. In Cellular Automata, this form of information gathering is called Extended Moore Neighborhood [Sarkar 2000].

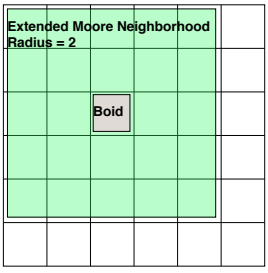


Figure 2: *Neighborhood matrix*

This structure enables the exact prediction of the performance, since the number of proximity queries will be constant over the simulation. This happens because instead of making distance queries, taking as parameters all entities inside its own coarse Voronoi cell and the ones in the adjacent regions, as in traditional implementations, each entity would query only a fixed number of surrounding individual matrix cells. However, this matrix has to be sorted continually in such a way that neighbors in geometric space are stored in cells close to each other. This guarantees that this extension of cellular automata may gather information about close neighbors.

One can notice that the use of a matrix is tied to the two dimensions of this particular case. For a 3D simulation the data structure could be a 3 dimensional array, with no loose of generality, similar to the usage of quadtrees or octrees in other applications of spatial subdivisions. Since in this work the implementation of every entity is mapped to one CUDA thread in both the sorting and simulation steps, it is important to mention that these matrixes are double buffered, so that each of these tasks does not write data over the input structures that can still be read by other CUDA threads.

3.2 Sorting Pass

The matrix that stores position information is used to perform a topological sorting over two dimensions of these vectors. The goal is to store in the top-leftmost cell of the matrixes the entity with the smaller values for X and Y, and the bottom-rightmost cell to the entity with highest values of X and Y respectively. Using both values to sort the matrix, the top lines will be filled with the entities with higher values of Y while the left columns will store those with lower values for X and so on. This kind of sorting allows builds automatic approximate proximity query based on data locality.

When performing a sorting over an one dimension array of float point values, the rule is that given an array A, the following rule must apply at the end:

- $\forall A[i] \in A,\ i > 0 \Rightarrow A[i-1] \leq A[i]$.

Extending this rule to a matrix M where each cell has two float point values *X* and *Y*:

- Eq.1: $\forall M[i][j] \in M,\ j > 0 \Rightarrow M[i][j-1].X \leq M[i][j].X$;
- Eq.2: $M[i][j-1].X = M[i][j].X \Rightarrow M[i][j-1].Y \leq M[i][j].Y$;
- Eq.3: $\forall M[i][j] \in M,\ i > 0 \Rightarrow M[i-1][j].Y \leq M[i][j].Y$;
- Eq.4: $M[i-1][j].Y = M[i][j].Y \Rightarrow M[i-1][j].X \leq M[i][j].X$;

The pass must be divided into four steps, one for odd and one for even elements for each of both directions. The first step runs the sorting between each entity position vector of the even columns against its immediate neighbor in the subsequent odd column, based first on the values of the X component. If it rules of Eq.1 or Eq. 2 are violated, the entities switch cells in the matrixes. It is important to notice that not only the cells in the position matrix have to be switched, but all data that is kept at the others as well, otherwise a violation of the data structure will occur. The other three sorting steps perform the same operation for even columns, odd lines and even lines, respectively.

As shown above, the sorting has to be performed in both directions. This process is sorting agnostic, which means that it is possible to use different sorting strategies, as long as the rules above are eventually or partially achieved during simulation. A partial sorting strategy was tested using only one pass of a parallel implementation for the odd-even transposition sort algorithm at each simulation step. The odd-even transposition sort is similar to the bubble sort algorithm and is possible to complete the pass, traversing the whole data structure, in a $O(n)$ sequential time. Because there are two steps, one for odd and other for even elements, this algorithm is suitable for parallel execution. In Figure 3 it is shown a schematic presentation of a complete odd-even transposition sort pass.

As seen from Figure 3, the four-step pass does one sort for odd and one for even elements on both directions. The first step runs the sorting between each entity position vector of even columns against its immediate neighbor in the subsequent odd column based first on the values of the *X* component. If it happens that sorting rules 1 or 2

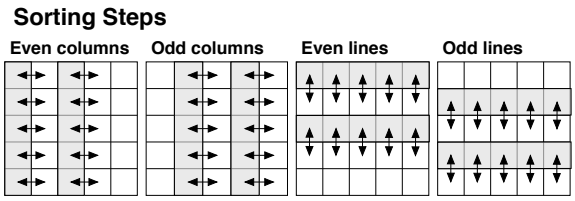


Figure 3: *Partial sorting pass with 4 odd-even transposition steps*

are violated, the entities switch cells in the matrixes. It is important to notice that not only the position vector have to be switched, but all data that is kept at the matrixes, otherwise violating the safety of the data structure. The other three sorting steps perform the same operation for even columns, odd lines and even lines, respectively.

At the present work it was sufficient to run only one complete odd-even pass for each simulation frame because we initialized the position matrix in an ordered state and, the flocking nature of the simulation algorithm imply that the entities do not overlap positions frequently. In practice, this means that in a very few simulation steps, the matrixes correctly represent the proximity relations between them. Depending on the simulation being performed, it may be necessary to perform a complete sorting at each frame step. In this case, it is recommended a sorting algorithm with better worst case complexity, such as a parallel merge sort. However, for all test scenarios built in this work, the incomplete odd-even transposition pass was enough to sustain a approximately correct simulation, with no visual noticeable artifacts, and with a parallel complexity (and performance) of $O(1)$ against $O(log\ n)$ of a complete sorting based on a parallel merge sort.

3.3 Simulation Pass

The simulation pass can perform any kind of crowd emergent behavior of entities that are constrained to the knowledge of data from their neighborhood, such as flocking boids or even dynamic fluids. This pass must be implemented as a CUDA kernel function that receives as arguments at least the neighborhood matrixes (double buffered as input and output) and the time passed since the last step.

4 Case-Study: Flocking Boids

For the purpose of this work, we chose to validate the proposed technique by implementing a well known distributed simulation algorithm called, flocking boids [Reynolds 1987]. This is a good algorithm to use because of its good visual results, proximity to real world behavior observation of animals and understandability. The implementation of the flocking boids model using our algorithm enables a real time simulation of up to one million animals of several species, with a corresponding visual feedback. The number of different species is limited only by the number of animals in the simulation.

Our model simulates a crowd of animals interacting with each other and avoiding random obstacles around the space. This simulation can be used to represent from small bird flocks to huge and complex terrestrial animal groups or either thousand of hundreds of different cells in a living system. Boids from the same type (representing the species) try to form groups and avoid staying close to the other type of species. The number of simulated boids and types is limited only by technology but, as demonstrated in the next section, our method scales very well due to the data structures used. In this section we focus at the extension of the concepts of cellular automata in the simulation step, in order to represent emergent animal behavior.

To achieve a believable simulation we try to mimic what is observable in nature: many animal behaviors resemble that of state machines and cellular automata, where a combination of internal and external factors defines which actions are taken and how they are made. A state machine is used to decide which actions are taken. The actions themselves performed by a cellular automaton algorithm. With this approach, internal state is represented by the boid type and external ones corresponds to the visible neighbors, depending from where the boid is looking at (direction), and their relative distances.

Based on this ideas, our simulation algorithm uses internal and external states to compute these influences for each boid: Flocking (grouping, repulsion and direction following); leader following; and other boid types repulsion (used also for obstacle avoidance). Additionally, there are multiplier factors which dictate how each influence type may get blended to another, in each step. In order to enable a richer simulation, these factors are stored independent for each type of boid in separate arrays. These arrays are indexed by the boid type, meaning that the element at position 1 represents the information that the array keeps for all boids of type 1. These arrays size corresponds to the number of different boid types. Below is a summary of the information kept for each boid type:

- Multiplier factors, one for each type of influence;
- Neighborhood matrix cell index for the leader of the type.

4.1 Vision

In nature, each animal species has a particular eye placement, evolved based on its survival needs such as focusing on a prey or covering a larger field of view to detect predators. To mimic this fact, our boids have a limited field of view, parameterized by an angle. Obstacles and other boids outside this field of view are not considered in the simulation. Figure 4 shows a comprehensible representation of this field of view.

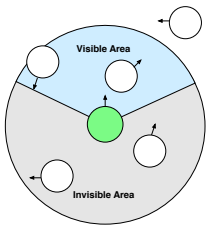


Figure 4: *The visual field of a boid*

When two boids are very close to each other, up to collide, corresponds to a special case where a boid takes into account a neighbor even if it is outside of the its field of view. If collisions where allowed to happen, the simulation could become unstable since neighbor boids comming from behind would suddently appear in front of another. It is possible to think of this as a collision detection for a prevention system, having the same effect as a movement made by animals that, even not seeing each other, would have gotten into a sudden contact.

4.2 Flocking Behavior

A boid keeps on moving by watching his visible neighbors and deciding what direction to take next. Each neighbor influences this direction in different conflicting manners, depending on its type and distance from the simulated boid. From neighbors of the same type, the simulated one receives three simulateous influences: grouping, repulsion and direction following.

4.2.1 Grouping Influence

By grouping we mean the tendency that animals from the same species have to keep forming relatively tight groups. To simulate this behavior we compute the group center position by averaging the positions of all visible neighbors of the same type as the one being simulated. This grouping influence will be multiplied by a grouping factor, unique for each type, and by the distance from the centre. The last factor will make the influence stronger to boids that are far from the group. Figure 5 illustrates grouping and repulsion influences.

4.2.2 Repulsion Influence

If only the grouping influence was taken into account, boids would tend to form very dense groups, resulting in very frequent collisions, not representing what we see in nature. To balance this grouping tendency a collision avoidance influence is computed. For each simulated boid, the relative distance to its neighbors is computed and divided by its lenght. This weighted vector is then multiplied by a specified repulsion factor and added as an influence to the desired motion vector. One can notice that the parameterized factors of both the grouping and distance influences play a major role in determining the density of the groups, since one cancels each other at a certain distance when equilibrium is reached between them.

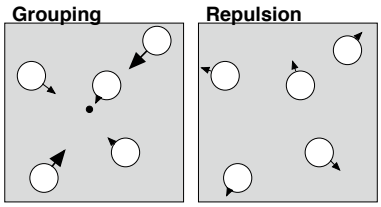


Figure 5: *Grouping and repulsion influences*

4.2.3 Direction Following Influence

Besides the tendency of forming groups, animals also tend to follow the same direction as its companions. To achieve this behavior we compute another influence every time a boid sees a neighbor of the same type. This influence is represented by the current velocity/direction followed by the neighbor. Figure 6 exemplifies this influence.

4.3 Leader Following

Besides from recognizing its neighbors of the same type and trying to move as a group, each type may have a leader to follow. Normal boids, when see the leader, have a stronger desire to follow it, represented by a larger multiplier factor, that gets blended with the other computed influences. Each leaders is simulated at the same time as normal boids but also being identified as such and acting accordingly. However, the movement of this leader is not driven by the desire to keep grouping, but only trying to reach a desired location and avoiding obstacles and other boid groups.

Inside the data structures, the leaders are represented as normal boids. There is a small auxiliary array keeping the current matrix index exclusively for the leaders of each boid type. The array size is the number of different boid types. Element n of this array contains the cell index of the leader for the boids of type *t*. To be correct along the time, this array must be updated by the sorting pass if any of the leaders change its cell.

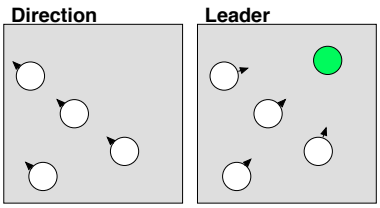


Figure 6: *Direction and leader following*

During the simulation step, for each boid the leaders array value for its type is fetched and the value returned identifies the leader index inside the matrixes. If the returned index corresponds to the boid being simulated, it means that corresponds to the group leader and follows to an alternative and more random simulation algorithm. For normal boids, this leader index is used to fetch its position and direction, so that the correct influence can be computed.

4.4 Obstacles, Enemies and Influences Composition

In this work, obstacles are also represented as boids inserted in the same data structures, also being sorted and simulated. To avoid movement during the simulation step, obstacles are initialized with a different type value, and are not simulated. However, if a neighbor of a specific simulated boid happens to be an obstacle, the only influence calculated a is repulsion force. This force is then multiplied by a factor that is stored in the unused direction vector of this still obstacle-boid, enabling the representation of obstacles of arbitrary sizes with a round repulsion field. Neighbors of different types that are not obstacles also have a strong repulsion influence calculated, but the multiplier factor is kept at the simulated boid type, representing an enemy-fearness factor. All calculated influences are added into an acceleration vector that is used to update the position and direction/speed vectors.

5 Performance and Analysis

In this work, we evaluated two versions of the described simulations. While the first was completely executed at the GPU, the second was built for the CPU. Both versions used the same partial sorting strategy, based on a single pass of the odd-even transposition sort algorithm. The tests were performed on an Intel Core 2 Quad 2.4GHz CPU, 4GB of RAM equipped with an NVidia 8800 GTS GPU. Each instance of the test ran for 300 seconds. The average time to compute a frame was recorded for each one. To assure the results are consistent, each test was repeated 10 times.

A total of 8 different test instances were executed for each implementation type varying only the number of boids, ranging from 64 up to 1,048,576. At preliminary tests, we observed that the number of boid types had little influence on the performance, so a fixed number of 8 types was used. Figure 7 brings a summary of the results showing how both implementations scale with the increasing number of simulated boids. As expected, the CPU version presents an early quadratical behavior, with more than one second to calculate and render each frame when more than 250,000 boids where present, which is another evidence of the better scalability of our GPU implementation.

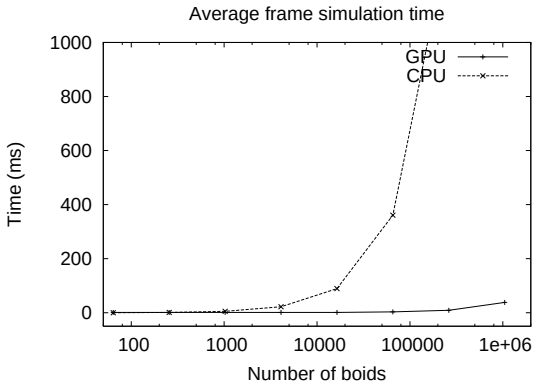


Figure 7: *CPU vs. GPU implementations*

From the raw results, shown in Table 1, it is possible to see that the performance of the GPU implementation bypassed the CPU at around 250 boids and sustained interactive frame rates for more than 250,000 boids. Impressive 25fps are achieved for a population of more than 1 million boids. In our performance experiments, each boid was rendered as a single point on the screen, which enabled us to measure the cost of the simulation steps in CUDA, instead since the rendering of complex models and other backgroung elements

Table 1: *Raw results*

# of boids	time GPU	fps GPU	time CPU	fps CPU
64	1,06	946,13	0,30	3307,90
256	1,08	929,89	1,38	689,85
1024	1,11	904,07	5,56	174,40
4096	1,22	822,57	22,53	43,84
16384	1,66	594,86	89,89	10,92
65536	3,50	280,02	361,27	2,67
262144	9,67	101,96	1390,29	0,66
1048576	38,96	25,45	5394,65	0,17

would make this task the performance bottleneck. Figure 8 shows a screenshot of a running simulation with 65536 flocking boids at an average framerate of 280fps.

Figure 8: *Simulation with 64K boids*

6 Conclusion

In this paper we showed a novel technique for simulating emergent behavior of spatial dynamic entities called Supermassive Crowd Simulation. We showed an implementation capable of running up to 1,048,576 autonomous flocking boids at an interactive frame rate using current graphics hardware and CUDA technology. The data structures are suitable for several different simulation algorithms as long as they can be modeled as cellular automata.

As future work we plan to extend this model to include the representation of more complex geometric obstacles such as buildings or mazes. These augmented data structures and more complex algorithms are being developed in order to have more complex 3D boids representation and consequently more realistic simulations. The project is being built as a crowd simulation library where users can just plug in sorting and simulation strategies. Today our strategies do not take into account time required for rendering complex geometry. However, simple experiments are showing that more polygons for each boid will not compromise to much the achieved performance.

We also plan to further analyse the complexity of the data structures. We are specially interested in the distribution of perturbations of the sorting rules during simulation when applying partial sorting strategies such as the odd-even transposition sort pass used in the presented example. This analysis will not be limited to the neighborhood matrix presented but extended to tridimensional arrays.

References

CHIARA, R. D., ERRA, U., SCARANO, V., AND TATAFIORE, M. 2004. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In *VMV*, 233–240.

CORDEIRO, O. C., BRAUN, A., SILVEIRA, C. B., AND MUSSE, S. R. 2005. Concurrency on social forces simulation model. In *Proceedings of the First International Workshop on Crowd Simulation*.

COURTY, N., AND MUSSE, S. R. 2005. Simulation of large crowds in emergency situations including gaseous phenomena. In *CGI '05: Proceedings of the Computer Graphics International 2005*, IEEE Computer Society, Washington, DC, USA, 206–212.

FUNGE, J., TU, X., AND TERZOPOULOS, D. 1999. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *Siggraph 1999, Computer Graphics Proceedings*, Addison Wesley Longman, Los Angeles, A. Rockwood, Ed., 29–38.

MUSSE, S. R., AND THALMANN, D. 1997. A model of human crowd behavior: Group inter-relationship and collision detection analysis. In *Workshop Computer Animation and Simulation of Eurographics*, 39–52.

NVIDIA, 2007. Cuda technology. http://www.nvidia.com/cuda.

PELECHANO, N., ALLBECK, J. M., AND BADLER, N. I. 2007. Controlling individual agents in high-density crowd simulation. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 99–108.

QUINN, M. J., METOYER, R. A., AND HUNTER-ZAWORSKI, K., 2003. Interaction with groups of autonomous characters.

REYNOLDS, C. W. 1987. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 25–34.

REYNOLDS, C. 1999. Steering behaviors for autonomous characters. In *Game Developers Conference*.

REYNOLDS, C. 2000. Interaction with groups of autonomous characters. In *Game Developers Conference*.

REYNOLDS, C. 2006. Big fast crowds on ps3. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, ACM, New York, NY, USA, 113–121.

SARKAR, P. 2000. A brief history of cellular automata. *ACM Comput. Surv. 32*, 1, 80–107.

SHAO, W., AND TERZOPOULOS, D. 2005. Autonomous pedestrians. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, 19–28.

STEED, A., AND ABOU-HAIDAR, R. 2003. Partitioning crowded virtual environments. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM, New York, NY, USA, 7–14.

TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, 1160–1168.

Uma Engine em XNA e Prolog para Apoio ao Ensino de Programação Declarativa

Alex F. V. Machado Esteban W. Clua *Flavio S. C. da Silva Marcelo da S. Corrêa

Universidade Federal Fluminense (UFF) *Universidade de São Paulo (USP)

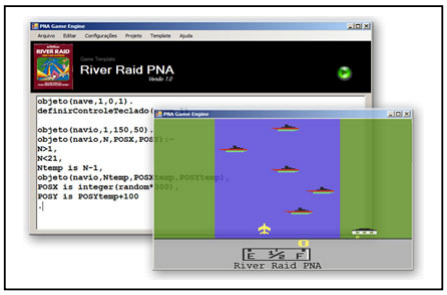


Fig. 1: Interface do *PNA Game Engine* com a visualização de uma instância do jogo do *template River Raid*.

Abstract

This work presents a novel *game* oriented tool which can motivate the teaching of declarative languages programming (Fig. 1). It uses the P# compiler [COOK 2003, COOK 2, 2003] as a middleware between the Prolog and the C# languages, allowing the usage of the XNA library for the creation of *game*s as a base for developing applications focused on declarative languages.

Keywords: Prolog, XNA, teaching

1. Introdução

Conforme apresentado em [SILVA & MELO 2006], a programação declarativa e a programação imperativa se fundamentam em modelos de computação que, embora matematicamente equivalentes, enfatizam conceitos distintos. A programação imperativa se baseia no conceito de máquinas de estados, e se presta melhor à resolução de problemas que conceitualmente sejam melhor caracterizados dessa forma. Já a programação declarativa se baseia no conceito de reapresentação de teorias formais, e muitos problemas difíceis de serem resolvidos imperativamente em linguagens como VB, C++ e Java podem se tornar simples com o uso do paradigma declarativo. Não há muitos esforços registrados na literatura no que se refere a compatibilizar sistemas de programação declarativa com plataformas de desenvolvimento modernas, fundamentalmente construídas visando o atendimento de linguagens imperativas. Por exemplo, não existem esforços no sentido de criar uma interface PROLOG para uma ferramenta de desenvolvimento como o *Visual Studio*, possibilitando a sua integração à plataforma .Net. De fato, a criação de um sistema completo nesta linguagem para a plataforma .Net pode trazer complicações de design, pois a programação em lógica não possui uma estrutura para a criação de sistemas complexos de *back-end* compatível com as estruturas existentes e já implementadas no *Visual Studio*. Como conseqüência, o aprendizado de técnicas de programação declarativa pode se tornar mais árido que o aprendizado de programação imperativa.

Uma das ferramentas mais conhecidas para apoiar o ensino de programação através do desenvolvimento de jogos é o Robocode [HARTNESS 2004]. Ele permite o exercício de conteúdos teóricos de forma prática em aulas de inteligência artificial. Em [HARTNESS 2004] demonstra-se que os alunos destas aulas foram capazes de compreender melhor a teoria e adquiriram maior confiança para implementar seus códigos.

Neste trabalho é proposto e documentado um sistema, denominado *PNA Game Engine*, que utiliza a programação em lógica como entrada para a definição do comportamento inicial de um jogo desenvolvido utilizando a biblioteca XNA. Esta ferramenta tem como finalidade auxiliar o docente no processo de ensino-aprendizagem, motivando o estudo deste paradigma por permitir também a geração de elementos visuais em jogos a partir de programas declarativos simples em PROLOG.

Esse sistema disponibiliza *templates* para jogos (com modelos 2D e 3D prontos, mas com as classes principais sem instanciação) que podem ser acessados através de uma API batizada de PNA (*Pna is Not an Acronym*) pela interface em PROLOG.

O artigo está organizado da seguinte forma: a seção 2 apresenta uma justificativa para o uso de PROLOG no ensino de computação e as várias maneiras de como esta linguagem pode ser utilizada para o desenvolvimento de jogos. Ainda nesta seção faz-se

uma revisão sobre uma ferramenta similar à proposta do presente trabalho. Na seção 3 são detalhadas as principais tecnologias presentes no desenvolvimento deste *software*, em especial a plataforma .Net, a integração de XNA com formulários e o compilador P#. Na seção 4 demonstra-se a aplicação dessas tecnologias no processo de desenvolvimento do *PNA Game Engine* e seus principais componentes e funcionalidades. Nas sessões 5 e 6 são apresentadas algumas conclusões e trabalhos futuros.

2. Visão Geral Sobre Prolog

A linguagem de programação PROLOG possibilita resolver problemas lógicos através da programação em um computador. Ela foi desenvolvida na década de 70, visando especificamente a formalização e resolução de certos problemas de lingüística computacional, e seu uso posteriormente se estendeu para outras áreas em que a programação declarativa e a caracterização conceitual de problemas baseada em lógicas formais – clássicas ou não clássicas – se mostrasse conveniente [SILVA & MELO 2006]. Dentre muitos problemas da vida real, que podem ser modelados por meio de linguagens lógicas, destacamos os jogos de computador. As características do PROLOG associadas às características estratégicas de um jogo proporcionam um ambiente adequado para a programação declarativa, especificamente a fundamentada em uma modelagem baseada em inferências lógicas.

Nesta seção será justificada a relevância do ensino de PROLOG, assim como sua área de aplicação no que diz respeito ao desenvolvimento de jogos de computador.

2.1 Vantagens do Ensino de Prolog

Além do fato de a linguagem de programação PROLOG ser adotada por muitos docentes da área de inteligência artificial, o ensino desta linguagem possui diversas outras vantagens [PALAZZO 1997], tais como:

- Aprendizado mais fácil e natural em comparação com as linguagens imperativas;
- Permite a implementação de sistemas reflexivos;
- Libera o aluno dos problemas associados ao controle de suas rotinas, permitindo-lhe concentrar-se nos aspectos lógicos da situação a representar.
- Exercita fundamentos de linguagens de especificação.
- Facilita a implementação de regras gramaticais de gramáticas livres de contexto.
- Permite o estudo avançado de recursividade através de mecanismos simples como *backtracking, cut* e *fail.*

No presente trabalho, destaca-se a contribuição do uso e desenvolvimento de jogos, a partir de técnicas de programação em lógica, para o processo de aprendizagem de PROLOG e programação declarativa.

2.2 Prolog Aplicado a Jogos

O uso de linguagens imperativas muitas vezes inibe o programador de usufruir das vantagens que a programação declarativa pode oferecer ao desenvolvimento de aplicações.

O desenvolvimento de um *engine* que permita visualizar os resultados de uma rotina declarativa sem a necessidade de implementação das funções imperativas fundamentais presentes em um jogo (como gerenciamento dos dispositivos gráficos, controle do teclado e carga dos arquivos de imagem), pode aumentar a motivação do aluno para o estudo de linguagens e ambientes de programação que permitam obter tais facilidades.

Podem-se destacar três grandes áreas para uso de PROLOG no desenvolvimento de jogos: controle da lógica, sistemas especialistas e criação de diálogo.

2.2.1 Controle da lógica

A programação em lógica é constituída por dois elementos principais: a lógica e o controle [PALAZZO 1997]. O componente lógico corresponde à definição (especificação) do que deve ser solucionado, enquanto que o componente de controle estabelece como a solução pode ser obtida. A estrutura lógica é responsável por gerar a base de conhecimento e a estrutura de controle coordena o entendimento sobre a mesma. É necessário somente descrever o componente lógico de um programa, deixando o controle da execução ser exercido pelo sistema de programação em lógica que se está utilizando. Portanto a tarefa do desenvolvedor passa a ser simplesmente a especificação do problema que deve ser solucionado, razão pela qual as linguagens lógicas podem ser vistas simultaneamente como linguagens para especificação e linguagens para a programação de computadores.

Trazendo o conceito de lógica e controle para os jogos podemos definir, respectivamente, o ambiente e o comportamento dos elementos. Por exemplo, em um jogo do estilo plataforma os obstáculos poderiam ser gerados seguindo um princípio lógico na base de dados de informação de posições e definir a IA de ações dos inimigos usando a estrutura de controle que analisará essa base lógica.

2.2.2 Sistemas especialistas

Um sistema especialista resolve problemas que normalmente são solucionados por pessoas especializadas. Ele recebe uma entrada do usuário, analisa as possíveis respostas da base de conhecimento e, dependendo da resposta, pode exigir uma nova entrada do usuário, repetindo este processo até uma solução ser obtida.

O estudo e a criação de sistemas especialistas representam uma das subdivisões mais importantes das linguagens de programação em lógica, bem como dos jogos com inteligência artificial. Pode-se criar, por exemplo, as possíveis ações de um NPC (*non-player character*, ou personagem sem-jogador) que pode variar com as ações do *player* (personagem controlado pelo jogador).

2.2.3 Criação de Diálogo

A maior parte dos jogos eletrônicos, e principalmente os do gênero RPG (*Role Playing Game*), necessitam de diálogo entre os NPCs e o *player*. Em [LEBBINK, WITTEMAN & MEYER 2004] é apresentado um sistema multi-agentes de diálogo para jogos que permite a análise semântica das sentenças através de um motor escrito em PROLOG.

A criação de diálogos entre *players* e personagens do mundo virtual representa uma das principais áreas de estudo da aplicação de PROLOG para jogos [LEBBINK, WITTEMAN & MEYER 2004].

2.3 Ferramentas de Programação Gráfica para o Ensino de PROLOG

Em [SILVA & SILVA 2006] foi apresentado um sistema com um ambiente virtual 3D, animações e interação com múltiplos agentes, para permitir que alunos de graduação exercitem a programação lógica.

Para tanto foram utilizados:

- Uma interface para a interação entre programas em PROLOG e programas em C++;
- O *engine* 3D Ogre para a visualização de ambientes virtuais tridimensionais, controlados por programas em C++;
- O interpretador SWI PROLOG para a construção e execução de programas em PROLOG.

O trabalho mostra, através de experimentos em sala, que o desenvolvimento de programas em PROLOG com visualização gráfica do comportamento dos mesmos representa uma valiosa ferramenta para docentes que desejam motivar os alunos no aprendizado de inteligência artificial, lógica formal e programação declarativa.

3. Tecnologias para a Criação da Ferramenta Proposta

Toda tecnologia utilizada na ferramenta proposta é gratuita, de forma a facilitar seu uso em fins acadêmicos. Além de programas e linguagens da plataforma .Net, incluem-se rotinas para integração do XNA em formulários, DLL do compilador P# e a própria linguagem PROLOG.

3.1 Tecnologias Microsoft

Optou-se pela plataforma *Microsoft Visual Studio .NET* como principal ferramenta de desenvolvimento, por ser uma ferramenta RAD (*Rapid Application Development*) e permitir a interoperabilidade entre múltiplas linguagens [LIBERTY 2001].

Dentre as principais linguagens suportadas por esta plataforma, escolheu-se C# por ser uma linguagem orientada a objetos similar ao JAVA (mas com determinadas configurações extras, herdadas do C++) e por ser a única a trabalhar com XNA.

O XNA (*XNA is Not an Acronym*) *Game* Studio Express é uma API da plataforma .Net que permite fácil acesso aos periféricos (como o teclado), ao hardware gráfico, controle de áudio e armazenamento de informações (em arquivos ou banco de dados) [CREATORS CLUB 2008]. Essa API pode também ser usada como base para o desenvolvimento de jogos para o console XBox 360.

Outra ferramenta importante do *framework* da Microsoft é o CSC (*C-Sharp Compiler*), aplicativo *stand alone* que é chamado para compilar classes .cs em arquivos executáveis (.exe) ou DLL's através de uma linha de comando.

3.2 XNA em Formulários

O *PNA Game Engine* foi desenvolvido utilizando os formulários do *Visual Studio* e a linguagem XNA. Estes formulários permitem de forma fácil e rápida a criação de menus de configuração e de uma interface amigável para o aluno. Entretanto, a união dessas duas tecnologias não é um procedimento nativo da plataforma. A dificuldade reside no fato do gerenciador de dispositivos gráficos do XNA criar sua própria janela e não cooperar com a janela normal dos formulários do *Visual Studio*. Isto exige ao desenvolvedor criar seu próprio código de gerência de dispositivo gráfico.

Existem dois caminhos para se criar um projeto do *Visual Studio* que use formulários da plataforma. Net e o *XNA framework*:

- Criar um novo projeto de aplicação do *windows* com formulário e referenciar as DLL's do XNA nele; ou,

- Criar um novo projeto de jogo do *windows* e referenciar uma diretiva para uso de formulários (opção adotada neste artigo).

As principais etapas para a integração do formulário no *Windows Game Project* no *Visual Studio* são:

1. Preparação do ambiente a partir de um novo *Windows Game*, substituindo a classe principal por um formulário.
2. Criação de um componente para exibição do conteúdo do XNA através de um *User Control*.
3. Criação de um procedimento para atualizar a cada momento o gráfico contido no *User Control*.
4. Criação de um procedimento no formulário para gerenciar o dispositivo gráfico. Ele possui duas funções principais, o *Draw()* para renderizar o conteúdo e o *Blit()* para aumentar a eficiência, pois este limitará o código de visualização para ser executado somente quando exigido, uma vez que armazena todo resultado de cada renderização em uma textura para evitar que a próxima renderização do frame comece do zero.
5. Criação da função *RenderToTexture()* para configurar o dispositivo gráfico para usar o *render target* e o *depth buffer*, de forma a esvaziar os *buffers*, desenhar a tela e adquirir o resultado em forma de textura.
6. Criação do método principal do controle do comportamento do jogo em uma linha do tempo. Para tanto é criado um conjunto de funções para usar do *game loop* do XNA no evento *OnIdle()* do formulário. Neste procedimento de repetição é incorporado as funções de *Update()* e *Draw()*, nesta ordem.

Portanto estas etapas têm por objetivo criar um componente capaz de simular o ambiente de desenvolvimento do XNA (por implementar os métodos *Draw()* e *Update()* entre outros) integrado ao código do formulário do *Visual Studio*.

3.3 O Compilador P#

O compilador P# [COOK 2003] foi desenvolvido a partir do projeto do PROLOG Café [BANBARA & TAMURA 1999] para produzir C# ao invés de Java a partir de um *script* PROLOG. Ele pode ser usado de forma *stand alone* para testar, executar ou gerar aplicações a partir de códigos PROLOG, ou como um módulo (uma única DLL) para compilar arquivos PROLOG em classes C#. Quando um código PROLOG é compilado no P#, ele gera uma classe para cada predicado definido. A convenção usada no nome do arquivo gerado é:

NomeDoPredicado_NumeroDeArgumentos.cs
(Ex.: Pai_2.cs)

Quando o *assembly* do módulo P# é usado em uma aplicação C# ele dispõe de diversas classes para o desenvolvedor comunicar com os predicados e configurações do lado ambiente do PROLOG (classes geradas a partir do código PROLOG). Um exemplo de *script* é:

```
1. VariableTerm pai = new VariableTerm( );
2. PrologInterface sharp = new PrologInterface( );
3. sharp.SetPredicate(new Pai_2(pai,SymbolTerm.MakeSymbol( "Zé" ),new ReturnCs(sharp)));
4. sharp.Call();
5. Console.WriteLine( "O pai é: {0}", pai.Dereference() );
```

Na linha 1 é instanciado um termo PROLOG (variável PROLOG) com o nome de pai; na linha 2 cria-se a interface para comunicação com o código PROLOG, *sharp*; na linha 3 realiza-se uma consulta através do método *SetPredicate* em uma classe denominada *Pai_2* (que é um arquivo .cs gerado a partir de um *script* PROLOG convencional) armazenando no termo pai todos os resultados cujo segundo termo é Zé; na linha 4 chama-se o primeiro resultado; e na ultima escreve-se esse resultado na tela.

4. Desenvolvimento do Sistema

Nenhum *engine* de desenvolvimento de jogos atual (como o XNA ou o *3D Game Studio*) utiliza qualquer linguagem de programação em lógica em seu módulo de *script*. Mas devido às suas inúmeras aplicações na área de Inteligência Artificial [CASANOVA 2006, GIORNO E FURTADO], muito estudo existe neste sentido [SILVA e SILVA 2006]. O *PNA Game Engine* é uma ferramenta que preenche esta lacuna (Fig. 2).

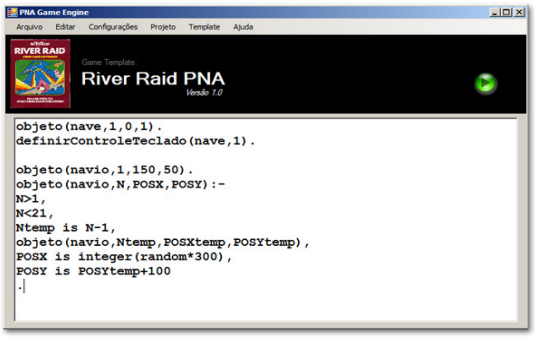


Fig. 2 - Interface do *PNA Game Engine*

Com base no compilador P# e na biblioteca de códigos gráficos para desenvolvimento de jogos da Plataforma .Net, a XNA, são criados jogos "vazios" ou *game templates* compostos de bibliotecas de classes, modelos 3D de objetos, modelos 2D de *sprites* e elementos sonoros. O *template* completo desenvolvido nesta primeira versão do programa foi o clássico *River Raid* (Fig. 3), que é um jogo do console Atari, no qual

um avião sobrevoando um rio deve destruir e ultrapassar os navios e helicópteros que são seus obstáculos.

Fig. 3 – Jogo River Raid

Nesta versão o programador está limitado a usar somente os *games templates* pré-definidos, excluindo a possibilidade de criação de um jogo mais personalizado.

4.1 O *Engine PNA*

O *engine PNA* é o *software* que vai integrar o *template* do jogo com o restante do sistema. Suas principais funcionalidades são (Fig. 4):

- Abrir arquivo PNA – permite abrir um novo código em sua área de edição.
- Definir *template* - permite trocar o *template* atual. Isto carrega as bibliotecas de código e altera todas as opções de configuração e compilação para este determinado modelo. Sempre que o programa for aberto, será exibida uma tela para a definição do *template* inicial. Durante o desenvolvimento de um programa se seu modelo for alterado, a área de edição será reiniciada (para não criar uma confusão de predicados, pois eles são diferentes para cada *template*) e será dado início a um novo arquivo PNA.
- Exemplo – permite abrir arquivos PNAs pré-definidos (ver tabelas Tab. 1 e Tab. 2). Esses arquivos têm a extensão .pl, porque embora possuam predicados pré-estipulados para o *game template* definido ainda assim não deixam de representar *scripts* do PROLOG.
- Arquivo existente - permite abrir um arquivo PNA desenvolvido pelo usuário.
- Salvar Arquivo PNA – salva o *script* desenvolvido pelo usuário. Somente depois de salvo que esse *script* pode ser compilado.
- Compilar – realiza as etapas: verifica o ultimo código PNA salvo e exibe mensagem de erro em console caso exista (P#); traduz o código para arquivos .cs, sendo uma classe para cada predicado (P#); e, transforma cada arquivo gerado em DLL (CSC).
- Executar jogo – carrega os parâmetros definidos nas DLLs e exibe as novas configurações do jogo na tela.
- Consultar ajuda – permite ver opções de ajuda para auxiliar aos novos usuários.
- Ver Sobre – exibe informações dos autores e do *software*.
- Ver Tutorial – exibe instruções gerais para o uso do PNA e os predicados específicos do *template River Raid*.

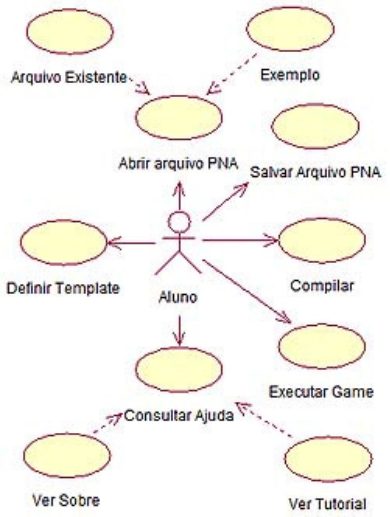


Fig. 4 - Diagrama UML de Caso-de-Uso do *PNA Game Engine*

4.2 Componentes do Sistema

Os principais componentes do *PNA Game Engine* são (Fig. 5):

- Fonte PROLOG – código inteiramente em PROLOG acrescido de predicados especiais que facilitam na comunicação com o C#.
- Compilador P# - Traduz o código PROLOG em arquivos de classe C#. Posteriormente ele será usado para comunicar com as DLLs geradas pelo CSC.
- CSC – *C# Compiler* é um executável da .Net Framework responsável por transformar as classes .cs em DLLs para permitir a exibição do jogo final em *runtime*.
- *XNA Framework* – fornece todo suporte para a execução de rotinas em XNA.
- *User Control XNA* – componente criado no *Visual Studio* para permitir a integração de formulários com XNA sem problema de compatibilidade entre os gerenciadores de dispositivos gráficos.
- *Game Template PNA* – pacote que integra os arquivos necessários para a execução do respectivo *template*.
- Inicializador – representa um componente vital do sistema. Possui um conjunto de funções que utilizam a DLL do compilador P#

para interpretar os predicados das DLLs das classes criadas pelo CSC e gerar o comportamento do jogo do *template*.

- Imagens e Sons – pacote que armazena os sons e imagens dos *sprites* necessários neste *template*.
- Exemplos – Códigos PNA de exemplo para o *template* carregado.

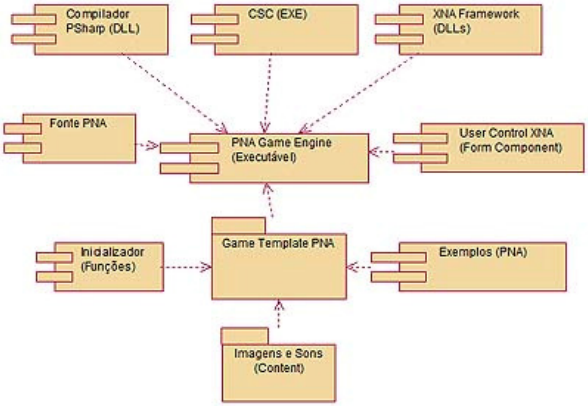


Fig. 5 - Diagrama de componentes do sistema desenvolvido.

4.3 O Código PNA para o *Template River Raid*

O PNA é uma mistura de PROLOG com predicados pré-definidos do *template* específico carregado. Nele é possível inserir objetos 3D ou 2D (dependendo do *template*), alterar o visual, fazer o *game design*, controlar o fluxo dos dados, gerenciar o comportamento, animar, etc.

Foi implementado, nessa primeira versão do programa, somente o comportamento inicial do jogo. No método *Update()* estará pré-configurada a animação do movimento dos objetos simulando o vôo de uma nave.

Como exemplo, na Tab. 1 e na Tab. 2 são criados códigos para o *template River Raid* com os seguintes predicados PNA:

- objeto – permite inserir um *sprite* 2D. Possui quatro parâmetros: nome do objeto (restrito aos objetos existentes no jogo), índice (identificador do objeto, deve ser único para cada de objeto), posição X e posição Y.
- controle – permite definir os controles básicos (movimentos e tiros) de evento do teclado para um determinado objeto. Os dois parâmetros permitem identificar o objeto a ser manipulado: nome e índice.

O exemplo da Tab. 1 cria a posição inicial de 20 navios e de 1 nave, e define o controle do teclado para a nave.

```
objeto(nave,1,0,0).
objeto(navio,1,150,50).
objeto(navio,2,50,150).
objeto(navio,3,250,250).
objeto(navio,4,150,350).
objeto(navio,5,150,450).
objeto(navio,6,250,550).
objeto(navio,7,50,650).
objeto(navio,8,150,750).
objeto(navio,9,150,850).
objeto(navio,10,100,950).
objeto(navio,11,150,1050).
objeto(navio,12,200,1150).
objeto(navio,13,200,1250).
objeto(navio,14,50,1350).
objeto(navio,15,150,1450).
objeto(navio,16,300,1550).
objeto(navio,17,300,1650).
objeto(navio,18,200,1750).
objeto(navio,19,300,1850).
objeto(navio,20,50,1950).
controle(nave,1).
```

Tab. 1 – Exemplo de um código em PNA

O exemplo da Tab. 2 gera um navio com uma posição pré-determinada e outros 19 com posições aleatórias no eixo x enfileirados no eixo y. Ele também insere e cria o controle para uma nave.

```
objeto(nave,1,0,0).
controle(nave,1).

objeto(navio,1,150,50).
objeto(navio,N,POSX,POSY):-
N>1,
N<21,
Ntemp is N-1,
objeto(navio,Ntemp,POSXtemp,POSYtemp),
POSX is integer(random*300),
POSY is POSYtemp+100
.
```

Tab. 2 – Mesmo código da Tab. 1 criado a partir de recursão.

Portanto, estes códigos das tabelas anteriores são semelhantes, pois distribuem navios no caminho à frente da nave (Fig. 6). Entretanto, eles exercitam técnicas distintas de PROLOG: o primeiro mostra claramente o uso de predicados e atributos formando um conjunto de fatos; no segundo é criado uma cláusula com o uso de recursividade e todo princípio definido no primeiro exemplo.

Fig. 6- Exemplo de código PNA para o jogo *River Raid*

5. Conclusão

Neste trabalho foi apresentado o *PNA Game Engine*, uma ferramenta com a entrada de dados em PROLOG e o retorno visual de um jogo desenvolvido em XNA. Para este fim foi desenvolvida a API PNA capaz de controlar o comportamento do *game template* através do paradigma de programação declarativo.

Esta engine é capaz de gerar elementos visuais e comportamentos de objetos em jogos a partir de programas declarativos simples. Assim como em [HARTNESS 2004], este aplicativo busca motivar o aprendizado de técnicas de programação, mais especificamente de programação em lógica, utilizando a linguagem PROLOG. Além deste objetivo pedagógico, demonstrou-se que a criação de um *engine* voltado para a programação de jogos através de uma interface em PROLOG pode auxiliar na exploração científica desta área de *games*.

Todos os componentes do sistema criado utilizam tecnologia *Microsoft* e foram desenvolvidos na versão *freeware* da plataforma .Net.

Na engenharia deste processo destacam-se, devido à interoperabilidade proporcionada, as rotinas de integração dos formulários do *Visual Studio* com o XNA e comunicação da linguagem C# com o PROLOG através do uso do compilador P# [COOK 2003].

6. Trabalhos Futuros

Uma continuidade natural deste trabalho é a criação de novos *game templates* em ambientes virtuais 3D, dentre os quais destacamos:

- Mundo de Wumpus - jogo de tabuleiro em que um caçador deve andar entre as casas a procura de um tesouro. Estas casas podem possuir elementos nocivos ao caçador (fogueira e o monstro Wumpus) e dicas para evitar estes elementos. O Mundo de Wumpus tem sido amplamente utilizado no ensino de conceitos fundamentais de Inteligência Artificial.
- Jogo do gênero *Racing* – jogo de corrida de carros com obstáculos e disputas de velocidade. Jogos de corrida possibilitam o aprendizado de técnicas avançadas de programação, como por exemplo programação *multithreaded*.

Embora existam experimentos preliminares com resultados positivos de aplicação em aulas de inteligência artificial, especificamente para o ensino de PROLOG, de ferramentas gráficas [SILVA & SILVA 2006], é de interesse dos autores documentar atividades práticas em sala de aula para uma avaliação mais criteriosa dos resultados.

Referências Bibliográficas

BANBARA, M.; TAMURA, N. Translating a Linear Logic Programming Language into Java. ICLP'99 Workshop, 1999.

CASANOVA, Marco A.; GIORNO, Fernando A. C.; FURTADO, Antonio L.. Programação em Lógica e a Linguagem Prolog. 2006.

COOK, Jonathan. P#: Using Prolog within the .NET Framework. Laboratory for Foundations of Computer Science, University of Edinburgh. 2003.

COOK, Jonathan. P# Manual (version 1.1.3). Manual do programa. 2003.

CREATORS CLUB. XNA definition. Disponível em: http://creators.xna.com/ . Acessado em Agosto/2008.

HARTNESS, Ken. Robocode: using games to teach artificial intelligence. Journal of Computing Sciences in Colleges archive. Volume 19. 2004

LEBBINK, Henk-Jan. WITTEMAN, Cilia. MEYER, John-Jules. A Dialogue *Game* Approach to Multi-Agent System Programming. Belgium-Netherlands Conference on Artificial Intelligence, 2004

LIBERTY, J. Programming C#. O'Reilly, 2001.

PALAZZO, Luiz A. M. Introdução à Programação Prolog. Editora da Universidade Católica de Pelotas, 1997

SILVA, Flávio Soares Corrêa da; SILVA, Filipe Corrêa Lima da. A *Game*-based Animation Tool to Support the Teaching of Formal Reasoning. SB*Game*s 2006.

SILVA, Flávio Soares Corrêa da; MELO, Ana Cristina Vieira de. Modelos Clássicos de Computação. Thomson, 2006.

A Real-Time Proxy for Flexible Teamwork in Dynamic Environments

Ivan M. Monteiro and Luis Otavio Alvares
Instituto de Informática, UFRGS, Brazil

Abstract

The demand for believable behavior in computer games has driven the research in artificial intelligence to improve character's behavior in games. However, little has been done to improve the collective behavior in partially observable, dynamic, and stochastic environments. In this work, teams of agents are developed, based on *Joint Intentions*, for environments with such features as the game Unreal Tournament 2004. Because of some limitations of existing tools, we introduce a new proxy-based tool to help agents to be a teammate. Experiments have shown that it is feasible to use a distributed approach and the advantages of our tool face to Machinetta.

Keywords:: Teamwork, Multi-agent System, Artificial Intelligence, Distributed System

Author's Contact:

{immonteiro,alvares}@inf.ufrgs.br

1 Introduction

In computer games, the demand for believable behavior has grown, in order to decrease the distance to the human behavior [de Byl 2004]. However, the importance given to behavioral aspects is usually less than the importance given to graphic aspects. When the subject is team-based games the situation is worst, because bots do not assume coherent roles inside the team. Thus, although there is progress on individual behavior, little has been done for social behavior.

Some team-based games, such as sport games, seem to use teamwork[1], but in reality they use a centralized solution that access the full knowledge about the state of the entities. Thus, a soccer game does not need a multi-agent coordination to perform a team move, because all bots have a centralized control. This centralized solution also allows bots to cheat the human, using knowledge about hidden state. It makes the game development easier but the result are games less believable.

The immediate advantages for distributed control of bots are the new possibility of gameplay and the load balance of bots computation. The intuition leads to belief that with more computation power it becomes easier to develop more believable behavior. This model is interesting for massive multiplayer games, that are inherently distributed. Indeed, it opens new possibilities for other kinds of games.

Teamwork has emerged as the paradigm for coordinating cooperative agents in dynamic environments and it has been shown to be capable of leading to flexible and robust behavior [Sycara and Sukthankar 2006]. It has been applied to many domains, such as: rescue disaster [Nair et al. 2000], militar combat [Hill et al. 1997] [Jones et al. 1999], robocup soccer [Kaminka 1999] [Marsella et al. 1999] [Marsella et al. 2001], and collaboration between human and agents [Scerri et al. 2003c]. Flexible teamwork, promoted by the explicit team model, that defines commitments and responsabilities for teammate, allows a robust coordination, keeping coherence even with individuals faults and unpredictable changes in the environment [Steven 2005].

Some theories have been proposed to formalize the behavior of a group of agents to act as a team. Two of these theories have important results for real problems. They are *Joint Intentions* [Levesque et al. 1990] and *Shared Plans* [Grosz and Kraus 1996]. Both are described through logic formalism, but with different approaches. *Joint Intentions* focus on a team's joint mental state, while *Shared Plans* focus on the agent's intentions towards its collaborator's action or towards a group's joint action.

Tools as *STEAM* [Tambe 1997] and *Machinetta* [Steven 2005] had successful in many complex environments [Hill et al. 1997] [Tambe and Zhang 1998] [Tambe and Zhang 2000] [Marsella et al. 2001] [Jones et al. 1999] [Pynadath and Tambe 2003] [Scerri et al. 2003a] [Schurr et al. 2005] [Scerri et al. 2003c] [Scerri et al. 2004] [Chalupsky et al. 2001] [Scerri et al. 2001] [Scerri et al. 2003b] [Yen et al. 2001]. The natural way would be to use some of these tools to improve teamwork on bot's development in computer games. However, practical issues have pointed out for the development of a new specialized tool. *STEAM* was developed using an old version of *SOAR* [Laird et al. 1987], that made changes in its language since than. *Machinetta*, that is a successor of *STEAM*, has shown many limitations for the game domain.

The aim of this work is to promote social behavior for agents inside computers games. The paradigm known in multi-agent systems as teamwork is used to achieve this goal. In this context, a team is a group of autonomous pro-active entities, with capability of reflection about their own abilities and the abilities of the group that share a common goal.

This work also introduces *TWProxy*, a new tool that enables agents to perform teamwork. *TWProxy* uses many of the good ideas from *Machinetta*, avoiding some of its limitations and adding new features. Thus, this article focuses on the development of *TWProxy* and its usage on the development of team bots. Tests and evaluation compare how good this new solution is.

The remainder of the paper is organized as follows: Section 2 shows the main related work, Section 3 describes the environment provided by the game, Section 4 describes the features of *TWProxy*, Section 5 presents tests and evaluation, and Section 6 concludes the paper.

2 Related Work

In a teamwork problem, a group of agents acts coordinated, in a cooperative way, to achieve a shared goal. The main difference between classical coordination and teamwork is how to achieve a shared goal. Classical coordination uses a set of pre-defined plans to apply according to the situation. The teamwork extends the classical coordination using cooperation and commitment between agents, been applicable in other domains that are not the *locker-room agreements* [Stone and Veloso 1998].

Theoretical works in agent teamwork [Levesque et al. 1990] [Grosz and Kraus 1996] [Grosz and Kraus 1998] define some features about team behavior. They describe what the agents need to share in order to perform a teamwork: goals to achieve, plans to perform together, and knowledge about the environment. The agents need to share their intentions to perform a plan in order to achieve a common goal, the teammates need to be aware about their capabilities and how to fill roles to perform the high level team plan. They must be able to monitor their own progress and the group progress toward team goals. Many systems have been developed using the teamwork idea as team with human collaboration [Scerri et al. 2003a], teams for disaster situation [Nair et al. 2000], and teams for industry production line [Jennings 1995].

The formalism about *Joint Intentions* [Levesque et al. 1990] focuses on teammates' joint mental states. A team jointly intends a team action if team members are jointly committed to completing such team action, while mutually believing that they were doing it [Tambe 1997]. *Joint Intentions* assume that the agents are modeled in a dynamic multi-agent environment, without complete or correct beliefs about the world or other agents. They have changeable goals and their actions may fail. Thus, the teammate jointly commits to

[1] A cooperative effort realized for teammate to achieve a goal.

make public the knowledge if a goal becomes achieved, impossible to be reached, or irrelevant.

In contrast with *Joint Intentions*, the concept of *SharedPlans* [Grosz and Kraus 1996] relies on a novel intentional attitude, *intending-that*. An individual agent's *intending-that* is directed toward its collaborator's action or toward a group's joint action. It is defined via a set of axioms that guide an individual to take actions, including communication actions, that either enable or facilitate its teammates, subteam or team to perform the assigned tasks [Tambe 1997].

Based on these theories, important teamwork infrastructures have been developed, like *STEAM* [Tambe 1997] and *Machinetta* [Steven 2005]. *STEAM* was developed to provide flexibility and reusability of agent coordination through the use of a general model of teamwork. Such model exploits the autonomous reasoning about coordination and communication. Previous works had failed in fulfilling responsibilities or in discovering unexpected opportunities, once they had used precomputed coordination plans, which are inflexible. Indeed, they had been developed for a specific domain, what hampers the reuse. *STEAM* is specially based on *Joint Intentions* in order to make the basic building blocks of teamwork, and it uses *Shared Plans* to avoid an agent undo the actions of another.

Machinetta is a proxy-based integration infrastructure where there is a beneficial symbiotic relationship between the proxies and the team members. The proxies provide teamwork models reusable for heterogeneous entities. *Machinetta* provides each agent with a proxy and the proxies act together in a team. The *Machinetta* project was build over previous works like *STEAM* and *TEAM-CORE proxies* [Pynadath and Tambe 2003]. The pair agent-proxy execute Team-Oriented Programs(TOP) [Scerri et al. 2004], abstract team plans that provide high-level description of the activities to be performed. Such programs specify joint plans of the team, but do not contain all of the required details of coordination and communication. The team plans are reactively instantiated and they allocate roles to capable agents.

3 The Environment

In this work, the case study is *Unreal Tournament 2004 (UT2004)*, a multiplayer *First-Person Shooter*(FPS) game. The classical team-based game type *Capture The Flag* was chosen to evaluate the team coherence, because it is easy to measure the results and simple to identify the team behavior. The *GameBOTs* [Andrew Scholer 2000] [Kaminka et al. 2002] project is used to communicate the developed bot with the game.

The *GameBOTs* project works as an extended game control that enables the character to be controlled via socket TCP. Thus, it is possible to develop agents to control characters using any language with socket support. This modification offers four new gametypes, *Remote Team DeathMatch*, *Remote Capture The Flag*, *Remote Double Domination* and *Remote Bombing Run*, that can be used as environment to test bots. Using the remote connection, the bot has perception about the environment, similar to the human player. Figure 1 shows the way points seen by the bots.

The UT2004 game provides an environment with the following features [Russel and Norving 2004]:

- **partially observable** - the agent, using its perception, can not access all relevant environment states;
- **stochastic** - the next state of the environment is not completely determined by current state and actions selected by the agents;
- **nonepisodic** - the quality of bot's actions depends on its previous actions;
- **dynamic** - the environment state can change between the agent's perception and agent's actions;
- **continuous** - the number of environment states is unlimited;
- **multi-agent** - there is cooperative and competitive interaction between agents.

Figure 1: *Waypoints seen by the bot.*

When the bot is connected to the game, sensory information is received through the socket connection and the agent can act by sending commands to the game. This commands define the character behavior inside the virtual world, controlling moves, attack, and communication with other characters. The interaction between an agent and UT2004 is realized through messages exchange. The perception messages may be synchronous or asynchronous, and their complete description can be found in [Andrew Scholer 2000].

The synchronous messages arrive in batch, with a configurable interval. They are formed by visual information and the state of the agent. The asynchronous messages indicate events occurred in the environment. This events may be a damage taken, hit an obstacle, broadcast message from another agent, and other information about the game.

Actions performed by the characters are results of commands sent by the agent. An important issue about this is that the commands have persistent effects, it means that a move command will execute until the bot achieves a point or until it receives a new command that conflict with the previous, like move in another direction. Thus, the command scheme is near to human interface, where the human player need release a mouse button to stop the attack.

4 TWProxy

In order to meet the requirements for the development of teams of agents in the games domain, this work introduces the *TWProxy*, a new lightweight and efficient infrastructure to provide coordination to teams of heterogeneous agents. The *TWProxy* is based on the *Joint Intentions* [Levesque et al. 1990] formalism and it is inspired on *Machinetta*, adding new important features, including: (i) a simple and extensible language to describe plans and beliefs, (ii) atomic communication between agents, (iii) reuse of terminated plans, and (iv) plans that are invariant to the number of agents.

In our approach, summarized in Figure 2, each agent is related to a proxy that performs the communication and the multi-agent planning. The team coordination is achieved using role allocation. Each proxy knows about the capabilities of other teammates and when plan preconditions are achieved, and roles are assigned according the current plan. When the plan postconditions are achieved, the teammate can release the role. The proxy does not tell how to execute the role, it just deliberates about what to do. Because of the high-level coordination, a flexible teamwork is possible, leaving the details about role execution with the agent.

The model about role allocation is similar to the common teamwork in the real world. When people need to execute some action together, they usually divide the task in subtasks that each one can perform. In the real world, people also use the role allocation to coordinate actions. Thus, the responsibility is distributed, and if there exists commitment between the teammates, the entire group

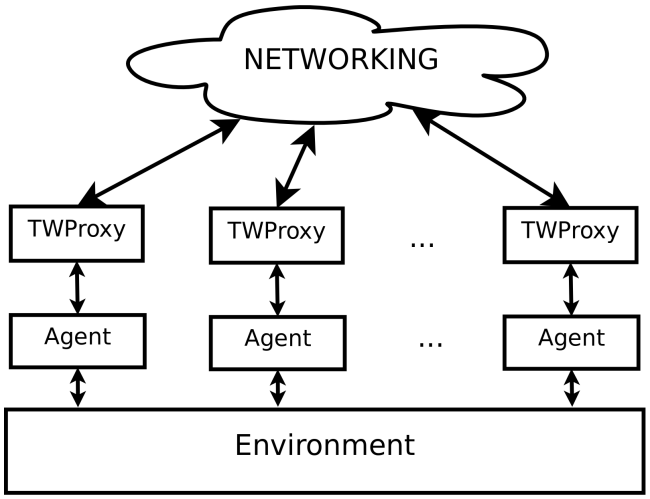


Figure 2: *Teamwork model based on proxy.*

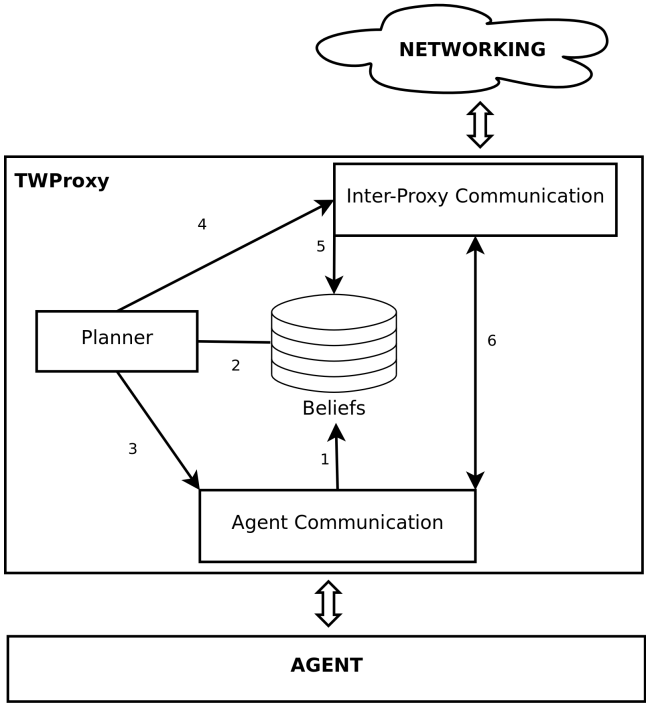


Figure 3: *TWProxy organization.*

acts consistently.

TWProxy also provides the functionality of distributed blackboard because it keeps the shared beliefs consistent. This belief repository helps the agents to take a decision. Due to the environment to be partially observable, the agents have sensory restrictions. Therefore, the *TWProxy* can store some shared information that is not directly sensed by the agent.

Figure 3 shows the *TWProxy* internal organization. Link 1 indicates that information from the agent can update the knowledge base. The set of beliefs can be used by the planner (link 2) that can deliberate about role allocation(link 3 and 4). Information from other proxies can update the set of beliefs (link 5) and an agent can share information using the *TWProxy*(link 6).

4.1 Describing Plan and Beliefs.

The initial agent beliefs and its plans are stored in a structured file described with the language shown in listing 1. This file contains agent beliefs about: capabilities, team composition and the specific domain. It also describes the roles and the high-level team plan.

Teams of Agent-proxy pairs execute Team-Oriented Programs (TOPs) [Scerri et al. 2004]. These TOPs specify the joint plans of the team, and the inter-dependencies between those plans, but do not contain all the required details of coordination and communication.

Listing 1: *Grammar of the description language*

```
          <blf> ::= <block> |
                    <block> <blf>

        <block> ::= "belief" <belief_block> |
                    "role" <role_block> |
                    "plan" <plan_block>

 <belief_block> ::= <id> "{" <inside_belief> "}"

<inside_belief> ::= <attr_value> |
                    <attr_value> <inside_belief>

   <attr_value> ::= <id> ":" <value> ";"

        <value> ::= <id> |
                    <id> "," <values> |
                    <number> |
                    <number> "," <value> |
                    "true" |
                    "false"

   <role_block> ::= <id> "{" <inside_role> "}"

  <inside_role> ::= <attr_value> |
                    <attr_value> <inside_role>

   <plan_block> ::= <id> "{" <inside_plan> "}"

  <inside_plan> ::= "roles" ":" <role_values> ";"
                     <pre_block> <post_block>

  <role_values> ::= <id> |
                    <id> "," <role_values> |
                    <id> "+" |
                    <id> "+" <role_values>

    <pre_block> ::= "precondition" "{" <expr> "}"

   <post_block> ::= "postcondition" "{" <expr> "}"

         <expr> ::= "(" <expr> ")" |
                     <id> "." <id> <comp> <number> |
                     <id> "." <id> <comp> <id> "." <id> |
                     <id> "." <id> <comp> "true" |
                     <id> "." <id> <comp> "false" |
                     <expr> "|" <expr> |
                     <expr> "&" <expr>

           <id> ::= "[a-zA-Z][a-zA-Z_0-9]*"

       <number> ::= "[+-]?[0-9]+|[+-]?[0-9]+\.[0-9]+"

         <comp> ::= "==" | "!=" | "<" | "<=" | ">" | ">="
```

Listing 2,3,4,5 and 6 show examples of different parts of the beliefs file for the *Capture The Flag* domain. Listing 2 defines that the agent *agent001* can perform the role *CTFProtectTheBase* if it is not executing another role.

Listing 2: *An example about capability belief*

```
belief capability_agent001_CTFProtectTheBase {
        type: capability;
        rapId: agent001;
        load: 100;
        roleId: CTFProtectTheBase;
}
```

Listing 3 shows the belief about the team composition, and in this case also describes that agents should communicate with a peer to peer connection.

Listing 3: *An example about team composition*

```
belief teamComposition {
    type: teamComposition;
    members: agent001, agent002, agent003;
    myself: agent001;
    agent001_host: localhost;
    agent002_host: localhost;
    agent003_host: localhost;
```

```
    agent001_port1: 6001;
    agent002_port1: 6002;
    agent003_port1: 6003;
    agent001_port2: 7001;
    agent002_port2: 7002;
    agent003_port2: 7003;
}
```

The domain specific belief is also described in the beliefs file, as shown in listing 4. The belief contents is flexible to describe new properties, because the new attributes are handled dynamically.

Listing 4: *An example about domain specific belief*

```
belief flag_enemy{
        type: flag;
        owner: enemy;
        have: false;
}

belief flag_friend{
        type: flag;
        owner: friend;
        have: true;
}
```

It is possible to define meta-information about role allocation. In the example of listing 5, *priority* helps to solve allocation conflicts, *before* defines actions to execute before allocation, and *preference* defines who should receive this role. These parameters of the meta-information should be handled through inheritances of base classes in the TWProxy, otherwise, just *priority* has effect.

Listing 5: *A role example*

```
role CTFRecoverMyFlag {
        priority: 3;
        before: getAllRapPosition;
        preference: nearToTheEnemyBase;
}
```

Listing 6 shows an example of a plan for the *Capture The Flag* context. When a team has its flag and it does not have the enemy flag, plan *p1* is activated launching a role allocation process. This plan specifies two roles to be allocated, *CTFProtectTheBase* and *CTFCaptureTheFlag*, where the second is flexible for more than one allocation, it is indicated by signal + in the end of the role's name. If a team has four free agents, one performs *CTFProtectTheBase*, while three execute *CTFCaptureTheFlag*.

Listing 6: *A plan example*

```
plan p1{
        roles: CTFProtectTheBase, CTFCaptureTheFlag+;

        precondition{
                ( flag_friend.have == true ) &
                ( flag_enemy.have == false )
        }

        postcondition {
                ( flag_friend.have == false ) |
                ( flag_enemy.have == true )
        }
}
```

4.2 Communication

The inter-proxy communication assumes the usage of atomic communication over a network without shared memory. To solve the problem of atomic communication, *TWProxy* was developed over a simple atomic broadcast layer. However, it is possible to extend base classes to develop a new atomic broadcast layer. Thus, the sharing beliefs is done by atomic broadcast [Défago et al. 2004], and just the specific order messages(like to assign a role) is performed via unicast.

Two communication channels are used by each instance of TWProxy. The first channel is used to share knowledge and the second is used to exchange information about role allocation. The messages sent through these channels are serialized objects that are recomposed by a factory in the other side.

4.3 Planner

The *TWProxy* planner uses team leader to launch a new plan. The first team leader is defined statically but it may be changed dynamically at run-time. The usage of team leader avoids to solve conflict problems in role allocation, saving time to react to the environment changes. The team leader has updated beliefs and it can launch a plan consistently, because everyone is committed to share relevant information.

The planner is like a rule-based system. Every plan contains rules to activate it (the preconditions) and to stop it (the postcondition). These rules are checked using the current knowledge. Roles are assigned in the activation of a plan, and in the stopping, the involved agents can release their roles.

4.4 Agent Interface

Each individual agent is attached to a *TWProxy* forming a social agent. The communication between the individual agent and the *TWProxy* is defined by a flexible interface, allowing interaction with several kind of agents.

The interface between proxy and agent is defined by a base class that must be extended to implement this communication. Such interface is based on message exchange and it can be adjusted to each kind of agent. In a new domain, the interface with the agent is the main modification needed. Thus, it is possible to make a team, to a new domain, developing just the interface between agent and proxy, and the team program. The individual agent does not need to have social commitment, because the *TWProxy* does this for it.

5 Experiments and Evaluation

In our experiments, the agents have been developed using the framework IAF [Monteiro and dos Santos 2007], that implements a hybrid agent architecture [Monteiro 2005]. Such agents communicate with the game controlling a player. For these experiments every agent has the same capabilities. Thus, just the teamwork is different. In order to compare the *TWProxy* with another proxy, *Machinetta* is used, because of large number of successful applications [Scerri et al. 2003a] [Schurr et al. 2005] [Scerri et al. 2003c] [Scerri et al. 2004] [Chalupsky et al. 2001] [Scerri et al. 2001] [Scerri et al. 2003b].

A great concern in the teamwork problem is to avoid teamwork inconsistency. In the *Capture The Flag* domain, a teamwork inconsistency may be the following: Given a team composed by agents *A*, *B* and *C*, if either *A* or *B* are trying to recover a flag that was recovered by *C*, then the team is in an inconsistent state. Every team in a dynamic environment will stay in a inconsistent state during some period of time. The goal is to minimize inconsistency states.

The first step to develop a team is the modeling of the problem in the multi-agent context. It is necessary to divide the team actions in roles that will be performed by the agents. After that, it is possible to develop, in the agent, the abilities to execute the roles. The next step is the definition of knowledge and plans under *Team-Oriented Programs*.

This work shows two kinds of experiments. The first measures the time of teamwork inconsistency, evaluating the elapsed time between an important change in the environment and the team adoption of a new strategy. The second kind of experiment are matches that intend to show teamwork efficiency using the TWProxy, in order to show that it is possible to use the multi-agent paradigm and distributed artificial intelligence in the domain of modern games.

In the experiment of the time of teamwork inconsistency, agents send a sequence of changes about the environment, and the time of team reaction using *TWProxy* and *Machinetta* is calculated. The battle experiments play *Capture The Flag* where one team uses *TWProxy* and the other team uses *Machinetta*, self game BOTS or human players.

In the battle, the game UT2004 is used to provide the *Capture The Flag* gameplay. Thus, it is possible to evaluate the team actions,

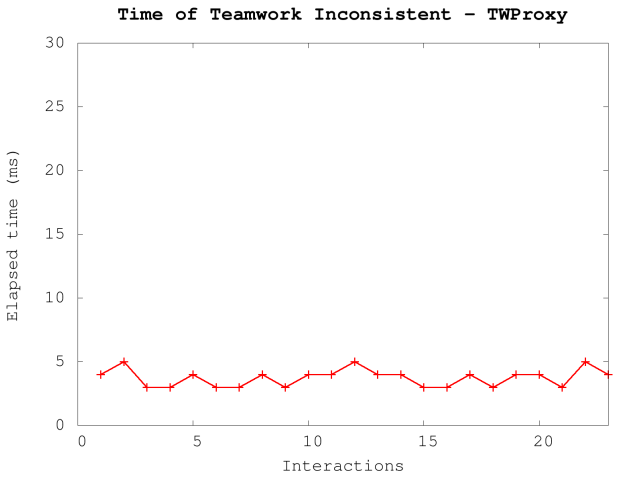


Figure 4: *Time of Teamwork Inconsistency - TWProxy.*

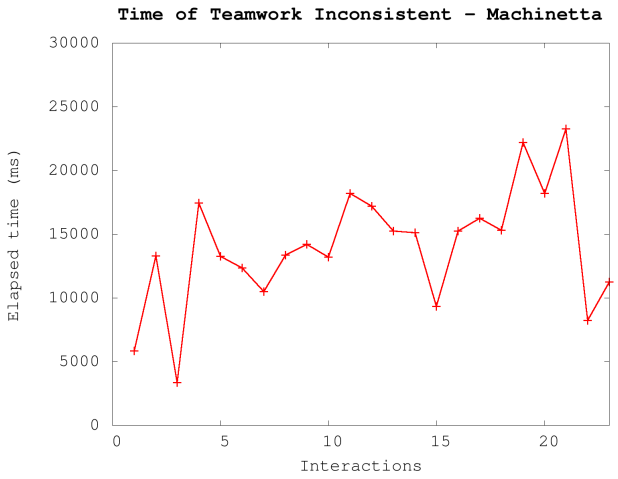


Figure 5: *Time of Teamwork Inconsistency - Machinetta.*

because it is easy to see which role the agent is performing, and it is also possible to measure the teamwork efficiency using the game score.

The team, that plays *Capture The Flag* using the *TWProxy*, has just four plans, that are activated when the state of a flag changes. Such plans generate the following team behavior:

1. when a team has its own flag but it has not the enemy's flag, someone needs to protect the base while others go to capture the enemy's flag;
2. when a team has both, its own and the enemy's flag, all agents must go back to their base;
3. when a team does not have its own flag but it has the enemy's flag, the agent that has the enemy's flag must go back to its base and the others should recover the team's flag;
4. when the team has no flags, every agent must search any flag.

The *Machinetta* proxy showed some limitations in the experiments, and a critical problem for this context was the impossibility to reuse a terminated plan. Thus, it was necessary to create many redundant plans to keep the bots playing the game. Other problem was the memory and the CPU usage, being impracticable to apply in a real game. However, *Machinetta* has many successful applications, being interest for comparison purposes, as will be shown in the next section.

5.1 Evaluating Time of Teamwork Inconsistency

The experiment about time of Teamwork Inconsistency(TTI) intended to evaluate the ability of *TWProxy* to react in (soft) real-time to environment changes. In order to compare its performance, *Machinetta* was exposed to the same situation. A group of three agents was updating their beliefs about the flag status, and an external entity was listening the communication between agents and proxies. Therefore, this external entity was computing the time between the environment change and the role allocation to the whole team.

The sequence of environment changes was the same for *Machinetta* and *TWProxy*. It was composed by 23 updates and it represented a whole match. Each proxy played 40 matches, in order to get a general behavior of the TTI. This experiment was performed in a single machine. For this reason, the network latency did not appear in the results.

The results shown in Table 1 describe the mean(μ) and the standard deviation(σ) of the experiments set. The measure unit is milliseconds and the time represents the total time between a new belief received and the modification of the strategy. *TWProxy*, in this experiment, achieved a mean of 3.9ms, while *Machinetta* got 12182.40ms. In other words, *TWProxy* was more than three thousand times faster than *Machinetta* to reach a consistent state. The

Table 1: *Mean and standart deviation of experiment.*

-	**Elapsed time(ms)**	
-	μ	σ
TWProxy	3.9	0.79
Machinetta	12182.40	7159.01

standard deviation shows how stable *TWProxy* execution was comparing with *Machinetta*.

Table 2 presents an example of values of TTI for one match. In this table it is possible to see the difference of magnitude between *TWProxy* and *Machinetta*. Figures 4 and 5 show the same example of Table 2 in a graphical format , in order to show the stability of the *TWProxy* in this application. It is important to say that these graphics are in different scales in an effort to compare the data dispersion.

The performance and stability of *TWProxy* against *Machinetta* was the main result from this experiment. In the midst of this circumstance, *Machinetta* also presented a high usage of memory (more than 100MB per proxy), making it difficult to use in modern games. *TWProxy*, in these experiments, did not use more than 1.6MB of memory. The key of these results relies on the kind of applications for which *Machinetta* was designed. While *TWProxy* design was concerned in achieve real-time requirements, *Machinetta* was originally developed to allow humans to participate in teams with artificial agents. Several points may explain this efficiency, including: the usage of C++ instead of Java; optimization in access to the beliefs; and mainly the fewest conflicts to resolve, because the communication with atomic broadcast guarantees the total order of message and the usage of team leader avoids the role allocation conflict.

5.2 Teamwork Efficiency

The second kind of experiment evaluates the teamwork efficiency using the gameplay *Capture The Flag*. Each experiment spent 15 minutes and it was played in order to assess different aspects. The first experiment evaluates *TWProxy*'s efficiency against *Machinetta* using the same agents in a battle, but with different proxies. The second experiment was composed by battles between a team using the *TWProxy* and the default bots from UT2004. The third experiment shows a battle between a team with *TWProxy* and an human team.

5.2.1 TWProxy-Team vs Machinetta-Team

The aim of this experiment is to assess how better can a team's efficiency be by changing just the proxy that provides the teamwork. Both teams had the same agents and they used the same strategy, the only difference was the teamwork proxy. Figure 6 shows the

Table 2: *An example of TTIs measured in one match played by TWProxy and Machinetta.*

Interaction	Elapsed time(ms)	
-	*TWProxy*	*Machinetta*
01	4	5838
02	5	13307
03	3	3360
04	3	17463
05	4	13256
06	3	12369
07	3	10497
08	4	13358
09	3	14211
10	4	13212
11	4	18211
12	5	17195
13	4	15251
14	4	15138
15	3	9337
16	3	15262
17	4	16251
18	3	15299
19	4	22208
20	4	18207
21	3	23274
22	5	8248
23	4	11264

result of a match, in which the team using *TWProxy* won but the game was very disputed with just two points of difference. The individual agents using *Machinetta* had higher score than those using *TWProxy*, however, the effort of the team should precedence over individual actions. The key point is that the best for the team is not necessarily the best for each individual.

5.2.2 TWProxy-Team vs UT2004-Team

An important point in the use of multi-agent teamwork in modern games is to decide if it is possible to use distributed artificial intelligence in a domain in which nowadays the centralized solution is simpler. The centralized solution can access the full state of game and decide the best to do. However, in a distributed approach, the full state is not accessible, and each entity needs to cooperate in an effort to coordinate their actions. Thus, this experiment confronts these two approaches, the distributed versus the centralized.

The battle was composed by bots from UT2004 in their default skill level and agents using the *TWProxy*. Figure 7 is a example of result in this experiment. The difference in scores was very high in favor of the team with the TWProxy, both individually and in team. Therefore, it shows the possibility of usage of the distributed approach which also creates a new possibility of gameplay.

5.2.3 TWProxy-Team vs Human-Team

The last experiment aims to evaluate the team coherence, putting the team with *TWProxy* against a human-team. A human-team has the ability to explore some coherence fault in a team of bots. For this reason, this experiment was interesting.

Although the agents were using a static set of strategies, the human-team failed to explore any teamwork inconsistency. The battle was very disputed even though the human players had average skill in First Person Shooter. Figure 8 shows the result of a match between TWProxy-Team and Human-Team, where the balance can be seen.

6 Conclusion

This work shows that it is possible to overcome the deficiencies about social behavior in team-based games using the teamwork approach. With this purpose, this work presents *TWProxy*, a team

Figure 6: *Results of a Capture The Flag match between TWProxy-Team and Machinetta-Team.*

Figure 7: *Results of a Capture The Flag battle between TWProxy-Team and UT2004Bots-Team.*

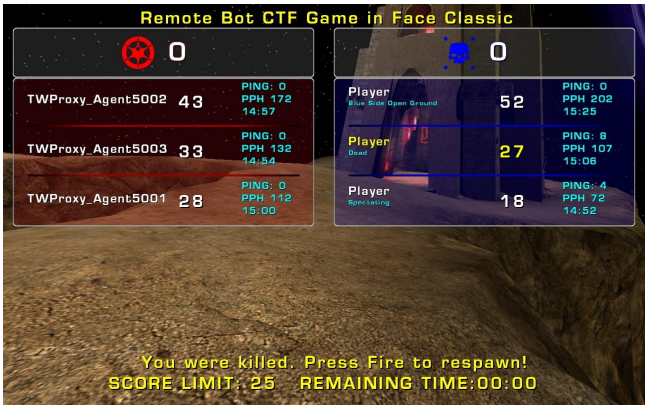


Figure 8: *Results of a Capture The Flag match between TWProxy-Team and Human-Team.*

proxy for dynamic environment with real-time requirements. In order to evaluate *TWProxy*, teams using it were developed, and the agents acted coherently as a team, without to use a central control. Such application creates new possibilities in the gameplay, specially in distributed games like massive multiplayer online.

Using *TWProxy* the group activities are described by a high-level team-oriented programming language, giving more flexibility and autonomy to the team to perform a task. Besides that, agents have a distributed blackboard provided by *TWProxy*, due to the maintenance of beliefs between proxies. The belief description language is also extensible, being able to use in several domains.

The experiments show that it is feasible to use a distributed approach and also the advantages of *TWProxy* face to *Machinetta*. Indeed, the experiments have shown the limitations of *Machinetta* for this specific domain. *TWProxy* had good results in these experiments, showing its stability and efficiency. Besides, *TWProxy* has new important features to team composition, like reuse of plans, easy and extensible language to describe team plan, atomic communication, invariant plans to the number of agents, and extensibility in role allocation using knowledge from a specific domain. *TWProxy* is also more lightweight than *Machinetta*. In the matches experiments, it is possible to conclude that: less time of team inconsistency gives better performance to a group; a distributed approach is feasible and its performance is comparable to team centralized control; and it is feasible to keep the team coherence making believable teams.

References

ANDREW SCHOLER, G. K., 2000. Gamebots. Último acesso em 01 de dezembro de 2006.

CHALUPSKY, H., GIL, Y., KNOBLOCK, C., LERMAN, K., OH, J., PYNADATH, D., RUSS, T., AND TAMBE, M., 2001. Electric elves: Applying agent technology to support human organizations.

DE BYL, P. B. 2004. *Programming Believable Characters for Computer Games*. Charles Development Series.

DÉFAGO, X., SCHIPER, A., AND URBÁN, P. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*, 4, 372–421.

GROSZ, B. J., AND KRAUS, S. 1996. Collaborative plans for complex group action. *Artificial Intelligence 86*, 2, 269–357.

GROSZ, B., AND KRAUS, S., 1998. The evolution of sharedplans.

HILL, R., CHEN, J., GRATCH, J., ROSENBLOOM, P., AND TAMBE, M. 1997. Intelligent agents for the synthetic battlefield: A company of rotary wing aircraft. In *Innovative Applications of Artificial Intelligence (IAAI-97)*.

JENNINGS, N. R. 1995. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence 75*, 2, 195–240.

JONES, R. M., LAIRD, J. E., NIELSEN, P. E., COULTER, K. J., KENNY, P. G., AND KOSS, F. V. 1999. Automated intelligent pilots for combat flight simulation. *AI Magazine 20*, 1, 27–41.

KAMINKA, G. A., VELOSO, M. M., SCHAFFER, S., SOLLITTO, C., ADOBBATI, R., MARSHALL, A. N., SCHOLER, A., AND TEJADA, S. 2002. Gamebots: a flexible test bed for multiagent team research. *Commun. ACM 45*, 1, 43–45.

KAMINKA, G. A. 1999. The robocup-98 teamwork evaluation session: A preliminary report. In *RoboCup*, 345–356.

LAIRD, J. E., NEWELL, A., AND ROSENBLOOM, P. S. 1987. Soar: an architecture for general intelligence. *Artif. Intell. 33*, 1, 1–64.

LEVESQUE, H. J., COHEN, P. R., AND NUNES, J. H. T. 1990. On acting together. In *Proc. of AAAI-90*, 94–99.

MARSELLA, S., ADIBI, J., AL-ONAIZAN, Y., KAMINKA, G. A., MUSLEA, I., AND TAMBE, M. 1999. On being a teammate: experiences acquired in the design of RoboCop teams. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, ACM Press, Seattle, WA, USA, O. Etzioni, J. P. Müller, and J. M. Bradshaw, Eds., 221–227.

MARSELLA, S., TAMBE, M., ADIBI, J., AL-ONAIZAN, Y., KAMINKA, G. A., AND MUSLEA, I. 2001. Experiences acquired in the design of robocup teams: A comparison of two fielded teams. *Autonomous Agents and Multi-Agent Systems 4*, 1/2, 115–129.

MONTEIRO, I. M., AND DOS SANTOS, D. A. 2007. Um framework para o desenvolvimento de agentes cognitivos em jogos de primeira pessoa. In *Anais do VI Workshop Brasileiro de Jogos e Entretenimento Digital*.

MONTEIRO, I. M. 2005. Uma arquitetura modular para o desenvolvimento de agentes cognitivos em jogos de primeira e terceira pessoa. In *Anais do IV Workshop Brasileiro de Jogos e Entretenimento Digital*, 219–229.

NAIR, R., ITO, T., TAMBE, M., AND MARSELLA, S., 2000. Robocup-rescue: A proposal and preliminary experiences.

PYNADATH, D. V., AND TAMBE, M. 2003. An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems 7*, 1-2, 71–100.

RUSSEL, S., AND NORVING, P. 2004. *Inteligência Artificial - Tradução da segunda edição.* Editora Campus.

SCERRI, P., PYNADATH, D., AND TAMBE, M. 2001. Adjustable autonomy in real-world multi-agent environments. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, ACM, New York, NY, USA, 300–307.

SCERRI, P., PYNADATH, D., JOHNSON, L., SCHURR, R., SI, M., AND TAMBE, M., 2003. A prototype infrastructure for distributed robot-agent-person teams.

SCERRI, P., PYNADATH, D., AND TAMBE, M., 2003. Towards adjustable autonomy for the real world.

SCERRI, P., JOHNSON, L., PYNADATH, D. V., ROSENBLOOM, P., SCHURR, N., SI, M., AND TAMBE, M., 2003. Getting robots, agents and people to cooperate: An initial report.

SCERRI, P., PYNADATH, D., SCHURR, N., FARINELLI, A., GANDHE, S., AND TAMBE, M. 2004. Team oriented programming and proxy agents: The next generation. In *Proceedings of 1st international workshop on Programming Multiagent Systems*.

SCHURR, N., MARECKI, J., TAMBE, M., AND SCERRI, P., 2005. Towards flexible coordination of human-agent teams.

STEVEN, N. S., 2005. Evolution of a teamwork model. Disponível em citeseer.ist.psu.edu/679191.html. Acesso em: Outubro 2007.

STONE, P., AND VELOSO, M. M. 1998. Task decomposition and dynamic role assignment for real-time strategic teamwork. In *Agent Theories, Architectures, and Languages*, 293–308.

SYCARA, K., AND SUKTHANKAR, G. 2006. Literature review of teamwork models. Tech. Rep. CMU-RI-TR-06-50, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, November.

TAMBE, M., AND ZHANG, W., 1998. Towards flexible teamwork in persistent teams.

TAMBE, M., AND ZHANG, W. 2000. Towards flexible teamwork in persistent teams: Extended report. *Autonomous Agents and Multi-Agent Systems 3*, 2, 159–183.

TAMBE, M. 1997. Towards flexible teamwork. *Journal of Artificial Intelligence Research 7*, 83–124.

YEN, J., YIN, J., IOERGER, T. R., MILLER, M. S., XU, D., AND VOLZ, R. A. 2001. CAST: Collaborative agents for simulating teamwork. In *IJCAI*, 1135–1144.

Neuronal Editor Agent for Scene Cutting in Game Cinematography

Erick B. Passos
Media Lab - UFF

Anselmo Montenegro
Media Lab - UFF

Vinicius Azevedo
UFSM

Cezar Pozzer
UFSM

Vitoria Apolinaro
Media Lab - UFF

Esteban W. G. Clua
Media Lab - UFF

Figure 1: *Shot library of an editor agent*

Abstract

The use of cinematography techniques in games aims to provide high level abstractions for operating the virtual camera based on concepts borrowed from the movie industry such as scene, shot and line of action, among others. One usual approach is the development of agents to execute tasks that are similar to their counterpart in a real movie set: director, editor and cinematographer. However, a game is an interactive application and resembles a live TV-show, where the actions taken by all the actors is not known previously. In such scenario, the role of the editor is of great importance since he is the one who ultimately decides what point of view should appear on the screen. In the context of game cinematography, most previous works have proposed ways of mapping scene concepts and automatically controlling the virtual camera but without paying much attention on the role of an editor agent. Our paper shows an intelligent editor agent that uses neuronal network classifiers to decide shot transition and has an intuitive user interface to the learning mechanism.

Keywords:: Game Cinematography, Virtual Camera, Shot Transition, Cut, Neuronal Networks

Author's Contact:

{epassos,anselmo,esteban}@ic.uff.br
socorosca@gmail.com
vitoriamachay@hotmail.com
pozzer@inf.ufsm.br

1 Introduction

Visual simulations such as games have always relied on cutting edge real-time graphics. However, the search for simulation complexity has also given rise to new challenges in other areas such as artificial intelligence, physics and also storytelling. All the efforts in such areas account for the level of immersion in the virtual world and, in this trend, intelligent real-time camera handling plays an important role. Good use of the virtual camera has gaining even more importance because of trends such as games with spectators [Drucker et al. 2002], the use of real-time game engines to create films [Elson and Riedl 2007; Morris et al. 2005] and also storytelling research [Courty et al. 2003; Amerson et al. 2005; Pozzer 2005]. Previous research has been applying movie industry standards and concepts, such as scenes and shots, to video games in order to create higher levels of abstraction to manipulate the virtual camera.

The most common concept in game cinematography is that of a film idiom, which represents the most usual way to present an specific type of scene event such as an *over the shoulder shot* for a dialogue sequence between two characters, or a *helicopter-mounted camera* for a fast paced car chase in a highway. Film idioms are good to represent specific camera behaviors that are then isolated and treated as a problem of its own with different possible solutions. Complete solutions for game cinematography, however, have to deal with other problems as well and are sometimes organized as agents that represent the various roles people play in a movie set [Hawkins 2004]. The most common approach is to consider three types of agents: director, editor and cinematographer.

Usually, the director agent is responsible for analyzing the scene and proposing film idioms to present it to the player. The editor agent is responsible for choosing which one to use, while the cinematographer agent directly controls the virtual camera [ref-hawkins]. It is important to remember that games are different from films in the sense that, being interactive applications, there is no prior knowledge of future events. A better analogy is to think of them as live TV-shows, where the scene and the actors are known, but not all the dynamic dialogues, events and actions taken are. In this kind of environment, one of the most important roles is that of the editor, who ultimately decides what to present to the audience. Based on these concepts, a distributed multi-agent system was proposed in [ref-short-paper-ours] to deal with the issues related to massive online games.

In such system, the role of the editor agent is of great importance, and these are the main concerns related to its implementation:

- What information should be available from the scene in order to enable good decisions;
- How to decide which of the available film idioms to use at each time;
- When to cut from one shot to the other without breaking frame-coherence.

In general, the ultimate goal is to create an agent that mimics the behavior of a human editor. In this paper we present the details of an editor agent that uses neuronal networks to enable consistent shot transitions in dynamic environments and also:

- Provides for a very fast learning mechanism based on real-time teaching by example;

- Requires minimal, not film-structured, information from the scene;
- Uses an intuitive, edit-like, real-time interface for the learning mechanism that requires no programming experience from the user.

The rest of the paper is organized as follows: section 2 compares our approach with previous research. Section 3 presents the system architecture and explains some details about the different agent layers. Section 4 brings a detailed look at the editor agent implementation, while section 5 details the game prototype developed to the experiments while section 6 analyzes these experiments and results. Finally, section 7 concludes the paper and presents our future work on the subject.

2 Related Work

A good amount of previous works were already dedicated to applying cinematographic techniques or other intelligent mechanisms of virtual camera control in video games. The majority of this research proposes the creation of higher level mechanisms for controlling the virtual camera with the adoption of cinema concepts, constructs and language such as scenes, shots, cuts, directors, editors and cinematographers [Drucker and Zeltzer 1995; wei He et al. 1996; Christianson et al. 1996; Halper and Masuch 2003; Hawkins 2004; Amerson et al. 2005; Tomlinson 2000; Hornung et al. 2003]. Some of these works use the concept of a *film idiom*, which encapsulates the combined knowledge of several personal roles in a traditional filming set. In the context of a complete solution, however, film idioms are good to solve only two parts of the problem: direct manipulation of the virtual camera; and creating a simple API for communicating with other layers.

In a work by Hawkins [Hawkins 2004], a three-layer architecture is proposed to split the issues through different agents categories: directors, editors and cinematographers. The director agent recognizes actions in the scene and suggests shot options to an editor agent, that chooses the best way to present them. This later issue is the responsibility of the third layer: a cinematographer agent, which directly operates the virtual camera. Bringing this organization to virtual cameras in computer graphics makes it easier to decouple the different modules needed by such system. Our work proposes a novel approach for the design and implementation of an editor agent that doesn't require the scene information to be structured in the form of scene events.

Other previous works are foundations on how to solve the issues related to the implementation of the virtual camera manipulation itself. Drucker [Drucker and Zeltzer 1995] studies the basic elements that compose and motivate the movements of a virtual camera, ending up proposing a agent-based framework to solve this task. Camera modules are designed as independent tasks that can be used independently or combined. Hermann e Celes [Hermann and Celes 2005] proposed a system that dynamically translates and orientates the camera with the use of physical constraint satisfaction. This later work is a good approach for implementing cinematographer agents, being complementary to our proposed editor.

Kneafsey and McCabe [Kneafsey and McCabe 2005] create an intelligent cinematography system for first person shooters based on a commercial engine. Their system dynamically switches camera shots based on specific information extracted from the 3D scene and use a Finite State Machine (FSM), augmented with fine grained activation rules, to represent the knowledge used to decide which shot to use at each time. Although appropriate to represent this king of decision making knowledge, FSMs lack the ability to learn, and the user has manually specify the conditions that trigger each shot type.

He [wei He et al. 1996] also uses a FSM to represent film idioms but organizes them as a hierarchical structure. Each film idiom includes activation information as well as camera-modules, which are the equivalent to our cinematographer agent. Other works [Christianson et al. 1996; Amerson et al. 2005] also use hierarchical data structures to encode film idioms and provide a decision mechanism for shot transitions. Our system uses neuronal networks to store this knowledge and also present a very intuitive learning interface.

Hornung [Hornung et al. 2003] proposes an agent based system that drives the virtual camera based on relevant scene information of the cut-scenes from the game Half-Life. The editor agent of this system also use neuronal network classifiers to choose shot types, but uses a different approach (from ours) in that it receives information about the scene in the form of narrative events, containing coded information such as actors and targets. More important, however, is that it needs data to be translated from the scene domain to that of a narrative: emotional level, action level, among others. Because of this dependency on domain translation, this approach relies on a very complex director agent and is only appropriate to narrative applications that can provide such semantic information. Our editor agent does not need any domain translation and only requires input normalization.

Our work provides for a shot transition system that is more suitable to games, but its also important to mention research on storytelling since intelligent cinematography is one of the building blocks of this area. Pinhanez [Pinhanez 1999] studied the foundations for the representation and recognition of action events in interactive systems and developed a discrete mechanism and algorithm for the temporal structure of such actions called PNF (past, now, future) networks. This propagation algorithm is used to recognize the actions in an interactive context in replacement of other temporal representations such as Finite State Machines. Our system does not rely on action recognition to learn the rules for shot transition.

Pozzer [Pozzer 2005] developed an architecture for the generation and representation of dynamic interactive stories, including the presentation on a 3D engine and cinematographic techniques. The goal of the work is to provide an integrated tool to manage story generation for the brazilian digital TV system. In a more recent work, Guerra [Guerra 2008] proposes the term *story engineering* for this process, that is divided into three sub-problems: ontologies for story-driven data; story generation and storytelling, in which intelligent cinematography is included. Even being game-driven, our proposed editor agent is a suitable option for the shot transition issue, given its learning mechanism and intuitive interface. In the next two sections, our game cinematography system will be explained together with the editor agent's architecture.

3 Cinematography System Architecture

The proposed system architecture comprises three separate layers modeled as agents: scene mapper, editor and cinematographer. The role of the scene mapper agent is to gather relevant information about the 3D scene and to map it to normalized values (between -1 and 1). This approach provides a knowledge mapping of a scene data converted to a float-valued domain, which is independent of the specific 3D environment implementation, enabling a simple data structure to be used by the editor agent, which is called input-hub. This data structure does not require information to be coded with movie semantics such as other works [Hornung et al. 2003].

The movie semantics are not required because our shot-transition system relies much more on learning by examples than in predefined cuts specified in declarative languages [Christianson et al. 1996; Amerson et al. 2005] The information kept at the input hub will be the sole input of the editor agent, which is thus independent of the specific 3D environment. Figure 2 illustrates the role of the scene mapper: feed an input hub (float array container) with normalized information gathered from the 3D scene.

The editor agent contains a set of neuronal network classifiers, one for each available shot (cinematographer). This agent uses the output of each neuronal network to classify the candidate cinematographers at runtime. The key feature of the system is the use of an interactive training mechanism that enables the user to teach his shot preferences at runtime while watching gameplay. The basic architecture of the editor agent is illustrated in Figure 3, while a detailed explanation is given in the next section.

In our architecture, cinematographer agents are components implemented almost independently of the other ones. Their responsibility is to represent film idioms, such as a chasing camera or an over the shoulder shot, among others. The available cinematographer imple-

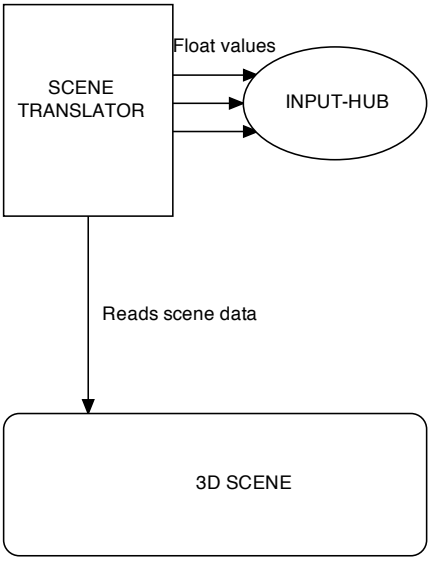


Figure 2: *Scene Mapper*

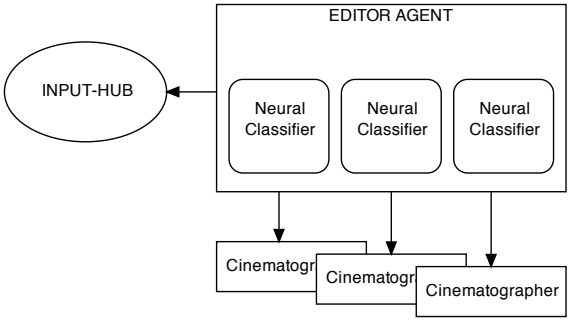


Figure 3: *Editor Agent*

mentations are enabled or disabled by the editor agent at runtime, based on the values given by the respective classifiers. In this current work, we are more interested in the cut/transition editing issue, so one is encouraged to read relevant previous research on the topic of film idiom implementations such as those pointed out in the related work section.

The communication between the three layers, shown in Figure 4, is made as follows:

1. The scene mapper, with the knowledge of the 3D environment, gathers and normalizes information about the scene to an array of float values within the range of [-1,1]. It constantly feeds these values into an input-hub, that is the only data-structure needed by the editor agent's neuronal network classifiers;

2. The editor keeps references to cinematographer agents and decides which one to use at each time, based on the output of the respective classifiers;

3. In our prototype, the cinematographers operate directly on the 3D environment, each one controlling its own virtual camera (viewport);

4. The position and orientation of the camera controlled by the chosen cinematographer are replicated to the editor viewport of the system.

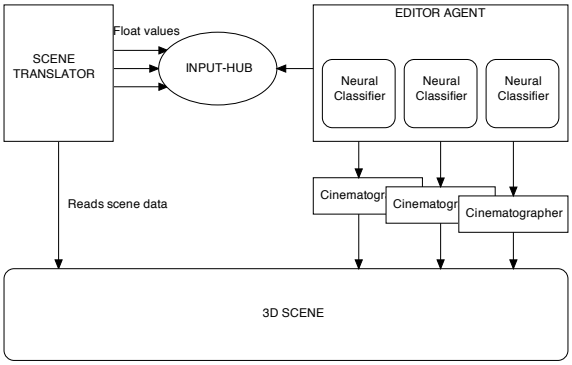


Figure 4: *Communication Architecture*

4 The Editor Agent

The role of the editor agent is to chose which shot, provided by the cinematographer agent, to use at each time for a given dynamic scene. It feeds all the values kept at the input-hub into several neuronal network classifiers simultaneously, one for each cinematographer. The output of each classifier is used as an activation value for the respective cinematographer. The goal of the training mechanism is to teach each classifier to recognize the situations that are favorable to its cinematographer. This training is performed by a user who only has to indicate his favorite camera at each relevant situation while interactively watching a player testing the game. In this section we explain in more detail the features of our proposed editor agent: the neuronal network classifiers; the learning mechanism; and the user interface.

4.1 Neuronal Network Classifiers

Neuronal network building blocks are called neurons, which are computation units that take as input a collection of values in a normalized range - usually from -1 to 1) and compute an output. Each input value has an associated weight that, after training, indicates its importance to that neuron. The output is given by two functions: input and activation. The input function often calculates the weighted sum of the input values, while the activation function is commonly the step, sigmoidal functions, among others. Figure 5 presents a schematic representation of a neuron. In our system, we chose to use the sigmoidal function because the output of the network has to be a float activation value instead of the binary output of the step function. The formulae for computing the output of each neuron is given bellow:

$$output = \frac{1}{1+\epsilon^{-\sum Wj*Ij}}$$

Where,

- j is an index for the neuron input and weight vectors, ranging between $[0, size - 1]$;
- W is the weight vector;
- I is the input vector.

Neuronal networks can be composed by sets with only one to several neurons, which are thus arranged in layers. Single layer networks are also called perceptrons. In our system, it's possible to use either perceptrons or multilayered networks as classifiers for the cinematographers. The choice will depend on the complexity of the scene input and the number of available cinematographers. Figures 6 and 7 illustrate how the perceptron and multi-layered neuronal networks are built in our system, showing the connection between input neurons and the input-hub. One can notice that all classifiers share the same input-hub, which is just a simple data-structure used for distributing the input vector, not a weighted neuron.

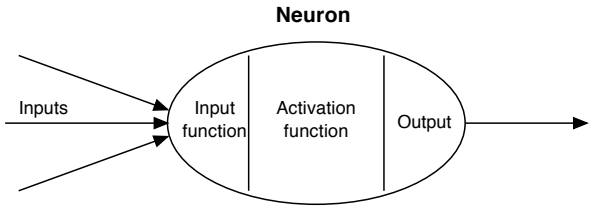


Figure 5: *Schematics of a Neuron*

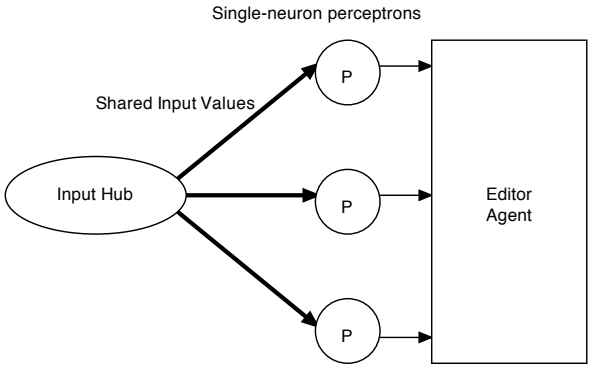


Figure 6: *Editor with Perceptron Classifiers*

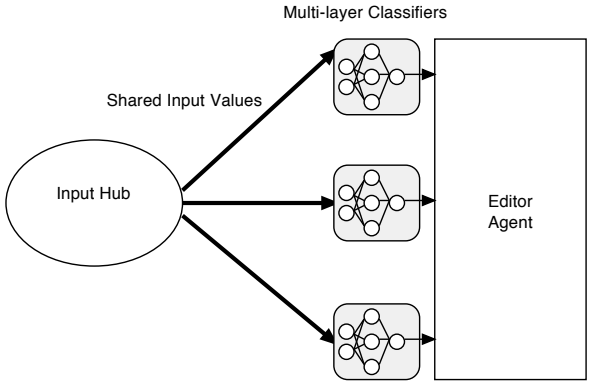


Figure 7: *Editor with Multi-layer Classifiers*

4.2 The Learning Mechanism

By modifying the weights associated to the inputs, one can teach each neuron the importance of each input value for its respective output. A set of know tuples of inputs together with the associated desired output is called a training set. To train a neuron network, one should feed its input with each example from the training set, comparing the calculated output with the desired one and feeding back the computed error to a training function, which adjusts the weights properly. Each time the training set is fed through the network is called a *training age*. For single neuron networks, the simple training rule given in the following formulae is applied to each classifier while for multi-layered networks the back propagation algorithm is used.

$$Wj = Wj + \alpha \times (desired - current) \times Ij$$

Where,

- α is the learning rate for the training mechanism;
- $desired$ is the correct output for the training sample;
- $current$ is the calculated output using current weights;

One can notice that the training function updates the weights of the neuron proportionally to the input (Ij), the output error ($desired - current$) and the training rate (α). It is important to balance between large training rates and the number of training ages. In section 6 we present experiments and analysis about these variables with a game prototype implemented exclusively for this work.

In our system, the training set is initially empty, and is populated as the user chooses favorite shots while watching gameplay. Each time he informs the system of a preferred shot, a snapshot of the current input is taken and the desired value for each cinematographer is set: 1 for the chosen one and 0 for all the others. Each sample consisting of the input values and the desired output is then added to the respective cinematographer agent's training set. At each time a new sample is included, the weights are reset and a new training is performed for all the classifiers. To keep the training cost low throughout the simulation, we use the minimum necessary number of training ages, which will be better explained in Section 6.

4.3 The Learning Interface

The proposed system interface used for training the editor agent in our work, shown in Figure 8, is composed of several viewports to show at the same time:

- All the available shots (cinematographers) - seen in the small viewports on the right;
- The current choice of the editor agent - lower-right viewport;
- The main screen for the player during training - bigger viewport.

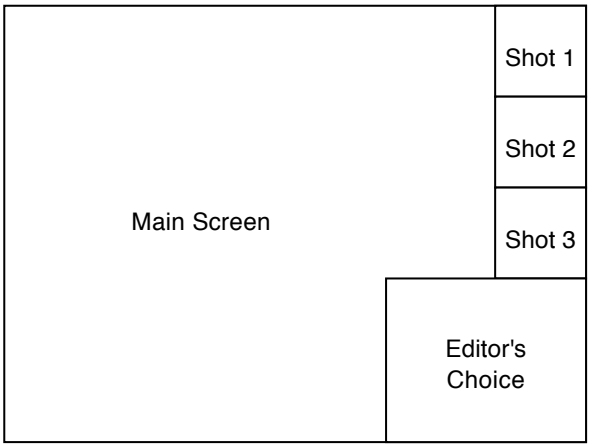


Figure 8: *Learning Interface with 3 Cinematographers*

When the system starts, the training set is empty and will be filled by the user (or a spectator beside him) as he plays. At each time the game developer finds the simulation state in a representative situation for a specific shot, he should press the numeric key of that cinematographer. This will fire the mechanism that feeds each cinematographer's training set by taking the input snapshot, adding the respective desired output. After the new samples are included in each set, a training age is performed for all cinematographers. In the next sections we show the prototype, exclusively developed to test the system, and the results of experiments performed to verify the accuracy of the learning mechanism.

5 Prototype: Race Game

We developed a game prototype to illustrate the use of our editor agent. It consists of a simple race game where the goal is to make the editor agent learn to use different camera shots to increase the dramatic appeal of action situations such as jumps and rocks crossing the road. The game takes place on a small modeled island with

a race track around it. There are several bumps through the track that make the car jump. In addition, there is a volcano that throws rocks over the track in the back part of the circuit. The player controls the car as usual, using the directional keys, while the editor agent training controls are the number keys corresponding to the available shots. Figure 9 shows a high view of the island, showing the race track and the volcano.

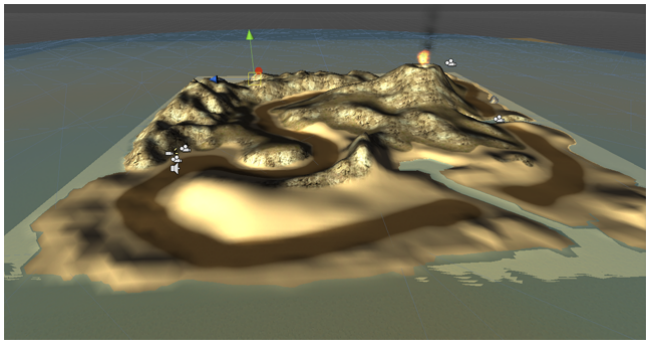


Figure 9: *Screenshot of the Island and the Race Track*

5.0.1 Editor Agent

The editor agent for the prototype includes three shot options and the respective classifiers, which are single neuron perceptrons. This example does not need multi-layer networks because the functions we are trying to teach for each classifier are linear. The goal is to use a chasing camera for normal racing situations, and different ones for jumps and for when the car is near the falling rocks. One can notice that it would be easy to implement a rule based system to achieve the same goals, but our intelligent editor is flexible to learn any other set of rules with the same simple teaching interface, and also, we target it for non technical users, who would find it difficult to specify logical rules.

The shot library comprises of three cinematographer agents, referenced by the editor and implemented by directly manipulating the virtual camera. Figure 10 shows a screenshot of the learning interface of the prototype game, while the description of the three available cinematographers is given bellow.

- *Chasing Camera* - follows the car from behind - default;
- *Front Camera* - keeps ahead of the car, pointing back to it - for jumping situations;
- *High-view Camera* - fixed at the volcano top, and its orientation follows the car - for the volcano area.

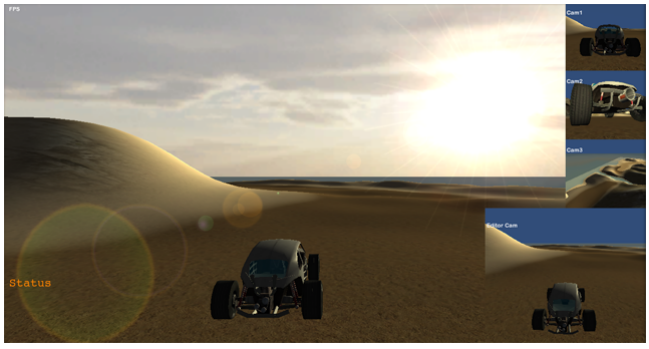


Figure 10: *Prototype game interface, showing the available shots*

The input for the editor, which is illustrated in Figure 11, comprises of only three values:

- Boolean value (0 or 1) saying if the car is touching the ground;
- Normalized values for the X and Z coordinates of the car, based on the limits and center of the island.

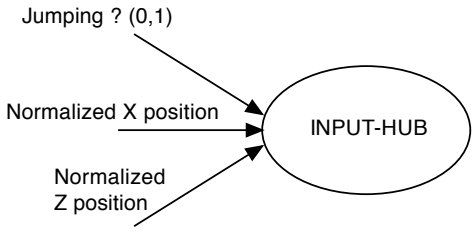


Figure 11: *Input-hub for the prototype*

6 Experiment Results

We were specially interested in experimenting with the learning mechanism to verify its performance with only a minimal number of training ages and a very small training set, so the user would not notice the computational penalty of this task. By classification performance we mean both classification quality, the correctness rate that the classifier generates against the expected output; and accuracy, characterized by small dispersion values over the test instances. We expected to confirm that the size of the training set could be small, so the user would not have to choose the adequate camera for each situation too many times. The less the user has to inform/teach the system, more simple it becomes to operate. But before actually performing interactive tests with real users and small training sets we did some experiments to find what was the ideal number of training ages and the influence of the learning rates of the neurons over the classification performance. This section describes both parts of the experiment: first, we analyze the influence of learning rates and number of ages over performance; second, we show the results of tests with real users.

6.1 Learning Rate and Ages

For both parts of the experiment we used an auxiliary sample set for validation purposes which consisted of 300 collected samples, with all three expected values for each cinematographer classifier. This training set was carefully built with actual application data to represent all possible situations during gameplay. The first experiment aimed to check the influence of the learning rate over accuracy. Using this validation sample set, we performed *Ten Fold Cross-validation* tests for each learning rate value, and measured the average output correctness and standard deviation for each instance of the test. *Ten Fold Cross-validation* works by shuffling the sample set and splitting it in ten parts. Then, each one of these 10% subsets are used to validate the system, which is trained with the remaining 90%.

We performed this test with the learning rates ranging from 0.05 to 0.5 and noticed that the learning rate does not influence much on the performance, given that the system converges to a trained state. The only influence is noticed in the training speed, meaning that small values take too long to converge, while higher values are too sensible to errors in the training set. Given this, we chose to run the remaining experiments with a learning rate of 10%.

With the second experiment, we wanted to find appropriate maximum value for the number of ages to be performed at each training of the networks, big enough to achieve good accuracy even if the system does not converge and still small enough to avoid high computational intensity. We ran the same kind of *Ten Fold Cross-validation* tests, this time fixing the learning rate in 10% and varying the maximum number of ages from 1 to 50. The results for this tests are presented in a chart in Figure 12. The horizontal axis represents the number of ages, while the vertical axis shows the measured classification performance (100% is full correctness).

One can notice from the chart that the number of ages has a strong influence over the trained network performance. Only with 30 or more ages has the accuracy converged to 100%, with a measured standard deviation of around 3%. Fortunately, even when executing 50 training ages with the small training sets expected to be used in

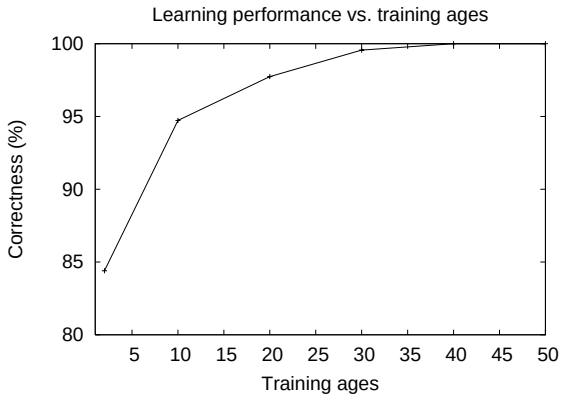


Figure 12: *Classifier performance with varying ages number*

actual interactive training, the computational penalty is almost imperceptible, so this was the number used in the final experiments. We also checked how robust the prototype classifiers were, by introducing random error on the validation sample set in the range of 5% to 25%. Figure 13 shows the influence of this errors over the different number of training ages. The axis represent the same variables as the previous test results, and the different lines represent the different error ranges.

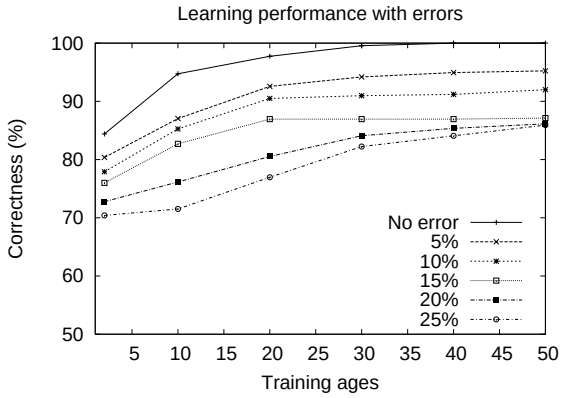


Figure 13: *Classifier performance in the presence of errors in the training set*

One can notice that the presence of error in the training set is particularly influent with the smaller number of ages. By training with 50 ages, even with in the presence of 25% incorrect samples, the accuracy is still high, reaching more than 85%. This robustness is one of the reasons neuronal networks are a good choice for such system. Following, we analyze the results of interactive training experiments with real users creating very small training sets.

6.2 Interactive Experiments

For the interactive experiments, we wanted to check if the learning mechanism was robust enough to perform well even with a very small training set, which is mandatory for the system to be considered user friendly from an usability point of view. The tests consisted of a user watching live gameplay and choosing the correct shot (cinematographer) during the simulation. We only instructed the users to choose the first shot (chase camera) for normal situations, the second one (front camera) for jumping moments, and the last one (volcano shot) when the car was near that part of the island. Since this prototype editor agent is composed of three different cinematographers, we used a minimal training set of at least three samples, one for each available shot. From this minimum value, the test users increased the number of samples by 3 for each new test category, always adding one more sample to favor each one of the cinematographers. To better validate the results through standard deviation, the users were asked to "teach" the system several times by starting from scratch for each desired training set size. Table 1 shows the computed results for this experiment with the training set size ranging from 3 to 12 samples, while Figure 14 shows the performance of the editor agent in a line chart.

Table 1: *Performance and deviation with small training sets*

Samples	3	6	9	12
Accuracy	83,134	95,945	98,894	99,447
Std. Dev.	17,020	3,938	0,232	0,463

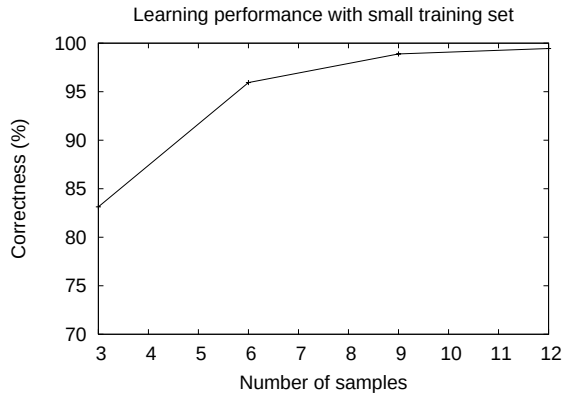


Figure 14: *Classifier performance with small training sets*

One can see that, apart from the trivial case of only one sample for each shot, the accuracy results are statistically very satisfactory. It is also important to say that again, apart from this trivial training set with only 3 samples, the system has always reached 100% correctness over the validation set in several test instances. The average accuracy was a little under this value probably because of some mistakes by the user, who failed to choose good samples, which is expected to happen even in production situations. The test results were very important because they showed us strong evidences that this mechanism is good enough for a real case production scenario, where such intelligent system becomes very handful, specially because non technical users are not expected to learn how to specify complex logical rules such as an equivalent decision tree.

7 Conclusion

In this paper we presented an editor agent for intelligent game cinematography that features a learn by example mechanism with very good accuracy, does not impose big performance penalties and is suitable to be included in a production pipeline, given its very easy user interface. This editor agent is part of a cinematography architecture that has been designed and developed for games and other interactive applications, even those where there is no prior knowledge of the action events that will take place in the 3D environment. Although designed to work in games, we think our system is suitable for other types of interactive 3D visualization applications as well and we plan to experiment it with those scenarios. In our opinion, the major contributions of this work are the robust learn by example mechanism, the independency of complex domain translation of scene data and the intuitive user interface.

The experiments made with the neuronal networks classifiers provided very important results that have lead to some modifications on the implemented ideas. For instance, we originally thought it

would be possible to run smaller number of training ages, but the results showed us that this number is much more important than the learning rate and even the training set size. The final tests with real users showed us very important evidence that the system is viable to a production scenario, given the near-ideal performance that was obtained with very small training sets.

Future work on this cinematography architecture include:

- Experimenting with more complex environments, input sets and shot types, which will need multi-layer neural network classifiers;
- Design a learning mechanism to interactively specify cinematographer agent's constraints;
- Integrate the system with a distributed architecture targeting spectators of massive online games.

References

AMERSON, D., KIME, S., AND YOUNG, R. M. 2005. Real-time cinematic camera control for interactive narratives. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, ACM, New York, NY, USA, 369–369.

CHRISTIANSON, D. B., ANDERSON, S. E., WEI HE, L., SALESIN, D., WELD, D. S., AND COHEN, M. F. 1996. Declarative camera control for automatic cinematography. In *AAAI/IAAI, Vol. 1*, 148–155.

COURTY, N., LAMARCHE, F., DONIKIAN, S., AND RIC MARCH. 2003. A cinematography system for virtual storytelling. In *in Int. Conf. on Virtual Storytelling, ICVS'03*, Springer, 30–34.

DRUCKER, S. M., AND ZELTZER, D. 1995. Camdroid: a system for implementing intelligent camera control. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 139–144.

DRUCKER, S., HE, L., COHEN, M., WONG, C., AND GUPTA, A., 2002. Spectator games: A new entertainment modality for networked multiplayer games. `http://research.microsoft.com/~sdrucker/papers/spectator.pdf`.

ELSON, D. K., AND RIEDL, M. O. 2007. A lightweight intelligent virtual cinematography system for machinima production. In *AIIDE*, The AAAI Press, J. Schaeffer and M. Mateas, Eds., 8–13.

GUERRA, F. W., 2008. Engenharia de estorias: um estudo sobre a geracao e narrao automatica de estorias.

HALPER, N., AND MASUCH, M. 2003. Action summary for computer games: Extracting action for spectator modes and summaries. In *Proc. of 2nd Int?l Conf. Application and Development of Computer Games*, 124–132.

HAWKINS, B. 2004. *Real-Time Cinematography for Games (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA.

HERMANN, R., AND CELES, W. 2005. Posicionamento automatico de cameras em ambientes virtuais dinamicos. In *Proceedings of IV workshop on games and digital entertainment of the Brasilian Simposium on Computer Games and Digital Entertainment*.

HORNUNG, E., LAKEMEYER, G., AND TROGEMANN, G. 2003. Autonomous real-time camera agents in interactive narratives and games. In *Proceedings of the IVA 2003: 4th International Working Conference on Intelligent Virtual Agents, 15.-17.9.2003, Irsee, Germany, Lecture Notes in Computer Science 2792*, Springer, 236–243.

KNEAFSEY, J., AND MCCABE, H. 2005. Camerabots: Cinematography for games with non-player characters as camera operators. In *DIGRA Conf.*

MORRIS, D., KELLAND, M., AND LLOYD, D. 2005. *Machinima: Making Animated Movies in 3D Virtual Environments*. Muska & Lipman/Premier-Trade.

PINHANEZ, C. S. 1999. *Representation and recognition of action in interactive spaces*. PhD thesis, Massachusetts Institute of Technology. Adviser-Aaron F. Bobick.

POZZER, C. T. 2005. *Um sistema para geracao, interacao e visualizacao 3D de historias para TV interativa*. PhD thesis, Pontificia Universidade Catolica - RJ.

TOMLINSON, B. 2000. Expressive autonomous cinematography for interactive virtual environments. In *In Proceedings of the Fourth International Conference on Autonomous Agents*, ACM Press, 317–324.

WEI HE, L., COHEN, M. F., AND SALESIN, D. H. 1996. The virtual cinematographer: a paradigm for automatic real-time camera control and directing. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 217–224.

IRTaktiks: Jogo de Estratégia para Mesa Multitoque

Willians S. Schneider Nilson C. Dias F.º Luis H. M. Mauruto Fábio R. Miranda

Centro Universitário Senac, Bacharelado em Ciência da Computação
São Paulo – SP, Brasil

Figura 1: *IRTaktiks* jogado simultaneamente por dois jogadores, que manipulam suas unidades diretamente com as mãos

Abstract

We present *IRTaktiks*, a real-time two-player strategy game playable on a custom multi-touch table.
Details regarding the construction of the interface hardware, the detection and recognition of events and the architecture of the software that was designed and implemented are presented. Aspects of how the game works and results of the finished project are also presented and discussed.

Keywords: Architectures, Engines, and Design Patterns; Computer Graphics, Human-computer Interfaces, Interface hardware, Programming Techniques.

Authors' contact:
willians.schneider@gmail.com,{lhmenezes, nilmesmo}@gmail.com,
fabio.rmiranda@sp.senac.br

Sample Video:
www.tinyurl.com/irtaktiks/

Source code:
http://www.codeplex.com/irtaktiks

1. Introdução

Em tempos recentes tem havido interesse crescente em dispositivos de interação multitoque, caracterizados por um ou mais usuários poderem interagir através dos dedos diretamente com uma interface gráfica, dispensando o uso de dispositivos de entrada mais convencionais, como teclados, mouses ou canetas especiais. Inicialmente protótipos de telas e mesas de interação multitoque desenvolvidos por pesquisadores de interação e entusiastas foram demonstrados, e agora este conceito já se encontra embutido em produtos computacionais, o telefone celular *iPhone* ou no futuro *Windows* 7.

Uma materialização bastante comum das interfaces multitoque é na forma de mesas, que são interessantes por ser um objeto do domínio cotidiano das pessoas e naturalmente um meio para colaboração, quer na forma de espaço de trabalho, quer na forma de jogos. As interfaces multitoque podem trazer para o ambiente computacional uma forma de colaboração e interação simultânea que normalmente é inviabilizada nas estações de trabalho convencionais por questões ergonômicas (estão disponíveis apenas um mouse e teclado), mas que muitas vezes está presente em jogos de mesa e tabuleiro.

Tanto na forma de mesas de interação quanto na de outros tipos de dispositivos, é razoável supor que o potencial de interação intuitiva dos dispositivos de interação multitoque aliado ao suporte crescente ao desenvolvimento de aplicações deste tipo em sistemas operacionais e em bibliotecas independentes constitui uma oportunidade interessante para o desenvolvimento de jogos inovadores.

Este trabalho trata do projeto e implementação do *IRTaktiks*, um jogo de estratégia jogável simultaneamente por dois jogadores (exemplificado na Figura 1) numa mesa de interação multitoque desenvolvida com materiais de baixo custo (que pode ser vista na Figura 5). Serão apresentados aspectos relacionados à construção do dispositivo de interação, detecção dos múltiplos toques a partir do uso de bibliotecas de código aberto, transformação dos toques em eventos de jogo e o significado destes eventos no domínio do jogo. Este trabalho também detalha elementos de projeto do jogo.

A organização deste artigo é descrita a seguir. Na próxima seção alguns trabalhos relacionados que tratam de interação multitoque são apresentados. Na seção 3 a reflexão total interna frustrada da luz (*FTIR*), que é o princípio do dispositivo de entrada usado neste trabalho, é apresentada. A infra-estrutura de software usada no projeto é discutida na seção 4 enquanto na seção 5 apresentam-se detalhes do desenvolvimento do projeto, cujos resultados são apresentados na seção 6. A seção 7 trata das conclusões e trabalhos futuros.

2. Trabalhos relacionados

A interação multitoque popularizou-se em 2006 com a ajuda de vídeos na Internet em que *Jefferson Y. Han*, pesquisador do instituto de ciências matemáticas *Courant* demonstrou seu trabalho de pesquisa de interação multitoque utilizando uma superfície com display gráfico interativa que permitia a interação simultânea de múltiplos usuários. Foram apresentadas implementações de diversas aplicações, entre as quais jogos multitoque simples. Os protótipos de *J. Han* [Han 2005] despertaram o interesse de diversas iniciativas de pesquisa sobre essa nova alternativa de interação, e logos estavam disponíveis na WWW diversos tutoriais e weblogs [Buxton 2008] que trocam experiências entre pesquisadores e fomentam o desenvolvimento de protótipos.

O multitoque teve seu início em 1982 [Multigesture.net 2008], com tablets feitos na universidade de *Toronto* e com telas dos laboratórios *Bell*. Nos anos 90 a universidade de *Delaware* desenvolveu um sofisticado sistema de reconhecimento de gestos e escrita, base para o mouse-pad *iGesture* e teclados *TouchStream*, comercializados pela *FingerWorks* em 2001. Estes teclados eram reconhecidos pela sua ergonomia, os usuários apenas precisavam apontar e arrastar com um ou mais dedos, eliminando totalmente a necessidade de um dispositivo apontador, como o mouse.

O primeiro dispositivo multitoque com display integrado a ser comercializado foi o *Lemur Input Device*, um controlador multimídia profissional da companhia francesa *JazzMutant* lançado em 2005 [Multigesture.net 2008]. Em julho de 2007, a *Apple Inc*. lançou o produto *iPhone* que tinha interação multitoque e a empresa Microsoft demonstrou logo a seguir uma mesa de interação chamada *Microsoft Surface*.

A mesa *ReacTable* [Kaltenbrunner et al. 2006] funciona como um instrumento musical colaborativo, que permite o reconhecimento de objetos postos sobre sua superfície e tem a possibilidade de permitir interação multiusuário. Na *ReacTable* o usuário pode sintetizar sons através de uma cadeia de fontes, filtros e osciladores manipuláveis, todos gerados por software. Cada objeto colocado sobre sua superfície é classificado por um software a partir de marcadores fiduciais situados em sua superfície e capturados por uma câmera situada abaixo da mesa. Cada objeto é classificado como um dos geradores e filtros de uma aplicação musical obtendo-se como resultado um som único, resultado da interação destes objetos. Este instrumento utiliza como base o software de detecção de fiduciais *ReacTIVision*, que reconhece os objetos sobre a mesa. A *ReacTable* ganhou notoriedade recentemente ao ser utilizado em shows e apresentações pela cantora *Björk*, no *Coachella Festival*, em 2007 na *California* [Wired 2008].

3. FTIR e Multitoque

Atualmente, existem diversas técnicas para a detecção de múltiplos toques em superfícies, por exemplo, análise da imagem de câmeras que enquadram o dispositivo, utilização de sensores medidores de pressão, utilização de um grids de filamentos eletrônicos, onde o toque simplesmente fecha contato permitindo a passagem de corrente elétrica, até a utilização de circuitos eletrônicos que percebem o contato dos dedos humanos por uma alteração de capacitância na região do toque.

As técnicas mais simples e baratas costumam utilizar iluminação infravermelha e uma superfície de acrílico. Em geral um anteparo de projeção é colocado junto ao acrílico, e a superfície de interação é iluminada lateralmente com luz infravermelha. A mesa multitoque utilizada neste trabalho (que pode ser vista na Figura 5) baseia-se em iluminação infravermelha, mais especificamente a *FTIR* (reflexão total interna frustrada da luz).

A reflexão da luz é o fenômeno físico em que um feixe de luz incide sobre uma determinada superfície e é refletida para o mesmo meio de propagação de origem. Quando a reflexão é total, todos os fótons do feixe de luz são redirecionados ao meio de propagação de origem, ao contrário da reflexão parcial, em que alguns fótons atravessam a interface entre os meios de propagação, caso em que ocorre um desvio no ângulo de incidência do feixe de luz emitido, chamado de refração. O pesquisador *J. Y. Han*, durante suas pesquisas sobre técnicas de interação multitoque, percebeu que a pele é um material difusor, ou seja, quando um feixe de luz que seria refletido totalmente entra em contato com a pele, ele é difundido em todas as direções. A esse efeito de difundir a luz que seria totalmente refletida, se deu o nome de reflexão total interna frustrada da luz, que encontra-se ilustrado na Figura 2.

Este fenômeno é passível de aplicação em interfaces multitoque, em que pode-se em iluminar as laterais de uma superfície de acrílico com diversos *LEDs* infravermelhos de modo que a luz emitida fique presa dentro do acrílico devido ao fenômeno da reflexão total da luz. Quando o dedo do usuário toca a superfície da mesa, a luz é difundida para baixo, onde

uma *webcam* obtém imagens. Essa difusão é reconhecida como pontos de alta luminosidade na imagem capturada e a posição dos toques é facilmente detectada. Pode-se ver este fenômeno em efeito na imagem mostrada na Figura 6.

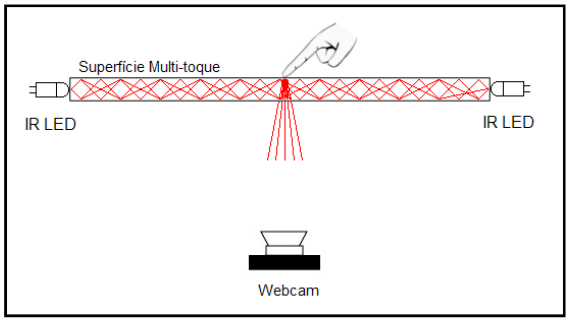


Figura 2: A reflexão total interna da luz na superfície de acrílico é frustrada no ponto de contato com o dedo do usuário

4. Infra-Estrutura e Ferramentas

Alguns softwares publicamente disponíveis foram utilizados para o reconhecimento de toques sobre a superfície da mesa multitoque. Existe uma grande quantidade de superfícies multitoque sendo desenvolvidas por uma comunidade de entusiastas que trocam informações pela Internet, e um padrão para o armazenamento das informações relacionadas aos toques e sua integração com outras aplicações foi sendo adotado pelos desenvolvedores de software.

Durante o projeto da *ReacTable*, desenvolveu-se um protocolo capaz de armazenar informações sobre toques e objetos em qualquer superfície multitoque. A esse protocolo deu-se o nome de *TUIO*.

Outro componente importante do sistema da *ReacTable* é o *reacTIVision*, [Kaltenbrunner et al. 2007] software responsável pela identificação de toques e fiduciais, que analisa a imagem da superfície e emite mensagens do tipo *TUIO* [Kaltenbrunner et al. 2006]. Para comunicar os pacotes *TUIO* com informação sobre os toques com aplicações voltadas a gerar sons e efeitos sonoros, na *ReacTable* tais pacotes são encapsulados num outro protocolo que é compatível com uma grande variedade de bibliotecas de código aberto, chamado *OSC* (*Open Sound Control*)

Após o desenvolvimento do *reacTIVision*, diversos softwares com o propósito de identificação de toques foram desenvolvidos. A grande maioria buscou seguir o mesmo padrão, ou seja, mensagens *TUIO* sob o protocolo *OSC*, tornando-os padrão no desenvolvimento de aplicações multitoque. Um exemplo de projeto que adotou mensagens *TUIO/OSC* é a biblioteca *Touchlib*, adotada no *IRTaktiks* para o reconhecimento dos toques na mesa desenvolvida devido à sua estabilidade e funcionalidades úteis ao projeto. A adoção do *Touchlib* é um fator interessante, pois permite que o projeto *IRTaktiks* funcione corretamente em qualquer superfície multitoque que siga os padrões propostos.

A seguir serão fornecidos mais alguns detalhes a respeito de *TUIO*, *OSC* e *Touchlib*.

4.1 Open Sound Control (OSC) e TUIO

O *Open Sound Control* (*OSC*) [Cnmat 2008] é um protocolo desenvolvido para a comunicação entre computadores, sintetizadores de som e outros dispositivos multimídia. É utilizado em produtos de Realidade Virtual, interfaces Web e também como meio de transporte para outros protocolos que não possuem facilidade de comunicação.

A biblioteca *OSCpack* [Bencina 2008] é um conjunto de classes em *C++* responsáveis por criar e ler pacotes do protocolo *OSC*, incluindo as funcionalidades mínimas para a comunicação utilizando *UDP* nas plataformas *Windows* e *POSIX*. Atualmente é utilizada em diversos projetos, como o *ReacTIVision*, *Touchlib* e *AudioMulch* por causa de sua capacidade de prover comunicação simplificada entre as plataformas *Windows*, *Linux* e *Mac OS X*.

TUIO é um protocolo de comunicação desenvolvido com a finalidade de atender os requisitos de comunicação entre interfaces tangíveis. Este protocolo define propriedades comuns baseadas no controle de objetos, toques e gestos. Há implementações do *TUIO* nas linguagens *Java*, *C*, *C++* e *Adobe Flash*, entre outras.

As mensagens do *TUIO* dividem-se em perfis baseados na interação com a interface tangível. Atualmente, o *TUIO* possui perfis para interfaces 2D, 3D e interfaces customizadas. Cada perfil, por sua vez, possui dois tipos de mensagens diferentes, usadas na representação da interação de objetos e toques com o dispositivo. A mensagem carrega diversas informações sobre a interação, dentre as quais destacam-se a sessão, um identificador da interação, posições no espaço 2D ou 3D, ângulo, vetor de movimento, vetor de rotação, aceleração de movimento e aceleração de rotação. [Tuio 2008]

4.2 Touchlib

O *Touchlib* é uma biblioteca que permite a detecção de toques em superfícies multitoque que utilizam o princípio da reflexão total interna frustrada da luz, quer se utilizem de iluminação traseira ou iluminação frontal. Esta biblioteca foi desenvolvida pela *Natural User Interface Group* em parceria com a *White Noise Audio* [Nuigroup 2008].

Através de algoritmos de divisão e comparação, detecta realces no histograma das imagens enviadas

por uma *webcam* transformando-os em informações sobre cursores, e disparando eventos que podem ser tratados em aplicações *C/C++*. Estes eventos são disparados nos momentos em que um dedo toca, percorre ou é retirado da superfície multitoque. Esta biblioteca permite a integração com demais aplicativos através de *TUIO/OSCpack*.

O *Touchlib* aplica técnicas de processamento de imagem no resultado capturado das *webcams* a fim de melhorar a percepção de toques. Atualmente esta biblioteca trabalha apenas na plataforma *Windows*, porém existem esforços sendo realizados para portá-la para outras plataformas, como *Mac OS X* e *Linux*.

Os eventos detectados pelo *Touchlib* são convertidos para uma escala que vai de zero até um. O canto superior esquerdo da imagem é o ponto (0, 0), enquanto o inferior direito é o (1, 1). O *Touchlib* limiariza e segmenta a imagem obtida pela *webcam* e a divide em 20 imagens menores. Quando um toque é detectado, sua posição é calculada através de uma interpolação linear apenas na respectiva fatia da imagem. Isso permite que a câmera não necessite estar perpendicular à superfície de projeção, pois a distorção provocada é compensada pela interpolação. A Figura 3 exemplifica este conceito: na sua parte inferior tem-se o detalhe de uma célula da grade em que a superfície de toque é dividida.

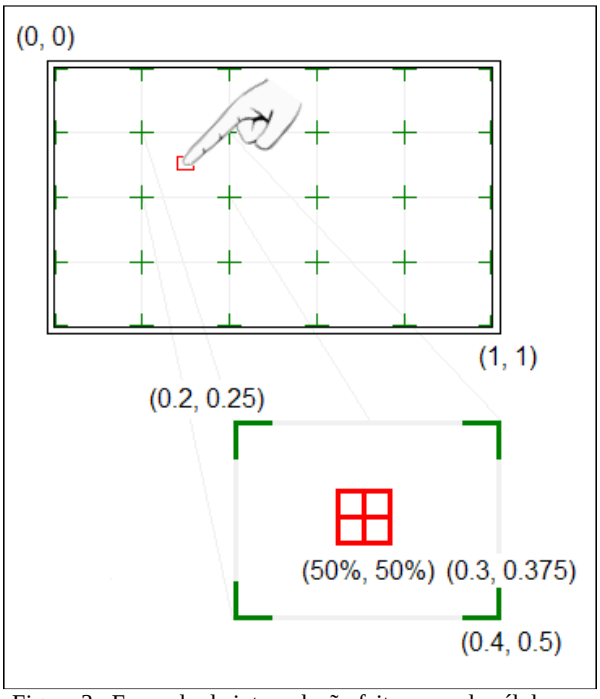


Figura 3: Exemplo de interpolação feita em cada célula no cálculo da posição do toque

5. O projeto IRTaktiks

O jogo *IRTaktiks* é um *RPG* tático, em que o jogador controla diversos personagens com características diferentes, cujo objetivo é derrotar o inimigo através de ataques, magias e itens, utilizando táticas, como por exemplo se beneficiar de uma determinada posição no campo de batalha para obter vantagens sobre o inimigo.

O processamento de eventos de entrada no jogo está exemplificado na Figura 4. As interações do usuário sobre a mesa, por exemplo o toque de um ou mais dedos sobre sua superfície, são reconhecidos pelo *Touchlib* através da análise das imagens enviadas por uma *webcam* posicionada sob a mesa e que enquadra a superfície de interação. O *Touchlib* processa as informações e envia uma mensagem *TUIO* para cada dedo sobre a mesa, contendo as informações como posição, ângulo de movimentação, velocidades calculadas entre outras.

O jogo utiliza a biblioteca *OSCpack* e implementa um cliente *OSC* que recebe as mensagens *TUIO/OSC* geradas pela *Touchlib* e as decodifica para obter informações a respeito dos toques efetuados pelo usuário na mesa. Há um módulo do *IRTaktiks* chamado *Input* que se encarrega de disparar um evento interno ao jogo para cada interação executada pelo usuário na mesa. Estes eventos se propagam para os diversos componentes do jogo que se atualizam conforme necessário. Finalmente, uma imagem do jogo é projetada com o auxílio de um projetor sob a superfície da mesa. O efeito percebido pelo jogador é o de manipular diretamente os objetos do jogo.

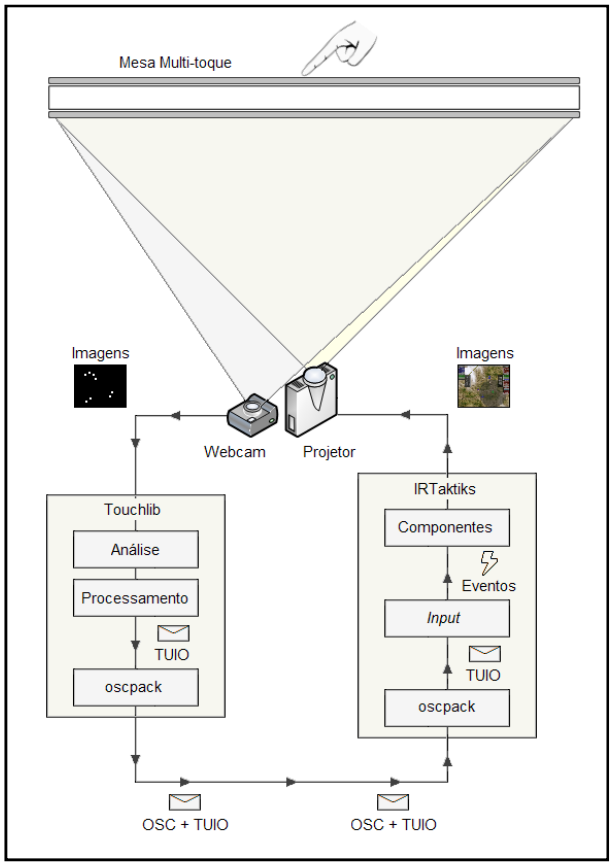


Figura 4: Arquitetura de deteção e processamento de eventos do sistema

5.1 Concepção

O *IRTaktiks* deve ser jogado por dois jogadores, e cada um terá várias unidades de combate. Cada unidade possui diversos atributos que quando configurados a tornam única e diferente das demais. Além de atributos, as unidades possuem classes que lhes dão características, vantagens, desvantagens e ações diferentes ampliando as possibilidades de estratégia de cada um dos usuários. O objetivo é derrotar todas as unidades do jogador adversário, utilizando as características de cada unidade de combate e suas respectivas ações em conjunto com o cenário onde a batalha acontece.

5.2 Mesa multitoque

A mesa multitoque utilizada no *IRTaktiks* é formada por uma superfície de acrílico transparente de aproximadamente 1,2m x 1,6m, acoplada a um suporte de madeira sobre rodas, que facilita seu deslocamento. O acrílico é encaixado numa esquadria de alumínio que possui 47 entradas nas laterais que funcionam como soquetes para *LEDs* infravermelhos, de modo que a luz por eles emitida percorra o interior do acrílico capturada de acordo com o princípio da reflexão total interna da luz. Os *LEDs* infravermelhos utilizados são de alto brilho com corrente máxima suportada de 100mA e tensão de barreira de potencial de 1.2V, subdivididos em 10 trechos de circuito elétrico que estão dispostos ao longo do perímetro do acrílico.

Figura 5: Mesa multitoque utilizada no projeto

Para obter as imagens dos toques foi utilizada uma webcam *Microsoft LifeCam VX-6000*. A escolha desta webcam se deveu a possuir ângulo de visão com 71º, sensor *CCD* (*Charge Coupled Device*) com resolução de 800 por 600 *pixels* e ser capaz de suportar uma taxa de atualização de 30fps (quadros por segundo). A *webcam* veio de fábrica com um filtro que inibe a passagem da luz infravermelha posicionado entre a lente e o *CCD* que teve de ser removido.

A fim de minimizar as influências da luz do projetor e da iluminação ambiente da sala na detecção dos eventos nas imagens adquiridas pela *webcam*, foi adicionado à câmera um filtro que bloqueia a maior parte da luz visível mas permite a passagem de luz infravermelha. Utilizou-se como filtro bloqueador de luz visível um pedaço de filme fotográfico queimado após ter sido revelado, pode-se ver a diferença representada por este filtro na Figura 6.

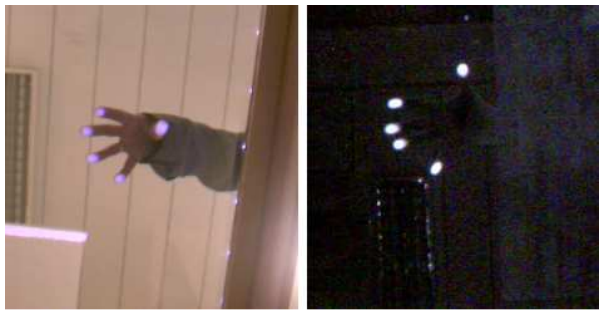


Figura 6: Toque sem e com o filtro inibidor da luz visível

A *webcam* fica posicionada sob a mesa com a superfície de acrílico contida em seu campo de visão, de modo a obter as imagens dos toques realizados pelos usuários. Devido à câmera utilizada ser dotada de um ângulo de visão maior que o de webcams convencionais, pôde ser colocada a uma distância menor em relação ao acrílico e mesmo assim cobrir uma área da mesa maior ou equivalente.

A projeção é feita com um projetor de resolução nativa de 800 por 600 pixels e um espelho simples, como pode ser visto na Figura 7. A imagem do projetor é direcionada pelo espelho para a superfície inferior do acrílico. É necessário posicionar um anteparo de material difusor sob a superfície do acrílico para que o usuário veja a projeção.

Figura 7: Sistema de projeção situado sob a mesa

O material ideal para este tipo de mesa é um polímero para projeções, fabricado pela empresa *Rosco*. Como este material não é encontrado facilmente no Brasil, sua utilização foi descartada. Para substituir o material difusor, foram realizados testes utilizando papel vegetal e sacolas plásticas brancas de polietileno.

Testes realizados indicaram que os sacos plásticos de polietileno permitiram maior nitidez na imagem

capturada pela *webcam* que é usada na detecção dos toques quando comparados com o papel vegetal. Este material não foi encontrado no tamanho necessário para cobrir uma área razoável da mesa sem que fossem necessárias emendas e foi também descartado. Utilizamos papel vegetal como anteparo na maior parte das execuções do protótipo e no vídeo disponibilizado online que mostra o trabalho em execução.

5.3 Arquitetura do software

A utilização do *Touchlib* para reconhecimento de toques e sua compatibilidade com *TUIO*/*OSC* não representa nenhuma restrição ou condição de contorno em relação a escolhas relacionadas à bibliotecas gráficas. A parte visual do jogo poderia ser desenvolvida em praticamente qualquer plataforma de computação gráfica ou *API* 3D, pois existem diversas implementações do protocolo *OSC*, em diversas plataformas e linguagens de programação. Dessa forma, a escolha foi baseada principalmente em qual ambiente a produtividade seria maior e qual proveria mais recursos, como controle de versões, gerenciadores de conteúdo e linguagens com suporte a programação orientada a objetos. Dentre frameworks existentes, optou-se pelo *Microsoft® XNA 2.0*, devido à experiência anterior dos integrantes do projeto com o ambiente de desenvolvimento da *Microsoft* com *C#*, grande variedade de recursos disponíveis, ampla documentação, desempenho do código gerado e possibilidade de utilizar o ambiente integrado de desenvolvimento voltado à produtividade, *Microsoft Visual Studio*.

Após a definição da plataforma em que o jogo seria desenvolvido, iniciou-se a construção de um módulo chamado *Listener*, responsável pela comunicação entre o jogo e o software que controla a detecção dos toques sobre a mesa (*Touchlib*). Dessa forma, futuros problemas de integração seriam eliminados, uma vez que a construção do jogo levaria este módulo de comunicação em consideração, sem alterá-lo. Foi decidido que este módulo utilizaria eventos internos para representar as interações dos usuários com a mesa. Desta forma o projeto do jogo foi simplificado e modularizado. O serviço que lê as informações contidas nas mensagens *OSC*/*TUIO* e dispara os eventos é executado em uma *thread* à parte; aumentando o desempenho do módulo de comunicação.

Este módulo foi construído utilizando as bibliotecas do *OSCpack*, que é responsável por obter as mensagens *TUIO* enviadas pela mesa, decodificá-las e transformá-las em entradas para o jogo através da comunicação com o módulo Input. Baseia-se em uma arquitetura cliente-servidor, exercendo a função de cliente.

As mensagens *TUIO* possuem informações sobre cada um dos toques e objetos que estão sobre a mesa. Cada cursor ou objeto possui um identificador, que é de base para o reconhecimento de ações mais complexas como, por exemplo, funcionalidades de arrastar e soltar.

Além de identificadores, cada cursor e objeto posicionado sobre a mesa possui três tipos de mensagens diferentes: *Down*, *Update* e *Up*.

As mensagens *Down* são enviadas quando o objeto ou o cursor são criados, ou seja, quando o objeto é colocado sobre a mesa ou quando o dedo do jogador encosta sobre sua superfície. As mensagens *Update* são enviadas para informar que o cursor ou o objeto estão ativos, ou seja, servem para informar que o objeto continua sobre a mesa, parado ou em movimento. Já as mensagens do tipo *Up* são enviadas quando o objeto ou o cursor não estão mais disponíveis e foram removidos da superfície da mesa. Com estes três tipos de mensagens é possível rastrear qualquer tipo de movimento sobre a mesa, seja ele usando objetos, toques, ou até mesmo uma combinação de ambos.

A arquitetura do jogo foi projetada de modo a deixar o *IRTaktiks* o mais leve e rápido possível, além de possibilitar a agregação de novas funcionalidades rapidamente e de maneira robusta. Utilizando reuso de módulos, processamento da placa de vídeo em conjunto com o do computador, *cache* de texturas e imagens e atualizações lógicas dos componentes somente quando necessário foi possível obter uma boa velocidade de execução sem comprometer a cobertura dos requisitos propostos ou a facilidade de manutenção do código fonte.

Dividiu-se a arquitetura em diversos módulos a fim de facilitar a implementação e extensão de funcionalidades, uma vez que com padrões definidos, a agregação de novas funcionalidades é bastante fácil e ágil. O diagrama da Figura 8 ilustra os módulos presentes no *IRTaktiks*.

O módulo *Listener* é responsável pela decodificação das mensagens *TUIO*/*OSC*, enviando as informações para o módulo *Input*, que dispara os eventos de adição, movimentação e remoção de dedos sobre a mesa. O módulo *Resource* efetua o carregamento dos recursos gráficos que serão desenhados pelo módulo *Drawable*, como texturas, imagens, partículas, efeitos e fontes. O módulo *Game* é a representação dos objetos do jogo, por exemplo, os jogadores e suas unidades. Cada unidade possui características que são descritas pelo módulo *Logic*, ações que são implementadas no módulo *Action* e menus que são construídos pelo módulo *Menu*. A interação entre as ações e os menus é realizada pelo módulo Interaction. O módulo *Screen* implementa as várias telas do jogo e suas transições, enquanto o módulo *Debug* é utilizando para auxiliar o desenvolvimento e testes de novas funcionalidades.

A arquitetura interna no *XNA* é centralizada na classe *Game*, que provê métodos para atualização e desenho de objetos, além de possuir uma lista de *GameComponents* e *Services*, que são atualizados e

desenhados automaticamente pela classe Game. Internamente, o *XNA* cria uma *thread* para cada componente e serviço e não há uma ordem de execução pré-especificada. A vantagem desta arquitetura é a velocidade na execução, uma vez que várias *threads* executando paralelamente se beneficiam dos processadores *multi-core*, bastante comuns hoje em dia.

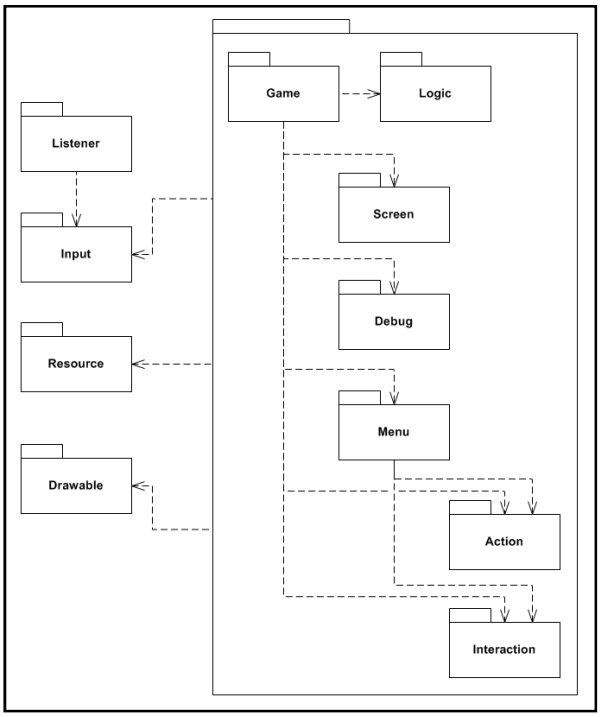


Figura 8: Arquitetura da versão final

6. Resultados

6.1 Funcionamento do jogo

O *IRTaktiks* é jogado por dois jogadores competindo entre si, que terão a disposição unidades de combates customizáveis. O controle destas unidades é feita através de *gestures*: para mover uma unidade basta tocar a unidade e arrastar o dedo pela superfície da mesa. Através do toque também são feitas as seleções de menu. Todas as ações podem ser executadas paralelamente por ambos os jogadores, com toques suaves, permitindo mais naturalidade ao jogar.

Não foi tomada nenhuma precaução específica relacionada a controle de acesso às unidades, ou seja, qualquer um que toque em determinada unidade poderá comandá-la. Considera-se que seja aceitável este funcionamento do jogo, pois este problema também está presente em jogos de tabuleiro convencionais como o xadrez, em que os jogadores ativamente obedecem às regras limitando-se a procurar controlar apenas suas peças.

Conforme se observa na Figura 9, a área de jogo é divida entre as unidades (1), os menus (2) e o mapa (3).

As unidades podem se movimentar pelo mapa e executar ações, controladas através dos menus laterais. As informações de cada unidade e dos jogadores também são exibidos neste menu. O posicionamento das unidades pelo mapa é um dos principais fatores de estratégia do jogo. A posição, de acordo com a altura do terreno, em combinação com as características das unidades, afeta o campo de visão e o alcance das habilidades.

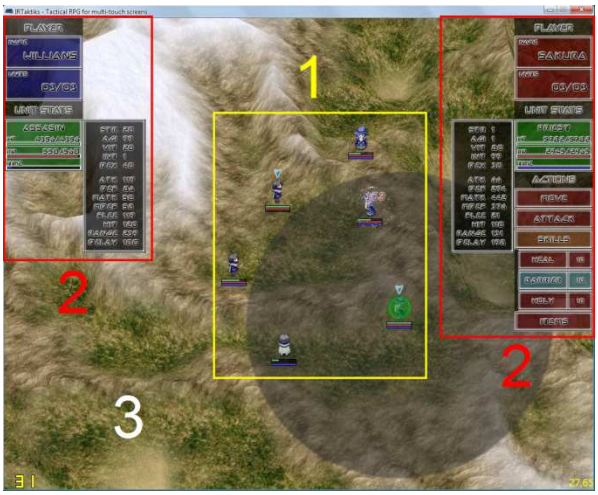


Figura 9: Elementos do jogo.

Outro fator que permite a estratégia é a personalização de cada uma das características das unidades que o jogador possui. Uma unidade possui características básicas que influenciam em desde seu ataque, velocidade de ação, defesa, magia e destreza até porcentagem de desvio e ataque, pontos de vida e mana, como podem ser visto na Figura 10. Estes atributos de cada unidade são configurados pelos dois jogadores simultaneamente numa tela pré-jogo.

Figura 10: Características de uma unidade

Além de características, as unidades também possuem tipos, que determinam as habilidades que a unidade vai possuir e quais características serão mais influentes. Sua combinação com as características básicas permite a construção de uma unidade voltada para ataque, defesa ou suporte de personagens, aumentando as possibilidades de jogo e uso de táticas.

Cada tipo de unidade possui ações específicas que podem ser disparadas sob o comando de seu jogador. Estas ações foram escolhidas de acordo com as características mais influentes, de modo a efetuar um balanceamento entre as unidades e não deixar um tipo mais forte que outro em todas as circunstâncias. Um exemplo do menu de ações com as habilidades de uma unidade pode ser visto na Figura 11.

Figura 11: Menu de ações de uma unidade

Todas as ações de um mesmo tipo seguem um mesmo padrão de utilização por parte do jogador. Há três tipos de ações: movimentação, uso e bônus. As ações de movimentação requerem que a unidade se movimente pelo mapa. Já as de uso necessitam que o jogador escolha uma unidade no mapa para ser o alvo da habilidade que será executada. Finalmente, as ações de bônus têm como alvo a própria unidade que a invocou, não necessitando de uma utilização especial por parte do jogador.

Quando uma ação de movimentação é escolhida, uma área circular é desenhada em volta da unidade, determinando os limites de movimento da mesma. De posse dessa informação, o jogador pode mover a unidade para qualquer posição dentro da área delimitada, como pode ser visto na Figura 12.

Quando uma ação de uso é selecionada, uma área é desenhada em volta do personagem determinando o limite de uso da habilidade escolhida e uma mira é criada. A escolha do alvo é feita arrastando essa mira sobre a unidade alvo. Se ela estiver sobre um aliado, sua cor será verde, enquanto sobre um inimigo ela ficará vermelha, como na Figura 13.

Figura 12: Unidade se movimentando dentro de uma área

Quando a mira é solta, a ação escolhida no menu é executada. Todas as ações são graficamente representadas com animações e as informações sobre modificações nos status das unidades são exibidos, como na Figura 14. Quando uma unidade recebe um bônus, a informação é exibida em verde, enquanto a informação de danos é mostrada em vermelho. Quando os pontos de vida de uma unidade chegarem a zero ela desmaia e se torna inativa. O jogo encerra quando um jogador conseguir deixar todas as unidades de seu adversário desmaiadas.

Figura 13: Mira sobre uma unidade inimiga

Quando uma ação é executada, independentemente de seu tipo, ela é regida por um fluxo, que determina como a ação deve ser executada. Isso se dá ao fato da ação poder ser executada sobre nenhuma unidade, ou ainda aplicar efeitos não instantâneos, ou seja, que duram um determinado periodo de tempo. O fluxo de execução de uma ação é exibido na figura 15.

Figura 14 - Exemplo de exibição de informações

6.1 Funcionamento do jogo

O *IRTaktiks* é executado em dois computadores: um deles tem a webcam e executa a *Touchlib*, que usa os protocolos *TUIO*/*OSC* para enviar os dados da interação ao módulo principal que executa em uma outra estação. Nos testes realizados utilizou-se um *Celeron* com 2.0 GHz para executar a *Touchlib* e uma estação com um *Intel Core 2 Duo* e placa gráfica *NVidea GeForce 8800* para execução da parte visual, que precisa ser compatível com *DirectX Pixel Shaders* 3.0 para poder gerar a malha 3D do terreno e executar um shader que define sua aparência com base em um mapa de alturas. Estes *shaders* não são um requisito para suportar a maior parte da funcionalidade do *IRTaktiks*, de modo que pode ser reescrito para executar em uma configuração mais modesta.

Cerca de 30 pessoas jogaram o *IRTaktiks* no dia 03/10 num evento no *Centro Universitário Senac*, e observou-se que a familiarização de novos usuários com os conceitos do jogo é bastante rápida. Depois de receber uma explicação sucinta sobre o que é o jogo e quais os objetivos e verem outros jogadores em ação, a maior parte das pessoas já era capaz de jogar rudimentarmente mesmo sem muita familiaridade com os diversos tipos de unidades, atributos e táticas permitidas pelo jogo.

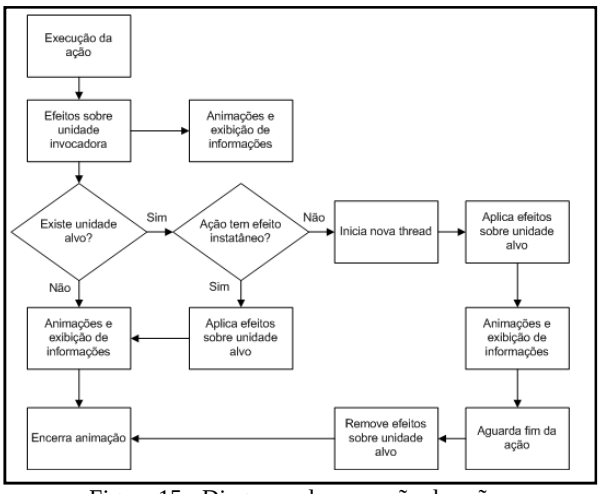


Figura 15 - Diagrama de execução de ações

7. Conclusões e Trabalhos Futuros

Durante a realização deste trabalho, percebeu-se a importância de uma adequada gerência nas interações dos diversos usuários da interface multi-toque. Ignorar este quesito no desenvolvimento de uma aplicação para este fim pode transformar a interação, que deveria ser fácil e natural, em algo difícil e cansativo. Na versão aqui apresentada supunha-se que apenas o jogador que é o legítimo dono de uma unidade tentaria comandá-la, mas às vezes por uma confusão sem má intenção por parte dos jogadores esta regra falhava em ser seguida.

Os principais objetivos traçados durante a concepção do jogo foram atendidos. Dois jogadores controlam suas respectivas unidades em batalhas, em que as características únicas de cada unidade, aliadas à tática são os fatores decisivos para derrotar o adversário. A interface natural e o bom *feedback* do jogo às ações do usuário também faz com que seja divertido mesmo para jogadores iniciantes.

Agradecimentos

Os autores gostariam de agradecer a João J. Maranhão Jr. pela construção da estrutura da mesa e pela primeira versão de sua parte elétrica. Agradecemos também ao Centro Universitário Senac por viabilizar com sua infra-estrutura o desenvolvimento do projeto.

Referências

HAN, J. Y. Low-Cost Multi-Touch Sensing through Frustrated Total Internal Reflection. *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, 2005.

BUXTON, Bill. *Multi-Touch Systems that I Have Known and Loved*. Disponível em: http://www.billbuxton.com/multitouchOverview.html (Acessado em 06 de outubro de 2008).

MULTIGESTURE.NET. *A Multi-Touch and Multi-Gesture research blog.* Disponível em: http://www.multigesture.net/ (Acessado em 06 de outubro de 2008).

NUIGROUP. *Touchlib: A multitouch Development Kit*. Disponível em: http://nuigroup.com/touchlib (Acessado em 06 de outubro de 2008).

CNMAT, UC Berkeley. *Open Sound Control*. Disponível em: http://opensoundcontrol.org/ (Acessado em 06 de outubro de 2008).

BENCINA, Ross. *OSCPack*. Disponível em: http://www.audiomulch.com/~rossb/code/oscpack/ (Acessado em 06 de outubro de 2008).

TUIO. *TUIO: A Protocol for Tangible User Interfaces*. Disponível em: http://tuio.lfsaw.de (Acessado em 06 de outubro de 2008).

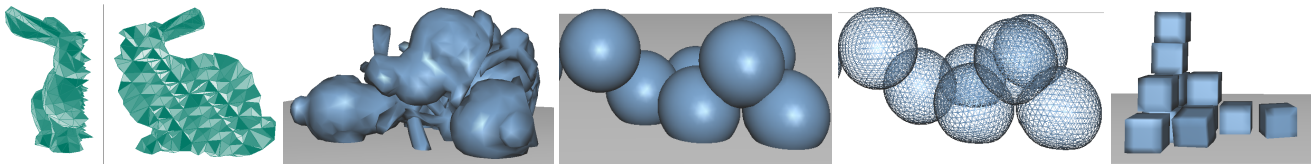
KALTENBRUNNER, MARTIN, ET AL. *TUIO: A Protocol for Table-Top Tangible User Interfaces.* Proceedings of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation (GW 2005). Vannes, France, 2005.

KALTENBRUNNER, MARTIN, BENCINA. *reacTIVision: A Computer-Vision Framework for Table-Based Tangible Interaction.* Proceedings of the First International Conference on Tangible and Embedded Interaction (TEI 2007). Baton Rouge, Louisiana, 2007.

KALTENBRUNNER, M. & JORDÀ, S. & GEIGER, G. & ALONSO, M. *The reacTable: A Collaborative Musical Instrument.* Proceedings of the Workshop on "Tangible Interaction in Collaborative Environments" (TICE), at the 15th International IEEE Workshops on Enabling Technologies (WETICE 2006). Manchester, U.K

Simulation of Deformable Bodies Based on Tetrahedral Meshes and Shape Matching

Guina Sotomayor Alzamora* Yalmar Ponce Atencio Claudio Esperança
Universidade Federal do Rio de Janeiro
Laborátorio de Computação Gráfica - LCG



Abstract

We present a simplified approach for animation of geometrically complex deformable objects represented by tetrahedral meshes. Our prototype system detects and responds to collisions of objects subject to elastic deformations of variable stiffness. The proposed approach combines several techniques, namely, collision detection using spatial hashing, collision response through a contact surface with consistent penetration depth obtained by propagation, estimated displacement vectors for the deformation region, and binary search to separate objects. The dynamics is based on shape matching and a modal analysis scheme, using an Euler explicit-implicit integrator. Preliminary results show that collisions between objects containing several hundreds tetrahedra can be animated in real-time.

Keywords:: Collision detection, physically based animation, deformable bodies

Author's Contact:

{yalmar, esperanca}@lcg.ufrj.br
*guinas@gmail.com

1 Introduction

Physically based animation is a computational process, where animated objects behave in a physically plausible manner. This is in contrast to "physics animation", which is employed to design simulations where physically correct behavior is sought. This work adopts the former rationale, since it employs simplified approaches, emphasizing performance over accuracy. Nevertheless, as with other physically based models, motion is still produced based on the same physical principles; namely, dynamics is based on Newton's laws.

Methods for animation of both rigid bodies and deformable bodies are commonly based on particle systems. In addition to the requirements for the animation of rigid bodies, deformable objects also require some physical deformation model such as mesh-based methods like mass-spring [Baraff and Witkin 1998] and finite elements, or meshless methods [Müller et al. 2004; Müller et al. 2005; Pauly et al. 2004]. Deformable objects also require a more involved representation, since their shapes can change in time, due to interaction with other objects or themselves.

This work focuses on the use of a simple methodology for animating geometrically complex deformable objects represented by tetrahedral meshes. The goal is obtain a stable dynamic simulation with physically plausible behavior. The presented method combines several techniques employed for the animation of deformable bodies, namely:

- A broad phase collision detection based on bounding spheres for filtering the set of potentially colliding objects.
- Narrow phase collision detection employing the *Spatial Hashing* approach [Teschner et al. 2003].
- Collision response is achieved by computing the penetration depth [Heidelberger et al. 2004] of all collided vertices. For asymmetric collisions we use the free obstacle method described in [Jakobsen 2001]. Finally, the separating process is based on a binary search [Heidelberger et al. 2004], which computes the contact surface.
- Dynamics is based on shape matching [Müller et al. 2005] which does not need a mesh representation of the model – the tetrahedral mesh is used only in the collision detection and response stages.
- The system integration uses semi-implicit Euler method [Müller et al. 2005].

Preliminary results are shown interacting objects with several hundred of tetrahedra in real time. On the other hand, the material properties are not yet modeled in our prototype.

The remainder of text is organized as follows: Section 2 reviews related works in collision detection and deformable models. Section 3 describes some required prerequisites. Section 4 describes the collision detection process in its two phases, broad and narrow. Section 5 describes the collision response process. Section 6 describes the shape matching technique employed for the dynamics. Section 7 describes the integration method. In Section 8 some results using our prototype are shown. Finally, concluding remarks and suggestions for future work are discussed in Section 9.

2 Related Work

Many methods and models have been proposed in computer graphics to simulate deformable objects including finite difference approaches [Terzopoulos et al. 1987], mass-spring systems [Baraff and Witkin 1998] Finite Element Methods (FEM) [Müller et al. 2002; Müller et al. 2004], modal analysis [Shen et al. 2002] and mesh-free particle systems [Müller et al. 2005]. Most of these approaches focus on the accurate simulation of elasto-mechanical properties which demand a costly processing, notwithstanding several acceleration strategies such integration schemes for large time-steps [Baraff and Witkin 1998] and shape matching in place of accurate modal analysis [Müller et al. 2005].

Collision detection encompasses a whole research area by itself. Approaches are usually divided into broad and narrow phase methods:

- The Broad phase aims at reducing the quadratic number of potentially colliding pairs of objects to a more manageable number. Several schemes using spatial data structures such as Octrees, Sphere Trees or BSP-trees have been proposed [Samet 2006]. The popular *sweep and prune* technique [Cohen et al. 1995] maintains a sorted list for each of the principal axes, where the elements in the list are intervals obtained by projecting the geometry of the object onto the axis.
- The goal of the Narrow phase is to detect pairs of objects which indeed collide. Methods are usually classified into four groups: Bounding volume (Axis Aligned Bounding

Box, Oriented Bounding Box, Bounding Sphere, discrete oriented polytope, among other) hierarchies [Cohen et al. 1995; Gottschalk et al. 1996; van den Bergen 1997; Bradshaw and O'Sullivan 2004; James and Pai 2004], spatial subdivision (grids) [Ganovelli et al. 2000], spatial subdivision hierarchies (Octrees, k-d Trees, BSP-Tree, among other) [Samet 2006; Luque et al. 2005] and image-space techniques [Shinya and Forgue 1991; Knott and Pai 2003; Heidelberger et al. 2004].

Among these, the preferred narrow phase methods for detecting the collision of rigid bodies tend to be those based on bounding volume hierarchies. Unfortunately, they are not very suitable for handling deformable objects, since the cost for updating the hierarchies cannot be amortized with the pre-processing of fixed shapes. On the other hand, spatial subdivision schemes lend themselves more easily to this task. In particular, we chose to employ the *spatial hashing* method from Teschner [Teschner et al. 2003], which supports efficient updating.

Collision response consists of updating properties such as shape, position and velocity of objects involved in collisions. A common approach is to compute penalty forces for penetrating object vertices as a function of their penetration depth, i.e., the minimum translation for separating them [van den Bergen 1999]. In this work we use Heidelberger's consistent penetration method [Heidelberger et al. 2004], which computes the penetration depth for all collided vertices. Later, that information is used for separating the objects.

3 Preprocessing

Firstly, a spatial hash data structure must be defined. In essence, the idea consists of using a grid to subdivide the space where collisions are expected to occur into cells of equal size. Similarly, each moving body is subdivided into tetrahedra. The method then detects collisions by keeping track of voxels (cells) intersected by tetrahedra associated with different objects. Rather than using a 3D array of cells, however, a more memory efficient scheme consists of mapping grid cells to a much smaller array using a hash function. Thus, the data structure is dependent on the following parameters:

Hash table size. Its optimal size is related with the number of primitives in the scene, and must be a high prime number in order to minimize address collisions in the hash table.

Grid cell width. Clearly, this should be of the same size order as the objects' tetrahedra. Thus, a reasonable choice is to employ the tetrahedra's average edge length.

***Hash* function.** Maps a grid cell to an arbitrary hash table address. An efficient function should be able to achieve a good distribution. In our prototype, following function is used:

$$h = hash(i, j, k) = (i\, c_i \oplus j\, c_j \oplus k\, c_k) \bmod n, \qquad (1)$$

where $\oplus$ stands for bitwise exclusive-or operation, i, j, k are grid coordinates, c_i, c_j and c_k are high prime numbers and n is the hash table size.

Additionally, the minimum bounding sphere for each object is precomputed. These are used in the broad phase collision detection (see Section 4.1). In order to accomodate the stretching of deformable objects, an error margin is added to the sphere radius which is dependent on the object's stiffness constant α (see Section 7). In particular, for a deformable object with stiffness α whose rest shape has a bounding sphere of radius r, we consider a sphere of radius $\frac{2r}{1+\alpha}$.

4 Collision detection

In this work, we are mainly interested in simulating a relatively small number of objects (a few tens), although each object may be arbitrarily complex. This has guided the choice of collision detection schemes detailed below.

4.1 Broad phase

The goal of this phase is to select regions on the grid where collisions may have taken place. Therefore, a simple verification between each pair of objects is done, considering only their bounding spheres. For each pair of objects A and B, collision potentially occurs if the distance between their centers is less than the sum of their radii, i.e.,

$$|c_1 - c_2| < r_1 + r_2. \qquad (2)$$

For each pair of objects, the vertices involved in the region of potential collision ($v \subset A \cap B$) are collected in order to be later checked for collision during the narrow phase. For this purpose, it suffices to verify if a vertex $v \in A$ is inside the bounding sphere of B and vice-versa (see Figure 1).

An important difference between our method and Teschner et al.'s approach is that rather than updating the grid with the new positions of all tetrahedra, only those associated with objects potentially involved in collisions are considered. Thus, a tetrahedron t_A associated with object A, whose bounding sphere collides with that of B, will be updated in the grid only if it also intersects B's bounding sphere.

It should also be noted that *timestamps*, as discussed in [Teschner et al. 2003] are also used in our prototype. The idea is to avoid cleaning up the grid after each frame by labeling each grid cell with a timestamp associated with the moment it was last updated. Thus, a collision with primitives inside a given grid cell is considered only if it was updated in the current iteration.

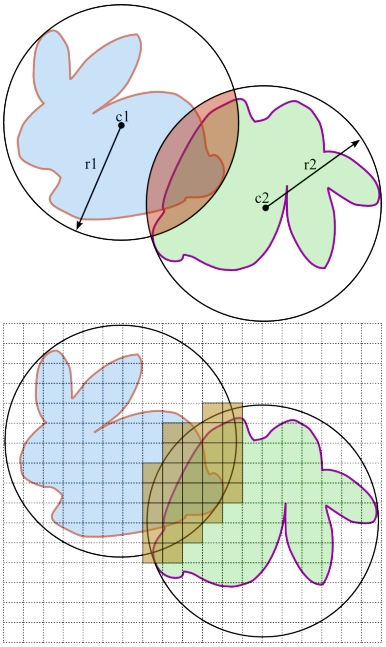


Figure 1: *Interference check during broad phase collects potentially colliding vertices inside intersecting bounding spheres (top); narrow phase will be restricted to grid cells intersecting a region of potential collision (bottom).*

4.2 Narrow phase

Once potential collision regions are found, participating primitives are submitted to an exact collision test. The test consists in using the hash function to visit the cell C associated with each potentially colliding vertex v. An exact intersection test is then conducted between v and each tetrahedron t_C referenced in C, provided that v is not one of the vertices of t_C.

5 Collision response

In a nutshell, the collision response process starts by computing the penetration depth of all collided vertices. Next, deformation

regions for all objects are computed. Finally, the objects are separated by approximately half of the penetration depth distances of all colliding vertices using binary search.

5.1 Penetration depth

The penetration depth between intersecting objects is the minimum translation necessary for separating them. Generally speaking, collision response for a pair of colliding rigid bodies can be computed based solely on this information. For deformable objects, however, the penetration depth of all colliding vertices must be computed. The use of tetrahedral meshes is very helpful in this task, since its incidence information can be used to track the deformation region and to compute the penalty forces, thus leading to a realistic collision response [Heidelberger et al. 2004].

The idea is to process the colliding vertices ranked by their penetration depth. Firstly, colliding vertices closest to the surface are evaluated. Later, this information is propagated to deeper colliding vertices using the tetrahedral meshes' incidence relations. As a result, penetration depth d and direction $\vec{r}$ are estimated for all relevant vertices.

Additionally, the process needs to find a contact triangle for each *border vertex* (described below) v. This triangle is a face on the penetrated object surface that intersects a ray with origin in v and direction $\vec{r}_v$.

Classification of colliding vertices: if a colliding vertex has one or more non-colliding incident vertices, it is called a *border vertex*, otherwise, it is called an *internal vertex* (see Figure 2).

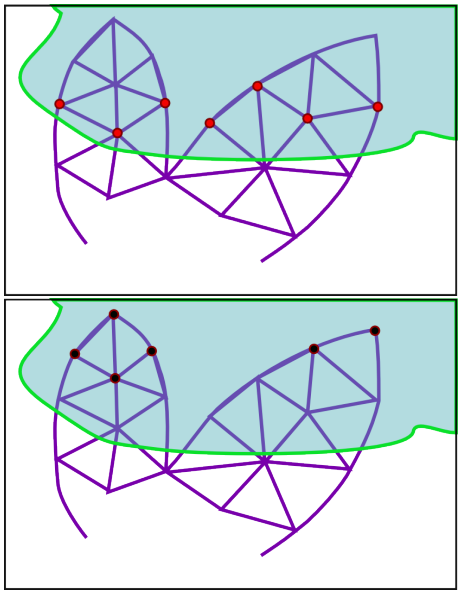


Figure 2: *border (top) and internal (bottom) vertices.*

Penetration depth of the border vertices: the penetration depth of border vertices depends on the computation of *contact points* and their normals. The underlying idea is to identify edges that contain one non-colliding point and one colliding point. These edges are called *intersecting edges*. Moreover, the exact intersection point of each edge with the surface of the penetrated object must be computed. This can be done efficiently by visiting all grid cells which intersect each edge and then testing all surface triangles in them against the edge.

The triangle-edge intersection is done in two passes: first, the triangle plane and the edge are tested for intersection. If successful, the barycentric coordinates (t_1, t_2, t_3) of the intersection point p is computed. Then, p is inside the triangle if $t_i > 0$ for $i = 1, 2, 3$. Additionally, the surface normal $\vec{n}$ is linearly interpolated from the three triangle vertex normals $(\vec{n}_1, \vec{n}_2, \vec{n}_3)$, i.e., $\vec{n} = \sum t_i \vec{n}_i$.

If more than one intersection point for a given edge is found, the point closest to the colliding vertex is chosen. The chosen points are called *contact points*. Finally, this data is used to compute the border vertices' penetration depth (see the Figure 3).

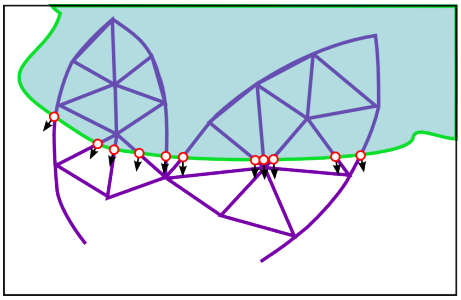


Figure 3: *contact points with their normals on the intersected faces.*

For each border vertex v, its penetration depth $d(v)$ and its penetrating direction vector $\vec{r}(v)$ are computed as follows: for each intersecting edge e_i incident in v, a weight w_i between the contact point p_i and v is given by:

$$w(p_i, v) = \frac{1}{\|p_i - v\|^2}. \tag{3}$$

Next, $d(v)$ and $\vec{r}(v)$ are computed using the following equations:

$$d(v) = \frac{\sum_{i=1}^{k} w(p_i, v)(p_i - v) \cdot \vec{n}_i}{\sum_{i=1}^{k} w(p_i, v)}, \tag{4}$$

$$\vec{r}(v) = \frac{\sum_{i=1}^{k} w(p_i, v)\vec{n}_i}{\sum_{i=1}^{k} w(p_i, v)}, \tag{5}$$

where k is the number of intersecting edges incident in v and p_i and $\vec{n}_i$ represents the i-th contact point and normal to be evaluated, respectively.

Penetration depth for internal vertices: After all border vertices are processed, their penetration information is employed for computing the penetration depth of associated internal vertices u_i. This is done by propagation, i.e., once the first layer of internal vertices are processed, edge-neighbor internal vertices are processed in turn. Thus, the penetration depth computation is done by levels, until the penetration depth information for all colliding vertices is determined.

Given an internal vertex u, its penetration depth $d(u)$ and penetration direction $\vec{r}(u)$ are computed using weights. First, for a non-processed internal vertex u incident in an already processed vertex v_j, a weight is computed as

$$\mu(u, v_j) = \frac{1}{\|v_j - u\|^2}.$$

Next, $d(u)$ and $\vec{r}(u)$ are computed by a weighted average of the contributions of all incident processed vertices using the following equations:

$$d(u) = \frac{\sum_{j=1}^{k} \mu(u, v_j)((v_j - u) \cdot \vec{r}(v_j) + d(v_j))}{\sum_{j=1}^{k} \mu(u, v_j)}, \tag{6}$$

$$\vec{r}(u) = \frac{\sum_{j=1}^{k} \mu(u, v_j)\vec{r}(v_j)}{\sum_{j=1}^{k} \mu(u, v_j)}, \tag{7}$$

where k is the number of processed vertices incident in u. Figure 4 illustrates the penetration directions of colliding vertices.

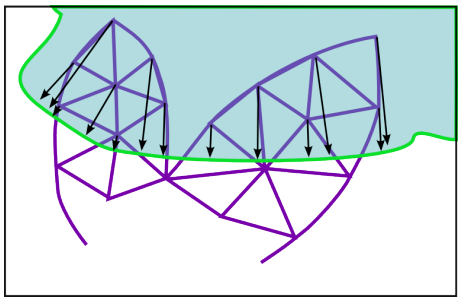


Figure 4: *penetration directions for colliding vertices.*

5.2 Deformation region

After the computation of the penetration depth for all colliding vertices, a separation process must be executed in order to reach a non-overlapping state. To this end, a region of deformation defined by colliding vertices is obtained. In addition to these vertices, the deformation region also includes vertices on colliding faces (contact faces), which not necessarily are colliding vertices (x_i, x_j and x_k in Figure 5). Collisions with this configuration are called *asymmetric collisions*.

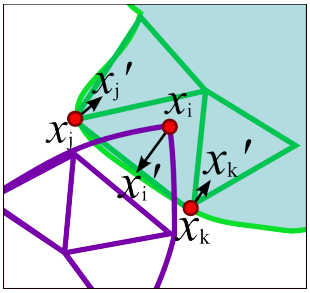


Figure 5: *treating asymmetric collisions.*

The handling of asymmetric collisions in our approach deviates from the technique proposed in [Spillmann and Teschner 2005]. Rather than using a scheme based on barycentric weights, we employ the simpler projection technique described in [Jakobsen 2001]. The idea consists in projecting the colliding vertices out of the obstacle, moving them until they are intersection-free. For example, in Figure 5, the displacement vector of x_i is its penetration depth, and the displacement vectors for the non-colliding vertices are

$$\vec{s}_j = \frac{\alpha_1}{\alpha_1^2 + \alpha_2^2}(x_i' - x_i),$$

$$\vec{s}_k = \frac{\alpha_2}{\alpha_1^2 + \alpha_2^2}(x_i' - x_i),$$

where x_i is the colliding vertex, x_i' is its projection on contact edge x_j-x_k, $\vec{s}_j$ and $\vec{s}_k$ are the displacement vectors for x_j and x_k respectively, and α_1 and α_2 represent the proportional displacement factors for the vertices on the contact edge, subject to $\alpha_1 + \alpha_2 = 1$. Notice that a contact edge in 2D corresponds to a contact triangle in 3D, and constants α_i are proportional areas. Namely, if the contact triangle has vertices x_k, x_j and x_l and x_i' is the projection of the colliding point onto that triangle, then, $A = Area(x_j, x_k, x_l)$, $\alpha_1 = Area(x_k, x_l, x_i')/A$, $\alpha_2 = Area(x_j, x_l, x_i')/A$, and $\alpha_3 = Area(x_j, x_k, x_i')/A$.

Once all displacement vectors for colliding vertices have been computed, a contact surface is defined. Conceptually, the contact surface should be equidistant from the colliding vertices, and thus an initial estimate of its position is obtained by translating collided vertices by half of their associated displacement vectors. This estimate is then refined by means of a binary search scheme [Spillmann and Teschner 2005]: if x^1 is the initial estimate for a surface vertex, and s is its associated displacement vector, then the i^{th} estimate for that vertex is given by

$$x^i \leftarrow x^{i-1} \pm \frac{1}{2^{(i+1)}} s, \qquad (8)$$

where the direction of the movement is determined by the sign of the pressure difference. In practice, three or four iterations are sufficient to produce a contact surface with adequate accuracy.

6 Dynamics based on shape matching

In order to estimate the dynamic behavior of collided objects, a shape matching technique [Müller et al. 2005] is used. A large part of its appeal is that, being a meshless method, only a set of particles (the object vertices) are considered. The idea is to match vertices in their initial states $\mathbf{x}_i^0$ with their positions in the current state $\mathbf{x}_i$, i.e., the positions of the vertices as a result of deformation. The matching consists of determining an affine [1] transformation matrix using a least-squares approach. This transformation is a model for a plausible deformation and provides a simple way for the object to recover its original shape (see the Figure 6). Once the transformation is computed, goal positions $\mathbf{g}_i$ are estimated for each vertex. These are then used to estimate the new position and velocity of each particle for the next time step.

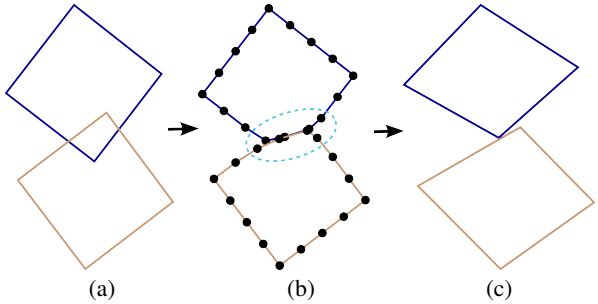


Figure 6: *(a) two colliding objects, (b) after a contact surface is determined, and (c) deformed with shape matching.*

The algorithm has two main components: (1) find the optimal rigid transformation that approximates a new position and orientation for the object (matching problem) and (2) move the vertices for the goal positions applying a linear deformation model.

Considering the particles weights, the matching problem consists of finding translation vectors $\mathbf{t}$ and $\mathbf{t}_0$ and rotation matrix $\mathbf{R}$ which minimize:

$$\sum_i w_i(\mathbf{R}(x_i^0 - \mathbf{t}_0) + \mathbf{t} - \mathbf{x}_i)^2.$$

For simplicity, $w_i = m_i$ is used, where m_i is the mass of particle i. The optimal translation vectors between the *centers of mass* of the initial position and the current position are given by

$$\mathbf{t}_0 = \mathbf{x}_{cm}^0 = \frac{\sum_i m_i \mathbf{x}_i^0}{\sum_i m_i}, \quad e \quad \mathbf{t} = \mathbf{x}_{cm} = \frac{\sum_i m_i \mathbf{x}_i}{\sum_i m_i}.$$

The optimal rotation $\mathbf{R}$ may be found by first determining a general linear transformation matrix $\mathbf{A}$. Notice that $\mathbf{A}$ not necessarily is orthonormal. Considering the relative particle positions defined by

$$\mathbf{q}_i = \mathbf{x}_i^0 - \mathbf{x}_{cm}^0 \quad and \quad \mathbf{p}_i = \mathbf{x}_i - \mathbf{x}_{cm},$$

the correspondence problem may be solved in the least-square sense. Thus, the optimal linear transformation matrix $\mathbf{A}$ is found minimizing the term:

$$\sum_i m_i(\mathbf{A}\mathbf{q}_i - \mathbf{p}_i)^2.$$

By setting to zero all the first derivatives of this expression with respect to the coefficients of $\mathbf{A}$, we obtain

$$\mathbf{A} = (\sum m_i \mathbf{p}_i \mathbf{q}_i^T)(\sum m_i \mathbf{q}_i \mathbf{q}_i^T)^{-1} = \mathbf{A}_{pq}\mathbf{A}_{qq}, \qquad (9)$$

[1] [Müller et al. 2005] also considers quadratic transformations.

where $\mathbf{A}_{pq}$ is a correlation matrix and $\mathbf{A}_{qq}$ is a symmetric matrix that can contain only scale information but not orientation. Therefore, for finding $\mathbf{R}$, it is necessary to find the rotation component of $\mathbf{A}_{pq}$. Following [Müller et al. 2005], we use a polar decomposition method [Shoemake and Duff 1992] to obtain $\mathbf{R} = \mathbf{A}_{pq}\mathbf{S}^{-1}$, where symmetric matrix $\mathbf{S} = \sqrt{\mathbf{A}_{pq}^T\mathbf{A}_{pq}}$. In order to obtain $\mathbf{S}^{-1}$, matrix $\mathbf{A}_{pq}^T\mathbf{A}_{pq}$ is diagonalized using from 5 to 10 Jacobi rotations [Sumali 1992].

Finally, all particles are moved to their goal positions computed as

$$\mathbf{g}_i = \mathbf{R}(\mathbf{x}_i^0 - \mathbf{x}_{cm}^0) + \mathbf{x}_{cm},$$

and the process is repeated in every time step.

7 Integration

As in [Müller et al. 2005], for the integration we use a modified Euler's method, which includes an explicit part (velocity updating) and an implicit part (position updating). The integration scheme is given by

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \alpha\frac{\mathbf{g}_i(t) - \mathbf{x}_i(t)}{\Delta t} + \frac{\Delta t}{m_i}f_{ext}(t), \qquad (10)$$

$$\mathbf{x}_i(t+\Delta t) = \mathbf{x}_i(t) + \Delta t\mathbf{v}_i(t+\Delta t), \qquad (11)$$

where $\mathbf{v}_i(t+\Delta t)$ is the velocity after the time step, $\mathbf{v}_i(t)$ is the current velocity, $\alpha \in [0,1]$ is a parameter which controls the rigidity of the object, $\mathbf{g}_i(t)$ is the goal position for vertex $\mathbf{x}_i(t)$, Δt is the time step, m_i is the particle mass and $\mathbf{f}_{ext}(t)$ are the external forces (gravity, wind, among others).

8 Results

A prototype implementation of the proposed system was developed using the C++ language and the OpenGL and Glut libraries. For performance reasons, *Vertex Buffer Objects* (VBOs) were used to store object geometry.

Several experiments were conducted with this prototype in order to evaluate the correctness and speed of the technique. These involved simulations of collections of objects with different shapes. All experiments were conducted in a PC running Linux (Fedora 8) operational system and equipped with a Intel Core 2 Duo processor running at 2.4Ghz, 1GB memory and an NVidia GeForce 8600 GS graphics board.

The experiments aimed to gauge the system performance as a function of both the number and geometric complexity of the objects involved in the simulation. Table 1 shows the object resolutions (number of vertices, number of faces and number of tetrahedra). In

Table 1: *objects with different resolutions.*

Object	surface vertices	total vertices	faces	tetraedra
bunny	436	510	868	1750
duck	424	519	846	1819
sphere	386	729	768	2560

all experiments, the rigidity coefficient α was set to 0.8, i.e., quasi-rigid objects. Additionally, it was collected information about: number of colliding primitives (vertices in potential collision, faces, tetrahedra and vertices in real collision) processed at each time step, and the percentage spent in each sub-process (broad phase collision detection, narrow phase collision detection, penetration depth computation and shape matching) in milliseconds.

A first experiment consists of the simulation of a scene with different kinds of objects (see Figure 7). The scene contains 3 ducks, 2 bunnies and 3 spheres (see Table 1). This experiment demonstrates that the prototype is able to handle objects with arbitrary geometry, provided that a 3D triangulation for them is known. From the experiment, some information was extracted regarding the various

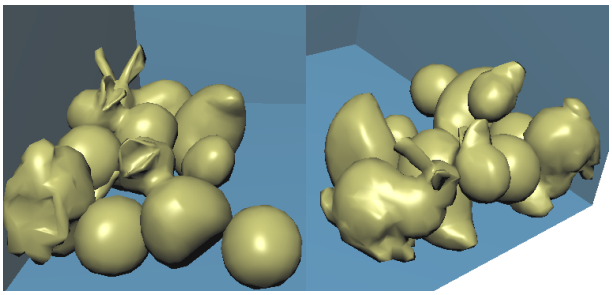


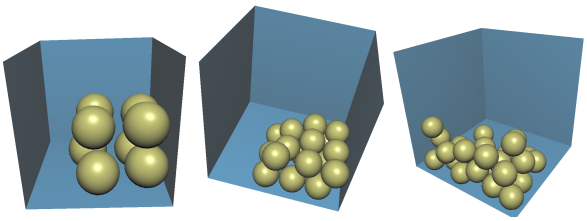
Figure 7: *eight objects in contact: 3 ducks, 2 bunnies e 3 spheres. The scene contains 2952 vertices and 9917 tetraedra that are animated at aprox. 32 fps.*

stages of our algorithm. Figure 8 shows as a function of simulation time the time in milliseconds spent in the four main sub-processes: broad phase collision detection, narrow phase collision detection, penetration depth computation and shape matching. Notice that the collision detection, in both stages, spends much more time than the other sub-processes. The penetration depth computation is almost constant and the time spent on shape matching is negligible.

A better notion about the behavior of the collision detection algorithms may be aprehended from Figure 9, which shows, for the same experiment, the quantity of primitives in collision (vertices in potential collision, faces, tetrahedra and vertices in real collision) for each time-step. Notice that the broad phase filters out a significative number of non colliding primitives and that, on average, only about 10% of the potential colliding vertices are effectively colliding vertices. Additionally, other three experiments were conducted, involving a varying number of spheres. Figure 10, shows simulations of scenes with 8, 18 and 27 spheres. The number of primitives contained in each scene as well as the simulation frame rates are tabulated in Table 2. A chart with the frame rate for each time-step is shown in Figure 11.

Table 2: *primitives for experiments with spheres.*

Number of spheres	vertices	tetraedra	average fps
8	5832	20480	62
18	19683	46080	41
27	19683	69120	22



(a) (b) (c)

Figure 10: *experiments with 8 (a), 18 (b) e 27 (c) spheres.*

9 Conclusions and future work

We have described a method for physically based animation of deformable objects which combines several techniques such as collision detection using spatial hashing [Teschner et al. 2003] and bounding spheres, collision response through the surface contact computation that utilizes penetration depth computation by propagation [Heidelberger et al. 2004], the computation of displacement vectors [Heidelberger et al. 2004] and handling of asymmetric collisions [Jakobsen 2001], binary search in order to separate the objects [Spillmann and Teschner 2005], and a deformation and dynamics model based on shape matching and an explicit-implicit Euler integrator [Müller et al. 2005].

The original structure of the collision detection was extended in order to use a broad phase mechanism, which detects regions on potential collision avoiding to update the hash table unnecessarily. The narrow phase is restricted to cell grids where potential collision was detected. In addition, asymmetric collisions are handled by projection [Jakobsen 2001].

The prototype system enables the simulation of scenes with plausible physics behavior and containing objects with complex geometry. Collision response is implemented in a robust manner and handles stacked objects.

As future work, we plan to extend this prototype with quadratic and plastic deformations. Both the broad and narrow phase collision detection can be improved using more efficient techniques. For instance, the bounding sphere scheme and vertex/tetrahedron tests can easily be implemented in GPU. Other aspects of the system may be harder to port to GPU, but we intend to investigate recent advances in this area such as the CUDA [NVIDIA™2007] technology.

Acknowledgements

We would like to acknowlege the support of Brazilian agencies CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnologico) and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior).

References

BARAFF, D., AND WITKIN, A. 1998. Large steps in cloth simulation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 43–54.

BRADSHAW, G., AND O'SULLIVAN, C. 2004. Adaptive medial-axis approximation for sphere-tree construction. *ACM Trans. Graph. 23*, 1, 1–26.

COHEN, J. D., LIN, M. C., MANOCHA, D., AND PONAMGI, M. 1995. I-collide: an interactive and exact collision detection system for large-scale environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 189–ff.

GANOVELLI, F., DINGLIANA, J., AND O'SULLIVAN, C. 2000. Buckettree: Improving collision detection between deformable objects. In *Spring Conference in Computer Graphics (SCCG)*.

GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D. 1996. Obbtree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 171–180.

HEIDELBERGER, B., TESCHNER, M., KEISER, R., AND MÜLLER, M. 2004. Consistent penetration depth estimation for deformable collision response. In *Proceedings of Vision, Modeling, Visualization VMV'04*, 157–164.

JAKOBSEN, T. 2001. Advanced character physics. In *Proceedings, Game Developer's Conference 2001*, GDC Press, SJ, USA.

JAMES, D. L., AND PAI, D. K. 2004. BD-Tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (SIGGRAPH 2004) 23*, 3 (Aug.).

KNOTT, D., AND PAI, D., 2003. Cinder: Collision and interference detection in real–time using graphics hardware.

LUQUE, R. G., JO A. L. D. C., AND FREITAS, C. M. D. S. 2005. Broad-phase collision detection using semi-adjusting bsp-trees. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 179–186.

MÜLLER, M., DORSEY, J., MCMILLAN, L., JAGNOW, R., AND CUTLER, B. 2002. Stable real-time deformations. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, 49–54.

MÜLLER, M., KEISER, R., NEALEN, A., PAULY, M., GROSS, M., AND ALEXA, M. 2004. Point based animation of elastic, plastic and melting objects. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 141–151.

MÜLLER, M., HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2005. Meshless deformations based on shape matching. In *Proceedings of SIGGRAPH'05*, 471–478.

NVIDIA™, 2007. CUDA Environment – Compute Unified Device Architecture. http://www.nvidia.com/object/cuda_home.html.

PAULY, M., PAI, D., AND GUIBAS, L. 2004. Quasi-rigid objects in contact. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 109–119.

SAMET, H. 2006. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

SHEN, C., HAUSER, K. K., GATCHALIAN, C. M., AND O'BRIEN, J. F. 2002. Modal analysis for real-time viscoelastic deformation. In *SIGGRAPH '02: ACM SIGGRAPH 2002 conference abstracts and applications*, ACM, New York, NY, USA, 217–217.

SHINYA, M., AND FORGUE, M.-C. 1991. Interference detection through rasterization. In *The Journal of Visualization and Computer Animation*, vol. 2, 132–134.

SHOEMAKE, K., AND DUFF, T. 1992. Matrix animation and polar decomposition. In *Proceedings of the conference on Graphics interface '92*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 258–264.

SPILLMANN, J., AND TESCHNER, M. 2005. Contact surface computation for coarsely sampled deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV'05*, 16–18.

SUMALI, H. 1992. *A New Adaptive Array of Vibration Sensors*. PhD thesis, Mechanical Engineering Virginia Polytechnic Institute and State University, Virginia, USA.

TERZOPOULOS, D., PLATT, J., BARR, A., AND FLEISCHER, K. 1987. Elastically deformable models. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 205–214.

TESCHNER, M., HEIDELBERGER, B., MÜLLER, M., AND POMERANETS, D. 2003. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV'03Proceedings of SPM 2005*, 47–54.

VAN DEN BERGEN, G. 1997. Efficient collision detection of complex deformable models using AABB trees. *J. Graph. Tools 2*, 4, 1–13.

VAN DEN BERGEN, G. 1999. A fast and robust GJK implementation for collision detection of convex objects. *Journal Graph. Tools 4*, 2, 7–25.

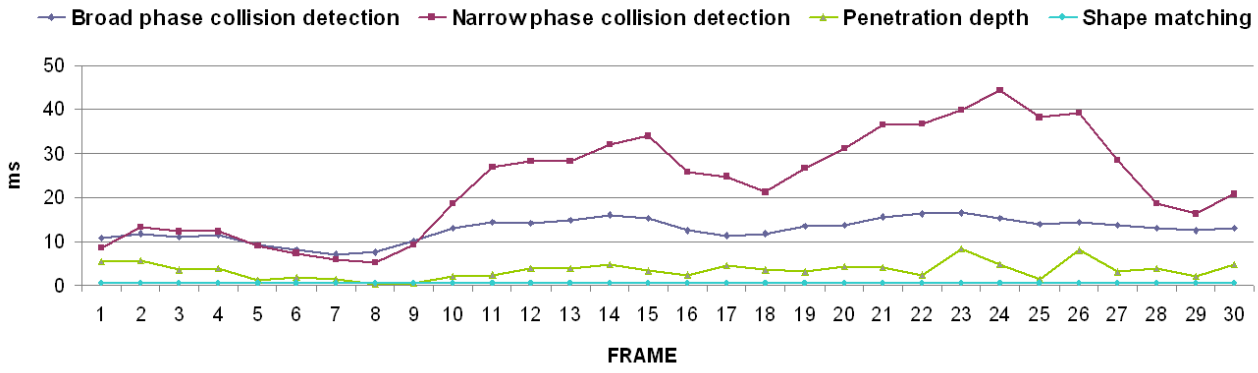


Figure 8: *spent time in milliseconds for each sub-process at each time-step.*

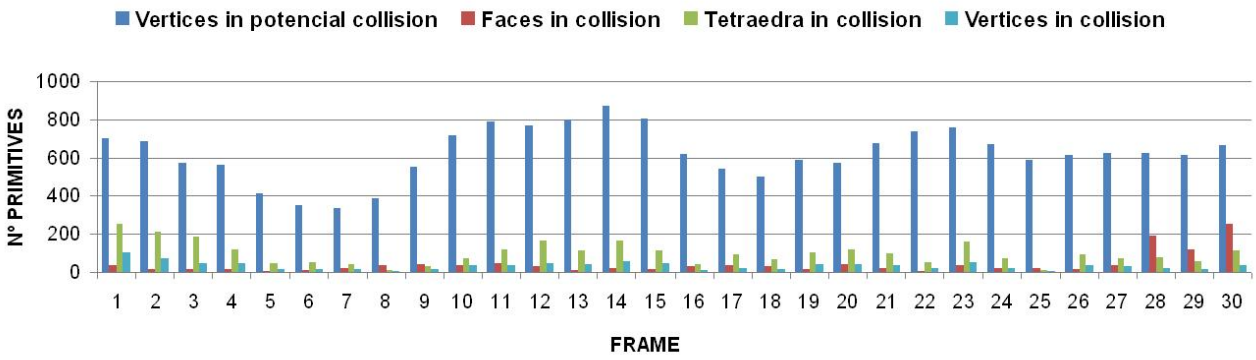


Figure 9: *quantity of primitives in collision for each time-step.*

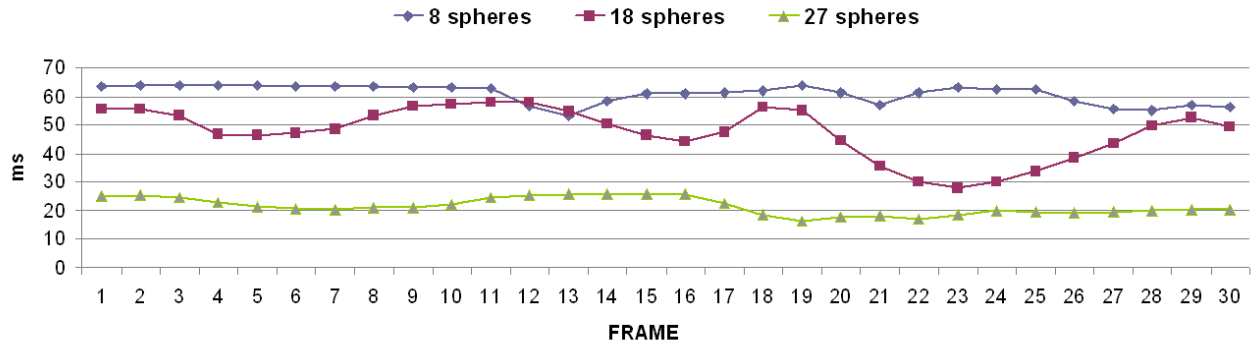


Figure 11: *frame rate for each time-step for scenes with 8, 18 and 27 spheres.*

An Adaptative Game Loop Architecture with Automatic Distribution of Tasks between CPU and GPU

Mark Joselli UFF,Medialab
Marcelo Zamith UFF,Medialab
Esteban Clua UFF,Medialab
Anselmo Montenegro UFF,Medialab
Regina Leal-Toledo UFF,IC
Aura Conci UFF,Medialab
Paulo Pagliosa UFMS,DCT
Luis Valente PUC-Rio,VisionLab
Bruno Feijó PUC-Rio,VisionLab

Abstract

This paper presents a new architecture to implement any game loop models for games and real-time applications that uses the GPU as a Mathematics and Physics co-processor, working in parallel processing mode with the CPU. The model applies concepts of automatic task distribution. The architecture can apply a set of heuristics defined in Lua scripts to get acquainted about what is the best processor for handling a given task. The model applies the GPGPU (General-Purpose Computation on GPUs) paradigm. The architecture that this work proposes acquires knowledge about the hardware by running tasks in each processor, and by studying their performance over time, learning about what is the best processor for a group of tasks.

Keywords:: Game loops, GPGPU, Task distribution

Author's Contact:

{mjoselli, mzamith, esteban, anselmo, leal,aconci}@ic.uff.br
pagliosa@dct.ufms.br
{lvalente, bruno}@inf.puc-rio.br

1 Introduction

Real time systems, like games, are defined as solutions that have time constraints to run their tasks. So, if for any reason the system is not able to execute its work under some time threshold, it will fail. In order to achieve such constraints game loops can be implemented.

Computer games are multimedia applications that employ knowledge of many different fields, such as Computer Graphics, Artificial Intelligence, Physics, Network and others. More, computer games are also interactive applications that exhibit three general classes of tasks: data acquisition, data processing, and data presentation. Data acquisition in games is related to gathering data from input devices as keyboards, mice and joysticks. Data processing tasks consist on applying game rules, responding to user commands, simulating Physics and Artificial Intelligence behaviors. Data presentation tasks relate to providing feedback to the player about the current game state, usually through images and audio.

Many computer games offer experiences where many actions seem to happen at once. However, computers usually have limited resources, so it is necessary to harvest the results of all processes involved in a game and present them to the player. If the application is not able to perform this work on time, the user may not receive continuous feedback, and the interactivity that the game should provide will not be acceptable. Hence, one of the main requirements of a game will not be fulfilled. This issue characterizes computer games as a heavy real-time application.

A common parameter to measure computer game performance is the number of frames per second (FPS) displayed on the screen. A frame represents an image displayed on the screen. A common accepted lower bound for interactive rates is 16 frames per second. Usually, a frame rate from 50 to 60 FPS is considered optimal.

Nowadays, computers and new video game consoles (such as the Xbox 360 and the Playstation 3) feature multicore processors. For this reason, game loops that take advantage of these resources are likely to become important in the near future. Therefore, parallelizing game tasks with multiple threads is a natural step. In order to take advantage of different hardware, a generic architecture for game loops and a multi thread game loop with this architecture are present.

The development of programmable GPUs (graphics processing units), has enabled new possibilities for general purpose computation (GPGPU) which now can be used to process some of the common tasks of the game loop, like data processing tasks. This is good news for games due to the parallel architecture of the latest GPUs, which have more processing power than CPUs. GPUs perform better than CPUs when large amounts of data are involved, but to take advantage of this power, it necessary to develop a different approach than the traditional CPU sequential model. Hence, due to architectural characteristics, the CPUs are more suitable for processing small amounts of data while the GPUs are more suitable for large amounts of data. In order to achieve better performance in both cases (small and large amount of data), it is necessary to implement an automatic method to distribute tasks between the CPU and the GPU. For this heuristics to work better with different hardware architectures, they are implemented in a script language.

The main objective of this work is present a new game loop architecture that can be used to implement any game loop model and can take advantage of automatic dynamic distribution of tasks between the CPU and the GPU. This distribution is based on heuristics that are defined in Lua [Lua] scripts. This work presents concepts that could be applied also to other hardwares like the PlayStation 3 with the Cell Processor [Hofstee 2005].

This paper is organized as follows. Section 2 presents GPGPU concepts. Section 3 presents related works in game loops and task distribution between CPU and GPU. Section 4 presents the generic architecture for game loops. Section 5 presents the test case with the test game loop and the test heuristics, and Section 6 presents the results. Finally, Section 7 presents the conclusions.

2 GPGPU

Graphics Processors Units or simply GPUs are processors dedicated to mathematical processing in the graphics pipeline. The evolution of those processors allows it to be used for processing other mathematical tasks.

The GPUs have been evolving constantly and in a faster way than the CPUs, acquiring superior computation power. A nVidia 8800 ultra [NVIDIA 2006], for instance, can sustain a measured 384 GFLOPS's against 35.3 GFLOPS's for the 2.6 GHz dual core Intel Xeon 5150 [NVIDIA 2008]. This fact is attributed to the parallel SIMD architecture of the GPUs (the nVidia GeForce 9800 GX2, for example, has 256 unified stream processors). Because of the GPUs' parallel architecture, they are very good for processing applications that require high arithmetic rates and data bandwidths.

Nvidia and AMD/ATI are implementing unified architectures in their GPUs. Each architecture is associated with it a specific language: Nvidia has developed CUDA (Compute Unified Architecture) [nVidia 2007b] and AMD developed CAL (Compute Abstraction Layer) [AMD 2007]. One main advantage in the use of these languages is that they allow the use of the GPU in a more flexible way (both languages are based on the C language) without some of the traditional shader languages limitations such as "scatter" memory operations, i.e. indexed write array operations, and others that are not even implemented as integer data operands like bit-wise logical operations AND, OR, XOR, NOT and bit-shifts [Owens et al. 2007]. Nevertheless, the disadvantage of these architectures is that

they are only available for the vendors of the software, i.e., CUDA only works on Nvidia and CAL only works on AMD/ATI cards. In order to have GPGPU programs that work on both GPUs it is necessary to implement them in shader languages like GLSL (OpenGL Shading Language), HLSL (High Level Shader Language) or CG (C for Graphics) with all the vertex and pixel shader limitations and idiosyncrasies.

In addition, Intel has recently presented a new architecture for GPUs called Larrabee [Seiler et al. 2008]. It is made up of several x86 processors in parallel which can be used to process both graphics and non-graphics data. The advantage of this architecture is that it does not need a special language, just plain C. Nevertheless, it will only be available in late 2009.

There are many areas that apply GPGPU: wheather forcast [Michalakes and Vachharajani 2008], chemistry [Ufimtsev and MartÃnez 2008] and of course graphics. Games use GPGPU mainly in two areas: Physics and AI.

PhysX [Ageia 2008] and Havok [Intel 2008] are examples of physics engines which have used the GPU to accelerate their physics loop (eight times of speedup in the case of Havok [Green 2007]). Also the book GPU Gems 3 [Nguyen 2007] presents a full section dedicated for physics using the GPGPU. In the field of AI it can be seen implementations of state machines [RudomÃn et al. 2005] and flocking boids [Erra et al. 2004] using the GPU to processes its data.

3 Related works

The game loop can be divided in three general classes of tasks:

- Data acquisition task is responsible for getting user commands from the various input devices;
- Data processing tasks, also referred as the update stage, are responsible for tasks that update the game state, for example: character animation, Physics simulation, Artificial Intelligence, game logic, and network data acquisition;
- Data presentation task is responsible for presenting the results to the user. In games, this corresponds usually to rendering graphics and playing audio.

The main objective of real-time game loop models in the literature is to arrange the execution of those tasks, in order to simulate parallelism. The work by Valente et al [Valente et al. 2005] provides a survey of real-time game loop models hat regards single-player computer games. But it does not cover the use of GPGPU as an update stage of the game loop.

The simplest implementation of a game loop with the GPGPU as an update stage is executing it sequentially, as shown in figure 1. Diverse GPGPU implementations use this game loop, as the CUDA particles demo [nVidia 2007a].

Another game loop with GPGPU presented in the literature is the multithread architecture with GPGPU stage uncoupled from the main loop [Joselli et al. 2008a; Joselli et al. 2008b]. This architecture is composed by two threads. One of them gathers user input, executes rendering, and updates the game state. The other thread runs the GPGPU. Figure 2 illustrates this game loop model.

The multithread uncoupled with a GPGPU stage [Zamith et al. 2007] is the other game loop with GPGPU available on the literature. This game loop consists of three threads: the first deals with gathering user input and updating the game state. The second thread is responsible for rendering the scene. The third one runs the GPGPU. Figure 3 illustrates this game loop.

The literature on task distribution between CPU and GPU is scarce. The work by [Zamith et al. 2007] implements a semi-automatic task scheduling distribution between CPU and GPU via a script file. Joselli et al. [Joselli et al. 2008a; Joselli et al. 2008b] implements some heuristics for automatic task distribution between CPU and GPU, using a Physics engine that has some methods implemented in both CPU and GPU.

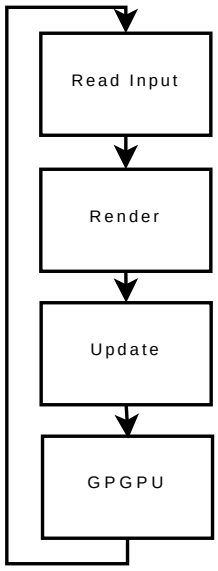


Figure 1: *Single coupled loop*

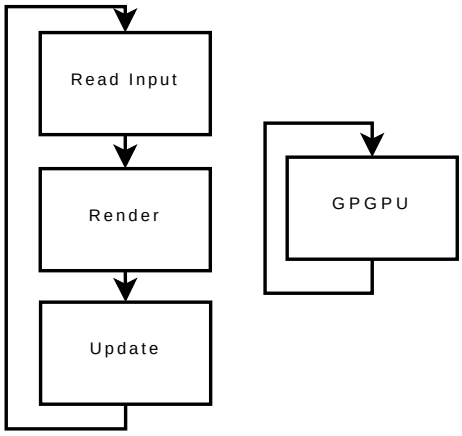


Figure 2: *Multithread architecture with GPGPU stage uncoupled from the main loop*

All of those game loops can be implemented in the adaptative game loop architecture presented in the next section.

4 The Adaptative Game Loop Architecture

This paper presents a new game loop architecture, named Adaptative Game Loop Architecture. This architecture is able to:

- Implement the game loop by using a multi-threads or a single-thread mode;
- Use coupled and uncoupled tasks;
- Use pixel shader or CUDA.

This architecture is based on the concept of tasks. A task corresponds to some work that the application should execute, for example: reading player input, rendering and update application objects.

In the proposed architecture, a task can be anything that the application should work towards processing. However, not all tasks can be processed by all processors. Therefore, the application has three groups of tasks. The first one consists of tasks that can be modeled only for running on the CPU, like reading player input, file handling, and managing other tasks. The second group consists of tasks that can only run in the GPU, like the presentation of the scene. The third group can also be modeled for running on both processors. These tasks are responsible for updating the state of some objects that belongs to the application, like AI and Physics.

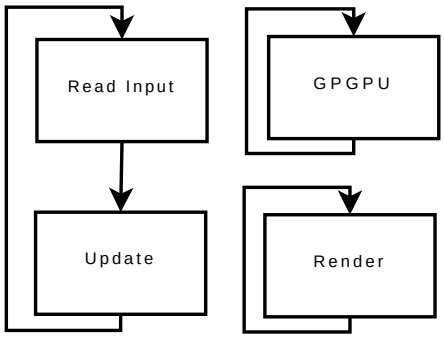


Figure 3: *Multithread uncoupled with an GPGPU stage*

The task concept is modeled as an abstract class that different threads are able to load. Figure 4 illustrates the UML class diagram for the Task and its subclasses.

The Task class is the virtual base class and has four subclasses: Input Task, Update Task, Presentation Task, and Automatic Update Task. The first three are also abstract classes. The latter is a special class whose work is to perform the automatic dynamic distribution between the CPU and the GPU. This distribution consists of choosing the processor that is going to run a task according to some heuristic specified in a script file. Also a special class, the Task Manager class, is responsible for creating and keeping all the tasks of the game loop (discussed in Subsection 4.1).

The Input Task classes and subclasses handle user input related issues. The Update Task classes and subclasses are responsible for updating the game state. The CPU Update class should be used for tasks that run on the CPU, and the GPU Update class corresponds to tasks that run on the GPU. The Presentation Task and subclasses are responsible for presenting information to the user, which can be visual (Render Task) or sonic (Sound Task).

4.1 The Task Manager

The Task Manager (TM) is the core component of the proposed architecture. It is responsible for instancing, managing, synchronizing, and finalizing task threads. Each thread is responsible for tasks that run either on the CPU or on the GPU. In order to configure the execution of the tasks, each task has control variables described as follows:

- THREADID: an id of the thread that the task is going to use. When the TM creates a new thread, it creates a THREADID for the thread and it assigns the same id to every task that executes in that thread;
- UNIQUEID: an unique id of the task. It is used to identify the tasks;
- TASKTYPE: the task type. The following types are available: input, update, presentation, and manage;
- DEPENDENCY: a list of the tasks (ids) that this task depends on to execute.

With that information, the TM creates the task and configures how the task is going to execute. A task manager can also hold another task manager, so to use it to manage some distinct group of tasks. An example of this case is the automatic update tasks that Subsection 4.2 presents.

The Task Manager acts as a server and the tasks act as its clients, as every time a task ends, it sends a message to the Task Manager. The Task manager then checks which task it should execute in the thread.

When the Task Manager uses a multi-thread game loop, it is necessary to apply a parallel programming model to be able to identify the shared and non-shared sections of the application, because they should be treated differently. The independent sections compose tasks that are processed in parallel, like the rendering task. The shared sections, like the update tasks, need to be synchronized in to guarantee mutual-exclusive access to shared data and to preserve task execution ordering.

Although the threads run independently from each other, it is necessary to ensure the execution order of some tasks that have processing dependence. The architecture accomplishes this by using the DEPENDENCY variable list that the Task Manager checks to know the task execution ordering.

The processing dependence of shared objects needs to use a synchronization object, as applications that use many threads do. Multi thread programming is a complex subject, because the tasks in the application run alternately or simultaneously, but not linearly. Hence, synchronization objects are tools for handling task dependence and execution ordering. This measure should also be carefully applied in order to avoid thread starvation and deadlocks. The TM uses semaphores as the synchronization object.

4.2 The Automatic Update Task

The purpose of this class is to define which processor will run the task. The class may change the task's processor during the application execution, which characterizes a dynamic distribution.

One of the major features of this new architecture is to allow dynamic and automatic task allocation between the CPU and GPU, in order to do that it uses the Automatic Update Task class. This task can be configured in order to be executed in three modes: CPU only, GPU only and in the automatic distribution between CPU and GPU. In order to execute on the CPU a CPU implementation must be provided, and in the GPU a GPU implementation must be provided, and in order to make use of the automatic distribution both implementation must be provided. The scheduling is by heuristic in a script file. Also a configuration on how the heuristic is going to behave is needed, and for that a script configuration file is presented in Subsection 4.2.1. The scripts files are implemented in Lua [Ierusalimschy et al. 2006] (Subsection 4.2.2).

The Automatic Update Task acts like a server and its tasks as clients. The role of the automatic update task is to execute a heuristic to automatic determine in which processor the task will be executed. The Automatic update task executes the heuristic and determines which client will execute the next task and will send a message to the chosen client, allowing it to execute. Also, every time the clients finish a task they send a message to the server to let it know it has finished. Figure 5 illustrate this process.

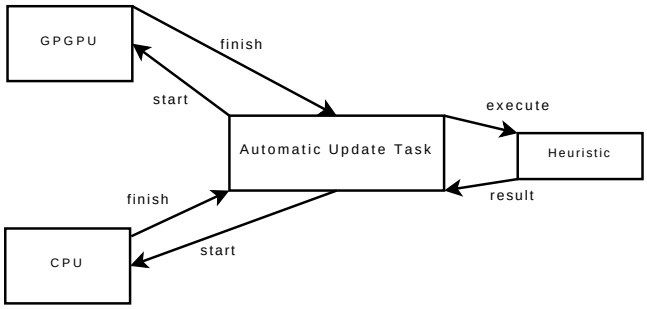


Figure 5: *The Automatic Update Task class and messages*

4.2.1 The Configuration Script

The configuration script is used in order to configure how the automatic update task will execute the heuristic.This script defines four variables:

- INITFRAMES: used in order to set how many frames are used by the heuristic to do the initial tests. These initial tests are used in order that the user may want that the heuristic make the initial tests different than the normal test;
- DISCARDFRAME: used in order to discards the first DISCARDFRAME frame results, because the main thread can be loading images or models and it can affect the tests;

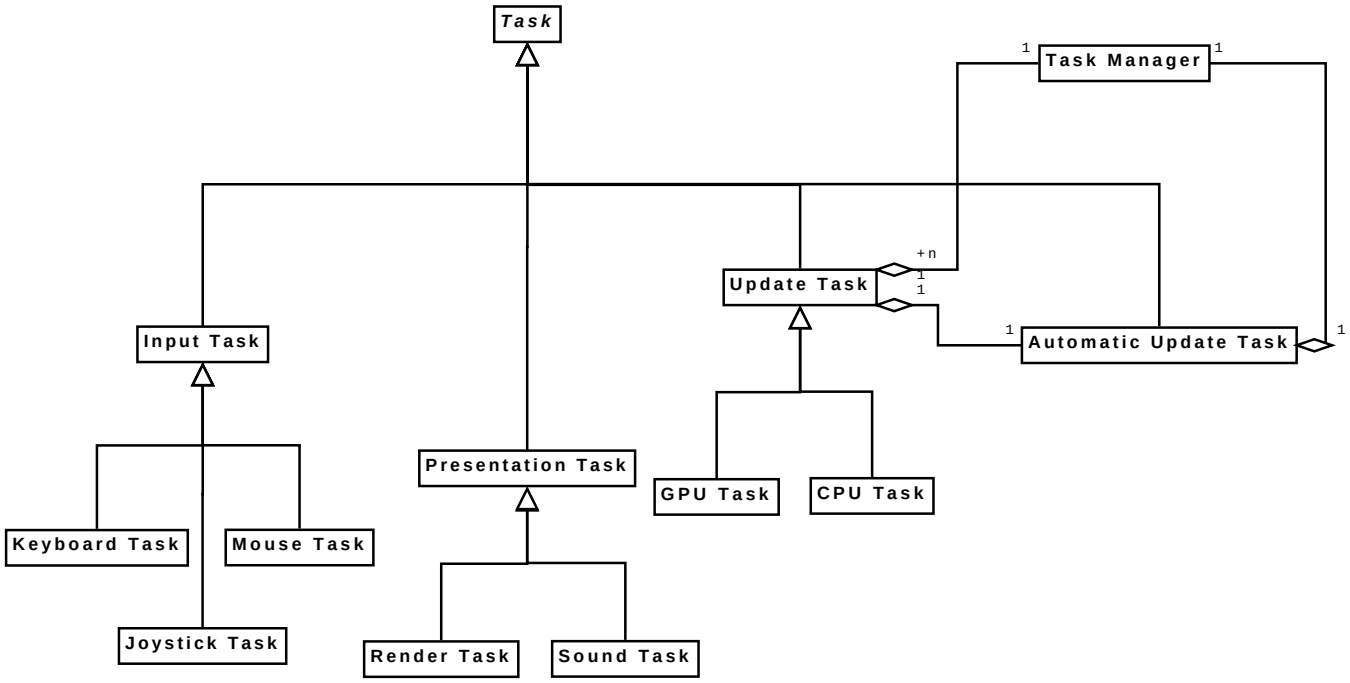


Figure 4: *Multithread uncoupled with an GPGPU stage*

- LOOPFRAMES: it is used to set on how frequency the heuristic will be executed. If this value is set to -1 the heuristic will be executed only once;
- EXECUTEFRAMES: it is used to set how many frames are needed before the decision on changing the processor will execute the next tasks.

An example of the configuration scrip file can be seen in script 1.

Script 1 Configuration Script

```
INITFRAMES      = 20
DISCARDFRAME    = 5
LOOPFRAMES      = 50
EXECUTEFRAMES   = 5
```

So the automatic update task begins executing after the DISCARDFRAME are executed, then it execute INITFRAMES frames in the CPU and the next INITFRAMES in the GPU then it decides where the next LOOPFRAMES frames will be executed. If the LOOPFRAMES is greater then -1, it executes EXECUTEFRAMES frames in the CPU and it executes EXECUTEFRAMES frames in the GPU then it decides where the next LOOPFRAMES frames will be executed and keep repeating until the application is aborted.

4.2.2 The Heuristic Script

The heuristic script is used in order to distribute automatically the tasks between the CPU and the GPU. This script defines three functions:

- reset(): reset all the variables that the script uses in order to decide which processor will execute the task. This function is called after the LOOPFRAMES frames are executed. The variable that are normally used by the heuristic are:
  - CPUTime: this is the sum of all the elapsed times that the task has been processed in the CPU;
  - GPUTime: this is the sum of all the elapsed times that the task has been processed in the GPU;
  - bestCPUFPS: the best frame rate achieved by the CPU;
  - bestGPUFPS: the best frame rate achieved by the GPU;
- setVariable(elapsedTime, processor): in this function is where all the variables that are used by the heuristic are set. This function is called after the EXECUTEFRAMES frames in each processor. This function can be seen on script 2.

Script 2 setVariable Script

```
function setVariable(elapsedTime,  processor)
   FPS = 1  / elapsedTime
   if (processor == CPU) then
     CPUtime = CPUtime + elapsedTime
     if (FPS > bestCPUFPS) then
        bestCPUFPS = FPS
   end
     else
      GPUtime = GPUtime + elapsedTime
      if (FPS > bestGPUFPS) then
        bestGPUFPS = FPS
      end
   end
end
```

- main (): This is the function that executes the heuristic and decides which processor will execute the task. This function is called just before the LOOPFRAMES frames are executed. A script with this function implemented with the decision of always executing in the GPU can be seen on script 3.

Script 3 Main Script

```
function main()
   return GPU;
end
```

5 Test Case

The test case corresponds to the n-bodies sample [Nyland et al. 2007] from the GPU Gems 3 [Nguyen 2007]. The authors had implemented this example only to validate the model of game loop proposed in this work, because this problem works with intense mathematics processing.

The n-bodies demo is an approximation of the evolution of a system of bodies in which every body interacts with every other body. It also applies to different simulations like protein folding, turbulent fluids, global illumination and astrophysics. In this case the n-bodies is a simulation of astrophysics in which each body repre-

sents a galaxy or an individual star, and each body attracts or repeals each other with gravitational forces.

This sample was implemented in both CPU and GPU. The GPU version uses CUDA. It is important to remark that even thought the demo uses CUDA, the game loop implementation could use CAL or shader languages (GLSL, HLSL or CG) without major modifications in the framework layer. Figure 6 illustrates a set of frames from the simulation.

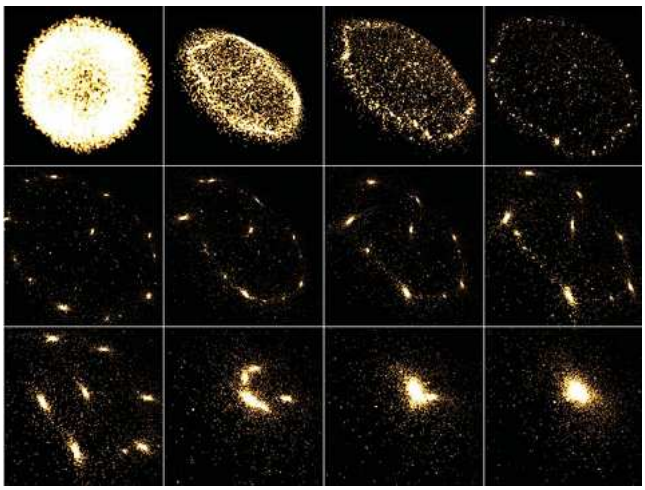


Figure 6: *N-bodies sample*

The authors did not emphasize the n-bodies problem, because it is not the aim of this work. The n-bodies is used such as example only for validate the method proposed.

5.1 The Tested Heuristic

The demo uses a very simple heuristic. It checks which processor was the fastest one in running the task and then selects this processor to run the next frames. Script 4 lists the heuristic.

Script 4 Tested Heuristic

```
function main()
   if (CPUtime < GPUtime) then
      mode = CPU;
   else
      mode = GPU;
   end
end
```

The heuristic is configured in script, and any kind of heuristics can be implemented, but the heuristics developed by the authors can work in two different ways:

- The first, that is called initial, is configured in order to execute 20 frames in the CPU and 20 frames in the GPU and then it decides for the fastest processor.
- The second, that is called looped, is configured to loop in the following state: execute 5 frames in the CPU and 5 frames in the GPU and decide to for the fasted processor to execute the next 200 frames.

5.2 The Tested Game Loop

To test the architecture, the demo implemented a game loop with an input task and a render task in the main loop, and an automatic update task with CPU/GPU in another thread (uncoupled). Figure 7 illustrates this game loop.

6 Results

The tests were done base on the fast n-bodies simulation with CUDA, such as described before. There are two groups of tests.

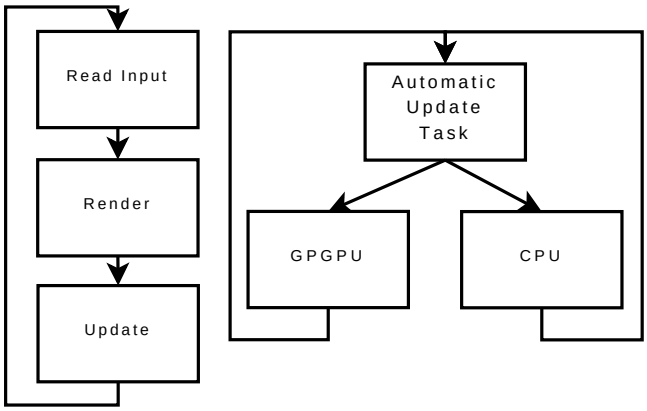


Figure 7: *The Multi thread loop with an automatic update task uncoupled from the main loop.*

The first group uses the initial heuristic where the fastest processor is selected at the beginning of the task execution. The second group uses the other heuristic (looped), that is, the heuristic is invoked for each cycle of the frames. The CPU tests were made with an Intel quad-core 2.4 GHz and the GPU tests were made with three different GPUs a nVidia Geforce 8800 GTS, a nVidia Geforce 8400 GS and a nVidia 8200M G.

For both groups, the example was executing the application and the work of heuristic to choice the processor. Table 1 illustrate the performance of the application and the processing for both processors. The initial number of bodies is 4 and it is increased until 8192 bodies. Figure 8 shows a comparison between the CPU and the nVidia 8800 GTS GPU when there are few number of bodies.

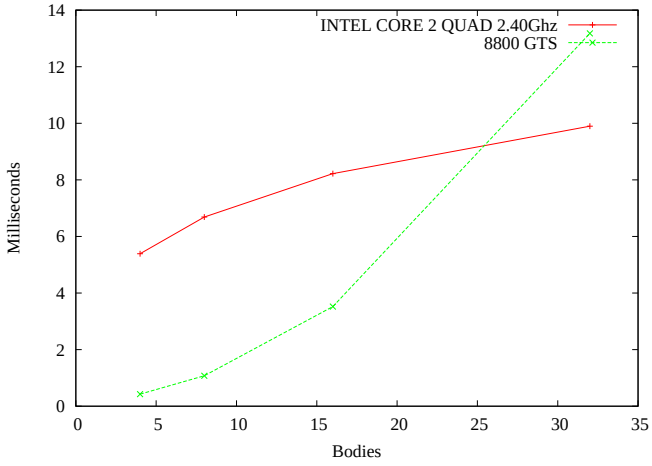


Figure 8: *Comparison between the CPU and the GPU*

Although the GPU is faster than CPU, until 25 bodies, approximately, the CPU is better and the heuristic choices it, after the GPU is already chosen. So, the CPU is faster with less bodies and the GPU, in this example, is more efficient with higher number of bodies. Figure 9 shows the comparison between the evolution of the tested GPUs.

7 Conclusion

Multicore hardware architectures are a tendency, and both the CPU and the GPU have developed great evolution in this aspect. Quadcode processors are present in the latest CPU architecture and unified architecture is presented in the latest GPUs. This tendency is not only in increasing the processing power but also increasing the number of cores available. With that the parallel processing, in this architectures, is a reality. With this hardware evolution the games will get much more sophisticated and multicore game loops with the use of the GPU will get more common.

Table 1: *Elapsed time of processors in 100 iterations measured in milliseconds*

bodies	CPU elapsed time	8800 GTS elapsed time	8400 GS elapsed time	8200M G elapsed time	processor
4	*0.404*	7.792	4.251	4.666	CPU
8	*1.010*	9.170	4.417	5.105	CPU
16	*3.279*	9.265	4.756	5.116	CPU
32	12.243	*10.600*	*5.700*	*10.343*	GPU
64	48.001	*11.250*	*10.931*	*26.496*	GPU
128	190.745	*15.182*	*30.658*	*89.628*	GPU
256	773.152	*29.244*	*107.664*	*318.036*	GPU
512	3124.517	*75.188*	*410.865*	*1186.663*	GPU
1024	12155.210	*282.648*	*1619.403*	*4584.704*	GPU
2048	48627.184	*989.581*	*6526.119*	*18097.682*	GPU
4096	195216.563	*3835.552*	*25815.580*	*71998.977*	GPU

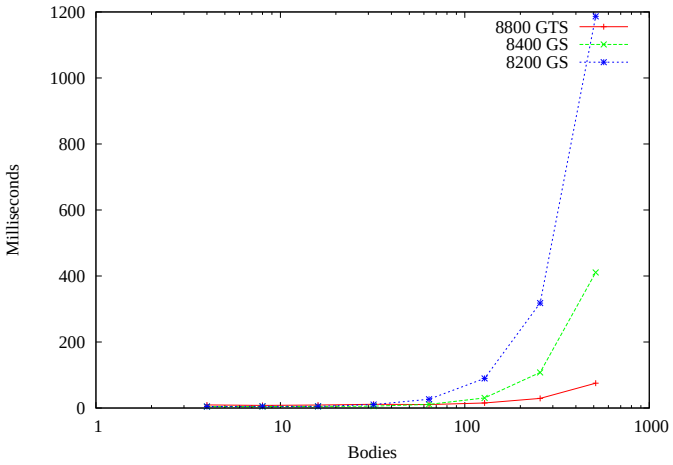


Figure 9: *Comparison between the GPUs*

References

AGEIA, 2008. Physx. Avalible at: `http://www.ageia.com`. 20/02/2008.

AMD, 2007. Amd stream computing. Available at: `http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf`. 20/02/2008.

ERRA, U., CHIARA, R. D., SCARANO, V., AND TATAFIORE, M. 2004. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. *Vision, Modeling, and Visualization*, 233–240.

GREEN, S., 2007. Gpgpu physics. Siggraph07 GPGPU Tutorial.

HOFSTEE, H. P. 2005. Power efficient processor architecture and the cell processor. *IEEE Proceedings of the 11th International Symposium on High-Performance Architecture*.

IERUSALIMSCHY, R., DE FIGUEIREDO, L. H., AND CELES, W. 2006. *Lua 5.1 Reference Manual*. Lua.org.

INTEL, 2008. Havok. Avalible at: `http://www.havok.com`. 20/02/2008.

JOSELLI, M., ZAMITH, M., VALENTE, L., CLUA, E. W. G., MONTENEGRO, A., CONCI, A., FEIJÓ, B., DORNELLAS, M., LEAL, R., AND POZZER, C. 2008. Automatic dynamic task distribution between cpu and gpu for real-time systems. *IEEE Proceedings of the 11th International Conference on Computational Science and Engineering*, 48–55.

JOSELLI, M., CLUA, E., MONTENEGRO, A., CONCI, A., AND PAGLIOSA, P. 2008. A new physics engine with automatic process distribution between cpu-gpu. *Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, 149–156.

LUA. The programming language lua. DisponÃvel em: `http://www.lua.org/manual/`. 20/12/2007.

MICHALAKES, J., AND VACHHARAJANI, M. 2008. Gpu acceleration of numerical weather prediction. *IEEE International Symposium on Parallel and Distributed Processing*, 1–7.

NGUYEN, H. 2007. *GPU Gems 3 - Programming Techniques for High-performance Graphics and General-Purpose Computation*. Addison-Wesley.

NVIDIA. 2006. Geforce 8800 gpu architecture overview. tb-02787-001_v0.9. Technical report, NVIDIA.

NVIDIA, 2007. Cuda particles. Avalible at: `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/particles/doc/particles.pdf`. 20/02/2008.

NVIDIA, 2007. Nvidia cuda compute unified device architecture documentation version 1.1. Avalible at: `http://developer.nvidia.com/object/cuda.html`. 20/12/2007.

NVIDIA. 2008. Nvidia - cuda compute unified device architecture. Programming guide, NVIDIA.

NYLAND, L., HARRIS, M., AND PRINS, J. 2007. Fast n-body simulation with cuda. *GPU Gems 3 Chapter 31*, 677–695.

OWENS, J. D., LEUBKE, D., GOVINDARAJU, N., HARRIS, M., KRÃGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26(1)*, 80–113.

RUDOMÃN, T., MILLÃN, E., AND HERNÃNDEZ, B. 2005. Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory 13(8)*, 741–751.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3.

UFIMTSEV, I. S., AND MARTÃNEZ, T. J. 2008. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. *Journal Chemistry Theory Computation 4 (2)*, 222 – 231.

VALENTE, L., CONCI, A., AND FEIJÓ, B. 2005. Real time game loop models for single-player computer games. In *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, 89–99.

ZAMITH, M., CLUA, E., PAGLIOSA, P., CONCI, A., MONTENEGRO, A., AND VALENTE, L. 2007. The gpu used as a math co-processor in real time applications. *Proceedings of the VI Brazilian Symposium on Computer Games and Digital Entertainment*, 37–43.

Using Game Engines in Digital Manufacturing through Immersive and Collaborative Visualization Systems

Silvia da Costa Botelho
Nelson Duarte Filho
Jonata Tyska Carvalho
Pedro de Botelho Marcos
Renan de Queiroz Maffei
Rodrigo Remor Oliveira
Rodrigo Ruas Oliveira
Vinicius Alves Hax
Universidade Federal do Rio Grande

Abstract

This paper proposes a methodology for immersive multiprojection visualization of manufacturing processes. It admits sceneries with dynamic components and allows collaborative interaction among geographically distributed users. The proposal has several stages to convert the complex industrial project into a model suitable for visualization and interaction. The Quake III Arena engine is used as rendering core system, which is a free and open-source software. The method can be applied to CAD projects, models and simulations used in industry. The proposed ideas are then validated through the study of a real case associated with Shipbuilding and Offshore Industries.

Keywords:: Digital Manufacturing, Shipbuilding, Virtual Reality, CAVEs, Game Engines

Author's Contact:

{silviacb, dmtnldf}@furg.br
{jonatatysca, pedrobmarcos, rqmaffei,
rodrigo.remor, rodropel, viniciushax}@gmail.com

1 INTRODUCTION

Digital manufacturing is an initiative to define every aspect of the design-to-manufacture process digitally. An open data management platform is the ideal basis of Digital Manufacturing. This platform can support multiple disciplines, including product design, analysis, manufacturing, data sharing and communication, etc. It relies on advanced technologies such as CAD, CAE, real time 3D simulations, CAM, PDM, CAPP, etc.

Large and high-end engineering projects such as automotive, aerospace and shipbuilding are already engineered almost entirely digital. Their complexity demand assembly simulations using mockups to test the several stages of the process. As physical mockups have a very complex and high cost design, digital 3D mockups are good solutions. In practice however, this digital design is not as simple as it might first seem. In such a process a large number of concurrently working design teams are involved, resulting in the development of typically thousands of different parts, each modeled with the highest possible accuracy. For instance, in the case of the Boeing 777 airplane project, more than 230 geographically dispersed groups had to be coordinated and more than 350 millions of individual polygons from the graphical model of CAD datasets must be processed in order to obtain a 3D projection of the whole model.

Eminent problems in large-scale manufacturing arise from the potential overlaps of assembly parts and the difficulty in properly fitting all individual components together in a final product. In those contexts, Virtual Reality (VR) techniques can provide an interactive high-quality visualization solution for evaluation of the full CAD design and the whole manufacturing process [Dietrich et al. 2005].

Virtual Reality techniques make possible 3D visualization of industrial sceneries, offering a realistic and interactive interface for different production stages and processes, aiming to anticipate problems like overlaps of assembly parts, obstacles collisions, risky situations, etc. These techniques include several levels of interactivity and immersive technologies.

CAVE devices [Cruz-Neira et al. 1992], for instance, are room-size VR display systems. In these systems the generated images are distributed among computers and multiprojected onto several walls. Each individual projection contains a part of the total scenery, obtained from a process of simultaneous rendering. This way, each wall acts as a window to the virtual world. The use of such configuration makes possible, among others, to increase the immersion and interactivity degrees associated with the sceneries. Some semi-immersive desktop-based VR systems exist but an obvious limitation of them is that designers are susceptible to environmental disturbances, diminishing their true feeling of the scene. As a result, they cannot be fully immersed in the virtual environment to study and improve the product design efficiently [Choi and Cheung 2006].

If in the past decades the benefits of VR and its integration with digital manufacturing systems were focused in the static product design (visualization of the CAD project of the product), nowadays the state-of-art is to focus on the modeling and simulation of dynamical manufacturing plants as a whole process. The life-cycle of the product, the ergonomic of the workers, the localization and performance of the machines, the assembly steps have became to be simulated and visualized in immersive CAVEs.

Besides, such VR techniques and technologies, associated with recent efficient simulation and communication systems, have become a new paradigm for integrating different levels of production processes that are temporal and geographically distributed. The large engineering industries have physical parts produced by separated key players, each with their own sets of standards and terminology. So, one of the biggest obstacles in the widespread implementation of digital manufacturing is the weak interoperability and collaboration between various systems in product design, manufacturing engineering, and production floor departments. Breaking down the walls between these departments and maintaining the crucial digital continuity of the product life-cycle will greatly foster digital manufacturing, and therefore help manufacturers improve efficiency of processes and quality of products [Wald et al. 2005].

Thus, due to the distributed nature of recent high-end engineering projects, the possibility of virtual, interactive and collaborative immersive visualization of dynamic manufacturing plants have become an important issue which can determine a company/consortia success [Stephens et al. 2006] [Bigler et al. 2006]. The use of this new paradigm can reduce cost, complexity and time associated with the process. For instance, large industrial conglomerates such as automobile (Volkswagen, Ford and General Motors), aerospace (Airbus, Embraer) and the shipbuilding/offshore industries are integrating VR concepts in their manufacturing processes [Kreitler 1995].

There are some attempts to create complete computational models of manufacturing plants, in what is considered the new generation of digital manufacturing tools: the Digital Mock-Up (DMU) systems. QUEST/DELMIA [Systemes 2008], PROMODEL [Benson 1996], ARENA [Hammann and Markovitch 1995] are typical ex-

amples of these systems. They allow to design the 3D sceneries of industrial plants but they cannot cope with the collaborative issues and complexity of dynamic immersive visualization of the components operations.

In the literature, there are related works that treat the immersive visualization issues in Digital Manufacturing using Ray Tracing systems [Bigler et al. 2006]. However they process only static sceneries and at high computational cost. For instance, [Dietrich et al. 2005] proposes a visualization system using an Altix NUMAflex architecture that provides a low-latency, high-bandwidth inter-connect between the distinct nodes, gaining peak transfer rates of up to 6.4 GByte per second. [Wald et al. 2005] treats complex CAD design also using a very hard computational infrastructure.

In this work we propose a low cost solution for immersive collaborative visualization of manufacturing plants. The proposal suggests integration of different successful tools produced for specific areas of computer graphics: engines for interactive games, computer aided design packages, graphic editors, etc. Our proposal treats sceneries with dynamic components and allows collaborative interaction among geographically distributed players. Such methodology was implemented and validated through a case study associated with shipbuilding and offshore industries.

The paper is structured in five sections. Next section addresses the main challenges associated with immersive visualization of industrial processes as well as a group of existing solutions for its individual treatment. Section 3 details our integrated framework to the 3D manufacturing plant visualization, through aspects of modeling, conversion, visualization and interactivity stages. Then section 4 describes a group of experiments that were accomplished to validate the proposal. Finally, section 5 presents the conclusions drawn from the entire project.

2 VISUALIZATION OF PLANTS

The visualization process of dynamic manufacturing plants using VR resources includes a group of challenges and issues. From the conception of the virtual plant model and its simulation, to its actual use in collaborative immersive visualization, we enumerate the following main stages: i. Modeling; ii. Conversion (simplification); iii. Visualization; and iv. Interactivity.

Modeling: Such stage refers to the 3D plant digital model creation, through modeling all production process dynamics (workshops, workers, tools, equipments and their interaction). The following aspects are involved:

- Study and development of modeling techniques and simulation of production processes: such subject is target of study in different engineering areas. Plants simulation can be implemented through the use of commercial DMU tools that enable different visual quality levels. For instance, the developed models can be visualized as simple 2D structures or like complex dynamic sceneries with three-dimensional features and interactivity, in CAVE devices. The manufacturing plants models can incorporate very detailed CAD files, as result of mechanical projects of the processed products, as well as schemes of machines functioning and human behavior simulation (workers) of the assembly lines. The generated simulations can show 3D dynamics of the different process components;
- 3D Description Format: supposing that the immersive scenery will be obtained from DMU/CAD tools, many commercial DMU systems just allow the exportation of 3D sceneries in simple video formats as avi and mpeg. Others systems export in virtual environment description languages as VRML (Virtual Reality Modeling Language) [VRML 2008], however, these exported files are frequently very complex and non-optimized. So, due to the extensive set of exportation formats released by the different tools, and the peculiar needs for VR, the best options to provide an immersive distributed visualization of plants is still a challenge.

Conversion (simplification): Such stage includes issues related to simplification of the 3D models generated in the modeling stage. Basically the following points must be treated:

- Definition of the attributes of the virtual model and the entities to be supplied to visualization API. Definition of virtual sceneries: to have a virtual 3D scenery, the geometric CAD design needs to be completed with information describing the appearance of objects (color, reflection characteristics, textures), the lighting environment, possible animations, interactions, sound, as well as behavior and functionality. Such definition can be done offline or applied during real time visualization;
- Reduction of the virtual model complexity. This issue is relevant in almost all VR applications, however it assumes larger importance on Digital Manufacturing. In this context, the DMU must be committed with the necessaries VR optimizations. The variety of formats and the large number of geometric structures can make prohibitive the rendering scenery (as mentioned in section 1, in the case of the Boeing 777 airplane program, where more than 350 millions of individual polygons must be processed).

Visualization: Aiming a more realistic immersion, it is a good choice to adopt an engaging n walls multiprojection. Traditional techniques for multiprojection treatment make use of specialized systems that need complex hardware architectures. Some specific questions in multiprojection visualization should be studied:

- Keeping consistence of the projected entities states, their attributes and dynamic behaviors among the projections on the n walls;
- Synchronizing virtual clocks (logical timers) of the different physical system components, to aim the coherence of the dynamic sceneries visualization and its relationship with the displayed frame rate;
- Increasing the system performance related with the number of entities versus the number of projection walls.

Interactivity: The visualization processes should support different interactivity degrees. For instance:

- In manufacturing plants sceneries, the collisions treatment, the use of different devices for sensory perception of environment features (mouses, keyboards, joysticks, glasses, gloves, trackers, etc) are important factors that should be foreseen;
- A method of enabling the plurality of geographically distributed users to collaboratively view and interact have to be offered. Due to a large number of key players associated with modern manufacturing processes, methods and apparatus for virtual interactive sceneries, by multiple remotely-located users, are necessary.

Nowadays, there aren't any DMU tool able to treat all the stages and aspects mentioned above, nor a methodology that implement the modeling, conversion, etc, supplying the dynamic characteristics of the industrial processes to a collaborative visual system. However, many are the studies and techniques associated with each individual subject (modeling, conversion, etc). Several tools allow the design and exportation of static CAD plants models in formats that make possible the use of specialized systems to conversion CAD-VR. WalkInside [VRContext 2008] is one of these systems. There are also visual immersive systems developed for CAD models visualization as ENVIRON [Corseuil et al. 2004]. This system, when associated with VR tools, allows the CAD models visualization in VR environments.

Another possibility related to virtual environments is the use of game engines. There are game-development toolkits that provide means to create custom content and mission scenarios which facilitate idea generation and concept exploration with a short turnaround time [Fong 2006]. Some examples of these systems are Unreal Tournament [EpicGames 2004] and Quake [Quake 1997]. The former have partial open code and the last is a free and open-source software. Both provide graphics with high detail levels and

have high performance and robustness, supporting distributed geographic visualization. A modification of the Quake III engine together with the FreeVR [Sherman 2008] library have been used in the case study presented in section 4. Among the advantages of using the Quake III engine and the FreeVR library is they open-source licenses multi-platform capabilities.

3 A METHODOLOGY FOR IMMERSIVE VISUALIZATION

Based on the aspects mentioned on section 2, a methodology that enables a collaborative visualization of industrial processes was built. It takes, as input, 3D models obtained from commercial DMU tools, more precisely in applications evolving high complexity design and manufacture, such as the case for ships and offshore platforms construction [Kim et al. 2003]. Figure 1 shows an overview of our proposal.

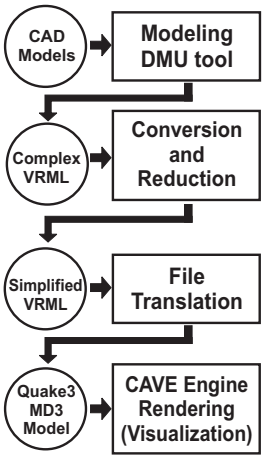


Figure 1: *Methodology for collaborative visualization of industrial processes*

Starting from CAD models, techniques of complexity reduction and conversion must be applied to obtain virtual sceneries, with their necessaries components and attributes. It is proposed to visualize such sceneries in a multiprojection immersive environment, through an adaptation of the game engine Quake III Arena. Such engine can provide the necessary structure for attributes definition, rendering and multiprojection, as well as the needed interactive resources. The details of each one of the stages shown on the figure is described as follows.

Modeling: It is supposed that some DMU tool is being used to model the manufacturing process. Such tool may be able to export different CAD formats with animated entities, aiming to study the life-cycle, ergonomic aspects, among others, of the modeled objects.

Conversion: A dynamic model of the plant is obtained from the complete CAD project. This model can be described in a file, using different formats: DWG, VRML, ASE, DGN, etc. Due to the lack of commitment between the DMU tools and the visualization stage, such file have frequently excessive geometric information. Thus, it is needed a meshes complexity reduction tool to compose each element with the just necessary information. After trials with several alternative tools, like DeepExploration [Exploration 2008], 3DStudioMax [3DStudioMax 2008], OkinoPoliTrans [PolyTrans 2008] and VizUp [VizUp 2008], VizUp and 3DStudioMax was chosen.

The VizUp system can be used to reduce the number of existing polygons, in VRML files format, according to the wished visualization quality, in a manual way. The VRML file needs to be loaded in VizUp System and the user must choose an acceptable maximum rate of reduction of the VRML file that avoids the lack of reality of the model. This rate depends of each model.

After reduction, this VRML file needs an optimization and compatibilization with 3DStudioMax. DMU tools generate some no necessary informations for the immersive visualization system that need to be removed. It is important to emphasize that the 3DStudioMax just supports animated VRML in Interpolated mode. If VRML file is in Event mode a conversion must be done. Another point is that the VRML file generated by DMU treats each face of one object like a unique object. For example, the six faces of a cube are six different objects instead one unique object with six faces. Then, it is necessary to aggregate this faces in one unique object to optimize the VRML file.

Translation: Once the VRML is compatible and optimized, we need to set up the textures. The format used by Quake III Arena to describe objects and animations is the MD3. This format does not support color in objects, only textures, so it is necessary to recognize the colors of each object in the VRML file, create an image (bitmap) for each color and apply the texture in each object with their respective colors. In this way, VRML file has textures with the same colors of the objects, what does not change anything in his appearence, just makes possible a correct conversion to MD3 in the next step.

In the last step of conversion, 3DStudioMax is used to import and convert the already treated VRML file generated by DMU into a MD3 file. In this point the model can be simulated in the CAVE Quake. This conversion may take several minutes depending on the size and number of frames on the model.

Visualization: The stage of visualization and interactivity is accomplished by a game engine. We have used the Quake III game engine [Rajlich 2008] to solve the rendering, multiprojection and interactivity aspects. It is composed by some part of the game core, multiprojection code from FreeVR [Sherman 2008] and GtkRadiant [GtkRadiant 2008] sceneries editor, as shown at figure 2.

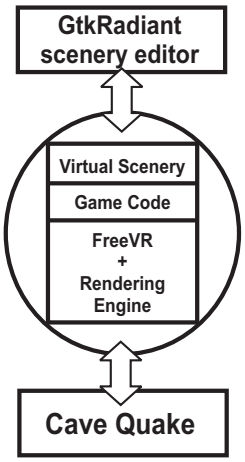


Figure 2: *Modular structure of a Game Engine*

The Quake Game Engine represents the virtual scenery using binary space partitioning (also known as BSP tree) structure for the surrounding ambient and MD3 structure for representing the models within this ambient. Each model may have its general attributes, which are passed as flags inside the MD3 structure, and its particular attributes, which are passed in each reference to MD3 structure, in the BSP. This allows easy customization. All the game rendering process is done using the cross-plataform graphical API OpenGL. The engine chooses what should be drawn, by looking in the BSP structure, and sends it to OpenGL. The collision detection is also done by the game engine.

The multiprojection visualization code is all done by FreeVR, an open-source virtual reality interface/integration library. It has been designed to work with a wide variety of input and output hardware, with many device interfaces such as gloves and head-mounted display (HMD). The library allows n-walls multiprojection and the adjustment of each projection wall, in accordance with the CAVE

morphology. The necessary modifications to create projections out of the axis are read as parameters from a configuration file.

GtkRadiant is the most used Quake scenery editor. It allows to apply and modify the outputs of Quake, such as illumination, texture and interactivity of components/objects. This editor can import files in some formats such as static models ASE and MAP and animated models in MD3.

4 CASE STUDY

The proposed methodology was validated in a case study on Shipbuilding and Offshore Industry, as described below.

As tipical large industrial conglomerates, the Shipbuilding and Offshore Industries use DMU technologies. DELMIA, CATIA, ENOVIA, QUEST and PROCESS ENGINEER are important toolsets for Product Life-cycle Management (PLM). For instance, DELMIA [Systemes 2008] is widely used in aerospace and shipbuilding industries and allows to optimize the factory layout; to determine and validate assembly sequences and ergonomics aspects; and to make possible global analysis and 3D simulation. QUEST presents a large set of resourses that includes analysis and simulation of resourses and process flow; layout analysis; etc. However, the 3D sceneries created by all these systems cannot be directly visualized in a multiprojection way.

Thus, starting from CAD plants of ships and plataforms, as well as from digital model of a shipyard, in this case generated on DELMIA and QUEST [Systemes 2008], bellow we show the performance of the engineering solution proposed.

Adopting the stages presented in section 3, it is pointed out the details used in the case study.

Modeling Stage. In the modeling stage, the different CAD projects were integrated in a 3D virtual scenery. We have tested three different complexities sceneries, called $Model_1$, $Model_2$ and $Model_3$, with the characteristics showed in table 1.

Model	Vertexes	Triangles	Storage (Kb)
$Model_1$	81,720	53,796	11,033
$Model_2$	62,372	55,794	6,958
$Model_3$	76,972	61,266	21,991

Table 1: *Complexity of the models*

Conversion Stage. Starting from the VRML exported sceneries it is achieved the models complexity reduction, in order to reach a good visual quality and to control the number of polygons. For this, as above, it is used Vizup. Different percentages of reduction were applied. Table 2 shows the reduced models of $Model_1$.

With a reduction rate of up to 50 percent it was still possible to identify the model, but with loss of details. The ideal reduction rate for $Model_1$ was around 44 percent.

Reduction (%)	Vertexes	Triangles	Storage (Kb)
50	60,202	26,898	10,254
48	61,277	27,973	10,305
46	62,349	29,049	10,354
44	63,425	30,125	10,402

Table 2: *Complexity of the reduced models of* $Model_1$

The $Model_2$ and the $Model_3$ had presented the best parameters shown in table 3, after apropriated reduction.

From the tables we can conclude that the reduction is really efficient for complex models with large amount of information. In models where the complexity is smaller it is not possible to apply high reduction rates. Due to simplicity already existing in small models, high reduction rates produce loss of quality at visualization process.

Once the models are converted into a reduced VRML file, some of their components attributes are manipulated through 3DStudioMax. The animated components are separated from the static components and the texture characteristics should be individually observed in order to verify the quality acquired from CAD models, and then possibly improved.

Visualization Stage. After modeling and conversion, the obtained scenery can be visualized in a multiprojection way. The methodology proposed allows projections among n screens. It was used a V-CAVE with two walls (see figure 3) to validate the proposal. Also different sceneries description formats, other than VRML, may be used as input of the system, reassuring its portability. The multiprojection system implemented by CaveQuake was shown efficient and simple. Just some adjustments must be done in the configuration parameters so that the correct projections in a V-CAVE can be obtained. Rendering rates were measured with the three models and all of they were around 100 fps.

In order to verify scalability of the proposal, the rendering rates were tested with four walls of projection. One more graphic card with two outputs was used and the rates of rendering remained at the same level.

Testing related to geographic distribution have been released, considering it is a feature of Quake duly established.

Some other issues were tested. For example, the CaveQuake obstacles treatment, that was showed adequate.

A picture of the V-CAVE used for the experiments can be observed in the figure 4.

Model	Reduction(%)	Vertexes	Triangles	Storage (Kb)
$Model_2$	49	46,739	28,461	6,197
$Model_3$	58	53,882	25,731	20,954

Table 3: *Complexity of the reduced models of* $Model_2$ *and* $Model_3$

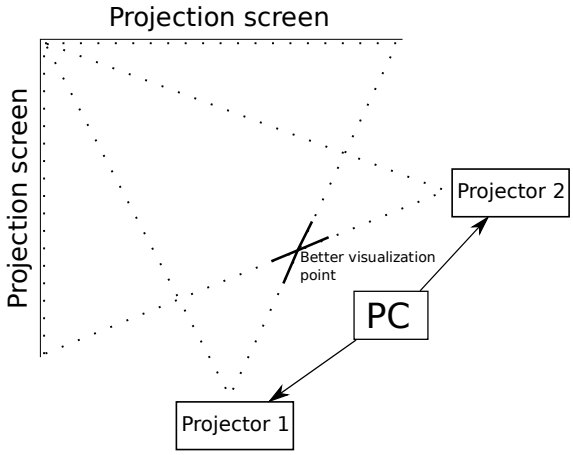


Figure 3: *CAVE Layout (V-CAVE)*

5 CONCLUSION

Digital manufacturing technologies are being used in many industrial production process. It is specially adopted on modeling and simulation of the manufacturing processes in large industrial conglomerates. Although the visualization of CAD projects is already a practice in these contexts, the use of advanced 3D immersive interfaces using VR resources is still a challenge.

In this work we have proposed a methodology to support collaborative multiprojection visualization of manufacturing processes.

It was proposed to start with dynamic plants obtained from scenery models supplied by DMU tools. After identifying the limitations, restrictions and needs associated with the visualization problem, a group of procedures that enables the projections in n walls, with immersive features and VR resources, integrating different existing tools, was proposed.

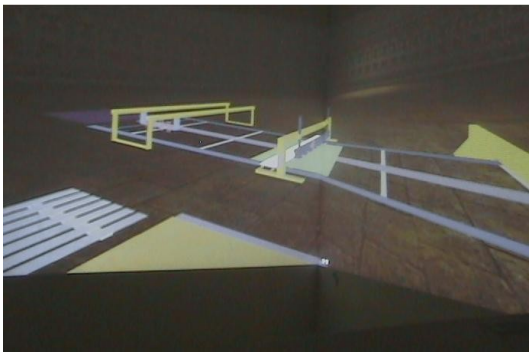


Figure 4: *Multiprojection visualization of scenery in V-CAVE.*

The engineering solution was validated in an actual application associated with the naval and offshore industry. Sceneries related to the plants of shipyards, ships and platforms were modeled, converted and visualized in centralized and collaborative environments.

The accomplished conversions and the use of Quake III game engine were proved efficient, concerning the scalability, heterogeneity, portability, rendering and multiprojection issues.

The complexity reduction along the conversions allowed the use of VR resources in real time visualization.

The proposed methodology can be applied directly in the plant by naval technicians with no need for specialized programming knowledge.

The solution using Quake III game engine is a significant low cost implementation solution.

As future perspectives, it is necessary to improve and expand the automation level associated with some stages of the methodology, such as the convertion and translation stages.

As a future work, it is necessary to obtain solutions for supporting a better interoperability between models (ontology), perhaps in phase with STEP standard [ISO10303 1994].

Finally, it must be improved some interactivity aspects as inclusion of mixed reality resources to Quake, enabling the visualization of either workers or machines, in real or virtual way, and their interaction.

ACKNOWLEDGMENTS

This work is sponsored by Petrobras and FINEP.

References

3DSTUDIOMAX, 2008. http://usa.autodesk.com/adsk/.

BENSON, D. 1996. Simulation modeling and optimization using promodel. In *Proceedings of the 28th conference on Winter simulation*, IEEE Computer Society, Washington, DC, USA, 447–452.

BIERBAUM, A., JUST, C., HARTLING, P., MEINERT, K., BAKER, A., AND CRUZ-NEIRA, C. 2001. Vr juggler: A virtual platform for virtual reality application development. In *VR '01: Proceedings of the Virtual Reality 2001 Conference (VR'01)*, IEEE Computer Society, Washington, DC, USA, 89.

BIGLER, J., STEPHENS, A., AND PARKER, S. 2006. Design for parallel interactive ray tracing systems. Tech. Rep. Technical Report number UUSCI-2006-027, SCI Institute, University of Utah, Salt Lake City, USA.

CHOI, S. H., AND CHEUNG, H. H. 2006. A cave-based multimaterial virtual prototyping system,. *Computer-Aided Design and Applications 3*, 557–566.

CORSEUIL, E., RAPOSO, A., SILVA, R., PINTO, M., WAGNER, G., AND GATTASS, M. 2004. Environ - visualization of cad models in a virtual reality environment. In *In Eurographics Symposium on Virtual Environments (EG-VE)*, 79–82.

CRUZ-NEIRA, C., SANDIN, D. J., DEFANTI, T. A., KENYON, R. V., AND HART, J. C. 1992. The cave automatic virtual environment. *Commun. ACM 35*, 6, 64–72.

DIETRICH, A., WALD, I., AND SLUSALLEK, P. 2005. Large-scale cad model visualization on a scalable shared-memory architecture. In *Proceedings of Vision, Modeling, and Visualization*, Akademische Verlagsgesellschaft Aka, Erlangen, Germany, 303–310.

EPICGAMES, 2004. Unreal tournament, http://www.unrealtournament.com.

EXPLORATION, D., 2008. Right hemisphere - visual product communication and collaboration, http://www.righthemisphere.com/products/dexp/.

FONG, G. 2006. Adapting cots games for military experimentation. *Simulation and Gaming 37*, 4, 452–465.

GTKRADIANT, 2008. Gtkradiant, http://www.qeradiant.com/cgi-bin/trac.cgi.

HAMMANN, J. E., AND MARKOVITCH, N. A. 1995. Introduction to arena. In *Proceedings of the 27th conference on Winter simulation*, IEEE Computer Society, Washington, DC, USA, 519–523.

ISO10303, 1994. Standard for the exchange of product model data. http://www.steptools.com/library/standard/.

KIM, H., LEE, J.-G., LEE, S.-S., AND PARK, J. 2003. A simulation-based shipbuilding system for evaluation of validity in design and manufacturing. *Systems, Man and Cybernetics, 2003. IEEE International Conference on 1* (Oct.), 522–529.

KREITLER, M. 1995. Virtual environments for design and analysis of production facilities. *IFIP WG 5.7 Working Conference on Managing Concurrence Manufacturing to Improve Industrial Performance, Washington-USA*.

POLYTRANS, O., 2008. Polytrans, http://www.okino.com.

QUAKE, 1997. Id software, http://www.idssoftware.com.

RAJLICH, P., 2008. Cave quake iii arena, http://www.visbox.com/cq3a/.

SHERMAN, B., 2008. Freevr: Virtual reality integration library, http://www.freevr.org/.

STEPHENS, A., BOULOS, S., BIGLER, J., WALD, I., AND PARKER, S. G. 2006. An application of scalable massive model interaction using shared memory systems. In *Eurographics Symposium on Parallel Graphics and Visualization*, 19–26.

SYSTEMES, D., 2008. Delmia digital manufacturing e production, http://www.3ds.com/products/delmia.

VIZUP, 2008. Vizup technology, http://www.vizup.com.

VRCONTEXT, 2008. Walkinside, http://www.vrcontext.com/walkinside/.

VRML, 2008. Virtual reality modeling language, http://www.web3d.org/x3d/vrml/.

WALD, I., BENTHIN, C., EFREMOV, A., DAHMEN, T., GÜNTHER, J., DIETRICH, A., HAVRAN, V., SLUSALLEK, P., AND SEIDEL, H.-P. 2005. A ray tracing based virtual reality framework for industrial design. Tech. Rep. Technical Report number UUSCI-2005-009, SCI Institute, University of Utah, Salt Lake City, USA.

Parallel Lazy Amplification: Real-Time Procedural Modeling and Rendering of Multi-Terabyte Scenes on a Single PC

Carlúcio S. Cordeiro
Luiz Chaimowicz
Universidade Federal de Minas Gerais

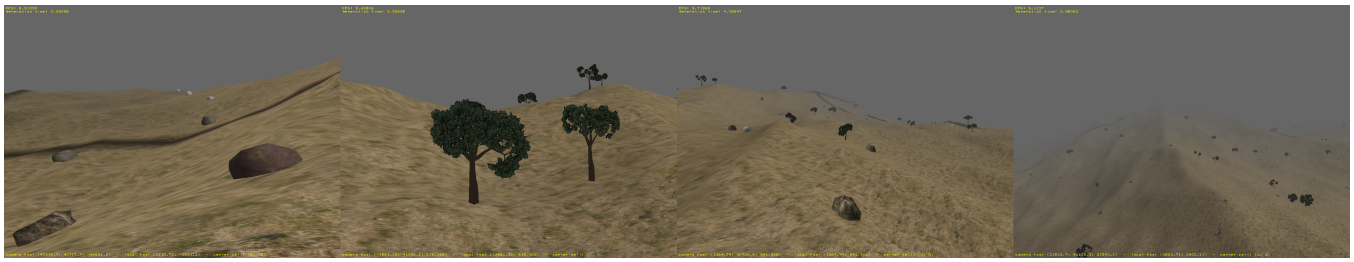


Figure 1: *Some images of our massive procedural world. Although quite simple, the entire scene has about 8 terabytes.*

Abstract

In this paper, we propose a new procedural modeling paradigm called *Parallel Lazy Amplification*. This paradigm may be understood as a combination between two traditional techniques of procedural modeling: *data amplification* and *lazy evaluation*. Data amplification consists in pre-synthesizing the whole geometry before viewing. Alternatively, in the lazy evaluation paradigm, the geometry is generated only when it is needed. The central idea of the paradigm that we propose is to pre-synthesize the geometry that will be potentially seen in the near future and keep it on a cache. A visibility prefetching algorithm determines which models should be generated and which will be discarded. The generation of models is done in parallel with the visualization, without interrupting the system. We implemented a prototype of this paradigm and some initial experiments with a procedural scene of about 8 terabytes showed the feasibility of this new paradigm, especially when performed on multi-core architectures.

Keywords: procedural modeling, geometry management, real-time rendering, parallel computing.

Author's Contact:

carlucio@gmail.com
chaimo@dcc.ufmg.br

1 Introduction

Procedural modeling is an research field in Computer Graphics that includes a number of alternatives to traditional geometric modeling. In this approach, the geometry is specified by a set of parameters and a procedure (algorithm) that creates a model from these descriptions. Some of the main motivations in this area have been the challenge of representing algorithmically the complexity of the objects in the real world both in terms of their form and their behavior. As examples of objects in that class we can mention terrains, vegetation, gases, liquids, fire, architectural buildings, cities and planets [Ebert et al. 2002].

The use of procedural modeling can greatly reduce the modeling time of massive and complex scenes. The use of these techniques is becoming increasingly common mainly in the entertainment industry. For example, procedural modeling techniques have been applied successfully in the movie industry [White 2006].

In computer games, procedural techniques were largely explored in the past. By that time, memory limitations imposed severe restrictions on storage and the use of procedural techniques was a creative way to solve these problems. Some examples includes the game *Elite*, originally published for the former *BBC Micro* in 1984, and *The Sentinel*, a game published for the *Commodore 64* in 1986.

With the modernization and the increase of available memory on computers and video-game consoles, the use of these techniques have been somewhat neglected by the game industry and have not been much used in the top games in the past years. Recently, with the great level of details that the games are presenting today, procedural techniques are becoming more popular again. If before the restriction was the hardware, now the limitation is the growing demand for artists. The relationship of artists by programmer is growing with each new generation of games. Currently, the game studios employ two to three artists per programmer. Thus, procedural modeling help reducing the need for artists generating scenarios automatically.

Another inspiration for this work is the Demoscene [Tasajärvi et al. 2004; Demoscene.info], a digital art subculture that produces audio-visual applications in real time, called *Demos*. This culture has emerged between users of old platforms, such as *Apple II*, *Commodore 64*, *ZX Spectrum* and *Commodore Amiga*. The Demos are applications whose executable code is generally composed of only a few kilobytes. The most common categories are 4 Kb and 64 Kb. They usually do not use any kind of external file, as models, pictures, music and sound. All resources are compressed or synthesized. The Demos are undoubtedly a form of digital art amazing and unique.

Procedural modeling brings at least two major challenges for research in computer graphics. The first challenge is to create procedures and algorithms that synthesize complex and realistic objects and textures. The second challenge is to manage the large amounts of data that are generated by procedural models. This second challenge is relatively less studied and is the main focus of this paper.

In a home PC today, a single procedural model of a terrain, a tree, or a building is easily generated and rendered in real-time. But to generate and visualize a massive procedural model in real-time, as a huge forest, a large urban center, or even an entire planet, more elaborate techniques and tools are required. In these scenarios, the amount of geometry can easily extrapolate the available main memory.

Basically, there are two main techniques for data generation: *data amplification* and *lazy evaluation* (these paradigms will be detailed in the section 3.3). Data amplification consists in pre-synthesizing the whole geometry before viewing while lazy evaluation paradigm generates the models only when they are needed. Lazy evaluation works well for offline rendering. However, to generate all the geometry of each frame in real-time becomes practically impossible. In the other hand, data amplification pre-generates all the geometry and is feasible to be displayed in real-time. But it applies only to models that fit in the main memory.

In this paper, we propose a paradigm that combine data amplification and lazy evaluation. The main idea is to generate the geometry on demand but, differently from lazy evaluation, when the geometry is generated, it will be kept in a cache to be used in subsequent frames. As with data amplification, we have a pre-processing time for building the cache before the beginning of visualization. During the visualization, the system determines which models or parts of it will be seen in the near future and which are already out of context. The models that have low priority of been viewed are discarded, and those who are potentially in the field of view are generated in parallel with the visualization, without interrupting user interaction with the system. We called this new paradigm *Parallel Lazy Amplification*.

The rest of this paper is organized as follows. In Section 2, we discuss some of the main works related to this paper. In Section 3, we do a quick review of basic concepts that are used in the text. The Parallel Lazy Amplification paradigm proposed in this paper is presented in Section 4. Section 5 gives an overview of the graphics engine implemented for the experiments. In section 6, we present and discuss the experimental results obtained. Finally, our conclusions and future work are show in Section 7.

2 Related Work

Procedural techniques have been used throughout the computer graphics history. Many researchers have developed their own procedures to simulate materials, objects and natural phenomena. One of the first procedural techniques used in the graphics community were fractals. Musgrave et al. [1989] describes a model for the synthesis of eroded terrains based on fractals. Later, he described procedural models of a whole planet [Musgrave 1999]. This work was the basis for *MojoWorld* software[MojoWorld] that was originally created by Ken Musgrave.

Another procedural modeling technique is called *L-Systems*, which was originally proposed by Lindenmayer [1968] to model the development of filamentous bodies in a work of theoretical biology. As will be explained later, L-Systems are quite similar to formal grammars used in compilers. The pioneer work in the use of L-Systems in computer graphics is Prusinkiewicz [1986]. The use of this technique is very powerful for the procedural modeling of plants and vegetation, as described in Prusinkiewicz and Lindenmayer [1990] and Deussen et al. [1998]. Inspired by the work of Prusinkiewicz, some authors developed an extension of L-Systems for procedural modeling of cities and buildings called *CGA shape* [Parish and Müller 2001; Müller et al. 2006; Müller et al. 2007]. Like the L-Systems, these techniques are similar to a formal grammar.

Another extension was proposed in [Lluch et al. 2003] to generate procedural models of plants and trees with multi-resolution information embedded in the model. This representation is more appropriate and does not fail in preserving the visual structure of the model, as normally occur with the use of a geometry simplification algorithm.

In an attempt to integrate the characteristics of different procedural modeling systems, Gangster and Klein [2007] presents a new kind of visual language in a single modeling environment. The system shows results of procedural models consisting of buildings, plants and terrains, without the need of external tools.

In the area of representation and management of procedural geometry, Hart [2002] shows how the scene graphs can be used for that purpose. The work presents the paradigms data amplification and lazy evaluation. It also described the Procedural Geometric Instancing technique, which follows the lazy-evaluation paradigm.

Researchers have studied the problem of rendering complex models at interactive frame rates for many years. Clark [1976] proposed many of the techniques for rendering complex models used today, including the use of hierarchical spatial data structures, level-of-detail (LOD) management, hierarchical view-frustum and occlusion culling, and working-set management (geometry caching). Garlick et al. [1990] presented the idea of exploiting multiprocessor graphics workstations to overlap visibility computations with rendering.

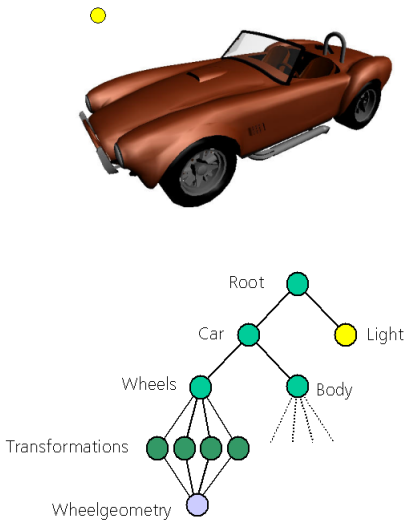


Figure 2: *Example of a scene graph: below is a description of a hierarchical scene and above we have the same scene rendered.*

Correa [2004] introduced a system for interactive viewing of large datasets. The system uses new techniques for out-of-core visualization of models larger than main memory. In a pre-processing phase, a hierarchical decomposition of the model is built using a octree, the coefficients used to test visibility are calculated, and levels of detail are determined. In run time, multiple threads are used to override calculations of visibility, managing cache and rendering.

Out-of-core visualization techniques could be used with data amplification to avoid the size limitations of the main memory. But with this approach the model should be fully generated on disk and pre-processed before viewing. In addition to the generation and pre-processing time, this method would demand extra storage space in disc, which can become a problem for massive multi-terabyte scenes. Pre-generate all the geometry is also less flexible, because the whole model would be completely static. As mentioned, the approach proposed in this paper try to overcome these problems combining Lazy Evaluation and Data Amplification.

3 Basic Concepts

3.1 Scene Graphs

Scene graphs are data structures that are much discussed and researched in computer graphics field. Basically, a scene graph is a spatially consistent data structure, which is used for the representation of three-dimensional virtual environments in computer graphics applications, including procedural modeling. Examples of these structures include bounding volume hierarchies, octrees and grids. Dollner and Hinrichs [2000] present a detailed discussion of scene graphs.

A scene graph is a directed acyclic graph. Each node has a set of attributes that may or may not influence its children nodes. The nodes are organized in a hierarchical fashion corresponding semantically and spatially to the modeling scene.

All scene graph nodes have an attribute called bounding volume. This attribute is a simple volume, usually a box or a sphere, which includes the geometry of all of its children nodes. The bounding volume is used for fast computations of approximate intersection tests of the node with other objects. Intersection tests are mainly used to determine visibility and collision. Figure 2 [OpenSG] illustrates the scene graph concepts.

3.2 Bounding Volumes

The bounding volumes for procedural models can be static or dynamic. A static bounding volume involves all possible geometries synthesized by the procedural model. In other words, a dynamic bouding volume is designed to fit tightly each particular instance of the procedural model. Dynamic bounding volumes are more efficient than static bounding volumes, but they are much more difficult to plan. Dynamic bounding volumes need to be computed at instantiation time.

All procedural models should provide a method for bounding volume estimation from its parameters. Depending on the procedural model, the bounding volume computation can be very simple or very hard. Heuristics to determine the bounding volume should be developed when computing the optimal bounding volume is not feasible.

3.3 Procedural Modeling Paradigms

Virtually all interactive graphics systems follow the same pattern of rendering pipeline, with three conceptual stages: application, geometry, and rasterization. The *user* articulates a conceptual model to the modeler. The *modeler* interprets the articulation and converts it into an intermediate representation suitable for rendering by the *renderer*.

The synthesis of procedural models follows the same data flow. But the way the intermediate representation is generated can follow two different paradigms: *data amplification* and *lazy evaluation* [Hart 2002].

3.3.1 Data Amplification

Models that follow the data amplification paradigm synthesize some sort of intermediary geometric representation. This intermediary representation is generally a description consisting of triangles, polygons or other primitives. Smith [1984] coined the term data amplification to explain how procedural models transform a small amount of data in models with rich details and described by a large amounts of geometry.

The intermediate representation of a complex scene can become very large. For example, a very simple procedural model of a tree, described by only a few floating-point numbers, when evaluated produces a few thousand triangles. A huge forest can easily extrapolates main memory limitations. Data amplification causes data explosion due to the fact that the procedural model is converted into a geometric representation to be rendered. Figure 3 [Hart 2002] illustrates the data amplification paradigm.

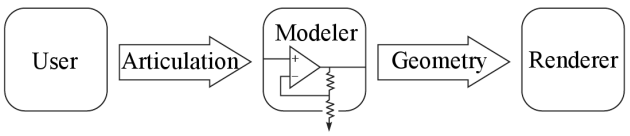


Figure 3: *Diagram that shows the data amplification paradigm.*

3.3.2 Lazy Evaluation

Lazy evaluation avoids the intermediate representation problems that we have with data amplification. The geometry synthesis procedure is performed only when necessary. Procedural models demand a large processing time, then to generate all the geometry each frame in real-time applications is practically impossible. Lazy evaluation keeps a dialogue between the Renderer and the Modeler, as illustrated in Figure 4 [Hart 2002].

Scene graphs also support lazy evaluation for procedural models. For example, the system can generate a bounding volume for a procedural model and perform a test to determine if the geometry contained therein is necessary to render the scene. If the test is negative, the system does not generates the procedural model. The heuristic here is to determine the bounding volume without actually performing the procedure for the model generation.

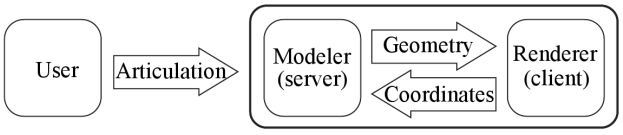


Figure 4: *Diagram that shows the lazy evaluation paradigm.*

3.4 Visibility Prefetching Algorithms

In the proposed paradigm, visibility prefetching algorithms will be used to determine which models, or parts of them should be generated on demand. Visibility prefetching algorithms can be based on the viewpoint of the observer and the bounding volumes [Corrêa et al. 2003]. In out-of-core rendering, the system uses the algorithm to determine the geometry most likely to be seen in the near future, which are read from disk and kept in a cache.

One difference between a out-of-core visualization system and procedural approach, is that in out-of-core visualization we have the whole model a priori. In a pre-processing stage the model is divided in a top-down fashion. In the procedural modeling, the model is generated at runtime. Then the partitioning can be built bottom-up in the model generation.

4 Parallel Lazy Amplification

The main idea of the Parallel Lazy Amplification proposed in this paper is to combine the paradigms of data amplification and lazy evaluation. During the visualization process, the system estimates a set of potentially visible models that will be seen in the near future. Similar to lazy evaluation, the geometry is generated on demand and also in parallel with the visualization. But not only the visible geometry is generated. As with data amplification, parts of the scene are pre-generated and maintained in a cache of geometry to be used in the following frames.

Figure 5 illustrates the Parallel Lazy Amplification paradigm using the same symbolic notation of Hart [2002]. When the *Renderer* needs of a certain model, it makes a request to the *Cache*. To try to ensure that the geometry will be available when requested, the *Cache* uses a visibility prefetching algorithm. When the algorithm determines that a procedural model should be present in the Cache, it makes a request to the *Synthesis Manager*. The *Synthesis Manager* task is to manage and load-balancing the *Modelers*, which are responsible for the generation of models. So far, the prototype implemented uses only one *Modeler* and the *Synthesis Manager* uses a First-In First-Out (FIFO) policy. When more *Modelers* are present, FIFO may not the ideal algorithm because it can generate load unbalance. For example, supose a particular situation when we have 10 models to generate and two modelers. If the fisrt and sixth models takes 0.4 seconds to generate each model, and the other eigth models takes 0.1 seconds each, one good strategy is one modeler generate the fisrt and sixth models and the second modeler generate the other eigth models. In Section 6, we will discuss a strategy for a better distribution of processing between the *Modelers*.

The visibility prefetching algorithm used in our approach is the *Prioritized-Layered Projection* (PLP) [Klosowski and Silva 2000]. PLP is a visibility test algorithm that needs very little processing. The PLP can be understood as the traditional hierarchical frustum culling algorithm, used to discard models outside the field of view. In the frustum culling algorithm, the scene graph is traversed in a depth first order, from the root node to the leaves. If a node is outside the field of view, the node and all its children are discarded. In the PLP, the leaf nodes are kept in a priority queue called front. When a node in the front is visited, it is added to a set of visible nodes. Then, the node is removed from the front and all its neighbors that have not yet been visited are added to the front. PLP requires that each node of the scene graph refers not only to their children but also all to its neighbors.

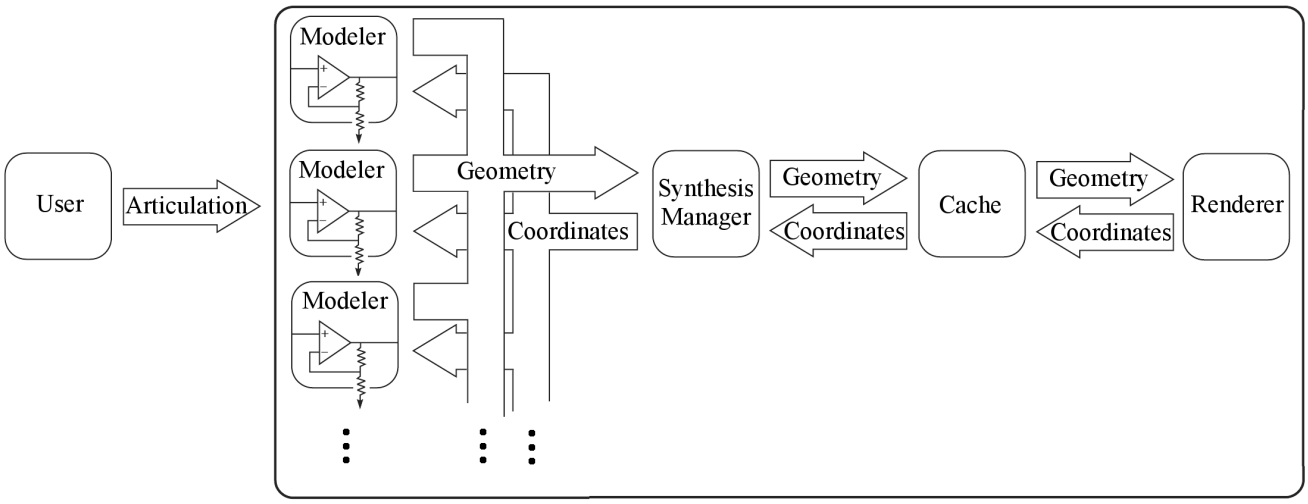


Figure 5: *Diagram that shows the parallel lazy amplification paradigm.*

The key point in the PLP implementation is the heuristic that determines priorities for each node. Klosowski and Silva [2000] presents several heuristics to compute the initial solidity of a node and accumulates it in a certain direction. The solidity of a node is an estimate of how much he occludes an object behind himself. The heuristic that we use was proposed by Correa et al. [2002]. This is a very simple heuristic to determine priorities for nodes. The node containing the camera receives priority -1, its neighbours receive priority -2, the neighbours of neighbours receive priority -3, and so on.

5 Implementation Details

To implement and test the paradigm described in this paper we developed a game engine (PSYGEN) that supports different procedural models. This section presents an overview of this engine and the models that have been implemented. It is important to clarify that it is not the objective of this paper to discuss the architecture of a game engine as the PSYGEN. We will only show some details related to procedural modeling employed by the system.

5.1 PSYGEN

PSYGEN (Procedural Synthesis Graphics Engine) is a graphics engine that allows the implementation and visualization of different procedural modeling algorithms and implements various techniques for real-time rendering. It is composed of several modules that abstracts and facilitates the implementation of procedural modeling algorithms. Figure 6 shows a UML diagram with the organization of the prototype that we implemented.

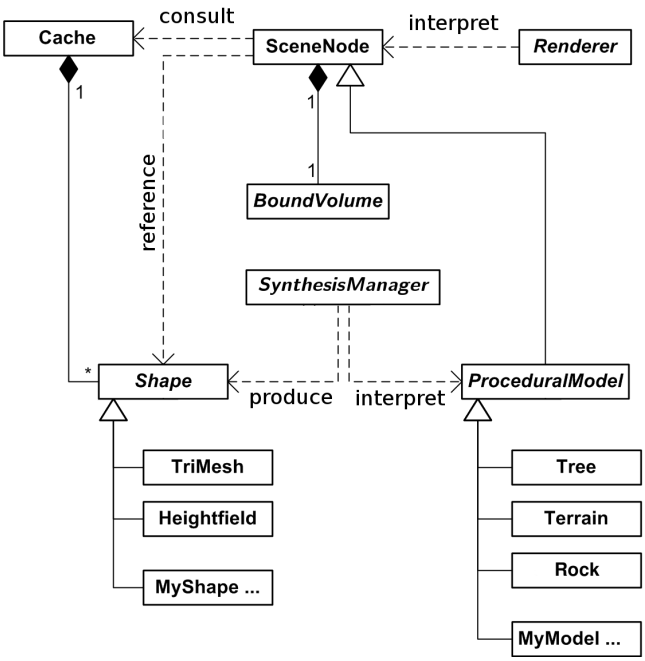


Figure 6: *UML diagram of the main classes involved in the PSYGEN's procedural modeling system.*

- **Renderer:** the renderer provides an abstract interface that encapsulates hardware calls and the graphics API. At the present time, PSYGEN has only an implementation of a renderer using OpenGL.
- **Cache:** is responsible for memory managing of the system, maintaining a collection of models and objects potentially seen by the observer. The cache determines which models should be generated, based on PLP algorithm. The cache should also discard geometry with low priority.
- **SynthesisManager:** is responsible for the generation of models. SynthesisManager runs in parallel with the renderer, allowing the generation of models without interrupting the interactive visualization. In the implemented prototype, the Synthesis Manager also plays the role of Modeler. This Manager was implemented using *pthreads*.
- **SceneNode:** the main structure of the system of data visualization of PSYGEN is a scene graph. SceneNode abstract class is the base of all of scene graph nodes. A SceneNode can reference zero or more Shapes, which share the same render state encapsulated by SceneNode.
- **ProceduralModel:** provides an abstract interface to procedural models supported by the system. Some examples of procedural models in PSYGEN implemented so far include trees, terrains and rocks.
- **Shape:** all models and geometry of the system derives from abstract class Shape, for example, the classes TriMesh and Heightfield. These are concrete classes that implement the methods of Shape.

5.2 Procedural Models

In this section we presented the three procedural models in PSYGEN implemented so far (rock, terrain and tree) and briefly discuss the asymptotic growth of the algorithms for model generation. Figure 7 shows an image of PSYGEN with the three procedural models implemented.

Figure 7: *A view of our simple procedural world. This image shows the three procedural models that we implemented: trees, rocks and terrains.*

5.2.1 Rock

The procedural model of a rock implemented in PSYGEN is a variation of a classic technique know as midpoint displacement or plasma fractal. The difference is that the rock algorithm starts with a tetrahedron and the classic algorithm is done on a plane. Each triangle in the tetrahedron is divided into four new triangles, so that new points are the average points of initial edges of the triangle. After the subdivision, the new corners are disturbed in the direction of the surface normal at the point in question. The parameters of the rock procedural model are:

- **Seed:** Seed to the pseudo-random number generator.
- **Subdivisions:** Number of divisions in which the original mesh will be divided.
- **Radius:** Average radius of the rock.
- **Amplitude:** Disturbance factor in the radius of each iteration.
- **Decay:** Decay factor of the amplitude of each iteration.

The parameter that dominates the generation time is the number of subdivisions. If n is the number of subdivisions and c is a constant that depends on the processor used, the time complexity of rock model is given by:

$$T(n) = c\, n\, 4^{n-1} \tag{1}$$

5.2.2 Terrain

Terrain is probably the most popular procedural model in literature. The PSYGEN terrain procedural model is an implementation of the Ridged Multifractal Algorithm described in Ebert et al. [2002]. The Ridged Multifractal parameters are:

- **Seed:** Seed to the pseudo-random number generator.
- **Heighmap Size:** Size of terrain block.
- **Height:** Determine the largest possible fractal dimension.
- **Lacunarity:** Gap between successive frequencies.
- **Octaves:** Number of frequencies in the multifractal..
- **Offset:** Factor that determines the multifractal characteristic.

The parameters that influence the generation time are the heightmap size and the octaves. If n is the heightmap size and m is the octaves, the time complexity of terrain generation algorithm is given by:

$$T(n, m) = c\, m\, n^2 \tag{2}$$

As the octaves does not change for a particular terrain, we can also describe the time complexity in a simplified form:

$$T(n) = c\, n^2 \tag{3}$$

As the terrain is virtually infinite, it should be built in blocks. We use block sizes of 64 and 9 octaves in our prototype.

5.2.3 Tree

Procedural models of trees are usually implemented using L-Systems [Prusinkiewicz and Lindenmayer 1990]. L-Systems are quite similar to formal grammars used in compilers. From an initial symbol of a particular L-System, the model is derived by a number of iterations. The derivation tree (data structure) represents structurally the model (object tree). The parameters of L-System that we implemented are:

- **Seed:** Seed to the pseudo-random number generator.
- **Iterations:** Number of iterations that will determine the tree height.
- **Branches:** Average number of ramifications. The algorithm select a random value between branches-2 and branches+2.
- **Size:** Size of the first branch.
- **Radius:** Radius of the first branch (trunk).

If n is the number of iterations and m is the average number of branches by iteration, the time complexity of tree generation algorithm is given by:

$$T(n, m) = c\, m^n \tag{4}$$

6 Experimental Results

We performed a set of initial tests and experiments for an preliminary analysis of the proposed paradigm and the impact of the generation of models in parallel with the visualization. To do this we defined a procedural world made of a large terrain divided into blocks. Each block has width 64 vertices, totaling 7,938 triangles per terrain block. For each block, 23 trees were distributed with 3 to 5 iterations and 2 to 4 branches by iteration, and also 40 rocks with 2 subdivisions. Whereas each tree has on average 5,000 triangles and each rock is accurate 64 triangles, each block has about 125,000 triangles. The total size of the world was set to 512x512 blocks. That totals approximately 32,768,000,000 triangles throughout the virtual world. As each triangle has information as its vertices, normal vectors, coordinates of texture and other values (about 256 bytes), the estimated total size of the world procedural hold about 8 terabytes, if it were entirely generated. That means even to limitations of secondary memory (disk) if we deemed current reality of home PCs.

To the first battery of tests, we established a path through which the camera flies in the virtual world. During the camera walk, a log was generated with the measurements of the framerate for each frame along the walk. The experiments were performed on two different computer configurations:

1. AMD Athlon 64 3400+ processor, 1 GB of RAM and a ATI Radeon X800 XT Platinum Edition GPU running Ubuntu Linux 8.04.
2. Intel Core 2 Duo T8300 2.4 GHz, 2 GB of RAM and a NVIDIA GeForce 8600 GT GPU running Mac OS X 10.5.3.

Figure 8(a) shows the results for the first configuration. The result clearly shows that when the generation takes place in parallel with visualization, the framerate drops sharply from about 25 frames per second to less than 15. Though not interrupting the process of interactive visualization, we can clearly see the performance drop.

The same experiment was performed in a dual-core processor (configuration 2). Figure 8(b) shows the results of this second experiment. We can see that when the generation occurs in a multicore processor, the framerate does not drop as in a single core processor. To confirm this assertion, we carried out a third experiment with a

configuration similar to the second test. However, this time one of processor cores was off and the test was conducted with only one core asset. The results are shown in Figure 8(c).

The results of this first battery of tests confirms our hypothesis that when the parallel generation is done in multicore architectures, we do not have major impacts on the framerate of the system.

A second battery of tests was performed to confirm the complexity analysis presented at the Section 5.2. For this we measured the generation time of each model for a given set of parameters. These data were used to carry out a curve fitting with their respective complexity functions of each model. The experiments were performed using the first computer configuration. Figure 9 shows the results.

Analyzing the results, we observed that the time complexity of generation algorithms reflects well the generation time. Thus, the system can calculate the constants for each complexity function in the startup. Once constants are computed, the complexity function provides a reasonable estimate of the generation time for a given procedural model. The generation time is a essetial information to have a better load balancing, as we have shown in Section 4.

7 Conclusions and Future Work

This article proposed a new procedural modeling paradigm called *Parallel Lazy Amplification*. In a comparative analysis of parallel lazy amplification with lazy evaluation and data amplification, we can see that parallel lazy amplification can manage large amounts of data that can not be hold by data amplification. We also noticed that lazy evaluation generates all the procedural models whenever they are seen and has no memory management. In a simple situation where the camera is spinning around its axis would do the same models are generated and discarded in a cycle. As the generation of procedural models may require great processing time, this can cause models not shown correctly, models popping in screen and other problems that makes the use of lazy evaluation in real-time impracticable. we believed that we showed the feasibility of the proposed paradigm for the generation and display of massive procedural models in real-time, testing with a very simple procedural world, but that hold terabytes of data if it were entirely generated. The tests also showed that the paradigm is suitable for multicore architectures. Tests done in a dual-core processor showed a minimal impact on the framerate.

The generation time of each procedural model was also analyzed. We showed that the asymptotic growth of algorithms for procedural models provide a good estimate of the generation time. With this estimative we can do a better schedule of models to be generated by available processors and can get a better load balancing of the system.

In a future work, we planned to use the asymptotic growth to determine a good schedule policy for the Synthesis Manager. Other future work includes an implementation with more than one Modeler running on different threads, enabling the generation of more than one model in parallel. We also plan a parallel distributed memory version of PSYGEN to run on visualization clusters. A multithread version is more suited to the reality of home PCs. Dual-core and quad-core processors are already common today. However, We intended to analyze the efficiency and scalability of the system in more than four CPUs, as well as the impact of parallelism in procedural generation. In this scenario, the distributed memory version will be more appropriate, allowing the execution of the system in a visualization cluster with 32 CPUs, for example. Other features planned to be implemented in the future include:

- **Level-of-Detail (LoD) Management:** As procedural models are multiresolution in nature, the system can generate the model with various levels of detail.
- **Geometry Shaders:** The latest GPU generations has a new type of shader called geometry shader. Unlike the vertex shaders and pixel shaders, geometry shaders can create new vertex during its evaluation. This new feature allows the generation of geometry directly into the GPU, allowing a more compact intermediate representation of the model.
- **Mixing traditional models with procedural models:** We also have plan to develop tools and editors to integrate traditional models along procedural models, providing more control to the user in the desired points in the procedural world.

Acknowledgements

We would like to thank Wagner T. Corrêa, for all comments and suggestions.

References

CLARK, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Commun. ACM 19*, 10, 547–554.

CORRÊA, W. T., KLOSOWSKI, J. T., AND SILVA, C. T. 2002. Fast and simple occlusion culling. In *Game Programming Gems 3*. Charles River Media, 353–358.

CORRÊA, W. T., KLOSOWSKI, J. T., AND SILVA, C. T. 2003. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of PVG 2003 (6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics)*, 1–8.

CORRÊA, W. T. 2004. *New techniques for out-of-core visualization of large datasets*. PhD thesis, Princeton, NJ, USA.

DEMOSCENE.INFO. A webportal providing information on the demoscene. `http://www.demoscene.info/`.

DEUSSEN, O., HANRAHAN, P., LINTERMANN, B., MĚCH, R., PHARR, M., AND PRUSINKIEWICZ, P. 1998. Realistic modeling and rendering of plant ecosystems. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 275–286.

DÖLLNER, J., AND HINRICHS, K. H. 2000. A generalized scene graph. In *Proceedings of Vision, Modeling and Visualization 2000*, IOS Press, Amsterdam, H. N. H.-P. S. B. Girod, G. Greiner, Ed., 247–254.

EBERT, D. S., MUSGRAVE, K. F., PEACHEY, D., PERLIN, K., AND WORLEY, S. 2002. *Texturing & Modeling: A Procedural Approach*, third ed. Morgan Kaufmann, December.

GANSTER, B., AND KLEIN, R. 2007. An integrated framework for procedural modeling. In *Spring Conference on Computer Graphics 2007 (SCCG 2007)*, Comenius University, Bratislava, M. Sbert, Ed., 150–157.

GARLICK, B., BAUM, D., AND WINGET, J. 1990. Interactive viewing of large geometric databases using multiprocessor graphics workstations. In *Siggraph Course: Parallel Algorithms and Architectures for 3D Image Generation*.

HART, J. C. 2002. Procedural synthesis of geometry. In *Texturing & Modeling: A Procedural Approach*, third ed. Morgan Kaufmann.

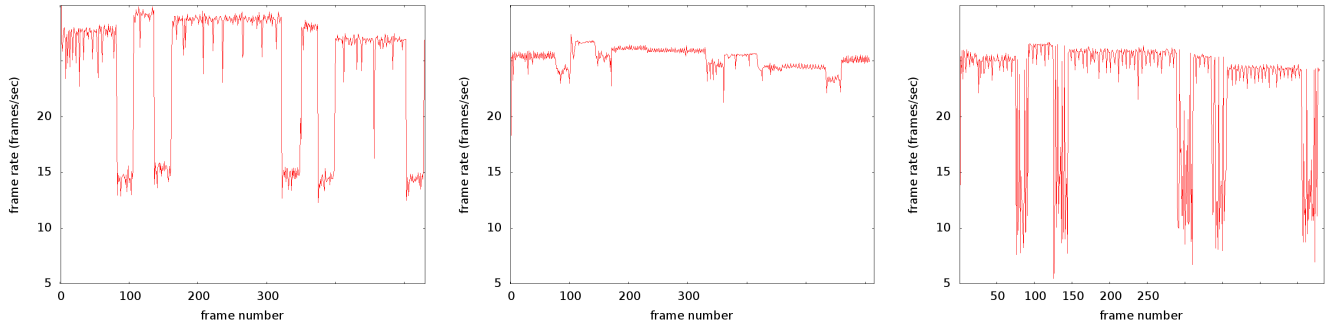
KLOSOWSKI, J. T., AND SILVA, C. T. 2000. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics 6*, 2, 108–123.

LINDENMAYER, A. 1968. Mathematical models for cellular interaction in development, parts i and ii. *Journal of Theoretical Biology 18*, 3.

LLUCH, J., CAMAHORT, E., AND VIVÓ, R. 2003. Procedural multiresolution for plant and tree rendering. In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, ACM, New York, NY, USA, 31–38.

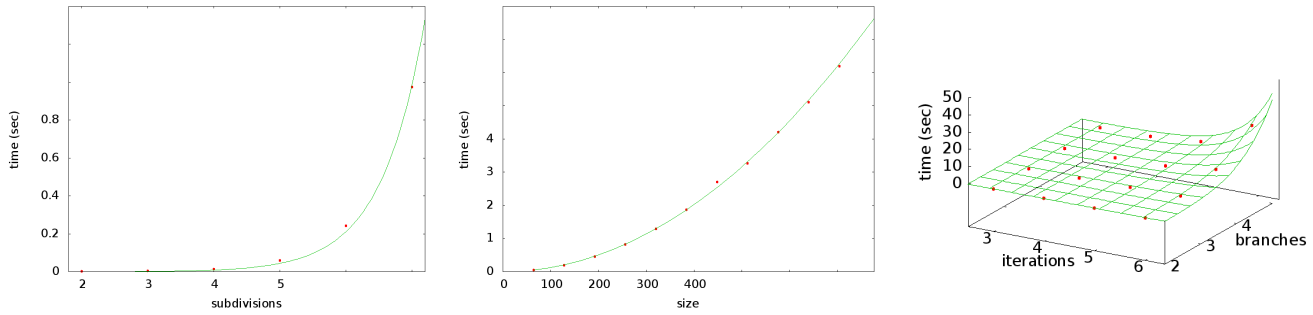
MOJOWORLD. Mojoworld 3. http://www.pandromeda.com/.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND GOOL, L. V. 2006. Procedural modeling of buildings. 614–623.



(a) AMD Athlon 64. (b) Intel Core 2 Duo with two active cores. (c) Intel Core 2 Duo with only one active core.

Figure 8: *Using a multi-core processor to improve frame rates. We measured the frame rates during a 27 seconds walkthrough of the procedural world under three configurations: (a) using a single-core CPU; (b) using a dual-core CPU; (c) using the same dual-core machine used in (b), but with one of the cores off.*



(a) Rock (b) Terrain (c) Tree

Figure 9: *Asymptotic growth for the three procedural models analysed.*

MÜLLER, P., ZENG, G., WONKA, P., AND GOOL, L. V. 2007. Image-based procedural modeling of facades.

MUSGRAVE, F. K., KOLB, C. E., AND MACE, R. S. 1989. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 41–50.

MUSGRAVE, K. 1999. Building worlds in cyberspace. In *CGI '99: Proceedings of the International Conference on Computer Graphics*, IEEE Computer Society, Washington, DC, USA, 164.

OPENSG. Open scene graph. http://www.opensg.org/.

PARISH, Y. I. H., AND MÜLLER, P. 2001. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press, New York, NY, USA, E. Fiume, Ed., 301–308.

PRUSINKIEWICZ, P., AND LINDENMAYER, A. 1990. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA.

PRUSINKIEWICZ, P. 1986. Applications of l-systems to computer imagery. In *Graph-Grammars and Their Application to Computer Science*, Springer, H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, Eds., vol. 291 of *Lecture Notes in Computer Science*, 534–548.

SMITH, A. R. 1984. Plants, fractals, and formal languages. *SIGGRAPH Comput. Graph. 18*, 3, 1–10.

TASAJÄRVI, L., STAMNES, B., AND SCHUSTIN, M. 2004. *Demoscene: the Art of Real-Time*. Even Lake Studios.

WHITE, C. 2006. King kong: the building of 1933 new york city. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, ACM, New York, NY, USA, 96.

Posicionamento de Câmeras através de Previsão das Simulações Físicas

Daniel Pires Erick Passos Esteban Clua Anselmo Montenegro

Universidade Federal Fluminense, Instituto de Computação, Brasil

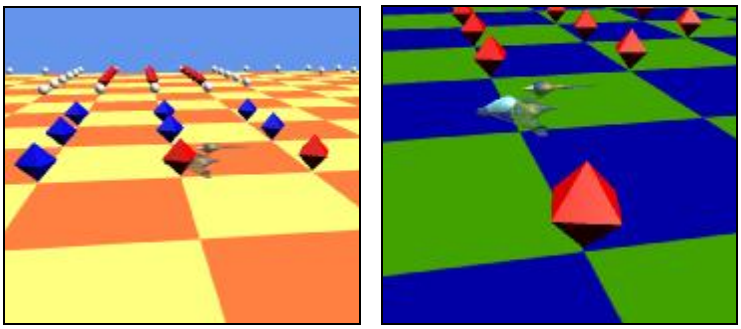


Figura 1: *Replay* gerado através das técnica de reconhecimento de ações e previsão física.

Resumo

Nos ambientes virtuais 3D, a liberdade para o posicionamento de câmeras vêm permitindo que as aplicações utilizem técnicas de filmagem semelhantes às usadas nas apresentações cinematográficas. Porém, com a existência de interatividade nesses ambientes, não existe uma sequência predeterminada de acontecimentos que permita o posicionamento dinâmico ideal. Faz-se necessária então a utilização de técnicas de reconhecimento e previsão de ações para se posicionar a câmera de forma satisfatória. Este trabalho apresenta uma solução para o posicionamento automático de câmeras baseada em técnicas de reconhecimento e previsão de ações no ambiente virtual interativo com base no estado da simulação física.

Palavras-chave: câmeras virtuais, previsão de ações, cinematografia

Contato:
{dpires, epassos, esteban, anselmo}@ic.uff.br

1. Introdução

Com o advento da tecnologia para geração de gráficos 3D, o conceito de uma câmera virtual tem sido indispensável, estando initimamente ligado a outros como objeto e ambiente 3D. Um objeto 3D existe em um ambiente 3D e é preciso mostrá-lo ao mundo exterior através de uma câmera virtual.

Câmeras virtuais possuem algumas propriedades próprias que as diferem das câmeras do mundo real. A princípio elas são desprovidas de atributos físicos como massa e volume, e podem ser posicionadas e orientadas livremente pelo ambiente 3D, podendo inclusive atravessar outros objetos. Além disso, câmeras virtuais possuem um limite de visualização bem definido, que dirá ao processador quais objetos, ou polígonos serão exibidos.

Ao se manipular uma câmera virtual sem se considerar propriedades físicas, obtem-se uma apresentação de uma cena 3D um tanto quanto irreal e estilizada. Com o objetivo de aproximar a simulação da realidade, desenvolvedores e *designers* passaram a modelar câmeras virtuais com atributos físicos, permitindo assim que esta possuísse uma aceleração quando se movimenta e levando em consideração colisões com outros objetos da cena.

Na cinematografia atual, a câmera é um dos principais elementos capazes de realçar o que se quer mostrar, além de introduzir subjetividade à cena dependendo do ângulo de visualização, do que está sendo focado e do tempo de tomada da cena [Martin 1985]. Com o passar do tempo, os desenvolvedores de ambientes (jogos) 3D se inspiraram nas produções cinematográficas para adicionar mais efeitos e emoção às suas produções [Hawkins 2005]. Termos como *zoom* e *pan* foram incorporados à interface de movimentação da câmera virtual. De fato, a programação de uma câmera virtual vem se tornando cada vez mais parecida com os comandos que um operador de câmera do mundo real receberia, com instruções de mais alto nível. A experiência da indústria do cinema explica e justifica o uso de cada tipo de tomada de câmera, constituindo assim uma linguagem própria, que também é aproveitada na produção de apresentações e jogos 3D.

No entanto, em aplicações 3D interativas, uma boa utilização de uma câmera virtual é bastante dificultada devido à imprevisibilidade dos acontecimentos. Em aplicações com seqüências pré-definidas, já se sabe exatamente o que vai acontecer com os elementos da cena, e isso torna possível o posicionamento e

orientação ideais da câmera, de acordo com os resultados desejados. Porém, quando não se sabe previamente as transformações que podem acontecer com os objetos do ambiente 3D, é necessário se utilizar técnicas alternativas para a manipulação da câmera. É preciso, principalmente, reconhecer as ações e identificar os objetos relevantes para que a cena seja transmitida de forma apropriada, ainda se considerando a probabilidade da ocorrência de algum outro evento. Após o reconhecimento das ações no ambiente interativo, o sistema precisa ainda escolher qual é a mais relevante a cada instante, exibindo-a de forma conveniente com a emoção esperada. Para isso é preciso escolher uma boa posição de câmera, definir o que será focado e determinar o tempo da tomada.

Tendo escolhido uma ação para ser filmada, uma estratégia para posicionar a câmera, de baixo custo computacional, é se basear na simulação física, prevendo o que irá acontecer nos próximos momentos dentro do ambiente virtual. Para exemplificar o uso da técnica foi desenvolvido um jogo, onde o usuário tem a sua visão de jogo, e após sua jogada é exibido um *replay* de toda a ação, o que também representaria a visão de um telespectador.

Neste trabalho, primeiramente serão discutidos os trabalhos da comunidade científica relacionados ao problema. Em seguida serão apresentados alguns conceitos relevantes da área de cinematografia, seguidos pela estrutura de câmeras virtuais na plataforma XNA (www.xna.com). Finalmente será abordado o problema, as soluções propostas e resultados obtidos, explicando todas as particularidades do jogo produzido como exemplo.

2. Trabalhos Relacionados

Desde o surgimento da cinematografia, diversos autores vêm estudando novas técnicas para deixar as produções mais atraentes para o telespectador, e as escolas de cinematografia também se multiplicaram. Hoje existem boas publicações sobre as técnicas cinematográficas, como o trabalho de Martin [1985], além de *websites* especializados, como o Mnemocine [2008]. Mais recentemente, as mesmas técnicas vêm sendo amplamente usadas em jogos, para aumentar a imersão do jogador no ambiente 3D. Além disso, começou-se a pensar na possibilidade de se exibir uma partida de um jogo 3D exatamente como nas transmissões esportivas. O trabalho de Drucker [1994] analisou a viabilidade dessa aplicação das técnicas cinematográficas.

Existem diversos trabalhos que visam facilitar o problema de posicionar a câmera no ambiente virtual. Alguns deles utilizam uma DSL (*Domain-Specific Language* – Linguagem Específica de Domínio). São extremamente úteis por posicionar a câmera através de comandos de bem mais alto nível do que as linguagens de programação convencionais, porém têm a desvantagem de deixar o posicionamento das câmeras predefinido, apenas ativam uma camera ou outra dependendo dos eventos que ocorrem no ambiente. Os trabalhos de He et al. [1996] e Amerson & Kime [2000] se baseiam no uso de DSLs para posicionamento dinâmico de câmeras.

Outros trabalhos, como os de Drucker [1994] e Hermann & Celes [2005], posicionam a câmera de forma mais dinâmica, sem a intervenção do usuário. Têm um alto apelo pela automatização de tarefas, mas oferecem um resultado cinematográfico mais pobre se comparado ao uso de DSLs. O sistema de Hermann e Celes [2005] ainda divide o trabalho de posicionar a câmera em módulos roteirista, diretor e cinegrafista. Porém, no XNA, o módulo cinegrafista proposto por eles se confunde com o próprio mecanismo de geração de imagens no XNA. Com isso, o módulo diretor pode dar a sua saída de dados diretamente para o *shader* da aplicação. Este trabalho apresenta uma aplicação capaz de reconhecer as ações do ambiente e posicionar a câmera de tal forma que sejam considerados os eventos do ambiente virtual e a previsão dos próximos acontecimentos, através de cálculos físicos e o aproveitamento de informações passadas pela *engine* do jogo.

3. Cinematografia

Cinematografia é uma ciência que vem se desenvolvendo há muitas décadas, e criou com o tempo o que chamamos de linguagem cinematográfica. Essa linguagem define um conjunto de regras para transmitir ao espectador uma certa emoção, dependendo da organização das tomadas. Isso inclui posicionamento da câmera, iluminação, tempo dos cortes e seqüenciamento das cenas. Por exemplo, em uma filmagem de um diálogo, os posicionamentos e transições de câmera dizem que tipo de emoção se quer transmitir. No caso, a câmera um pouco inclinada (movimento conhecido como *roll*) pode enfatizar uma desorientação do personagem ou um mistério na narrativa [Hawkins 2005].

Devido à popularidade do cinema e ao desenvolvimento da linguagem cinematográfica, hoje as pessoas acham essa forma de comunicação muito natural. Quando aplicada de forma incorreta, porém, pode deixar o telespectador confuso. Às vezes a intenção do diretor em um certo momento pode ser mesmo instaurar a confusão, mas caso contrário, pode trazer uma interpretação errada da cena ou da produção como um todo. O objetivo da cinematografia é deixar as ações mais coerentes ao telespectador, tornando sua experiência mais agradável.

Posicionar uma câmera real no ponto em que o diretor gostaria pode envolver o uso de braços mecânicos, trilhos e outros equipamentos especializados [Martin 1985]. Dependendo do tempo e do orçamento, pode acontecer que o diretor use uma

posição diferente da ideal. Felizmente esse tipo de problema não existe em ambientes 3D, mas mesmo assim algumas questões devem ser consideradas. Dentro de um ambiente 3D é possível posicionar e mover a câmera livremente. Mas vale lembrar que em ambientes virtuais de tempo real um planejamento das cenas e dos acontecimentos pode ser impossível, e isso pode acarretar alguns problemas. A existência de um obstáculo entre a câmera e o alvo pode representar um problema difícil de detectar e de prevenir. Toda a computação consumida para enquadrar o objeto na tomada pode ter sido em vão se existirem obstáculos que bloqueiam a visão. Deixar os obstáculos invisíveis pode ser uma solução, mas pode se tornar uma armadilha, pois afeta a consistência espacial da cena; deve ser usado com cautela. Outro problema que pode surgir é em relação à movimentação da câmera. Se ela passa por dentro de objetos, torna o ambiente irreal, e conseqüentemente incoerente para o telespectador.

Nos *games* a cinematografia pode ser usada da mesma forma que no cinema para as apresentações e passagens pré-definidas. Mas para o ambiente de jogo é preciso fazer algumas adaptações devido à sua natureza mais dinâmica e para não prejudicar a jogabilidade. Hoje em dia os posicionamentos de câmera, definindo como será a visão do jogador, são uma importante questão de usabilidade a ser considerada. Dependendo do grau de interatividade, os conceitos cinematográficos podem ser suprimidos ou adaptados em um certo grau. Por exemplo, nas primeiras versões do *game* Resident Evil (www.residentevil.com), a câmera era fixa mas colocada em pontos que já traziam alguma expectativa de ação ou dificuldade. Em outros jogos, como God of War (www.godofwar.com), a experiência cinematográfica se dá através da movimentação da câmera seguindo uma trajetória pré-definida dependendo da localização do personagem principal no cenário. Mas em jogos como Super Mario 64 (www.nintendods.com/sm64ds/), maximizou-se a interatividade e a cinematografia foi um pouco sacrificada. Neste jogo é possível controlar a câmera livremente, além do personagem. A Figura 2 mostra os jogos citados.

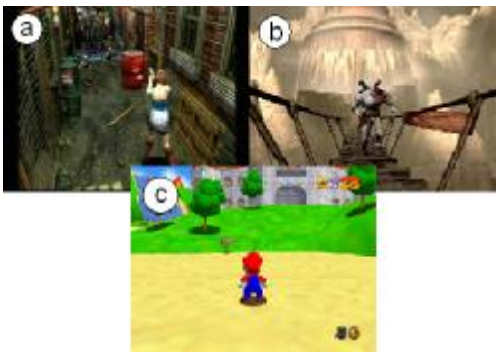


Figura 2: Cinematografia em jogos. (a) Resident Evil. (b) God of War. (c) Super Mario 64

Outra aplicação com alto apelo cinematográfico é a exibição dos acontecimentos do jogo para uma platéia, em tempo-real. Neste caso, os conceitos da cinematografia podem ser aplicados em totalidade, visando mostar a ação da forma mais atraente possível. Nota-se que as imagens geradas para o telespectador são bem diferentes das exibidas para o jogador; com essas imagens o jogo se torna impossível. As questões que aparecem são o reconhecimento de ações, a escolha da melhor ação e a exibição.

4. Câmeras Virtuais

A plataforma XNA, lançada pela Microsoft há aproximadamente 2 anos, se tornou muito popular no meio acadêmico para produção de jogos para Windows e X-box 360, e a cada dia ganha mais adeptos ao redor do mundo. Com o crescimento, surgiram diversos *websites* especializados na ferramenta, oferecendo desde tutoriais a classes e bibliotecas prontas. Existem hoje diversos trabalhos demonstrando diversas técnicas para implementação de câmeras em XNA. Entre eles destacam-se Riemers [2008] e Fegelein [2008], por explicar todos os fundamentos matemáticos em se tratando de manipulação de câmeras em ambientes 3D. Também surgiram muito bons livros sobre a ferramenta, tais como a publicação de Nitschke [2007].

Como o público-alvo do XNA são estudantes e curiosos, sua própria documentação é repleta de exemplos e ainda conta com explicações sobre os fundamentos de processamento gráfico e técnicas de programação.

Utilizando-se a ferramenta XNA, para renderizar uma cena é preciso uma coleção de objetos 3D que compõem a cena e um *shader* que é o responsável por desenhá-la. Este *shader* toma uma configuração de câmera como parâmetro. Normalmente, para desenhar a cena corretamente, todo objeto 3D deve ser passado ao *shader* usando a mesma configuração de câmera. Uma câmera é composta por 2 elementos principais: as matrizes 4x4 conhecidas como *view* e *projection*. Ambas são processadas pelo *shader* na GPU. A matriz *view* estabelece a orientação da câmera; ela contém informações como sua posição, definida por um ponto no espaço, o alvo, que é o ponto para onde a câmera aponta, e o vetor que representa a orientação vertical da câmera. A matriz *projection* estabelece como a câmera projeta sua visão para a tela; e contém informações como a abertura focal, a relação entre altura e largura da janela de exibição, e os limites (a distância) do que a câmera pode enxergar, conhecidos como *nearplane* e *farplane*. Será efetivamente exibido na tela tudo o que estiver dentro do volume definido pela matriz *projection*, conforme a Figura 3.

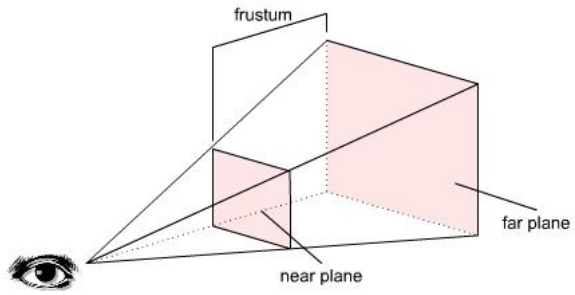


Figura 3: *Frustum* de visão

É possível adicionar outros atributos à câmera transformando-a em uma classe. Para que a câmera tenha propriedades físicas, basta adicionar membros como velocidade, massa e aceleração como atributos da classe. Também podem-se adicionar propriedades e métodos que facilitem seu uso, assim a manipulação de camera será mais eficiente e versátil [Fegelein 2008].

O XNA fornece um método muito útil para a manipulação de câmeras, Vector3.Transform(). Ele permite que um vetor seja facilmente rotacionado no espaço de acordo com o quatérnio ou matriz(de rotação) passada como parâmetro. Com este método a implementação de câmeras em primeira pessoa ou de câmeras que tentam manter uma distância fixa dos seus alvos é facilitada. O uso de quatérnios é recomendado quando é preciso armazenar rotações relativas ao objeto; são muito mais eficientes que armazenar rotações alinhadas com os eixos cartesianos [Riemers 2008]. Também são muito úteis para implementar movimentos relativos da câmera, tais como *yaw*, *pitch* e *roll*. Esses movimentos são originários da aviação mas que são muito usados em cinematografia. A Figura 4 ilustra o significado de cada um desses movimentos.

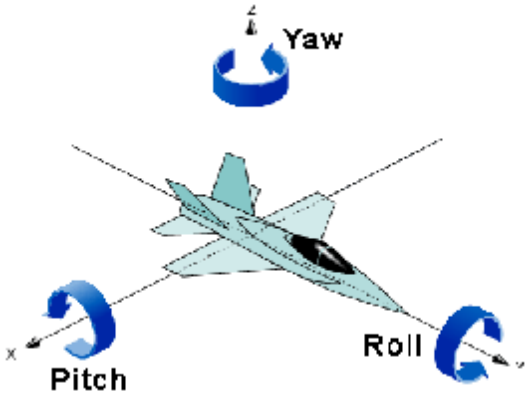


Figura 4: *Yaw*, *pitch* e *roll*

Para usar a câmera durante a programação do jogo, depois de todos os cálculos da lógica da classe câmera, é preciso informar ao *shader* as matrizes *view* e *projection*. O XNA fornece formas para agilizar o cálculo das matrizes view e projection, com os métodos Matrix.CreateLookAt() e Matrix.CreatePerspectiveFieldOfView(), respectivamente. Com isso, o programador só precisa se preocupar em em passar os parâmetros necessários.

5. O Jogo, o Ambiente e Física

Para ilustrar o uso da técnica de posicionamento de câmeras aproveitando resultados da simulação física do ambiente, foi produzido um jogo em XNA chamado Crystal Arena. Este jogo consiste em controlar uma espaçonave que desliza pelo solo, coletando cristais azuis e evitando cristais vermelhos. O jogador tem uma vista do terreno relativamente ampla, permitindo visualizar os objetos que estão na frente da nave. A câmera acompanha o movimento desta, mantendo uma distância fixa. A nave tende a se mover continuamente para a frente, e pode ser rotacionada em 90 graus para a esquerda ou 90 graus para a direita, seguindo as divisões contidas no solo, que é quadriculado. Quando a nave colide com uma bola branca, ela passa a se movimentar na direção contrária. A Figura 5 mostra a tela de jogo.

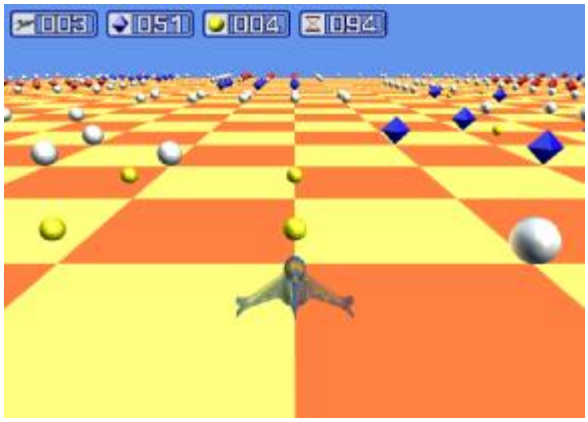


Figura 5: O jogo Crystal Arena

Uma fase de jogo é representada por uma matriz 32x32 contendo todos os elementos (cristais azuis, cristais vermelhos, tesouros e bolas brancas) nas interseções das linhas do cenário. O conteúdo da matriz vai sendo atualizado à medida que o jogo avança, ela representa o que existe no momento. Por exemplo, se um tesouro é coletado, o valor naquela posição da matriz muda para indicar que agora contém um espaço vazio.

O jogador também pode fazer uso de uma jogada especial, que consiste em, enquanto coletando cristais azuis que vão se tornando vermelhos, circundar um grupo de cristais azuis com cristais vermelhos, conforme a Figura 6. Ao concluir esta manobra, todos os cristais neste conjunto se transformam em tesouros. Neste momento a matriz que representa a fase também é atualizada, localizando as posições da matriz que representam aquele grupo de cristais, e mudando seus valores para que representem agora tesouros.

Quando o jogador perde uma vida ou passa de fase, existe a opção de visualizar o *replay* de sua jogada. Neste ponto a câmera pára de seguir a nave e entra no modo cinematográfico, onde os resultados da previsão física no ambiente são aproveitados para posicionar a câmera. A nave volta à posição inicial e a simulação

recomeça, mas desta vez sem a intervenção do usuário para movimentar a nave.

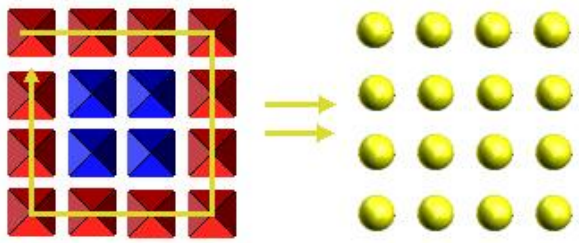


Figura 6: Transformação de cristais em tesouros

Devido à simplificação nos movimentos, a nave só pode colidir com objetos de frente ou de costas, sempre perpendicularmente, mas isso não inviabiliza os cálculos físicos e a técnica utilizada. Um jogador telespectador receberia as mesmas imagens que aquelas produzidas no *replay*. Porém, em um *replay*, os próximos estados do teclado, dos objetos ou do ambiente são conhecidos. Este fato não deve ser considerado nos algoritmos, para não prejudicar o caráter de previsão.

5.1 Reconhecendo Ações

Reconhecer as ações significa determinar corretamente o que está ocorrendo no ambiente virtual no momento. Em um jogo, ações mais objetivas ou pontuais fazem parte da própria *engine* do jogo ou fazem parte das propriedades ou máquina de estados de alguma classe. Neste caso, saber que uma ação ocorreu torna-se uma tarefa trivial. No jogo desenvolvido, os eventos que fazem parte da sua lógica são:

- Início da fase
- Nave se move para a frente ou para trás
- Curva à direita ou à esquerda
- Impacto com bola branca
- Impacto com cristal azul, que se torna vermelho
- Impacto com cristal vermelho, que destrói a nave
- Coleta de tesouro
- Conclusão de jogada especial
- Abrir a saída quando acabam os cristais azuis
- Alcançar a saída da fase
- Tempo esgotado

Às vezes uma modelagem de todas as ações pode ser muito difícil, devido à subjetividade de alguns eventos. Computacionalmente esses tipos de eventos são mais difíceis de reconhecer por causa da necessidade de uma interpretação dos acontecimentos ao longo do tempo. Para o domínio do jogo produzido, são ações interessantes:

- Nave começa a cercar cristais azuis para jogada especial
- Fazer manobras complicadas em espaços apertados
- Se aproximar da saída
- Chegar perto das extremidades da fase
- Nave se move em linha reta

Para reconhecer esses eventos pode ser desejável a utilização de um agente que fique monitorando as ações da nave e o estado da matriz da fase ao longo do tempo, e que avise ao sistema caso o padrão de movimento da nave e os objetos em volta formem um padrão que se assemelhe com os eventos desejados. Por exemplo, quando a nave fizer uma curva depois de tocar 2 cristais azuis nas extremidades de uma área prenchida por outros cristais azuis, então muito provavelmente trata-se de uma jogada especial. Se o jogador falha ao coletar os cristais necessários para a jogada especial, apenas o evento subjetivo "tentativa de jogada especial" será registrado; não existirá, porém, a ocorrência do evento de jogo "conclusão de jogada especial".

5.2 Previsão da Simulação Física

A previsão é usada para descobrir o que vai acontecer nos próximos intantes. Como no ambiente virtual um certo objeto pode estar sujeito a várias coisas, a previsão ajuda a determinar a probabilidade com que um dado acontecimento pode ocorrer ou como um dado acontecimento pode se concluir.

A previsão dos acontecimentos do ambiente virtual ajuda no reconhecimento de ações subjetivas, e é especialmente útil para posicionamento automático de câmeras. Por exemplo, em um ambiente virtual onde uma motocicleta salta por uma rampa, é possível, através de cálculos de física, descobrir onde e como ela vai cair, tomando assim as devidas providências para mostrar o fato da melhor forma possível. Logicamente, por meio da previsão, seria possível executar outros processos tais como fazer o piloto sair da motocicleta caso o salto se mostre desastroso. Previsão física também é útil para fins de melhoria da inteligência artificial. Por exemplo, na lógica de um lutador inteligente, que calcula seu próximo ataque de acordo com o tempo restante para que seu inimigo aterrisse depois de um salto.

Para prever o que pode acontecer no ambiente virtual podem-se usar vários artifícios, tais como cálculos de física ou outros cálculos, às vezes agilizados pelas próprias estruturas de dados usadas no sistema. Sempre é preciso considerar o que o objeto tende a fazer (como gravidade e outras velocidades) e os objetos que estão em volta, além de considerar como o usuário ou outros componentes podem intervir na cena. No jogo produzido, como a nave tende sempre a ir para a frente, a probabilidade de colisão à frente se torna maior conforme o tempo passa, já que sempre existirá algo à frente para colidir. Com isso o sistema estará cada vez mais preparado para filmar a colisão, que pode eventualmente não acontecer, devido à intervenção do usuário. Também, para a jogada especial, é muito provável que o jogador consiga concluí-la se já coletou mais de ¾ dos cristais

necessários para tal; a câmera estará preparada para filmar a jogada bem-sucedida.

5.3 Posicionando a Câmera

A partir das ações que foram reconhecidas e pelas inferências obtidas pela manipulação física, é possível posicionar a câmera e fazê-la se comportar segundo os ensinamentos de cinematografia. Todos os dados obtidos nas etapas anteriores serão processados por um agente diretor, que possui seu próprio estilo de filmagem. Como resultado o agente diretor altera diretamente as propriedades da classe câmera, que passa ao ***shader*** da aplicação a visualização a ser renderizada. A Figura 7 mostra um esquema de funcionamento.

Figura 7: Processo de posicionamento de câmera

Como mostrado na Figura 7, o posicionamento de câmera é resultado de todas as etapas anteriores. A *engine* de jogo informa os eventos relativos à lógica do jogo ou as propriedades e máquinas de estados dos objetos. Com base nessas informações, é possível que eventos subjetivos sejam reconhecidos, e que próximos eventos sejam previstos. Então o agente diretor pode posicionar a câmera de acordo com a linguagem cinematográfica utilizada, pois ele já tem acesso a todas as informações que ele necessita. Também existe um fator de aleatoriedade na escolha da posição da câmera, para que o processo não se torne totalmente determinístico. Dadas de uma a cinco posições aceitáveis, uma é escolhida. Com isso o *replay* pode ser visto diversas vezes, cada uma com uma configuração de câmera diferente. Após isso, a cena é renderizada, e a partir daí o processo se repete até que aconteça outro evento ou o agente diretor entenda que agora existe uma melhor posição de câmera. Finalmente, como o usuário tem o controle de parte do ambiente virtual (o personagem que ele controla), outros eventos podem ocorrer devido à sua intervenção.

Segundo Hawkins [2005], as principais questões que devem ser consideradas ao posicionar a câmera no ambiente virtual são:

- Enquadramento: Com o uso da plataforma XNA, é possível definir o alvo da câmera como sendo o centro da nave, por exemplo, para fazer a câmera segui-la automaticamente, independente da posição da câmera. Da mesma forma, pode-se programar a câmera para focar em qualquer ponto ou objeto do cenário. No entanto, enquadrar o objeto de forma que ele fique muito grande ou muito pequeno na cena tem seu impacto para o telespectador. Métodos tradicionais consistem em calcular a razão entre a área da tela e a área da projeção da *bounding box* ou *bounding sphere* do objeto focado. No ambiente virtual produzido, como os tamanhos de todos os objetos já são conhecidos, para simplificar os cálculos basta afastar ou aproximar a câmera em fatores já conhecidos.

- Oclusão: Em ambientes virtuais, pode ser muito difícil detectar quando um objeto está bloqueando a visão. Métodos tradicionais compreendem lançar raios até o alvo da câmera e avaliar se os raios interceptam algum outro objeto que está entre a câmera e o alvo. No jogo produzido, porém, sabe-se que não existe nenhum objeto acima de certa altura, e como o espaço foi todo discretizado, é possível, consultando-se a matriz da fase, saber se em uma posição (x, z) do cenário existe algum objeto. Como solução pode-se mover a câmera para um outro local, ou deixar todos os objetos que atrapalham a visão invisíveis. Essa segunda opção pode ser perigosa pois leva à inconsistência espacial.

- Consistência espacial: Existem algumas regras cinematográficas para manter a consistência espacial, isto é, não deixar o telespectador ficar confuso quanto ao posicionamento dos componentes da cena em função de um mal posicionamento de câmera. A Figura 8 mostra um exemplo.

A linha tracejada na Figura 8 é conhecida como linha de ação. Para que a cena fique consistente, é desejável que todas as tomadas de câmera fiquem do mesmo lado do espaço definido pela linha de ação [Hawkins 2005]. A violação desta regra (Figura 8.b) dá a impressão que os personagens trocam de lugar durante a conversa, e pode deixar o telespectador confuso. Para o jogo produzido, para manter a consistência espacial, a câmera tende a ser posicionada de tal forma que sua componente Z no espaço fique sempre entre a componente Z da nave e zero, que é a componente Z do ponto de início da fase. Isto é, a posição Z da câmera é sempre maior que a da nave. Também, não existem cortes ou movimentos de câmera enquanto a nave está fazendo uma curva, colidindo com uma bola branca, ou está muito próximo da saída da fase (o único momento que a componente Z da câmera pode ser maior que a da nave). Se a

câmera está se movendo nesse momento, ela pára imediatamente.

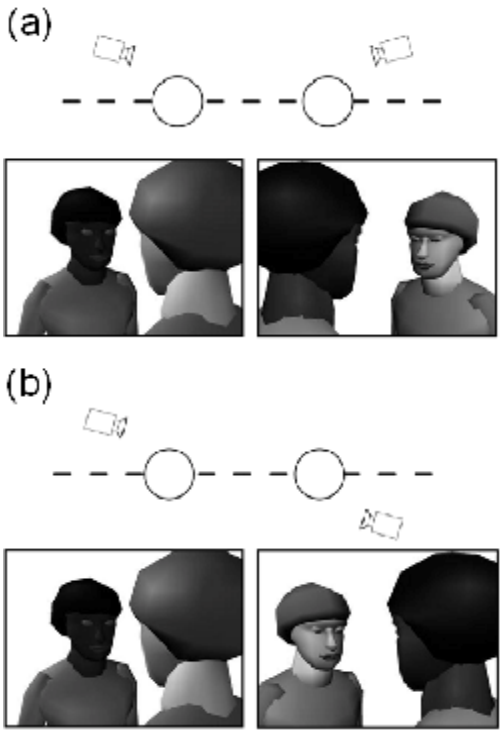


Figura 8: (a) Cena com consistência espacial
(b) Cena inconsistente.

Usando conceitos da linguagem cinematográfica, segundo Martin [1985], também existem outras restrições para dar mais dramacidade à ação, aplicadas ao contexto do jogo:

- Se a nave está muito próxima de uma região onde é possível executar a jogada especial, a câmera tende a ir para um ponto em que pode enquadrar todos os cristais do grupo e mostrar sua transformação.
- Se o usuário erra a jogada especial, a situação pode se tornar perigosa, e a câmera deve se aproximar.
- Se a nave está numa situação segura (evento reconhecido quando existem poucos cristais ou bolas brancas próximas da nave), a câmera fica mais longe e mostra todo o cenário.
- Quando existem muitos obstáculos em espacos apertados, existe muito perigo de colisão. Neste caso a câmera fica sempre perto da nave, e o tempo da tomada é bastante curto.
- Quando a nave está rodeada de muitos cristais vermelhos, existe muito perigo. Neste caso a câmera fica sempre perto da nave, e o tempo da tomada é bastante curto. Pode-se executar então o movimento de sutilmente inclinar a câmera (movimento de *roll*) para enfatizar o perigo [Hawkins 2005].

Exceto o fato de a saída se abrir, nenhum outro evento acontece muito longe do objetivo principal da ação, que é a nave em movimento. Portanto, em quase todas as tomadas a nave passará na frente da câmera em algum momento, mas isso não significa que o alvo será sempre fixo na nave. Principalmente nas situações de perigo, a câmera pode ter como alvo uma bola branca em que existe alta probabilidade de colisão. Neste momento, provavelmente a nave irá passar, colidindo ou desviando a tempo.

Outro fator que deve ser considerado é o tempo das tomadas. Hoje em dia, o tempo em que uma tomada dura, para cenas de ação e/ou reflexo é de 2 segundos [Hawkins 2005]. Isso ajuda a criar no telespectador a mesma sensação de tensão vivida pelo protagonista ou jogador. Quanto mais confortável for a situação da nave e quanto mais afastada estiver a câmera do solo, mais longa será a tomada, indicando tranqüilidade.

5.4 Resultados

No jogo produzido, o *replay* é construído por meio de uma estrutura de dados que armazena todos os comandos do jogador ao longo do tempo. Existe uma variável responsável por contar o tempo de jogo restante ao jogador, que é atualizada a cada *frame*. Uma entrada do jogador é armazenada juntamente com o número de *game-loops* passados desde o início da fase. Esta abordagem é mais robusta do que armazenar o tempo passado em milissegundos, pois o *replay* pode se tornar defasado em função da execução de outros processos no sistema, como a própria técnica para geração do *replay*.

Para não comprometer a intenção da previsão, apenas a posição atual da lista de comandos deve ser observada, simulando uma situação de jogo em tempo-real. Elementos que representam as próximas entradas do jogador não serão considerados. Para o reconhecimento de ações, não considera-se os comandos anteriores, mas sim os eventos anteriores, tais como virar, colidir etc. Para isso, durante o *replay*, uma outra estrutura de dados é utilizada. Ela armazena os últimos 20 eventos da fase, juntamente com a posição da nave naquele momento. A partir daí o programa faz suas previsões e posiciona a câmera. Com a adição dessas variáveis, o consumo de memória aumenta em média 5 Kb, o que representa uma porcentagem muito pequena do consumo do jogo inteiro.

A diferença de desempenho entre o jogo e o *replay* está relacionada com os processos que são executados a mais para posicionar a câmera corretamente. O único processo que não está presente na hora do *replay* é a exibição dos painéis informativos (HUD – *Head Up Display*), o que representa aproximadamente 10% de todo o trabalho de renderização. A entrada de dados é substituída pela leitura da lista de comandos, que é mais custosa para o sistema. A Figura 7 mostra um ciclo completo do *game-loop*, que é executado no momento do *replay*. Durante a interação com o jogador, as etapas de reconhecimento de ação, previsão

física e posicionamento da câmera não estão presentes. Todas essas etapas podem representar um acréscimo de até 50% no processamento da aplicação como um todo. A diferença foi medida através do *frame rate* do jogo em execução.

Outro fato observado foi que, durante o estado em que o jogador pode controlar a nave, existe um processamento muito maior quando a nave colide com um cristal azul, pois é preciso testar se a jogada especial foi concluída. Em caso positivo, ainda existe o trabalho de transformar os ítens da matriz da fase. Porém, durante o *replay*, a colisão com um cristal azul não aumenta tanto o trabalho de processamento, pois já existem muitos outros processos acontecendo nesse momento. O gargalo de processamento se transforma da renderização durante o jogo, para a lógica de posicionar a câmera dinamicamente durante o *replay*.

6. Conclusões

Este trabalho mostrou uma solução para posicionamento de câmeras, a partir de informações passadas pela *engine* de jogo, usando as técnicas de reconhecimento de ações e previsão física dentro do ambiente virtual, além de conceitos de cinematografia. Foi mostrado como funciona a plataforma XNA, com a qual o jogo para exemplo foi construído, e algumas técnicas cinematográficas que foram seguidas.

Todas as adaptações no ambiente virtual e na dinâmica do jogo foram mostradas de forma que a técnica pudesse ser aplicada corretamente e de forma otimizada. Outro fator fundamental para o sucesso da técnica foi a adição de um fator de aleatoriedade, que evita o determinismo e deixa o resultado final mais interessante para o telespectador.

De acordo com as experiências tomadas e resultados alcançados, conclui-se que a utilização da técnica é viável, tanto para a geração de *replays*, quanto para a transmissão em tempo-real para um grupo de telespectadores, como explicado em [Drucker 2005].

Referências Bibliográficas

AMERSON, D. E KIME, S., 2000. *Real-time Cinematic Camera Control for Interactive Narratives*.

DRUCKER, S., 1994. *Intelligent Camera Control for Graphical Environments*. Tese de Doutorado, Massachusetts Institute of Technology.

ERLEBEN, K., 2002. *Module Based Design for Rigid Body Simulators*.

FEGELEIN, 2008. *Microsoft XNA Framework; Creating a Freelook Camera* [online]. Disponível em: http://www.fegelein.com/?p=18 [Acessado em 8 de agosto de 2008].

HAWKINS, B., 2005. *Real-Time Cinematography for Games*. Editora Charles River Media, 2005.

HE, L. ET AL., 1996. *The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing*.

HERMANN, R. E CELES, W., 2005. *Posicionamento Automático de Câmeras em Ambientes Virtuais Dinâmicos*.

MARCHAND, É. E COURTY, N., 2002. *Controlling a Camera in a Virtual Environment*.

MARTIN, M., 1985. *A Linguagem Cinematográfica*. Editora Brasiliense, 1985.

MNEMOCINE, 2008. *Linguagem e técnica cinematográfica* [online]. Disponível em: http://www.mnemocine.com.br [Acessado em 8 de agosto de 2008].

NITSCHKE, B., 2007. *Professional XNA Game Programming for Xbox 360 and Windows*. Editora Wrox, 2007.

PINHANEZ, C., 1999. *Representation and Recognition of Action in Interactive Spaces*. Tese de Doutorado, Massachusetts Institute of Technology.

RIEMERS, 2008. *Quaternion Camera* [online]. Disponível em: http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2/Quaternions.php [Acessado em 8 de agosto de 2008].

SEUGLING, A. E RÖLIN, M., 2006. *Evaluation of Physics Engines and Implementation of a Physics Module in a 3D Authoring Tool*. Tese de Mestrado, Umea University.

Algoritmos de busca em tempo real aplicados a jogos digitais

Eder L. Trindade Ricardo P. Martins Ferreira Eduardo P. C. Fantini Hugo B. de Paula

Pontifícia Universidade Católica de Minas Gerais

Resumo

Restrições de tempo de resposta e de recursos computacionais oferecem um desafio para o emprego de técnicas sofisticadas de Inteligência Artificial nos jogos. Neste artigo estudamos como aprimorar os algoritmos de busca em tempo real encontrados na literatura. Em especial, investigamos como permitir que vários agentes se desloquem simultaneamente no ambiente de um jogo. Também foi feito um estudo de caso em um jogo real, o *Ultima Online,* para avaliar o comportamento desses algoritmos em jogos digitais.

Palavras Chaves: Inteligência Artificial, Agentes, *Real-time Search*, *pathfinding*, RTA*, *Games*

Contatos dos autores:
eder.lucio.trindade@gmail.com
{poley,hugo}@pucminas.br
eduardofantini@gmail.com

1. Introdução

Os jogos digitais são ambientes propícios para a aplicação de técnicas de inteligência artificial (IA). Por um lado, estes ambientes limitados, onde as regras costumam ser bem definidas, servem como laboratórios para testes [Waveren, 2001]. Por outro lado, a IA ainda tem muito a contribuir para o desenvolvimento dos jogos. Restrições de tempo de resposta e de recursos computacionais oferecem um desafio para o emprego de técnicas sofisticadas de IA em jogos digitais.

O planejamento de trajetória é um dos problemas presentes nos jogos onde técnicas de inteligência artificial são usadas com sucesso. Em muitas situações os movimentos de um agente/personagem não podem ser tratados como deslocamentos simples controlados pelo jogador. Nestes casos, é útil o emprego de algoritmos para o mapeamento do ambiente em um grafo e de algoritmos de busca de caminhos entre a posição atual no grafo e a posição desejada do agente no jogo [Nonnenmacher et al, 2007].

A maioria dos jogos digitais resolve o problema de busca de caminhos com soluções baseadas no algoritmo A* [Millington, 2006]. O algoritmo A* é um algoritmo informado de busca ótima, *off-line*. Nos algoritmos de busca *off-line* [Russell and Norvig, 2004], a fase do planejamento – mapeamento do ambiente e busca no grafo – antecede a fase da execução, quando o agente se desloca pelo caminho escolhido. Os algoritmos de busca *off-line* nem sempre são capazes de atender as necessidades de tempo de resposta em um jogo. Quando um personagem fica imóvel aguardando o cálculo de uma trajetória, a cadência do jogo pode ficar prejudicada. Assim, uma decisão rápida para o próximo movimento, mesmo que não ótima, em algumas situações é mais importante do que esperar o cálculo do melhor movimento. Estas dificuldades ficam evidentes quando é necessário calcular simultaneamente a trajetória de diversos personagens.

Entre as soluções adotadas na literatura para lidar com os requisitos de tempo de resposta para o problema de encontrar caminhos, os algoritmos de busca *on-line*, conhecidos como algoritmos de busca em tempo real são freqüentemente adotados [Russell and Norvig, 2004]. Apesar de não tratarmos características relacionadas a tempo, adotamos o termo "tempo real" por ser o termo utilizado na literatura. Os algoritmos de busca em tempo real são algoritmos onde as fases de planejamento e de execução são alternadas. O agente planeja e executa alternadamente os movimentos até alcançar o objetivo. Embora não sejam ótimos, se algumas condições forem atendidas, como condições de conexidade do grafo, eles conseguem orientar o agente até o objetivo, mesmo quando o grafo sofre alterações durante o deslocamento do agente. Este comportamento é relevante em jogos digitais porque, muitas vezes, o ambiente é dinâmico.

Neste artigo estudamos como aprimorar os algoritmos de busca em tempo real encontrados na literatura. Em especial, investigamos como permitir que vários agentes se desloquem simultaneamente no ambiente de um jogo. Realizamos um estudo comparativo entre algoritmos de busca em tempo real presentes na literatura e nossas adaptações. Também foi feito um estudo de caso em um jogo real, o *Ultima Online,* para avaliar o comportamento de alguns algoritmos de busca em tempo real. O *Ultima Online* é um jogo digital do tipo MMORPG (*Massive Multiplayer Online Role-Playing Game*) em terceira pessoa [Origin, 2008], o qual possibilita a interação do jogador com um complexo ambiente virtual. Os resultados obtidos nos experimentos quantitativos e qualitativos realizados indicam a eficácia do uso de algoritmos de busca em tempo real em jogos digitais.

O restante do artigo é organizado da seguinte forma. A próxima seção apresenta alguns dos principais algoritmos de busca em tempo real. A Seção 3 apresenta as adaptações propostas e na Seção 4 são apresentados os resultados computacionais e as alterações feitas no emulador do *Ultima Online*, o *RunUO*. Na última seção são apresentadas algumas conclusões e oportunidades para trabalhos futuros.

2. Algoritmos de busca em tempo real - Trabalhos Relacionados

Os algoritmos de busca em tempo real propõem uma solução de compromisso entre a qualidade da decisão de qual é o próximo estado que um agente deve visitar e o tempo para calcular esta decisão [Koenig, 2006]. Existem vários artigos que sugerem a implementação de algoritmos de busca em tempo real em jogos digitais [Koenig, 2006][Koenig, 2007][Sun, 2008]. Entretanto, estes artigos não apresentam nenhuma implementação em um ambiente de jogo real.

A busca em tempo real foi motivada primeiramente pela robótica sensível a tempo [Koenig, 1999] e em seguida por jogos digitais, abrangendo os jogos de RTS (*real-time strategy*), de FPS (*first-person shooter*) e de RPG (*role-playing games*). Em todos eles o tempo desempenha um papel fundamental, uma vez que vários agentes fazem buscas de caminhos simultaneamente e é exigido do jogo um tempo de resposta muito rápido para garantir o *gameplay* [Bulitko et al., 2007]. Encontrar caminhos em jogos é computacionalmente muito caro. No jogo RTS "*Age of Empires II*", por exemplo, o *pathfinding* consome entre 60 e 70% do tempo de simulação [Pottinger, 2000].

O problema de encontrar caminhos em grafos é representado da seguinte forma: considere um Grafo *G(V, E)* onde *V* representa um conjunto de vértices, *E* representa um conjunto de arestas, *I* representa o vértice inicial e *O* representa o conjunto de vértices finais. Um caminho é um subconjunto conexo do grafo *G*. O problema consiste em determinar um caminho de *I* para *O*.

Nas subseções seguintes descrevemos dois algoritmos de busca em tempo real clássicos, *Learning Real-Time A** e o *Real-Time A** [Korf 1990].

2.1 Learning Real-Time A* - LRTA*

O algoritmo LRTA* [Korf 1990] (*Learing Real-Time A**) é um algoritmo de busca informada e em tempo real. Ele não é ótimo, ou seja, não garante que será encontrado o melhor caminho entre o nó inicial e o nó objetivo. O algoritmo possui conhecimento da localização do vértice objetivo e utiliza esta informação para eleger a próxima posição do agente. No contexto onde realizamos os experimentos utilizamos a distância *Manhattan* como medida heurística da distância até o objetivo.

O LRTA* inicia com um agente localizado no vértice inicial. No passo seguinte, o agente se desloca para o vizinho mais promissor de acordo com a função de custo. A busca termina quando o agente encontra um vértice objetivo. A cada iteração o agente atualiza a estimativa da distância até o objetivo melhorando a informação sobre o grafo. Na fase de planejamento o agente calcula para os nós vizinhos a função de custo

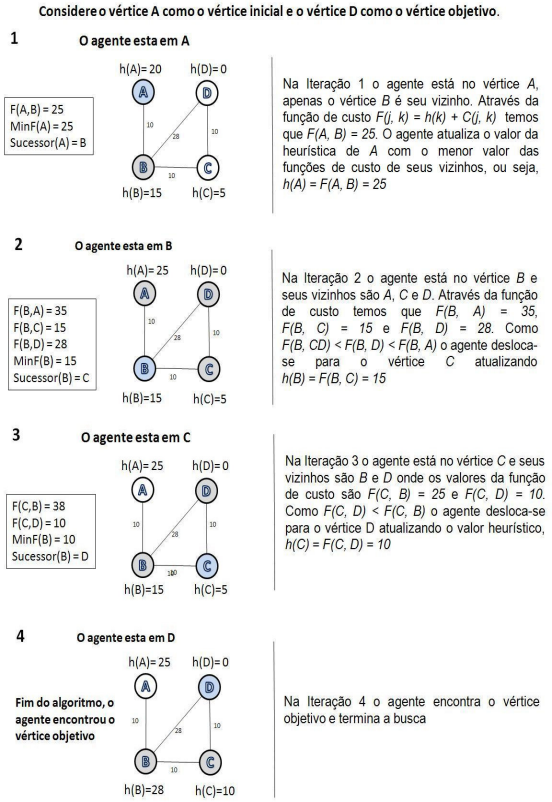


Figura 1: Execução do LRTA*

Constantes e funções:

I	*vértice inicial*
O	*vértice objetivo*
MinF(j)	*função que retorna o valor do menor custo estimado dos caminhos que partem de j até o objetivo*
Sucessor(j)	*função que retorna o vértice vizinho de j cujo custo estimado do caminho até o objetivo passando por ele é o menor*
Heuristica(j)	*função que retorna a heurística da distância do vértice j até o objetivo.*

Variáveis:

j	*vértice corrente*
h(j)	*valor da heurística do vértice j*
V	*conjunto de vértices do grafo*

Procedimento:

```
[01]  Para cada j em V faça
[02]        h(j) = Heuristica (j)
[03]  j = I
[04]  Enquanto (j != O) faça
[05]  {
[06]        h(j) = MinF(j)
[07]        j = Sucessor(j)
[08]  }
```

Figura 2: Pseudocódigo do LRTA*

F(j, k) = h(k) + C(j, k) onde *k* é um vértice vizinho, *j* é o vértice corrente, *h(k)* representa o valor atual da heurística de *k* até o objetivo *O* e *C(j, k)* é o custo da aresta do vértice corrente *j* até *k*. O agente atualiza a heurística do vértice corrente com o menor valor de *F(j, k)* e move-se para ele. O processo então reinicia.

A Figura 1 ilustra um exemplo de execução do LRTA*. O nó inicial é o vértice A e o nó objetivo é o vértice D. O algoritmo calcula a heurística de cada vértice, consideremos o seguinte estado corrente: *h(A) = 20, h(B) = 15, h(C) =5 e h(D) = 0.*

2.2 Real-Time A* - RTA*

O Algoritmo RTA* [Korf 1990] (*Real Time A**) é um algoritmo de busca por qualquer caminho e em tempo real semelhante ao LRTA*. A diferença entre o LRTA* e o RTA* é que o RTA* atualiza a heurística do vértice corrente com o segundo melhor valor de *F(j, k)* mantendo uma informação mais precisa do grafo. A Figura 3 apresenta o pseudocódigo do RTA*.

Constantes e funções:

I	*vértice inicial*
O	*vértice objetivo*
SegundoMinF(j)	*função que retorna o valor o segundo menor custo estimado dos caminhos que partem de j até o objetivo*
Sucessor(j)	*função que retorna o vértice vizinho de j cujo custo estimado do caminho até o objetivo passando por ele é o menor*
Heuristica (j)	*função que retorna a distância de heurística do vértice j até o objetivo.*

Variáveis:

j	*vértice corrente*
h(j)	*valor da heurística do vértice j*
V	*conjunto de vértices do grafo*

Procedimento:

```
[01]  Para cada j em V faça
[02]        h(j) = Heuristica (j)
[03]  j = I
[04]  Enquanto (j != O) faça
[05]  {
[06]        h(j) = SegundoMinF (j)
[07]        j = Sucessor(j)
[08]  }
```

Figura 3: Pseudocódigo do RTA*

3. Variações propostas dos algoritmos

Neste trabalho estudamos algoritmos de busca em tempo real e propomos novos algoritmos adotando duas estratégias:

- Investigamos o efeito nos algoritmos da falta de informação sobre o ambiente de busca. Propomos uma variação no algoritmo RTA* melhorando o aprendizado do agente sobre o ambiente;
- Estudamos diversos agentes no processo de busca. Propomos um algoritmo baseado em sistemas multiagentes onde agentes compartilham informação, o que pode agilizar o processo de busca.

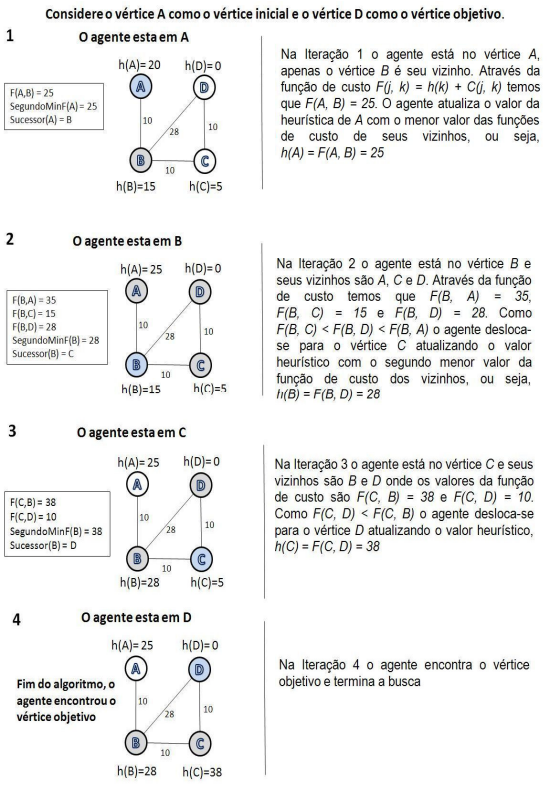


Figura 4: Execução do RTA*

A seguir apresentamos dois algoritmos o *Real Time A* plus,* onde melhoramos o aprendizado do algoritmo *Real Time A** e o *Relay Real Time A**, onde mais de um agente cooperam para agilizar o processo de busca.

3.1 Real-Time A* Plus – RTA*P

O *Real-Time A* Plus* (RTA*P) é uma variação do algoritmo *Real-Time A**, onde a novidade é acrescentar mais informação ao algoritmo. No RTA* o algoritmo analisa todos os vizinhos do vértice corrente para escolher o vértice mais promissor e atualizar a heurística do vértice corrente com o segundo melhor valor de custo. Contudo esse algoritmo descarta as informações previamente calculadas sobre os outros vizinhos. O algoritmo RTAP armazena também qual é o vizinho do segundo melhor valor de custo e qual é o valor do terceiro melhor custo. Caso o agente retorne a um vértice já expandido o agente desloca-se para o vértice de segundo melhor custo e atualiza o valor da heurística do vértice corrente com o terceiro melhor custo, sem a necessidade de realizar nova análise dos vizinhos.

O RTA*P ilustra o comportamento do agente quando aumentamos seu aprendizado sobre o ambiente. Comparamos o RTA*P e o RTA* em diversas situações explorando ambientes com e sem informação. O RTA*P inicia com um agente localizado no vértice inicial, no passo seguinte o agente se desloca para o vizinho mais promissor. A busca encerra quando o agente encontra um vértice objetivo.

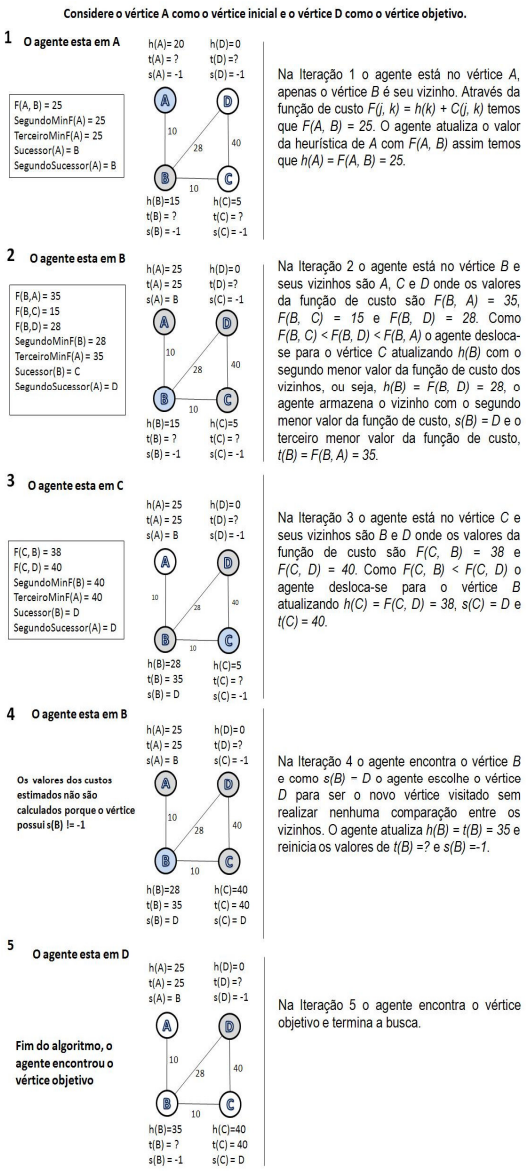


Figura 5: Execução do RTA*P

A cada iteração o agente atualiza a estimativa heurística do vértice. Na fase de planejamento o agente calcula para todos seus vizinhos a função de custo $F(j, k) = h(k) + C(j, k)$ onde j é o vértice corrente, k é um vértice vizinho de j, $h(k)$ representa o valor da heurística da distância de k até O e $C(j, k)$ é o custo da aresta do vértice corrente j até o vizinho k. O algoritmo também armazena qual o vértice $s(j)$ possui o segundo menor valor da função de custo e o valor do terceiro menor valor da função de custo, $t(j)$. Se o vértice j ainda não foi visitado o agente atualiza a heurística do vértice corrente com o segundo menor valor de $F(j, k)$ e move-se para o vizinho de menor $F(j, k)$. Se o vértice k já foi visitado o agente não calcula a função de custo para os vizinhos de k e caminha para o vértice $s(k)$ atualizando a heurística $h(k) = t(k)$. A Figura 5 apresenta um exemplo de execução do RTA*P. A Figura 6 apresenta o algoritmo RTA*P.

Constantes e funções:

I	*vértice inicial*
O	*vértice objetivo*
SegundoMinF(j)	*função que retorna o valor do segundo menor custo estimado dos caminhos até o objetivo que partem de j*
TerceiroMinF(j)	*função que retorna o valor do terceiro menor custo estimado dos caminhos até o objetivo que partem de j*
Sucessor(j)	*função que retorna o vértice vizinho de j cujo custo estimado do caminho até o objetivo passando por ele é o menor*
SegundoSucessor(j)	*função que retorna o vértice vizinho de j cujo custo estimado do caminho até o objetivo passando por ele é o segundo menor*
Heuristica (j)	*função que retorna a distância de Manhatam do vértice j até o objetivo.*

Variáveis:

j	*vértice corrente*
h(j)	*valor da heurística do vértice j*
s(j)	*vizinho de j com a segundo menor custo estimado*
t(j)	*valor da heurística do vizinho de j com o terceiro menor custo estimado*
vizinho(j)	*conjunto de vértices vizinhos de j.*
V	*conjunto de vértices do grafo*

Procedimento:

```
[01]  Para cada j em V faça
[02]  {
[03]        h(j) = Heuristica (j), t(j) = 0, s(j) = -1
[04]  }
[05]  j =I
[06]  Enquanto (j ≠ O) faça
[07]  {
[08]        se s(j) ≠ -1 então
[09]        {
[10]              j = s(j), h(j) =t(j), s(j) = -1
[11]        }
[12]        Senão
[13]        {
[14]              h(j) = SegundoMinF (j)
[15]              t(j) = TerceiroMinF (j)
[16]              s(j) = SegundoSucessor (j)
[17]              j = Sucessor(j)
[18]        }
[19]  }
```

Figura 6: Pseudocódigo do RTA*P

3.3 Relay Real-Time A* - RRTA*

O *Real Time A** (RRTA*) é um algoritmo de busca informada em tempo real baseado em sistemas multiagentes [Yokoo and Ishida, 1999] [Wooldridge, 2002]. O RRTA* é formado por dois ou mais agentes que dividem informação para encontrar um caminho entre dois nós. Vamos discutir neste artigo o caso com apenas dois agentes. Chamamos o primeiro agente de A e o segundo agente de B. O agente A é o agente responsável por encontrar um caminho entre o nó inicial e o nó final passando pelo nó origem de B. O agente B tem o objetivo de encontrar um caminho entre um nó qualquer do grafo que é seu nó inicial e o nó objetivo. O nó de origem de B deve pertencer ao caminho de A. Assim, o agente B antecipa o conhecimento sobre o grafo para A. Neste artigo vamos utilizar como estratégia para a escolha da posição inicial de B o ponto médio da semi-reta entre o nó inicial e o nó objetivo.

O RRTA* é formado por duas esferas de decisão: uma esfera tática onde o algoritmo coordena os agentes e uma esfera operacional onde os agentes executam a busca.

Na esfera tática o agente A começa a busca no nó inicial. O algoritmo elege um vértice para B iniciar a busca, neste caso, o ponto médio da semi-reta entre o nó inicial e nó objetivo. O agente B tem a função de encontrar o nó objetivo e A, a priori, tem o objetivo de encontrar o vértice por onde B iniciou a busca. Quando o agente A encontrar o nó por onde B partiu ou algum vértice que foi caminho de B o agente A passa a procurar o nó objetivo. A busca termina quando o agente A encontra o nó objetivo.

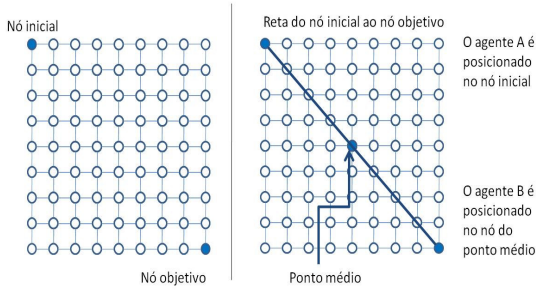


Figura 7: Estratégia de posicionamento dos agentes - RRTA* - Ponto Médio – Dois Agentes

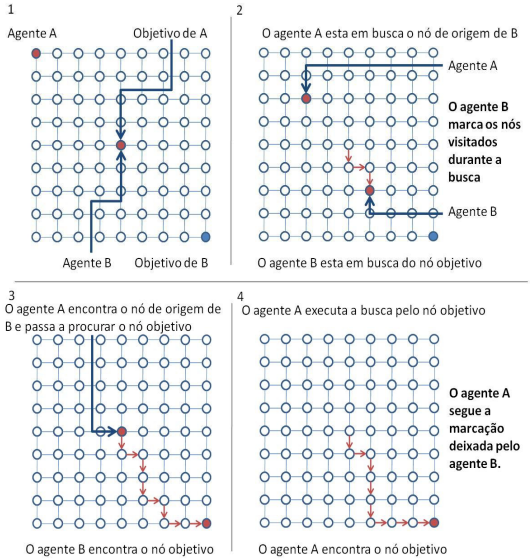


Figura 8: Execução do RRTA*

Na esfera operacional, ambos agentes utilizam o algoritmo RTA* para realizar a busca. O agente B realiza a busca a procura do nó objetivo marcando os nós por onde passou, o agente B marca no nó visitado qual é o próximo nó que ele irá visitar. O Agente A realiza a busca e quando encontra um nó visitado por B segue as marcações deixadas por B sem fazer qualquer verificação, assim o agente A utiliza do conhecimento do grafo adquirido na busca de B. A Figura 7 ilustra a definição das posições inicias dos agentes.

Nos exemplos adotados neste trabalho o algoritmo traça uma semi-reta entre o nó inicial e o nó objetivo.

O algoritmo calcula o ponto médio desta semi-reta. As coordenadas do ponto médio são definidas pela equação: $X = \frac{x1+x2}{2}, Y = \frac{y1+y2}{2}$ onde X representa a ordenada do ponto médio, $x1$ a ordenada do nó inicial e $x2$ a ordenada do nó objetivo, Y representa a abscissa do ponto médio, $y1$ representa a abscissa do nó inicial e $y2$ representa a abscissa do nó objetivo.

Constantes e funções:

I	*vértice inicial*
O	*vértice objetivo*
Estrategia(x, I, O)	*função que posiciona o agente x, esta função é responsável pela estratégia de posicionamento*
RTA(x)*	*função que executa uma iteração da busca RTA* para o agente x*
Getcorrente(x)	*função que retorna o vértice corrente do agente x*
SetObjetivo(x, j)	*função que substitui o vértice objetivo do agente x por j*
GetObjetivo(x)	*função que retorna o vértice objetivo temporário do agente x*
PontoMarcado(j, x)	*função que retorna se o vértice j esta marcado pelo agente x*
Sucessor(x)	*função que retorna o agente que x persegue*

Variáveis:

j	*vértice corrente*
x	*agente corrente*
A	*Conjunto de Agentes*

Procedimento:

```
[01]  Para cada agente x ∈ A faça
[02]        Estrategia(x, I, O)
[03]  Enquanto existir (Getcorrente (x) != O ∀ x ∈ A) faça
[04]  {
[05]        Para cada agente x ∈ A faça
[06]        {
[07]              RTA*(x)
[08]              se(Getcorrente (x) = GetObjetivo(x)) ou
                  (PontoMarcado (Getcorrente (x), Sucessor(x) ))
[09]                    Setobjetivo (x, GetObjetivo(Sucessor(x)))
[10]              Se Getcorrente (x) = O
[11]                    A = A - x
[12]        }
[13]  }
```

Figura 9: Pseudocódigo do RRTA*

A estratégia para posicionamento dos agentes depende muito do tipo do grafo. Apresentamos um critério mais didático, contudo outros critérios podem ser adotados. A Figura 9 apresenta o pseudocódigo do algoritmo RRTA*, o procedimento RTA* além de executar uma iteração da busca RTA* marca os nós por onde B passou e o agente A quando encontra um nó visitado pelo agente B segue a marcação sem analisar os vizinhos. O primeiro passo do algoritmo é posicionar os agentes, o agente A é posicionado no nó inicial e o agente B é posicionado no nó escolhido. O algoritmo atribui ao agente A a função de encontrar o nó por onde B iniciou a busca e ao agente B a função de encontrar o nó objetivo. No segundo passo os agentes executam a busca utilizando o algoritmo RTA*, o agente B durante a busca deixa marcas que depois poderão ser seguidas pelo agente A.

Quando o agente A encontra o ponto por onde B iniciou a busca ele passa a procurar o nó objetivo. O agente B termina sua busca quando encontra o nó objetivo, quando encontra com o agente A ou quando o agente A encontra o nó objetivo. Quando o agente A encontra as marcas deixadas pelo agente B ele passa a segui-las, assim o agente A aproveita o conhecimento do grafo adquirido por B durante sua busca. A busca termina quando o agente A encontra o nó objetivo. A Figura 8 apresenta a execução do RRTA*.

Realizamos experimentos como o RRTA* apenas em grafos conexos, a estratégia de posicionamento dos agentes utilizando o ponto médio deve ser usada em somente em grafos conexos.

4. Experimentos Computacionais

Foram realizados dois grupos de experimentos. O primeiro grupo trata de um estudo comparativo entre os diferentes algoritmos propostos na literatura e os novos algoritmos propostos neste trabalho em um ambiente estruturado. São ambientes típicos para testes de algoritmos de busca [Koenig, 2006][Koenig, 2007][Sun, 2008]. O segundo grupo trata de experimentos qualitativos onde algoritmos de busca *on-line* são aplicados em um jogo MMORPG - o *Ultima Online* - para avaliar seus comportamentos em um ambiente de jogo real.

Os ambientes estruturados são labirintos representados em grafos 4-conexos gerados aleatoriamente de tamanho entre 100 a 1.000.000 vértices. São labirintos quadrados de tamanhos entre 10 x 10 a 1000 x 1000. Escolhemos o jogo eletrônico *Ultima Online* por sua grande popularidade e por possuir um emulador de servidor – o *RunUO* – de código aberto o qual permite alterações nos algoritmos de busca. Os experimentos foram realizados em uma máquina com processador *Intel Core 2 Duo* 2.2 Ghz, 2 GB de Memória RAM, sistema operacional *Windows* XP *Professional* 2002 *Service Pack* 2. Utilizamos como compilador o *Borland C++ Builder* 6.0.

4.1 Experimentos quantitativos em grafos baseados labirintos

Para comparar o desempenho dos algoritmos consideramos o número médio de vértices expandidos e o tempo médio de execução de cada algoritmo. Consideramos como vértices expandidos os vértices por onde os agentes passaram e o tempo de execução de um algoritmo representa o tempo em milissegundos gasto para atingir o objetivo. Estes dados são as médias das execuções do mesmo algoritmo no mesmo grafo. Para cada grafo executamos repetidamente o mesmo algoritmo. Limitamos o tempo total de execução do algoritmo e, por isso, os grafos maiores foram menos executados.

Para a realização dos experimentos foram desenvolvidos dois geradores de grafos: gerador de grafos aleatórios (GCA), e gerador de grafos conexos (GCC). O tamanho do grafo corresponde ao produto do número de colunas pelo número de linhas do grafo. Por exemplo, um grafo de tamanho igual *10.000* vértices possui *100* colunas e *100* linhas.

Definimos como vértices obstáculos os vértices que possuem grau zero. Os geradores recebem como parâmetros os vértices iniciais e finais, o tamanho, o percentual do número de arestas ou o percentual de vértices obstáculos, o grau médio do grafo e o tipo do grafo. O tipo do grafo é definido pelas distâncias entre nós vizinhos que podem ser constantes ou aleatórias.

Algoritmo Gerador do grafo	Algoritmo	Pesos das arestas	Heurística	Número de execuções
GCA	RTA*	Constante	*Manhattan*	18600
GCA	RTAP*	Constante	*Manhattan*	18600
GCA	RTA*	Aleatório	*Manhattan*	18600
GCA	RTAP*	Aleatório	*Manhattan*	18600
GCA	RTA*	Aleatório	*Sem heurística*	18600
GCA	RTAP*	Aleatório	*Sem heurística*	18600
GCC	RTA*	Constante	*Manhattan*	18600
GCC	RTAP*	Constante	*Manhattan*	18600
GCC	RRTA*	Constante	*Manhattan*	18600
GCC	RTA*	Aleatório	*Manhattan*	18600
GCC	RTAP*	Aleatório	*Manhattan*	18600
GCC	RRTA*	Aleatório	*Manhattan*	18600
GCC	RTA*	Aleatório	*Sem heurística*	18600
GCC	RTAP*	Aleatório	*Sem heurística*	18600
GCC	RRTA*	Aleatório	*Sem heurística*	18600
Total de execuções				279000

Tabela 1: Número de execuções dos experimentos quantitativos em grafos baseados em labirintos

O gerador de grafos aleatórios (GCA) garante a geração de grafos com pelo menos um caminho entre o vértice inicial e o vértice final, mas não garante que o grafo seja conexo. Por este motivo, nos grafos gerados pelo o GCA não é possível garantir o funcionamento correto do algoritmo RRTA* utilizando a estratégia de posicionamento pelo ponto médio. No GCA o percentual de vértices obstáculos define a densidade do grafo, uma vez que o obstáculo possui grau zero e conseqüentemente o número de arestas é menor. A

O Gerador de grafos conexos (GCC) garante a existência de caminhos entre quaisquer vértices do grafo, ou seja, todos os vértices estão interligados. Este algoritmo contempla a geração de grafos de tamanhos e densidades diferentes. Os grafos são no máximo 4-conexos, ou seja, cada vértice possui no máximo quatro vizinhos. Os pesos das arestas representam as distâncias entre os vértices que podem ser uniformes ou aleatórias. No GCC a quantidade de arestas define a densidade do grafo, consideramos um grafo com 100% de densidade quando todos os vértices possuírem grau igual a quatro.

Os experimentos foram separados em dois subgrupos distintos, o primeiro são experimentos executados nos

grafos gerados pelo GCA e o segundo nos grafos gerados pelo GCC.

Utilizamos como heurística a distância de *Manhattan* e para cada subgrupo estudamos o comportamento dos algoritmos em grafos onde os pesos das arestas são iguais, em grafos onde os pesos das arestas são aleatórios ou em situações sem heurística. Para os grafos gerados pelo GCA não estudamos o RRTA* porque a estratégia de posicionamento através do ponto médio não é adequada neste tipo de grafo.

A Tabela 1 apresenta o número de execuções dos algoritmos RTA*, RTA*P e RRTA* nos experimentos deste artigo, foram realizados 111600 execuções para o Grupo I e 167400 execuções para o grupo II.

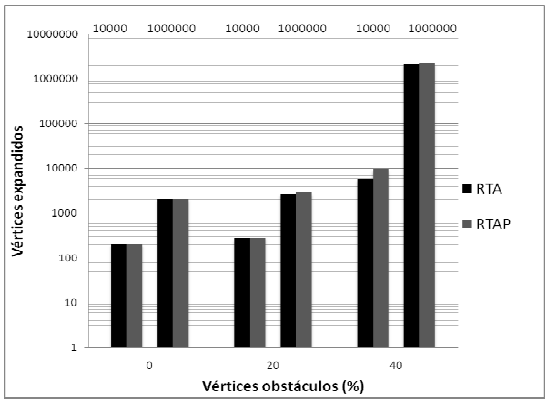


Figura 11: Gráfico da média do número de vértices expandidos para grafos gerados pela GCA com pesos variados entre 1 á 10

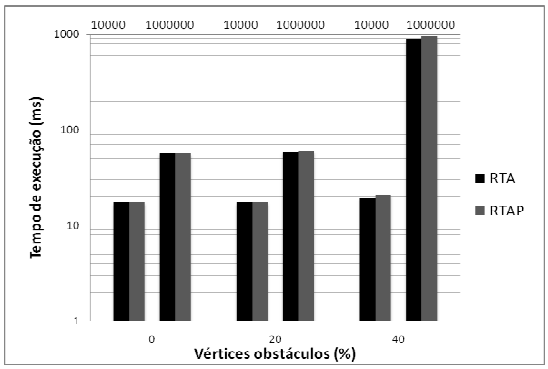


Figura 12: Gráfico da média do tempo de execução para grafos gerados pela GCA com pesos variados entre 1 a 10

Foram gerados 200 grafos para cada configuração, onde 100 têm peso uniforme nas arestas e 100 possuem pesos aleatórios. Cada configuração é definida pelo algoritmo que o criou: GCA, ou GCC; pelo seu número de vértices: 100, 10.000, 1.000.000; e pelo percentual de vértices obstáculos, no caso dos grafos gerados pelo GCA ou pela percentual de arestas, para os grafos gerados pelo GCC.

Os resultados são apresentados em gráficos de barras onde as colunas estão ordenadas de acordo com a legenda, não sendo necessário orientar-se pela cor.

Grupo I: os experimentos do Grupo I foram realizados em grafos com pelo menos um caminho entre o vértice inicial e o vértice final (grafos gerados pelo GCA), mas não necessariamente conexos. Cada vértice possui no máximo quatro vizinhos, a quantidade de vértices obstáculos define a densidade do grafo.

A Figura 11 compara o número de vértices expandidos versus o percentual de vértices obstáculos nos grafos do grupo I. O número de vértices expandidos aumenta quando o percentual de vértices que são obstáculos aumenta. O RTA* e o RTA*P empatam quando o grafo não possui obstáculos. À medida que aumenta o número de vértices que são obstáculos o RTA* apresenta resultados melhores que o RTA*P. Esta diferença é menor em grafos com pesos aleatórios.

A Figura 12 compara o tempo de execução (em milissegundos) com o percentual de vértices obstáculos nos grafos do grupo I. Os algoritmos gastaram mais tempo para solucionar os problemas quando o grafo possui 40% de vértices obstáculos. O RTA* e o RTA*P apresentam comportamentos semelhantes. Não há diferenças nos resultados entre os grafos com arestas com pesos uniformes e os grafos com arestas com pesos aleatórios.

Grupo II: os experimentos do Grupo II foram realizados em grafos conexos gerados pelo algoritmo GCC. Cada vértice possui no máximo quatro vizinhos, a quantidade de arestas define a densidade do grafo. As figuras 13 e 14 comparam o número de vértices expandidos versus o percentual de densidade nos grafos do grupo II. Os gráficos não indicam uma relação direta entre o número de vértices expandidos e a densidade do grafo.

O RTA*P e o RTA* apresentam resultados semelhantes. O RRTA* apresenta número de vértices expandidos maior que o RTA*. Entretanto, se dividirmos o número de vértices expandidos pelo número de agentes temos uma média de vértices expandidos menor que o RTA*. Sem utilizar heurísticas, o RTA*P expande menos vértices que o RTA*.

As figuras 15 e 16 comparam o tempo de execução (em milissegundos) versus o percentual de vértices de densidade nos grafos do grupo II. O RTA* e o RTAP apresentaram desempenhos semelhantes. Não há diferenças nos resultados dos grafos com arestas com pesos uniformes e arestas com pesos aleatórios.
O RTA* gasta mais tempo para executar a busca quando não utiliza heurística. O RTA*P é mais estável e mais eficiente que o RTA* em todas as densidades. Em grafos com pesos aleatórios a diferença entre o RTA*P e o RTA* é menor, mesmo assim o RTA*P é mais eficiente.

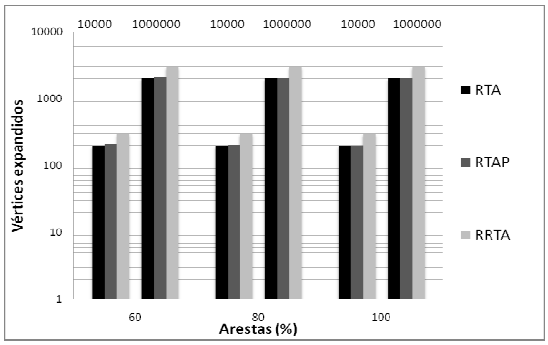


Figura 13: Gráfico da média do número de vértices expandidos para grafos gerados pela GCC com pesos constantes

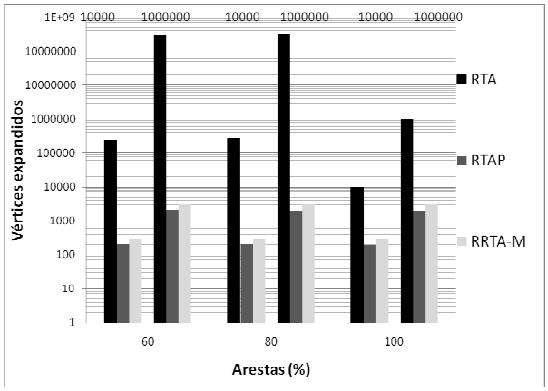


Figura 14: Gráfico da média do número de vértices expandidos para grafos gerados pela GCC com pesos constantes e sem a utilização de heurística

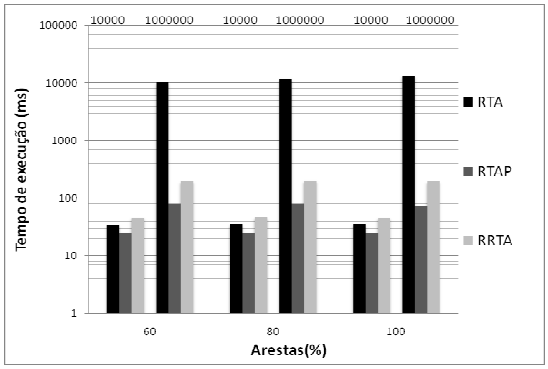


Figura 15: Gráfico da média do tempo de execução para grafos gerados pela GCC com pesos constantes

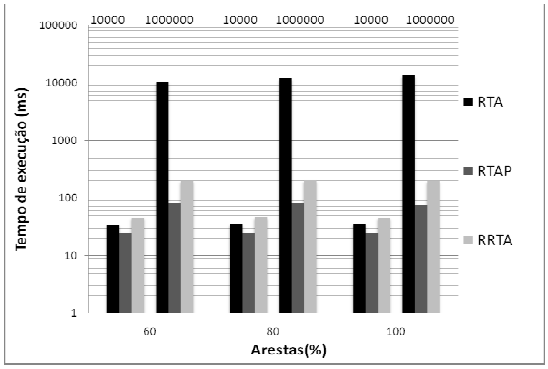


Figura 16: Gráfico da média do tempo de execução para grafos gerados pela GCC com pesos variados entre 1 a 10 e sem a utilização de heurística

4.2 Experimentos realizados no Ultima Online

Existem vários emuladores de servidor para o *Ultima Online*: *UOX*, *POL*, *SPHERE* e *RunUO* [Wikipedia, 2008]. Escolhemos o *RunUO* versão 2 RC2 pelo fato do mesmo ser bastante utilizado. O *RunUO* possui código aberto e recria por meio da plataforma Microsoft.NET o ambiente de um servidor para o *Ultima Online*. No *RunUO* é possível personalizar grande parte das características funcionais do jogo e manipular *NPCs* (*non player characters*). O teste de algoritmos de busca *on-line* implementados no *RunUO* teve como objetivo avaliar qualitativamente, no fluxo do jogo, a capacidade de um agente *NPC* determinar sua trajetória e executá-la.

O *Ultima Online* apresenta situações onde o deslocamento dos *NPCs* é dirigido por algoritmo de busca. Os movimentos de animais de estimação do jogador, denominados *pets,* representam uma destas situações. Os *pets* são agentes *NPCs* que se descolam pelo mapa entre obstáculos em busca de um determinado alvo, o qual pode ser um agressor, um alvo de perseguição móvel ou o seu respectivo dono, o jogador, Figura 17.

O algoritmo de busca utilizado pelo emulador *RunUO* é o A*, com área de busca bastante limitada (38 x 38 quadrantes). O mapa completo do jogo possui 6144 x 4096 quadrantes, sendo que cada um possui até oito vizinhos. A restrição na área de busca é motivada pelo ônus no tempo de resposta e de recursos computacionais.

Ao analisar os algoritmos de busca do *RunUO*, os quais se comportam de forma equivalente aos servidores oficiais de *Ultima Online*, constatamos a combinação de dois algoritmos O primeiro algoritmo é executado quando não há obstáculos entre o *NPC* e o seu alvo. Este algoritmo é uma simples tentativa de andar em linha reta na direção do alvo, intercalando com movimentos aleatórios quando é necessário um desvio simples. Nos casos onde essa tentativa não obtém sucesso, um algoritmo A* é executado. Se um obstáculo intransponível surge repentinamente na rota do *NPC*, a busca é cancelada e o primeiro algoritmo é executado novamente.

Os algoritmos de busca *on-line* são capazes de encontrar um caminho até o objetivo, se o mesmo existir. O processamento dessa busca não ocasiona lentidão perceptível no servidor. O A* também resolve a busca em todo o mapa do jogo se ampliamos sua área de busca. Contudo percebemos que as respostas do jogo ficam criticamente lentas, principalmente quando mais de vinte agentes executam a busca *off-line* simultaneamente.

A vantagem dos algoritmos *off-line* em relação aos *on-line* é a ótima solução com o menor número de passos entre dois pontos. Por esse motivo, a busca *off-line* é utilizada em grande parte dos jogos. Porém, quando o

jogo possui características de um sistema em tempo real, como os *MMORPG* e *RTS* (*real-time strategy*) , a busca *off-line* compromete a sua cadência.

A utilização de busca *off-line* consome maior tempo de processamento para o primeiro passo do agente do que a busca *on-line*. Na busca *off-line*, toda a rota é calculada antes do primeiro passo do agente, enquanto que na busca *on-line* cada passo pode ser planejado e executado logo em seguida. Pelo fato dos jogos digitais em tempo real terem características dinâmicas, os alvos também se movimentam e as rotas se modificam, inutilizando grande parte do cálculo da busca *off-line*, a qual deve ser refeita.

O *RunUO* utiliza algoritmo de busca *off-line* com área de busca bastante limitada em torno do agente (Figura 18a) e ignora os obstáculos dinâmicos do trajeto calculado. Porém esses ajustes no algoritmo provocam dois problemas de jogabilidade. O primeiro é a impossibilidade de calcular uma rota quando há obstáculos fixos seccionando a área de busca do agente, Figura 18b. O segundo é a sobreposição de agentes durante e após o caminhamento, impossibilitando a seleção unitária dos mesmos para demais atividades *in-game*, Figura 18d.

Implementamos no *RunUO* os algoritmos de busca *on-line* RTA* e LRTA* com o intuito de observar, na perspectiva do jogador, o diferencial em relação ao uso do A*.

Nos experimentos, diferentemente dos pseudocódigos da literatura, não pré-calculamos as heurísticas de cada vértice do mapa, pois no *Ultima Online* cada *NPC* necessitaria calcular e armazenar 6144 x 4096 valores heurísticos cada vez que uma nova busca fosse iniciada. Então optamos por calcular as heurísticas necessárias em tempo de execução e associamos a cada agente uma árvore binária para armazenar apenas as heurísticas atualizadas durante a busca. Essa árvore começa vazia e à medida que o agente se desloca, as atualizações heurísticas são inseridas na árvore. Quando o agente alcança o seu objetivo ou quando o alvo se desloca, a busca é dada como encerrada ou se prepara para calcular outro trajeto. Nesse instante a árvore binária é reiniciada.

Observamos que os algoritmos *on-line* resolvem os problemas gerados pela utilização de busca *off-line* com área limitada. Os agentes são capazes de caminhar até o seu alvo sempre que existir um caminho válido em qualquer parte do mapa e os agentes jamais se sobrepõem, Figura 18e. Essas soluções melhoram consideravelmente a jogabilidade com os *pets*.

Um problema observado com os algoritmos *on-line* ocorre quando há muitos obstáculos na vizinhança dos *NPCs* formando uma área côncava. Nesse caso os *NPCs* parecem, por um momento, estarem perdidos. Este efeito acontece porque os movimentos dos *pets* acompanham o planejamento do algoritmo. Apesar dessa desvantagem, os agentes encontram seu objetivo, Figura 18c. Este problema pode ser resolvido calibrando o número de iterações que o agente deve planejar a rota antes de se deslocar efetivamente no ambiente do jogo.

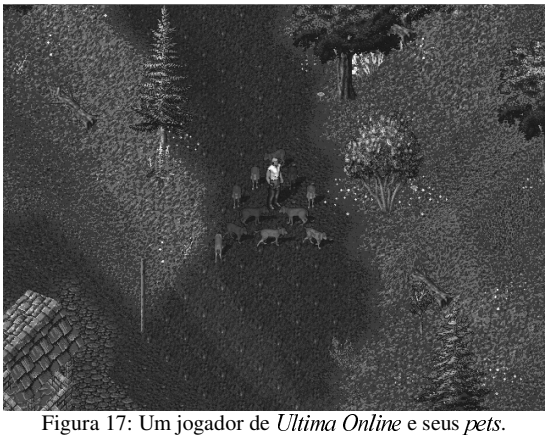


Figura 17: Um jogador de *Ultima Online* e seus *pets*.

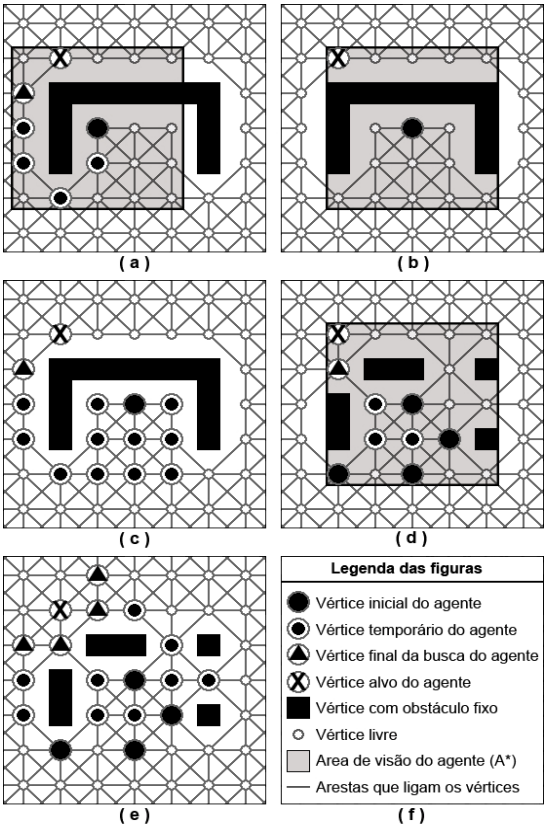


Figura 18: Comparações entre algoritmos *off-line* (a, b, d) e *on-line* (c, e) em situações observadas no *RunUO*.

5. Considerações Finais

Em jogos digitais existem situações onde é necessária a utilização de algoritmos de busca. A maioria dos jogos eletrônicos resolve o problema de busca com soluções baseadas no algoritmo A*. Os algoritmos de busca *off-line* consomem mais recursos que os algoritmos de

busca *on-line*. A utilização de algoritmos de busca *on-line* em jogos digitais economiza processamento e memória viabilizando a busca simultânea de vários agentes em ambientes mais complexos. Outra vantagem é que os algoritmos de busca *on-line* encontram a solução mesmo em ambientes dinâmicos sem a necessidade de recalcular toda a trajetória.

O RRTA* (*Relay Real Time A**) distribui a busca em mais de um agente e é útil em situações onde a busca possa ser dividida, em jogos digitais, com vários jogadores podemos distribuir a busca e também utilizar este algoritmo em situações de perseguição. Apesar do RRTA* apresentar números que indicam que ele é menos eficaz que o RTA*, ele encontra caminhos para mais de um agente. Para conseguir o mesmo resultado, o RTA* teria que executar uma busca para cada agente.

Quando dispomos de heurísticas adequadas o RTA* é mais eficiente que o RTA*P. Entretanto existem situações em que não possuímos heurísticas, por exemplo, em alguns jogos digitais não conhecemos a posição do vértice objetivo. Para estas situações o RTA*P é mais eficiente que o RTA*. O RTA*P é mais estável que o RTA*, em grafos 4-conexos o número de vértices expandidos e o tempo de execução do RTA*P sofrem menos variação que os do RTA* quando acrescentamos, retiramos ou apenas modificamos a heurística. Quando o RTA* apresenta-se mais eficiente que o RTA*P esta diferença não é grande, por isso a utilização do RTA*P em jogos digitais é indicada.

O emulador *RunUO* é um ambiente adequado para teste de algoritmos de busca em jogos pois mostrou ser suficientemente completo e flexível.

Aplicar algoritmos de busca *on-line* em jogos digitais é motivado pelo fato desses experimentos nos levarem a uma melhor observação do comportamento dos agentes da inteligência artificial (IA) em situações diversas. Tais observações são inspirações para novas idéias de implementações. A percepção humana sobre a inteligência de um agente de IA pode ser presumida quando há uma imersão no universo de ação do mesmo, essa especificidade é bastante comum nos jogos. Os algoritmos *on-line* são eficientes nos jogos possibilitando a busca em ambientes dinâmicos e provendo a possibilidade de executar a busca para um número maior de agentes.

5.1 Trabalhos Futuros

Propusemos alguns algoritmos e estudamos o comportamento de algoritmos de busca em tempo real em jogos digitais. Contudo ainda é necessário avaliar estes algoritmos em outros jogos e em contextos diferentes.

A estratégia de posicionamento do RRTA* é fundamental para sua eficiência, é necessário estudar outras estratégias e validá-las no contexto de jogos digitais além de explorar o potencial do RRTA* em situações de perseguição.

É importante investigar o comportamento de outros algoritmos de busca *on-line* em jogos eletrônicos e estudar a complexidade e a completude dos algoritmos propostos.

Referências

Bulitko, V, Sturtevant, N, Lu, J., & Yau, T. 2007. *Graph Abstraction in Real-time Heuristic Search. Journal of Artificial Intelligence Research JAIR*

Koenig, S. 2006. *Real-Time Adaptive A*. AAMAS*

Koenig, S. 1999. *Exploring unknown environments with real-time search or reinforcement learning. In Proceedings of the Neural Information Processing Systems NIPS.*

Koenig,, S. , Likhachev, M., and Sun, X., 2007. *Speeding up Moving-Target Search. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS.*

Korf, R. E. 1990. *Real-time heuristic search. Artificial Intelligence, 189–21.*

Millington, I. 2006. *Artificial Intelligence for Games. Elsevier.*

Nonnenmacher, V., Ferreira, S. and Osorio, F. 2007. *A-Constelação: Extensões do A* para aperfeiçoar a I.A de NPCs em jogos. SBGames*

Origin – Ultima Online. *Play guide starting up em: http://guide.uo.com/start3d_welcome.html [Acessado 8 de Agosto de 2008].*

Pottinger, D. C. 2000. *Terrain analysis in real-time strategy games. In Proceedings of Computer Game Developers Conference GDC.*

Rabin, S. 1995. *AI Game Programmig Wisdon.*

Russel, S. and Norvig, P. 2004. *Inteligência Artificial. Campus.*

Sun, X., Koenig, G., and Yeoh, W.. 2008. *Generalized Adaptive A*. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems,AAMAS*

Van Waveren, J. M. P. 2001. *The Quake III Arena Bot. Master's thesis, Delft University of Technology, Delft, Netherlands.*

Wikipedia - *Ultima Online em: http://en.wikipedia.org/wiki/Ultima_Online [Acessado 8 de Agosto de 2008].*

Wooldridge, M. 2002. *An Introduction to Multiagent Systens. John Wiley Sons, LTD*

Yokoo, M.And Ishida, T. 1999. *Search Algorithms for Agents, Multiagent Systems: A Modern Approach toDistributed Artificial Intelligence 1999., 36-44.*

A Facial Animation Interactive Framework with Facial Expressions, Lip Synchronization and Eye Behavior

Rossana B. Queiroz Marcelo Cohen Soraia R. Musse

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga, 6681, Porto Alegre, RS, Brazil, 90619-900

Figure 1: *Example of characters performance in our prototype during an expressive talk.*

Abstract

In this paper we describe our approach of generating convincing and empathetic facial animation. Our goal is to develop a robust facial animation platform that can be scalable, usable and easily extended, in order to allow integration of research on the area and also the direct incorporation of the characters in interactive applications, such as Embodied Conversational Agents and games. We have developed a framework capable of animating MPEG-4 parametrized faces easily through high-level description of facial actions and behaviors. We also present a case study which integrates computer vision techniques in order to provide interaction between the user and a character, that interacts with different facial actions according to detected events in the application.

Keywords:: Facial Animation, MPEG-4, Facial Expressions, Lip Synchronization, Eye Behavior, Interactivity, Facial Animation Description Language

Author's Contact:

{rossana.queiroz, marcelo.cohen, soraia.musse}@pucrs.br

1 Introduction

Animated virtual human faces have been widely used in many applications, such as movies, games and Embodied Conversational Agents (ECAs). In all of these applications, characters' reactions should help in user immersion as well as to provide a believable interaction. Thus the facial behavior coherence is very important, to increase the credibility of the character.

Recent research on facial animation has lead to models that explore expressiveness, communication and interactivity. Many of these research are focused on ECA development [de Rosis et al. 2003; Smid et al. 2004; Cosi et al. 2007; Huang et al. 2008] and provide us a range of studies that correlate human psycho-social behavior and facial animation. Together with a facial animation platform development, we find several studies about face parameterization and scripting languages for assisted and automatic animation generation [Perlin 1997; Cassell et al. 2001; Rutledge 2001; Arafa and Mamdani 2002; Byun and Badler 2002; Carolis et al. 2004; Not et al. 2005; Arya and DiPaola 2007; Vilhjálmsson et al. 2007]. Those studies have provided ways to the higher-level description of a character's face actions. Most of these languages are based on XML and try to describe face actions in different levels, such as attributing values of some specific parameters, or specifying more complex behaviors which implicate the animation of various facial attributes in a synchronized way.

We observe that in some works focused on complex facial models, such as the physically-based animation by [Sifakis et al. 2005], the implementation of the facial animation platform is constrained to a single character model where all tests are performed on. In fact, if a facial animation prototype is focused only on its main research subject, it becomes difficult the addition of new functionalities or its use in other environments or applications. Concerning the project of a facial animation platform, we can emphasize some requirements in order to obtain a robust and usable tool:

- A consistent set of face animation parameters, which allows us to get a satisfactory control of face attributes in different faces;
- Minimal need of animator manual work, such as preparing a set of animation keyframes to be interpolated;
- Mesh-deformation algorithms which produce realistic representation of facial muscle actions;
- An interface which can be understood both by computer and humans, enabling them to determine or edit characters' actions;
- The real possibility of easy incorporation of the animated faces in other applications;
- A framework which enables interactivity with real time animation generation, without the need of previously recorded animations.

In this context, our work presents an interactive facial animation framework which considers facial expressions, synchronized speech and eye behavior generation where the users define the characters' actions by high-level description and can use the produced animations on different face models. Our goal is to develop a robust facial animation platform that can be easily extended, scalable and usable in order to allow integration of other research on the area and also the direct incorporation of the characters in interactive applications such as ECAs and games. Our main contribution is the methodology of building an animation platform using free or open source tools, integrating some known animation models and capable of generating a substantial amount of different face actions in good quality animations.

This paper is organized as follows: the next section presents some related work. Section 3 describes the architecture of our model and the main technologies and tools that we have used for development of the framework. In Section 3.4 we describe a case study: an interactive application where a character reacts according to detection of the user's face. Finally we make some final remarks and suggest future work in order to improve and evaluate our model.

2 Related Work

Parameterization techniques for facial animation have been an area of active research since Parke [Parke 1982]. We can identify in the

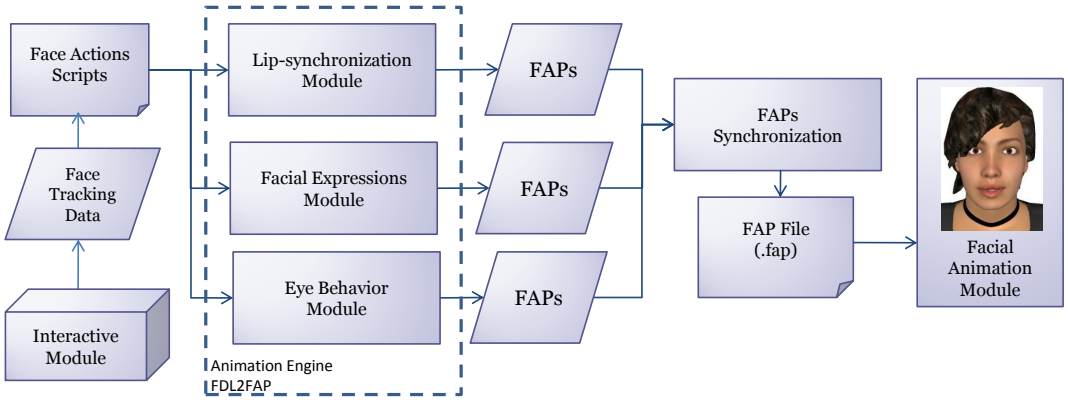


Figure 2: *Overall architecture diagram of the framework*

literature two standards of parameter sets that are widely used in facial animation works: Facial Coding System (FACS) [Ekman and Friesen 1978] and the MPEG-4 Facial Animation (MPEG-4 FA) [Pandzic and Forchheimer 2003]. However, these standards provide a rather low-level basis for animators, as they describe the parameters as a set of pseudo-muscles which can be activated in a given moment, producing a facial expression (Subsection 3.3 overviews the MPEG-4 FA). Hence there are several efforts to produce a more high-level parametric approach to generate facial animations in a effective way [Perlin 1997; Cassell et al. 2001; Rutledge 2001; Byun and Badler 2002; Carolis et al. 2004; Not et al. 2005; Arya and DiPaola 2007].

In the bibliography, we can also find some works that describe their facial animation frameworks, which aim to provide a desirable platform for facial animation research or the development of talking heads applications. Wang [Wang et al. 2007] describes a methodology to build an expressive facial animation system with lip synchronization using affordable off-the-shelf components. This include the *FaceGen Modeller*[1] software for face and key meshes generation, and the Microsoft Speech SDK [2] as the speech API.

Cosi [Cosi et al. 2005] proposes a facial animation toolkit implemented in MATLAB[3] created mainly to speed up the procedure for building the LUCIA talking head [Cosi et al. 2007] through motion capture techniques, translated to MPEG-4 parameters. Although it seems to be a promising approach, it was built over proprietary software.

DiPaola and Arya [DiPaola and Arya 2007] propose a facial animation framework compatible with the MPEG-4 standard called iFace, whose binaries are available on the web[4]. iFace allows interactive non-verbal scenarios through a XML-based scripting language called FML (Face Modeling Language) [Arya and DiPaola 2007], which allows both parallel and sequential description of face actions. Face actions include talking, expressions, head movements and low-level MPEG-4 parameters. Actually, iFace has been used for behavioral animation studies in order to reach a comprehensive association of facial actions to personality types, creating a higher-level facial parameter set in a personality-space [Arya et al. 2006].

Balci *et al.* [Balci et al. 2007] designs Xface, a set of open source tools for creation of talking heads using MPEG-4 and keyframe-based animation driven by the SMIL-Agent scripting language. The toolkit is freely available [5] and aims to supply the lack of free and open source tools for research as mentioned above. However, Xface uses the SMIL-Agent scripting language for its keyframe-based animation module. In other words, for each face model it is necessary a set of corresponding key-meshes with the different facial expressions and visemes.

Our approach uses Xface as the facial animation engine and integrates a model to generate MPEG-4 animation automatically through facial actions scripts with lip synchronization [Rodrigues 2007] and eye behaviors [Queiroz et al. 2007], without the need of keyframing meshes. Moreover, in our framework external controls are possible in order to interactively define characters' actions. As a case study of the framework capabilities, we have developed a simple application where the character eye movements are driven by detection of the user's face, in an approach slightly similar to [Courty et al. 2003].

[1] http://www.facegen.com/
[2] http://www.microsoft.com/speech/download/sdk51
[3] http://www.mathworks.com/
[4] http://ivizlab.sfu.ca/research/iface/
[5] http://xface.itc.it/

3 Model

This section presents the overall architecture of our framework, as well the scripting language for the description of facial actions. We also present the approach to animation synthesis and interactivity.

3.1 Architecture

The overall architecture of our framework is presented in Figure 2. The input for the *Animation Engine* is a script file containing the description of one or more facial actions. Our scripting language is called FDL (Face Description Language) hence we name our script files as FDL files. The *Animation Engine* interprets the facial actions within a FDL file and generates the animation according to them. In the current stage of our research, the FDL files can describe a sequence of three types of high-level face actions: talking, facial expressions and eye behaviors. Each of these types of face actions are independent, and the processing of them in our *Animation Engine* is performed by three different modules: *Lip-Synchronization Module*, *Facial Expression Module* and *Eye Behavior Module*. This provides the values of each MPEG-4 facial animation parameter (FAP) corresponding to the desired facial actions through time. After this step, the *FAPs Synchronization Module* receive the FAP values and resolves possible conflicts among them, such as making the combination of facial expressions and visemes. The output is a FAP file containing the final animation, which can run on any MPEG-4 compliant player and with different face models. Our *Facial Animation Module* uses Xface for the facial animation synthesis (described in Subsection 3.3).

The *Eye Behavior Module*, whose architecture is illustrated in Figure 3 contains the implementation of [Queiroz et al. 2007] a model for automatic generation of eye animation. This provides a set of eye behaviors which can be used in combination with different affective states. The model uses a known statistical model [Lee et al. 2002] (*Default Model*) as a saccadic eye movement engine and creates differentiated behaviors (*Behavioral Database*) through changes in the gaze parameters, such as direction, magnitude and interval between movements. Saccadic movements (or saccades) are rapid movements of both eyes from one gaze position to another [Lee et al. 2002].

Eye behaviors are described as high-level actions in FDL scripts. When combined with facial expressions, they contribute for increasing both the expressiveness and engagement in communica-

tion. The *Expressive Gaze Generator* receives descriptions of face actions and returns the FAP values for each gaze generated according to the specified eye behavior, also providing head and eyelids movements according to the model rules. The *Interactive Module*, described in Subsection 3.4, can provide the input eye behavior sequence (e.g. following the user face) in a interactive way.

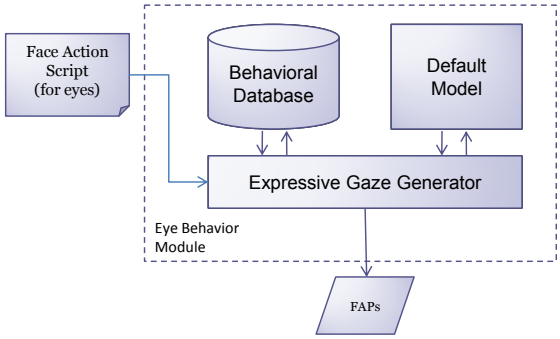


Figure 3: *Diagram of the Eye Behavior Module*

The job of the *Facial Expression Module* (Figure 4) is simpler: currently, it only produces facial expressions predefined as a set of FAP values in the *Emotional Database*, representing "pure" emotions. But the framework can be easily extended, in order to allow new facial expressions and algorithms to generate derived expressions. The facial expressions are also described as facial actions in the FDL language.

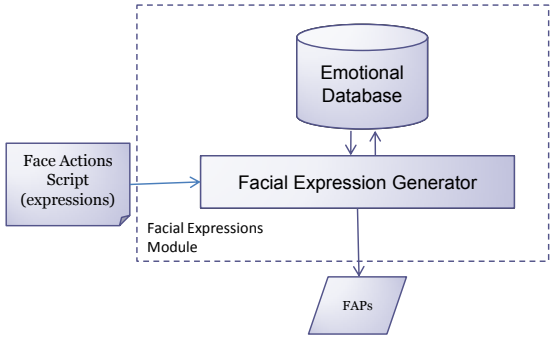


Figure 4: *Diagram of the Facial Expressions Module*

The *Lip Synchronization Module* follows the methodology described in [Rodrigues 2007]. Basically the module receives a sound file (.wav) as input, containing the character speech and an auxiliary file containing its textual description. For the generation of lip animation, we use the CSLU Toolkit [6] which aligns the sound file with the textual description and provides a file with the timeline of the phonemes through time. This file is the input for the *Phoneme-Viseme Mapping Module*, which connects the phonemes to their visual representation (visemes), according to [Rodrigues 2007]. The CSLU toolkit alignment script generates 41 English phonemes that are mapped to the 14 visemes specified in the MPEG-4 standard. There is also an auxiliary "viseme" we have called *silence* to represent the pauses in speech. After this step, we have the complete timeline of visemes, whose mapping to FAPs are carried out in the *Viseme-FAP Mapping Module*. In this stage the blending of visemes (i.e. the transition between two visemes) is also performed. For this transition, we apply a linear interpolation between them. Figure 5 illustrates the architecture of this module.

After the three modules of the *Animation Engine* have finished, the *FAPs Synchronization Module* receives the output FAP values and resolves possible conflicts among them. Meanwhile, our framework solves three types of conflicts:

[6]http://cslu.cse.ogi.edu/toolkit/

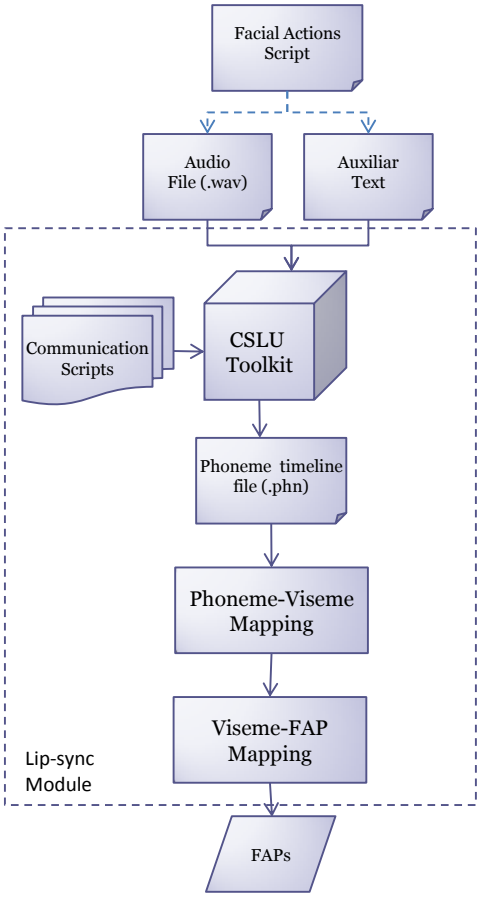


Figure 5: *Diagram of the Lip Synchronization Module*

1. Viseme-Expression conflict: occurs mainly in mouth region FAPs, when the facial expression includes mouth movement. To solve this conflict, we blend the mouth FAP streams generated by the *Facial Expression* and *Lip Synchronization* modules, in a weighted sum of them. Pyun [Pyun et al. 2003] proposes that the weights for different visemes and expressions should variate according to the "importance" of them. However, in the current stage our framework uses the same weights for all combinations of expressions and visemes, as shown in equation 1. Therefore, each final FAP (F_i) will be generated through the weighted sum of both expression (E_i) and viseme (V_i) FAPs.

$$F_i = \frac{E_i}{4} + \frac{3V_i}{4} \quad (1)$$

2. Eyes-Expression conflict: the *Eye Behavior Module* generates eyelid movements related with the gaze direction and shift [Queiroz et al. 2007]. But some facial expressions also include the eyelids conformation. In such cases, we also make a weighted sum, as in the Viseme-Expression conflict (25% for eye behavior and 75% for expression FAPs)

3. Head control: during FDL processing, some eye behaviors lead to head movements. If that happens, the *Eye Behavior Module* signals the *FAPs Synchronization Module*, which in turn preserves the generated head FAPs. If there are no head movements, then our framework provides an implementation of Perlin Noise [Perlin 1985] to generate subtle head movements instead of none, in order to approach a more realistic head behavior.

We also include in our framework a more generic module we named *Interactive Module* which is a *hotspot* to enable the inclusion of interactive events detection and automatic generation of animation with their respective characters' responses in runtime. Through this module, we can incorporate Virtual Reality approaches (e.g.

through Computer Vision techniques), emotional models which lead to facial expressions, Natural Language Processing techniques to verbal or textual communication (e.g. a chat), and such. We can also define rules of the interactivity of two or more virtual agents in a simulation or game application.

3.2 Facial Description Language

In our previous work [Queiroz et al. 2007], we have encoded our face actions as a high-level parameterized set of commands in Lua[7] scripts we called Facial Description Language (FDL). We opted to use Lua as auxiliary scripting language instead of XML-like notations (as SMIL-Agent and FML) for two reasons: *i)* Lua syntax is clearer and its structure helps intuitively to understand the sequencing of actions; and *ii)* it is a powerful way to easily incorporate new functionalities in the framework.

Our approach describes each expression, viseme or eye behavior as a facial action which is triggered by script files supplied by an interactive application, or simply selected by an user. These script files describe one or more face actions, which are interpreted, processed and translated to low-level MPEG-4 Facial Animation Parameters (FAPs). The main script file contains the information about the speech (sound and text file names) and the scripts with the eye behavior and expressions "storyboard", as shown below. If there is no speech in the desired animations, the *speechSound* and *speechText* fields should be filled with the `"none"` string value.

```
FDL = {
   speakSound = "hastalavista.wav",
   speakText = "hastalavista.txt",
   expressionScript = "expressions.sb",
   head = "default",
   eyesScript = "eyes.sb",
   output = "animation",
}
```

A sequence of facial expressions can be described as the script below, which informs the desired expressions through time, with their respective duration times in frames. The example shows some of the facial expressions which are part of our *Emotional Database*.

```
expressions = {
   {"joy", 100},
   {"sadness",100},
   {"surprise", 100},
   {"anger", 50},
   {"disgust",50},
   {"fear",100},
   {"trust",100},
   {"tongueout",100},
   {"inlove",100},
   {"worry",100},
   {"sleepy",100},
}
```

Similarly, the eye behavior sequence can be described as the example below. The set of eye behaviors description and its parameters are the same of GDL (*Gaze Description Language*) files of [Queiroz et al. 2007] adding the "pursuit" behavior we describe in Section 4 in order to generate the eyes pursuit movement.

```
eyes = {
   {"lookTo","left",17.0,"yes",50},
   {"lookTo","upleft",10.0,"yes",150},
   {"lookTo","up",15.0,"no",150},
   {"lookTo","upright",7.0,"yes",50},
   {"lookTo","right",15.0,"yes",50},
   {"lookTo","downright",12.0,"no",50},
   {"lookTo","down",12.0,"yes",50},
   {"lookTo","downleft",12.0,"yes",50},
   {"default", "talking", 0.9, 500},
   {"concentration", 0.01,50},
   {"discomfort", 0.99, 0.6,500},
   {"distress", 0.5,400},
   {"ironic", 0.5,50},
}
```

As we show above in the first FDL script, we generate mouth speech animation automatically through a sound file and its textual description. After the phoneme-viseme mapping, our system generates a FDL script such as example below, which is processed by the *Viseme-FAP Module*. This type of script can be alternatively used for, e.g. the output of other speech-processing system, or even edited manually. Following the same syntax of the other FDL scripts, it describes the sequence of visemes through time.

```
Sentence = {
   {"silence", 11},
   {"A", 6},
   {"kg", 1},
   {"Q", 1},
   {"nl", 3},
   {"sz", 3},
   {"I", 6},
   {"A", 2},
   {"silence", 27},
}
```

3.3 Animation Platform

Our framework follows the MPEG-4 Facial Animation standard for face and animation parameterization. The MPEG-4 FA describes the steps for creation of an animated face by definition of a set of parameters in a standardized way. First, MPEG-4 defines 84 feature points (the FPs - see Figure 6) placed on a character head, which in turn define animation parameters, as well as calibrating the models when they are exchanged between different players.

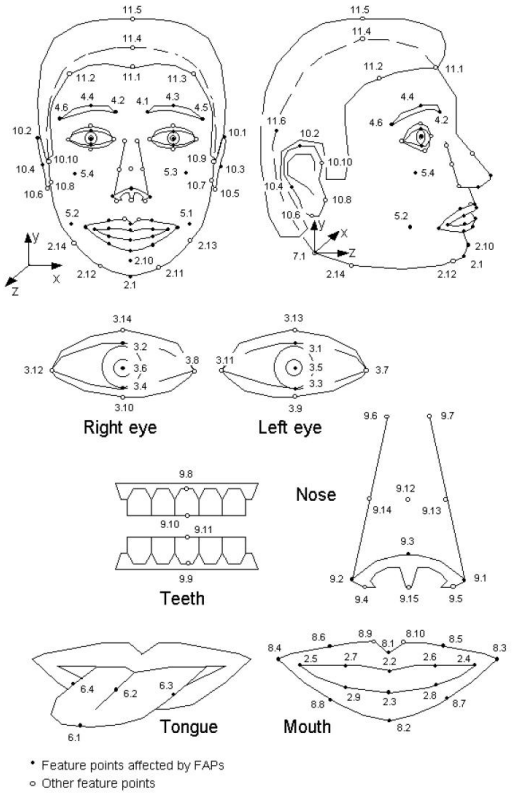


Figure 6: *MPEG-4 Feature Points*

In the standard, 68 values (the *Facial Animation Parameters*, or FAPs) define the deformation between two frames of animation. The first two suit a framework with high level parameters, representing visemes and the six basic emotions defined by Ekman [Ekman 1999]. The next ones deal with specific regions on the face, as left eyebrow, right corner lip, tongue tip, etc. FAP values are independent of model geometry. For this reason, FAPs have to be calibrated prior to use on a face model. This is done using *Face Animation Parameter Units* (FAPU, illustrated in Figure 7) which are defined as fractions of distances between key facial features. Moreover, the information about the 3D model is provided through the *Facial Definition Parameters* (FDPs) which allow one to configure the 3D facial model to be used at the receiver, either by adapting a previously available model or by sending a new model. The new or the adapted model is then animated by means of FAPs [Abrantes and Pereira 1999; Balci 2004].

Each model geometry has its own FAPU and every FAP value is

[7] http://www.lua.org

calibrated by a corresponding FAPU value as defined in the standard. Together with FPs, they serve to achieve independence of the face model for MPEG-4 players [Balci 2004]. Figure 8 illustrates the FAP file format, which can be read by any MPEG-4 compliant player.

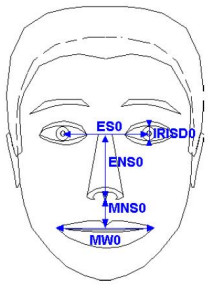


Figure 7: *MPEG-4 FAPU*

We use Xface as the MPEG-4 FA engine which plays the FAP animation generated by the *Animation Engine Module*. The Xface toolkit is implemented in C++ using the OpenGL API and incorporates four pieces of software [Balci et al. 2007]: *i)* the Xface core library, which enables the developers to embed 3D facial animation into their applications; *ii)* XfaceEd, an authoring tool to generate MPEG-4 parameterized meshes; *iii)*; Xface Player, a sample application that demonstrates the toolkit in action; and *iv)* XfaceClient, which allows remote network control of XfacePlayer. Our Facial Animation framework module is implemented over the XfacePlayer, adding a FDL loader function and integrated to the *Interactive Module*, as we describe in Subsection 3.4. Our 3D face models were generated in *FaceGen Modeller*, and body and hair was modeled by artists. Our faces were parameterized according to the MPEG-4 standard using the XfaceEd tool. XfaceEd generates a configuration file containing FAPU and FDP data, which are read by XfacePlayer to open a face model.

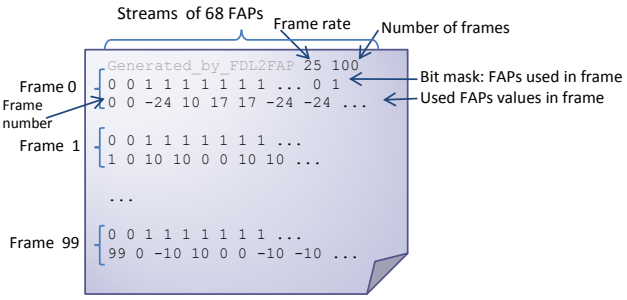


Figure 8: *FAP file scheme*

Figure 1 illustrates the visual results during the performance of some scripted expressive talks. The 3D model independence of our generated animations is illustrated in Figure 9, which shows our two different characters performing the same facial expressions (i.e. playing the same generated FAP file for both characters).

3.4 Interactivity

In our prototype, we can currently generate new animations by two ways:

- loading FDL files, which calls the FDL processing routines and then runs the generated FAP file; or
- enabling our *Interactive Module* to detect events and generate runtime specific FDL/FAP files with the respective character responses for these events.

The next section presents a case study in which our framework was extended to an interactive application using a Computer Vision technique to drive the animation of a character's eye.

Figure 9: *Performance of facial expressions by two different characters in our system.*

4 A Case Study: Following Faces

As we described in the previous sections, a robust facial animation platform should be extensible enough to allow the developers to build its applications in an effective way. When referring to facial animation, we see that many applications need automatic runtime generation of animation generation, in order to produce coherent characters' reactions throughout the interaction with other agents or users.

In order to provide support for these features, we have extended our framework with the *Interactive Module* and created an interactive application that generates eye motion driven by Computer Vision techniques. Specifically, our interactive application presents a virtual character, which follows the user's face through gaze behavior. The module that detects and tracks the user's face employs the face

detection method provided by the OpenCV[8] Computer Vision library as a starting point [Viola and Jones 2001].

The face tracker runs on a separate thread, hence face detection data is readily available. The OpenCV detection function returns the location and the face radius (in pixels) of all recognized faces, but at this moment we consider just the face which is closest to the virtual character, i.e. the face with the largest radius. Based on this information, the direction and magnitude (angle of eyeball rotation) of the virtual character's eyes are calculated as follows:

1. Obtain x and y coordinates from the face tracker for the closest face. These coordinates are converted to the range $[-1, 1]$ (zero being the center of the camera image).

2. Determine gaze direction independently for x and y:

   - If $x < -0.1$ we assume that the face is towards the right side of the camera image. This is 10% of the right half size, so very small face movements will not cause any change in the eye direction. Note that the meaning is reversed, as the camera image is normally mirrored.

   - Likewise, if $x > 0.1$ then the face is towards the left side of the camera image. The process is carried out similarly for y.

3. Now to determine the direction, we just combine the vertical and horizontal information. This produces one out of eight discrete directions (up, up and left, left, etc).

4. Finally, the magnitude is computed through the Euclidean distance of the face to the center of the camera image, scaled by a factor of 20. This produces roughly 12-15 degrees of horizontal and vertical rotation, as the face tracker does not return coordinates for faces that are very close to the edges of the screen, as there is not enough data.

After that, these parameters are used to generate an eye behavior FDL script file, which is processed by the *Animation Engine Module*. Note that there is a single computation for both eyes: we assume, for the purposes of this test application, that the user will never get too close to the camera, thus the virtual character eyes will never need to converge inwards.

In order to make the virtual character's gaze follow the user's face, a new gaze behavior was implemented. This behavior (called *pursuit*), allows eyeball rotation without returning to a central position, allowing continuity between eye's face actions. Pursuit movements occur when the eyes follow a moving object, either voluntarily or involuntarily. They are quite different from saccadic movement: they are smooth, slower and have a smaller latency [Lee et al. 2002]. After each face detection, a FDL script is generated, as shown in the example below:

```
eyes = {
   {"pursuit","right",17.0,"yes",50},
   }
```

The parameters recognized by the *pursuit* behavior are the following:

- gaze direction (2D rotation axis), described in a textual representation as eight discretized directions (such as 0°, 45°, 90°as "left", "upleft","right"). The use of this discretized values is inherited of the Default Model implementation of the model developed by [Lee et al. 2002];

- magnitude angle of eyeball rotation;

- a yes/no value indicating if the head should follow eye movement or not;

- duration (in frames).

Figure 10 illustrates our "Following Faces" application.

[8] http://sourceforge.net/projects/opencvlibrary/

5 Final Remarks

This paper presents an interactive facial animation framework which considers facial expressions, synchronized speech and eye behavior generation where the users define the characters' actions by high-level description. For animation generation, we follow the MPEG-4 Facial Animation standard. Consequently, the generated animations can be used in different face models. We also present a case study that incorporates Computer Vision techniques in a simple Virtual Reality application, in which character's interacts with user following his/her face with gaze behavior. This application shows that our *Interactive Module* is promising for the incorporation of different events and runtime interactive issues.

As future work, we aim to:

- improve the *Facial Expression Module*;

- provide head control, also allowing independent head motion as facial actions (head behaviors);

- support visemes from other languages;

- incorporate a GUI tool for editing facial actions in an easier way;

- evaluate some framework features with subjects.

In summary, our visual results show that in the current stage of development, our approach is promising, since our framework allows the integration of other research on the area and also provides interactive control of characters for applications such as ECAs and games.

Acknowledgements

This work was developed in collaboration with HP Brazil R&D. Thanks to Prof. Avelino Zorzo for providing his pictures.

References

ABRANTES, G., AND PEREIRA, F. 1999. Mpeg-4 facial animation technology: survey, implementation, and results. *Circuits and Systems for Video Technology, IEEE Transactions on 9*, 2 (Mar), 290–305.

ARAFA, Y., AND MAMDANI, A. 2002. Multi-modal embodied agents scripting. *Multimodal Interfaces, IEEE International Conference on 0*, 454.

ARYA, A., AND DIPAOLA, S. 2007. Face modeling and animation language for mpeg-4 xmt framework. *Multimedia, IEEE Transactions on 9*, 6 (Oct.), 1137–1146.

ARYA, A., JEFFERIES, L. N., ENNS, J. T., AND DIPAOLA, S. 2006. Facial actions as visual cues for personality: Research articles. *Comput. Animat. Virtual Worlds 17*, 3-4, 371–382.

BALCI, K., NOT, E., ZANCANARO, M., AND PIANESI, F. 2007. Xface open source project and smil-agent scripting language for creating and animating embodied conversational agents. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, ACM, New York, NY, USA, 1013–1016.

BALCI, K. 2004. Xface: Mpeg-4 based open source toolkit for 3d facial animation. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, ACM Press, New York, NY, USA, 399–402.

BYUN, M., AND BADLER, N. I. 2002. Facemote: qualitative parametric modifiers for facial animations. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, 65–71.

CAROLIS, B. D., PELACHAUD, C., POGGI, I., AND STEEDMAN, M. 2004. Apml, a mark-up language for believable behavior generation. *H. Prendinger, ed., Life-like Characters.*, 65–85.

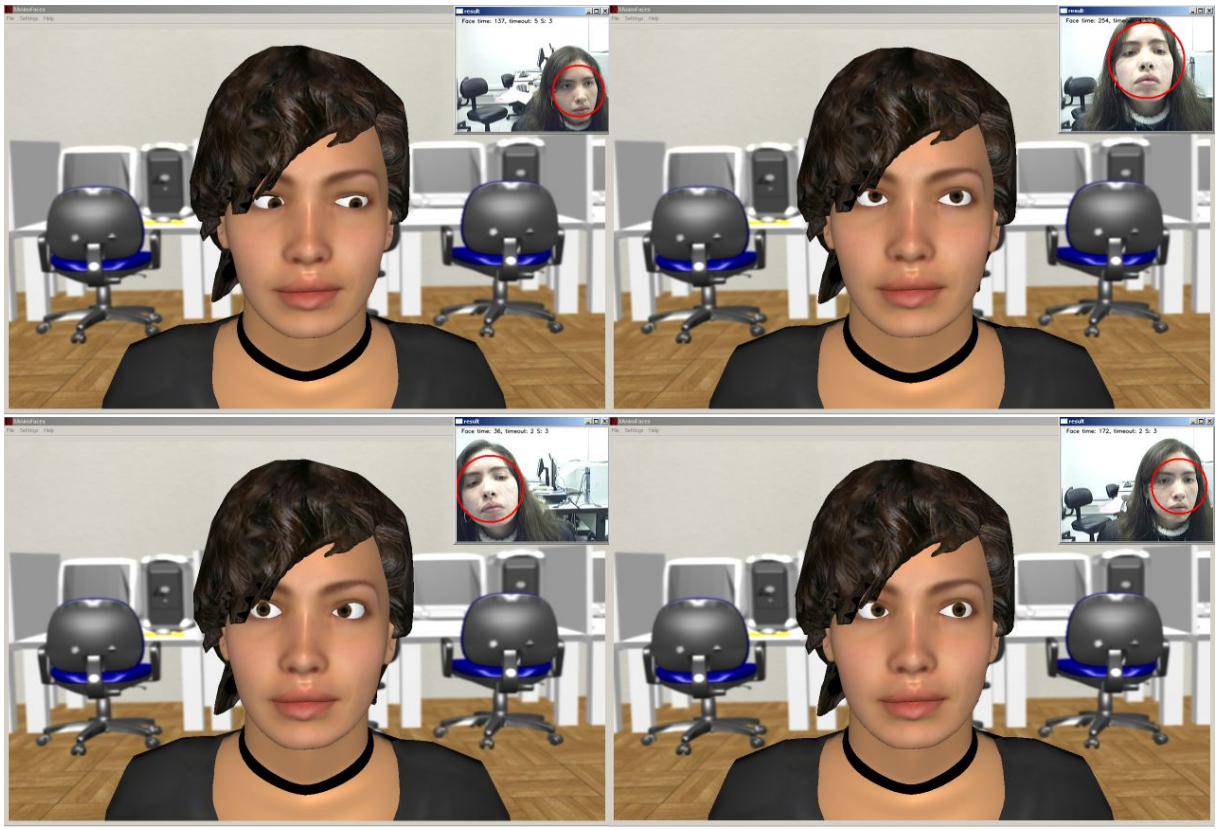


Figure 10: *Snapshots showing our application following user faces. The top right window shows the detection of the user's face.*

CASSELL, J., VILHJÁLMSSON, H. H., AND BICKMORE, T. 2001. Beat: the behavior expression animation toolkit. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 477–486.

COSI, P., DRIOLI, C., TESSER, F., AND TISATO, G. 2005. Interface toolkit: a new tool for building ivas. 75–87.

COSI, P., MAGNOCALDOGNETTO, E., AND TISATO, G. 2007. Emotional talking head: The development of "lucia". In *CD Proceedings Workshop "Toni Mian"*. Available in `http://www2.pd.istc.cnr.it/Papers/PieroCosi/cp-TONI2007.pdf`.

COURTY, N., BRETON, G., AND PELÉ, D. 2003. Embodied in a look: Bridging the gap between humans and avatars. In *IVA*, 111–118.

DE ROSIS, F., PELACHAUD, C., POGGI, I., CAROFIGLIO, V., AND CAROLIS, B. D. 2003. From greta's mind to her face: modelling the dynamics of affective states in a conversational embodied agent. *Int. J. Hum.-Comput. Stud. 59*, 1-2, 81–118.

DIPAOLA, S., AND ARYA, A. 2007. A framework for socially communicative faces for game and interactive learning applications. In *Future Play '07: Proceedings of the 2007 conference on Future Play*, ACM, New York, NY, USA, 129–136.

EKMAN, P., AND FRIESEN, W. 1978. *Facial Action Code System*. Consulting Psychologists Press, Inc., Palo Alto, CA.

EKMAN, P. 1999. Facial expressions. In *Handbook of Cognition and Emotion*, Dalgleish and M. Power, Eds. John Wiley & Sons, ch. 16.

HUANG, H.-H., NISHIDA, T., CEREKOVIC, A., PANDZIC, I. S., AND NAKANO, Y. 2008. The design of a generic framework for integrating eca components. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 128–135.

LEE, S. P., BADLER, J. B., AND BADLER, N. I. 2002. Eyes alive. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 637–644.

NOT, E., BALCI, K., PIANESI, F., AND ZANCANARO, M. 2005. Synthetic characters as multichannel interfaces. In *ICMI '05: Proceedings of the 7th international conference on Multimodal interfaces*, ACM, New York, NY, USA, 200–207.

PANDZIC, I. S., AND FORCHHEIMER, R., Eds. 2003. *MPEG-4 Facial Animation: The Standard, Implementation and Applications*. John Wiley & Sons, Inc., New York, NY, USA.

PARKE, F. 1982. Parameterized models for facial animation. *IEEE Computer Graphics and Applications 2*, 9, 61–68.

PERLIN, K. 1985. An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3, 287–296.

PERLIN, K. 1997. Layered compositing of facial expression. In *ACM SIGGRAPH - Technical Sketch*.

PYUN, H., KIM, Y., CHAE, W., KANG, H. W., AND SHIN, S. Y. 2003. An example-based approach for facial expression cloning. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 167–176.

QUEIROZ, R. B., BARROS, L. M., AND MUSSE, S. R. 2007. Providing expressive gaze to virtual animated characters in interactive applications. In *SBGames 2007*, vol. 1, 197–206.

RODRIGUES, P. S. L. 2007. *Um Sistema de Geração de Expressões Faciais Dinâmicas em Animações Faciais 3D com Processamento de Fala*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro.

RUTLEDGE, L. 2001. Smil 2.0: Xml for web multimedia. *Internet Computing, IEEE 5*, 5 (Sep/Oct), 78–84.

SIFAKIS, E., NEVEROV, I., AND FEDKIW, R. 2005. Automatic determination of facial muscle activations from sparse motion capture marker data. *ACM Trans. Graph. 24*, 3, 417–425.

SMID, K., PANDZIC, I., AND RADMAN, V. 2004. Autonomous speaker agent. *Proceedings of Computer Animation and Social Agents Conference (CASA'04)* (July).

VILHJÁLMSSON, H., CANTELMO, N., CASSELL, J., CHAFAI, N. E., KIPP, M., KOPP, S., MANCINI, M., MARSELLA, S., MARSHALL, A. N., PELACHAUD, C., RUTTKAY, Z., THÓRISSON, K. R., VAN WELBERGEN, H., AND VAN DER WERF, R. J. 2007. The behavior markup language: Recent developments and challenges. In *Intelligent Virtual Agents*. Springer Link.

VIOLA, P., AND JONES, M. 2001. Rapid object detection using a boosted cascade of simple features. *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on 1*, I–511–I–518 vol.1.

WANG, A., EMMI, M., AND FALOUTSOS, P. 2007. Assembling an expressive facial animation system. In *Sandbox '07: Proceedings of the 2007 ACM SIGGRAPH symposium on Video games*, ACM, New York, NY, USA, 21–26.

Um Algoritmo Evolutivo para Aprendizado On-line em Jogos Eletrônicos

Marcio Kassouf Crocomo Eduardo do Valle Simões
Laboratório de Robótica Móvel, Universidade de São Paulo, Brasil

Figura 1: Imagem do jogo produzido

Abstract

Abstract. The goal of this work is to verify if it is possible to apply Evolutionary Algorithms to on-line learning in computer games. Some authors agree that evolutionary algorithms do not work properly in that case. With the objective of contesting this affirmation, this work was performed. To accomplish the goal of this work, a computer game was developed, in which the learning algorithm must create intelligent and adaptive strategies to control the non-player characters using an evolutionary algorithm. Therefore, the aim of the evolutionary algorithm is to adapt the strategy used by the computer according to the player's actions during the game. A review on Evolutionary Computation and the techniques used to produce intelligent behaviors for the computer controlled characters in modern game is presented, exposing the advantages, the problems and some applications of each technique. The proposed game is also explained, together with the implemented algorithms, the experiments and the obtained results. Finally, it is presented a comparison between the implemented algorithm and the Dynamic Script technique. Thus, this work offers contributions to the fields of Evolutionary Computation and Artificial Intelligence applied to games.

Keywords: games, Evolutionary Algorithms, Artificial Intelligence, online adaptation.

Authors' contact:
marciokc@gmail.com
simoes@icmc.usp.br

1. Introdução

Nos últimos anos, muita ênfase foi dada ao realismo dos jogos eletrônicos, fazendo surgir uma variedade de motores gráficos, físicos, de áudio e de Inteligência Artificial (IA) [Bittencourt e Osório 2006]. Grandes esforços foram empregados principalmente na ambientação gráfica utilizada. Porém, já há algum tempo, a previsão para o mercado de jogos era a de que o seu próximo foco seria nos personagens que se comportam de forma realista e podem aprender e se adaptar, ao invés de personagens com maiores resoluções e com mais quadros por segundo [Sweetser 2002]. Atualmente, é possível observar essa tendência em empresas que possuem jogos que têm se destacado pela qualidade de sua IA, como Black & White[1] e The Sims[2].

Máquinas de estado finito, nesse contexto, vêm sendo empregadas na elaboração das estratégias utilizadas pelo computador com uma freqüência maior do que outros métodos nos jogos atuais [Ponsem 2004]. Isso se deve ao fato de serem fáceis de implementar e geralmente suficientes para atingir o propósito desejado. No entanto, essa técnica torna previsível a estratégia utilizada pelo computador. Uma alternativa promissora para contornar os comportamentos previsíveis é a utilização de Algoritmos Evolutivos (AEs) [Yannakakis 2005].

Os AEs possuem diversas aplicações em jogos [Yannakakis 2005], como por exemplo, adaptar as estratégias utilizadas pelo computador àquelas utilizadas pelo usuário, encontrar caminhos e evoluir comportamentos de personagens controlados pelo computador. Apesar das possibilidades de suas aplicações serem numerosas, os AEs encontram resistência em serem utilizados pelos desenvolvedores de jogos. Alguns autores afirmam que essa técnica (assim como a técnica de Redes Neurais Artificiais) permite a obtenção de comportamentos "não

[1] Lionhead Studios. **Black & White 2.** Disponível em: <http://www.lionhead.com/bw2/>. Acesso em: 16 ago. 2007.
[2] Eletronic Arts. **The Sims.** Disponível em: <http://thesims.ea.com/>. Acesso em: 16 ago. 2007.

aceitáveis" durante o processo de aprendizado dos personagens [Ponsem 2004]. Entre esses autores, Spronck [2004] afirma que AEs não são uma alternativa que funcione na prática para o aprendizado on-line em jogos, por não satisfazerem os critérios de eficiência e efetividade necessários. Este artigo é uma resposta a essas afirmações, pois propõe um AE que satisfaça esses requisitos.

Dessa forma, é esperado com este trabalho contribuir para o campo de IA em jogos respondendo a pergunta: É possível construir um AE para o aprendizado on-line de jogos que possa ser utilizado na prática, por satisfazer os requisitos de rapidez, efetividade, robustez e eficiência?

2. Trabalhos Relacionados

Diversos trabalhos envolvendo IA em jogos podem ser encontrados, utilizando técnicas como, por exemplo, Lógica Fuzzy, Flocking, Árvores de Decisão, Máquinas de Estado Finito, Scripting, RNAs e AEs. Uma revisão do uso destas técnicas em jogos pode ser encontrada em [Sweetser 2002]. Com relação ao aprendizado em jogos, o mesmo pode ocorrer de duas formas: on-line e off-line [Spronck et al. 2004]. O aprendizado off-line é aquele onde não é necessária uma interação com o usuário, podendo ocorrer até mesmo antes do jogo ser lançado. Por sua vez, o aprendizado on-line permite que no decorrer do jogo, a adaptação da estratégia utilizada pelo computador em relação ao comportamento do jogador. Spronck et al. [2004] afirma que AEs não satisfazem os dois últimos dos quatro requisitos apontados por ele como necessários no aprendizado on-line e, por isso, não são candidatos a serem utilizados nesses jogos. Os requisitos levantados por esse autor são:

- Rápido: o algoritmo de aprendizado deve ser computacionalmente rápido, pois é realizado durante a execução. Dessa forma, um algoritmo lento atrapalha o desempenho do jogo.
- Robusto: o mecanismo de aprendizado deve suportar uma quantidade de aleatoriedade significativa, normalmente presente nos jogos comerciais.
- Efetivo: as estratégias adaptadas devem ser pelo menos tão desafiadoras quanto as programadas explicitamente. P. Spronck afirma que este requisito exclui métodos de aprendizado aleatórios, como os AEs, pois permitem a geração de estratégias muito ruins para serem utilizadas contra o jogador.
- Eficiente: um pequeno conjunto de testes deve ser suficiente para que o aprendizado ocorra. P. Spronck diz que este requisito exclui técnicas de aprendizado lento, como RNAs, AEs e Aprendizado por Reforço, pois tomariam muito tempo dos jogadores para poderem ajustar suas estratégias.

No trabalho de Spronk et al. [2004] é afirmado que AEs e Redes Neurais não são técnicas utilizáveis na prática para a obtenção de um bom aprendizado on-line em jogos e uma técnica chamada Dynamic Scripting (DS) é proposta como alternativa. Como um contra-exemplo para tal afirmação, o trabalho de Stanley et al. [2005] descreve um jogo que utiliza RNA em seu aprendizado on-line, demonstrando com sucesso a viabilidade de utilização prática dessa técnica. Tal trabalho ganhou o título de melhor artigo no CIG (IEEE Symposium on Computational Intelligence and Games) 2005. Em um trabalho desenvolvido anteriormente, um AE utilizado no aprendizado on-line apresentou o critério de eficiência, no entanto não o de efetividade [Crocomo et. al 2005].

3. O Jogo Desenvolvido

O jogo implementado segue o modelo do simulador elaborado para a realização dos experimentos de Spronck et al. [2004]. Nesse jogo, o jogador será responsável pela estratégia de quatro personagens, contra quatro personagens controlados pelo computador. O jogo é um simulador das batalhas que ocorrem em jogos de RPG do estilo Baldur's Gate[3]. Esse estilo utiliza os sistemas de jogos mais complexos encontrados em RPGs eletrônicos.

O jogo ocorre com o encontro de dois grupos de personagens. Cada um desses grupos (um controlado pelo computador e outro pelo jogador) é composto por dois personagens magos e dois guerreiros. O equipamento dos personagens é estático. Cada personagem possui duas poções mágicas dentre as três possíveis e cada mago possui sete magias de um total de 21 possíveis. As magias existentes são de variados tipos, como, por exemplo, maldições, invocações, danos e bênçãos.

Cada um dos personagens possui as ações que serão tomadas representadas por scripts compostos por cinco regras para um guerreiro (selecionadas de uma base contendo 20 regras) e 10 regras para um mago (selecionadas de uma base de 50 regras). A Tabela I mostra o exemplo de um script de guerreiro, no qual cada linha representa uma regra.

Tabela 1 – Exemplo de um script

Ação	Alvo	Condições
Beber Poção de Cura	Próprio personagem	Pontos de vida < 50% e personagem possuir poção
Beber Poção contra Paralisação	Próprio personagem	Personagem estar paralisado e possuir poção
Atacar	Oponente com condição paralisado	Existir oponente com condição paralisado e personagem não estar paralisado
Atacar	Mago mais próximo	Existir mago oponente vivo e personagem não estar paralisado
Atacar	Oponente com menor saúde	Personagem não estar paralisado

Quanto mais acima na tabela, maior a prioridade da regra: a cada turno, o guerreiro controlado pelo script da Tabela I testa as condições de

[3] BIOWARE. **Baldur´s Gate.** Disponível em: <http://www.bioware.com/games/baldurs_gate/>. Acesso em: 16 ago. 2007.

sua primeira regra. Caso elas sejam satisfeitas, a ação da regra é executada. Caso contrário, são testadas as condições da próxima regra. Esse procedimento se repete até que alguma ação seja executada. No caso de nenhuma ação poder ser executada, a ação de "passar o turno" entra em execução (isso significa que o personagem abre mão de seu movimento).

As regras que compõem os scripts são formadas a partir de cinco ações básicas: atacar, beber uma poção, se mover, executar uma magia ou passar a vez. O conjunto contendo quatro scripts (dois para magos e dois para guerreiros) constitui uma estratégia a ser usada por um time. Dois times são colocados no mesmo ambiente e entram em combate utilizando, cada um, sua respectiva estratégia. Quando um time fica sem personagens vivos, é considerado derrotado. Se o time do jogador vencer o time controlado pelo computador, um novo time é criado para dar continuidade ao jogo, resultando em uma próxima batalha.

3.1 O Algoritmo Evolutivo Proposto

O AE proposto possui como cromossomos as estratégias de comando de um time composto por quatro personagens: dois guerreiros e dois magos (um conjunto de quatro scripts). Cada regra que compõe esses scripts é considerada um parâmetro do cromossomo sendo avaliado, resultando em um total de 30 regras por cromossomo: 5 para cada guerreiro e 10 para cada mago. O cromossomo descrito pode ser visto na Figura 2.

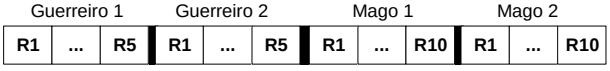


Figura 2: O cromossomo proposto

A população evoluída é composta por quatro cromossomos. O funcionamento do AE proposto é composto por três etapas básicas: geração da população inicial, teste dos indivíduos no ambiente e geração de uma nova população. A Figura 3 ilustra o funcionamento do algoritmo.

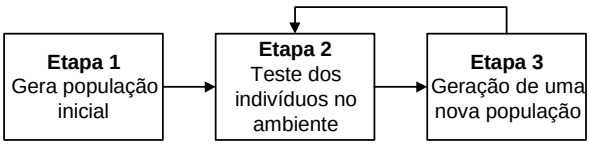


Figura 3: Funcionamento do Algoritmo Evolutivo

Na Etapa 1, é gerada a população inicial contendo quatro cromossomos. Para cada um dos quatro membros da população, são geradas oito estratégias de jogo selecionando-se de forma aleatória as regras para os scripts. É realizado um torneio entre essas estratégias, como ilustra a Figura 4, e a estratégia vencedora é o cromossomo inicial do correspondente membro da população. Os confrontos realizados entre essas estratégias ocorrem colocando as duas para combater no ambiente de jogo em background, ou seja, esse confronto não é visto pelo usuário do jogo. Com a finalidade de gerar os quatro indivíduos da população inicial, são realizados quatro torneios de estratégias, como o mostrado pela Figura 4. Essa Etapa não compromete o desempenho do jogo, pois ocorre antes das partidas com o usuário começarem.

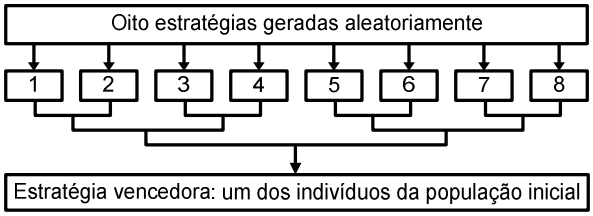


Figura 4: Torneio entre oito estratégias (geradas aleatoriamente). A vencedora irá compor um dos quatro indivíduos da população inicial.

Na Etapa 2, ocorre o confronto entre as estratégias da população sendo evoluída com a estratégia do jogador. Os seguintes procedimentos são realizados:

1. O valor aptidão de cada cromossomo é iniciado como –1, significando que os cromossomos ainda não possuem um valor de aptidão. É selecionado o primeiro cromossomo da população.
2. Os scripts que compõem o cromossomo são utilizados para controlar o grupo de personagens do computador.
3. Ocorre uma batalha entre os personagens do computador e os personagens do jogador. Após o término da batalha, calcula-se uma pontuação para a estratégia utilizada pelo computador, dada pela função de aptidão do grupo: Fit(p), representada pela Equação 1 a seguir.

$$Fit(p) = \begin{cases} 0{,}5 - 0{,}125 \sum_{m \in o} \frac{h(m)}{H(m)} \{\forall n \in p \mid h(n) \le 0\} \\ 0{,}5 + 0{,}125 \sum_{n \in p} \frac{h(n)}{H(n)} \{\exists n \in p \mid h(n) > 0\} \end{cases}$$

Equação 1

sendo:

- F(p): Função de aptidão do grupo p
- H(n): Saúde total do personagem n
- h(n): saúde restante do personagem n após a batalha (número natural entre 0 e H(n))

A aptidão do grupo possui um valor real no intervalo [0,1]. É 0 se o grupo perdeu a luta e 0,5 mais a metade da média da percentagem de saúde restante dos personagens do time.

4. Caso a estratégia do computador tenha sido vitoriosa, a mesma estratégia é mantida para a próxima batalha (retorna para o procedimento 3).
5. Caso o computador tenha sido derrotado pelo jogador, o indivíduo com o cromossomo responsável pela estratégia utilizada recebe um valor de aptidão (*fitness*) igual à média aritmética das pontuações obtidas por sua estratégia no procedimento dois, dada pela Equação 2 abaixo:

$$aptidão\ [j] = \frac{\sum_{i=1}^{n} x_i}{n}$$ **Equação 2**

sendo:

- j: cromossomo sendo avaliado
- aptidão[j] : valor de aptidão do cromossomo j
- n: número de vitórias + 1 (número de batalhas utilizando a estratégia de j)
- xi: pontuação obtida na i-ésima batalha dada pela Equação 1

6. Se ainda houver um cromossomo na população sem um valor de aptidão (aptidão = -1), então retorna para o procedimento 2, testando o cromossomo em questão. Caso contrário, a etapa dois chega ao fim.

Após todos os indivíduos serem avaliados, é executada a etapa 3, com o objetivo de aproveitar as estratégias que obtiveram maior sucesso para gerar uma nova população, da qual é esperado um maior potencial para superar a estratégia utilizada pelo jogador. Para se gerar a nova população, é inicialmente utilizado o elitismo [Bramlette 1991] como técnica de seleção, escolhendo o melhor indivíduo da população atual, isto é, o indivíduo com maior valor de aptidão (chamado de "pai" da população). Após o "pai" da população ser selecionado, o operador evolutivo de crossover [Bramlette 1991] é responsável por determinar três cromossomos da nova população, tendo 50% de chance de receber o valor do cromossomo do "pai" e 50 % de chance de receber o valor do cromossomo antigo; o quarto cromossomo é exatamente igual ao cromossomo do "pai".

Após realizado o crossover, cada parâmetro do cromossomo resultante possui 5% de chance de ser substituído por uma regra aleatória que não pertença ao script em questão. Após a nova população ter sido gerada, a mesma é avaliada chamando novamente a etapa dois. O laço entre as etapas dois e três (Figura 3) representa a capacidade de adaptação do algoritmo, presente durante todo o jogo. O laço só termina no momento em que o jogo for encerrado pelo jogador.

3.2 Os experimentos

Com a finalidade de responder a questão apresentada ao final do capítulo 1 (Introdução), é necessário verificar se o AE implementado durante o trabalho apresenta os quatro requisitos para um aprendizado on-line utilizável na prática: Rápido, Efetivo, Robusto e Eficiente. O AE proposto foi projetado de maneira a cumprir pelo menos três dos requisitos:

• Rápido: o AE proposto é rápido, pois funciona entre as batalhas exigindo apenas uma pequena quantidade de operações lógicas e aritméticas. Desta forma a próxima batalha é rapidamente carregada, sem que o processamento feito pelo computador seja notado pelo usuário, ou seja, o jogador não precisa ficar esperando uma grande quantidade de processamento para continuar jogando.

• Robusto: o AE proposto é robusto, pois nenhuma regra é excluída em momento algum, ou seja, todas as regras podem voltar a ser utilizadas em algum momento do jogo devido ao operador evolutivo de mutação. Desta forma, mesmo que uma regra seja prejudicada devido ao estilo de um jogador, a mesma não é excluída permanentemente. Ela poderá reaparecer devido à mutação e caso o jogador mude de estratégia, ela poderá ter um efeito positivo na pontuação, sendo incorporada na população de scripts.

• Efetivo: P. Spronck afirma que AEs são incapazes de garantir que as soluções geradas sejam pelo menos tão desafiadoras quanto soluções projetadas manualmente [Spronck et al. 2004]. No entanto, também afirma que a técnica DS desenvolvida sempre produz soluções pelo menos tão desafiadoras quanto aquelas projetadas manualmente, por utilizar regras baseadas em domínio de conhecimento (as regras pertencentes à base de regras são projetadas manualmente) [Spronck et al. 2004]. O AE proposto neste trabalho possui como função realizar a seleção de regras também baseadas em domínio de conhecimento. Sendo assim, utilizando a mesma argumentação de P. Spronck, o AE proposto também cumpre o requisito de ser efetivo.

Foi necessário realizar experimentos para verificar se o quarto requisito (eficiência) é cumprido pelo algoritmo. Para esta finalidade, foram repetidos os testes realizados durante o trabalho de P. Spronck para verificar a eficiência da técnica DS. Tais experimentos se baseiam em projetar scripts manualmente que simulem um jogador humano. Após o script ser projetado, o jogo é realizado, sendo que um conjunto de personagens (personagens do jogador) é controlado pelo script pré-concebido, e os outros (personagens do computador) controlados pelos scripts gerados pelo AE proposto. As estratégias projetadas para simular um jogador humano são:

• Ofensivo: os personagens do jogador possuem como prioridade diminuir a vida de seus oponentes na maior velocidade possível: guerreiros sempre atacam o inimigo mais próximo, enquanto os magos usam suas magias de dano mais fortes.

• Deteriorante: os guerreiros iniciam a batalha utilizando poções contra paralisia; após isso atacam o inimigo mais próximo. Os magos utilizam todas as suas magias para debilitar os oponentes (como, por exemplo, magia de paralisação) durante os primeiros turnos.

• Amaldiçoante: guerreiros sempre atacam o inimigo mais próximo; os magos utilizam magias de invocação, redução de atributos, e magias de controle.

• Defensivo: guerreiros começam bebendo poções que reduzam dano do elemento fogo (diminuindo dano de algumas magias); após isso, atacam o inimigo mais

próximo. Os magos usam todas suas magias defensivas e magias de invocação.

As técnicas propostas até o momento não apresentam variações durante a batalha. Isto geralmente não ocorre com um oponente humano, portanto outras três técnicas compostas foram criadas:

- Técnica aleatória: a cada batalha, uma das quatro táticas apresentadas anteriormente é selecionada de maneira aleatória.
- Técnica aleatória para cada personagem: para cada batalha, cada um dos personagens utiliza aleatoriamente um script relativo a uma das primeiras quatro técnicas apresentadas anteriormente. A escolha de um script para um personagem não depende da escolha dos scripts para os outros personagens do grupo.
- Mudança de estratégias: O grupo começa utilizando uma das quatro primeiras técnicas aleatoriamente. Enquanto a técnica utilizada pelo grupo é vitoriosa, ela é mantida; quando a técnica é derrotada, outra é selecionada.

Para avaliar a eficiência das técnicas desenvolvidas pelo AE em relação às técnicas apresentadas para simular o jogador, a cada batalha foi calculada a aptidão média dos grupos durante as últimas 10 batalhas. Quando este valor for maior para o grupo controlado pelo computador, é dito que a estratégia do computador dominou a estratégia do usuário. A partir da estatística apresentada, serão verificados dois valores:

1. Ponto de equilíbrio médio: número da primeira batalha após a qual o grupo controlado pelo computador domina o grupo do usuário por pelo menos 10 batalhas consecutivas.
2. Ponto de equilíbrio absoluto: número da primeira batalha após a qual um número consecutivo de batalhas em que o grupo do computador vence o grupo do usuário nunca é seguido por um número maior de batalhas consecutivas em que o grupo do jogador vence o grupo do computador.

Os valores destes pontos de equilíbrio são avaliados para revelar a eficiência do algoritmo, sendo que quanto menores os valores encontrados mais eficientes são os algoritmos. As simulações realizadas são mostradas abaixo.

1. DS X Jogador Humano
2. AE X Jogador Humano
3. AE X DS

Os resultados coletados durante estas simulações foram avaliados com o objetivo de verificar a eficiência do AE implementado. Estes resultados também possibilitam a comparação entre os dois algoritmos de aprendizado on-line: o DS proposto por P. Spronck e o AE proposto neste trabalho.

3.3 Resultados Obtidos

Depois de implementados os algoritmos de aprendizado, foram realizados experimentos para verificar se os mesmos apresentam capacidade de adaptação. Para isso, os personagens adaptados pela técnica DS disputaram duas mil partidas contra personagens com a estratégia ofensiva, explicada no capítulo anterior. Em cada partida, as pontuações das duas equipes são comparadas. A equipe com maior pontuação é considerada a vencedora na partida em questão.

Para melhor compreensão dos dados coletados neste experimento, os resultados foram representados em um gráfico, utilizando média móvel com largura 10. A equação da média móvel utilizada está representada pela Equação 2 abaixo:

$$p_i = \frac{\sum_{k=i-9}^{i} f(k)}{10} \quad , i \geq 10$$

Equação 2

sendo:
p_i: "pontuação média" explicada no texto acima.
***f(k)*:** A pontuação da equipe sendo avaliada, explicada pela função de avaliação da técnica DS.
A função f(k) acima, é dada pela Equação 3 abaixo. Tal equação é utilizada na técnica DS de maneira análoga a Equação 1, utilizada no AE deste trabalho.

$$F(p) = \begin{cases} 0 & \{\forall n \in p \mid h(n) \leq 0\} \\ 0{,}5 + 0{,}125 \sum_{n \in p} \frac{h(n)}{H(n)} & \{\exists n \in p \mid h(n) > 0\} \end{cases}$$

Equação 3

sendo:

- F(p): Função de aptidão do grupo p
- H(n): Saúde total do personagem n
- h(n): saúde restante do personagem n após a batalha (número natural entre 0 e H(n))

Os pontos de equilíbrio médio e absoluto encontrados no experimento representado pela Figura 5 foram 54 e 92 respectivamente. O mesmo experimento foi realizado para se verificar a capacidade de adaptação do AE. Desta vez, para a exibição dos dados, foi utilizada a média móvel feita com a pontuação obtida pela função de avaliação da equipe utilizada no AE, representada pela Equação 1 Os dados coletados desta forma estão representados no gráfico da Figura 6, e apresentaram pontos de equilíbrio médio e absoluto de 41 e 43 respectivamente.

A Figura 5 e a Figura 6 permitem verificar que os algoritmos implementados fornecem ao computador a capacidade de gerar estratégias que vençam a estratégia sendo enfrentada. Ambos os algoritmos necessitaram de um pequeno número de partidas para superar a estratégia pré-elaborada, sendo isso uma evidência de que os algoritmos satisfazem o critério de eficiência.

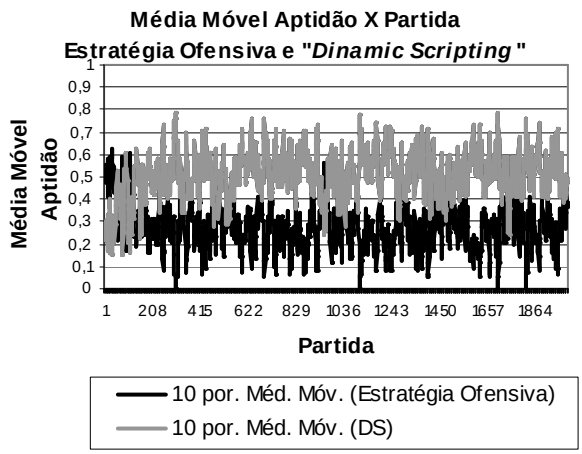


Figura 5 - DS Disputando Contra a Estratégia Ofensiva, Gráfico de Média Móvel Aptidão por Partida

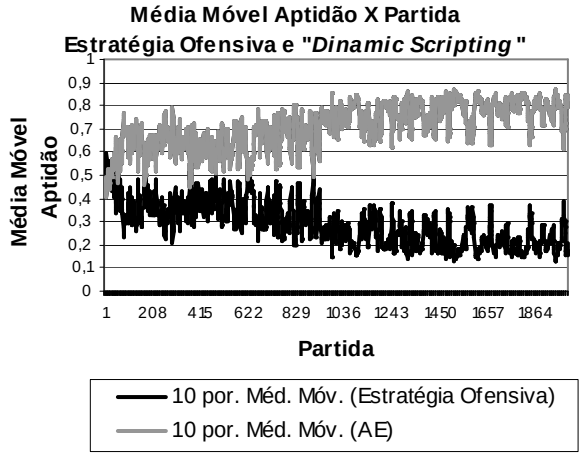


Figura 6 - AE Disputando Contra a Estratégia Ofensiva, Gráfico de Média Móvel Aptidão por Partida

Para cada uma das estratégias pré-elaboradas explicadas no capítulo anterior, foram realizados 21 testes como os descritos acima contra cada algoritmo de aprendizado implementado, e para cada uma das estratégias compostas (técnica aleatória, aleatória para cada personagem, e mudança de estratégia) foram realizados 11 testes. O número de testes realizados foi assim escolhido para reproduzir os testes realizados pelo criador da técnica DS no artigo [Spronck et al. 2004]. Os resultados baseados nos pontos de equilíbrio médio e absoluto podem ser visualizados na Tabela II e na Tabela III. É importante observar que, de acordo com as definições dos pontos de equilíbrio, quanto menores os valores encontrados, mais eficiente é o algoritmo.

A Tabela I representa os resultados obtidos pela técnica DS neste trabalho. A tabela referente aos mesmos experimentos realizados pelo criador da técnica DS pode ser encontrada em [Spronck et al. 2004]. Nos dois trabalhos, a estratégia "Mudança de Estratégia" foi a indicada pela mediana como a mais difícil de ser vencida. É possível verificar também que as estratégias pré-elaboradas possuem algumas diferenças:

Tabela II – Resultados obtidos confrontando os scripts desenvolvidos pela técnica DS com as estratégias pré-elaboradas da primeira coluna.

Dynamic Scripting – Ponto de Equilíbrio Médio				
Estratégias	Menor	Maior	Média	Mediana
Ofensiva	12	1129	88	38
Deteriorante	11	36	20	18
Amaldiçoante	11	51	26	22
Defensiva	11	48	18	12
Aleatória	11	214	70	53
Personagens Aleatórios	11	35	17	16
Mudança de Estratégia	11	102	60	56
Dynamic Scripting – Pontos de Equilíbrio Absoluto				
Estratégias	Menor	Maior	Média	Mediana
Ofensiva	15	1919	158	47
Deteriorante	1	832	112	28
Amaldiçoante	6	115	39	31
Defensiva	1	80	21	12
Aleatória	70	---	406[4]	174
Personagens Aleatórios	1	70	21	18
Mudança de Estratégia	62	1938	507	254

1. A estratégia "Deteriorante" apresentou maior dificuldade a ser vencida neste trabalho do que em [Spronck et al. 2004].
2. A estratégia Defensiva apresentou maior facilidade a ser vencida neste trabalho do que em [Spronck et al. 2004].
3. A estratégia "Aleatória" apresentou neste trabalho um caso em que a técnica DS não conseguiu obter um ponto de equilíbrio absoluto (tal verificação encontra-se explicada mais adiante). Tais diferenças são naturais dado que o ambiente construído neste trabalho não é exatamente igual, embora tenha sido montado de maneira a se assemelhar ao ambiente criado por Spronck [2004]. Além disso, as estratégias foram montadas por especialistas diferentes, embora sigam a mesma idéia nos dois trabalhos.

O importante é verificar a eficiência da técnica DS, independente das diferenças do ambiente e das estratégias enfrentadas. Com relação a isso, observações feitas em [Spronck et al. 2004] são também válidas para a Tabela II e encontram-se realizadas mais adiante.

Tabela III – Resultados obtidos confrontando os scripts desenvolvidos pelo AE com as estratégias pré-elaboradas da primeira coluna.

Algoritmo Evolutivo – Ponto de Equilíbrio Médio				
Estratégias	Menor	Maior	Média	Mediana
Ofensiva	11	167	61	40
Deteriorante	11	170	30	17
Amaldiçoante	11	354	54	22
Defensiva	11	124	18	11
Aleatória	11	214	70	53
Personagens Aleatórios	11	35	17	16
Mudança de Estratégia	11	102	60	56

[4] Neste caso, o valor da média foi calculado utilizando apenas os 10 testes em que um ponto de equilíbrio absoluto foi encontrado.

Algoritmo Evolutivo – Ponto de Equilíbrio Absoluto				
Estratégias	Menor	Maior	Média	Mediana
Ofensiva	1	1331	192	107
Deteriorante	1	328	81	28
Amaldiçoante	1	1326	174	38
Defensiva	1	137	13	4
Aleatória	6	1798	505	102
Personagens Aleatórios	1	340	67	25
Mudança de Estratégia	4	1996	776	506

Analisando as duas tabelas (Tabela II e Tabela III), é possível observar que:

1. Em todos os experimentos, e para os dois pontos de equilíbrio, a média sempre possui valor maior que a mediana. Isso se explica pelo fato de serem raras as ocorrências de pontos de equilíbrio com valores altos.
2. Ambos os algoritmos mostraram médias e medianas factíveis para os pontos de equilíbrio médio, isto é, para que a adaptação ocorra, é necessário que o jogador jogue uma quantidade de partidas aceitável (levando em consideração que cada partida leva geralmente menos de um minuto no jogo implementado). Tal constatação indica que os algoritmos são eficientes.
3. Comparando as duas tabelas, é difícil afirmar qual é o melhor algoritmo, sendo que em alguns testes, a técnica DS obteve melhores resultados, e em outros a utilização do AE teve melhor desempenho.
4. Ao enfrentar a estratégia "Aleatória", A técnica DS não conseguiu obter um turning point absoluto em um dos experimentos. A ocorrência de tal acontecimento é dada ao fato de que em algumas partidas, devido ao fator sorte, a técnica DS pode aumentar o peso de regras não adequadas, e até mesmo excluir regras necessárias para obter uma estratégia vencedora. Tal limitação do algoritmo já havia sido constatada pelos desenvolvedores da técnica. O fato de permitir que o peso mínimo para uma regra seja zero, resulta em uma grave desvantagem, pois com o passar do tempo, a técnica perde a capacidade de adaptar suas estratégias.

Para chegar a uma melhor comparação entre os algoritmos, a próxima etapa dos testes foi realizada, confrontando no jogo as duas técnicas. Ou seja, enquanto um grupo de personagens é evoluído pelo AE, o grupo oponente foi adaptado pela técnica DS. Os pontos de equilíbrio obtidos pelo AE podem ser verificados na Tabela IV. Enquanto os obtidos pela técnica DS se encontram na Tabela V. Para obtenção destes pontos de equilíbrio foram realizados 21 testes, cada um composto por duas mil partidas.

Tabela IV– Pontos de Equilíbrio obtidos pelo AE ao confrontar a técnica DS

AE – Ponto deEquilíbrio Médio			
Menor	Maior	Média	Mediana
11	393	49	12
AE – Ponto deEquilíbrio Absoluto			
Menor	Maior	Média	Mediana
2	--	598	391

Tabela V– Pontos de Equilíbrio obtidos pela técnica DS ao confrontar o AE

DS – Ponto deEquilíbrio Médio			
Menor	Maior	Média	Mediana
11	240	55	38
DS – Ponto deEquilíbrio Absoluto			
Menor	Maior	Média	Mediana
76	--	1172	1997

Olhando os pontos de equilíbrio obtidos a partir deste experimento, é possível observar que:

1. Tanto o AE quanto a técnica DS encontraram casos em que não foi possível obter um ponto de equilíbrio absoluto. Dos 21 testes realizados, o AE não conseguiu obter um ponto de equilíbrio absoluto em 2 testes. Já a técnica DS não foi capaz de obter este ponto de equilíbrio em 5 testes.
2. A mediana do ponto de equilíbrio absoluto obtida pela técnica DS foi de 1997. Este valor é muito alto considerando o número de partidas realizadas em cada teste (2000), o que é um indicador de que o AE dominou a técnica DS na maioria das partidas.
3. As médias e as medianas apresentaram melhores resultados para as estratégias desenvolvidas pelo AE.

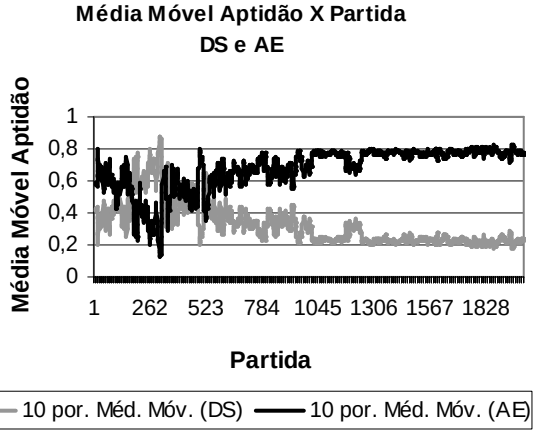


Figura 7 - AE Disputando Contra a Técnica DS, Gráfico de Média Móvel Aptidão por Partida

A fim de verificar as constatações 2 e 3 realizadas, foram verificados os comportamentos dos testes realizados. Foi constatado que o grupo adaptado pela técnica DS perde a capacidade de adaptação após alguns jogos. Tal constatação é coerente com a observação realizada no experimento anterior, no qual a técnica DS não encontra um ponto de equilíbrio absoluto ao confrontar a estratégia "Aleatória". A Figura 7 mostra um dos testes realizados onde é possível visualizar a perda da capacidade de adaptação da técnica DS.

Para colher dados capazes de confirmar a superioridade do AE sobre a técnica DS ao longo de um jogo composto por 2000 partidas, foi utilizado um método que utiliza como critério o erro máximo de estimação com um nível de confiança de 95% [Bussab e Morettin 2006]. O método utilizado é explicado pela Figura 8.

1 Repita
2 Realize um jogo composto por 2000 partidas;

3 Calcule a freqüência das partidas vencidas pelo grupo adaptado pelo Algoritmo Evolutivo;

Até que (Número de Jogos > 200**)** ou (erro máximo < 0.05 * (Média dos valores calculados no passo 3)).

Figura 8 - Método Adotado para a Coleta dos Dados

No passo 2, o jogo é realizado como no experimento relatado anteriormente, realizando 2000 partidas. No passo 3 o valor calculado da amostra (freqüência das partidas vencidas pelo grupo adaptado pelo Algoritmo Evolutivo durante o jogo i) é chamado de x_i e dado pela razão abaixo:

$$x_i = \frac{VAE}{NP}$$

sendo:

- ***VAE*** Número de vitórias obtidas pelo grupo cuja estratégia é evoluída pelo AE
- ***NP*** Número total de partidas realizadas, (2000 neste caso).

No critério de parada, o erro máximo (e) é calculado a partir dos valores amostrados até o momento [Bussab e Morettin 2006]. A fórmula para cálculo deste erro é dada pela Equação 7:

$$e = \sqrt{\frac{s^2}{n}}$$

Equação 7

sendo:

- e erro máximo
- s2 variância amostral
- n número de jogos realizados (amostras)

A variância amostral (**s^2**) é estimada através da Equação 8 abaixo.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Equação 8

sendo:

- xi Valor da amostra
- $\bar{x}$ Média das amostras
- n Nmero de jogos realizados (amostras)

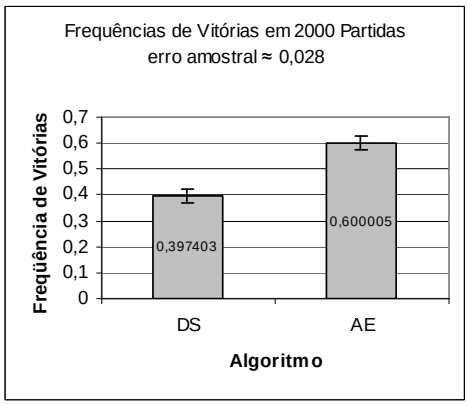


Figura 9 – Freqüência de Vitórias das Técnicas AE e DS em 2000 Partidas

Os resultados obtidos encontram-se no gráfico da Figura 9, onde as colunas indicam as freqüências obtidas de vitórias do AE e da técnica DS, e o intervalo indicado sobre as colunas representa o intervalo de possíveis valores para as freqüências esperadas, considerando o erro máximo encontrado.

Tais resultados permitem concluir, com um nível de confiança de 95%, que em um jogo composto por 2000 partidas, as estratégias geradas pelo AE vencem as estratégias geradas pela técnica DS na maioria das vezes. A afirmação é possível de ser realizada, pois é visível no gráfico que o pior caso do AE (freqüência encontrada de vitórias do AE **menos** o erro máximo) é melhor que o melhor caso da técnica DS (freqüência encontrada de vitórias da técnica DS **mais** erro máximo). Tal resultado é esperado, dada a constatação de que a técnica DS perde a capacidade de adaptar sua estratégia após um determinado número de partidas, sendo que 2000 partidas são suficientes para se verificar tal acontecimento, como visto no experimento anterior. Os casos de empate não foram exibidos no gráfico, pois se mostraram desprezíveis (casos de empates e erro máximo de aproximadamente 0%).

Os resultados explicados até o momento são suficientes para apontar os Algoritmos Evolutivos como uma melhor opção a ser utilizada em jogos nos quais a adaptação deve ocorrer continuamente. Em jogos em que ocorrem poucas partidas, as duas técnicas seriam aceitáveis, pois a técnica DS só perde sua capacidade de adaptação após um número de partidas suficiente para reduzir os pesos de suas regras a zero (a quantidade de partidas varia de acordo com fatores como o espaço de busca e a quantidade de aleatoriedade do jogo). No entanto, isso não permite comparar as duas técnicas com relação ao critério de eficiência, pois o mesmo diz respeito a um número pequeno de partidas para que o aprendizado ocorra. Para comparar este critério, o último experimento foi repetido, mas desta vez o número de partidas por jogos foi diminuído, realizando testes para 300, 200, 100 e 50 partidas por jogo. Os resultados do teste podem ser visualizados na Tabela VI abaixo.

Tabela VI – Comparação das freqüências de vitórias entre o AE implementado e técnica DS. Resultados coletados utilizando 300, 200, 100 e 50 partidas por jogo.

Partidas por jogo	Freqüência de Vitórias		
	DS	AE	Erro amostral ≈
300	0,514	0,475	0,026
200	0,505	0,487	0,025
100	0,509	0,479	0,025
50	0,463	0,521	0,023

É valido avisar que a soma das freqüências de uma linha da tabela não é exatamente um, pois existem os casos de empate (que possuem valores a partir da quarta casa decimal) e também devido a arredondamentos realizados pelo computador. Estes casos de empate, mais uma vez não foram mostrados na tabela acima, pois se mostraram desprezíveis. Observando os dados resultantes dos experimentos, é possível verificar que as freqüências de vitórias obtidas

para o AE e para a técnica DS foram similares. Só é possível concluir que a eficiência de uma técnica de aprendizado 'A' é superior à eficiência da outra técnica de aprendizado 'B', se o pior caso de 'A' for superior ao melhor caso de 'B'. Ao dizer pior ou melhor caso, é considerado o erro máximo, assim como feito no experimento com 2000 partidas por jogo. De outra maneira, podemos afirmar a superioridade da técnica 'A' com relação à técnica 'B' caso a inequação abaixo seja satisfeita [Bussab e Morettin 2006]:

$$FVA - ea > FVB + eb$$

Sendo:

- FVA Freqüência de Vitórias do Grupo cujos scripts foram adaptados pela técnica 'A'
- FVB Freqüência de Vitórias do Grupo cujos scripts foram adaptados pela técnica 'B'
- ea erro máximo da Percentagem de Vitórias do Grupo cujos scripts foram adaptados pela técnica 'A'
- eb erro máximo da Percentagem de Vitórias do Grupo cujos scripts foram adaptado pela técnica 'B'

Como conseqüência dos casos de empate terem se mostrado desprezíveis, o erro amostral aproximado (representado na Tabela VI) é igual para **ea** e **eb** (ea ≈ eb). Em nenhum dos resultados apresentados pela Tabela VI pode ser constatada superioridade na eficiência da técnica DS sobre o AE, já que, embora alguns experimentos apontem a técnica DS como vencedora (experimentos com 300, 200 e 100 partidas por jogo), ao considerar o erro amostral encontrado tal superioridade não pode ser admitida, pois nenhum dos experimentos torna verdadeira a inequação apresentada. De maneira análoga, ao se comparar a superioridade do AE sobre a técnica DS, é possível verificar a superioridade do AE no último experimento realizado (50 partidas por jogo), pois os dados coletados tornam válida a inequação verificada.

É possível concluir, com base nos dados obtidos, que a eficiência apresentada pelo AE se mostrou melhor ou próxima a da técnica DS, que já teve sua eficiência verificada no trabalho de Spronck. Com tal verificação tornou-se evidente que Algoritmos Evolutivos podem satisfazer o critério de eficiência para um aprendizado *on-line* que funcione na prática. Atingido este objetivo, os experimentos foram finalizados.

3. Conclusão

A proposta deste trabalho foi a de verificar a possibilidade de se construir um Algoritmo Evolutivo (AE) capaz de ser aplicado de fato no aprendizado on-line em jogos, capacidade esta apontada como impossível por alguns autores [Spronck et al. 2004] [Ponsem 2004]. Para o cumprimento de tal objetivo, inicialmente foi realizada uma revisão bibliográfica que apresentou fatos encorajadores, pois possibilitou encontrar trabalhos que diziam ter obtido aprendizado on-line eficiente utilizando Algoritmos Evolutivos e Redes Neurais Artificiais.

Para constatar que o AE funciona de fato no aprendizado on-line, foi verificado se o mesmo satisfaz os requisitos: rapidez, robustez, efetividade e eficiência. O AE proposto foi projetado de maneira a cumprir os primeiros três dos requisitos. Para mostrar a presença destes critérios, foi realizada uma comparação da estrutura do algoritmo a técnica de aprendizado DS que satisfaz os quatro requisitos. Para verificar o critério de eficiência, foi necessário:

1. Produzir um jogo como ambiente de testes assim como uma linguagem de scripting capaz de controlar o comportamento dos personagens. Para a produção deste jogo, foi procurado recriar o ambiente de testes produzido pelos criadores da técnica DS, cujos detalhes explicados no artigo [Spronck et al. 2004] exigiram grande atenção para estarem presentes também no jogo produzido neste trabalho.
2. Propor e codificar um AE, realizando testes sobre a função de avaliação utilizada, e otimizando suas funções evolutivas. Inicialmente a função de avaliação utilizada seria igual a função de avaliação do time utilizada pela técnica DS. Os testes realizados durante esta etapa mostraram a necessidade de se modificar esta avaliação para o funcionamento junto ao AE implementado. A taxa de mutação e o tamanho da população do AE foram definidos empiricamente, realizando alguns testes nesta etapa do projeto.
3. Implementar a técnica DS e realizar testes para verificar a capacidade de adaptação da mesma. Para isso foi necessário um estudo cuidadoso da técnica a fim de não cometer enganos em sua implementação. Os testes realizados permitiram verificar a correta implementação desta, assim como mostrar sua capacidade de adaptação.
4. Elaborar estratégias de jogo, com o objetivo de simular um jogador humano para permitir a realização de grandes quantidades de testes com as técnicas de aprendizado. Estas estratégias foram extraídas do trabalho de Spronck [2004], e procuraram ser reproduzidas neste trabalho.
5. Reproduzir, para os dois algoritmos de aprendizado, os testes realizados para verificar a eficiência da técnica DS no trabalho onde a mesma foi proposta, assim como realizar testes confrontando diretamente os dois algoritmos de aprendizado. Nesta etapa foi necessário um estudo detalhado sobre os testes realizados no artigo que apresenta a técnica DS [Spronck et al. 2004].
6. Avaliar os dados coletados segundo os critérios estatísticos adotados durante a etapa seis. Dificuldades foram encontradas com o rigor estatístico a ser utilizado. Foi então buscado o auxílio de um especialista da área de estatística, que se mostrou de grande ajuda para o cumprimento desta etapa.

Ao final da última etapa descrita acima, foi possível verificar que o algoritmo de aprendizado proposto neste trabalho satisfaz o requisito de eficiência. Também foi observado que o algoritmo em questão

apresenta eficiência melhor ou similar a da técnica DS, cuja eficiência já havia sido comprovada no trabalho [Spronck et al. 2004]. Tal constatação permite responder à pergunta chave deste trabalho: sim, é possível desenvolver um AE que satisfaça os quatro requisitos: rapidez, robustez, efetividade e eficiência.

Os testes realizados permitiram verificar uma deficiência da técnica DS: esta perde a capacidade de adaptação após um determinado tempo de funcionamento. Uma forma para corrigir tal deficiência seria definindo um peso mínimo de valor maior que zero a ser utilizado. Assim, nenhuma regra seria permanentemente descartada e, portanto, a técnica preservaria sua capacidade de adaptação. É provável que com esta correção a eficiência da técnica diminua, pois regras que já tenham sido testadas e mostradas inadequadas contra a estratégia utilizada continuariam apresentando a possibilidade de serem testadas.

A obtenção de um AE como o implementado neste trabalho traz contribuições para o campo de Inteligência Artificial aplicada a jogos, no qual é reconhecido que as possíveis aplicações para AE são numerosas, embora a técnica encontre certa resistência para ser aceita pelos desenvolvedores de jogos. Tal resistência é reforçada por afirmações como as contestadas e negadas neste trabalho. Portanto, espera-se com este trabalho, que os desenvolvedores de jogos tornem-se mais receptivos às técnicas evolutivas de IA.

Uma aplicação em potencial de AEs seria em jogos como MMORGPs, que estão em destaque atualmente. Este tipo de jogo permite que uma grande quantidade de usuários jogue simultaneamente. A avaliação de seus indivíduos poderia ocorrer de forma paralela, sendo que cada jogador funcionaria como uma função de avaliação para os oponentes que os enfrentassem.

O nível de complexidade do jogo produzido foi escolhido de maneira a ser alto quando comparado a outros jogos mais simples (espaço de busca de tamanho 25*1027 aproximadamente) [Fairclough et al., 2001]. Desta forma, ao se constatar que o AE apresentou bom desempenho neste jogo, é esperado que em jogos similares, mas com soluções contidas em um espaço de busca menor do que o deste projeto o AE apresente um desempenho ainda melhor. Em suma, foram obtidos neste trabalho:

1. Uma revisão sobre técnicas utilizadas na Inteligência Artificial dos jogos atuais.Um jogo[5] com alta complexidade que serve como ambiente de testes para técnicas de aprendizado on-line.
2. Um AE capaz de produzir estratégias inteligentes adaptadas às estratégias utilizadas pelo usuário em tempo de jogo. Além disto, foi verificado que este algoritmo funciona na prática, pois satisfaz quatro requisitos: rapidez, efetividade, robustez e eficiência.
3. Resultados experimentais que: (1) mostram que o AE implementado é mais eficiente ou apresenta eficiência similar a da técnica DS desenvolvida por Spronck; (2) apontam deficiências da técnica DS e sugestões de melhorias para a mesma; e (3) negam afirmações que dão suporte à resistência encontrada para a aceitação de AEs por desenvolvedores de jogos.

Ao se concluir este trabalho, várias alternativas para sua continuação foram encontradas, dentre as quais podem ser citadas:

1. Implementar a correção proposta para a técnica DS e realizar outros experimentos para verificar se o mesmo continua satisfazendo o critério de eficiência.
2. Comparar outros algoritmos de aprendizado com os utilizados neste trabalho.
3. Tentar encontrar AEs que se mostrem ainda mais eficientes que o deste trabalho

[5] **Site do jogo ZantArena.** Disponível em: <http://quadros.no-ip.org/zantarena>. Acesso em: 02 fev. 2008.

Referencias

BITTENCOURT, J. AND OSÓRIO, F., 2006. Motores de jogos para criação de jogos digitais – gráficos, áudio, interface, rede, inteligência artificial e física. *In: Anais da V ERI-MG SBC,* v. 1, 1–36.

BRAMLETTE, M., 1991. Initialization, mutation and selection methods in genetic algorithms for function optimization. *In: Proceedings of the 4th International Conference on Genetic Algorithms,* San Mateo, CA, USA, 100–107.

BUSSAB, W.O., Morettin, P. A. 2006. **Estatística Básica**. 5. ed. SARAIVA. P. 540. ISBN 8502034979.

CROCOMO, M. K.; MIAZAKI, M.; SIMÕES, E.V. 2005. Algoritmos evolucionários no desenvolvimento de jogos adaptativos, In: *Simpósio Brasileiro de Jogos Para Computador e Entretenimento Digital – WJogos, São Paulo, Brail, 342-347.*

FAIRCLOUGH, C. ET AL., 2001. *Research Directions for AI in Computer Games. In: Irish conference on AI and cognitive science, 12. Kildare, AICS, 1 – 12.*

PONSEM, M., 2004. *Improving adaptive game AI with evolutionary learning*. Dissertação de Mestrado, Faculty of Media & Knowledge Engineering, Delft University of Technology.

SPRONCK, P., YEE, N., SHPRINKHUIZEN-KUYPER, I. AND POSTMA, E., 2004. Online adaptation of game opponent AI in theory and practice. *In: Proceedings of the 4th International Conference on Intelligent Games and Simulation,* London, UK, 45–53.

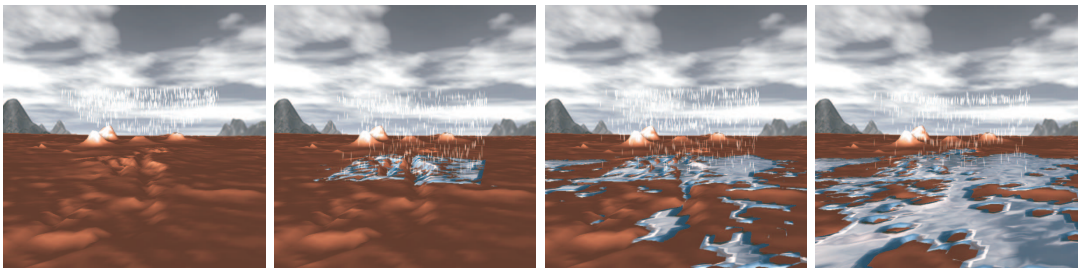
STANLEY, K., BRYANT, B. AND MIIKKULANINEN, R., 2005. Evolving neural network agents in the NERO video game. *In: Symposium on Computational Intelligence and Games,* Piscataway, NJ, USA, IEEE Press, 182–189.

SWEETSER, P., 2002. *Current AI in games: a review*. Relatório Técnico, School of ITEE, University of Queensland.

YANNAKAKIS, G., 2005. *AI in computer games: generating interesting interactive opponents by the use of evolutionary computation*. Tese de doutorado, Faculty of Media & Knowledge Engineering, University of Edinburgh.

A Cellular Automata Framework for Real Time Fluid Animation

Sicilia F. Judice and Bruno Barcellos S. Coutinho and Gilson A. Giraldi
National Laboratory for Scientific Computing



Abstract

In this work, we focus on fluid modeling and animation through Lattice Gas Cellular Automata (LGCA) for computer games. LGCA are discrete models based on point particles that move on a lattice, according to suitable and simple rules in order to mimic a fully molecular dynamics. In this paper, we combine the advantage of the low computational cost of LGCAs and its ability to mimic the realistic fluid dynamics. The new animating framework is composed by two LGCAs models: (a) A $3D$ fluid animation technique; (b) A GPU surface flow animation over terrain models. In this work we highlight the advantages of our proposal for computer games applications. In the experimental results we emphasize the simplicity and power of the proposed models when combined with efficient techniques for rendering.

Keywords:: Real-time Simulation, Cellular Automata, Fluid Animation,Terrain Models

Author's Contact:

{sicilia,lsdelphi,gilson}@lncc.br

1 Introduction

Physically-based techniques for the animation of natural elements like fluids (gas or liquids), elastic, plastic and melting objects, among others, have taken the attention of the computer graphics community. The motivation for such interest rely in the potential applications of these methods and in the complexity and beauty of the natural phenomena that are involved. In particular, techniques in the field of Computational Fluid Dynamics (CFD) have been applied for fluid animation in applications such as virtual surgery simulators and visual effects [Müller et al. 2004a; Müller et al. 2004b; Müller et al. 2003].

The traditional fluid animation methods in computer graphics rely on a top down viewpoint that uses 2D/3D mesh based approaches motivated by the Eulerian methods of Finite Element (FE) and Finite Difference (FD), in conjunction with Navier-Stokes equations of fluids [Stam 2003b; Foster and Metaxas 1997; Hirsch 1988].

Animation of fluids is important for computer games applications in order to immerse players into plausible virtual worlds [Gam 2004]. However, real time is a fundamental requirement in this area which is the main challenge for traditional CFD methods due to the cost of the computational simulation [Stam 2003a].

Recently, we demonstrated the advantages of changing the viewpoint to the bottom up models of the Lattice Gas Cellular Automata (**LGCA**) [Xavier et al. 2005; Barcellos et al. 2007]. These are discrete models based on point particles that move on a regular grid structure (**lattice**), according to suitable and simple rules in order to mimic a fully molecular dynamics [Frisch et al. 1987a]. Particles can only move along the edges of the lattice and their interactions are based on simple collision rules. Such framework needs low computational resources for both the memory allocation and the computation itself. Such models have been applied for scientific application in two-phase flows description (gas-liquid systems, for example), numerical simulation of bubble flows [Inamuro et al. 2004], among others.

Besides, a multiscale technique was applied to demonstrated that a particular type of LGCAs, the so called **FHP** model, can reproduce Navier-Stokes behaviors for $2D$ fluids [Frisch et al. 1986; Wolfram 1996]. However, there is no need to solve Partial Differential Equations (PDEs) to obtain a high level of description. Besides being fast and realistic from the physical viewpoint, it is based on binary variables, and so, there are no numerical instabilities at all [Doolen 1990]. In [Giraldi et al. 2005] we show the FHP capabilities for animating $2D$ fluid-fluid interactions. In [Barcellos et al. 2007] we demonstrated the advantages of a technique inspired in the FHP model for surface water flow simulation in digital terrain models.

In this paper we propose a framework for real time fluid animation based on two elements: (a) A three dimensional fluid simulation model based on the FHP and interpolation techniques; (b) The surface flow simulation in Digital Terrain Models (DTMs) proposed in [Barcellos et al. 2007]. In the former, our proposal consists of regularly distribute independent FHP planes along the x and z directions and perform simple interpolations to generate a $3D$ macroscopic flow. The latter is a LGCA which is based on (water) particles which can move over the terrain surface but the projections of their displacements are constrained to the lattice directions. The DTM is a piecewise linear approximation of the terrain surface built as a digital elevation model (**DEM**); that means, there is a lattice that keeps the elevation of the terrain at each grid node. We can apply the $3D$ fluid simulation method to animate rainfall. On the other hand, we can simulate flood and watershed rainfall by using simple interaction model between the fluid and the terrain.

The main contributions of this work is the development of a particle based framework, combining LGCA techniques and DTMs to create realistic animations of systems that involve $3D$ fluid and fluid-terrain interaction for computer graphics and computer games applications.

The paper is organized as follows. Section 2 gives a review of related works. Section 3 describes the FHP model and our extension for $3D$. In sections 4 and 5 we describe our technique for simulating surface water flow over terrains and its GPU implementation. The interaction between models is discussed on section 6. Section 7 presents experimental results. In section 8 we present the conclusions and future works.

2 Related Work

This work focuses on the animation of three dimensional fluids and fluid-terrain interaction. The former involves numerous works that can be coarsely classified in non-physically and physically based models [Iglesias 2004; Deusen et al. 2004]. Our work belongs to the latter class, which can be subdivided in PDEs and Lattice based techniques [Frisch et al. 1987b; Iglesias 2004]. PDEs methods in-

cludes continuous fluid equation, like the Navier-Stokes ones, and numerical techniques based on discretization approaches that can be Lagrangian Smoothed Particle Hydrodynamics (SPH) [Liu and Liu 2003], method of characteristics [Stam 1999], Moving-Particle Semi-Implicit [Premoze et al. 2003] or Eulerian (Finite Element) ones [Foster and Metaxas 1997].

Lattice based techniques, like HPP, FHP and Lattice Boltzmann methods, work following a different viewpoint [Frisch et al. 1987b; Benzi et al. 1992]. For instance, in the case of HPP and FHP, instead of applying continuous mechanics (and, consequently, PDEs) principles, they model the system as a set of point particles, that move on a lattice, interacting according to suitable and simple rules in order to mimics a fully dynamics [Frisch et al. 1987b]. These are *bottom up* approaches in which the macroscopic behavior of the fluid can be recovered by interpolation techniques [Frisch et al. 1987b].

Lattice models have a number of advantages over more traditional numerical methods, particularly when fluids mixing and phase transitions occur [Rothman and Zaleski 1994]. The simulation is always performed on a regular grid and can be efficiently implemented on a massively parallel computer. Solid boundaries and multiple fluids can be introduced in a straightforward manner and the simulation is performed equally efficiently, regardless of the complexity of the boundary or interface [Buick et al. 1998]. In addition there are not numerical stability issues because the evolution follows integer arithmetic. However, system parametrization (viscosity, for example) is a difficult task in such lattice models and they are less realistic than PDE based models.

For the purpose of game application, real time is a fundamental requirement. So, the trad-off between realism and frame rate becomes the main challenge. Therefore, Navier-Stokes equations solvers, based on finite difference approaches, with special care about stability and speed were proposed [Stam 2003a] as well as Lattice Boltzmann methods (LBM) on graphics hardware [Zhao et al. 2007] and real time techniques for simulating atmospheric effects [Wenzel 2006].

SPH has been also applied for fluid simulation and interaction of the fluid with rigid bodies through penalty techniques, contact (pressure) forces, and a discrete angular velocity representation ([Gam 2004], Vol. 6, page 189). Besides, Linearized Bernoulli's equations and convolution techniques for discretization have been applied for real time computer-generated ocean surface ([Gam 2004], Vol. 4, page 256).

Interaction between fluids models and digital terrains is a subclass of fluid-surface interaction [Ye et al. 1996]. It can be addressed by hybrid methods in which the fluid is a continuum medium, simulated by Navier-Stokes plus SPH or grid based techniques, and the surface is represented as a discrete one [GENEVAUX et al. 2003; Solenthaler et al. 2007; Batty et al. 2007]. These approaches deal with the specific problem of preventing the leaking of fluid across the polygonal surface [Guendelman et al. 2005; Bridson et al. 2002].

In addition, fluid flows can be simulated on $2D$ manifolds represented by (continuous) subdivision surfaces that have a natural quad patch parametrization [Stam 2003b]. Besides, a hybrid particle and implicit surface approach to simulating water was proposed in [Foster and Fedkiw 2001], which led to the particle level set method [Enright et al. 2002].

Finally, rendering techniques must be applied to ensure the desired level of realism but keeping the frame rate suitable for game applications. Realistic rendering can be properly addressed through several algorithms incorporating usual refraction, reflection and Fresnel terms ([Gam 2004], Vol. 1, page 583), refraction approaches based on perturbing the texture coordinates used in a texture lookup of an image of the nonrefractive objects in the scene ([GPU 2005], Chap. 19) and accelerated volume rendering on GPU [Krger and Westermann 2005].

3 FHP

The FHP was introduced by Frisch, Hasslacher and Pomeau [Frisch et al. 1986] in 1986 and is a model of a two-dimensional fluid. It can be seen as an abstraction, at a microscopic scale, of a fluid. The FHP model describes the motion of particles traveling in a discrete space and colliding with each other. The space is discretized in a hexagonal lattice.

The FHP particles move in discrete time steps, with a velocity of constant modulus, pointing along one of the six directions of the lattice. The dynamics is such that no more than one particle enters the same node at the same time with the same velocity. This restriction is the *exclusion principle*; it ensures that six Boolean variables at each lattice node are always enough to represent the microdynamics.

The velocity modulus is such that, in a time step, each particle travels one lattice spacing and reaches a nearest-neighbor node. When exactly two particles enter the same node with opposite velocities, both of them are deflected by 60 degrees so that the output of the collision is still a zero momentum configuration with two particles. The deflection can occur to the right or to the left, indifferently, as shown in Figure 1. For symmetry reasons, the two possibilities are chosen randomly, with equal probability.

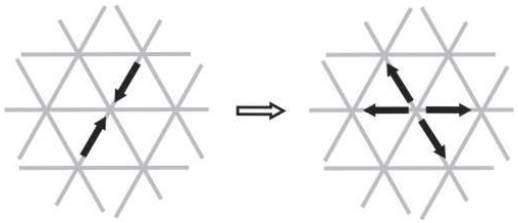


Figure 1: *The two-body collision in the FHP Source.*

When exactly three particles collide with an angle of 120 degrees between each other, they bounce back to where they come from (so that the momentum after the collision is zero, as it was before the collision). Both two- and three-body collisions are necessary to avoid extra conservation laws. For all other configurations no collision occurs and the particles go through as if they were transparent to each other.

The full microdynamics of the FHP model can be expressed by evolution equations for the occupation numbers defined as the number, $n_i(\mathbf{r}, t)$, of particle entering node $\mathbf{r}$ at time t with a velocity pointing along direction $\vec{c}_i$

$$\vec{c}_i = \left(\cos\frac{2\pi i}{6}, \sin\frac{2\pi i}{6}\right), \tag{1}$$

where $i = 1, 2, \ldots, 6$ labels the six lattice directions, as shown in Figure 2.

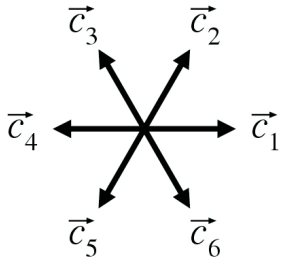


Figure 2: *The six lattice directions.*

The numbers n_i can be 0 or 1. We also define the time step as Δ_t and the lattice spacing as Δ_r. Thus, the six possible velocities $\vec{v}_i$ of the particles are related to their directions of motion by

$$\vec{v}_i = \frac{\Delta_r}{\Delta_t}\vec{c}_i. \tag{2}$$

The microdynamics of a LGCA is written as

$$n_i\left(\mathbf{r}+\Delta_r\vec{c}_i, t+\Delta_t\right) = n_i\left(\mathbf{r},t\right)+\Omega_i\left(n\left(\mathbf{r},t\right)\right) \qquad (3)$$

where Ω_i is called the collision term [Chopard and Droz 1998].

Now we propose an extension to 3D, using the 2D FHP model explained above. In practice, the system of a given cellular automata rule cannot deal with an infinite lattice, it must be finite and have boundaries [Chopard and Droz 1998]. So, first we must define a domain in the three dimensional space, where the microscopic particles can evolve. Then, our proposal consists of regularly distribute planes along the x and z axis, like in Figure 3. Each of these planes is a system whose cellular automata rule is the 2D FHP.

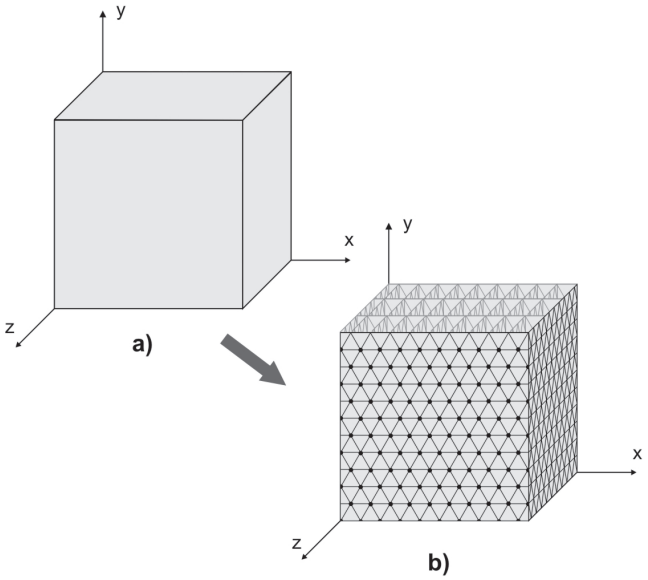


Figure 3: *(a) Domain in 3D Space and (b) the distribution of 2D FHP planes.*

Once simulated the two dimensional FHP in each plane independently, we perform a simple interpolation to generate a $2D$ macroscopic flow in each plane. In this step, we add new nodes to the FHP grid in order to complete a rectangular grid in each plane, as pictured in Figure 4.

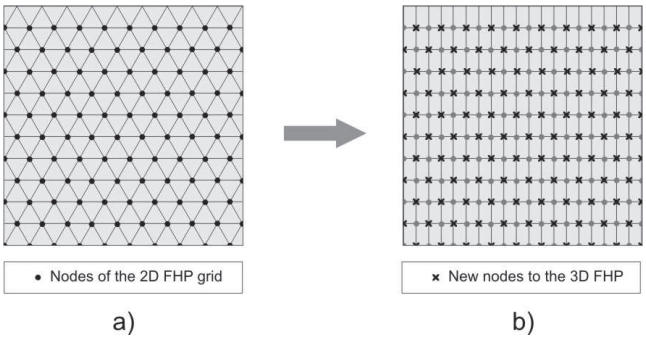


Figure 4: *Extend the (a) FHP grid to generate a (b) rectangular mesh.*

Following the usual definition of statistical mechanics, we compute the macroscopic density in each node (x_i, y_i, z_i) of the plane $x = x_i$ through the expression:

$$\rho_x\left(x_i,y_i,z_i,t\right) = \sum_{j\in V}\sum_{k=1}^{6} n_k\left(x_j,y_j,z_j,t\right), \qquad (4)$$

where V is a neighborhood of point (x_j, y_j, z_j).

An analogous expression can be used for the plane $z = z_i$. Now, we must render a $3D$ macroscopic flow. We shall observe that each node (x_i, y_i, z_i) in Figure 3 belongs to the planes $x = x_i$ and $z = z_i$. So, the $3D$ density can be finally obtained through a simple mean of the corresponding values in the FHP planes, that means:

$$\rho\left(x_i,y_i,z_i,t\right) = \frac{\rho_x\left(x_i,y_i,z_i,t\right)+\rho_z\left(x_i,y_i,z_i,t\right)}{2}. \qquad (5)$$

We also calculate the velocity field through macroscopic samples of the FHP behavior. In order to do that, we use a set of test particles whose initial positions in $3D$ domain are random. For each macroscopic particle p_{mac}, we take a neighborhood $V\left(p_{mac}, radius\right)$, where $radius$ is an radius of influence, and we calculate the sum of the velocities $\vec{c}_i$ (equation (1)) of the microscopic particles inside the neighborhood:

$$\vec{v}\left(p_{mac}\right) = \sum_{\vec{r}\in V\left(p_{mac},radius\right)} \vec{c}_i\left(\vec{r},t\right). \qquad (6)$$

Once we have calculated the velocity field, given a test particle p, in $\vec{x}\left(p,t\right)$ position at instant t, the position of the particle p at next time is:

$$\vec{x}\left(p,t+\Delta t\right) = \vec{x}\left(p,t\right)+\sigma\vec{v}(p), \qquad (7)$$

where σ is a constant for the macroscopic step in time.

4 Surface Flow Simulation Model

The model is composed by two basic elements: (1) A DEM that approximate the terrain; (2) A LGCA for water flow simulation [Barcellos et al. 2008]. The DEM is composed by a regular lattice and an elevation function $\varphi(i,j)$ that gives the elevation of the terrain in the lattice node (i,j). The LGCA is a particle based model that uses the $2D$ regular lattice of the DEM. The Figure 5 highlights a (i,j) node of the lattice and its 4 neighbors given by $(i-1,j)$, $(i,j-1)$, $(i+1,j)$ and $(i,j+1)$. These nodes are numbered V_1, V_2, V_3 and V_4, respectively. We also can compute the terrain slope at point $s\ (i,j)$. It is defined as the projection of the surface normal over the domain D.

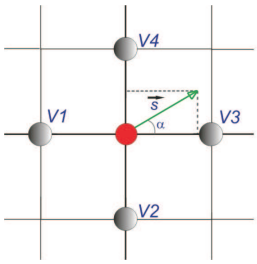


Figure 5: *Neighborhood of a lattice node.*

For each grid node (i,j) a particle counter is associated, which mimics the water held at terrain point $(i,j,\varphi(i,j))$. The particle system is used in order to update the field of counters of the lattice. The elevation and the slope define the stationary fields of the model. The water particles evolution updates the counter field which will be a non-stationary one.

The particles movement are discrete; that means, each particle moves from one lattice node to another one in the neighborhood, in the time step $\triangle t$. Therefore, given a particle at the position (i,j) and its slope $\overrightarrow{s}(i,j)$, we can define its direction of movement through the Algorithm 1.

Specifically, the particle moves from a node (i,j) to a nearest neighbor P, if the corresponding lattice direction $P-(i,j)$ is closer the slope direction at (i,j) and the high of the water free surface at (i,j) is greater than in P. This is verified in the 4th line in the Algorithm 1. A four bit string $n_1(i,j), n_2(i,j), ..., n_4(i,j)$ is related to each node (i,j) of the lattice. We set $n_k(i,j) = 1$ if the node (i,j) has one particle to send to the neighbor k, as we will see in the Algorithm 2, in the 14th line, explained below.

Algorithm 1 : findVelocityDirection

$P \Leftarrow findVelocityDirection(i, j, \overrightarrow{s})$;
$V \Leftarrow \{(k,l) \mid (i,j) - (k,l) \mid \leq 1\}$;
while V $\neq$ NULL **do**
  $P \Leftarrow arg\min\{arc(\overrightarrow{s}, (k,l) - (i,j)), (k,l) \in V\}$;
  Delete P from V;
  $P \Leftarrow NULL$;
end while
return P;

The Algorithm 2 summarizes the whole method. The inputs are the elevation function φ and the precipitation $P(t)$. In the initialization step, we must initialize the field $n_k(i,j)$, the particle counter $counter(i,j)$ and the slope field $s(i,j)$. In the first loop of the simulation, given a particle at the position (i,j) and its slope $\overrightarrow{s}(i,j)$, the direction of movement is obtained through Algorithm 1, and, consequently, the four bits string n are updated. The second loop of the simulation step updates the particle counters of each node (i,j) and its neighbors.

Algorithm 2 : calculateWaterSurface()

Input:
  Elevation function φ.
  Precipitation $P(t)$.
Initialization:
  Define the lattice L;
  $n_k(i,j) \Leftarrow 0, counter(i,j) \Leftarrow 0, (i,j) \in L$;
  Compute the slope $\overrightarrow{s}(i,j), (i,j) \in L$;
Simulation:
for each $(i,j) \in L$ **do**
  $P \Leftarrow findVelocityDirection(i, j, \overrightarrow{s})$;
  $n_l(i,j) = 0, l = 1,2,3,4$;
  for $l \in \{1...4\}$ **do**
    if $P = V_l$ **then**
      set $n_l(i,j) = 1$;
    end if
  end for
end for
for each $(i,j) \in L$ **do**
  $counter(i,j) \Leftarrow counter(i,j) - n_1(i,j) - n_2(i,j) - n_3(i,j) - n_4(i,j) + P(t)$;
  $counter(i-1,j) \Leftarrow counter(i-1,j) + n_1(i,j) + P(t)$;
  $counter(i,j+1) \Leftarrow counter(i,j+1) + n_2(i,j) + P(t)$;
  $counter(i+1,j) \Leftarrow counter(i+1,j) + n_3(i,j) + P(t)$;
  $counter(i,j-1) \Leftarrow counter(i,j-1) + n_4(i,j) + P(t)$;
end for

The result is a function

$$f(i,j,t) = \varphi(i,j) + \beta \cdot counter(i,j,t), \tag{8}$$

which gives the high of the free surface flow f, at the point (i,j) in the simulation time t (β is a scale parameter). We shall emphasize that the algorithm does not suffer from numerical stability issues because the floating point fields are stationary and the update of the counter field is based on simple comparisons and integer arithmetic. Besides, volume conservation, an important issue to accurate simulates the flood distribution, is straightforward verified because all operations are conservative with respect to the number of particles. This algorithm can be generalized for DTMs that approximate the terrain surface for triangulations, as demonstrated in [Barcellos et al. 2008; Barcellos et al. 2007].

5 GPU-Based Simulation Model

The algorithm described in section 4 for surface flow simulation in DTMs is based on a cellular automaton approach, with local and simple rules. Therefore it may be very suitable for a GPU implementation. The main idea is to encode the information needed in the simulation as textures into the video memory (see [Barcellos et al. 2008] for more details).

The Figure 6 gives an overview of the whole GPU processing. At each interaction of the main loop a $2D$ texture, called **Property Texture**, encoding the automaton configuration is generated as follows. To each texture point (i,j) it is associated the index k, such that each color channel represents a different data: the four bit string $n_k(i,j) = 1$; $k = 1,2,3,4$, the terrain elevation $\varphi(i,j)$ and a particle counter, respectively channels R, B and G. Based on the **Property Texture** a $RGBA$ texture is generated, named **Incident Directions Texture**, with each channel been a flag that is set to 1 if the corresponding neighbor of a node has one particle to send to its position and set to 0 otherwise.

By using the **Incident Directions Texture** and the particles counter at time interaction t, two new fields are generated at time $t+1$. One with the updated particle counters and a second field with the elevation of the free surface flow $f(i,j)$ at each (i,j) node of the lattice. These fields will be composed to form the new **Property Texture** and the **Free Surface Texture**, at time $t+1$, represented on Figure 6. Then, the visualization pipeline starts, producing a new texture to encode the normal field of the free surface, called **Normal Vector Texture** in Figure 6.

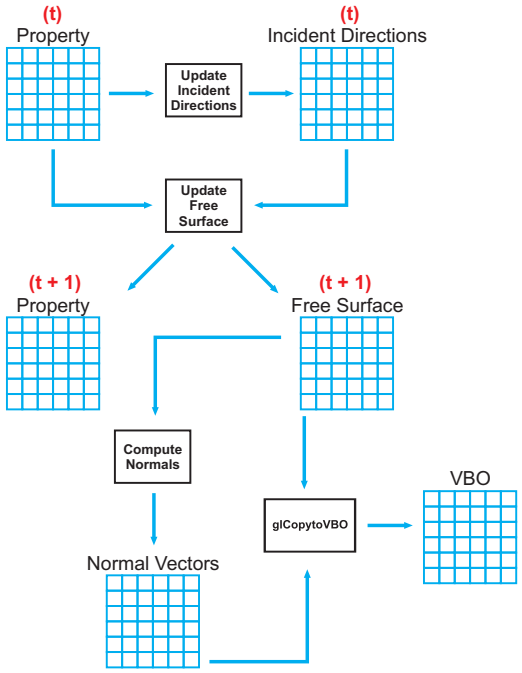


Figure 6: GPU *processing for the surface flow animation.*

The **Normal Vector Texture** and the **Free Surface Texture** are used as input to the *Vertex Buffer Object* (**VBO**), as Figure 6 shows. The **VBO** will be managed as a *Vertex Array* to render the final visualization of the free surface of the fluid. Therefore, the whole computation (simulation and visualization) is performed in the GPU. This scheme allow us to minimize the information flow between the CPU and the GPU and improve the performance of the application. The GPU implementation is summarized by the algorithm 3. The corresponding shader are described in [Barcellos et al. 2008].

Algorithm 3 : GPU Simulation()

Input:
  Property Texture (Velocity, Terrain Elevation, Particles Counter);
Simulation:
  Update Incident Directions;
  Update Free Surface;
  Compute Normals;
  Send Results to Vertex Array;
Visualization:
  Draw Vertex Array;

6 Interaction between Models

Our proposed framework for real time fluid animation is based on two elements: (a) A $3D$ fluid simulation model based on the FHP and interpolation techniques, that is implemented in CPU; (b) The surface flow simulation in DTMs, that is implemented in GPU. To integrate this two elements we have to implement a communication between then through a suitable boundary condition.

We use the $3D$ FHP to simulate a rainfall over a random region of the terrain, chosen by the user. After setting the initial configuration and parameters, we can start the evolution of the system. The Figure 7 shows the interaction between the models in each time step.

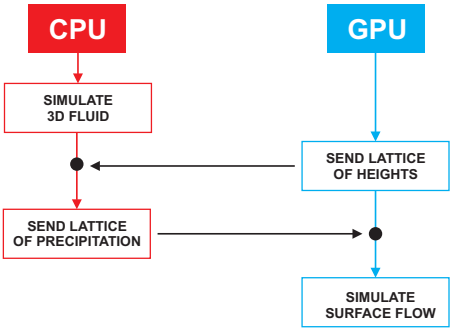


Figure 7: *Interaction between models in each time step.*

Basically, the rainfall has to evolve to reach the terrain. This is what happens in the **Simulate** $3D$ **Fluid** box in CPU (Figure 7). In each time step the microscopic particles of the $3D$ FHP evolve following the rules explained in section 3. After that, the system has to know if some of these particles have overtaken the limits of the three dimensional fluid (FHP). To perform this task, the GPU implementation has to send to the CPU implementation a lattice with the information about the field computed by the expression (8) at each node (i, j). With this data, the system can check all the new position of the $3D$ FHP particles: if the particle has reached the limit it dies in the FHP grid and feed the lattice of the LGCA for water flow simulation. So, the CPU implementation has to send to the GPU implementation the updated particles counter field. The corresponding precipitation is used in the algorithm 2, updating the $P(t)$ variable. Then, the GPU implementation is able to simulate the fluid behavior over the terrain, what is represented in Figure 7 by the **Simulate Surface Flow** box in GPU. The GPU implementation has to update the free surface elevation, given by expression (8).

7 Experimental Results

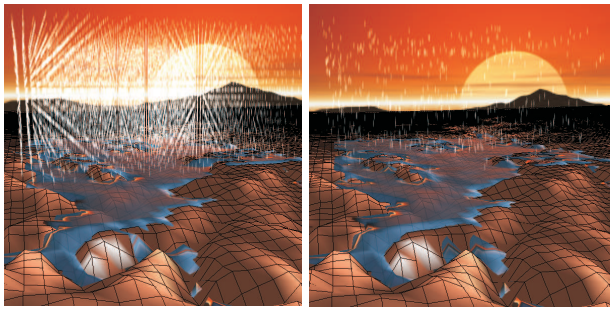
In this section we describe details of the implementation and some experiments with the proposed framework. In these experiments we show the behavior of the fluid in different digital terrain models and highlight aspects that can be useful for computer graphics and real time applications: simple to initialize the animation, simulation over complex topography and computational efficiency.

The framework was developed in $C/C++$ language, with *GLUI* [Rademacher 1999; Stewart 2006] library for the graphic interface. The visualization was implemented in OpenGL [Wright et al. 2007]. To increase the scene realism we applied the environment mapping technic , the point sprite method to render the rainfall and Fresnel effects (based on the physical laws of reflection and refraction) to render the water free surface. The shaders was implemented in OpenGL Shading Language.

The experiments were performed in a Intel Core 2 Duo 2.66 GHz, with 4 GB of RAM and a Video Card NVidia GeForce 8800 GTX, running Windows XP. The pictures included in this paper are snapshots obtained from the framework. The corresponding videos can be found in http://virtual01.lncc.br/∼barcellos/videosSBGames2008.zip

7.1 Particles Visualization

The microscopic particles of the FHP are initialized in random positions at the top of the bounding box. As the system evolve, new microscopic particles are born to feed the rainfall flow. As explained in section 3, with simple interpolation techniques we can fill the $3D$ macroscopic fields (density and velocity) over a regular grid. In this way we can estimate the particle density (equation (5)) at each node of the grid, and visualize then using the Point Sprite method. Meanwhile, we can observe the patterns of a regular grid, as shown in Figure 8.(a).



(a) (b)

Figure 8: *Particles visualization: (a) Microscopic particles; (b) Macroscopic particles.*

To avoid this patterns we use a set of macroscopic particles at random positions, inside the bounding box of the rainfall. This macroscopic particles move according to the macroscopic velocity field, following expressions (6) and (7). In this way we have a more realistic visualization of the rainfall, as shown in Figure 8.(b). That is the way in the following examples, we render macroscopic particles to visualize the rainfall.

7.2 Fluid Behaviors

Before initialize the framework the user has to decide which digital terrain model he wants to use in the simulation. Then, the graphical user interface implemented allows to interactively place the rainfall domain over the terrain. Figure 9 shows some examples of digital terrain models and a bounding box that represents the location of the $3D$ fluid model. With the environment mapping technique we are able to generate different environments, like a sunny or cloudy day.

The Figure 10 shows some snapshots of the fluid motion with the downhill path over the terrain surface and affected regions. As expected, the water flows directly downhill and it takes the easiest downhill path that is available.

The lake formation can be better observed in Figure 11. We can notice, along time evolution, the enlargement of the flooded area, generating a watershed. A watershed is an area of land that drains downhill to a body of water, such as a river. It includes both the waterway and the land that drains to it.

The Figure 12 shows a concave region in the terrain digital model and the the water accumulation along time evolution.

7.3 Computational Efficiency

The examples of section 7.2 where performed with $20 \times 30 \times 30$ (height $\times$ width $\times$ depth) nodes in the $3D$ fluid simulation. We use a 256×256 lattice resolution for the surface flow simulation. Both initialization as particle generation methods for the particles of the FHP and for the set of macroscopic particles are random. The microscopic particles iterate following the rules explained in section 3. In our implementation the FHP particle are restricted to the bounding boxes pictured in Figure 9. If a microscopic particle goes outside this volume it is discarded by the simulation. The set of macroscopic particles iterate following the FHP velocity field. We use $radius = 5$ in expression (6) and $\sigma = 0.0015$ in expression 7. All the examples were simulated until 3000 frames.

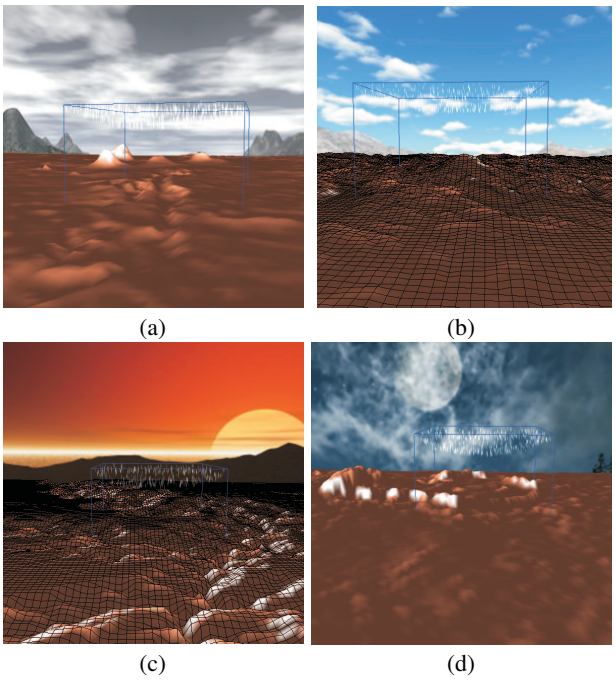


Figure 9: *Examples of FHP bounding box over some DTMs with different environments: (a) Cloudy day at Mars terrain; (b) Sunny day at Puget Sound terrain; (c) Canyon terrain with sunrise; (d) Starry night at Venus terrain.*

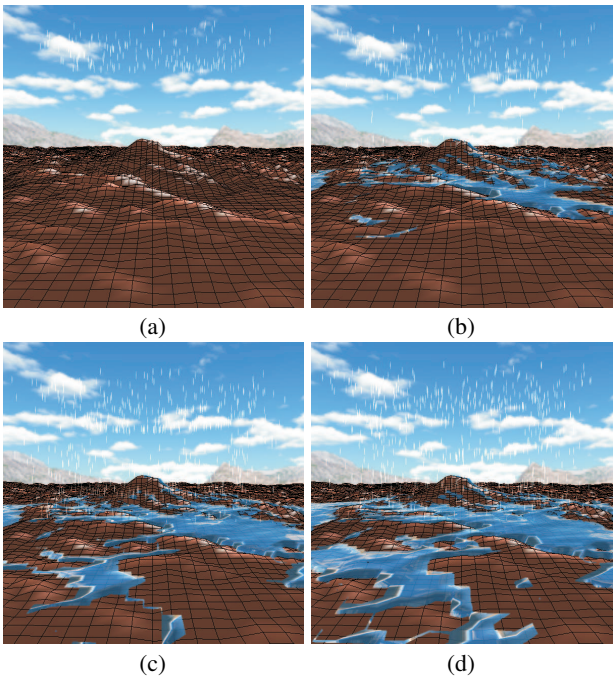


Figure 10: *Fluid motion: (a) At initial time steps; (b) After 400 steps; (c) Configuration after* 1200 *time steps; (d) After 3000 steps.*

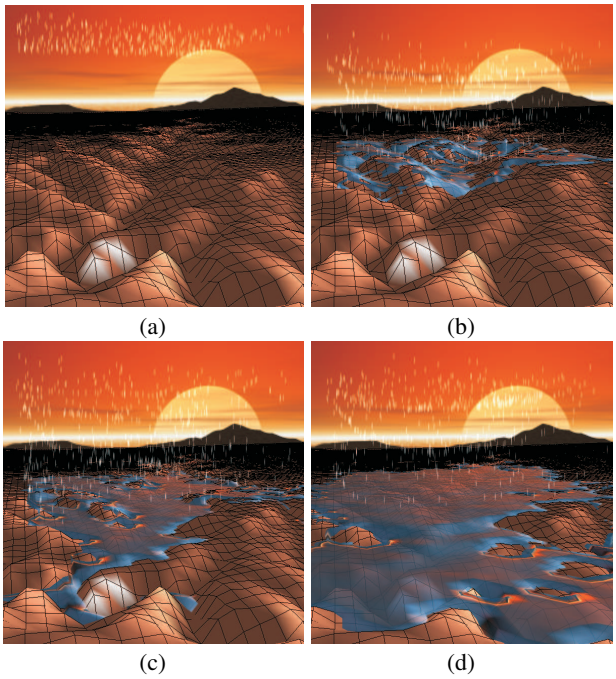


Figure 11: *The lake formation: (a) At initial configuration; (b) After 60 steps; (c) After 900 steps; (d) After 3000 steps.*

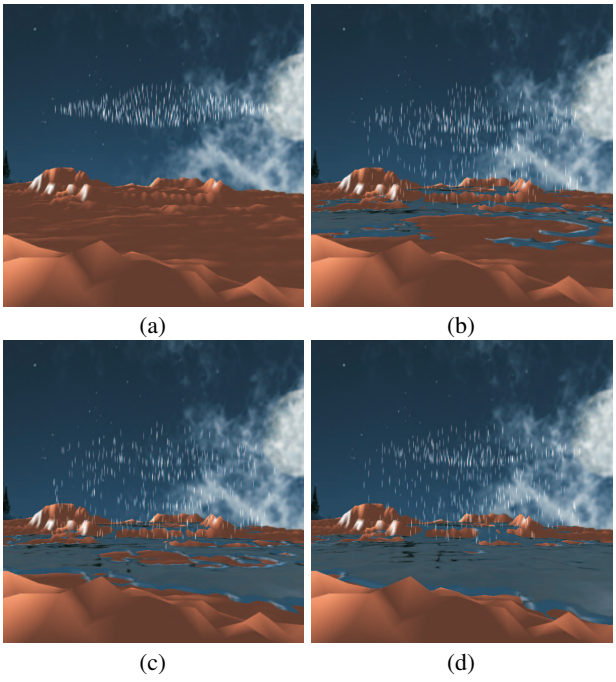


Figure 12: *The water accumulation: (a) At initial configuration; (b) After 940 steps; (c) After 1500 steps; (d) After 3000 steps.*

The time measurements presented in Table 1 were performed in order to compare the increase of the FPS when using different $3D$ FHP grid dimensions, as to observe what happens with the number of the FHP particles per frame and the accumulated particles in the water flow over the terrain. We maintain the configuration explained above and the Puget Sound as the terrain model.

Table 1: *Table listing the $3D$ FHP grid dimensions (height $\times$ width $\times$ depth), the average number of FHP particles per frame, the accumulated number of particles in the water flow over the terrain, the average number of Macro particles per frame and finally the frames per second rates(FPS).*

FHP Grid	FHP	Water	Macro	FPS
1) $10 \times 10 \times 10$	772	479	61	60
2) $10 \times 20 \times 20$	4333	26033	178	44.78
3) $10 \times 30 \times 30$	10653	64549	383	31.91
4) $10 \times 40 \times 40$	19343	121747	651	22.72
5) $20 \times 10 \times 10$	6541	14958	35	60
6) $20 \times 20 \times 20$	5686	1029	274	37.50
7) $20 \times 30 \times 30$	16464	41086	553	25
8) $20 \times 40 \times 40$	32219	104146	937	16.04
9) $30 \times 10 \times 10$	11620	16343	44	56.60
10) $30 \times 20 \times 20$	36535	54453	165	35.29
11) $30 \times 30 \times 30$	18397	1381	765	19.35
12) $30 \times 40 \times 40$	40038	56094	1230	12.24
13) $40 \times 10 \times 10$	14380	14853	53	53.57
14) $40 \times 20 \times 20$	52295	54265	191	30.61
15) $40 \times 30 \times 30$	104865	118215	454	17.65
16) $40 \times 40 \times 40$	42559	2039	1572	10.27

The $7th$ line shows the $3D$ FHP grid configuration used to generate the examples of section 7.2. We can observe that it returns a 25 FPS, which is closer a real time one. Other configurations, like the $30 \times 20 \times 20$ (line $10th$), gives a FPS that goes beyond the real time frame rate.

The frame rate is very dependent from the number of FHP nodes and particles. However, once the FHP particles that goes outside the bounding box is simply discarded, it is possible that two similar configurations show different frame rates. That is why the experiments reported in the $2nd$ and $13th$ lines have the same number of nodes (4000) but the FPS is larger in the latter one.

We can observe that the experiment reported in the $13th$ line has a highest bounding box (height = 40) than the one in the $2nd$ line (height = 10), which ensures a large average number of FHP particles per frame in the first case (14380 against 4333). However, due to be higher, the experiment reported in the $13th$ line has a larger number of discarded particles (the particles that goes outside the bounding box), which means that it contributes less to the precipitation over the terrain (14853 against 26033).

Once the surface flow simulation is performed in GPU, an important concern is the accuracy of the numerical simulation. The graphics hardware used supports 4 bytes per color channel which fits the requirement on the accuracy of the computation.

Table 2: *Table listing the lattice dimensions, the average number of FHP particles per frame, the accumulated number of Water Particles in the surface flow and the frame rate.*

Lattice	FHP	Water	FPS
128×128	15884	53911	32.9
256×256	19163	71345	24.8
512×512	19219	77390	13.6

In the Table 2 we report the FPS evolution against the lattice resolution. If we keep parameters for the rainfall simulation and visualization unchanged, we get lower frame rates when increasing the lattice resolution because the computational cost of the surface flow simulation gets larger. Figure 13 pictures the effect of the lattice resolution in the final scene visualization. According to Table 2 we have a real time frame rate in this case. We observe also that the visual quality of the generated scene was preserved quite well.

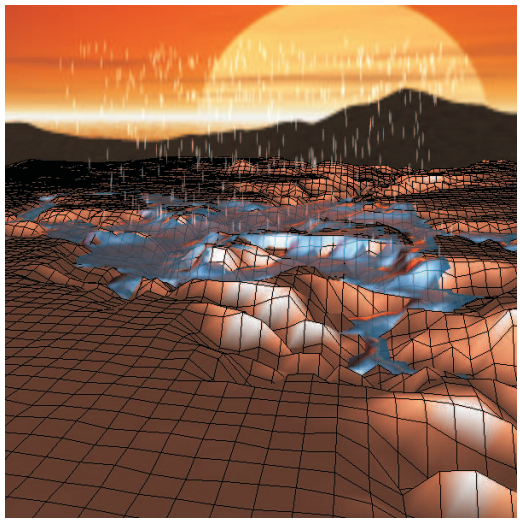


Figure 13: 128×128 *digital terrain model.*

8 Conclusion and Future Works

In this work, we propose a framework for fluid animation based on Cellular Automata models for computer games. The new animating framework is composed by a $3D$ fluid animation technique and a GPU surface flow simulation over terrain models. In the experimental results we emphasize the simplicity and power of the proposed models when combined with efficient techniques for rendering.

Future directions in this work are to perform a GPU implementation of the fluid model in order to improve performance and to compare the obtained results with the Lattice Boltzmann model to quantify the level of physical realism obtained.

Acknowledgements

To João Vicente P. Reis Filho, for all the discussions.

References

BARCELLOS, B., GIRALDI, G. A., SILVA, R. L., APOLINARIO, A. L., AND RODRIGUES, P. S. S. 2007. Surface flow animation in digital terrain models. In *SVR 2007 - IX Symposium on Virtual an Augmented Reality*, 123–132.

BARCELLOS, B., GIRALDI, G. A., APOLINARIO, A. L., AND RODRIGUES, P. S. S. 2008. Gpu surface flow simulation and multiresolution animation in digital terrain models. Tech. rep., National Laboratory of Scientific Computing.

BATTY, C., BERTAILS, F., AND BRIDSON, R. 2007. A fast variational framework for accurate solid-fluid coupling. *ACM Trans. Graph.*.

BENZI, R., SUCCI, S., AND VERGASSOLA, M. 1992. The lattice boltzmann equation: Theory and applications. *Physics Reports*.

BRIDSON, R., FEDKIW, R., AND ANDERSON, J. 2002. Robust treatment of collisions, contact and friction for cloth animation. In *SIGGRAPH '02*.

BUICK, J. M., EASSON, W. J., AND GREATED, C. A. 1998. Numerical simulation of internal gravity waves using a lattice gas model. *Int. J. Numer. Meth. fluids*.

CHOPARD, B., AND DROZ, M. 1998. *Cellular Automata Modeling of Physical Systems*. Cambridge University Press.

DEUSEN, O., EBERT, D. S., FEDKIW, R., MUSGRAVE, F. K., PRUSINKIEWICZ, P., ROBLE, D., STAM, J., AND TESSENDORF, J. 2004. The elements of nature: Interactive

and realistic techniques. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*.

DOOLEN, G. 1990. *Lattice Gas Method for Partial Differential Equations*. Addison-Wesley.

ENRIGHT, D., MARSCHNER, S., AND FEDKIW, R. 2002. Animation and rendering of complex water surfaces. *ACM Trans. Graph.*.

FOSTER, N., AND FEDKIW, R. 2001. Practical animation of liquids. In *SIGGRAPH '01*.

FOSTER, N., AND METAXAS, D. 1997. Modeling the motion of a hot, turbulent gas. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*.

FRISCH, U., HASSLACHER, B., AND POMEAU, Y. 1986. Lattice-gas automata for the navier-stokes equation. *Phys. Rev.*.

FRISCH, U., D'HUMIÈRES, D., HASSLACHER, B., LALLEMAND, P., POMEAU, Y., AND RIVET, J.-P. 1987. Lattice gas hidrodynamics in two and three dimension. *Complex Systems*, 649–707.

FRISCH, U., D'HUMIÈRES, D., HASSLACHER, B., LALLEMAND, P., POMEAU, Y., AND RIVET, J.-P. 1987. Lattice gas hydrodynamics in two and three dimension. *Complex Systems*.

2004. In *Game Programming Gems*, K. Pallister, Ed. Charles River Media, Cambridge, MA.

GENEVAUX, O., HABIBI, A., AND DISCHLER, J.-M. 2003. Simulating fluid-solid interaction. In *Graphics Interface*.

GIRALDI, G., XAVIER, A., JR, A. A., NETO, A., AND RODRIGUES, P. 2005. Animation of gas-liquid systems through lattice gas cellular automata and smoothed particle hydrodynamics. In *Proceedings of the CNMAC 2005*.

2005. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley.

GUENDELMAN, E., SELLE, A., LOSASSO, F., AND FEDKIW, R. 2005. Coupling water and smoke to thin deformable and rigid shells. In *SIGGRAPH '05*.

HIRSCH, C. 1988. *Numerical Computation of Internal and External Flows: Fundamentals of Numerical Discretization*. John Wiley & Sons.

IGLESIAS, A. 2004. Computer graphics for water modeling and rendering: a survey. *Future Gener. Comput. Syst.*.

INAMURO, T., OGATA, T., AND OGINO, F. 2004. Numerical simulation of bubble flows by the lattice boltzmann method. *FUTURE GENERATION COMPUTER SYSTEMS 20*, 6, 959–964.

KRGER, J., AND WESTERMANN, R. 2005. Gpu simulation and rendering of volumetric effects for computer games and virtual environments. In *In Proceedings*, Eurographics, 685–693.

LIU, G. R., AND LIU, M. B. 2003. *Smoothed Particle Hydrodynamics : a Meshfree Particle Method*. World Scientific.

MÜLLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of ACM SIGGRAPH symposium on Computer animation*.

MÜLLER, M., KEISER, R., NEALEN, A., PAULY, M., GROSS, M., AND ALEXA, M. 2004. Point based animation of elastic, plastic and melting objects. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*.

MÜLLER, M., SCHIRM, S., AND TESCHNER, M. 2004. Interactive blood simulation for virtual surgery based on smoothed particle hydrodynamics. *Technol. Health Care 12*, 1, 25–31.

PREMOZE, S., TASDIZEN, T., BIGLER, J., LEFOHN, A., AND WHITAKER, R. 2003. Particle-based simulation of fluids. *Computer Graphics Forum 22*, 3.

RADEMACHER, P. 1999. *GLUI: A GLUT-Based User Interface Library*. Último acesso em 25/05/2006, June.

ROTHMAN, D. H., AND ZALESKI, S. 1994. Lattice-gas models of phase separation: Interface, phase transition and multiphase flows. *Rev. Mod. Phys*.

SOLENTHALER, B., SCHLAFLI, J., AND PAJAROLA, R. 2007. A unified particle model for fluid-solid interactions. *Comput. Animat. Virtual Worlds*.

STAM, J. 1999. Stable fluids. In *Siggraph 1999*.

STAM, J. 2003. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*.

STAM, J. 2003. Flows on surfaces of arbitrary topology. In *SIGGRAPH '03*.

STEWART, N. 2006. *GLUI User Interface Library*. Último acesso em 25/05/2006.

WENZEL, C. 2006. Real-time atmospheric effects in games. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, 113–128.

WOLFRAM, S. 1996. *Cellular automata and complexity*. Addison-Wesley, http://www.stephenwolfram.com/publications/articles/ca/86-fluids/index.html.

WRIGHT, R. S., LIPCHAK, B., AND HAEMEL, N. 2007. *OpenGL SuperBible (4rd Edition)*. Addison-Wesley Professional.

XAVIER, A. V., GIRALDI, G. A., RODRIGUES, P. S., JR, A. L. A., AND NETO, A. A. S. 2005. Lattice gas cellular automata for computational fluid animation. In *SIBGRAPI 2005 - Proceedings of the 4th Workshop of Theses and Dissertations*, 1–6.

YE, Z., MAIDMENT, D., AND MCKINNEY, D. 1996. Map-based surface and subsurface flow simulation models: An object-oriented and gis approach. Tech. rep., Center for Research in Water Resources, University of Texas at Austin.

ZHAO, Y., QIU, F., FAN, Z., AND KAUFMAN, A. E. 2007. Flow simulation with locally-refined lbm. In *SI3D*, ACM, 181–188.

Plataforma Saberlândia: Integrando Robótica e Multimídia no Desenvolvimento de Jogos Educacionais

Ivete Martins Pinto[1], Silvia Costa Botelho[1], Rodrigo Chaves de Souza[1], Thiago Sonego Goulart[2], Rafael Colares[2], Raphael Leite Campos[2]

[1]Programa de Pós-Graduação em Educação em Ciências
[2]Curso de Engenharia de Computação
Universidade Federal do Rio Grande – FURG, Rio Grande – RS – Brasil

Resumo

Este artigo apresenta o Projeto SABERLÂNDIA, uma plataforma para o desenvolvimento de jogos educacionais que, a partir de contextos e conteúdos fornecidos,, propicia a geração automática de jogos de ação. O projeto SABERLÂNDIA tem como focos principais: i. o desenvolvimento de sistemas de autoria que estimulem a construção do conhecimento, de forma lúdica, propiciando aos diferentes atores (professores, aprendizes) atuarem como autor no desenvolvimento destes jogos; ii. a utilização de recursos multimídias como motivação, fazendo uso de recursos de Realidade Virtual e Robótica. São utilizadas técnicas de planificação associadas à IA (Inteligência Artificial) para gerar a seqüenciabilidade do jogo, utilizando-se o formalismo STRIPS para definição genérica das possíveis ações permitidas, definidas pelo autor. O roteiro e conteúdos, fornecidos pelo autor, são então inseridos de forma automática no plano do jogo. Um conjunto de cenários pré-disponíveis, armazenados em uma biblioteca de cenários e personagens, são reproduzidos de forma virtual (com visualização 3D) e em maquetes. Os personagens também são disponibilizados de forma virtual e através de sistemas robóticos, em kit fornecido. Habilidades necessárias à jogabilidade são adquiridas através de atividades de motricidade associadas ao sistema robótico. A proposta encontra-se em fase final de implementação, sendo apresentados testes e análises do protótipo hoje existente.

Palavras-chave: jogos eletrônicos, jogos educacionais, robótica, Planejamento (*planning*), sistemas de autoria

1. Introdução

Conceber a aprendizagem como um processo de interação em que todos participam, trocam e compartilham conhecimentos é um dos caminhos para substituir a concepção de aprendizado, na qual os conhecimentos são apresentados pelo professor, cabendo aos alunos a recepção passiva. A troca por uma concepção mais dinâmica, na qual os conhecimentos são socialmente (re) construídos, conduzirá à uma interação que, não é senão uma nova maneira de descrever o que se passa na história de um sujeito, entre ele e o mundo, é a própria dinâmica da aprendizagem (Merieu, 1998, p. 57).

As atividades lúdicas estimulam a curiosidade, a autoconfiança, o potencial criador e a autonomia, proporcionando o desenvolvimento da linguagem, do pensamento, da concentração e atenção, exercendo assim, função educativa.

Os jogos educativos são atividades lúdicas que possuem objetivos pedagógicos especializados para o desenvolvimento do raciocínio e aprendizado de crianças e adultos.

Tendo o pedagógico como meta, deve-se atentar ao fato de que a motivação não deva ser desconsiderada. Por exemplo, por divertir e motivar sabe-se que jogos de ação retêm a atenção e o interesse de crianças e adolescentes, exercitando as funções mentais e intelectuais dos jogadores. Se convenientemente planejados, estes poderiam ser um recurso pedagógico eficaz para a construção do conhecimento.

O uso de recursos tecnológicos, dentre eles o computador e a Internet, pode vir a potencializar os aspectos acima descritos, produzindo jogos eletrônicos educativos, sempre que atrelado a princípios teórico-metodológicos claros e bem fundamentados. Defende-se que tais princípios passam pela individualização dos aprendizes, respeitando sua cultura, interesses, dúvidas e incertezas. Busca-se assim aliar-se o lúdico às necessidades e contexto do aluno, permitindo a estes, e aos professores desenvolverem jogos educativos, cujos cenários, roteiros, personagens, e outros elementos, façam parte da sua realidade.

O presente trabalho visa apresentar um sistema baseado em técnicas de planificação associadas à Inteligência Artificial (IA) para a autoria de jogos

eletrônicos educacionais de ação, onde, a partir do roteiro definido pelo autor, o jogo de ação é gerado de forma automática. Além de o sistema oferecer aos educadores e educandos a oportunidade de interagir num ambiente lúdico, flexível e dinâmico, propõe-se também neste artigo combinar a motivação associada a jogos de ação/estratégia com diferentes recursos tecnológicos como a Realidade Virtual e a Robótica.

A proposta está sendo implementada no projeto SABERLÂNDIA, atualmente financiada pela FINEP, através do Edital FINEP/MCT/MEC para desenvolvimento de jogos educacionais para o ensino fundamental.

As principais características deste sistema são: a possibilidade dos atores envolvidos, alunos e professores, serem "autores" dos roteiros de ação a serem criados (jogo que faz jogo); a possibilidade de utilização colaborativa do ambiente de aprendizado; e a utilização de diferentes mídias integrando recursos de realidade virtual e robótica; com portabilidade para diferentes tipos e custos de máquinas.

Neste enfoque, a proposta propicia a criação de jogos de aventura em terceira pessoa, onde o roteiro do jogo possa ser fornecido pelos autores. Associados ao roteiro, professores e alunos poderão criar situações problemas, cenários e personagens de acordo com a realidade das crianças envolvidas, além da possibilidade de serem compartilhados de forma remota entre diferentes grupos. Crê-se que através da interação aluno/aluno, aluno/professor e professor/professor, a educação pode ser transformada em um processo dinâmico, motivador e significativo.

A plataforma será validada em algumas escolas do ensino fundamental da cidade de Rio Grande/RS.

A seguir contextualiza-se a proposta, seguida pela apresentação da arquitetura do sistema desenvolvido, seu estado atual de desenvolvimento, conclusões e trabalhos futuros.

2. Contextualização

As tecnologias da informação e comunicação, especialmente o computador, propiciam a transformação dos mais diversos setores da sociedade. A inserção da tecnologia computacional na educação tem sido alvo de muitas pesquisas (Lévy, 1995, 1999; Fagundes et al, 1999; Mantoan et al, 1999; D'Abreu e Chella, 2001; Palloff e Pratt, 2002; Ramal, 2002; Behar et al, 2005; etc...), evoluindo desde a introdução dos laboratórios de informática nas escolas, desenvolvimento de softwares educacionais, ambientes de ensino na Web, sistemas de autoria e tutores inteligentes, robótica pedagógica, realidade virtual e jogos educativos.

Apesar de algumas limitações, a escola busca acompanhar os avanços tecnológicos, em sintonia com os processos de transformação da sociedade em meio a tais demandas. A utilização dos computadores e da Internet permite a criação de ambientes de aprendizagem que possibilitam novas formas de pensar e aprender, favorecendo a aprendizagem ativa e o desenvolvimento de processos metacognitivos levando os alunos a aprender a aprender.

Os jogos educacionais, se convenientemente planejados, são um recurso pedagógico eficaz para a construção do conhecimento. Segundo Vygotsky (1984), o brinquedo estimula a curiosidade e a autoconfiança, proporcionando desenvolvimento da linguagem, do pensamento, da concentração e da atenção; através do brinquedo a criança aprende a agir numa esfera cognitivista, habilitando-se a escolher suas próprias ações.

Nesta perspectiva, salienta-se a importância do jogo para o desenvolvimento de crianças e adolescentes, como uma atividade que envolve aspectos lúdicos, intelectuais, afetivos e sociais. Além disso, os jogos educacionais podem ser utilizados para introduzir e aprofundar conteúdos e, se desenvolvidos considerando-se os pressupostos pedagógicos adequados, podem se apresentar como motivadores e facilitadores na construção do conhecimento.

2.1 Saberlândia como sistema de autoria

Ao enfocar o uso de sistemas computacionais no desenvolvimento de jogos, destaca-se a possibilidade de potencialização do desenvolvimento de situações significativas, levantamento de hipóteses e a reconstrução de conceitos, bem como da interação dos usuários em ambientes virtuais de aprendizagem.

Sob essa ótica, a construção dos conhecimentos é um processo em que o sujeito elabora os significados e não simplesmente os assimila, construindo o caminho específico de sua evolução (D´Ambrosio,1986, p. 14). Neste caminho, não existem estruturas rígidas e únicas de desenvolvimento pré-fixadas, o que existem são caminhos individuais e coletivos que se pretende alcançar. Caminhos estes associados às experiências dos atores, as suas vivências e expectativas.

Esta proposta busca desenvolver um sistema de autoria de jogos educacionais voltados ao ensino fundamental, tendo como principais premissas a motivação no processo de ensino e aprendizagem, o tratamento do erro como parte desse processo, e a interdisciplinaridade.

Estas diretrizes conduzirão à implementação da possibilidade de autoria, a partir da concepção do professor e mesmo dos próprios alunos, os quais definem o cenário, os personagens e o roteiro; tudo isso aliado à utilização de recursos tecnológicos diversos.

As ferramentas de autoria se caracterizam por permitirem aos seus usuários que estes criem suas próprias produções. Tais sistemas devem ser de fácil operação, não necessitando de profundos conhecimentos de programação (Valle Filho et al, 2000).

A utilização de ferramentas de autoria para o desenvolvimento de jogos apresenta-se como uma alternativa para diminuir o custo, o tempo e a dependência em relação aos conhecimentos específicos em computação que seriam necessários para sua criação.

2.1 Saberlândia e os recursos tecnológicos

No sistema proposto, os recursos tecnológicos serão disponibilizados como ferramentas didático-pedagógicas, conforme o pressuposto do próprio termo SABERLANDIA, o qual é entendido como um espaço onde se propicia a criação dos seus próprios saberes, de explicação, reformulação, de criação de 'teorias' através da ação, da operação e mesmo da construção de sistemas simbólicos diferenciados.

Entende-se tal proposta como um sistema cognitivo, considerando os sujeitos em atividade cognitiva na interação uns com os outros e com as tecnologias disponíveis. Este será construído na interação entre sujeitos-sujeitos e sujeitos-objetos, transformando-se na medida em que as interações vão ocorrendo, sendo atualizado a cada solução provisória, e sua natureza modificando-se a cada problematização. Da mesma forma os sujeitos são transformados na/pela interação.

Nesta proposta é apresentada a possibilidade da autoria, sendo que no jogo, podem ser acoplados sistemas robóticos, que visam reproduzir em um tabuleiro as ações do avatar além dos recursos multimídia. Dessa forma é disponibilizada uma diversidade de mídias (som, vídeo, sistemas robóticos,...) que privilegiam não só o acesso à informação, mas também a troca e o compartilhamento de idéias e ações, conduzindo a educação a um processo cognitivo, abrangente e transdisciplinar.

3. A Arquitetura SABERLÂNDIA

Formalismo do Jogo. Para implementar o sistema proposto, formaliza-se o jogo das seguinte forma:

- *elementos gráficos*: conjunto de objetos virtuais a serem confeccionados pelo autor do jogo. Um objeto, obj_i, é classificado pelo autor como sendo um *prêmio*, *barreira*, *chave* ou *cenário*. Prêmios são os elementos gráficos que devem ser resgatados, barreiras são elementos gráficos que restringem o acesso à obtenção dos prêmios e as chaves são elementos gráficos que permitem a liberação dos diferentes prêmios espalhados pelo cenário. Finalmente, cenários são elementos gráficos que compõe o ambiente de jogo, não podendo, ao contrário dos demais, terem seu estado modificado ao longo do jogo.

- *atributos*: são características associadas aos diferentes elementos do jogo, que podem ser modificadas durante a partida. Um atributo, $attr_j$, descreve uma característica de um determinado elemento obj_i, $attrj(obj_i)$. O conjunto de atributos verdadeiros em determinado instante de tempo *t* descreve o estado do mundo *W(t)* associado ao jogo.

- *ações*: ao longo da partida, os atributos dos diferentes elementos são modificados por eventos que os jogadores realizam, conduzindo a diferentes seqüências de estado de mundo. Estes eventos são formalizados através de ações descritas através do formalismo STRIPS (Silva, 2000). Neste formalismo, as ações são compostas de três campos básicos: i. pré-condições: conjunto de atributos que necessitam ser verdadeiros para que a ação possa ser aplicada, ii. efeitos de adição: atributos que são adicionados ao estado do mundo W(t) após aplicação da ação e iii. efeitos de remoção: atributos que são deletados do mundo W(t) após a realização da ação.

Dinâmica do Jogo. O formalismo adotado visa permitir a descrição de diferentes dinâmicas de jogo. Entretanto, devido à necessidade de visualização gráfica dos efeitos das diferentes ações no ambiente de jogo, neste primeiro momento busca-se desenvolver um jogo de aventura em terceira pessoa que permita a criação de diferentes roteiros que envolvam situações virtuais e reais, onde avatares/robôs devem procurar/resgatar elementos dispostos em diferentes regiões do cenário desenvolvido. Assim, parte-se de um conjunto de ações cuja lista de efeitos é preestabelecida do ponto de vista gráfico.

De acordo com o roteiro criado pelo autor, os personagens explorarão tais cenários de forma virtual, movimentando-se através de ambiente 3D ou real, através da navegação teleoperada pelo computador, e/ou dos veículos robóticos pelas maquetes desenvolvidas.

Ao longo da evolução do jogo, obtida através da realização de *ações* pelos seus jogadores, *prêmios* são resgatados, sendo apresentadas, ao jogador, partes do *roteiro* criado, bem como os conteúdos relacionados ao conhecimento a ser adquirido. Conteúdos podem ter associados à testes e desafios, que validarão o elemento resgatado ou não conforme resposta dada pelo jogador.

Os resgates podem ser realizados de forma individual por um jogador, ou de forma colaborativa em grupo de jogadores.

A seqüência de *prêmios* a serem resgatados é obtida de forma dinâmica através do uso de *planificadores* advindos da Inteligência Artificial. O *planificador* estabelece uma seqüência adequada de ações a ser cumprida pelo jogador(es), em função do conjunto de elementos descritos, nível do aluno e características do conteúdo a ser apresentado.

Tal dinâmica é implementada em uma plataforma multi-tecnológica, onde a partir de cenários e personagens, próprios à realidade dos grupos, utilizam-se situações-problema como ferramentas pedagógicas de ensino colaborativo. Para tal, o sistema SABERLÂNDIA é dividido em três módulos principais: o i. Módulo de Autoria, ii. Módulo Jogo, e o iii. Módulo Recursos Multi-Tecnológicos. A seguir detalham-se as características e metodologia associadas a cada um destes módulos.

3.1 Módulo Autoria

O módulo autoria é responsável por permitir a criação de diferentes cenários e personagens, bem como diferentes roteiros e conteúdos associados às características do grupo e conhecimentos a serem trabalhados. O módulo é composto de dois principais blocos: *i*. o editor de jogo e *ii*. a biblioteca de elementos .

3.1.1. O Editor de Jogo

Foi desenvolvido um sistema de edição visando possibilitar a autoria do roteiro, conteúdos, cenários e personagens. Tal sistema é baseado na biblioteca de código aberto *Radiant*, utilizado pela *Id Software* na criação do jogo *Quake*.

Editando cenários e Personagens: partindo-se de um cubo que encapsula todo o cenário onde o jogo se passará, o Editor de Jogo permite a criação de objetos tridimensionais simplificados, utilizados para criação de paredes e demais objetos que possuam formas geométricas primitivas. Podem também ser definidos *scripts* que possibilitam a criação de efeitos como: texturas com diversos graus de transparência, chamadas de *Alpha Channel*; propriedades de penetração e densidade. *Elementos gráficos* podem ser criados e colocados no cenário, associando-se a estes *atributos* (*regiões* a que pertencem, estados quanto a *resgate*, *abertura*, etc), bem como *categorias* (*prêmio, barreira, chave ou cenário*) e roteiros/conteúdos relacionados. Cada *atributo* especificado, se modificável ao longo do jogo, deve possuir descrição de *ação* que permita tal modificação.

***Editando Roteiros e Conteúdos*:** como apresentado anteriormente, o SABERLÂNDIA apresenta a implementação de um jogo de aventura em terceira pessoa, cujo objetivo do(s) personagens é resgatar elementos (*prêmios*) distribuídos no cenário. Mesmo partindo de tal dinâmica de jogo fixa, o sistema permite a criação de diferentes *roteiros*, que associados à *conteúdos* distintos, permite o desenrolar de diferentes histórias ambientadas em diferentes cenários e realizadas por diferentes personagens.

O roteiro é apresentado ao longo do jogo. *Ações* realizadas pela obtenção de *chaves*, aberturas de *barreiras* e resgate de *prêmios* conduzem a apresentação de partes da história ao jogador. Esta história é fornecida pelo autor através de uma interface que permite a criação de textos com diferentes recursos multi-mídia (som, imagens e filmes). Conteúdos podem ser associados a tais *ações*, inclusive com a disponibilidade de inserção de questões de múltipla escolha associadas a estes. Cabe ao autor definir um ordenamento parcial de apresentação de tal roteiro/conteúdo ao jogador.

O autor pode fixar conteúdos/trechos de roteiros à ações específicas, ou pode delegar tal associação ao próprio planificador do SABERLÂNDIA. Neste caso de forma automática, o sistema atrelará o roteiro a ser apresentado ao aluno às diferentes possíveis ações a serem realizadas ao longo da partida.

3.1.2 A Biblioteca de Elementos

Para criação de objetos mais complexos, com formas orgânicas, como avatares, terrenos, carros ou árvores, é requerida a utilização de softwares que envolvam maior complexidade na manipulação. A fim de que, professores e estudantes, não estivessem privados do uso destes objetos, foi criado um banco de objetos modelados em *low-poly* (modelos com menor número de polígonos) e texturizados com mapeamento UVW. Desta forma, ao criarem seus mapas os usuários poderão simplesmente agregar estes elementos. Assim, a Biblioteca de Elementos implementa uma biblioteca virtual com objetos em 3D, tais como casas, ruas, relevos, rios, etc, permitindo compor diferentes cenários. Um conjunto de personagens virtuais também é disponibilizado, desde elementos mais genéricos como meninos e

meninas, passando por animais, personagens de lendas, etc...

De posse da biblioteca virtual, figuras representativas dos cenários e personagens poderão ser impressas (via impressora comum) e anexadas às maquetes de elementos de cenários e sistemas robóticos entregues através de kit. As figuras poderão ser facilmente anexadas, recolocadas e retiradas das maquetes e sistemas robóticos.

3.2 Módulo Jogo

Este módulo é responsável pelo controle e realização do jogo, incorporando conceitos pedagógicos à plataforma. De forma mais precisa, o módulo é responsável pelas seguintes atividades: *i.* Roteiro de Partida, *ii.* Motor de Jogo e *iii.* Interface Interativa.

3.2.1 O Roteiro da Partida

A proposta visa o uso de jogos como ferramenta educacional. Conteúdos serão associados ao jogo, de forma que ao longo de cada partida sejam apresentados ao jogador/aluno conhecimentos a serem transmitidos. O Autor do jogo estabelece tais conteúdos fixando, a priori, apenas uma ordenação parcial entre estes. O planificador além de estabelecer a seqüência de ações a ser realizada para obtenção dos prêmios a cada partida, distribui também o conteúdo informado pelo autor, respeitando o ordenamento parcial fornecido.

A seqüência de ações, com seus devidos conteúdos, fornecida pelo planificador constitui o *roteiro* da partida. Mais do que a implementação de um jogo de resgaste de elementos através de cenário, o SABERLÂNDIA permite que a cada nova jogada, sejam estabelecidas diferentes *ações* de resgate, resultando em diferentes seqüências de obtenção de *chave/barreira/prêmio*, respeitando apenas a relação de precedência entre os *conteúdos* fornecidos pelo autor do jogo.

***Criando Novas Partidas*:** o planificador, a partir do estado de mundo inicial, W_{t0}, definido pelo autor, gera a cada nova partida, de forma autônoma, uma seqüência de ações de procura/resgate a ser realizado pelo jogador.

Conjunto de diferentes custos podem ser atribuídos a cada tipo de ação conduzindo a partidas diferentes. Pode-se modificar tais custos ao longo do jogo, em função da atuação do(s) jogadores(s), conduzindo a possibilidade de uma dinâmica de jogo auto-configurável *on-the-fly* pelo sistema. Ou seja, o jogo percebendo a atuação do aluno, pode ter seus objetivos modificados durante cada partida.

3.2.2 O Motor do Jogo

O Motor do Jogo é o conjunto de procedimentos que controlam a evolução do jogo. Tais procedimentos associam-se a duas atividades principais: *i.* a verificação, monitoramento e liberação das possíveis ações, $action_i$, a serem realizadas no tempo *t; ii.* atualização do estado do mundo $W_{(t+1)}$ após a aplicação das ações, $action_i$, em *t*.

A cada instante *t*, tem-se como verdadeiro um conjunto de atributos $attr_i(obj_j)$ que descrevem o estado do mundo, W_t. O motor de jogo verifica quais ações são aplicáveis em *t*, . permitindo que estas sejam realizadas pelo jogador.

Uma vez realizadas as ações, $action_i$,pelo jogador em *t*, o Motor de Jogo é responsável por atualizar o novo estado do mundo em $(t+1)$, $W_{t+1.}$ através da adição e deleção de atributos associados ao efeitos de $action_{i,,}$ bem como é responsável pela apresentação dos conteúdos e pontuação do(s) jogadore(s) de acordo com o roteiro estabelecido pelo autor e fornecido pelo planificador.

Cabe também ao Motor do Jogo a avaliação do desempenho do jogador, de forma a adequar a dinâmica do jogo aos atores envolvidos a partida corrente.

3.3.3 A Interface Interativa

A interface interativa é responsável pela renderização e visualização do sistema 3D. A interface deve apresentar graficamente ao(s) jogador(es) os diferentes objetos e seus atributos presentes a cada estado do mundo W_t . Tal representação deve considerar aspectos inclusive de simulação física dos elementos.

A interface interativa desenvolvida é baseada no motor de jogo do *Quake 3*: Arena[1] [__]. A interface permite o gerenciamento das diversas possibilidades de interação com o ambiente, de forma a seguir estritamente a dinâmica gerada pelo planificador. É também responsável por gerenciar os conteúdos que foram inseridos pelo educador e apresentá-los ao jogador de uma forma que estes não possam ser simplesmente ignorados, e sim que sejam parte fundamental da estrutura do jogo.

3.3 Módulo Recursos Multi-Tecnológicos

O módulo Recursos Multi-Tecnológicos congrega uma série de recursos multi-mídias que podem ser inseridos pelo autor e utilizados durante o jogo. Uma série de recursos estão presentes, tais como:

[1] O jogo *Quake* 3 teve seu código liberado pela desenvolvedora *Id Software* [__[, sob licença GPL [__].

- possibilidade de inclusão de vídeos, imagens e sons nos ambientes, retratando, inclusive, em tempo real a situação dos participantes,
- utilização de sistemas robóticos de baixo custo, compostos de pequenos veículos teleoperados, que permitem a realização de ações da mesma forma que personagens virtuais no cenário virtual do jogo.
- maquetes compostas de diferentes blocos cujo *layout* possa ser modificado, e que compõem cenário real a ser explorado pelos robôs.
- sistema para a comunicação entre o jogo virtual e o sistema robótico;
- disponibilização online da plataforma com possibilidade de uso remoto por diferentes jogadores.

Com base no formalismo apresentado e nos módulos descritos é possível o desenvolvimento de um sistema de autoria para jogos de ação bastante genéricos. De forma a permitir sua efetiva utilização por professores do ensino fundamental, optou-se por restringi-lo a criação de jogos de ação em terceira pessoal, associado a busca/resgate de prêmios. O sistema efetivamente implementado apresenta-se descrito na próxima sessão.

4. A Implementação do Jogo SABERLÂNDIA – Testes e Análises

Um sistema de autoria que possibilite a criação de um jogo de ação onde o objetivo principal é fazer com que o jogador explore os vários ambientes existentes em busca de um objetivo, possibilitando a existência de diferentes tramas foi desenvolvido.

Para avançar no jogo, o jogador deve seguir uma ordem específica de interações com objetos do ambiente, fornecidas de forma automática por um planificador. Tais interações liberam ações que antes não seriam possíveis.

As interações ocorrem da seguinte forma: enquanto move-se livremente pelos locais acessíveis do mapa, o jogador é sinalizado na tela ao encontrar um objeto com o qual pode executar alguma ação; ao interagir podem ser apresentados conteúdos e desafios que devem ser superados para possibilitar o avanço.

Os conteúdos/desafios são fornecidos pelos próprios professores, contextualizados conforme sua vivência, e podem ser apresentados em forma de questões de múltipla escolha, para facilitar a "comunicação" entre o aluno e o computador, ou através de respostas discursivas. Inserir os conteúdos na criação do jogo é uma tarefa simples, não necessitando conhecimento técnico por parte dos educadores.

Além de um jogo-modelo completo acompanhar a Plataforma Saberlândia, é disponibilizada uma ferramenta de edição de jogos, permitindo ao usuário a criação de seus próprios jogos, bem como bibliotecas de modelos prontos. A ferramenta, baseada na biblioteca *Radiant*, utilizado pela *Id Software* na criação do jogo *Quake,* é apresentada na figura 1.

Esta ferramenta permite a criação de *brushes*, os quais são objetos tridimensionais simplificados, ideais para criação de paredes e demais objetos que possuam formas geométricas primitivas. Neste mesmo programa é possível a aplicação de texturas sobre a superfície destes brushes, seu mapeamento, a iluminação do cenário e a inserção de pontos específicos de materialização dos personagens no jogo.

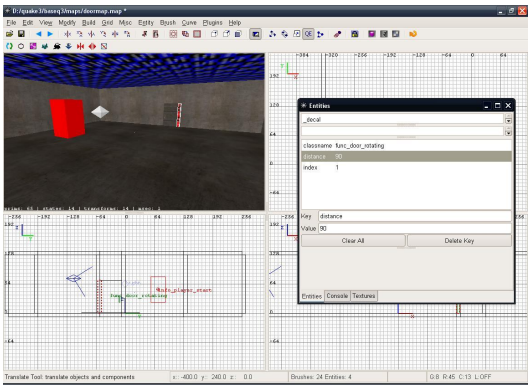


Figura 1: Editando cenários e personagens

Por meio da utilização de scripts, já implementados, é possível a criação do *skybox*, composto por texturas especiais aplicadas sobre as superfícies de um cubo que envolve o cenário, o qual dará origem a atmosfera do jogo. A utilização de scripts permite também a criação de efeitos como: texturas com diversos graus de transparência, a criação de água, onde os *brushes* terão propriedades de penetração e densidade; água corrente e movimento de fogo e fumaça.

A figura 2 apresenta alguns dos cenários desenvolvidos e que compõem a biblioteca do jogo:

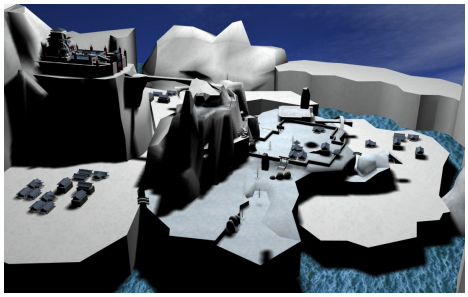


Figura 2: Alguns Cenários do jogo “Os quatro elementos”

Além de criar e compilar os mapas, o editor também é responsável por gerar uma estrutura lógica das áreas de um mapa que será usada como entrada no planificador. Esta entrada é apresentada em linguagem específica, gerada a partir de uma análise lógica dos modelos e estruturas do mapa.

Foi utilizado o planificador IPP escrito em C++ utilizando linguagem PDDL no domínio STRIPS. A seguir é apresentado um exemplo de elementos gráficos fornecidos ao planificador e que compõem os componentes do jogo.

```
(((DOMAINS::PEGA-ITEM DOMAINS::AVA DOMAINS::ITEM3 DOMAINS::REG1
DOMAINS::AREA3)
 (DOMAINS::PEGA-ITEM DOMAINS::AVA DOMAINS::ITEM1 DOMAINS::REG1
DOMAINS::AREA3)
 (DOMAINS::PEGA-ITEM DOMAINS::AVA DOMAINS::ITEM2 DOMAINS::REG1
DOMAINS::AREA3))
 ((DOMAINS::DESTRAVA-BARREIRA DOMAINS::AVA DOMAINS::BAR1
DOMAINS::ITEM3 DOMAINS::AREA1 DOMAINS::REG1))
 ((DOMAINS::MOVE DOMAINS::AVA DOMAINS::AREA1 DOMAINS::AREA3
DOMAINS::REG1))
 ((DOMAINS::DESTRAVA-BARREIRA DOMAINS::AVA DOMAINS::BAR3
DOMAINS::ITEM2 DOMAINS::AREA3 DOMAINS::REG1))
 ((DOMAINS::MOVE DOMAINS::AVA DOMAINS::AREA3 DOMAINS::AREA2
DOMAINS::REG1))
 ((DOMAINS::DESTRAVA-BARREIRA DOMAINS::AVA DOMAINS::BAR2
DOMAINS::ITEM1 DOMAINS::AREA2 DOMAINS::REG1)))

(define (domain saberlandia-strips)
 (:requirements :typing)
 (:types regiao area barreira item avatar)
 (:predicates (regiao ?r - regiao)
        (regiao-liberada ?r - regiao)
        (area ?a - area)
        (barreira ?b - barreira)
        (item ?i - item)
        (area-regiao ?r - regiao ?a - area)
        (barreira-area ?a - area ?b - barreira)
        (area-aberta ?area - area)
        (fechada ?b - barreira)
        (aberta ?b - barreira)
        (item-barreira ?b - barreira ?i - item)
        (avatar ?A - avatar)
        (item-avatar ?A - avatar ?i - item)
        (avatar-regiao ?r - regiao ?A - avatar)
        (avatar-area ?a - area ?A - avatar)
        (item-regiao ?r - regiao ?i - item)
        (item-area ?a - area ?i - item)
        (sem-item ?A - avatar ?i - item)
        (deletado ?i - item)
```

Um conjunto de ações pode ser definido, que modificarão o estado do mundo, permitindo a

evolução do mesmo. A seguir apresenta-se um exemplo de um conjunto de ações definidas para a dinâmica de resgate de prêmios.

```
(:action destrava-barreira
   :parameters (?Avatar - avatar ?barreira - barreira ?item - item
?area - area ?regiao - regiao)
   :precondition(
          and
            (regiao ?regiao)
            (regiao-liberada ?regiao)
            (area ?area)
            (area-regiao ?regiao ?area)
            (barreira ?barreira)
            (barreira-area ?area ?barreira)
            (fechada ?barreira)
            (avatar ?Avatar)
            (avatar-area ?area ?Avatar)
            (item ?item)
            (item-avatar ?Avatar ?item)
            ;;(item-barreira ?barreira ?item)
   )
   :effect (
        and (aberta ?barreira)
           (area-aberta ?area)
           (deletado ?item)
           (not (fechada ?barreira))
           (not (item-avatar ?Avatar ?item))
   )
)
(:action pega-item
 :parameters (?Avatar - avatar ?item - item ?regiao - regiao ?area - area)
 :precondition (
          and (regiao ?regiao)
            (avatar ?Avatar)
            (item ?item)
            (area ?area)
            (sem-item ?Avatar ?item)
            (not (deletado ?item))
            ;;(or (
            ;;     (an?itemd (item-regiao ?regiao ?item)
(avatar-regiao ?regiao ?Avatar))
            ;;     (and (item-area ?area ?item) (avatar-area
?area ?Avatar) (area-regiao ?regiao ?area))
            ;;   )
            ;;)
 )
 :effect (and (item-avatar ?Avatar ?item)
          (not (sem-item ?Avatar ?item))
          ;;(not( or (item-area ?area ?item) (item-regiao
?regiao ?item)))
 )
)
(:action move
 :parameters (?Avatar - avatar ?origem - area ?destino - area
?regiao - regiao)
 :precondition (
          and (regiao ?regiao)
            (area ?origem)
            (area ?destino)
            (area-regiao ?regiao ?origem)
            (area-regiao ?regiao ?destino)
            (avatar ?Avatar)
            (avatar-area ?origem ?Avatar)
            (not(area-aberta ?destino))
 )
 :effect (
          and (avatar-area ?destino ?Avatar)
            (not (avatar-area ?origem ?Avatar))
 )
```

O planificador pode ser chamado a qualquer instante de tempo, t, com a descrição do mundo, Wt e a meta a ser atingida. A seguir apresenta-se um exemplo de tal chamada.

```
(define (problem saberlandia)
 (:domain saberlandia-strips)
 (:objects Ava - avatar Area1 Area2 Area3 - area Bar1 Bar2 Bar3 -
barreira Reg1 - regiao Item1 Item2 Item3 - item)
 (:init
   (avatar Ava)
   (area Area1)
   (area Area2)
   (area Area3)
   (barreira Bar1)
   (barreira Bar2)
   (barreira Bar3)
   (regiao Reg1)
   (item Item1)
   (item Item2)
   (item Item3)
   (avatar-regiao Reg1 Ava)
   (area-regiao Reg1 Area1)
   (area-regiao Reg1 Area2)
   (area-regiao Reg1 Area3)
   (barreira-area Area1 Bar1)
   (barreira-area Area2 Bar2)
   (barreira-area Area3 Bar3)
   ;(item-barreira Bar1 Item1)
   ;(item-barreira Bar2 Item2)
   ;(item-barreira Bar3 Item3)
   ;(item-regiao Reg1 Item1)
   ;(item-regiao Reg1 Item2)
   ;(item-regiao Reg1 Item3)
   (fechada Bar1)
   (fechada Bar2)
   (fechada Bar3)
   (regiao-liberada Reg1)
   (sem-item Ava Item1)
   (sem-item Ava Item2)
   (sem-item Ava Item3)
   (avatar-area Area1 Ava)
 )
 (:goal
   (and
    (aberta Bar1)
    (aberta Bar2)
    (aberta Bar3)
   )
 )
```

O Motor do Jogo gerencia as diversas possibilidades de interação dos personagens com o ambiente, de forma monitorar se as ações destes seguem a dinâmica gerada pelo planificador. É também responsável por gerenciar o roteiro e conteúdos que foram inseridos pelo educador e apresentá-los ao jogador de uma forma que estes não possam ser simplesmente ignorados, pois são parte fundamental da estrutura do jogo.

Os elementos robóticos utilizados são construídos a baixo custo utilizando a plataforma móvel FURGBOL. Tais robôs móveis (figura 3) representam em tabuleiro o movimento dos personagens no jogo virtual.

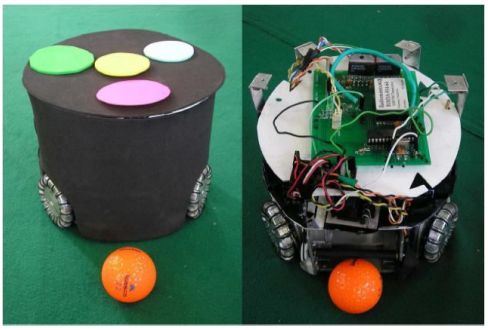


Figura 3: Fotos do robô móvel

Os diferentes módulos da arquitetura compõem hoje o primeiro protótipo do jogo. O sistema permite a visualização estereoscópica e utilização em rede.

Outra função importante desempenhada pela Interface do Sistema é apresentar, de forma amigável, informações como inventário dos elementos e alguns tipos de animações para tornar o jogo mais atrativo.

5 Conclusão e trabalhos futuros

Face às mudanças no paradigma pedagógico e à evolução das tecnologias, tais como o computador e a *Internet*, os professores têm buscado o uso de recursos que extrapolem a visão tradicional e os métodos meramente discursivos no processo de ensino-aprendizagem. Assim, os jogos educacionais se configuram num recurso motivador tanto para o professor como para o aluno, como uma ferramenta complementar na construção do conhecimento, chegando à sala de aula.

A possibilidade de autoria de jogos educacionais, de forma fácil, sem que seja necessário um conhecimento aprofundado em computação, pode fazer com que professores e pedagogos se interessem pelo desenvolvimento de jogos educacionais, e passem a utilizar este recurso lúdico em sua sala de aula.

Além disso, busca-se a possibilidade de autoria de jogos cuja dinâmica seja bem atraente a crianças e jovens, como por exemplo jogos de ação e aventura.

Assim, neste artigo apresentou-se um sistema de autoria para jogos de aventura educacionais, que integra várias mídias em um ambiente virtual de aprendizagem. O sistema permite ao professor conceber diferentes roteiros para relacioná-los com diversas disciplinas, possibilitando que novos desafios sejam propostos.

O jogo é descrito formamente através do formalismo STRIPS. Um planificador do tipo GRAPHPLAN é utilizado como gerador automático de ações e seqüências de conteúdos.

Foi apresentada a arquitetura do sistema, bem como o estado atual da sua implementação. Foi também apresentado o editor de jogo, conjunto de componentes da biblioteca, exemplo de descrição de mundo e ações, bem como o kit robótico disponibilizado.

Em sua fase atual desenvolve-se o avaliador, capaz de solicitar, *on-the-fly*, adequações da seqüenciabilidade do jogo, em função do desempenho do(s) jogador(es). Também se finaliza o desenvolvimento do Módulo Interface e a implementação das maquetes associadas ao kit-robótico.

Como trabalhos futuros, pretende-se associar as diferentes vertentes pedagógicas as diferentes possibilidade de apresentação dos conteúdos, estabelecendo-se correlação entre os diferentes algoritmos de busca por ações e as diferentes linhas psico-pedagógicas.

Referências

____ **FURGBOL**. Disponível em http://www.ee.furg.br/~furgbol/, acessado em jun. 2008.

____ **GNU General Public License, Free Software Foundation**. Disponível em http://www.gnu.org/copyleft/gpl.html, acessado em jun. 2008.

____ **ID SOFTWARE, Game Developer.** Disponível em http://www.idsoftware.com/, acessado em jun. 2008.

___ **QUAKE 3: Arena**, Id Software, 1999. Disponível em http://www.idsoftware.com/games/quake/quake3-arena/, acessado em jun. 2008.

BEHAR, P. et al. **Projeto ROODA : a construção de um ambiente para EAD baseado em Software Livre.** Disponível em http://www.nuted.edu.ufrgs.br/biblioteca/arquivo.php?arq=37, acessado em jul. 2005

D'ABREU, J. V. V., Chella, M. T. **Ambiente Colaborativo de Aprendizagem a Distância Baseado no Controle de Dispositivos Robóticos**. In: XII Simpósio Brasileiro de Informática na Educação – SBIE2001, Universidade Federal de Espírito Santo – UFES, Vitória – ES. Anais.... . 2001.

D'AMBRÓSIO, Ubiratan. **Da Realidade à Ação: reflexões sobre educação e Matemática**. 2ªed. São Paulo: Summus, 1986.

FAGUNDES, L. C., Sato, L. S. e Maçada, D. L. **Aprendizes do futuro: as inovações começaram!.** Brasília: MEC. 1999.

LÉVY P. **As Tecnologias da Inteligência - O Futuro do Pensamento na Era da Informática.** Editora 34. Rio de Janeiro. 1995.

LÉVY, P. **Cibercultura**. São Paulo: Editora 34. 1999.

MANTOAN, M. T. E.; Stegun, M. C, B.; Baranauskas, M. C. C.; Barcellos, G. C. **O Processo Comunicativo em Ambientes Virtuais de Aprendizagem: Uma Proposta, um Estudo Exploratório**. In: X Simpósio Brasileiro de Informática na Educação – SBIE99 "As Novas Linguagens da Tecnologia na Aprendizagem". Universidade Federal do Paraná – UFPR, Curitiba – PR Anais... 1999.

MERIEU, P. **Aprender...Sim, mas como?** Porto Alegre: Artmed, 1998.

PALLOFF E PRATT. **Construindo Comunidades de Aprendizagem no Ciberespaço**. Artmed, Porto Alegre. 2002.

RAMAL, A. C. **Educação na cibercultura: hipertextualidade, leitura, escrita e aprendizagem**. Porto Alegre: Artmed. 2002.

SENA, G.; Moura, J. **Jogos eletrônicos e educação: novas formas de aprender**. Disponível em http://www.gamecultura.com.br/index.php?option=com_content&task=view&id=438&Itemid=9, acessado em junho, 2008.

SILVA, F. **Algoritmo para Planificação baseada em STRIPS**. Dissertação de Mestrado. Programa de Pós-Graduação em Informática da Universidade Federal do Paraná. Curitiba, 2000.

TAHAN, M. **O homem que calculava**. Rio de Janeiro: Record,1968.

VALLE FILHO, A. M. ; SOUZA, P. ; ALVES, J. B. M. ; WAZLAWICK, R. S. ; LUZ, R. P. **Ferramentas de Autoria de Realidade Virtual - um estudo comparativo**. In: VI Congreso Internacional de Ingeniería Informática, 2000, Buenos Aires. VI Congreso Internacional de Ingeniería Informática. Buenos Aires : ICIE 2000, 2000.

VIGOTSKY, L.S. **Pensamento e linguagem**. São Paulo: Martins Fontes, 1984.

Jogo Simulador de Vida Artificial Implementado em Hardware Reconfigurável

Felipe A. Navas Eduardo V. Simões

Universidade de São Paulo – Instituto de Ciências Matemáticas e de Computação

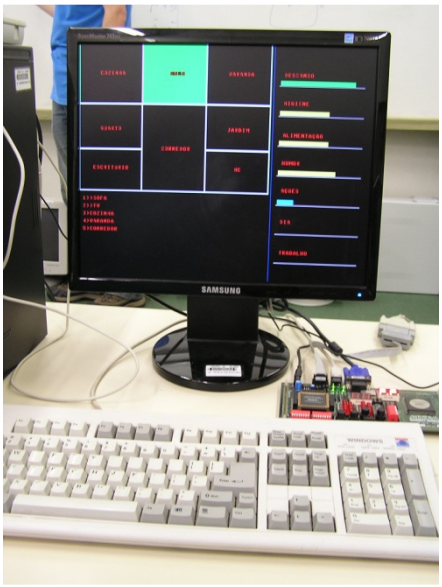


Figura 1: O jogo rodando na placa de desenvolvimento UP1, com suas interfaces de teclado e monitor

Abstract

This paper presents a one-chip hardware implementation of a strategic life-simulation game. This work makes use of a FPGA (Field-Programmable Gate Array) that can be configured by the user in the field, through a Computer-Aided Design software. The game environment is implemented in VHDL language, together with a keyboard capture module and a VGA video interface. The article shows how this hardware solution is optimized to run in a low-cost FPGA device, with severe speed and memory limitations. The resulting game makes use of the intrinsic parallelism of the logic cell array of the FPGA, that is able to read the user input, analyse the selected actions, redraw the graphic environment and generate video signals that control the monitor. The results show a medium complexity life-simulation game, integrated with its keyboard interface and video control module, that runs in a 25MHz hardware.

Palavras Chave: hardware reconfigurável, FPGA, vida artificial

Contato dos Autores:
felipenavas@gmail.com
simoes@icmc.usp.br

1. Introdução

Este artigo descreve a implementação de um jogo desenvolvido completamente em hardware, integrando em um único chip, o hardware e os algoritmos de um simulador de vida artificial. Foi utilizado, para isso, um FPGA (Field-Programmable Gate Array), que permite a configuração de suas células programáveis pelo usuário em campo, através de uma ferramenta de CAD (Computer Aided Design) [QUARTUS, 2004]. O resultado equivale a um videogame completo em um único chip, contendo interface com teclado e vídeo (ver Figura 1).

As vantagens da implementação de um jogo em hardware incluem [WOLLINGER, T. , GUAJARDO, J., PAAR, C. 2004]: uma proteção maior contra abusos de falhas de segurança e pirataria, uma vez que o usuário não tem acesso aos recursos do chip depois que este é programado e que nenhuma parte do jogo é implementada em software, este não pode ser copiado ou alterado sem que se realize o rompimento do encapsulamento do chip do FPGA. Outras vantagens são a não dependência das limitações impostas por uma plataforma específica e a possibilidade de criação de hardware dedicado para executar de maneira mais eficiente as necessidades do jogo, como, por exemplo, a física e a inteligência artificial [BOULÉ, M. and ZILIC, Z. 2002].

Os primeiros jogos eram implementados em hardwares específicos, mas a generalização das plataformas e o surgimento dos videogames tornaram mais rápido e eficiente a produção de jogos, que passaram a ser implementados em software [DEMARIA R., WILSON, J. 2002]. Com a produção em larga escala dos videogames, o custo de um hardware específico para cada jogo tornou-se impraticável, assim a

indústria de jogos passou a se dedicar ao desenvolvimento de softwares para plataformas específicas. Atualmente, os jogos tem se tornado cada vez mais complexos e os fabricantes preocupam-se novamente com o desenvolvimento de hardwares específicos para executar necessidades como aceleração de gráficos 3D, como as placas de vídeo e, mais recentemente, a aceleração de física [LIENHART, G., KUGEL, A., MANNER, R. 2002].

O desenvolvimento de jogos em hardware específico tem apresentado alguns problemas como a impossibilidade de modificar o jogo após este ser comercializado [CRABILL, E. 2005]. Isso faz com que o jogador perca rapidamente o interesse após jogar diversas vezes. Em contraste, nos videogames atuais, o jogador tem a possibilidade de utilizar o mesmo hardware para jogar diversos jogos, apenas sendo necessária a mudança da mídia na qual o jogo em software está armazenado. Entretanto, a utilização de hardware reconfigurável, como os FPGAs, vem tornar possível modificar a configuração do jogo e até mesmo a substituição do mesmo por meio da completa reprogramação do hardware do chip [WATERMAN, S., 2005]. Assim, novos jogos poderão ser jogados se o FPGA for reinicializado com novos arquivos de configuração. Isso vem estimular o desenvolvimento de novos jogos principalmente para plataformas móveis, que passam a poder contar com soluções totalmente embarcadas em um único chip reconfigurável de baixo custo.

2. Descrição do Jogo

Para a implementação do jogo foi utilizada a placa de Desenvolvimento com FPGA UP1 da Altera e, como plataforma de desenvolvimento, o software Quartus II [QUARTUS, 2004]. A placa possui conectores para teclado e vídeo tipo VGA, mas o hardware de interface com o teclado de computador e o monitor de vídeo tiveram de ser implementados utilizando-se o próprio FPGA. A interface com o monitor de computador implementada para este projeto suporta resolução de 640x480 pixels e oito cores por pixel. O hardware utilizado oferece uma pequena quantidade de memória, apenas 4KB, o que exigiu o desenvolvimento de técnicas para lidar com essa limitação.

O tema do jogo escolhido é a simulação da vida de um aluno universitário da Universidade de São Paulo que realiza o trabalho final de uma disciplina e, paralelamente, enfrenta os afazeres diários em sua casa. O jogador deve administrar os atributos do avatar como alimentação, higiene, cansaço e humor; levando-o pela casa e executando ações como tomar banho, dormir e estudar; de maneira que consiga manter uma qualidade de vida mínima enquanto o avatar desenvolve o seu trabalho de faculdade. O avatar morrerá caso algum dos seus atributos chegue à zero. Não conseguindo terminar o trabalho até o prazo ou em caso de morte do avatar, o jogador perde o jogo. O jogador vence se conseguir entregar o trabalho antes do final do prazo.

O jogador começa com sete pontos em cada um dos atributos: `descanso`, `higiene`, `alimentação` e `humor`. As variáveis `trabalho`, `dia` e `ações` são inicializadas com zero. O jogador, através de comandos pelo teclado, pode mover-se pela representação da casa e executar diferentes ações em cada cômodo. Cada ação incrementa ou decrementa um ou mais atributos ou variáveis, sendo necessário o jogador manter um planejamento e raciocínio constante para equilibrá-los, de modo a conseguir paralelamente desenvolver o seu trabalho antes do final do prazo. O valor máximo para os atributos é dez e o mínimo é zero. Pode-se ver, na Tabela I, as alterações provocadas em cada atributo e variável ao executar cada ação.

Tabela I: Mudanças provocadas por cada ação

Ação	Mudança	
Navegar na Internet	Descanso	−2
	Higiene	−1
	Alimentação	−1
	Ações	+2
Dormir na Cama	Descanso	+5
	Higiene	−1
	Humor	+2
	Alimentação	−2
	Ações	+4
Usar o Banheiro	Descanso	+1
	Higiene	+5
	Humor	+1
	Alimentação	−1
	Ações	+2
Ir para a "Balada"	Descanso	−3
	Higiene	−2
	Humor	+4
	Alimentação	−2
	Ações	+2
"Atacar" a geladeira	Descanso	−1
	Higiene	−1
	Alimentação	+4
	Ações	+2
Ir para a Faculdade	Descanso	−5
	Higiene	−3
	Humor	−2
	Alimentação	−2
	Trabalho	+1
	Ações	+4
Ir para a Casa	Descanso	−3
	Higiene	−1
	Humor	+2
	Alimentação	−2
	Ações	+4
Fazer Trabalho	Descanso	−4
	Higiene	−1
	Humor	−5
	Alimentação	−4
	Trabalho	+3
	Ações	+4
Ir para o Bar	Descanso	−1
	Higiene	−1
	Humor	+2
	Alimentação	+1
	Ações	+3

Ao executar uma ação, o número de `ações` é incrementado conforme demonstrado na Tabela I. Ao atingir um número de `ações` maior que 11, o número de `dias` é incrementado e zero é atribuído ao número de `ações`. O jogo começa no `dia` igual a zero, e termina quando atinge o `dia` dez. Se o jogador chegar ao décimo `dia` com o `trabalho` igual a 10 ou maior, o jogador vence o jogo; caso contrário, ele não conseguiu terminar o trabalho antes do prazo final perdendo o jogo. O jogador também perderá se a qualquer momento um dos atributos possuir um valor igual ou abaixo de zero.

3. Implementação do Jogo em Hardware

Três módulos responsáveis por todas as funções realizadas pelo jogo foram desenvolvidos e implementados em linguagem VHDL (Very High Speed Integrated Circuit Hardware Description Language): a interface com teclado, a lógica do jogo e o módulo de vídeo [WATERMAN, S., 2005]. A Figura 2 mostra a relação entre estes módulos.

3.1 Módulo de Interface com o Teclado

A placa UP2 possui um conector para o teclado de computador do tipo PS2 ligado aos pinos de entrada e saída do FPGA. Foi implementada a lógica para a decodificação dos sinais emitidos pelo teclado em sinais utilizáveis pelo jogo. Este bloco consiste de uma interface serial para receber bit a bit os bytes correspondentes às teclas pressionados e um bloco de filtragem que traduz os bytes enviados no código ASCII de cada tecla. Estes códigos são enviados ao módulo que processa a lógica de jogo. A figura 2 apresenta um diagrama representativo desta configuração e também mostra a arquitetura do sistema e a conexão entre os blocos. As nove ações possíveis são escolhidas através das teclas numéricas de "1" a "9" do teclado.

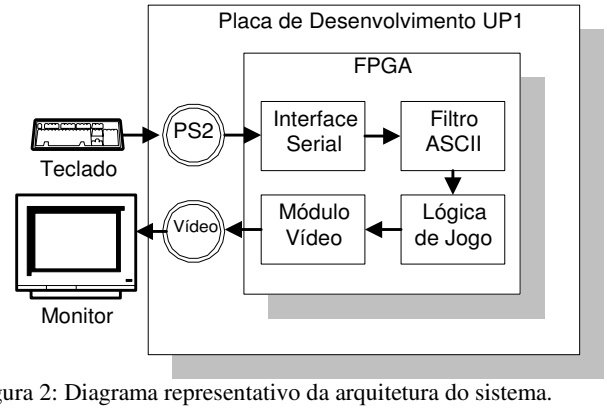


Figura 2: Diagrama representativo da arquitetura do sistema.

3.2 Módulo de Lógica do Jogo

A lógica do jogo foi implementada como uma máquina de estados, conforme descrito na Figura 3. Cada ação que o jogador pode executar é um estado da máquina de estado. Foram implementados os nove estados da Tabela I. O estado da máquina é alterado conforme os comandos dados pelo usuário através do teclado. Em cada estado, as variáveis do jogo são alteradas conforme explicado na Tabela I.

É preciso notar que o estado atual do jogador limita as possíveis ações que este pode realizar. Por exemplo, se este estiver em "Casa", não poderá "navegar na Internet" e terá que "ir para a faculdade" para depois escolher entre "fazer o trabalho", "ir para o bar" ou "navegar na Internet".

3.3 Módulo de Vídeo

Devido à restrição de memória do FPGA da placa UP1, de apenas 4kb, foi necessário buscar outra abordagem para armazenar e exibir o ambiente do jogo em uma resolução 640x480 pixels. A solução encontrada foi descrever toda a interface gráfica do jogo diretamente em código VHDL, fazendo assim a utilização de elementos lógicos em vez de espaço em memória para representar os gráficos do jogo.

O FPGA da placa UP1 possui pinos de saída ligados a um conector para vídeo do tipo VGA, como pode ser visto na Figura 2. Foi implementado um bloco de controle no qual é gerado o sinal de vídeo baseado nos dados apresentados pelo módulo de lógica.

Ao gerar o sinal de vídeo, é necessário gerar o sinal que posiciona o feixe de elétrons do monitor na tela em relação aos pulsos de sincronismo vertical e horizontal. Ao gerar esse sinal, sabe-se a posição do feixe de elétrons em cada instante, em coordenadas (X,Y), através de contadores que contam os pulsos (480 para cada linha e 640 para cada coluna). Com base nisso, foi possível desenhar os gráficos do jogo, fazendo-se uso da linguagem VHDL. Abaixo, é apresentado um exemplo de código VHDL para desenhar um quadrado vermelho com as coordenadas mostradas na Figura 4. Neste código, as variáveis *CINT* e *LINT* são os contadores de coluna e linha, respectivamente, e *Ra*, *Ga* e *Ba* são os valores de vermelho, verde e azul para o monitor colorir cada pixel. Não são fornecidos sinais intermediários de intensidade de cor, e *Ra*, *Ga* e *Ba* podem apenas ligar ou desligar cada cor, formando oito combinações possíveis.

```
IF ( (cint > 0) AND (cint < 421) AND
(lint > 0) AND (lint < 301) ) THEN
          Ra := '1';
          Ga := '0';
          Ba := '0';
END IF;
```

Para a construção dos textos dos menus do jogo foi armazenada a posição de cada caractere como constante, utilizando assim elementos lógicos ao invés de memória. Esses dados então são utilizados para buscar em uma memória do tipo ROM a representação do caractere e seu posicionamento na tela. Abaixo, um trecho onde o caractere 'D', representado pelo número quatro, está na coluna nove e linha 44.

```
CONSTANT letra232_c : INTEGER := 9;
CONSTANT letra232_l : INTEGER := 44;
CONSTANT letra232ch : INTEGER := 4;
```

Para a execução dessa extensa tarefa, foram implementados *scripts* na linguagem Ruby para auxiliar o processo. Esses *scripts* transformam representações textuais da interface em código VHDL, que posteriormente foram inseridas no código. Isso permitiu que fossem escritos ou gerados em torno de 4 mil linhas de código em linguagem VHDL para o vídeo.

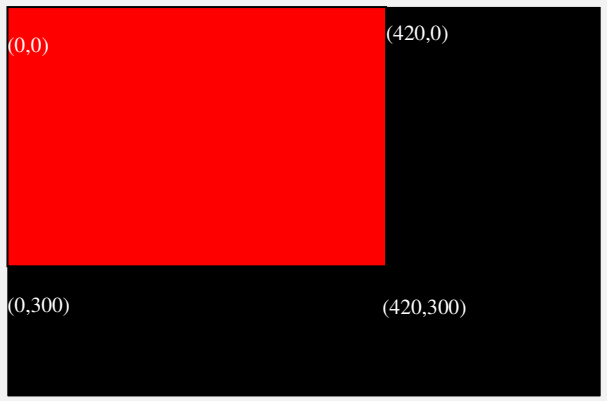


Figura 4: Representação do resultado ao executar.

O jogo é constituído de várias telas contendo a descrição dos ambientes (como *Casa*, *USP*, *Bar*, por exemplo) e uma lista dos possíveis lugares que o jogador pode ir a partir da posição atual. Também é apresentado uma lista das variáveis em forma de barras, indicando o valor dos parâmetros (*alimentação, higiene, descanso*, etc...). A Figura 5 apresenta uma dessas telas do jogo, representando a planta da casa do jogador. Quando o jogador se movimenta, novas telas são desenhadas, juntamente com suas novas opções e valores dos parâmetros.

3.4 Detalhes da Implementação

Cada módulo é representando por um ou mais blocos que foram implementados utilizando linguagem VHDL. A ligação entre os blocos foi feita através do editor de esquemático do QuartusII utilizando linhas, que representam os fios e os barramentos de interconexão e também *labels* demarcando algumas ligações. A Figura 6 mostra quais blocos foram utilizados para implementar cada módulo e as ligações entre eles.

O módulo de teclado, não demonstrado na Figura 6, envia os sinais já decodificados do teclado pelo barramento `key_s`, demarcado na figura em amarelo. O módulo de lógica (demarcado em azul), implementado pelo bloco `state_sim`, possui a implementação da máquina de estados do jogo em VHDL. O módulo de vídeo (demarcado em azul) é composto por 3 blocos: `VGA_SYNC`, que é responsável pela interface com o monitor; `lpm_rom0`, que é uma memória do tipo ROM e `decoder`, que é o responsável por receber os sinais do módulo de lógica e transformar as informações em representações gráficas e sinal de vídeo para o monitor.

4. Resultados

Um grupo de 50 alunos do terceiro e quarto ano do curso de Bacharelado em Ciência da Computação da Universidade de São Paulo foram convidados para avaliar o jogo, em um período de duas horas. Com base nos parâmetros especificados na Tabela I, somente um jogador conseguiu concluir o objetivo proposto (*terminar o trabalho*) e salvar seu avatar, como pode ser visto na Tabela II. Isso mostrou que a especificação inicial dos parâmetros do jogo, representados pelos valores numéricos apresentados na Tabela I, tornou muito difícil a conclusão do jogo.

Tabela II: Resultado do 1º teste com 50 alunos

	Nro de Jogadores	% Total
Vencedores	1	2%
Não concluíram	49	98%

Os valores dos parâmetros que modificam as variáveis *descanso, higiene, alimentação, trabalho, humor* e *ações* foram então modificados conforme a Tabela III apresentada abaixo, e um novo grupo (no qual não participaram nenhum dos jogadores do experimento anterior) foi convidado para avaliar o jogo. Os resultados da Tabela IV mostram que esta alteração tornou possível que 46% dos jogadores completassem o objetivo antes de duas horas ou do limite de 10 turnos, vencendo o jogo.

Tabela III: Alteração nas Mudanças provocadas por cada ação para o segundo experimento

Ação	Mudança	
Navegar na Internet	Descanso	-2
	Higiene	-1
	Alimentação	-1
	Ações	+3
"Atacar" a geladeira	Descanso	-1
	Higiene	-1
	Alimentação	+5
	Ações	+2
Ir para a Faculdade	Descanso	-2
	Higiene	-2
	Humor	-2
	Alimentação	-1
	Trabalho	+2
	Ações	+4
Ir para a Casa	Descanso	-2
	Higiene	-1
	Humor	+3
	Alimentação	-1
	Ações	+4
Fazer Trabalho	Descanso	-3
	Higiene	-1
	Humor	-4
	Alimentação	-3
	Trabalho	+4
	Ações	+4

Estes experimentos permitiram então a especificação de dois níveis de dificuldade para o jogo desenvolvido. Foi decidido, inclusive, permitir que jogadores mais experientes modifiquem diretamente o arquivo de parâmetros antes de reprogramar o jogo no FPGA.

Tabela IV: Resultado do 2º teste com 50 alunos

	Nro de Jogadores	% Total
Vencedores	23	46%
Não concluíram	27	54%

5. Conclusão

Este artigo apresentou a implementação em hardware de um jogo do tipo simulador de vida artificial. Diferentemente de outros trabalhos que apresentam a implementação de jogos de baixa complexidade, do tipo *arcade*, este artigo apresenta um jogo de maior complexidade, no qual o jogador pode realizar em cada turno até nove ações diferentes. Após extensa revisão bibliográfica, é possível concluir que este artigo traz pela primeira vez a descrição de um jogo deste tipo implementado totalmente em hardware, em linguagem VHDL, sendo executado juntamente com sua interface de teclado e vídeo em um único chip FPGA.

Este trabalho também contribui com a área de desenvolvimento de jogos para plataformas móveis como PDAs e telefones celulares ao apresentar soluções criativas para os problemas emergentes da utilização de um hardware de baixo custo, pouca memória e pequena velocidade de operação. Desta maneira, foi feito uso extensivo do paralelismo intrínseco de matrizes de células lógicas programáveis, como os FPGAS. Assim, vários módulos do jogo foram distribuídos no hardware e operam em paralelo no chip, realizando a interface com o usuário, a análise das ações escolhidas, o redesenho da tela gráfica e a geração dos sinais de vídeo para o monitor.

Também para compensar os poucos recursos de memória do hardware utilizado, a tela gráfica do jogo foi implementada na forma de um algoritmo em linguagem VHDL, ao invés de ser desenhada em memória. Assim, foi possível descrever uma tela de resolução 640 por 480 pixels em apenas 4 Kbytes de memória.

Outra contribuição diz respeito à produção de um hardware específico para o jogo, que inclui em um único chip a interface com o usuário e o módulo de geração dos sinais de vídeo para um monitor. Assim, foi produzido um jogo completo embarcado em um dispositivo eletrônico do tipo FPGA de baixo custo: cerca de U$ 1,00.

Agradecimentos

Os autores gostariam de agradecer o Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq, pelo apoio financeiro recebido durante este projeto.

Referências

BRADSHAW, L. , 2005. Postmortem: Avoiding Sequelitis in The Sims 2. Game Developer Magazine, January 2005. *Game Developer*, 40-44.

QUARTUS, 2004. *Quartus II Handbook, Design Implementation & Optimization*. Volume 2, Printed by Altera Corporation, San Jose, CA, 2004.

WATERMAN, S., 2005. *Lab Manual for Digital Electronics with VHDL (Quartus II Version)*. Publisher: Prentice Hall (May 27, 2005) ISBN-10: 0131715143, 400p.

WOLLINGER, T. , GUAJARDO, J., PAAR, C. 2004 Security on FPGAs: State-of-the-art implementations and attacks, ACM Transactions on Embedded Computing Systems (TECS), v.3, n.3, 534-574.

DEMARIA R., WILSON, J. 2003. *High Score!: The Illustrated History of Electronic Games*., Publisher: McGraw-Hill Osborne Media.

LIENHART, G., KUGEL, A., MANNER, R. 2002. Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations, 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02), p. 182.

CRABILL, E. 2005. FPGA-Based Video Games, Xcell Journal Online, July 11, 2005. Acessado em 10 de agosto de 2008. http://china.xilinx.com/publications/xcellonline/xcell_54/xc_pdf/xc_atari54.pdf.

BOULÉ, M. and ZILIC, Z. 2002. An FPGA Based Move Generator for the Game of Chess, IEEE Custom Integrated Circuit Conference 2002, 71-74.

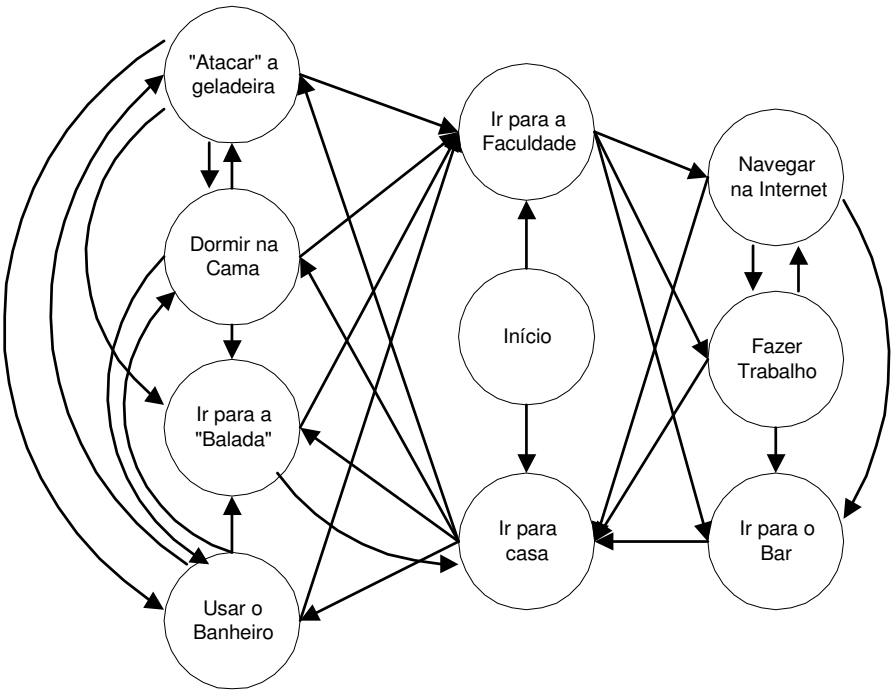


Figura 3: Diagrama representativo da máquina de estados que descreve o funcionamento do sistema e as possíveis ações que o jogador pode realizar a partir de cada estado.

Figura 5: Representação da tela do jogo, mostrando os ambientes e os locais que o jogador pode visitar de acordo com cada ação possível.

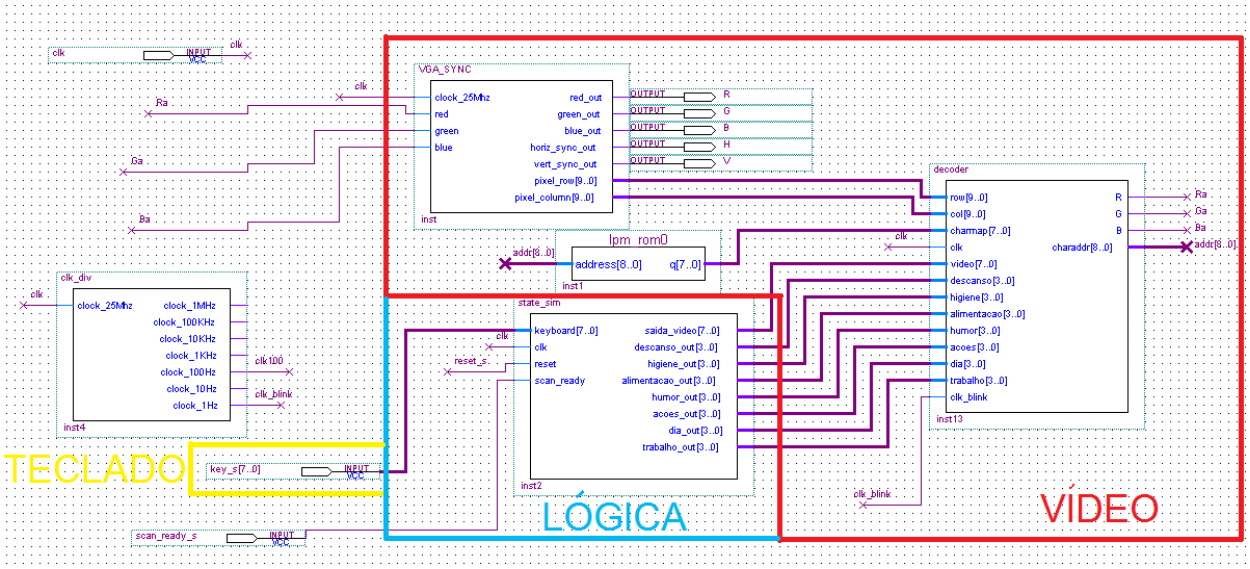


Figura 6: Representação dos blocos que implementam os módulos de vídeo e da lógica do jogo.