Capítulo

1

Motores para Criação de Jogos Digitais: Gráficos, Áudio, Interação, Rede, Inteligência Artificial e Física

João Ricardo Bittencourt[1,2] e Fernando S. Osório[1,3] - UNISINOS
[1]Curso de Desenvolvimento de Jogos e Entretenimento Digital
[2]Curso de Engenharia da Computação e [3]PPG em Computação Aplicada

Abstract

This tutorial aims to present an overview in the context of digital games development and more specifically the use of game engines. A general description of these technologies is presented, including some examples of implementations and applications. We present the different components usually included in the game engines, as graphics visualization modules, audio control, interaction devices control, network resources and multiplayer modules, physical simulation engine and Artificial Intelligence engine. So, we hope this text can be helpful to a novice and even for an initiated digital game developer, introducing someone more easily to this field and presenting a quite practical view of this theme.

Resumo

Este tutorial visa apresentar uma visão geral dentro do contexto do desenvolvimento de jogos digitais e mais especificamente do uso de motores de jogos (game engines) em destaque na atualidade. É apresentado um panorama geral destas tecnologias, complementando com exemplos de utilização e aplicações das mesmas, incluindo os diferentes componentes dos motores de jogos para a manipulação de gráficos, de áudio, de dispositivos de interação, para o uso de recursos de rede (jogos multiplayer), de simulação física e de Inteligência Artificial. Assim, espera-se que este texto sirva para iniciar o leitor com mais facilidade na área de projeto e desenvolvimento de jogos digitais, apresentando um panorama abrangente e bastante prático sobre este tema.

1.1. Introdução

A área de Desenvolvimento de Jogos e Entretenimento Digital tem apresentado um grande crescimento no exterior, e mais recentemente no Brasil. Um tema de grande importância é a criação, disponibilização e uso de ferramentas capazes de possibilitar um desenvolvimento mais rápido e de maior qualidade de jogos digitais. Os motores de jogos e ferramentas RAD (*Rapid Application Development*) são uma peça-chave no desenvolvimento destes, pois permitem adicionar recursos sofisticados aos jogos, tornando-os mais interessantes, com uma maior realismo, melhor jogabilidade, e por conseqüência com um maior apelo comercial e competividade junto ao mercado.

Um outro elemento importante a ser considerado é o *gap tecnológico* existente entre os grandes estúdios desenvolvedores de jogos do exterior, resultando na oferta de produtos de alta qualidade, se comparados com a maioria dos produtos desenvolvidos no país. É de grande importância que seja realizada uma discussão sobre como podemos diminuir esta distância entre os produtos nacionais e estrangeiros, o que certamente passa pelo estudo, desenvolvimento e acesso/disponibilização/compartilhamento de ferramentas e tecnologias de apoio ao desenvolvimento rápido de jogos.

Considera-se que este tema é de capital importância, em função do atual estágio de desenvolvimento em que se encontra a indústria de Jogos Digitais do país. Uma discussão como esta pode contribuir positivamente para que se busque um melhor desenvolvimento e desempenho da indústria de desenvolvimento de jogos digitais no país.

O objetivo deste tutorial é apresentar um painel sobre motores de jogos (*game engines*) atualmente empregados no desenvolvimento de jogos digitais. Será apresentado um apanhado sobre as diferentes soluções atualmente disponíveis, onde serão discutidas as soluções integradas (motores completos) específicas para o desenvolvimento de jogos digitais, bem como os diferentes módulos especializados em prover recursos específicos, ou seja, serão abordados os módulos dos motores para a parte gráfica 2D e 3D, de interface e interação com dispositivos de E/S, de áudio, de comunicação e jogos em rede, de Inteligência Artificial e de Física.

O foco será a discussão sobre os motores de jogos, considerando-se:

(i) os recursos oferecidos por estes;

(ii) a facilidade de uso e integração em diferentes tipos de jogos digitais;

(iii) disponibilidade vs. custo vs. Plataformas;

(iv) adequabilidade em relação ao mercado brasileiro de desenvolvedores de jogos;

(v) a importância de desenvolvimento de tecnologias nacionais.

1.1.1. Desenvolvimento de Jogos Digitais

A criação de um jogo envolve uma série de processos e ferramentas a fim de obter o produto final. A indústria de jogos tem crescido de forma muito rápida nos Estados Unidos e na Europa, e o Brasil não pode perder a oportunidade de acompanhar o crescimento desta área. Em função disto, precisamos acompanhar os avanços nesta área e desenvolver meios de fomentar a nossa produção de jogos digitais, o que certamente passa pelo estudo, desenvolvimento e utilização de motores de jogos que permitam acelerar a produção de novos títulos.

O desenvolvimento de um jogo é um processo complexo que envolve inicialmente a etapa de projeto, usualmente através da criação de um *Design Document*, onde serão especificados os componentes do jogo, suas funcionalidades e interfaces, entre outros elementos. Na etapa de projeto já será necessária a seleção de ferramentas de suporte ao desenvolvimento do jogo, que inclui:

- Ferramentas de gerência de projeto;
- Editores gráficos 2D (texturas, bitmaps, cenários 2D, etc);
- Modeladores 3D e ferramentas de animação de modelos 3D;
- Editores de efeitos de áudio e trilha sonora;
- Editores de níveis dos jogos (fases);
- Ferramentas de suporte ao desenvolvimento do jogo => MOTOR (*Engine*)!

O desenvolvedor de um jogo terá portanto uma tarefa-chave no projeto e seleção das ferramentas de suporte ao desenvolvimento, ligada a definição do motor de jogo que é uma peça fundamental para a implementação do jogo. O motor de jogo muitas vezes está diretamente relacionado as demais ferramentas de suporte, pois estas devem ser compatíveis com os modelos, formatos de arquivo e funcionalidades oferecidas por um determinado motor de jogo. Uma perfeita integração entre as diferentes ferramentas sempre será desejável, pois aumentará a produtividade da equipe desenvolvedora.

Portanto, o motor de jogo será o integrador de diferentes componentes, que vão dos gráficos 2D, modelos 3D e animações, áudio, interfaces com dispositivos de E/S até a parte referente aos recursos de rede. O motor de jogo irá permitir ao desenvolvedor um enorme ganho de tempo e reaproveitamento de código, pois as funcionalidades disponibilizadas por estes permitirão um desenvolvimento rápido de aplicações, sendo comumente considerado como uma ferramenta RAD – *Rapid Application Development*.

O desenvolvimento de jogos, seja este um jogo 2D, um jogo casual para Web ou um jogo 3D, pode ser acelerado de forma dramática através do uso adequado de um motor de desenvolvimento de jogos. Existem diferentes motores para jogos, onde podemos citar como exemplos o GameMaker[1] que é mais focado para o desenvolvimento de jogos 2D, o RPG Maker[2] focado no desenvolvimento de jogos RPG, o DarkBasic Pro[3] que serve para o desenvolvimento rápido de jogos 3D de nível médio, e ferramentas (*engines*) como o OGRE3D[4], Unreal Engine, CrystalSpace, Torque, TrueVision TV3D, entre outras focadas no desenvolvimento de jogos 3D profissionais.

Neste tutorial iremos abordar os motores de jogos, e seus diferentes módulos, de um modo mais geral, podendo ser aplicados no desenvolvimento de diferentes tipos de jogos. Entretanto, será dado um certo destaque maior ao desenvolvimento de jogos 3D baseados no uso das ferramentas que iremos descrever a seguir.

---

[1] GameMaker - http://www.gamemaker.nl/ (Ferramenta 2D - Curso de Jogos da Unisinos, 1o. semestre)

[2] RPG Maker XP - http://www.enterbrain.co.jp/tkool/RPG_XP/eng/

[3] DarkBasicPro- http://www.thegamecreators.com/ (RAD 3D- Curso de Jogos da Unisinos, 2o. semestre)

[4] Ogre3D - http://www.ogre3d.org/ (Engine 3D – Curso de jogos da Unisinos, 3o. semestre)

1.2. Arquiteturas de Motores de Jogos Digitais

Na seção 1.9 serão apresentados motores de jogos proprietários e livres, soluções que já podem ser adotadas em um projeto de jogo com poucas modificações. Entretanto em alguns projetos pode ser necessário a criação do próprio motor. Muitas empresas de entretenimento digital desenvolvem seus motores e são estes o principal diferencial tecnológico diante dos concorrentes. Por exemplo, a *Rockstar*, *developer* da série *Grand Thief Auto,* está desenvolvendo seu motor denominado *R.A.G.E* para criar títulos para XBox 360.

Independente da criação de um novo motor, adaptação de uma solução livre ou aquisição de uma solução proprietária um motor de jogo possui uma arquitetura de *software* capaz de interligar uma série de componentes e propiciar uma experiência gráfica interativa em tempo real para os usuários.

É importante ter o conhecimento e domínio de técnicas de Engenharia de *Software*, tais como, *frameworks*, padrões de projeto e componentização para projetar a arquiteturas de motores de jogos. Mesmo na aquisição de soluções proprietárias, tais como, *RenderWare* e o *Torque Game Engine* é necessário ter o conhecimento da estrutura de *software* para efetuar as devidas extensões usando os chamados *hot spots* disponíveis na solução. Nesta etapa de concepção é extremamente importante preocupar-se quanto de generalização pretende-se atribuir à arquitetura, tendo em vista que quanto maior o nível de abstração, maior será o impacto no desempenho da aplicação. Isto significa que projetar um motor de jogo que é capaz de gerar títulos de diferentes gêneros para inúmeras plataformas irá requerer uma arquitetura com muitas abstrações capazes de generalizar as diferentes especificidades dos gêneros e plataformas. Por outro lado, ao projetar um motor para executar sob uma plataforma específica e para um único gênero é possível explorar ao máximo os recursos do *hardware* criando uma solução com menor abstração, mas em compensação com alto desempenho. O que deve ser considerado é o fator relativo a portabilidade e se determinado jogo digital requer ou não alto desempenho. Um *casual game*, em geral, não irá requer um processamento igual ao *Unreal III*.

Existem muitas propostas de arquiteturas, entretanto serão apresentadas somente alguns exemplos de arquiteturas de motores que servem para ilustrar a estrutura de *software* destas soluções computacionais. Um modelo arquitetural bastante interessante foi proposto por Domingues [DOM 03] baseando-se em Hodorowicz [HOD 06] e Madeira [MAD 01] para o motor *Forge V8 3D*. Basicamente o *Forge V8 3D* é baseado no padrão de projeto arquitetural MVC (*Model-View-Controller*) proposto por Buschmann, Meunier, Rohnert et al [BUS 96]. Na Figura 1.1 está representado o *framework* desenvolvido por Domingues [DOM 03]. Destaca-se que na proposta de Domingues o enfoque é construir um motor de jogo para criação de jogos 3D.

O módulo *Modelo* trata dos modelos geométricos e dos atributos visuais dos objetos de uma cena e provê as funcionalidades para organização da cena, detecção de colisão e outras operações gráficas, tais como geradores de partículas. O módulo *Visão* é responsável pela geração das imagens. Possui classes responsáveis pelo volume de visão, projeção, câmera, área de visualização e o renderizador. O *Controlador* é responsável pela dinâmica do sistema. Trata-se dos gerenciadores responsáveis pela manipulação dos dados de entrada, simulação física, dados de entrada e saída da rede de comunicação e os

dados referentes aos algoritmos de Inteligência Artificial. Domingues não abordou os controladores de rede e de Inteligência Artificial. Neste módulo estão classes de sinais para comunicação entre objetos do modelo, temporizador, gerenciador de entrada, gerenciador de simulação física e o gerenciador de animação.

A camada de *Sistema* fornece duas abstrações, uma de *hardware* e outra de *software*. A abstração de *hardware* comunica-se com dispositivos físicos e com o sistema operacional, enquanto a abstração de *software* fornece alguns componentes fundamentais para criação dos jogos, tais como estruturas de dados e funções matemáticas.

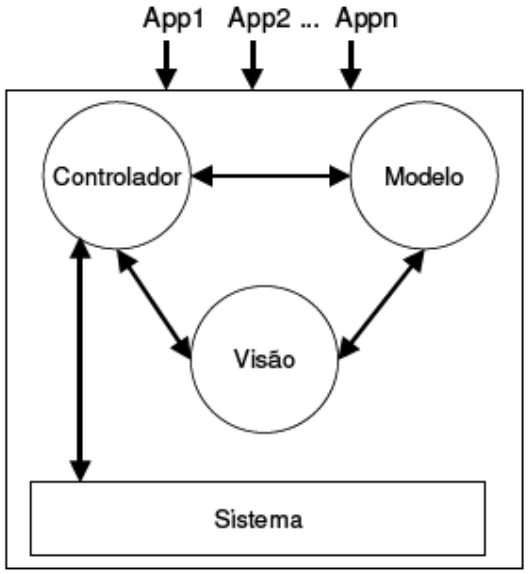


Figura 1.1. Estrutura MVC adotada pelo Forge V8 3D [DOM 03]

Bittencourt [BIT 04] define uma estrutura genérica de um motor de jogo com ênfase na modularização (ver a Figura 1.2). Basicamente um motor de jogo possui os seguintes componentes: *Interação, Comunicação, Controle* e *Visualização*. O componente de *Interação* relaciona-se com os periféricos de entrada, tais como teclado, *mouse, joystick*, caneta utilizada por *handhelds* e teclas de telefones celulares. Este componente é responsável pelo tratamento de eventos gerados por estes periféricos quando ocorre alguma interação do usuário com estes.

O componente de *Comunicação* permite a criação de jogos em rede com múltiplos participantes. Na maioria das aplicações utiliza-se um processo de comunicação por *sockets* e estes por sua vez são implementados de forma diferenciada em cada plataforma de execução. Desta forma existem *sockets* para plataforma de computadores pessoais, *handhelds* e telefones celulares. O componente de *Comunicação* e o de *Interação* são responsáveis por fornecer as entradas para o motor de jogo.

O componente de *Controle* mantém à lógica do jogo propriamente dita e manipula os objetos do jogo. Para desempenhar tais tarefas utiliza uma descrição do

jogo, que por sua vez utiliza uma série de recursos e entradas oriundas do componente de *Interação* e/ou do componente de *Comunicação*. Um motor de jogo pode conter inúmeros controladores cada um responsável por uma operação lógica, ou seja, podem existir controladores de simulações físicas e controladores de Inteligência Artificial, por exemplo. É importante destacar que estes controladores poderão ser reusados em diferentes jogos digitais.

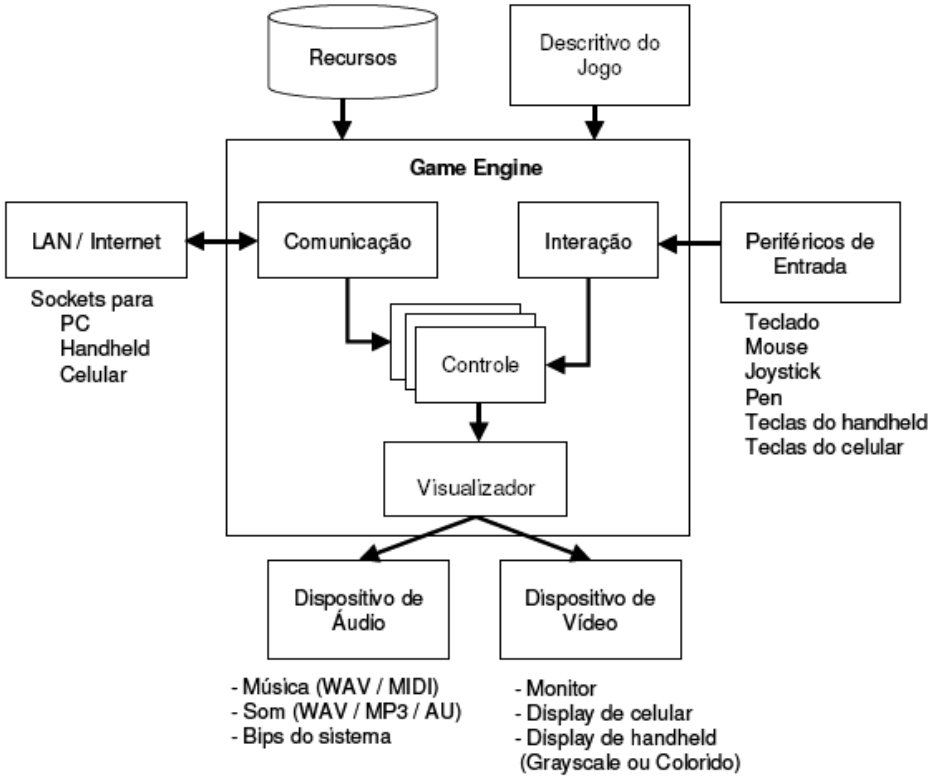


Figura 1.2. Arquitetura genérica de motor de jogo [BIT 04]

Após o processamento concluído pelos componentes de *Controle*, o *Visualizador* cria a imagem que representa o estado atual do jogo sob a perspectiva de determinada entidade controlada pelo jogador. O *Visualizador* utiliza um dispositivo de vídeo, tal como um monitor, displays dos *handhelds* ou telefones celulares para apresentar a imagem. Reprodução de sons, música ou simples *beeps* emitidos pelo sistema também podem ser utilizados para representar mudanças de estados dos objetos do jogo. É importante destacar a diversidade de cada um destes componentes, pois podem existir inúmeros tipos de estruturas de comunicação, periféricos de entrada, dispositivos de áudio e de vídeo. Esta arquitetura proposta propõe-se a ser genérica de forma que permita a integração destes componentes para criar um motor de jogo de forma independente de sua implementação, ou seja, o usuário pode interagir com o jogo através de um dispositivo com teclas, seja este um teclado de um computador pessoal ou as teclas de um telefone celular.

Dentro dos conceitos de Engenharia de Software um motor trata-se da parte do projeto de software que executa certas funcionalidades para um programa. Dentro da área de jogos, um motor de jogo se encarregará por lidar com o *hardware* gráfico, irá controlar os modelos para serem renderizados, tratará das entradas de dados do jogador, tratará de todo o processamento de baixo nível e outras tarefas que o desenvolvedor de jogos normalmente não deseja fazer ou não tem tempo para se preocupar. Existem inúmeras definições para um motor, entretanto, estas definições convergem em algumas características:

- Permitir que o desenvolvedor possa criar diversos jogos diferentes, usando um mesmo motor. É comum, entretanto, que os *engines* sejam catalogados de acordo com os tipos de jogos para os quais eles foram concebidos [EBE 00], por exemplo, motor de RTS, motor de RPG, e assim sucessivamente;
- Poder reaproveitar com facilidade o código desenvolvido em projetos anteriores;
- Abstrair a manipulação de APIs padronizadas, tal como OpenGL [OGL 06a, OGL 06b], mas em muitos casos, o desenvolvedor irá usar as próprias APIs dentro do ambiente do motor, para implementar funcionalidades específicas;

Sem dúvida, desenvolver um motor é uma tarefa complexa, repleta de desafios. O objetivo deste documento não é ensinar a implementar um motor, mas sim compreê-los e permitir a criação de motores baseando-se em soluções já existentes.

Em suma, pode-se afirmar que um motor é composto por diversos "sub-motores", sendo cada um responsável por tratar um tipo de problema envolvido em jogos. Os principais componentes são de interação, de visualização, de som, de interconexão em rede, de física e de Inteligência Artificial. Nas próximas subseções cada um desses componentes serão detalhados.

1.3. Dispositivos de Interação

A interação com jogos usualmente é feita usando os seguintes dispositivos de entrada [IDW 06]: *teclado*, *mouse* ou *joystick*. O ***teclado*** permite a entrada de informação textual, teclas de comandos e teclas de controle de movimentação (e.g. uso das setas), sendo que a leitura do teclado pode permitir: leitura de caixa de texto, leitura de teclas (com pausa), leitura de teclas sem pausa (modo contínuo – usado em um laço onde verifica-se se tem ou não uma tecla pressionada). Além disto, as rotinas de acesso ao teclado devem permitir a leitura das teclas lendo o seu código ASCII, assim como as teclas especiais (Alt, Shift, Ctrl), sendo importante o aceso as informações de mais baixo nível (scancodes), de modo a permitir combinações de diferentes teclas (e.g. Alt+Shift+Z). As implementações dos módulos de interação de um motor, usualmente oferecem interfaces de tratamento de entrada de dados através de chamadas de rotinas de acesso direto (leitura) aos dados de entrada, ou através de rotinas implementadas para tratar eventos gerados pelos dispositivos. Quando o tratamento de entrada de dados é feito por eventos, o usual é que a rotina de tratamento de eventos crie uma fila de dados de entrada, de comandos e/ou requisições. Algumas funcionalidades e teclas do teclado podem ser emuladas através do uso de um *gamepad* digital (vide Figura 1.3(a)).

(a) Gamepad (b) Mouse (c) Teclado (d) Joystick (e) Direção

Figura 1.3. Dispositivos de Entrada

O ***mouse*** é um dispositivo de entrada que provê coordenadas X e Y absolutas (usualmente referenciadas em termos de coordenadas de tela), ou também, uma indicação relativa de deslocamento Dx, Dy da posição atual do cursor. Além da leitura de posição absoluta e/ou relativa, o mouse também permite a leitura de 1 a 3 botões e opcionalmente inclui ainda um botão de rolamento, indicando um valor Z ou um deslocamento Dz. As funções de uso do mouse devem permitir: ajustar suas coordenadas atuais, ler os valores de X, Y e Z (ou sua variação Dx, Dy e Dz) e ler o status dos botões. O ***joystick*** (Fig. 1.3(d)) fornece um conjunto de entradas analógicas, assim como o mouse, indicando a inclinação do bastão de controle (valores X, Y dentro de uma faixa de [–Xini;-Yini] à [+Xfim,+Yfim]), além de todo um conjunto de botões complementares que podem controlar disparos (entradas digitais, como no gamepad digital), e até mesmo entradas analógicas complementares (e.g. aceleração). A direção (*steering wheel*) pode atuar exatamente como um joystick, onde o giro da direção (ângulo) é mapeado para um valor X (ou Y) dentro de uma determinada faixa pré-estabelecida. A direção pode ter outras entradas analógicas complementares, como o pedal e/ou controle de aceleração e freio (Y, Z), e também entradas digitais com diferentes botões de controle. Usualmente consideramos os dispositivos como as setas do teclado, o mouse e o joystick como sendo dispositivos de entrada de coordenadas no plano XY (*pointing devices*). Interações em 3D irão exigir do desenvolvedor o mapeamento dos comandos XY e a inclusão de controles adicionais de modo a explorar e navegar no espaço tridimensional (controle de *roll*, *pitch* e *yaw*).

Do ponto de vista do controle dos dispositivos de interação, podemos destacar que temos dois tipos principais de interação: dispositivos de entrada digital, orientados a botões (gamepad, teclado, teclas do mouse ou do joystick) e dispositivos de entrada analógica, orientados a valores e coordenadas (mouse, joystick ou direção). Um motor de jogos deve prover recursos que facilitem a gerência e leitura destes dispositivos de entrada. Note que um bom gerenciador deve inclusive prover ao usuário uma certa transparência/abstração, onde o jogador poderá configurar se ele deseja redirecionar a leitura da entrada para o teclado, mouse ou joystick, inclusive reconfigurando as teclas e controles usados para jogar. Os motores atuais usualmente não chegam a oferecer este tipo de abstração, oferecendo apenas rotinas de acesso direto e específico para cada um dos três dispositivo a seguir: teclado/gamepad digital, mouse e joystick (joystick convencional, manche, direção ou gamepad digital).

É importante ainda destacar que certos dispositivos, como o joystick podem ainda ter um efeito de retorno (*force feedback*), o que torna estes dispositivos ainda mais específicos. O resultado destas especificidades é que atualmente muitos jogos são

programados de modo a oferecer (ou não) acesso a cada um destes recursos específicos, onde muitas vezes inclusive podem ser incompatíveis ou não oferecer suporte a determinados tipos de interface de entrada.

Em relação a gerência dos dispositivos de entrada, alguns motores incluem facilidades para este fim, como por exemplo a biblioteca DirectInput (DLLs) do DirectX da Microsoft [DXM 06], o GLUT (OpenGL Utility toolkit) [GLT 06] e o SDL_Input que faz parte da biblioteca SDL (descrita mais em detalhes na próxima sessão). Os motores mais completos como o DarkBasic, 3D Game Studio, Torque, TrueVision3D, Crystal Space, Ogre, entre outros [DMN 06], fornecem um conjunto de rotinas de acesso a diferentes dispositivos, o que facilita a tarefa de desenvolvimento de interfaces com diferentes dispositivos de entrada. A seguir apresentamos um exemplo de código de acesso ao mouse e teclado usando a SDL (Fig. 1.4).

```
#include "SDL.h"

int SDL_PollEvent(SDL_Event *event);


SDL_Event event;
.
.
.
while(SDL_PollEvent(&event))
    {
     switch(event.type)
     {
        case SDL_KEYDOWN:
         out<<"Tecla pressionada!\n";
        break;
        case SDL_MOUSEMOTION:
         out<<"Deslocamento do mouse\n";
        break;
        case SDL_QUIT:
            i=-1;
        break;
        default:
         out<<"Não sei que evento foi este!\n";
      }
    }

/*
Nota: A SDL deve ser previamente inicializada bem como alguns
dos dispositivos a serem usados... (ver sessão 1.4)
SDL_Init(SDL_INIT_VIDEO);
SDL_InitSubSystem(SDL_INIT_JOYSTICK);
*/
```

Figura 1.4. Código de exemplo da SDL Input

Em relação aos dispositivos de interação, um ponto interessante a ser discutido é em relação aos novos e diferentes tipos de dispositivos que vem sendo introduzidos no mercado de jogos, onde podemos destacar [IDW 06]: o wiimote (controle do Wii da Nintendo), as luvas (*data-gloves*), o tapete de interação (*power pad*), o uso de câmera

digitais (*webcam*, como no *EyeToy da Sony*), dispositivos 3D com 6DOF (*SpaceBall* – 6 *Degrees-of-Freedom*), dispositivos hápticos e até mesmo o uso de GPS (localizador espacial) para jogos em ambientes de AR (*Augmented Reality*). Espera-se que em breve novas funcionalidades sejam incluídas nos motores de jogos de modo a poder obter informações de modo simples e funcional entradas de movimento relativo em 3D (wiimote, data-gloves, spaceball), posições absolutas em 2D e/ou 3D (sensores do tipo flocks, GPS, câmeras), e até mesmo o uso de recursos mais avançados de retorno de força ao usuário (dispositivos hápticos).

1.4. Visualização 2D/3D

O módulo de visualização, também conhecido pelo neologismo renderizador oriundo da palavra inglesa *renderer* certamente é um dos módulos que exerce maior fascinação dos desenvolvedores, pois trata da síntese das imagens do jogo digital em tempo real. É responsável pela representação visual em um dispositivo de vídeo dos dados pertinentes à logica do jogo que estão alocados em memória. Devido a importância do atrativo visual para os usuários, cada vez mais os consoles da nova geração incrementam seu poder de processamento de forma a criar imagens cada vez mais realísticas.

Se as imagens são criadas antes da execução do jogo e somente são usadas para criar uma composição de cena em tempo real é dito que trata-se de um visualizador 2D. Jogos clássicos da década de 80/90, tais como, *Sonic: The Hedgehog*, *Super Mario Bros*, *Megaman*, entre inúmeros outros títulos utilizavam esta abordagem devido a precariedade do hardware da época que oferecia baixo processamento e pouca memória. Atualmente os jogos 2D são bastante comuns em telefones celulares e jogos para *web*, em função destas plataformas requererem aplicações simplificadas.

Por outro lado quando as imagens são sintetizadas em tempo real baseando-se em modelos geométricos que descrevem entidades tridimensionais, trata-se então de um visualizador 3D. Muitos filmes de animação ou efeitos especiais utilizam técnicas de sintetise de imagens que possuem um alto custo computacional sendo necessários diversos dias para criação de uma seqüência de animação. No caso dos jogos digitais, as imagens devem ser produzidas em tempo real, em uma taxa mínima ideal de 24 quadros por segundo. Logo, muitas técnicas realísticas não podem ser usadas em função de seu alto custo computacional. Por exemplo, um dos primeiros jogos 3D, *Virtua Fighter*, trata-se de um jogo 3D com modelos com poucos polígonos, nenhuma textura e poucas animações. Basicamente após a massificação das placas aceleradoras 3D inúmeros títulos passaram a ser produzidos com excelente qualidade gráfica, extremamente próxima da realidade. A promessa da próxima geração de consoles, XBox 360 e Playstation III, é produzir imagens realísticas em tempo real assemelhando-se a um filme.

Por tais razões a construção deste módulo é extremamente importante no motor de jogo. Serão apresentadas duas alternativas para comunidade, uma com enfoque aos jogos 2D e outra para criação de jogos 3D. Ambas são soluções livres e gratuitas que podem ser utilizadas inclusive em projetos de jogos comerciais com licença proprietária. Esta flexibilidade é um fator positivo principalmente para indústria nacional de entretenimento digital.

A primeira proposta trata-se da SDL (*Simple DirectMedia Layer*) [SDL 06c], uma biblioteca livre multiplataforma que permite acesso ao hardware de áudio, teclado,

mouse, joystick, hardware 3D via OpenGL [OGL 06a] e *frame buffer* 2D. Assemelha-se ao DirectX da Microsoft [DXM 06] entretanto possui as vantagens de ser multiplataforma, livre e muito simples de ser utilizada. A SDL executa sob Linux, Windows, WindowsCE, BeOS, MacOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX e QNX. Além de oferecer suporte não-oficial à outros sistemas, tais como, SymbianOS, OS/2 e Dreamcast. Originalmente foi criada em C, mas é integrável com C++ e pode ser usada através de bindings com diversas linguagens de programação – Java, Python, Lua, Pascal, PHP, entre outras.

Para a criação de jogos com visualização 2D é importante saber que partes de imagens predefinidas serão copiadas e combinadas no *buffer* de vídeo para fazer a composição da cena. Os *tiles* são bastante conhecidos nesta abordagem. Tratam-se de pequenas porções de imagens com um determinado padrão que podem ser desenhadas lado a lado formando uma imagem mais complexa.

O uso básico da SDL consiste em inicializar a biblioteca e criar uma superfície de vídeo que consiste da tela principal. Este superfície é denominada *surface*. Quando é criada define-se uma resolução, profundidade de cor (*deep color*), se será criada em *hardware* (placa aceladora de vídeo) ou na memória RAM do sistema e se irá dar suporte as uso de primitivas OpenGL. Todas as imagens podem ser carregadas utilizando uma biblioteca de apoio à SDL denominada *SDL_image* que permite carregar imagens em diferentes formatos, tais como, BMP, PNM, XPM, LBM, PCX, GIF, JPEG, PNG, TGA e TIFF. Cada imagem também é carregada em uma *surface*.

O processo de composição da cena vai constituir na cópia de porções das *surfaces* referentes as imagens na *surface* da tela principal. Esta operação é denominada de *blit*. Quando a cena estiver pronta no buffer esta é liberada para ser enviada para o dispositivo de visualização através de uma operação de *flip*. O código na Figura 1.5 é um exemplo simplificado de tais operações.

```
//Inicializar a SDL
if(SDL_Init(SDL_INIT_VIDEO)<0){
  cout << "Problemas no SDL_Init(): " << SDL_GetError();
  return 1;
}

/* Cria uma superfície que será usada como framebuffer.
   A superfície terá as dimensões 800x600, deep color de 32 bits
   e deve ser criada na placa aceleradora
*/
SDL_Surface *screen;
screen = SDL_SetVideoMode(800,600,32,SDL_HWSURFACE);

//Se foi possivel criar a tela entao
if(screen != NULL){
      //Carrega a imagem ovni.png na superfície ovni
      SDL_Surface *ovni;
      ovni = IMG_Load("ovni.png");
      //Cria as informacoes para posicionar o ovni na tela
      SDL_Rect dest;
      dest.x = 35; dest.y = 100;
      dest.w = 81; dest.h = 52;
```

```
        //Copia todo o ovni para tela considerando o 'dest'
        SDL_BlitSurface(ovni,NULL,screen,&dest);
        //Por ultimo 'libera' a cena para o dispositivo grafico
        SDL_Flip(screen);
}
else{
  cout << "Problemas no modo de video: " << SDL_GetError();
  return 1;
}
//Encerra a SDL
SDL_Quit();
```

Figura 1.5. Código de exemplo da SDL

A outra proposta referente à visualização 3D trata-se do motor gráfico Ogre3D[OGR 06]. O *Ogre3D* (*Object-Oriented Graphics Rendering Engine*) trata-se de uma ferramenta construída em C++, orientada a objetos e distribuído livremente e gratuitamente sob a licença GNU LGPL. Foi desenvolvido como a dissertação de mestrado de Jeff Plummer na Universidade do Arizona/EUA. Destaca-se que o *Ogre3D* não se trata de um motor de jogo e não foi projetado somente para esta funcionalidade. Constitui-se de um conjunto de componentes para renderização 3D que podem ser usados na criação do módulo gráfico de um motor de jogo.

O *Ogre3D* oferece ao desenvolvedor possibilidades de personalização e extensão, desta forma potencializando a reusabilidade de código, abstraindo primitivas de baixo nível (*OpenGL/DirectX*) permitindo o desenvolvimento de novas técnicas que poderão ser "plugadas" na arquitetura dos motores. O *Ogre3D* oferece suporte ao *DirectX* e a *OpenGL*, *shaders*, suporte a texturas, além de uma série de outras funcionalidade de suporte para criação de jogos 3D. É importante destacar que o *Ogre3D* é bastante documentado inclusive com a especificação UML (*Unified Modeling Language*) dos seus componentes. Oferece um wiki com muitos tutoriais e um fórum bastante ativo com desenvolvedores de todo mundo.

Este motor gráfico fundamenta-se no uso de um grafo de cena, ou seja, modelos 3D, fontes de luz, câmera, geradores de partículas, elementos 2D, matrizes de transformação entre outros objetos são organizados em uma estrutura hierárquica em geral na estrutura de uma árvore. Quando a imagem vai ser sintetizada inicia-se o processo pelo nodo raíz e continua-se o processo para os nodos filhos. Cada nodo quando é visitado ativa um contexto que irá servir para os seus filhos, por exemplo, se um nodo é uma matriz de translação, esta será aplicada em todos os nodos filhos. O Ogre3D não permite utilizar nenhuma primitiva diretamente, tais como, linhas, pontos e polígonos, nem usar comandos OpenGL/DirectX. Primeiramente isto garante a portabilidade da solução e em segundo lugar facilita o processo de renderização.

O funcionamento do Ogre3D basicamente fundamenta-se em um sistema orientado a eventos. O tratamento de teclas e mouse são efetuados através de *listeners* que são registrados no nodo raíz. Além destes, sempre que um frame vai ser renderizado é lançado um evento denominado *FrameEvent*. Para a criação da cena são adicionados objetos em um grafo de cena. Também é importante destacar que os modelos 3D são carregados através de um formato próprio com a extensão *.mesh*. Este arquivo é gerado

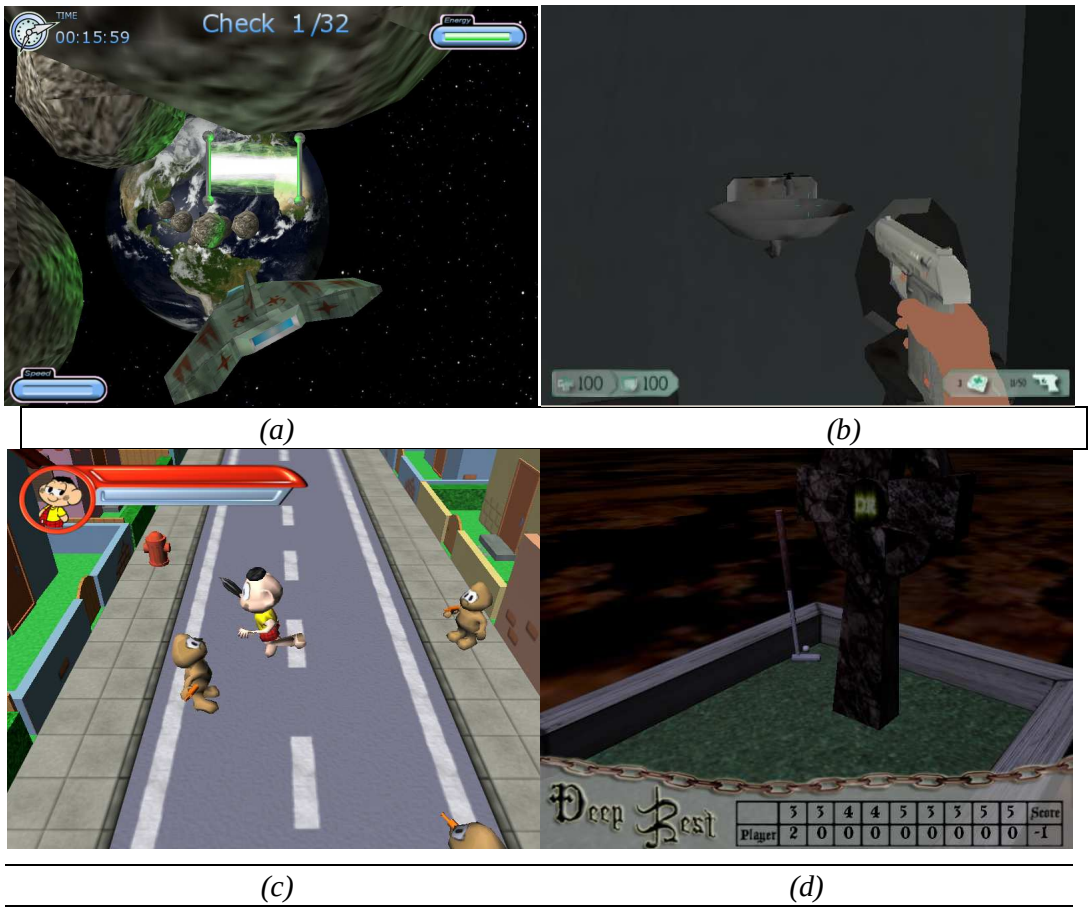
através de uma especificação em XML. Isto facilita a integração do Ogre3D com diferentes ferramentas, desta forma existem inúmeros exportadores para 3DS, Maya, Blender, Wing3D, entre outros. Na Figura 1.6 são apresentados alguns trechos de código que ilustram o uso elementar do Ogre3D.

```
class ExemploListener: public KeyListener{
//...
    bool frameStarted(const FrameEvent& evt){
        //Faca alguma coisa, modifica camera,
        //movimenta objetos. Basicamente eh o game loop
        return true;
    }
//...
};

class ExApplication{
//...
//...
    void run(){
        //Cria o objeto raíz do grafo de cena
        Root *root = new Root();

        //Configura o driver de video usando um arquivo
        //config.txt
        if(root->restoreConfig()){
            //Se for recuperado, inicializa uma janela
            Window *window = root->initialise(true);

            //Carrega todos os recursos que serao usados na cena.
            //Por exemplo, modelos 3D.
            ResourceGroupManager man;
            man = ResourceGroupManager::getSingleton();
            man.initialiseAllResourceGroups();

            //Define-se um gerenciador de cena (grafo de cena)
            SceneManager *manager;
            manager = root->getSceneManager(ST_GENERIC);
            //Define a luz ambiente como branca
            manager->setAmbientLight(ColourValue(1.0f,0.0f,1.0f));

            //Cria-se um cubo usando o modelo Cube.mesh
            //O modelo foi criado no Blender
            Entity *ent = manager->createEntity("Cube","Cube.mesh");
            //Ativa o uso das normais
            ent->setNormaliseNormals(true);
            SceneNode *nodo = root->createChildSceneNode("CubeNode");
            nodo->attachObject(ent);

            //Eh feito o registro do listener
            root->addFrameListener(new ExemploListener());

            //O root é responsável pelo rendering
            root->startRendering();
        }
    }
};
```

```
void main(){
    //Cria um objeto aplicacao
    ExApplication app;

    //Solicita para este objeto executar
    app.run();
}
```

Figura 1.6. Código de exemplo do Ogre3D

Quanto ao uso do *Ogre3D* no âmbito acadêmico destaca-se a experiência da Universidade do Vale do Rio dos Sinos (UNISINOS) no curso de graduação tecnológica de Desenvolvimento de Jogos e Entretenimento Digital. No terceiro semestre do curso é oferecida uma atividade acadêmica denominada *Projeto de Jogos: Motores de Jogos*. Além das demais bibliotecas que estão sendo apresentadas neste material os estudantes basearam seus projetos no Ogre3D. As Figuras 1.7(a-d) ilustram alguns projetos desenvolvidos pelos estudantes do curso. Para maiores informações sobre a experiência prática que integra tais ferramentas livres consulte o site *LudensWiki* [LUD 06].



(a) *(b)*

(c) *(d)*

Figura 1.7. *Screenshots* de jogos produzidos com o Ogre3D. *Space Rally* (a) produzido por Cristiano Bartel e Márcio Zani; *Live Is Not Enough* (L.I.N.E) (b) produzido por Maicon Filippsen e Paulo de Araújo; (c) Sujolândia criado por Márcio Daniel da Rosa; e (d) *Deep Rest* criado por Franco Sebben, Marcio Furtado e Bruno Larentis.

1.5. Áudio

Este componente do motor permite controlar arquivos de som da biblioteca de recursos do jogo. Normalmente utiliza alguma API adequada para manipular este tipo de arquivos, tal como o *DirectSound* [DXM 06], *OpenAL* [OPE 06] ou *SDL*. Além de permitir abrir e tocar estes arquivos, os motores de som podem ser projetados para sintetizar o som tridimensionalmente, ou seja, permitir que objetos da cena emitam sons e estes se comportem conforme o posicionamento do objeto na cena.

Neste documento serão destacadas duas soluções para manipulação de sons: *SDL_mixer* [SDL 06a] e a *OpenAL* [OPE 06]. A primeira proposta trata-se somente da reprodução de arquivos de áudio e a segunda permite sintetização de som 3D.

A própria SDL dá suporte básico ao *hardware* de som, entretanto utilizar a *SDL_mixer* facilita o trabalho do desenvolvedor. Trata-se de uma biblioteca gratuita e distribuída sob a licença GNU LGPL, multiplataforma que permite a execução de sons nos formatos WAV, MIDI, MOD, MP3 e Vorbis Ogg como música de fundo e execução nos formatos WAV, AIFF, RIFF, Vorbis Ogg e VOC como os demais efeitos sonoros. A *SDL_mixer* permite que seja executada uma música de *background* e pode reproduzir vários sons simultaneamente em diferentes canais de áudio. Também permite aplicar efeitos especiais nos sons. É bastante simples utilizar a SDL_mixer, basicamente deve-se inicializar o módulo de áudio da SDL, a API SDL_mixer, carregar os sons e reproduzi-los. Na Figura 1.8 são apresentados um trecho de código que ilustra o uso geral desta biblioteca.

```
if(SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO|SDL_INIT_NOPARACHUTE)<0){
      exit(1);
}
else{ //Inicializa o som com esta qualidade de audio
      if(Mix_OpenAudio(44100,MIX_DEFAULT_FORMAT,2,1024)<0){
            exit(1);
      }
      else{
          //Carrega trilha sonora e um som
          Mix_Music *trilha = Mix_LoadMUS("damage.mp3");
          Mix_Chunk *shotgun = Mix_LoadWAV("shotgun.wav");

          //Cria a "mesa de sintetizacao" com 2 canais
          Mix_AllocateChannels(2);

          //Reproduz em loop a trilha sonora
          Mix_PlayMusic(trilha,-1);

          //Toca uma vez o som no canal 0
          Mix_PlayChannel(0,shotgun,0);
      }
      //Encerra o modulo de audio
      Mix_CloseAudio();

      //Encerra a SDL
      SDL_Quit();
}
```

Figura 1.8. Código de exemplo da SDL_mixer

A OpenAL segue o formato de sucesso da OpenGL. Trata-se de uma biblioteca multiplataforma, gratuita e distribuída sob a licença GNU LGPL para a sintetização de áudio 3D. Títulos de sucesso, tais como Doom III, Battlefield II e Lineage 2, utilizam a OpenAL. A biblioteca suporta um conjunto de primitivas básicas para manipulação de áudio 3D. Basicamente para desenvolver som tridimensional define-se *buffers* que irão conter os sons em geral na forma de *stream*. Os arquivos de som devem ser salvos no formato mono. Estes buffers são associados às fontes (*sources*) que estão localizadas em algum lugar do espaço e estão emitindo ondas sonoras em uma direção ou em um determinado raio. Mais de um buffer pode ser associado à uma fonte sonora, os sons serão combinados. Além das fontes sonoras deve-se definir um ouvinte (*listener*) que também está localizado no espaço e voltado para uma determinada direção. O som será sintetizado considerando estes dados do ouvinte em relação as fontes sonoras. Em suma é bastante simples desenvolver aplicações com áudio 3D. Na Figura 1.9 é apresentado um trecho de código que ilustra o uso básico da OpenAL

```
//Abra o dispositivo de audio
ALCdevice *device = alcOpenDevice(NULL);
if(device){
      //Cria um contexto de audio e torna-o o atual
      ALCcontext *context = alcCreateContext(device, NULL);
      alcMakeContextCurrent(context);

      //Cria um buffer para conter o arquivo wav
      ALuint buffer;
      alGenBuffers(1,&buffer);

      //Aloca a wave stream no buffer
      buffer = alutCreateBufferFromFile("sfx.wav");

      //Cria a fonta sonora
      Aluint source;
      alGenSources(1,&source);

      //Aumenta o volume e posiciona no ponto (0,10,0)
      alSourcef(source, AL_GAIN, 1.0f);
      alSourcef(source,AL_POSITIION,0.0f,10.0f,0.0f);

      //Atribui o buffer na fonte
      alSourcei(source,AL_BUFFER,buffer);

      //Posiciona o ouvinte no ponto (0,5,0) direcionado
      //para a fonte sonora
      ALfloat *pos;
      pos[0] = 0.0f; pos[1] = 5.0f; pos[2] = 0.0f;
      ALfloat *dir;
      pos[0] = 0.0f; pos[1] = 10.0f; pos[2] = 0.0f;
      alListernerf(AL_POSITION,pos);
      alListernerf(AL_ORIENTATION,dir);

      //Inicia a reproducao do som em loop
      alSourcei(source,AL_LOOPING,AL_TRUE);
      alSourcePlay(source);
}
```

Figura 1.9. Código de exemplo da OpenAL

1.6. Inteligência Artificial (I.A.)

Os motores de jogos usualmente necessitam oferecer funcionalidades relacionadas ao uso de técnicas de Inteligência Artificial (I.A.), onde existe um grande número de possíveis implementações que podem ser disponibilizadas aos desenvolvedores. Em função deste tutorial ser amplo e abordar a parte de I.A. juntamente com os outros diversos módulos do motor, focaremos nossa discussão em alguns tópicos mais específicos, sendo relacionados principalmente as seguintes técnicas: I.A. clássica para jogos de tabuleiro, I.A. aplicada na busca de caminhos (*path finding*) e I.A. para o controle de personagens autonomos (agentes) tipo NPCs (non-player characters). Para aqueles que estiverem interessados em se aprofundar na aplicação e uso de técnicas de Inteligência Artificial em jogos, sugerimos a leitura das seguintes obras AI Game Programming Wisdom[RAB 02, 03], AI Game Engine Programming[SCH 04], AI: A modern approach[RUS 95], Fly3D Engine[WAT 01], AI4Games[FUN 06] e a série Game Programming Gems[RIV 06].

A ***I.A. clássica*** vem sendo aplicada em jogos de tabuleiro desde o início das pesquisas nesta área, pois estes jogos servem como um excelente desafio para a reprodução da Inteligência em máquinas. Os jogos de tabuleiro típicos são: jogo de Xadrez, jogo de Damas, jogo da Velha (*Tic-Tac-Toe*), Othello (ou Reversi), Go, Gamão, entre outros. Estes jogos apresentam um grande desafio, pois usualmente existe um grande número de jogadas possíveis a explorar, e deste modo devemos a empregar heurísticas que permitam uma redução do espaço de configurações possíveis e busca das melhores jogadas a fazer (aquelas que levam a uma vitória). A Figura 1.10 apresenta um esquema (parcial) das possíveis jogadas em um jogo da velha, onde devemos aplicar técnicas que conduzam o jogo (e o oponente) em direção a configurações do tabuleiro que sejam favoráveis em relação ao oponente.

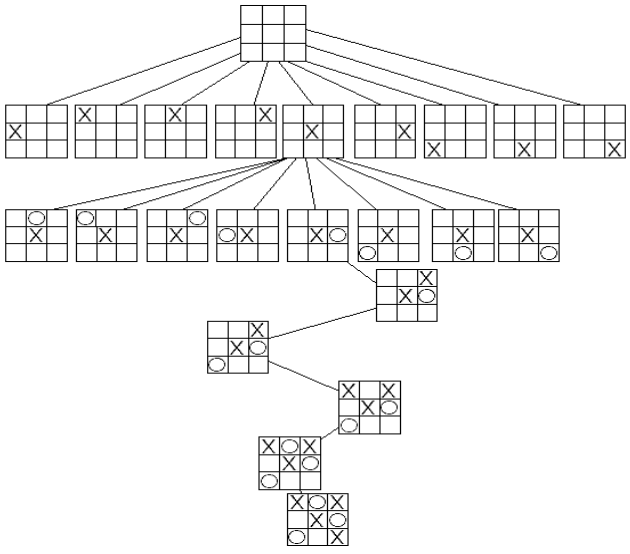


Figura 1.10. Árvore de estados (configurações) do tabuleiro no Jogo da Velha

No caso do jogo da velha (ou xadrez), o que fazemos é definir um valor em relação à situação do jogo nas folhas da árvore formada por todas as possíveis jogadas (ou pelas ‘n’ jogadas a partir de uma certa configuração). Baseado na avaliação da situação nas folhas desta árvore (ganhou, perdeu, empatou), aplicamos um algoritmo

denominado Mini-Max [RUS 95] que irá permitir a escolha das jogadas que mais possivelmente nos levarão a vitória, mesmo tendo como oponente um jogador cuja ação não pode ser totalmente prevista. O **algoritmo Mini-Max** [RUS 95] é conhecido como uma heurística de busca condicionada em espaço de estados, e no caso do jogo da velha garante com 100% de certeza um empate ou uma vitória.

Em jogos de ação, seja em um jogo como o Pac-Man ou como o DOOM, usualmente precisamos estabelecer trajetórias para guiar os NPCs (*non-player characters*). Um NPC pode ser um agente sem inteligência e que realiza tarefas simples e repetitivas, porém, em um jogo o interessante é a competição contra NPCs dotados de algum tipo de inteligência. Por exemplo, os fantasmas do Pac-Man podem usar algoritmos (heurísticas) bastantes simples para se movimentar em direção ao jogador, entretanto, também é possível usar algoritmos de I.A. para traçar uma rota entre a posição do fantasma e o Pac-Man. Os **algoritmos de *path-finding*** (traçado de rotas e trajetórias) tem sido intensivamente investigados na I.A. e principalmente em áreas como a robótica autônoma. O algoritmo mais usado em jogos é o **A Star** (A* - A estrela) [RUS 95], que através de uma heurística permite identificar o melhor trajeto entre dois pontos. Usualmente o ambiente é descrito por uma grade, definindo um mapa de ocupação espacial, sobre o qual é aplicado o A*, um algoritmo de busca informada, pois considera o quanto já avançamos em direção ao destino, e uma estimativa de quanto falta para alcançar nosso objetivo. Usando esta heurística podemos explorar o espaço de possíveis configurações (trajetórias), selecionando sempre a melhor trajetória de acordo com a heurística empregada. Uma outra possibilidade de se identificar o melhor caminho é usando uma descrição geométrica do ambiente e criar um grafo ponderado (pelas distâncias) que uma o ponto de partida ao destino, considerando a menor distância a percorrer, e aplicando o **Algoritmo de Dijkstra**. A Figura 1.11 apresenta um exemplo de trajetória gerado com cada uma das duas técnicas descritas acima, onde uma interessante descrição destes também pode ser encontrada na Wikipedia.

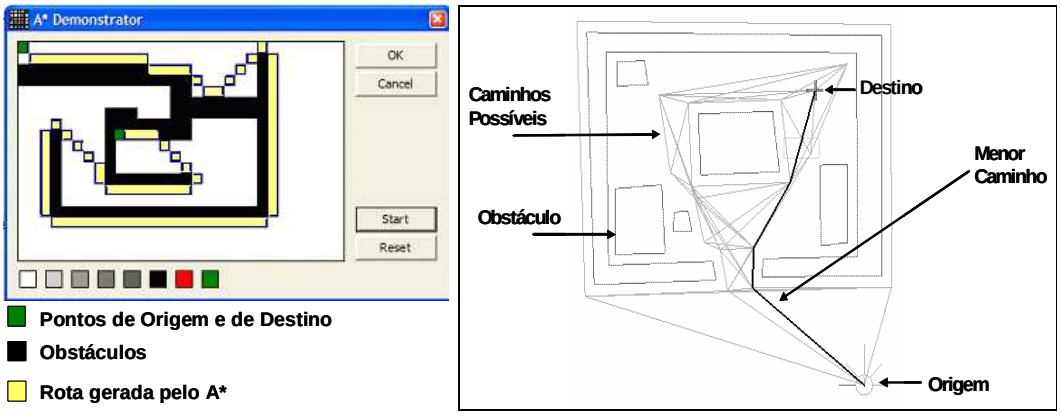


Figura 1.11. Path-Finding usando o A* e o Algoritmo de Dijkstra

É importante destacar que nem sempre todos os jogos irão usar algoritmos mais sofisticados como estes, pois um Pac-Man pode ser implementado usando regras simples (e.g. se está abaixo do alvo, move para cima; se está a esquerda, move para a direita), ou um NPC pode se deslocar apenas cumprindo uma "ronda" que passa por uma rota previamente definida. Entretanto, está claro que a possibilidade de definir rotas ótimas de modo a evitar bloqueios e usar comportamentos inteligentes dará melhores resultados.

A **I.A. para o controle de NPCs** (non-player characters) visa dotar de inteligência os agentes/personagens autônomos de um jogo. Conforme foi indicado anteriormente, o uso de um algoritmo mais eficiente de planejamento de trajetórias também pode ser usado para este fim. Além dos algoritmos de busca de caminhos, existem outras técnicas que são correntemente utilizadas em jogos para o controle do comportamento dos NPCs:

- Autômatos finitos – FSA (*Finite-State Automata*): Os NPCs podem ter um comportamento que simula inteligência, mas que na realidade é bastante fixo e pré-definido, através do uso de um autômato. Esta é uma das técnicas de controle mais utilizadas nos jogos na atualidade. Os autômatos definem uma seqüência de estados e de condições para passagem de um estado a outro, fazendo com que o NPC aparentemente execute ações coerentes. Veja um exemplo na Figura 1.12.

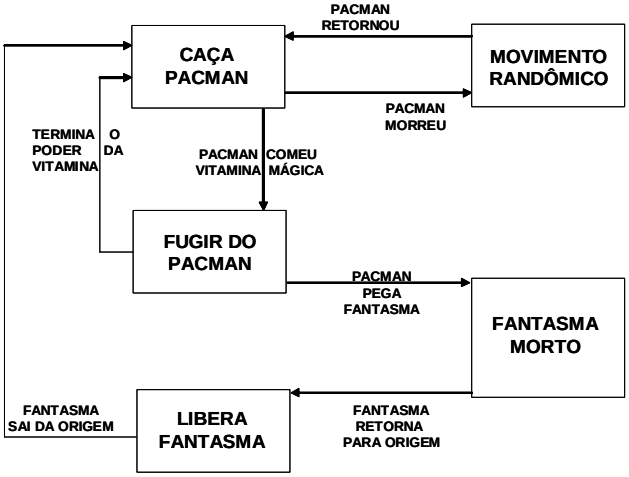


Figura 1.12. Exemplo de um autômato (FSA) usado para o controle do Fantasma no jogo do Pac-Man [adaptado de SCH 04]

- Controle baseado em regras – RBS (*Rule Based Systems*): Um NPC pode ser controlado por um conjunto de regras, como por exemplo, *se* Jogador próximo *e* NPC forte *então* NPC atira; *se* Jogador próximo *e* NPC fraco *então* NPC foge. Na realidade um FSA pode ser alternativamente descrito por um conjunto de regras, sendo portanto um mecanismo similar ao uso dos autômatos, apenas descrito na forma de regras. Note que é possível fazer uso de um sistema baseado em regras mais complexo, onde muitas vezes podemos compor um verdadeiro sistema especialista para controlar um jogo, incluindo uma base de conhecimentos e um mecanismo de inferência que atua sobre os fatos e regras de produção definidas. Além de regras usuais, também é possível usar um sistema de regras baseado na lógica nebulosa (*Fuzzy Logic*), que pode ser bastante útil para que se possa definir regras com elementos como "forte", "fraco", "próximo", "distante", citados no exemplo logo acima. Um exemplo do uso de regras nebulosas em jogos é apresentado por Bittencourt [BIT 02a, 02b].
- Arquiteturas de controle de agentes autônomos: controle deliberativo, reativo, hierárquico, híbrido ou BDI. Estas arquiteturas são bastante estudadas na implementação de agentes autônomos (em robótica ou em ambientes virtuais)[OSO 04].

Agentes baseados no controle reativo sentem o ambiente em que estão inseridos (podem sentir a proximidade de um inimigo, simulando a visão, por exemplo) e a partir das informações sensoriais reagem de modo imediato a estas, executando uma ação. Os agentes reativos usualmente não possuem um raciocínio mais elaborado, ao contrário da arquitetura de controle deliberativa, onde os agentes deliberam (raciocinam) sobre suas ações, planejando estas com antecedência. As arquiteturas reativa e deliberativa possuem cada uma suas deficiências e limitações, sendo assim, usualmente são adotadas arquiteturas modulares do tipo híbrido ou hierárquico, combinando assim diferentes módulos de controle. Por fim, uma arquitetura de controle muito usada é baseada na arquitetura BDI (*Belief-Desire-Intention*), onde os agentes possuem explicitadas suas crenças (incluindo suas percepções), seus desejos e intenções, a partir dos quais realizam o planejamento de suas ações. A Figura 1.13 apresenta exemplos de organização de arquiteturas de controle de agentes (NPCs).

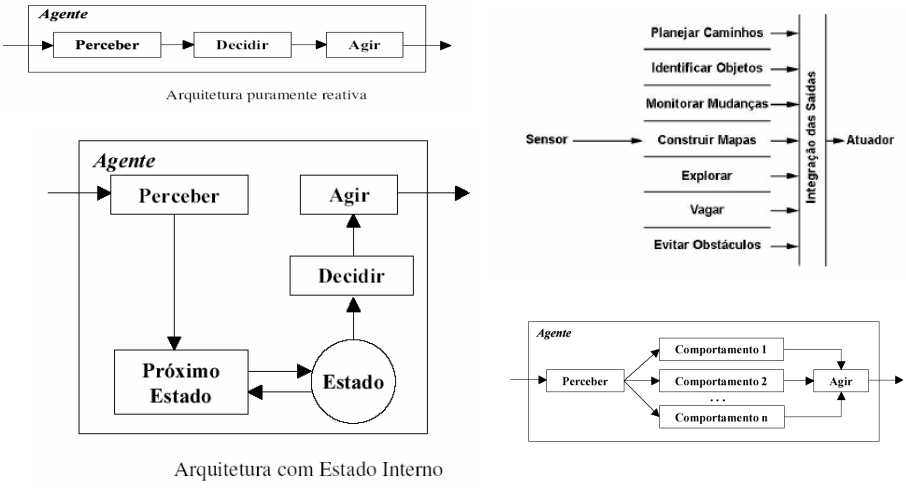


Figura 1.13. Exemplo de arquiteturas de controle de agentes autônomos

- Sistemas multi-agentes: coordenação (*ant systems*, *flocks, swarm)*, comunicação, cooperação, organização e estratégia. Os jogos onde temos vários NPCs que interagem entre si e com o ambiente devem possuir um controle que leve em consideração os seguintes elementos: coordenação, comunicação e cooperação. Existem diversas técnicas de I.A. que buscam promover este tipo de comportamento coletivo, onde podemos destacar os sistemas de *ant colonies*, *flocks* e *swarms*. Podemos imaginar um conjunto de naves espaciais que buscam atacar o inimigo como sendo um "enxame de abelhas" ou um "bando de pássaros", buscando assim, através de algumas regras e comportamentos pré-estabelecidos, reproduzir estes comportamentos de forma organizada e coordenada. O comportamento coletivo em jogos é um tema bastante complexo e exige muitas vezes soluções específicas, que incluem diferentes técnicas de coordenação, organização e cooperação entre os agentes. Podemos citar como referências nesta área a obra Multiagent Systems [WEI 99] e as bibliotecas OpenSteer e Boids criadas por Reynolds [REY 06].
- Sistemas adaptativos e com aprendizado (*Machine Learning*) [MIT 97, REZ 03]. Os agentes autônomos de um jogo também podem ser controlados baseados em técnicas de Machine Learning estudadas na Inteligência Artificial, como por exemplo:

Raciocínio baseado em Casos - CBR (*Case Based Reasoning*), Redes Neurais - ANN (*Artificial Neural Networks*), Algoritmos Genéticos - GA (Genetic Algorithms), RL (*Reinforcement Learning*), IDT (*Induction of Decision Trees*) e Raciocínio Probabilista (*Bayesian Networks*). Usualmente os motores de jogos não incluem todas estas técnicas e ferramentas, sendo que atualmente podemos encontrar diversas ferramentas e bibliotecas para a implementação destas técnicas, mas usualmente implementadas em pacotes específicos e separados. Por exemplo, os algoritmos genéticos podem ser implementados com o uso da **GALib**[5], as redes neurais artificiais com o uso do **SNNS**[6] e as árvores de decisão com o **C4.5**[7]. Algumas poucas implementações, como o **WEKA**[8] reúnem diversos modelos de aprendizado, entretanto por ser implementada em Java e com fins de uso mais para a pesquisa, sua aplicação em jogos pode acabar ficando um pouco mais restrita. A questão da performance é de grande importância quando abordamos a implementação de sistemas adaptativos em jogos, e este tema é ainda um tema de pesquisa atual na área, com poucas implementações realmente funcionais (jogos comerciais) que utilizam de modo prático o aprendizado. Entretanto, devemos destacar que em breve os jogos deverão cada vez mais integrar este tipo de ferramentas de modo a tornar o comportamento dos NPCs e funcionamento destes cada vez mais realista.

Os motores de I.A. para jogos vem sendo continuamente aperfeiçoados e atualmente encontramos algumas soluções comerciais, como o DirectIA e o Dark A.I., e iniciativas de código livre, como o OpenAI. O **DirectIA** [DIA 06] oferece um *kernel* para a implementação de comportamentos autônomos e adaptativos, podendo ser integrado em aplicações através do uso de um SDK. O **Dark A.I.** [DBP 06b] é um módulo de extensão da ferramenta RAD (*Rapid Application Development*) para jogos, o DarkBasic Pro. Esta ferramenta oferece funções para a criação de trajetórias com desvio de obstáculos, usando o A*, além de oferecer também funções para a implementação de comportamentos inteligentes em agentes (e.g. comportamento reativo). O **OpenAI** [OAI 06] é uma iniciativa de código aberto, que visa oferecer ferramentas e implementações de técnicas de Inteligência Artificial. O OpenAI oferece atualmente implementações de Redes Neurais Artificiais, Algoritmos Genéticos e Autômatos Finitos.

A maioria dos motores profissionais de jogos apresenta alguma funcionalidade que implementa funções de controle de trajetórias ou comportamento de NPCs, entretanto podemos afirmar que ainda não existe disponível alguma solução mais completa de I.A. que integre desde a I.A. clássica, passando pelos algoritmos de planejamento de trajetória, controle de comportamento de agentes, aprendizado e adaptação, assim como comportamentos inteligentes de grupos de agentes. O desenvolvedor que buscar ter acesso a tais ferramentas terá que integrar soluções de diferentes origens.

---

[5] GALib - http://lancet.mit.edu/ga/

[6] SNNS e JavaNNS - http://www-ra.informatik.uni-tuebingen.de/SNNS/

[7] C4.5 - http://www.rulequest.com/Personal/

[8] WEKA - http://www.cs.waikato.ac.nz/ml/weka/

1.7. Física

Um dos importantes módulos de um motor de jogos é aquele responsável pela simulação do comportamento físico dos elementos do jogo. Por comportamento físico entende-se: a movimentação de elementos (e.g. carros, aviões, queda livre), através da simulação da cinemática destes; a simulação dos efeitos de gravidade, atrito, colisão, aceleração e desaceleração, considerando as propriedades dos elementos (e.g. peso, dimensões, juntas, elasticidade), através da simulação da dinâmica destes; a simulação da aplicação de forças em corpos rígidos e/ou elásticos (deformáveis), incluindo a reação a colisões (e.g. quebrar, deformar, saltar e quicar); a simulação da interação entre múltiplos corpos e a simulação de efeitos naturais (e.g. água, fogo, vento, explosões, sistemas de partículas). A simulação física, seja ela implementada de forma matematicamente exata ou através de aproximações cujo objetivo é apenas um bom aspecto visual final, é de grande importância para o aumento do realismo em jogos digitais.

Os usuários de jogos vêm aumentando seu nível de exigência em relação ao realismo físico de jogos e atualmente o quicar da bola de tênis virtual em um jogo como o Pong da Atari vem sendo substituído pela sofisticação de colisões e interações entre centenas de elementos móveis de uma cena de guerra ou por colisões espetaculares de veículos de corrida que se despedaçam e deformam de modo quase perfeito. Jogos aparentemente simples como Pebolim (Totó ou Dedobol), Sinuca, PinBall, Boliche, Bolas de Gude e Futebol, todos tem em comum uma grande dificuldade: simular os efeitos físicos de modo a dar uma sensação realista ao jogador em relação a interação entre os elementos presentes no jogo.

A movimentação dos elementos dando a eles um comportamento físico, algumas vezes é tão complexa ou sofisticada, que por certas vezes pode levar os desenvolvedores de jogos a chamar incorretamente esta simulação física de módulo de Inteligência Artificial (I.A.) de controle de comportamento dos elementos do jogo. É interessante que seja diferenciado o controle da física, do controle do comportamento inteligente dos agentes em um jogo. No entanto, esta separação em algumas situações pode mesmo acabar se misturando um pouco. Atualmente encontramos na literatura referências do uso de técnicas de I.A. para o controle do comportamento (físico e dinâmico) de elementos animados em jogos e simulações [FUN 99, TER 98, HEI 06]. As técnicas de I.A. muitas vezes buscam acelerar a simulação física, que se fosse realizada pelos métodos tradicionais poderiam demandar um grande tempo de processamento.

Em relação aos motores de física, uma das ferramentas mais conhecidas e utilizadas, principalmente devido a ser uma ferramenta de código aberto, é o **ODE – Open Dynamics Engine** [ODE 06]. A biblioteca ODE, definida como um *Physics SDK*, permite simular efeitos de física, incluindo a cinemática e dinâmica de corpos rígidos, onde destacam-se os recursos de simulação de veículos e aviões, forças (gravidade, atrito), detecção de colisões, interação entre objetos rígidos em colisão e interação entre objetos rígidos com juntas e articulações. A ODE vem sendo usada por diversos desenvolvedores de jogos, onde podemos citar os seguintes jogos que usam as funções da ODE: Amsterdam Taxi Madness, Elite Heli Squad, Manhattan Chase, Shanghai Street Racer, Taxi3, Hellforces, BloodRayne2 e Pedal To The Metal, entre outros. A ODE também tem sido utilizada na simulação de veículos e robôs autônomos em realidade virtual [HEI 06a, WBT 06], que necessitam de uma simulação física o mais realística

possível. Além disto, a ODE também vem sendo integrada em outros motores, como por exemplo na ferramenta RAD para jogos FPSCreator e DarkBasic Pro [DBP 06a] e pode também ser usada junto ao OGRE3D [OGR 06]. Podemos ver na Figura 1.14 alguns exemplos de uso da ODE, que apesar de ser uma das bibliotecas mais completas de simulação física, não inclui a simulação de corpos deformáveis.

Taxi3 Flight Simulator Webots Xpand Rally

Figura 1.14. Exemplo de jogos e simuladores que usam a ODE

Em relação as ferramentas comerciais, uma das mais conhecidas é a HAVOK [HAV 06] que permite a integração de funcionalidades para simulação de colisão e de veículos em jogos. Entretanto, tanto a ODE quanto a HAVOK são ferramentas de simulação em software da física em jogos, e atualmente uma tendência que tem se verificado é a migração da simulação física para placas de hardwares dedicadas. Assim como ocorreu com a visualização 3D, que era processada em software e atualmente em praticamente todos jogos 3D faz uso de placas de hardware (GPUs – *Graphics Processing Unit*: Nvidea, ATI), o mesmo está ocorrendo com a simulação física.

O novo hardware da PPU – *Physics Processing Unit* oferece recursos avançados e alta performance para a simulação física de movimento e interação, o que inclui: colisões, cinemática, dinâmica e sistemas de partículas, e também efeitos de simulação de explosões, fogo, fumaça, nuvens, tecidos e fluídos. A empresa AGEIA domina este mercado com a sua solução em hardware, o processador AGEIA PhysX [AGE 06]. A solução proposta pela AGEIA possuí vantagens como:

- Disponibilização de APIs em software compatíveis com este novo hardware de modo que possamos desenvolver aplicações com emulação em software dos novos recursos oferecidos pela PPU. Algumas ferramentas podem ser obtidas de graça se não forem para fins comerciais;
- Disponibilização de ferramentas para o desenvolvimento e teste de aplicações que sejam baseadas no hardware da PPU, de modo a permitir um melhor desenvolvimento de jogos e ajuste dos recursos de simulação física utilizados;
- Possibilidade de criação de novos jogos que façam uso do novo hardware que poderão ter um desempenho muito superior as versões em de simulação em software, o que permite a criação de efeitos com mais partículas, elementos móveis e detalhes que em qualquer outro jogo disponível na atualidade;
- Integração com motores, como o DarkBasic Pro e OGRE3D (NxOGRE) que já oferecem soluções orientadas para o uso dos recursos do AGEIA PhysX.

Concluindo, é uma tendência atual importante esta a da adoção de um conjunto de 3 processadores (ou mais) dedicados para diferentes funções em jogos, a saber, a CPU (gerencia e executa tarefas centrais, incluindo a parte de Interação, Rede, Som e I.A. do jogo), uma GPU (parte gráfica) e uma PPU (parte da simulação física).

1.8. Jogos *Multiplayer* – Módulo de Rede

Nas décadas passadas oferecer a possibilidade de jogar com múltiplos jogadores em uma rede local, Internet ou modem era um diferencial. Atualmente praticamente é requisito mínimo dos jogos oferecer um modo *multiplayer* em rede. A maioria dos jogos para consoles oferecem suporte para mais de um jogador, entretanto todos devem estar reunidos presencialmente e interconectados no mesmo dispositivo. Com a popularização da web tornou-se mais fácil a criação de jogos com multi-jogadores sendo que estes não precisam estar reunidos presencialmente e não requer que estão interconectados no mesmo dispositivo.

No âmbito das redes de computadores dois conceitos importantes precisam ser destacados, pois estão fortemente relacionados aos motores de jogos *multiplayer*. O primeiro os protocolos TCP e UDP e os modelos arquiteturais cliente/servidor e *peer-to-peer* (P2P).

O protocolo TCP (*Transmission Control Protocol*) é o mecanismo de comunicação utilizado pela Internet atual. Cria uma conexão entre dispositivo emissor e receptor e garante a entrega de todos os pacotes na mesma seqüencia de envio. Em contrapartida estes mecanismos de segurança acabam adicionando um *overhead* indesejado nos jogos digitais. Quando deseja-se velocidade de processamento em geral adota-se o protocolo UDP (*User Datagram Protocol*) que não é orientado à conexão, os dados são enviados em pacotes (*datagramas*) para uma determinada porta de uma máquina servidora sem nenhuma garantia que esta irá recebê-lo, muito menos na ordem adequada. Os jogos em geral combinam estes protocolos, por exemplo, podem usar TCP para autenticação do usuário em uma rede de assinantes e o UDP para trocar mensagens de *chat*. Também na própria lógica do jogo os protocolos podem ser combinados. Supondo, um RTS (*Real Time Strategy*) pode ser utilizado TCP no momento das criações das construções/unidades e o UDP para enviar informações referentes a movimentação das unidades.

O segundo conceito tratam-se das arquiteturas de redes. Pode-se criar uma estrutura cliente/servidor. Em uma rede uma das máquinas é escolhida como servidora, ou seja, recebe as mensagens dos clientes, processa-as (aplica regras do jogo), cria novas mensagens e envia-as para os clientes. Isto pode ser feito usando tanto o protocolo TCP quanto UDP. É claro que esta centralização é um gargalo quanto ao desempenho do jogo, mas sua principal vantagem é evitar *cheating* e autenticar usuários. Se não for adotada esta estrutura com um computador central, onde cada máquina pode comunicar-se diretamente com as demais, denomina-se uma arquitetura *peer-to-peer* (P2P). Da mesma forma que os protocolos podem ser combinados, estas arquiteturas também podem. A partir destes conceitos básicos é possível projetar diferentes arquiteturas de motores com suporte à múltiplos jogadores, desde simples formas de comunicação até redes híbridas com múltiplos servidores.

Para implementar este conceitos é possível utilizar duas bibliotecas livres, um módulo para SDL denominado *SDL_net* [SDL 06b] ou *HawkNL* [HAW 06]. Ambas soluções são distribuídas sob a licença GNU LGPL, multiplataforma (abstração dos sockets em cada uma das plataformas, por exemplo, no Windows utiliza Winsock) e dão suporte para comunicação TCP/UDP. Nenhuma destas bibliotecas foram criadas especificamente para jogos digitais, portanto disponibilizam funções e estruturas de mais

baixo nível. A *HawkNL* é uma solução menos conhecida que a *SDL_net*, portanto com menos documentação e com uma comunidade menor de desenvolvedores. Na Figura 1.15 é apresentado um código que ilustra a cria de um módulo servidor e na Figura 1.16 um módulo cliente. Adotou-se uma arquitetura cliente-servidor e protocolo TCP.

```
//Inicializa a SDL
if(SDL_Init(0)<0){
      //Trata erro
}
else{
      //Inicializa o modulo de rede
      if(SDLNet_Init()<0){
            //Trata erro
      }
      else{
            //Cria a definicao do endereco IP e a porta
            //que o servico sera oferecido
            IPaddress end;
            if(SDLNet_ResolveHost(&end,NULL,5678)>=0){
                  //Abre o socket do servidor
                  TCPsocket server = SDLNet_TCP_Open(&end);

                  TCPclient client;
                  char message[256];
                  //Se o servidor foi criado
                  if(!server){
                        while(1){
                              //Tenta aceitar uma conexao
                              client = SDLNet_TCP_Accept(server);
                              if(!client){
                                    //Aguarda 10ms e tenta novamente
                                    SDL_Delay(10);
                                    continue;
                              }
                              //Recebe e fecha esta conexao
                              SDLNet_TCP_Recv(client,msg,256);
                              SDLNet_TCP_Close(client);

                              //Escreve a mensagem recebida
                              cout << "Mensagem: " << msg << endl;
                              //No caso da mensagem ser Q
                              //encerra o servidor
                              if(msg[0]=='Q') break;
                        }
                        //Encerra o modulo de rede e a SDL
                        SDLNet_Quit();
                        SDL_Quit();
                  }
            }
            else{
                  exit(1);
            }
      }
}
```

Figura 1.15. Código de exemplo da SDL_net para criar um módulo servidor

```
//Inicializa a SDL
if(SDL_Init(0)<0){
      cout << "SDL_Init: " << SDL_GetError() << endl;
      exit(1);
}
else{
      //Inicializa o modulo de rede
      if(SDLNet_Init()<0){
            cout "SDLNet_Init: " << SDLNet_GetError() << endl;
            exit(1);
      }
      else{
            //Cria a definicao do endereco IP usando
            // o endereco e a porta do servidor
            IPaddress end;
            if(SDLNet_ResolveHost(&end,"127.0.0.1",5678)>=0){
                  //Abre o socket com o servidor
                  TCPsocket client = SDLNet_TCP_Open(&end);

                  //Envia uma mensagem para o servidor
                  //O tamanho eh 6 (5 caracteres + NULL)
                  SDLNet_TCP_Send(client,"Teste",6);
                  //Fecha o socket de comunicacao
                  SDLNet_TCP_Close(client);
                  //Encerra o modulo de rede e a SDL
                  SDLNet_Quit();
                  SDL_Quit();
            }
      }
}
```

Figura 1.16. Código de exemplo da SDL_net para criar um módulo cliente

1.9. Motores de Jogos Comerciais e Livres

Todo jogo digital requer um moto de jogo. Pode ser que um motor seja usado para um único título ou será reusado para diversos títulos do mesmo gênero, o que importa e seja qual for o projeto de jogo digital, independente do seu porte será necessário adotar um motor de jogo. Entretanto isto não significa que sempre será necessário projetar e desenvolver um novo motor, este poderá ser comprado ou adaptado a partir de soluções já existentes na comunidade. Seja um grupo de pesquisa na universidade, um grupo de estudantes ou uma empresa deve-se definir o jogo que será desenvolvido e após isto refletir sobre o motor. Para alguns grupos a criação de um novo motor pode representar uma vantagem mercadológica em função da diferenciação tecnológica. Entretanto esta alternativa é a mais complexa e irá requer que parte do projeto seja despendida em tarefas de apoio ao motor. Pode ser que adaptar um motor já existente seja uma alternativa mais viável pois a equipe irá focar exatamente no diferencial. Por outro lado um determinado projeto pode ser mais simplificado, ou gerar um único título não justificando a criação de um motor.

Tendo em vista tais questões o objetivo desta seção é apresentar algumas alternativas comerciais e projetos livres de motores de jogos. Todas estas soluções possuem uma determinada arquitetura de software que precisa ser conhecida com detalhes pelo desenvolvedor e é constituída dos módulos que foram apresentados nas seções anteriores.

1.9.1. Motores Comerciais AAA

Nesta subseção serão apresentados alguns exemplos de motores de jogos que são utilizados nos principais títulos comerciais. Entretanto estas soluções não estão à venda, são soluções que podem ser utilizadas através de acordos comerciais entre *publishing/developer*/criador do motor. Estes motores requerem uma elevada contrapartida financeira do desenvolvedor tornando bastante difícil sua aquisição para o desenvolvedor nacional.

CryTek – The Entertainment Engine

O *CryENGINE* [CRY 06a] foi desenvolvido por uma empresa germânica denominada *CryTek*. Este motor basicamente foi criado e utilizado primeiramente no jogo *Far Cry* e publicado pela *UbiSoft* uma das parceiras da empresa em conjunto com a *Eletronic Arts*. Além de jogos para PC o motor é compatível com jogos para *Playstation II* e XBox. Veja a Figura 1.17, alguns exemplos das cenas sintetizadas com o motor.

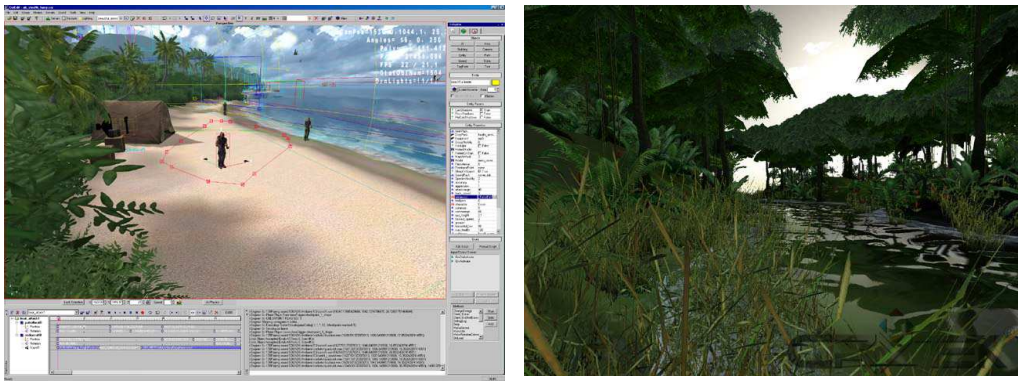


Figura 1.17. Screenshots do editor do CryENGINE e uma imagem renderizada pelo motor.

Conforme o site oficial, o motor oferece suporte à *bump mapping*, luzes estáticas, suporte à OpenGL/DirectX8/9, integração com sistema de física, sistema de Inteligência Artificial programável através de *scripts*, som 3D, suporte aos jogos *multiplayer*, uso de shaders, uso de *heightmap* para modelagem de terrenos, *stencil shadows*, integração com 3DS e Maya, sistema de scripts baseado na linguagem LUA e altamente modularizado em bibliotecas (DLL) escritas em C++.

RenderWare

O *RenderWare* [REN 06] foi criado pela Criterion Software e é utilizada por inúmeros títulos de sucesso para diferentes plataformas (Playstation II, XBox, PSP, GameCube e PC). Pode-se citar Grand Thief Auto: San Andreas, Sonic Heroes, Call of Duty: Finest Hour, The Movies, NBA Ballers, ESPN Major League Baseball, entre outros. No final de 2005, o *RenderWare* foi escolhido pela *Eletronic Arts* como seu motor oficial para criação de jogos digitais. Além da EA, entre seus clientes pode-se citar Rockstar, Atari, Activision, Konami, Sony Entertainment e UbiSoft.

Figura 1.18. Screenshots de jogos criados com o RenderWare.

O *RenderWare* divide-se em duas soluções: *RenderWare Platform* e *RenderWare Studio.* O primeiro trata-se de um motor com suporte à Gráficos 3D, Física, Inteligência Artificial e Áudio. Todos os módulos seguem as tecnologias mais recentes, visam alto desempenho, multiplataforma e permitem que sejam extendidos. Já a segunda solução trata-se de um *framework* para facilitar o desenvolvimento de jogos. Oferece um editor de mundos, gerador de *builds* e um gerenciador de dados para jogos. Veja a Figura 1.18 algumas screenshots de jogos produzidos com o *RenderWare*. Possivelmente o *RenderWare* seja um motores de jogos de maior sucesso na indústria de games, os jogos que utilizaram movimentou 3 bilhões de dólares em 2005 e passou a ser adotado pela Eletronic Arts, a líder de mercado no setor. Para os padrões nacionais é bastante difícil desenvolver algum título com tal tecnologia. Alguns estúdios nacionais chegaram a ter uma experiência com o motor, antes da aquisição da Criterion pela EA, mas desenvolveram somente *tech demos.*

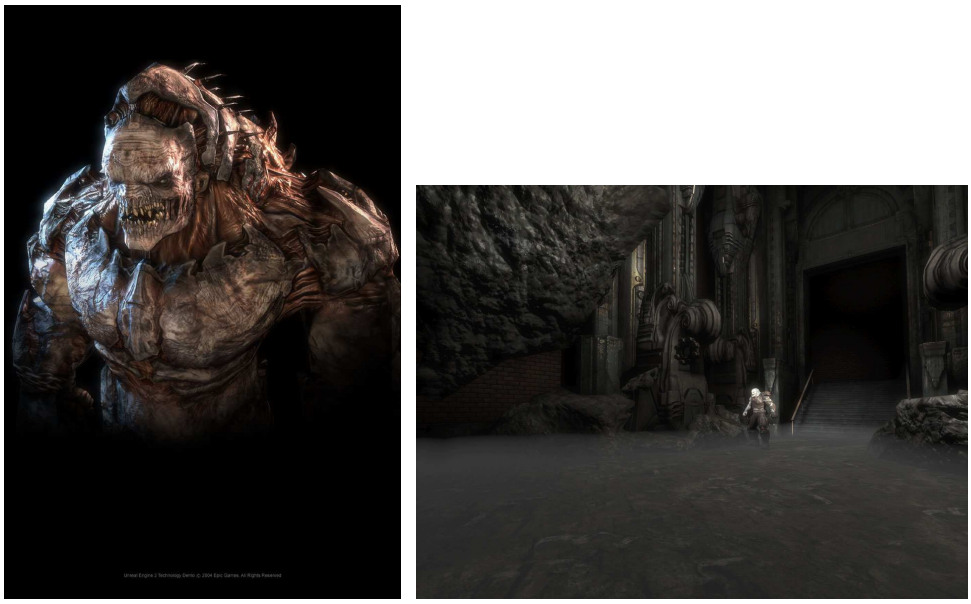


Figura 1.19. *Screenshots* do *Unreal III* criado com a *Unreal Engine III*

Unreal Engine 3

O *Unreal Engine 3* [UNR 06] foi desenvolvido para a próxima geração de consoles pela empresa *Epic Games*. Jogos, tais como, Unreal II, Splinter Cell: Pandora Tomorrow, Unreal Tournament 2003 e Unreal Tournament 2003 foram desenvolvidos com a versão 2 do *Unreal Engine*. Os títulos que estão anunciados que irão utilizar a versão 3 são Duke Nukem Forever, Pariah, Star Wars: Republic Commando, Unreal Championship II entre outros. Veja na Figura 1.19 algumas screenshots do Unreal III. Quanto as características do motor é uma solução extremamente completa. Oferece componentes para gráficos 3D com as técnicas mais recentes de Computação Gráfica, simulação física, módulo de animação de personagens, módulo de Inteligência Artificial, áudio 3D, suporte à jogos multiplayers, integração com 3DS e Maya e uma ferramenta gráfica completa para criação dos títulos que integra todos estes módulos facilitando o processo de desenvolvimento de um determinado título.

Os custos para aquisição da versão 3 do motor não são anunciados pelo fabricante, mas para a versão 2 do motor o custo para licenciamento é U$350 mil dólares para uma plataforma base mais U$ 50 mil dólares por plataforma adicional. Além disso, exige-se um royaltie de 3% sob os lucros do jogo. Para a versão 2 é disponibilizada uma versão demo licenciada para produtos não comerciais ou com propósitos educativos.

1.9.2. Motores Comerciais de Baixo Custo

As soluções que serão apresentadas nesta seção tratam-se de motores de jogos comercializados por um valor mais baixo. Certamente não servem para criação de títulos do mesmo porte que os títulos produzidos pelos motores vistos anteriormente, mas permitem criar jogos digitais bastante completos e com qualidade comercial.

3D Game Studio

O *3D Game Studio* [3DG 05] é um motor de jogo que possibilita o desenvolvimento de um título sem a necessidade de grandes conhecimentos de programação. Por um lado a produção se torna simples e rápida, mas por outro lado as possibilidades são limitadas e as interfaces gráficas dos jogos terão ligeira semelhança entre si. Ou seja, os jogos acabam se tornando muito similares, com pouca diversificação. Quanto ao custo do motor, a versão extra, é U$89,00, que permite produzir jogos sem nenhum marca d'água, entretanto não oferece suporte aos jogos *multiplayers*, simulação física e *shaders*, por exemplo. A versão profissional custa U$899,00, mas sem nenhuma restrição. No site está disponibilizada a versão de demonstração que gera títulos, mas com marca d´ água.

Figura 1.20. *Screenshots* criadas pelo *3D Game Studio*.

Os jogos produzidos são executáveis somente na plataforma Microsoft Windows. Uma característica bastante importante do motor é o fato de ser extensível usando DLLs (*Dynamic Link Library*).

DarkBasic Pro

O *DarkBasic Pro* – DBPro [DBP 06] é um motor comercial para Windows desenvolvido pelo TheGameCreators. A vantagem do uso do DBPro é que esta é uma ferramenta de desenvolvimento rápido de jogos (RAD) é bastante fácil de aprender e simples de usar. Esta ferramenta é adotada na Unisinos no curso de Desenvolvimento de Jogos e Entretenimento Digital, de modo a apresentar aos alunos, já no início do segundo semestre do curso, o desenvolvimento de um jogo 3D completo. A ferramenta oferece facilidades de criação de jogos, incluindo: manipulação de objetos em formatos 3DS, .X (padrão adotado no DirectX) e PK3 (BSP compatível com UT); visualização 2D (bitmaps e textos) e 3D, interação via teclado, mouse ou joystick; funções de áudio (sons midi, wav e MP3); funções de rede (conexão TCP/IP, FTP); assim como extensões que podem ser compradas a parte para Simulação de Física (Dark PhysX) e de inteligência Artificial (Dark A.I.), bem como editores de mapas, de som, modeladores de objetos, ou até mesmo bibliotecas prontas de modelos 3D.

O DBPro é um motor bastante completo e oferece uma IDE para facilitar a criação dos jogos, onde seu custo é de U$89,99 pelo pacote básico do DBPro. A linguagem de programação é o Basic, com extensões de manipulação dos diferentes elementos de um jogo (bitmaps, sons, modelos 3D, câmera, texturas, luzes, interação, etc). A curva de aprendizado do uso do DBPro é bastante rápida, apesar da ferramenta apresentar alguns pequenos problemas (principalmente pela sua constante evolução, o que leva a mudança em alguns comandos) e pelo fato da documentação não ser muito completa e detalhada. Entretanto, esta ferramenta permite ao desenvolvedor iniciante um contato mais direto com os procedimentos de desenvolvimento de um jogo 3D, facilitando a aquisição de conhecimentos sobre a estrutura de um jogo e funcionalidades a serem implementadas e/ou usadas através das funções disponibilizadas por este motor de desenvolvimento de jogos. A Figura 1.21 apresenta alguns exemplos de jogos criados com o uso do DBPro.

Figura 1.21. *Screenshots* de jogos criados com o *DarkBasic Pro*.

Torque Engine

O *Torque Engine* [TOR 06] é um motor comercial multiplataforma (Windows, Linux e MacOS) desenvolvido pelo grupo GarageGames e possui um modelo de negócios bastante interessante. O motor é comercializado com duas licenças: Indie (independente) por U$100,00 e uma comercial por U$495,00. Ambas as licenças são cobradas por desenvolvedor na empresa. Na aquisição da licença comercial a empresa é livre para

desenvolver qualquer título sem efetuar o pagamento de royalties e desenvolver qualquer número de títulos. Na aquisição da licença indie só pode ser feita por pessoas físicas, a empresa não pode ter faturamento superior à U$250 mil dólares, não permite fazer modificações no motor e portar para consoles. Além disso, deve adicionar uma splash screen em tela cheia por quatro segundo exibindo o logo da Torque Game Engine e na tela de créditos/sobre adicionar um link para GarageGames e mencionar *"This games is powered by the Torque Game Engine."*

É um motor bastante completo. Oferece uma IDE para facilitar a criação dos jogos, tem editores de mundos e oferece uma linguagem de scripts "like C" dispensando a programação diretamente em C++. É possível programar o comportamento inteligente e físico das entidades com os scripts. O código completo em C++ é comercializado, oferece suporte à jogos multiplayer, renderização 3D sem suporte à shaders, animações, e som 3D. MarbleBlast Ultra para XBox 360 e Minions of Mirth foram desenvolvido com o *Torque Engine*. Veja na Figura 1.22 algumas screenshots referentes à estes jogos.

Adotar o *Torque Engine* em um projeto de jogo digital pode ser uma excelente solução para os desenvolvedores brasileiros, pois irá dispensar o período de criação do motor e permitirá focar no título. Entretanto vai ocorrer com os títulos gerados pelo Torque o mesmo que acontece com 3D Game Studio, os títulos acabam tornando-se manufaturados, muito semelhantes uns com os outros.

Figura 1.22. *Screenshots* criadas com o *Torque Engine*.

1.9.3. Motores Livres

Tratam-se de projetos que estão disponíveis gratuitamente e livremente para comunidade. Muitos desenvolvedores despendem grande parte de seus projetos recriando um sistema de tratamento de eventos do sistema, por exemplo, que já fora implementado por diversas APIs e/ou motores. Tais desenvolvedores poderiam partir das soluções já existentes, modificando-as, adaptando-as e inclusive redistribuindo-as para comunidade.

Infelizmente muitos desenvolvedores ainda possuem dúvidas quanto as licenças livres. É importante destacar que caso de uma biblioteca ou motor que adote a licença GNU LGPL (*GNU Lesser General Public License*) seu software poder ser distribuído

como uma aplicação proprietária sem violar a licença. O mesmo ocorre para licença BSD. A única licença livre que o desenvolvedor deve ter um pouco mais de cuidado é a GNU GPL, pois esta obriga que a aplicação derivada ou que utilize uma biblioteca/motor com esta licença seja distribuído sob as mesmas condições da licença GNU GPL. Tendo claro estas diferenciações, qualquer produtora pode começar a desenvolver sua tecnologia a partir de soluções livres sem precisa re-inventar elementos já existentes.

Crystal Space

Crystal Space é um projeto *Open Source* desenvolvido por Jorit Tyberghein [CRY 05] [WEN 02]. É distribuída gratuitamente sob a licença LGPL. Desenvolvida usando C++ é compatível com as seguintes plataformas: Linux, Windows e MacOS. No caso do sistema Windows o *Crystal Space* suporta o *DirectX* e em todos os sistemas oferece suporte a biblioteca *OpenGL*. Destaca-se que o *Crystal Space* não é um *engine*, mas trata-se de um *kit* de desenvolvimento para jogos computadorizados 3D. No guia do desenvolvedor [CRY 06] está anunciado: "*Crystal Space é um pacote de componentes e bibliotecas na qual podem ser úteis para criação de jogos computadorizados*". Entretanto usando a arquitetura do *Crystal Space* é possível desenvolver os próprios *engines*.

Apesar de permitir o desenvolvimento de jogos bidimensionais, a maioria de suas funcionalidades estão voltadas para renderização de gráficos 3D em tempo real. Também apresenta suporte a uma série de formatos de som e persistência dos mundos usando o formato XML (*eXtensible Markup Language*). Também permite carregar arquivos no formato 3DS, MDL, MD2, ASE, OBJ e POV. Existe uma farta documentação incluindo tutoriais, guias, referências da API e listas de discussão. Segundo o site do *Crystal Space* [CRY 05] existem mais de 650 pessoas que colaboram com o projeto.

Wen [WEN 02] destaca que o *Crystal Space* possui um diferencial em relação os *engines* 3D proprietários, tais como *Quake III* e *Unreal* pelo fato do *Crystal Space* ser uma ferramenta de propósito geral. Ou seja, não é específica para jogos, ela consiste de uma API gráfica que pode ser utilizada por qualquer aplicação multimídia. Este *toolkit* é bastante usado pela comunidade pelo fato desta ser bastante estável e pela grande quantidade de funcionalidades 3D [WEN 02]. A sua principal fraqueza trata-se do mecanismo para detecção de colisão [WEN 02].

Figura 1.23. Algumas imagens do jogo *PlaneShift* desenvolvido com a Crystal Space

Comparando o *3D Game Studio* com *Crystal Space* certamente é muito mais fácil e intuitivo de utilizar o primeiro do que o segundo, pois oferece uma série de recursos gráficos em uma IDE projetada para criação de jogos. Entretanto o segundo é multiplataforma é permite a criação de novos *plug-ins* para serem adicionados na infra-estrutura geral do motor. Conforme o jargão computacional o *Crystal Space* é uma ferramenta de mais baixo nível, mas em contrapartida permite uma personalização mais ampla. Veja algumas *screenshots* de *games* criados usando o *Crystal Space* na Fig. 1.23.

Fly3D

O Fly3D [FLY 03] foi desenvolvido pelo brasileiro Fábio Policarpo em parceria com Alan Watt durante a elaboração do livro *3D Games Vol. 1 – Real-Time Rendering and Software Technology* [WAT 01]. O livro é a principal fonte de documentação do motor. Atualmente está disponível a versão 2.10. Durante a redação deste documento estava sendo lançada a versão 3.0 alfa com inclusão de luzes dinâmicas. O motor está sendo disponibilizado de forma gratuita sob a licença GNU GPL tanto para uso comercial quanto para o uso não comercial. Tal modificação na licença ocorreu recentemente no projeto que até então não permitia seu uso livremente para aplicações comerciais.

O Fly3D é implementado em C++ e executa somente na plataforma Windows e suporta tanto o DirectX quanto OpenGL. Foi desenvolvido com ênfase para visualizar gráficos 3D, mas permite criar jogos 2D. Apresenta suporte a jogos *multiplayer* e reprodução de sons. Destaca-se que o Fly3D é uma ferramenta de propósito geral, apesar de enfocar jogos de ação em primeira pessoa, jogos de estratégia e de corrida.

O projeto *O Pequeno Guardião do Tempo* que foi premiado no Festival de Jogos Independentes promovido pelo SBGames 2005 utiliza o Fly3D. Além deste projeto, o motor é usado pelo jogo *Penguin Racer* (vencedor do Festival de Jogos Independentes promovido pelo SBGames 2004) e pelo jogo *Taikodom* desenvolvido pela Hoplon.

Figura 1.24. *Screenshots* criadas pelo Fly3D.

1.10. Considerações Finais

Este tutorial teve por objetivo apresentar uma visão global dos motores de jogos e seus diferentes módulos, usados como apoio ao desenvolvimento de jogos digitais. Esperamos que este texto tenha servido como um apoio aqueles que estejam iniciando no desenvolvimento de jogos, bem como, para aqueles que queiram se aperfeiçoar nesta área, onde os conceitos e ferramentas apresentadas possam ajudar os desenvolvedores a criar novos e melhores jogos, aproveitando da melhor forma possível os diferentes recursos e funcionalidades oferecidos pelos diferentes motores de jogos disponíveis.

Referências Bibliográficas

[AGE 06] *AGEIA – PhysX*. Disp. em: http://www.ageia.com/ Acesso em: 10 jul. 2006.

[BIT 04] Bittencourt, João R. **Um Framework para Criação de Jogos Computadorizados Multiplataforma.** Porto Alegre: PUCRS/PPGCC, 2004, 199 p.

[BIT 02a] Bittencourt, João R.; Osório, F. S. **GNU Mages: Um Ambiente para Simulação de Múltiplos Agentes Autônomos, Cooperativos e Competitivos**. In: Anais do III Workshop sobre Software Livre. Porto Alegre, 2002.

[BIT 02b] Bittencourt, João R.; Osório, F. S. **FuzzyF - Fuzzy Logic Framework: Uma Solução Software Livre para o Desenvolvimento, Ensino e Pesquisa de Aplicações de Inteligência Artificial Multiplataforma.** In: Anais do III Workshop sobre Software Livre. Porto Alegre, 2002.

[BUS 96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; et al. **Pattern-Oriented Software Architecture: A System of Patterns**. England: John Wiley & Sons Ltd., 1996, 476p.

[CRY 06a] *CryENGINE*. Disp. em: http://www.crytek.de/. Acesso em: 06 jul. 2006.

[CRY 06b] *Crystal Space*. Disponível em: http://www.crystalspace3d.org. Acesso em: 06 jul. 2006.

[DBP 06a] *DarkBasic Pro Version 6.2*. The Game Creators. Disponível em: http://darkbasicpro.thegamecreators.com/ Acesso em: 10 jul. 2006.

[DBP 06b] *Dark A.I. - DarkBasic Pro*. The Game Creators. Disponível em: http://darkbasicpro.thegamecreators.com/?f=dark_ai Acesso em: 10 jul. 2006.

[DIA 06] *DirectIA – MASA SCI*. Disponível em: http://www.masa-sci.com/directia.htm Acesso em: 10 jul. 2006.

[DOM 03] Domingues, Rodrigo G. **Projeto de um framework para auxilio no desenvolvimento de aplicações com gráficos 3D e animação**. São Carlos, UFSCAR, 2003, 196 p.

[DMN 06] *DevMaster's Game and Graphics Engines Database*. Disponível em: http://www.devmaster.net/engines/ Acesso em 10 jul. 2006.

[DXM 06] DirectX Microsoft. Disponível em: http://www.microsoft.com/directx/ ou http://www.microsoft.com/brasil/msdn/Tecnologias/arquitetura/DirectX.mspx. Acesso em 10 jul. 2006.

[EBE 00] Eberly David H., 3D **Game Engine Design : A Practical Approach to Real-Time Computer Graphics**. Morgan Kaufmann, September, 2000.

[GLT 06] *GLUT – OpenGL Utility Toolkit (Library)*. Disponível em: http://www.opengl.org/resources/libraries/ Acesso em: 10 jul. 2006.

[FUN 06] *John Funge – AI4Games*. Disponível em: http://www.ai4games.org/ Acesso em: 10 jul. 2006.

[FUN 99] Funge, J. D. **AI for Games and Animation: A Cognitive Modeling Approach**. Natick, MA: AK Peters, 1999. 212 p.(Web: http://jfunge.googlepages.com/)

[HAV 06] *Havok Physics*. Disp. em: http://www.havok.com/ Acesso em: 10 jul. 2006.

[HAW 06] *HawkNL*. Disponível em: http://www.hawksoft.com/hawknl/. Acesso em: 06 jul. 2006.

[HEI 06] Heinen Milton; Osório, Fernando S.; Heinen, Farlei ; Kelber, Christian. SEVA3D**: Using Artificial Neural Networks to Autonomous Vehicle Parking Control**. In: IEEE WCCI - IJCNN - Intenational Joint Conference on Neural Networks, Vancouver, Canadá : IEEE Press, 2006.

[HEI 06a] Heinen, Milton; Osório, Fernando S. **Applying Genetic Algorithms to Control Gait of Physically Based Simulated Robots.** In: IEEE WCCI - CEC - Congress on Evolutionary Computation, Vancouver,Canadá: IEEE Press, 2006.

[HOD 06] Hodorowicz, Luke. **Elements of a Game Engine**. 2001. Disponível em: http://www.flipcode.com/tutorials/tut_el_engine.shtml Acesso em 08 jul. 2006.

[IDW 06*] Input Devices Wikipedia*. Disp. em: http://en.wikipedia.org/wiki/Input_device. Acesso em: 10 jul. 2006.

[LUD 06] *LudensWiki*. Disponível em: http://www.ludensartis.com.br/wikimedia. Acesso em: 06 jul. 2006.

[MAD 06] Madeira, André G. **Forge V8: Um Framework para o Desenvolvimento de Jogos de Computador e Aplicações Multimídia.** Recife, UFPE, 2001, 99 p.

[MIT 97] Mitchell, T. M. **Machine learning**. New York: McGraw-Hill. Series in Computer Science, 1997. 414p.

[ODE 06] *ODE – Open Dynamics Engine*. Disponível em: http://www.ode.org/ Acesso em: 10 jul. 2006.

[OGL 06a] *OpenGL*. Disponível em: http://www.opengl.org Acesso em: 10 jul. 2006.

[OGL 06b] *OpenGL Wikipedia*. Disponível em: http://en.wikipedia.org/wiki/OpenGL Acesso em: 10 jul. 2006.

[OGR 06] *Ogre3D*. Disponível em: http://www.ogre3d.org. Acesso em: 06 jul. 2006.

[OAI 06] *OpenAI*. Disp. em: http://openai.sourceforge.net/ Acesso em: 10 jul. 2006.

[OPE 06] *OpenAL*. Disponível em: http://www.openal.org. Acesso em: 06 jul. 2006.

[OSO 04] Osório F. S. et al. **Ambientes Virtuais Interativos e Inteligentes: Fundamentos, Implementação e Aplicações Práticas**. JAI – Jornada de Atualização em Informática – XXIV Congresso da SBC, Salvador, 2004.

[RAB 02] Rabin, Steve (ed.). **AI Game Programming Wisdom.** Charles River Media; 2002, 672 p. (Web: http://www.aiwisdom.com/)

[RAB 03] Rabin, Steve (ed.). **AI Game Programming Wisdom 2.** Charles River Media; dec. 2003, 732 p.

[REN 06] *RenderWare*. Disp. em: http://www.renderware.com. Acesso em: 06 jul.2006.

[RES 03] Rezende, Solange Oliveira. **Sistemas Inteligentes: Fundamentos e Aplicações**. Manole Editora. 2003. 525p.

[REY 06] *Craig W. Reynolds* – Boids and OpenSteer. Disponível em: http://www.red3d.com/cwr/ Acesso em: 10 jul. 2006.

[RIV 06] *Game Programming Gems Book Series* – Charles River Media. Disponível em: http://www.gameprogramminggems.com Acesso em: 10 jul. 2006.

[RUS 95] Russel, S. J. and Norvig, P. **Artificial intelligence : a modern approach.** Upper Saddle River : Prentice-Hall, 1995. 932 p.

[SCH 04] Schwab, Brian. **AI Game Engine Programming**. Charles River Media. 2004. 624 p.

[SDL 06a] *SDL_mixer*. Disponível em: http://www.libsdl.org/projects/SDL_mixer. Acesso em: 06 jul. 2006.

[SDL 06b] *SDL_net*. Disponível em: http://www.libsdl.org/projects/SDL_net. Acesso em: 06 jul. 2006.

[SDL 06c] *SDL - Simple DirectMedia Layer*. Disponível em: http://www.libsdl.org. Acesso em: 06 jul. 2006.

[TER 98] Grzeszczuk R, Terzopoulos D, Hinton G. **NeuroAnimator: fast neural network emulation and control of physics-based models.** ACM SIGGRAPH 98 - International Conference on Computer Graphics and Interactive Techniques. pages 9--20., Addison Wesley, July 1998.

[TOR 06] *Torque Game Engine*. Disponível em: http://www.unrealtechnology.com. Acesso em: 06 jul. 2006.

[UNR 06] *Unreal Engine III*. Disponível em: http://www.unrealtechnology.com. Acesso em: 06 jul. 2006.

[WAT 01] Watt, Alan; Policarpo, Fabio. **3D Games: Real-Time Rendering and Software Tecnology**. Addison-Wesley, 2000, 800 p.

[WBT 06] *Webots – Cyberbotics*. Disponível em: http://www.cyberbotics.com/ Acesso em: 10 jul. 2006.

[WEI 99] Weiss, Gerhard. **Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.** MIT Press, 1999. 643 p.

[WEN 02] Wen, Howard – Crystal. Crystal Space: An Open Source 3D Graphics Engine. In: **Linux Journal** v.2002 issue 97, Maio/2002. Disponível em: http://www.linuxjournal.com/article.php?sid=5514. Acesso em: 08 jul. 2006.