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.

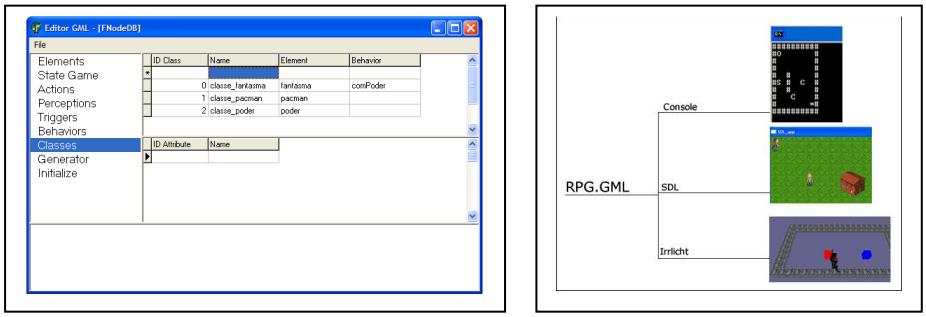

Figure 1: Projeto SDK Gameplay. (a) Editor de gameplay. (b) Visualização da portabilidade do formato GML.

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).

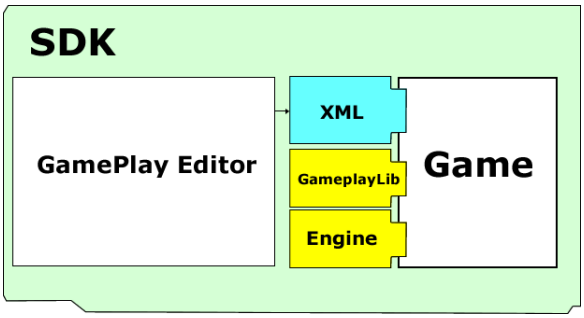

Figura 1 – Organização do SDK e comunicação do editor com o jogo; pode-se ver que o editor está exportando o arquivo XML para o jogo, que o utiliza em conjunto com a biblioteca *GameplayLib* e o *engine*.

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.

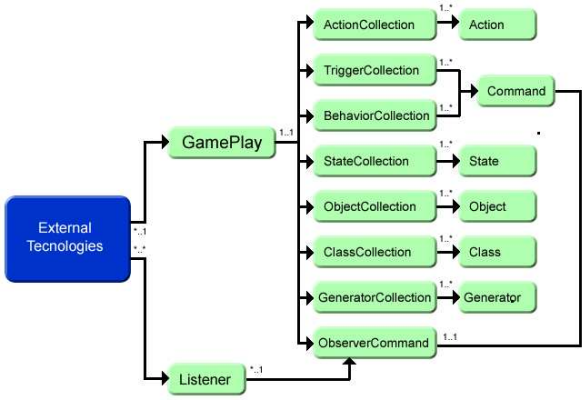

Figura 2 – Organização da biblioteca *GameplayLib*; a classe *gameplay* é a responsável pelo gerenciamento da biblioteca, podendo usar as classes conectadas a ele; *listener* é a classe que interage com as tecnologias externas.

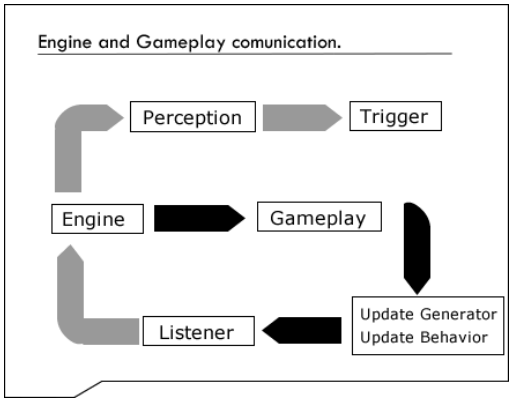

Figura 3 – Ciclo de comunicação do *engine* com a classe *gameplay*. Inicia com o *engine* ativando *gameplay*, que por sua vez atualiza o *generator* e o *behavior*, que por sua vez escuta os *commands* executados, gerando o *engine perceptions* de volta para o *trigger*.

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).

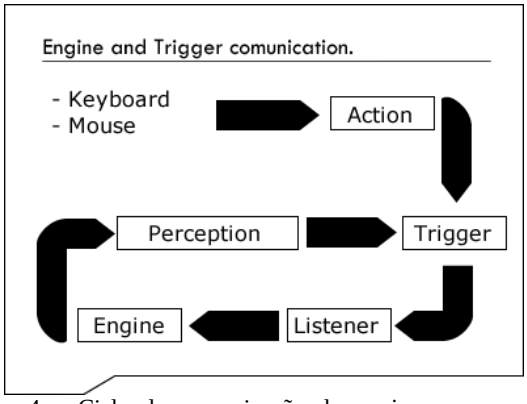

Figura 4 – Ciclo de comunicação do *engine* com o evento *trigger*. Inicia com a entrada de dados ativando uma *action*, sendo enviada para um *trigger*. O *engine*, por sua vez, escuta os *commands* executados gerando *perceptions*, que voltam para o *trigger*.

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.

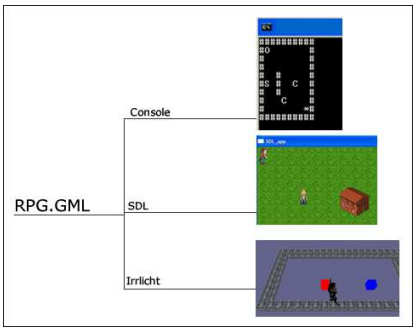

Figura 6 – Jogo de RPG de console em estado de navegação; um fator importante é que os três *engines* estão usando o mesmo arquivo GML.

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/>.