GUFF: UM SISTEMA PARA DESENVOLVIMENTO DE JOGOS

LUIS VALENTE

Dissertação de Mestado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces.

Orientadora: Aura Conci

Niterói, Março de 2005.

GUFF: UM SISTEMA PARA DESENVOLVIMENTO DE JOGOS

LUIS VALENTE

Dissertação de Mestado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces.

Avaliada em Março de 2005

Banca Examinadora

_____

Profª. Aura Conci – IC/UFF

_____

Prof. Bruno Feijó – PUC-Rio

_____

Prof. Esteban Walter Gonzalez Clua – PUC-Rio

Niterói, Março de 2005.

Agradecimentos

A todos os meus amigos e colegas que torceram por mim durante todo o curso e na elaboração desta dissertação.

À minha orientadora, Aura Conci, pela paciência nos momentos difíceis e por ter me aturado durante todo esse tempo :-)

Ao Fabio Policarpo pelas várias dicas e conversas que trocamos durante os últimos meses.

A Daniel Monteiro e Jonh Edson Ribeiro de Carvalho pelas revisões e sugestões em relação ao texto.

Quero também agradecer ao Prof. Bruno Feijó pela oportunidade de cursar a cadeira Animação e Jogos 3D da PUC-Rio.

Resumo

Jogos para computador permitem aplicar conhecimentos de diversas áreas da Computação, como Computação Gráfica, Engenharia de *Software*, Inteligência Artificial e Redes de Computadores, entre outras áreas. Podem ser consideradas como aplicações interativas de tempo real, onde o desempenho é um dos fatores principais que são considerados.

Entretanto, essa preocupação com a eficiência na execução dos programas (e com a aparência visual) tornou comum a prática de se implementar novamente todas as funcionalidades necessárias para a criação de um jogo, a cada projeto. Com o aumento da complexidade dos projetos de jogos, esse tipo de prática tem se tornado inviável. Dessa forma, é preciso buscar outras abordagens de desenvolvimento.

Este trabalho, além de iniciar uma linha de pesquisa na UFF voltada ao desenvolvimento de jogos, tem como objetivo apresentar uma proposta de um *framework* para jogos (o Guff, ou games-uff) que virá a ser aprimorado em trabalhos posteriores nessa linha de pesquisa. Um *framework* para jogos oferece uma possível arquitetura reusável para o desenvolvimento de jogos.

Abstract

Computer games apply concepts from several fields of Computer Science such as Computer Graphics, Software Engineering, Artificial Intelligence and Computer Networks, among others. Additionally, computer games may be regarded as interactive real-time applications, where performance is one of the key considered aspects.

However, this concern about efficient algorithm execution (as well as visual presentation) has led to the recurring practice of implementing all the required functionality to develop a game in a new project. As game development complexity increases, this practice is becoming infeasible. Hence, it's necessary to seek other approaches for game development.

This work, besides initiating a new reasearch field at UFF related to game develoment, presents the design and implementation of the Guff (games-uff) framework, which will be improved in further works in this research field. A game framework offers a reusable architecture for development of new games.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Desenvolvimento de jogos para computador é uma área que permite aplicar conhecimentos de diversos campos da Computação, como Computação Gráfica, Engenharia de *Software*, Inteligência Artificial e Redes de Computadores, entre outras. Dessa forma, essa área possui um grande potencial em termos de pesquisa.

Na Universidade Federal Fluminense, até o presente momento, foram realizados vários trabalhos relacionados a desenvolvimento de jogos em nível de graduação, como (Policarpo e Conci, 2001), (Lopes, 2002), (Valente, 2002) e (Silva et al, 2004). Este trabalho inicia a linha de pesquisa em desenvolvimento de jogos, nesta universidade, em nível de pós-graduação. Em termos nacionais, essa linha de pesquisa é bastante recente, tendo sido inaugurada por (Poyart, 2002).

Uma uma possível classificação para os gêneros de jogos existentes é descrita no capítulo 2. Para cada gênero, são discutidas características que os distinguem dos demais, baseando-se no critério de jogabilidade.

A área de desenvolvimento de jogos para computador é conhecida por tentar explorar ao máximo as capacidades da máquina onde o jogo é executado, para não prejudicar a interatividade. Antigamente, essa característica era praticamente obrigatória em virtude das limitações existentes no equipamento disponível, mais precisamente baixo poder de processamento e pouca memória.

Sendo assim, as maiores preocupações dos desenvolvedores eram relacionadas à eficiência de execução dos algoritmos (aplicação de "truques de programação" e do paradigma "é bom porque funciona") e com a aparência visual do programa ("se é bonito então está certo"), em detrimento a fatores como reusabilidade de código e manutenibilidade. Era comum também que camadas intermediárias entre a aplicação e o *hardware* fossem eliminadas, de forma a acessar o *hardware* diretamente, para obter a melhor eficiência possível. Dessa forma, tornou-se comum a prática de se implementar todas as funcionalidades necessárias para a criação de um jogo, a cada projeto, mesmo com o grande crescimento da capacidade das máquinas.

Entretanto, com o aumento da complexidade dos projetos de jogos, esse tipo de prática tem se tornado inviável. Dessa forma, é necessário procurar novas alternativas de desenvolvimento. O crescimento da complexidade dos projetos de jogos é analisado no capítulo 3, que serve como motivação para a aplicação dos diversos tipo de ferramentas discutidos neste trabalho.

O objetivo principal deste trabalho é apresentar a proposta, os conceitos e implementação para um possível *framework* para jogos, o Guff (games-uff). Um *framework* para jogos representa uma possível arquitetura reusável para desenvolvimento de jogos. Além de *frameworks*, existem outros tipos de ferramentas como bibliotecas, *toolkits* e *engines*. A conceituação desses tipos de ferramentas é um tema bastante polêmico, o que acarreta em falta de consenso e confusão no emprego desses termos. O capítulo 3 apresenta uma possível conceituação com o objetivo de contextualizar a ferramenta desenvolvida neste trabalho.

O Guff é composto por duas partes: a camada de aplicação e o *toolkit*. A camada de aplicação define uma possível arquitetura para jogos, baseada em uma máquina de estados. A máquina de estados é usada para modelar o funcionamento das diversas partes de um jogo (como introdução, menus e fases). O *toolkit* representa um conjunto de classes que oferecem diversos serviços para o desenvolvedor. O capítulo 4 detalha essas camadas e suas principais funcionalidades.

Como já foi apresentado, jogos permitem aplicar conhecimentos de diversas áreas da Computação. Neste trabalho, essas áreas são descritas como módulos computacionais. No capítulo 3, são feitas considerações sobre o uso de visualização, áudio, dispositivos de entrada, linguagens para *script*, redes, simulação física e Inteligência Artificial em jogos. Para cada módulo, são descritas características quando aplicadas a jogos e alguns problemas encontrados.

Um dos fatores para a grande preocupação em relação ao desempenho apresentado por um jogo pode ser explicado ao se tratar jogos como aplicações de tempo real. Mais precisamente, jogos podem ser considerados como aplicações interativas de tempo real. O capítulo 3 apresenta argumentos para justificar essa consideração.

Um outro problema relacionado a jogos refere-se ao modelo de execução. Um modelo de execução implementa a execução das tarefas de um jogo, que podem ser divididas em três partes: aquisição de dados, processamento (atualização do estado do jogo) e apresentação (áudio e vídeo). O capítulo 3 apresenta algumas alternativas para a implementação de um modelo de execução.

No capítulo 5, são relatados os resultados de desempenho obtidos por uma aplicação que utiliza os serviços oferecidos pelo Guff.

O anexo A apresenta diversos exemplos de uso e implementação de diversas partes do Guff.

As conclusões deste trabalho, principais dificuldades encontradas e sugestões para trabalhos futuros são apresentadas no capítulo 6.

2. Gêneros de Jogos

Neste capítulo, é apresentada uma possível classificação para tipos de jogos para computador e *video-games*, resumida na figura 2.1. Essa classificação é baseada nos critérios objetivo e jogabilidade.

Entretanto, é importante ressaltar que não existe um formalismo em relação às classificações para gêneros de jogos. Em (Battaiola et al, 2001), são definidas algumas classificações baseadas no tipo de interface (2D, 3D, primeira pessoa, terceira pessoa, etc), nos objetivos (semelhante à classificação apresentada neste capítulo) e em relação ao número de usuários (monousuários, massivos, etc). Em (Rollings e Adams, 2003), é apresentada uma classificação segundo os objetivos do jogo, assim como em (Hall, 2001) e (Perdersen, 2003). Um outro exemplo é a classificação proposta em (Wikipedia, 2004), que mescla os critérios objetivo, interface e jogabilidade.

Na classificação da figura 2.1, algumas categorias se sobrepõem. Isso ocorre devido à natureza subjetiva das classificações para gêneros de jogos. Jogos que apresentam características de diversos gêneros são considerados como jogos híbridos.

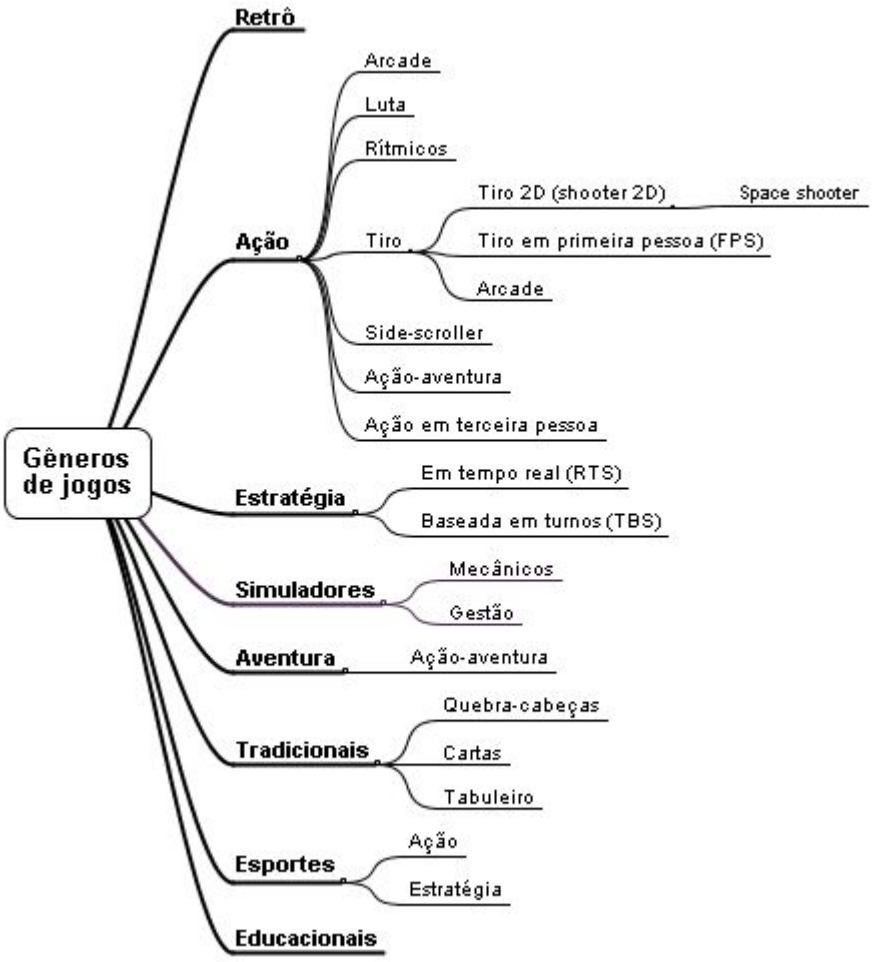

Figura 2.1: Uma possível classificação para gêneros de jogos.

2.1. Jogos de Ação

Segundo (Rollings e Adams, 2003), este estilo caracteriza-se por apresentar ação ininterrupta e veloz, onde as principais habilidades testadas do usuário são tempo de reação e coordenação motora (essencialmente, entre mãos e olhos). Por causa dessas características, a mecânica dos jogos tende a ser simples. Essa caracterização abrange outros subtipos que serão descritos nesta seção.

2.1.1. Fliperama (*Arcade*)

Os jogos fliperama (*arcade*) são os precursores do gênero de ação, sendo que o cenário correspondia a uma única tela. O estilo fliperama é denominado dessa forma porque esse

jogos eram encontrados em máquinas de fliperama (início da década de 80). Atualmente, esse estilo de jogo foi resgatado em virtude da proliferação de dispositivos móveis como telefones celulares.

Em geral, a mecânica desses jogos é bastante simples (até porque na época em que foram concebidos a capacidade das máquinas existentes era bastante limitada). Esses jogos podem apresentar objetivos variados, como percorrer labirintos coletando ítens (como em *Pac-Man*), atirar em tudo que se move (como em *Space Invaders*), salvar um personagem (como uma princesa, em *Donkey Kong*) ou auxiliá-lo em alguma tarefa (conduzir um sapo a atravessar uma rua cheia de carros, como em *Frogger*).

2.1.2. Tiro (*shooters*)

Representam jogos onde o jogador controla algum personagem (um soldado, uma nave, etc) cujo objetivo é, basicamente, permanecer vivo no jogo e atirar em praticamente tudo. Existem outros estilos derivados do gênero, que foram criados de acordo com a ambientação do jogo ou a tecnologia utilizada, por exemplo.

2.1.2.1. Tiro 2D (*shooter* 2D)

Esta variação recebe esse nome por causa dos jogos precursores do estilo, cuja visualização era de duas dimensões. Os cenários utilizados eram fixos ou com rolamento (*scrolling*). A ação é visualizada lateralmente ou por cima.

Geralmente, a mobilidade do usuário no cenário é bastante restrita, sendo possível determinar a direção de tiro ou mover o personagem em uma direção fixa (por exemplo, para cima ou para baixo, na área da tela). Nos jogos que usavam rolamento (*scrolling*), o caminho percorrido pelo usuário no cenário era pré-determinado pelo jogo.

Particularmente, jogos em que o jogador controlava uma nave espacial eram tão difundidos que eles recebiam uma denominação especial: *space shooter* (Rollings e Adams, 2003).

Entre exemplos de jogos do gênero tiro 2D encontram-se: *Gradius*, *R-Type* e *Contra*.

2.1.2.2. Tiro em Primeira Pessoa

Em 1992, foi lançado o jogo *Wolfenstein 3D* (Wolfenstein3D, 2004), que criou mais uma variação para o gênero de tiro: o tiro em primeira pessoa[1].

A proposta usual deste estilo de jogo é colocar o usuário em um ambiente hostil, implementado sob a forma de um ambiente fechado (labirinto), usando um ponto de vista de primeira pessoa. É comum que esses jogos possuam um enredo fraco (ou nenhum), de modo que foco é voltado totalmente para a ação (percorrer o ambiente, atirando em outros objetos e/ou jogadores), embora isso não seja uma regra (Hall, 2001).

Os jogos deste estilo empregam visualização 3D, sendo um dos estilos mais avançados nessa categoria. Outras características encontradas são a possibilidade de partidas em rede (através da *internet*, por exemplo) e sistemas de Inteligência Artificial complexos.

Entre os jogos pertencentes ao estilo de tiro em primeira pessoa, encontram-se *Quake*, *Doom*, *Half-Life* e *Unreal*.

2.1.3. Luta

Enfatizam o combate entre dois personagens, sendo o cenário restrito ao local da luta. É comum que os lutadores utilizem artes marciais, embora a simulação realista das lutas não seja um dos objetivos principais deste estilo de jogo.

Os jogos mais antigos do gênero (como a clássica série *Street Fighter*), utilizavam cenários e personagens bidimensionais, de modo que a ação era visualizada lateralmente. Atualmente, os jogos de luta utilizam ambientes e personagens tridimensionais. Isso possibilita recursos como a visualização da ação a partir de diferentes pontos de vista, mas não altera a mecânica do jogo.

Exemplos de jogos de luta: *Street Fighter*, *Mortal Kombat*, *Tekken* e *Soul Calibur*.

1 *first person shooter*, ou FPS.

2.1.4. Rítmicos

Neste tipo de jogo, o usuário deve seguir seqüências ou criar ritmos usando os controles (que podem ser dispositivos não-convencionais, como tapetes, criados especialmente para o jogo). O uso de controles pode variar desde apertar botões específicos ou literalmente dançar conforme a música. Nesse caso, o usuário deve seguir as seqüências apresentadas conforme o ritmo da música apresentada, acionando (geralmente com os pés) o controle correspondente.

Exemplos de jogos rítmicos: *Dance Dance Revolution*.

2.1.5. Outros

Correspondem aos jogos de ação que não são focados nos ítens anteriores. Elementos comuns nesses jogos incluem correr, pular e lutar. Um exemplo é o gênero *side scroller*. Jogos desse gênero usam cenários e personagens bidimensionais, onde a visualização é lateral. O cenário não cabe em uma única tela, de modo que é preciso efetuar um rolamento de tela (*scroll*) à medida que o usuário o percorre (essa é a origem do nome). Exemplos de jogos *side scroller* incluem: *Sonic The Hedgehog* e *Super Mario Bros*.

Versões atuais de outros tipos de jogos de ação tipicamente utilizam cenários totalmente tridimensionais, onde o jogador tem liberdade de locomover-se em qualquer direção dentro do cenário. O ponto de vista é, geralmente, localizado atrás do jogador. Essa configuração do ponto de vista (ou câmera) é conhecida como visão em terceira pessoa.

2.2. Estratégia

Jogos de estratégia focam em planejamento e desenvolvimento de táticas. O usuário participa de simulação como um espectador que tem o poder de alterá-la, sem controlar um personagem específico. São divididos em dois ramos, que são estratégia em tempo real[2] e estratégia baseada em turnos[3].

2 *real-time strategy*, ou RTS.
3 *turn-based strategy*, ou TBS.

A diferença entre os jogos RTS e TBS é que neste último, o desenrolar do jogo dá-se em turnos, ou seja, o jogador da vez não tem limite de tempo para fazer seu planejamento e tomar decisões. Um exemplo de jogo TBS é *Civilization II*.

Nos jogos RTS, o planejamento e as conseqüências das decisões tomadas ocorrem simultaneamente, para todos os jogadores envolvidos.

É comum que os jogos de estratégia utilizem uma visão panorâmica do cenário onde o jogo é ambientado. Essa visualização pode ser implementada em duas ou três dimensões. Atualmente, é comum o uso de visualização 3D nesses jogos, mas o realismo visual não é um dos objetivos principais deste gênero de jogo.

As partidas podem ser realizadas localmente ou em rede. No caso de jogos locais, os oponentes virtuais são implementados usando técnicas de Inteligência Artificial, que podem ser sofisticadas.

De acordo com (Wikipedia, 2004), jogos de estratégia também podem ser conhecidos como jogos de guerra, pois é comum a utilização de temáticas militares.

Outros exemplos de jogos de estratégia: *Warcraft* (RTS), *Command and Conquer* (RTS) e *Alpha Centauri* (TBS).

2.3. RPG

O estilo RPG para computador (*computer role playing game*) originou-se do seu equivalente em papel (Adams, 2003). Neste estilo de jogo, o usuário assume o papel de um ou mais personagens, envolvidos em algum tipo de jornada. Esses personagens possuem uma série de características e peculiaridades individuais. Segundo (Rollings e Adams, 2003), os jogos deste estilo caracterizam-se por:

- Permitir que os personagem sejam personalizados e que suas características evoluam com o decorrer do jogo;
- Possuir um enredo muito desenvolvido.

De acordo com (Hall, 2001), as características mais importantes de um RPG são o enredo, a interação e a profundidade de sua estória. Os jogos RPGs podem usar visualização 2D ou 3D.

As estórias apresentadas pelos RPGs freqüentemente possuem temáticas fantasiosas ou de ficção científica. Essas estórias apresentam vastos cenários que podem ser explorados pelo jogador, de maneira não-determinística.

Analogamente aos jogos de estratégia, os RPGs mais antigos eram baseados em turnos. Os jogos mais recentes possuem características de ação em tempo real. Exemplos de RPGs baseados em turnos: *Diablo* e *Final Fantasy*.

A tendência atual dos RPGs é proporcionar partidas em rede, através da *internet*. Essas partidas podem concentrar milhares de usuários. Essa variação é conhecida como *massively multiplayer online role playing game* (MMORPG). Uma característica muito importante dos MMORPGs é o de serem mundos persistentes, ou seja, o mundo continua a existir e progredir mesmo com o jogador desconectado. Segundo (Perdersen, 2003), todo RPG possui um determinado objetivo e um fim. Entretanto, alguns MMORPGs podem não possuir um final específico, sendo o objetivo final definido pelo usuário (por exemplo, completar todas as tarefas atribuídas inicialmente). Exemplos de MMORPGs: *Ultima Online* e *EverQuest*.

2.4. Aventura

Segundo (Rollings e Adams, 2003), jogos de aventura (*adventure*) são estórias interativas sobre um personagem que é controlado pelo jogador. Esses jogos não são baseados em competições ou simulações, nem oferecem processos a serem geridos ou um oponente a ser derrotado através de táticas e criação de estratégias. Tipicamente, são apresentados vários quebra-cabeças para o usuário resolver.

Em jogos desse tipo, o jogador inicia uma jornada em busca de algum objetivo pré-definido, normalmente com poucos recursos. Ao longo do jogo, é possível coletar objetos que podem ser úteis para resolver os quebra-cabeças (é comum que esses objetos sejam parte da solução). Os quebra-cabeças podem ser de vários tipos, como mover peças em uma determinada ordem, resolver charadas, explorar labirintos, entre outros. Usualmente, o jogo não progride enquanto o jogador não conseguir resolver algum desafio proposto.

O primeiro jogo do gênero, *Colossal Cave Adventure* (Jerz, 2004) (que surgiu nos anos 70), era totalmente baseado em texto. Mais tarde, outros jogos mais avançados (mas ainda baseados em texto) foram lançados por empresas como a *Infocom*. Logo depois que os primeiros PCs com capacidade para processamento gráfico tornaram-se disponíveis, surgiram os primeiros jogos de aventura com arte gráfica, que contribuíram para popularizar o gênero. Títulos como *Maniac Mansion* e *Monkey Island* estão entre os grandes clássicos deste estilo de jogo.

Atualmente, existe uma variação do gênero conhecida como *action-adventure* (Rollings e Adams, 2003), que consiste em uma combinação entre ação e aventura. Esses tipos de jogos possuem um ritmo mais acelerado que os jogos de aventura tradicionais, incluindo outros tipos de desafio, como lutas. Um exemplo é *Indiana Jones and the Infernal Machine*.

2.5. Esportes

Os jogos de esportes podem ser divididos em duas categorias, segundo (Perdersen, 2003): os que são focados na ação da modalidade esportiva e os que são focados em estratégia (gestão esportiva). Os jogos focados em estratégia possuem características bastante semelhantes aos jogos de estratégia tradicionais, podendo mesclar elementos de jogos TBS e RTS. Um título representativo é *Championship Manager*.

Os jogos que são focados na ação ambientam-se no local de disputa da modalidade esportiva (quadra, campo, etc). Existem jogos que procuram oferecer uma simulação realista do esporte, o que torna complexos seus sistemas de visualização, de Inteligência Artificial e de simulação física. Alguns títulos que se enquadram nessa descrição são *FIFA Soccer* e *NBA Live*.

Alguns jogos procuram satirizar a modalidade esportiva, trocando o realismo pelo deboche. Exemplo: *NBA Jam*.

2.6. Simuladores Mecânicos

Este tipo de jogo procura oferecer aos jogadores experiências encontradas no mundo real, sem os perigos associados a elas. Os tipos de simuladores mais comuns são simuladores de veículos, que procuram simular o funcionamento de aviões, carros (corridas), motos e helicópteros, entre outros.

Um simulador ideal deveria apresentar um alto grau de realismo em visualização, áudio e simulação física. Na prática, existe uma grande variação em relação ao grau de realismo. Vários jogos (como *Microsoft Flight Simulator*), procuram oferecer uma experiência tão realista quanto o possível. Alguns outros, como *Mechwarrior*, simulam máquinas e ambientes fictícios. Como essas máquinas não existem, elas podem conter vários tipos de características imaginárias (como armamentos, acessórios, etc).

Existem ainda títulos que possuem um frágil compromisso com a realidade, como *Crazy Taxi* e *OutRun 3D*, que são jogos de corrida. Esses jogos trocam o realismo por um maior grau de entretenimento.

2.7. Simuladores de Gestão

Segundo (Rollings e Adams, 2003), jogos deste estilo proporcionam ao usuário gerir algum tipo de processo. Por exemplo, gerir uma cidade, gerir uma empresa, administrar um parque de diversões, administrar o crescimento e desenvolvimento de uma colônia de formigas, entre outras atividades. O objetivo do jogador não é derrotar inimigos, mas controlar o processo em torno do qual o jogo foi projetado.

Exemplos de simuladores de gestão: *SimCity*, *The Sims*, *Theme Park* e *Capitalism II* .

2.8. Retrô

Este estilo representa releituras de jogos consagrados em outras épocas. As novas versões procuram atualizar a parte visual, com visualização 3D (já que os jogos antigos usavam somente 2D), sem alterar a mecânica do jogo original.

Jogos originais atuais também podem vir a ser classificados nesta categoria, por causa da mecânica de jogo. Por exemplo, *Viewtiful Joe* (Capcom, 2004) é um jogo recente que possui cenários tridimensionais mas utiliza elementos de jogos *side scroller*.

2.9. Educacionais

Segundo (Wikipedia, 2004), jogos educacionais procuram ensinar algum conceito ao usuário utilizando o jogo como meio. Embora os jogos educacionais possam ser semelhantes a vários outros de outros estilos, o seu diferencial é a ênfase no aprendizado. Por exemplo, os jogos da série *Carmen Sandiego* exploram a geografia mundial e a cultura de diversos países.

2.10. Quebra-cabeças, Tabuleiro, Cartas

Estes tipos de jogos desafiam o jogador ao apresentar problemas que requeiram raciocínio e paciência (Hall, 2001). Os desafios podem ter equivalentes no mundo real (um jogo de cartas) ou não possuir nenhum tipo de relação (como *Tetris*).

Exemplos de jogos de quebra-cabeças, tabuleiro ou cartas incluem *Tetris*, *Blockout*, *Monopoly*, *Sokoban* e *Minesweeper*.

2.11. Conclusão do Capítulo

Não existe uma classificação padronizada para os estilos de jogos. Muitas delas são realizadas de maneira subjetiva. Neste capítulo, foi apresentada uma possível classificação baseada em aspectos de jogabilidade.

As classificações de jogos podem ser realizadas segundo diversos critérios. Por exemplo, em (Battaiola et al, 2001), são definidas classificações baseadas no tipo de interface (2D, 3D, primeira pessoa, terceira pessoa, etc), nos objetivos (semelhante à classificação apresentada neste capítulo) e em relação ao número de usuários (monousuários, massivos, etc).

3. Desenvolvimento de Jogos

Os jogos de computador mais antigos (início da década de 80) eram relativamente simples em virtude dos poucos recursos computacionais disponíveis na época, mais precisamente poder de processamento e quantidade de memória existente.

Dessa forma, as maiores prioridades dos desenvolvedores eram a eficiência na execução do programa e o tamanho do código gerado. O código precisava ser bastante eficiente ou a máquina não seria capaz de executar o jogo de forma satisfatória. O tamanho do código não poderia ser muito grande ou ele não caberia na memória disponível. Para cumprir esses requisitos, os desenvolvedores usavam *assembler*. Era comum também que camadas intermediárias entre o programa e o *hardware* fossem eliminadas, de forma a acessar o *hardware* diretamente, para obter a melhor eficiência possível.

De fato, essa necessidade tornou-se um padrão em desenvolvimento de jogos. Devido a esse padrão de "máxima eficiência", é comum que os desenvolvedores de jogos queiram reescrever o código necessário para a criação de um jogo ao invés de reutilizar códigos (ou componentes) desenvolvidos por terceiros. Tal prática é conhecida também como "*not built here*" (Rollings e Morris, 2003). Essa expressão significa "se não foi construído por nós, então não deve ser bom". Entretanto, com o aumento da complexidade dos projetos de jogos, esse tipo de prática tem se tornado inviável.

A utilização de *assembler* para o desenvolvimento de jogos tornou-se menos importante conforme o poder de processamento dos computadores evoluiu. Em 1993, foi lançado o jogo *Doom* (Doom, 2004), um marco na época, que foi implementado com a linguagem C em quase sua totalidade (*assembler* foi usado em poucas partes do programa). Esse jogo, de certa forma, ajudou a diminuir a resistência contra o uso de linguagens de alto nível na implementação de jogos.

Em 1994, a estrutura de um típico jogo da época (por exemplo, um *side scroller*) poderia ser representada pela figura 3.1.

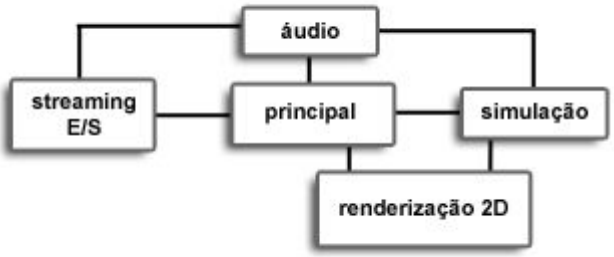

Figura 3.1: Estrutura de um jogo 2D de 1994 (Blow, 2004).

A figura 3.1 ilustra as conexões entre os módulos existentes na aplicação. O módulo principal representa a estrutura principal do programa, que controla os outros módulos e agrega funcionalidades que não são complexas o suficiente para formar um módulo único.

A figura 3.2 ilustra uma estrutura para um jogo de 1996. Nessa época, foram lançados jogos como *Quake* (Quake, 2004), que utilizavam visualização tridimensional. As primeiras placas gráficas aceleradoras surgiram também nesse período, que pode ser considerado como o início da popularização dos chamados *engines* 3D. A grosso modo, um *engine* 3D é um programa que implementa uma arquitetura pré-determinada para jogos com visualização tridimensional. Esse tipo de ferramenta é discutida neste capítulo. Ao observar a figura 3.2, é possível perceber um ligeiro aumento de complexidade para a estruturação dos jogos.

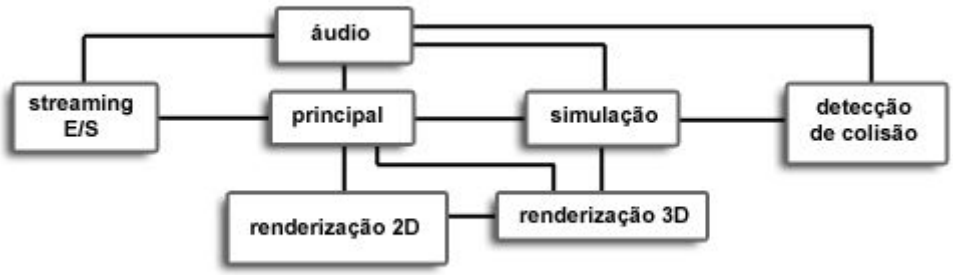

Figura 3.2: Estrutura de um jogo 3D de 1996 (Blow, 2004).

Atualmente, a estrutura dos jogos atuais é mais complexa. Por exemplo, a figura 3.3 ilustra uma possível estrutura para um jogo monousuário (*single-player*) de 2004.

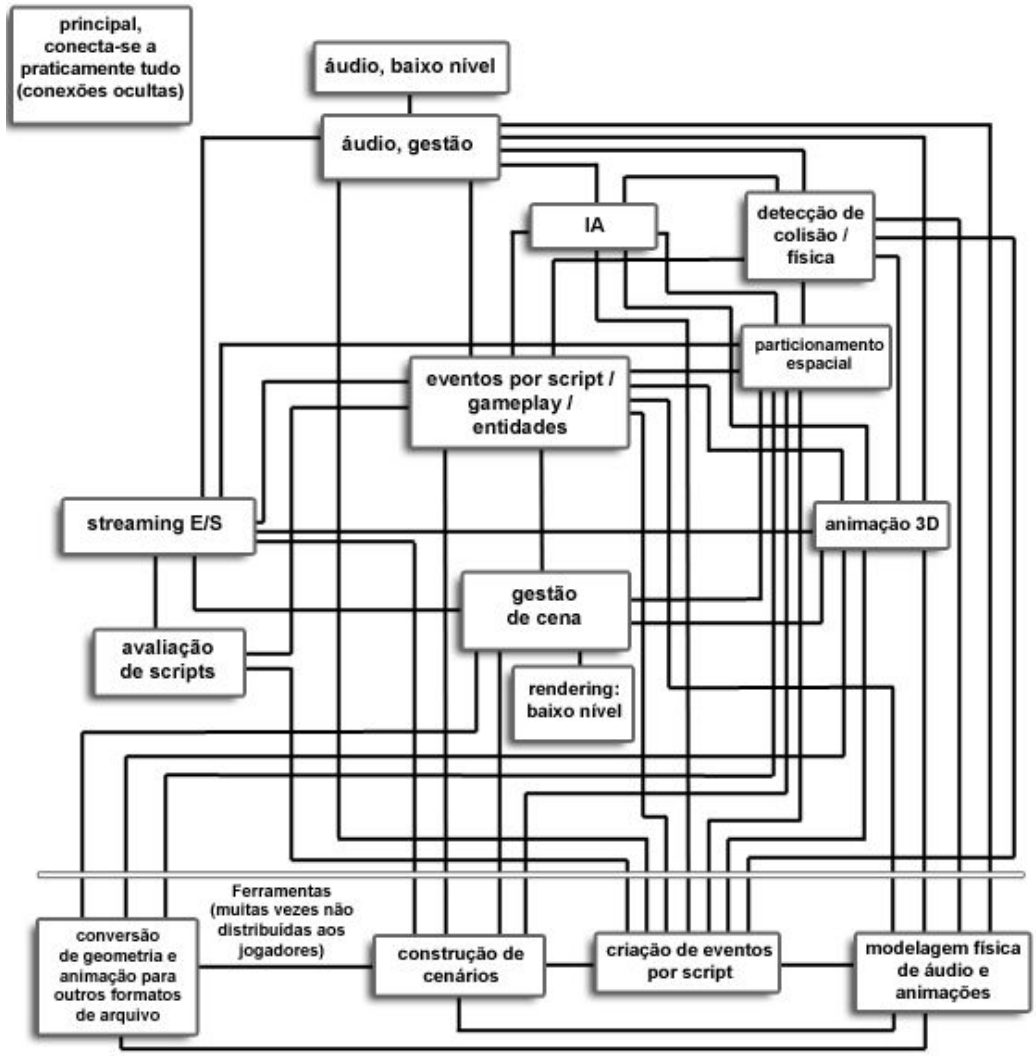

Figura 3.3: Estrutura de um jogo 3D de 2004 (Blow, 2004).

Comparando-se as figuras 3.1, 3.2 e 3.3, é possível apreciar porque a aplicação de práticas como "*not built here*" em projetos atuais torna-se bastante proibitiva, em termos de custo.

As próximas seções fazem considerações a respeito de desenvolvimento de jogos, como descrição de módulos presentes (como os representados nas figuras 3.1, 3.2 e 3.3), modelos de execução e alguns tipos de ferramentas disponíveis para combater a prática do "*not built here*".

3.1. Componentes de Jogos

Esta seção apresenta algumas características de módulos computacionais presentes em jogos de computador. Cada módulo, teoricamente, é responsável por uma determinada tarefa específica. Na prática (implementação), pode existir um grande grau de acoplamento entre eles, mas esse fator não é considerado nesta discussão. A figura 3.4 ilustra um diagrama que retrata os módulos apresentados neste capítulo.

Figura 3.4: Módulos computacionais em jogos discutidos neste trabalho.

3.1.1. Visualização

O módulo de visualização é responsável por expressar o estado do jogo de forma visual, seja em duas ou três dimensões (Hall, 2001). Junto com o módulo de áudio, formam a camada de apresentação da aplicação. Considera-se neste trabalho aspectos relacionados à visualização tridimensional.

Os desenvolvedores utilizam os serviços oferecidos pelo *hardware* gráfico através de APIs[4]. Atualmente, as duas APIs mais utilizadas para implementação de visualização tridimensional[5] em jogos de computador são *OpenGL* (Segal e Akeley, 2004) e *Direct3D* (DirectX, 2004). *OpenGL* foi desenvolvida inicialmente por *Silicon Graphics Inc*. (SGI) e atualmente é mantida pela ARB[6]. A ARB congrega vários representantes da indústria de *hardware* e *software* para visualização. O *Direct3D* é um produto desenvolvido pela *Microsoft Corporation*.

4 *Application Programming Interface*.
5 Embora sejam projetadas para visualização 3D, essas APIs também podem ser utilizadas para visualização 2D.
6 *Architectural Review Board* (http://www.opengl.org/about/arb).

Essas APIs oferecem algumas primitivas básicas (como pontos, linhas, triângulos, quadriláteros e polígonos), operações para manipulação do sistema de coordenadas e operações com matrizes (translação, rotação e escala) e efeitos como mapeamento de textura, entre outros comandos. Outra característica comum entre essas duas APIs é que elas são de modo imediato (*immediate mode*). As APIs de modo imediato são aquelas em que os comandos submetidos alteram o estado do *hardware* gráfico, assim que são recebidos. Dessa forma, ambas podem ser consideradas de baixo nível em termos de acesso ao *hardware*.

Segundo (Dalmau, 2003), o trabalho realizado pelo módulo de visualização pode ser dividido em duas partes: renderização de cenários (ambiente virtual) e renderização de personagens.

3.1.1.1. Renderização do Ambiente Virtual

Além do cenário propriamente dito, nesta etapa são renderizados elementos estruturais do ambiente virtual e objetos passivos. Objetos passivos são objetos estáticos e/ou que possuem animação simples.

É comum que jogos de computador utilizem cenários tridimensionais como parte de seus ambientes virtuais. A solução trivial a ser implementada seria simplesmente desenhar todos os objetos e polígonos[7] que compõem a cena. Entretanto, é comum que essa quantidade de dados seja muito grande para ser processada em tempo real pelos componentes de *hardware* existentes atualmente. Além do mais, essa solução é ineficiente porque conforme pode ser observado nesses tipos de aplicações, o usuário visualiza apenas uma parte do cenário (aquela que está dentro de seu campo de visão). Dessa forma, uma grande parte do esforço computacional aplicado é desperdiçado. Tal esforço poderia ser aplicado em outras partes da aplicação, como na Inteligência Artificial e nos cálculos para simulação física, por exemplo.

APIs como *OpenGL* e *Direct3D* oferecem apenas comandos para acessar serviços do *hardware*, não oferecendo nenhum tipo de ferramenta global em termos de cena. Por exemplo, todos os polígonos que compõem a cena são processados, mesmo que

7 Supondo que a cena seja modelada com polígonos, o que é bastante usual.

inicialmente estejam fora campo de visão do usuário. Os polígonos que estão fora do campo de visão do usuário são detectados apenas ao final do *pipeline* responsável pela renderização da imagem, quando é muito tarde.

Para resolver o problema da gestão de cena, é comum que sejam utilizadas técnicas de *culling*. *Culling* é o processo de eliminação de dados[8] que não contribuem para a imagem final (Valente, 2004a). Existem diversas técnicas de *culling*, como *backface culling*, *frustum culling* e *occlusion culling*.

O *backface culling* é aplicado em polígonos. Um polígono possui dois lados: o lado da frente e o lado de trás. Normalmente, o observador visualiza apenas um desses lados (o lado da frente, por exemplo). O *backface culling* é usado para descartar os lados e polígonos que não podem ser vistos pelo usuário. Esse processo está disponível em *hardware* atualmente.

O *frustum culling* é um teste de visibilidade feito entre o *frustum* (volume de visão) e objetos inteiros (compostos por vários polígonos). O *frustum* é um volume tridimensional em forma de uma pirâmide imaginária limitada pelos planos delimitadores *near* e *far*, que correspondem a distâncias relativas ao observador, na direção de sua linha de visão. O *frustum* é ilustrado na figura 3.5.

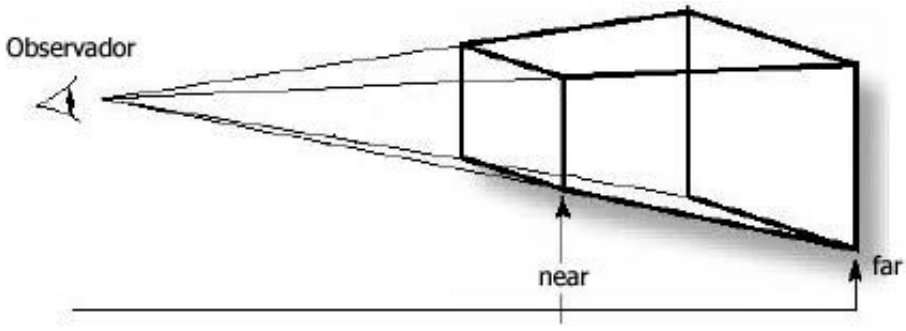

Figura 3.5: *Frustum* (Wright e Sweet, 1999).

Como esses objetos podem ter formas arbitrárias (e de diferentes complexidades), são usadas formas geométricas simples conhecidas como volumes delimitadores (exemplos: paralelepípedos e esferas) para aproximar o volume ocupado por um objeto. O objetivo de se usar formas geométricas simples é padronizar e facilitar o cálculo de visibilidade. O

8 Malhas de polígonos, polígonos inteiros, faces, etc, dependendo da técnica utilizada.

frustum culling também pode ser usado em combinação com algum tipo de estrutura hierárquica. A organização da cena em estruturas hierárquicas visa a descartar grandes quantidades de dados. Essas estruturas hierárquicas geralmente são algum tipo de árvore ou grafo, que representam relações espaciais entre seus elementos (embora isso não seja uma regra).

Durante a criação da hierarquia, os volumes delimitadores (esfera ou paralelepípedo, por exemplo) de cada nó são calculados. Posteriormente, o descarte de dados pode ser feito ao verificar se o volume delimitador de um nó intercepta o volume de visão. Caso não exista intercessão, o nó (e todos os seus filhos) podem ser descartados. Caso contrário, o nó (e seus filhos) estão visíveis, pelo menos parcialmente.

O *occlusion culling* é uma variação mais sofisticada de *culling* onde objetos que são encobertos por outros são eliminados do *pipeline* gráfico. Como um exemplo, um observador está em frente a um muro. Atrás do muro, existem diversos objetos complexos. Nesse exemplo, uma grande quantidade de processamento computacional pode ser poupada ao se descartar inicialmente todos os objetos que se encontram por trás do muro, já que estes não podem ser visualizados pelo observador.

Uma outra técnica que pode ser aplicada para reduzir a quantidade de dados a serem enviados ao *hardware* é a utilização de níveis de detalhe (LOD)[9] na geometria visível. Em projeções em perspectiva, quanto maior é a distância de um objeto em relação ao observador, menor é o seu tamanho (esse efeito é conhecido como *perspective foreshortening* (Foley et al, 1990)). Conseqüentemente, menor é o seu nível de detalhe. Entretanto, sem utilizar nenhum tipo de otimização, ao se enviar um objeto que contenha mil polígonos (por exemplo) para o *hardware*, esses mil polígonos serão processados independentemente do nível de detalhe que pode ser percebido pelo observador. Isso ocorre em APIs de baixo nível porque essas APIs não possuem noção de objetos e modelos, mas apenas de vértices e polígonos. A técnica de LOD consiste em representar um modelo geométrico por diversas instâncias ou malhas. Dependendo da distância do modelo para o observador, envia-se para a placa gráfica uma malha mais refinada ou mais simplificada.

9 *Level of detail.*

3.1.1.2. Renderização de Personagens

A renderização de personagens envolve a seleção daqueles que estão dentro do campo de visão do usuário (e que possivelmente participam da ação) e a aplicação de técnicas de animação.

A seleção dos personagens visíveis pode ser feita aplicando-se técnicas de *culling*, como foram descritas na seção anterior. Esse passo é importante porque pode evitar que objetos com animações complexas (portanto custosas) sejam processados sem necessidade.

De acordo com (Dalmau, 2003), as técnicas de animação usadas em jogos são divididas em dois grupos: métodos explícitos e métodos implícitos.

Métodos explícitos armazenam a geometria necessária para representar cada quadro que compõe a animação. A principal vantagem desses métodos é que a implementação tende a ser mais simples. A principal desvantagem é o custo de utilização de memória (principalmente se existirem muitos quadros com geometria complexa). Como exemplo, têm-se as técnicas de animação baseadas em quadro-chave[10] (*keyframe animation*). Nesse tipo de animação, somente os quadros-chave são armazenados e os outros são gerados através de interpolação.

Os métodos implícitos utilizam estruturas que descrevem a animação. Essas estruturas são usadas para gerar a animação em tempo real. Por exemplo, em sistemas de animação baseados em esqueleto (*skeletal animation*), são armazenadas as configurações (ângulos de rotação) de cada parte que une dois ossos (juntas). A principal desvantagem de métodos implícitos é o custo de utilização da CPU (por causa da complexidade dos cálculos). Por outro lado, esses métodos consomem pouca memória (em relação aos métodos explícitos). Outro atrativo dos métodos implícitos é que a animação pode ser adaptada ao cenário, segundo (Dalmau, 2003). Por exemplo, um personagem precisa andar em um terreno irregular. Se um método implícito for utilizado, é possível alterar a configuração da animação para que o personagem seja posicionado corretamente. Caso um método explícito fosse utilizado, seria necessário calcular previamente todos os quadros necessários para reproduzir essa animação.

10 São os quadros principais que definem uma animação (Watt e Watt, 1992).

Assim como na renderização de ambientes virtuais, também é possível aplicar técnicas de LOD para reduzir a complexidade da geometria e animação dos personagens.

3.1.2. Inteligência Artificial

A função da Inteligência Artificial (IA) em jogos é produzir um comportamento para os objetos controlados pela aplicação de modo que seja convincente ao usuário e que torne o jogo interessante. Segundo (Dalmau, 2003), os sistemas de IA podem ser basicamente de dois tipos: agentes e controladores abstratos.

Os agentes representam personagens que habitam o ambiente virtual do jogo. Podem ser estruturados por percepção, memória, análise para tomada de decisões e execução de ações.

A percepção corresponde à etapa onde é realizada uma pesquisa no ambiente por dados. Esses dados são utilizados em etapas posteriores do processamento de IA, como tomada de decisão. Por exemplo, um determinado personagem sempre foge quando está localizado a menos de dez metros do jogador. Dessa forma, a primeira etapa a ser realizada é verificar no ambiente a localização do jogador em relação ao personagem.

A memória é usada para armazenar resultados anteriores e melhorar a qualidade da tomada de decisão. Pode ser usada também na implementação do processo de aprendizagem do agente.

A análise para tomada de decisão corresponde à etapa onde os dados obtidos durante o processo de percepção e a memória (se houver) são usadas para analisar a situação atual e tomar alguma decisão. A velocidade de tomada de decisão depende da quantidade de alternativas disponíveis e dos dados obtidos na primeira etapa.

A execução de ações corresponde à etapa onde as ações que refletem as decisões tomadas são executadas. Esta etapa pode ser usada para criar "personalidades" para os agentes, fazendo com que ações com objetivo similar sejam executadas de maneiras diferentes.

Os controladores abstratos são usados para administrar um grupo de agentes. Podem ser usados, por exemplo, para implementação de táticas de grupo (*Team AI*).

A implementação de IA deve ser realista o bastante para representar algum desafio ao jogador, mas não "realista demais". De fato, o computador possui mais conhecimento que o jogador, por conhecer todos os dados do jogo em execução. Dessa forma, a implementação precisa levar em consideração alguns fatores para que a simulação seja, de certa forma, justa. Como um exemplo, em um jogo onde existem guardas patrulhando um determinado local por onde o personagem do jogador precisa passar, é razoável simular o campo de visão dos guardas para que só percebam a presença do jogador quanto este estiver dentro desse campo de visão.

Outros autores, como (Simpson, 2002), descrevem outros tipos de IA em jogos, como o chamado *emergent game play*. Esse termo é definido como a criação de algumas regras que farão partes do jogo, cuja aplicação pode resultar em situações não previstas inicialmente pelo desenvolvedor. Em essência, é apenas um conjunto de regras de um jogo, às quais o computador e o jogador irão obedecer, de modo que situações diversas serão criadas dependendo da interação entre computador, jogador e as regras. Um exemplo simples desse tipo de jogo é o jogo de xadrez. Outros exemplos mais complexos incluem jogos como *Black & White* e *The Sims*. Nesses jogos, existem regras básicas que são usadas para moldar o desenrolar da estória conforme a vontade do jogador (ou seja, sem existir um caminho único pré-determinado).

3.1.3. Redes

O módulo de redes é utilizado para conectar vários computadores de modo que vários jogadores possam participar da ação ao mesmo tempo. É responsável por informar aos outros computadores o estado do jogo, para que todos os jogadores possam estar sincronizados. Jogos em rede tipicamente utilizam redes locais (LAN)[11] ou a *internet*.

Segundo (Adams, 2003), existem duas arquiteturas de rede que podem ser implementadas: *peer-to-peer* e cliente/servidor.

Em uma arquitetura *peer-to-peer*, todos os computadores estão conectados diretamente uns aos outros. Dessa forma, um computador conhece todos os outros que estão conectados na rede. Nas sessões de jogo baseadas nessa arquitetura, cada máquina é responsável por

11 *Local area network.*

enviar seus dados para todas as outras máquinas na rede. Algumas das desvantagens de se utilizar esse modelo são:

- A dificuldade para manter a consistência (sincronismo) do jogo em todos os computadores;
- O alto número de conexões requeridas caso existam muitas máquinas na rede (o que acarreta um alto tráfego de dados na rede). O andamento do jogo também pode ser prejudicado caso alguns dos participantes possua uma conexão bem mais lenta que os demais.

Por esses motivos, essa arquitetura é indicada para redes pequenas.

No modelo cliente/servidor, todos os participantes (clientes) estão conectados a um computador central (servidor). O servidor é responsável por receber os dados de cada computador e repassar para os outros computadores da rede. Os clientes, a princípio, não possuem conhecimento sobre os outros computadores. Esse tipo de arquitetura possui uma série de vantagens, como:

- Redução do número de conexões na rede, pois cada cliente só precisa estar conectado a um único computador (o servidor). O único computador que necessita de uma conexão rápida é o servidor, para que este possa enviar os dados para cada cliente adequadamente;
- Melhor escalabilidade, ou seja, é possível aumentar a carga sobre o sistema sem prejudicar a qualidade de serviço percebida pelos jogadores (até um certo limite);
- A centralização da gestão do estado do jogo no servidor possibilita que seja feita uma validação dos dados enviados, o que é importante para combater trapaças nos jogos em rede.

A principal desvantagem desse tipo de arquitetura é que o servidor precisa ter alto poder de processamento e uma conexão veloz para suprir a demanda. Por possuir uma grande quantidade de jogadores simultâneos (da ordem de milhares), os chamados MMOG (*massively multiplayer online games*) utilizam a arquitetura cliente/servidor.

3.1.4. Simulação Física

A simulação física pode ser usada em jogos para:

- Modelar comportamentos de objetos que habitam o ambiente virtual (incluindo animações);
- Modelar interações de objetos entre si;
- Modelar interações entre objetos e o ambiente virtual;
- Simular a dinâmica de corpos rígidos.

A simulação das Leis da Física em jogos ajuda a reforçar a sensação de solidez do mundo virtual (Hecker, 1996). Esse efeito pode ser um estímulo sutil para melhorar o realismo do jogo.

Realismo aqui significa um comportamento similar (ou esperado) ao que ocorre no mundo real. O nível de realismo físico aplicado em jogos é variável. Um jogo pode apenas impedir que o jogador atravesse paredes e o chão, ou pode implementar interações de corpos humanos atingidos por projéteis com o cenário, de modo que as animações dependam da parte do corpo atingida e do ambiente que cerca o personagem, como é descrito em (Jacobsen, 2003). Outros jogos, como em (NBA Street, 2004), permitem ações impossíveis no mundo real.

De acordo com (Jacobsen, 2003), a implementação de métodos precisos para a simulação física não é um dos objetivos principais de jogos e aplicações interativas, devido ao custo de processamento requerido. Em vez disso, são priorizadas a execução eficiente dos algoritmos e aplicação de truques de programação para a obtenção do efeito final desejado (sem que o usuário perceba a diferença).

Um dos problemas mais comuns de simulação física em jogos é o tratamento de colisões. O tratamento de colisões pode ser dividido em duas partes: a detecção de colisão e a resposta à colisão (Policarpo e Conci, 2001).

A detecção de colisão pode ser feita entre objetos diversos ou entre um determinado objeto e o cenário. Esse processo consiste em um teste geométrico para verificar se existe

interseção entre os objetos testados. Geralmente, esses objetos são representados por malhas arbitrárias de polígonos. Com o intuito de simplificar (e agilizar) os cálculos do teste, o volume dos objetos é aproximado por formas geométricas mais simples como paralelepípedos ou esferas. De acordo com (Gomez, 1999), um problema que pode ocorrer é dois objetos se chocarem de fato, mas devido a um problema de amostragem de tempo, esse choque não ser detectado. Isso pode ocorrer porque no primeiro teste os objetos ainda não se chocaram, e no segundo já se chocaram mas encontram-se disjuntos novamente. Essa falha é ilustrada pela figura 3.6.

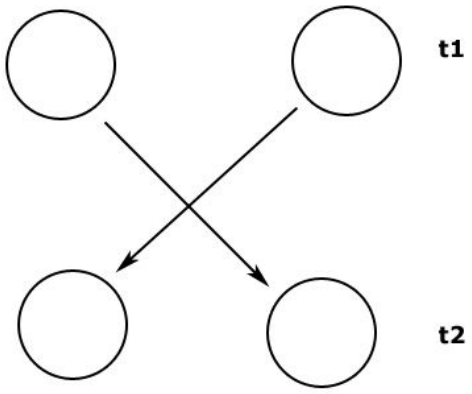

Figura 3.6: Falha na detecção de colisão.

A causa desse erro é devido ao intervalo de tempo entre dois testes consecutivos não ser pequeno o suficiente para detectar a colisão. A solução para esse problema é utilizar "testes de varredura" (*sweep tests*), que levam em consideração a posição atual e a posição gerada no passo anterior (em vez de utilizar só a posição gerada no passo atual).

A etapa de resposta à colisão é responsável por decidir que ações tomar no caso de existência de colisões. Essas ações estão relacionadas com o modelo físico simulado. Por exemplo, um jogo de sinuca implementa um modelo simplificado em que todas as colisões são totalmente elásticas. Dessa forma, duas bolas quaisquer terão suas velocidades trocadas entre si, ao colidirem. Outros tipos de jogos, como os de tiro em primeira pessoa, adotam uma política sutil de modo a manter a continuidade de movimentação do jogador. Quando o jogador colide com alguma parede do cenário, a direção da velocidade do jogador é alterada para que seja possível deslizar na parede e, conseqüentemente, continuar o movimento. Tal esquema é ilustrado na figura 3.7.

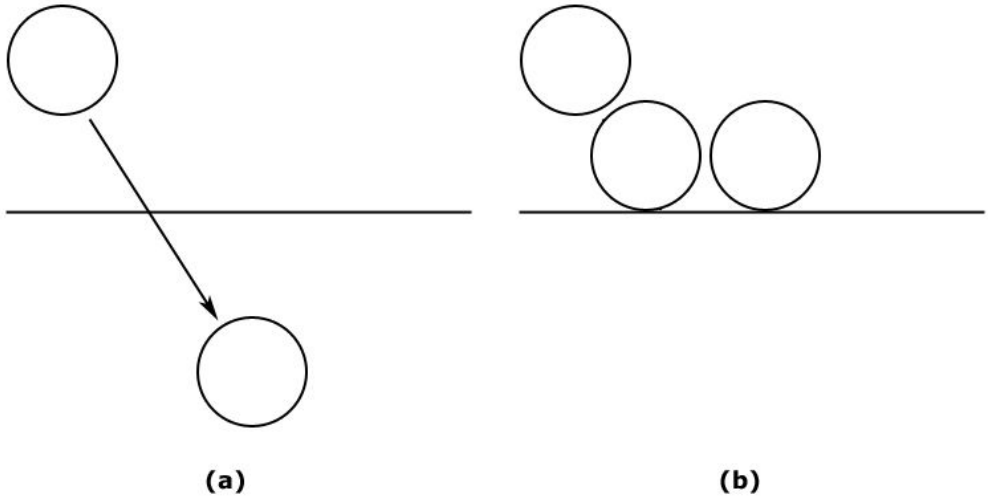

Figura 3.7: Tratamento da colisão. (a) Movimento desejado. (b) Movimento corrigido.

Um outro problema cuja abordagem está se tornando cada vez mais comum em jogos é a simulação da dinâmica de corpos rígidos. Esse tipo de simulação pode ser aplicado na busca pelo realismo de movimento e animação de figuras articuladas. Uma consequência interessante é que os personagens podem ser adaptáveis ao cenário onde eles se encontram, livrando o artista do trabalho de criar todas as sequências possíveis de animação que poderiam existir devido à interação do personagem com o cenário. Um outro exemplo é a simulação física de um veículo, que pode envolver variáveis como transmissão, motor, controle de direção, suspensão (molas, amortecedores) e aerodinâmica para definir o seu comportamento.

3.1.5. Áudio

A utilização de áudio em jogos resumia-se à sonorização: aplicação de efeitos sonoros e reprodução de músicas de fundo (estáticas). Efeitos sonoros consistem de pequenos arquivos de áudio de curta duração que são reproduzidos na ocorrência de algum evento específico. Por exemplo, quando um personagem pula, pode ser reproduzido um efeito sonoro para sinalizar esse evento. Músicas de fundo estáticas podem ser consideradas como melodias que são reproduzidas repetidamente, sem interrupção.

Segundo (Simpson, 2002), a importância de sons e música em jogos tem crescido nos últimos anos devido à evolução dos gêneros de jogos e da tecnologia. Uma das tendências é a utilização de som tridimensional.

Atualmente, vários jogos possuem grandes ambientes modelados com gráficos 3D, e o áudio tridimensional é utilizado como um estímulo sutil para melhorar a sensação de imersão do usuário na simulação. Segundo (Battaiola et al, 2001), o áudio tridimensional é a possibilidade de se colocar som em qualquer lugar em torno da cabeça do ouvinte.

O áudio tridimensional introduz conceitos como ouvintes (*listeners*) e fontes sonoras (*sources*). Um ouvinte representa o usuário no mundo tridimensional. Possui propriedades como posição no mundo, velocidade e orientação. Uma fonte sonora é uma entidade que emite som e está presente em algum lugar no ambiente tridimensional. As propriedades do ouvinte e das fontes sonoras são utilizadas para se gerar efeitos diversos. Por exemplo, ao combinar essas propriedades é possível criar efeitos sonoros que sofram atenuação conforme a distância em relação ao ouvinte varia, como o efeito Doppler[12]. A utilização de áudio tridimensional pode ser implementada através de APIs como *OpenAL* (OpenAL, 2000) e *DirectSound3D* (DirectX, 2004). *OpenAL* foi desenvolvida de modo a funcionar em vários sistemas operacionais, possuindo convenções e nomenclaturas semelhantes àquelas encontradas em *OpenGL*. O *DirectSound3D* é parte integrante do *DirectX*, desenvolvido pela *Microsoft Corporation*.

Em relação à música, uma das aplicações modernas é a chamada música dinâmica (*dynamic music*). Como descrito em (McCuskey, 2003), música dinâmica é a composição de uma trilha sonora complexa em tempo real, em resposta a eventos que ocorram no jogo. Por exemplo, o ritmo da música (ou o seu conteúdo) pode ser alterado de acordo com a ação presenciada pelo jogador. Uma música de ritmo lento poderia ser usada para situações de calmaria, enquanto uma outra de ritmo acelerado poderia ser usada em situações de tensão ou combate. Essa trilha sonora seria então construída à medida que essas ações ocorressem.

3.1.6. Dispositivos de Entrada

O módulo de dispositivos de entrada é um dos mais importantes pois sem ele não há interação com o usuário.

12 Aumento ou diminuição da percepção do som dependendo da distância relativa entre o ouvinte e a fonte sonora.

Os dispositivos mais comuns encontrados em computadores são o teclado e o *mouse*. Entretanto, existem outros tipos a serem considerados, como *joysticks*.

De acordo com (Dalmau, 2003), a utilização de *joysticks* começou na década de 70 como uma maneira de representar dados posicionais facilmente. Os primeiros modelos possuíam um botão e permitiam a representação de nove valores direcionais possíveis, sendo um para a posição neutra e os outros para ao sentidos norte, sul, leste, oeste, sudoeste, noroeste, sudeste e nordeste. Atualmente, os *joysticks* possuem diversas configurações e formatos, assim como quantidade variável de botões. Alguns autores como (LaMothe, 1999) chegam a considerar *joystick* como termo representativo para todos os dispositivos que não sejam teclado nem *mouse*.

Existem outros dispositivos que são projetados para um determinado tipo de jogo específico. Exemplos incluem manches (simuladores de vôo), volantes (simuladores de automóveis), e tapetes de dança (como existentes em *Dance Dance Revolution*), entre outros. A intenção do uso desses dispositivos é oferecer uma simulação de melhor qualidade. A figura 3.8 ilustra um dispositivo sob a forma de um volante.

Alguns dispositivos mais recentes possuem motores programáveis que são capazes de produzir vibrações e outros tipos de força. Esse tipo de dispositivo é conhecido como dispositivo *force-feedback*. Os efeitos podem ser aplicados como mais um fator para o aumento da imersão do usuário. Por exemplo, em um jogo de corrida, um volante equipado com *force-feedback* poderia ser programado para simular o efeito da inércia quando o jogador fizesse uma curva fechada.

Figura 3.8: Volante.

Segundo (Hall, 2001), uma das tarefas mais pertinentes de um sistema de dispositivos de entrada é acomodar a grande variedade de dispositivos existentes. Para que isso seja possível, pode ser implementada uma abstração para os dispositivos conhecida como controle virtual, como é descrito em (Dalmau, 2003) e (LaMothe, 1999). Esse tipo de implementação é orientada a comandos em vez de ser orientada a dispositivos (onde a aplicação precisa conhecer a origem dos dados). Sendo assim, considera-se que cada dispositivo é capaz de gerar determinados comandos, dependentes da aplicação (como "virar à esquerda", "pular", "encerrar o jogo"). Durante o processamento, os dados provenientes de vários dispositivos de entrada são coletados, convertidos e repassados ao programa. Tal esquema pode ser visualizado na figura 3.9.

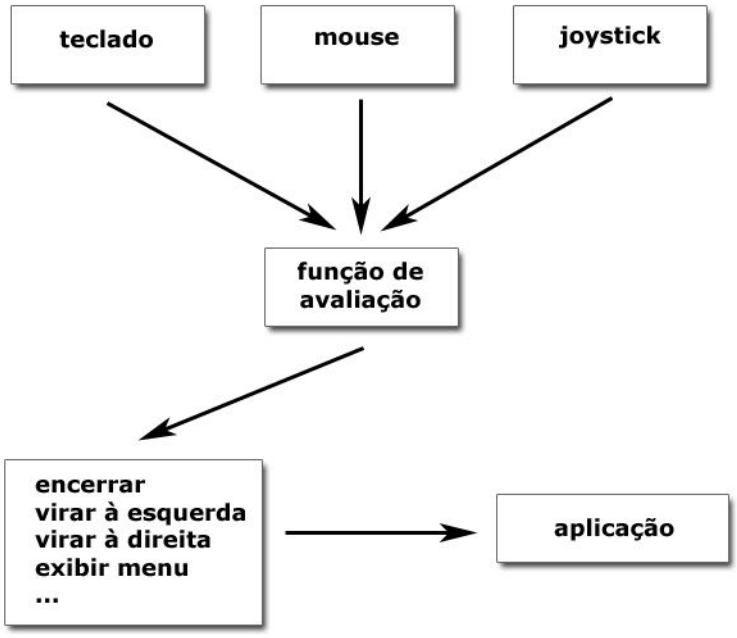

Figura 3.9: Controle virtual.

3.1.7. Scripts

Os *scripts* provêem uma maneira de modificar certas funcionalidades do jogo (ou definições de dados) sem precisar compilar novamente o código-fonte da aplicação, após as alterações. O objetivo é isolar essas partes, para que possam ser editadas independentemente. Os sistemas de *scripts* podem ser aplicados para a implementação de Inteligência Artificial, definição de dados (por exemplo, descrição de ítens usados pelo jogador em um RPG) e definição de configuração da aplicação, entre outras.

Os *scripts* consistem em programas escritos em alguma linguagem de programação específica, conhecida como linguagem para *script* (*scripting language*). Geralmente, a linguagem para *script* é diferente (e mais simples) da linguagem de programação usada no desenvolvimento do jogo (C++, por exemplo).

Os programas implementados em *scripts* são executados por módulos integrados ao programa principal. Esses módulos podem ser interpretadores ou máquinas virtuais. Os interpretadores recebem como entrada o código-fonte do script (como texto), geram uma representação compilada (conhecida como *bytecode*) e executam as instruções. As máquinas virtuais executam instruções diretamente a partir de *bytecodes*.

As principais vantagens de se usar linguagens para *script*, de acordo com (Dalmau, 2003), são:

- Os *scripts* podem ser executados em um ambiente isolado da aplicação (a máquina virtual). Dessa forma, é possível que um *script* mal feito (ou malicioso) seja desativado e/ou impedido de ser executado, o que contribui para aumentar a segurança da aplicação.

- O desenvolvimento da aplicação torna-se mais flexível, pois partes do programa controladas por *scripts* podem ser implementadas de maneira independente por pessoas que não estejam envolvidas com o desenvolvimento principal. Em outras palavras, essas pessoas não precisam requisitar que a aplicação seja compilada novamente, a cada alteração, para que os resultados possam ser conferidos.

A principal desvantagem é que existe um custo (*overhead*) relacionado com a interpretação dos *bytecodes* pela máquina virtual, o que pode ocasionar uma degradação de desempenho na aplicação.

Existem várias alternativas em relação ao uso de linguagens para *script*. O desenvolvedor pode optar por projetar e implementar uma linguagem específica para a sua aplicação, mas esse processo pode ser demorado e complexo. Atualmente, uma opção popular é utilizar uma linguagem embutida. Uma linguagem para *script* embutida (*embedded scripting language*) é uma linguagem de propósito geral, desenvolvida principalmente para ser integrada a outras aplicações. São linguagens de alto nível, que podem ser procedurais ou orientadas a objeto. Exemplos incluem Lua (Lua, 2004) e *Python* (Python, 2004).

De acordo com (Dalmau, 2003) e (Varanese, 2003), existem outros tipos de sistemas baseados em *scripts*, classificados como baseados em comandos, baseados em ligação dinâmica e baseados em *sockets*.

Os sistemas baseados em comandos possuem uma coleção de comandos específicos para um determinado programa. O *script* utiliza esses comandos pré-definidos, o que acarreta em uma grande especialização do sistema.

Os sistemas baseados em ligação dinâmica, essencialmente, são blocos de código nativo (em linguagem de máquina) compilados como um módulo de ligação dinâmica (como as DLLs[13], no *Windows*). Esse tipo de sistema utiliza linguagens de programação como C++. Os módulos são carregados dinamicamente conforme a necessidade de uso. Os jogos que utilizam esse tipo de sistema podem disponibilizar algum tipo de API para uso nesses módulos, possibilitando que estes controlem o jogo de alguma maneira.

A vantagem desse tipo de sistema é possuir o melhor desempenho, por ser compilado diretamente em código de máquina. A principais desvantagens são:

- Pouca segurança da aplicação. Como são compilados como DLLs, sua execução (e suas ações) não pode ser controlada facilmente pelo programa principal;
- A estruturação de módulos de ligação dinâmica é dependente de sistema operacional;
- A necessidade de se usar linguagens como C++, que pode ser problemática caso as pessoas responsáveis por implementar os programas em *scripts* não sejam programadores (como *game designers*, por exemplo).

Os sistemas baseados em *sockets* também utilizam linguagens de programação como C++. Nesse tipo de sistema, a comunicação entre o programa principal e o módulo de *script* é realizada através de *sockets*. Entre as vantagens dessa abordagem, incluem-se:

- Melhor controle sobre a segurança do sistema, já que o programa principal e o módulo de *script* são executados em processos diferentes;
- Como a comunicação é realizada através de *sockets*, esse tipo de sistema é independente de plataforma (já que *sockets* estão disponíveis em um grande número de plataformas);
- O módulo de *script* pode se compilado em código de máquina, o que contribui para um melhor desempenho.

Entre as desvantagens desse tipo de sistema, incluem-se:

13 *Dynamic Link Lybrary.*

- É necessário usar linguagens de programação como C++, que pode ser problemática caso as pessoas responsáveis por implementar os programas em *scripts* não sejam programadores especializados;

- A velocidade de comunicação por *sockets* pode não ser suficiente para tarefas com restrição de tempo;

- Gestão de um grande número de *scripts* pode ser problemática (por causa do alto número de *sockets*).

3.2. Jogos como Aplicações de Tempo Real

De acordo com (Silberchatz e Galvin, 1998), sistemas de tempo real são aqueles que possuem restrições de tempo para a execução de suas tarefas. Em outras palavras, a seqüência de aquisição de dados, processamento e apresentação do resultado não pode ultrapassar uma quantidade de tempo pré-definida.

De acordo com essa definição, jogos de computadores também podem ser considerados como aplicações de tempo real, interativas. O sistema de aquisição de dados corresponde aos dispositivos de entrada. A apresentação do resultado é feita através do vídeo e do sistema de áudio. O processamento relacionado ao jogo não pode ultrapassar uma quantidade de tempo máxima ou a interatividade proporcionada pelo programa será prejudicada. Um dos parâmetros utilizados para a medição do desempenho é a quantidade de quadros gerados por segundo (*frames per second* ou fps). Um quadro corresponde a uma imagem exibida no dispositivo de saída (o vídeo, por exemplo). O limite inferior comumente aceito é de cerca de 16 quadros por segundo. A freqüência considerada ideal situa-se entre 50 e 60 quadros por segundo.

3.2.1. Modelos de Execução

A princípio, conforme foi descrito na seção 3.2, jogos de computador possuem três etapas independentes: leitura de dispositivos de entrada, atualização do estado da aplicação e a apresentação dos resultados. A atualização do estado da aplicação envolve todo o processamento realizado com base nos dados dos dispositivos de entrada e nas regras do jogo.

A percepção do usuário é que todas as etapas ocorrem simultaneamente. Idealmente, essas etapas seriam executadas em paralelo, em vários processadores diferentes. Na prática, a maioria dos computadores atuais possui apenas um processador, sendo necessário organizar essas etapas de modo a simular o paralelismo.

A solução mais simples para esse problema é executar todas as etapas seqüencialmente (em um laço), como exemplificado pela figura 3.10.

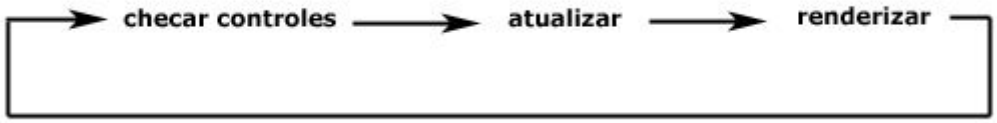

Figura 3.10: Modelo totalmente acoplado.

Essa solução é conhecida como método totalmente acoplado de acordo com (Dalmau, 2003) e (Rollings e Morris, 2003). Essa configuração poderia ser utiliza em plataformas com configuração fixa (consoles de *video-games*, por exemplo). Entretanto, no caso de computadores pessoais (PCs), essa solução é problemática, devido à variedade de configurações possíveis. Um computador com grande poder de processamento teria capacidade de executar o laço mais vezes. Dessa forma, o jogo seria executado mais rapidamente, mas não necessariamente isso seria uma melhoria. Por exemplo, em um jogo qualquer, um personagem poderia ser projetado para andar com velocidade igual a dois metros por segundo. Utilizando esse tipo de laço de execução, a velocidade efetiva do personagem dependeria da máquina onde a aplicação estivesse sendo executada. Outro problema existente é que se uma das etapas demorar muito tempo para ser executada, a degradação de desempenho será facilmente percebida pelo usuário (na forma de perda da suavidade de uma animação, por exemplo). Uma variação existente desse modelo (como observadas em (LaMothe, 1999) e (Rollings e Morris, 2003)) é sincronizar a execução do laço segundo uma determinada freqüência fixa, como ilustrado na figura 3.11.

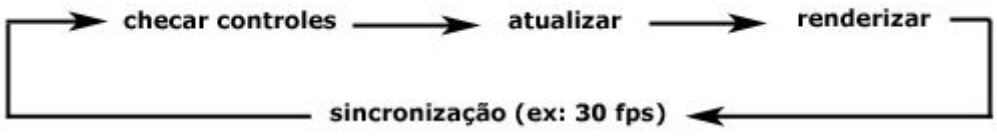

Figura 3.11: Sincronização do laço de execução.

Essa abordagem impõe um limite artificial ao programa. Embora o jogo possa ser executado da mesma maneira em máquinas com configurações diversas, essa não utiliza plenamente os recursos de computadores mais poderosos para oferecer uma melhor experiência ao usuário (ex: animações mais suaves).

A solução para esse problema é tornar as etapas de atualização e apresentação independentes entre si. Uma das alternativas é criar dois laços de execução para cada etapa, em *threads* diferentes, como ilustrado na figura 3.12.

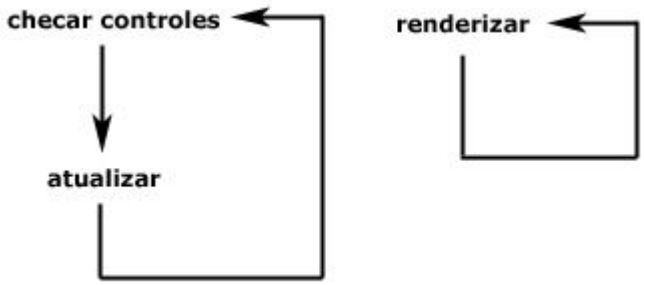

Figura 3.12: Execução em laços distintos.

A implementação em várias *threads* apresenta problemas típicos de programação concorrente, como sincronização de acesso aos dados. A etapa de apresentação necessita de acesso aos dados da aplicação para realizar o seu trabalho. Dessa forma, é preciso utilizar sincronização explícita entre elas para que não sejam apresentados resultados inconsistentes (por exemplo, desenhar um objeto cujo processamento ainda não terminou).

A execução da *thread* ocorreria com a maior freqüência possível. Para evitar problemas descritos anteriormente no modelo totalmente acoplado, a atualização da simulação deve ser feita utilizando-se o tempo decorrido entre a execução de dois laços consecutivos. Dessa forma, a atualização será feita corretamente em diversos tipos de computadores. A diferença será a qualidade dos resultados. Computadores mais velozes executarão o laço mais vezes, proporcionando animações mais suaves, por exemplo. Esse esquema também pode ser implementado em uma única *thread*, como ilustrado na figura 3.13.

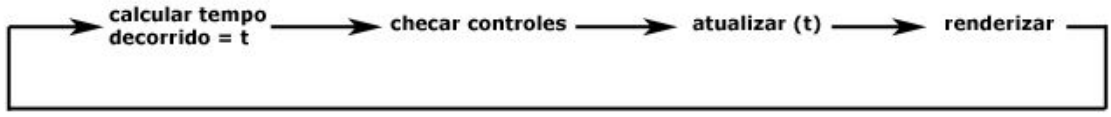

Figura 3.13: Modelo com uma *thread*.

Entretanto, existe um fator que precisa ser considerado. A etapa de atualização, como descrita até aqui, envolve todo o processamento capaz de alterar o estado do jogo. Isso inclui tipos de tarefas como execução da lógica do jogo (que pode realizar o processamento de Inteligência Artificial), atualização (interpolação) de animações e simulação do realismo segundo sob o ponto de vista físico, entre outras. Ao se executar a etapa de atualização (como um todo) o maior número de vezes possível, pode existir desperdício computacional. A razão é que tarefas como a execução da lógica do jogo, se for realizada muitas vezes, pode não apresentar uma diferença significativa (ao contrário de etapas como interpolação de animações). É necessário, então, dividir a etapa de atualização em pelo menos duas partes: uma responsável por tarefas que possam ser executadas com freqüência fixa (como Inteligência Artificial) e outra responsável por executar tarefas com a maior freqüência possível (como interpolação de animações). Um possível diagrama para essa configuração é ilustrado pela figura 3.14.

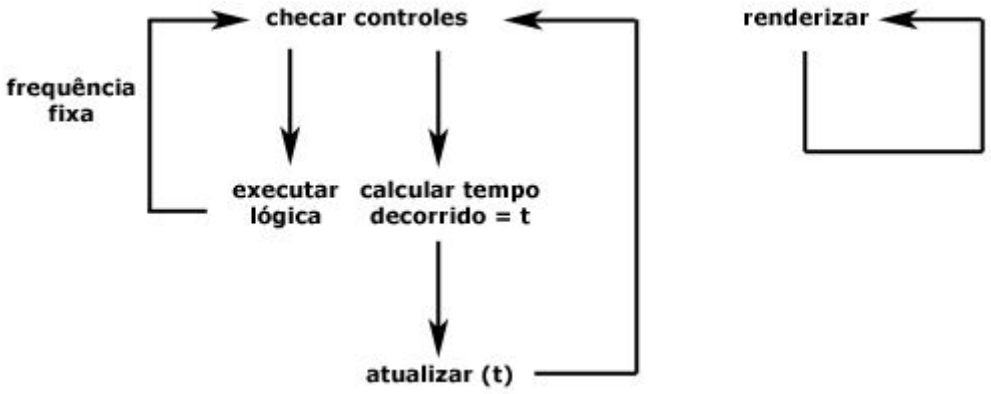

Figura 3.14: Atualização com duas etapas.

3.3. *Frameworks* e Outras Soluções para Jogos

Existem diversos tipos de soluções (em termos de reusabilidade) para desenvolvimento de jogos, como bibliotecas, *toolkits*, *frameworks* e *engines*. A diferença entre esses tipos de soluções é o nível de especialização apresentado por elas. É preciso ressaltar que a

distinção entre essas ferramentas é bastante controversa. Esta seção apresenta uma possível definição para esses tipos de ferramentas, com o objetivo de contextualizar a ferramenta desenvolvida neste trabalho. Na prática, é difícil separar de maneira precisa todos esses conceitos.

O uso de soluções como as descritas nesta seção têm se tornado mais relevante nos últimos tempos devido ao crescimento da complexidade de desenvolvimento de jogos. Os projetos de jogos atuais têm duração de 24 a 36 meses (Rollings e Morris, 2003). Alternativas que contribuam para reduzir esse tempo de desenvolvimento (e conseqüentemente o custo) são, sem dúvidas, benéficas.

Frameworks são definidos em (Gamma et al, 1995) como um conjunto de classes que formam uma estrutura reusável para um determinado tipo de aplicação. Um exemplo de *framework* para jogos é o *RealmForge GDK* (RealmForge, 2005).

Bibliotecas são coleções de rotinas projetadas para oferecer algum tipo de funcionalidade (como a API do *OpenGL*, por exemplo).

Segundo (Gamma et al, 1995), *toolkits* são um conjunto de classes relacionadas e reusáveis, projetadas para oferecer funcionalidades de propósito geral. Por exemplo, a biblioteca padrão do *C++* (STL[14]) pode ser considerada como um *toolkit*, segundo essa definição.

O exemplo anterior de *toolkit* (STL) é denominado biblioteca. Esse exemplo reflete um fato que é bastante comum: existem contradições e confusões em relação ao uso desses termos (incluindo *frameworks* e *engines*).

A diferença entre *toolkits* e bibliotecas é que os *toolkits* são projetados segundo o paradigma de orientação a objetos. *Toolkits* e bibliotecas não impõem uma estrutura (ou arquitetura) à aplicação, sendo apenas ferramentas que podem ser usadas na resolução de um dado problema. Ambos promovem reuso de código.

Ao contrário de bibliotecas e *toolkits*, os *frameworks* determinam a estrutura de um sistema, oferecendo um ponto de partida para o desenvolvimento de aplicações. Um

14 *Standard Template Library.*

framework fornece um esqueleto do sistema ao dividir o projeto em classes abstratas, definindo suas responsabilidades e colaborações. Esse esqueleto freqüentemente possui pontos de personalização (conhecidos como *hot spots*), onde as funcionalidades específicas de uma aplicação podem ser definidas. A concretização de uma determinada aplicação é realizada, também, por herança e composição de instâncias de classes do *framework*. Os *frameworks* promovem o reuso de projeto.

Um *game engine* é um programa que implementa uma determinada arquitetura para jogos. Esse programa aceita determinados dados e configurações (conteúdo) que determinam o aspecto do jogo.

A principal diferença conceitual entre *frameworks* e *game engines* é que os últimos representam aplicações concretas.

A figura 3.15 apresenta uma possível relação entre os tipos de ferramentas descritos nesta seção.

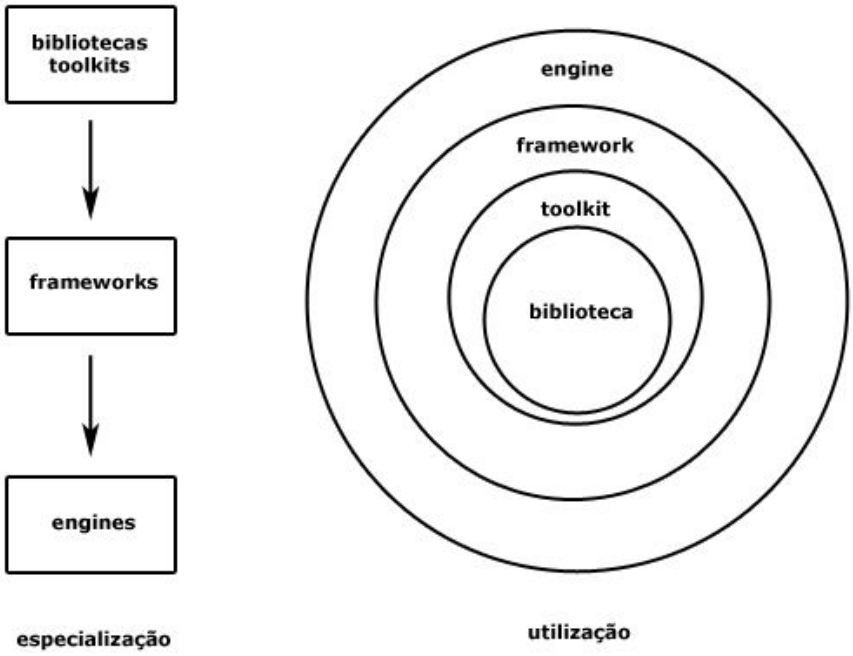

Figura 3.15: Relações de especialização e utilização.

Como é sugerido na figura 3.15, um *game engine* pode ser construído a partir de um *framework*, que por sua vez pode ser projetado com aplicação de vários *toolkits* e bibliotecas.

Os *game engines* promovem o reuso de aplicação, mas por serem altamente especializados, muitas vezes só podem ser usados para construir um determinado tipo de jogo. Isso ocorre porque é comum que os estilos de jogos possuam diferentes necessidades (em termos de implementação). Por exemplo, jogos de estilo FPS possuem uma política de gestão de cena diferente de jogos que se passam em cenários abertos. Exemplos de *game engines*: *Fly3D* (Fly3D, 2004) e *engines* dos jogos *Quake* (Quake, 2004) e *Unreal* (Unreal, 2004).

Os *game engines* são, essencialmente, orientados a dados (*data-driven*). Esses dados podem estar na forma de conteúdo (arte, imagens, arquivos de áudio, etc) ou extensões de funcionalidades (*plugins*). O conceito de projetos orientados a dados está ligado diretamente aos conceitos de *hard architecture* e *soft architecture* (Rollings e Morris, 2003).

A *hard architecture* representa partes de um projeto que podem ser reutilizadas em outros projetos. Basicamente, corresponde à infra-estrutura da aplicação, incluindo subsistemas que lidam com o *hardware* e bibliotecas (como *OpenGL* e *DirectX*), entre outros. A *hard architecture* pode ser um conjunto de componentes ou uma solução completa (no caso de um *game engine*).

A *soft architecture* representa partes do projeto que são específicas do jogo, como a implementação do *gameplay*. O *gameplay* representa a união de regras, objetivos, estratégias, mecânica e outros aspectos que definem o jogo. Geralmente, a *soft architecture* é única para cada jogo, não sendo reutilizável.

Essas duas divisões, embora assim definidas, não são totalmente independentes entre si. Em geral, a *soft architecture* é responsável por manipular os dados do jogo e utiliza os serviços oferecidos pela *hard architecture* (o contrário não deve ocorrer).

A premissa dos projetos orientados a dados é retirar do código-fonte ítens que requeiram refinamento constante (como aqueles relacionados ao *gameplay*). Dessa forma, o ideal é isolar essas partes em uma determinada camada (como a *soft architecture*).

A motivação para esse tipo de abordagem é que certos aspectos do jogo (como o *gameplay*, por exemplo) necessitam de vários refinamentos até atingirem um nível

satisfatório. O *gameplay* é um dos aspectos principais de um jogo, pois é um grande responsável por seu sucesso ou fracasso. Ao embutir esses aspectos diretamente no jogo, será necessário recompilar o programa para conferir os resultados. Dependendo da complexidade do programa, esse processo pode ser bastante demorado, o que contribui para diminuir a produtividade. Outro fator a ser considerado é que comumente as pessoas que são responsáveis por definir esse tipo de funcionalidade não são programadores.

O que pode ser feito, então, é oferecer ferramentas para que a edição dessas funcionalidades seja conveniente ao usuário. Dessa forma, os dados alterados podem ser lidos pelo programa principal sem a necessidade de modificá-lo (e sem a intervenção de um programador), tornando o desenvolvimento mais ágil. Por exemplo, vários *game engines* possuem editores de fase. Uma fase representa a geometria do cenário e suas propriedades (que são específicas do jogo). A razão para isso é que os *game engines* freqüentemente armazenam esses dados em um formato proprietário. Por exemplo, o *engine* do jogo *Quake III* (Quake 3, 2004) possui um editor próprio chamado *Q3Radiant* (Jaquays, 2000). Esse editor permite definir a geometria que compõe o cenário, definir regiões com as quais os objetos do jogo podem colidir, definir *triggers*[15], posições iniciais do jogador, entre outras.

Como construir um editor próprio pode ser uma tarefa demorada e complexa, alguns *game engines* (como *Fly3D*) optam por estender ferramentas já existentes. A criação da geometria do cenário é feita através do *3ds max* (3dsmax, 2004), um programa para animação e modelagem tridimensional. A seguir, essa cena é armazenada em um formato proprietário definido pelo *Fly3D*. As propriedades da cena (e dos objetos que fazem parte dela) podem ser alteradas posteriormente em um outro editor que é parte integrante do conjunto de ferramentas oferecidas pelo *Fly3D* .

3.4. Conclusão do Capítulo

Jogos para computador permitem a aplicação de conceitos de diversas áreas da Computação. Neste capítulo, foram apresentados alguns módulos como visualização, Inteligência Artificial, redes, simulação física, áudio, dispositivos de entrada e s*cripts*.

15 São entidades ou objetos (muitas vezes invisíveis) que podem gerar algum tipo de evento (caso o jogador colida com eles, por exemplo).

Os primeiros jogos de computador eram implementados com uso de linguagens de programação de baixo nível, como *assembler*. Isso era necessário devido às limitações de *hardware* existentes na época. Mesmo com o avanço das linguagens de programação e do *hardware*, esse costume perdurou por vários anos. A principal alegação para essa prática era de que a implementação deveria ser a mais eficiente (otimizada) quanto o possível, ou o desempenho apresentado pelo programa não seria suficiente para manter um nível satisfatório de interatividade. Por essa razão, tornou-se comum que os desenvolvedores de jogos implementem novamente todas funcionalidades necessárias para se criar um jogo, a cada projeto. Essa prática é conhecida também como "*not built here*" (Rollings e Morris, 2003). Entretanto, com o aumento da complexidade dos projetos de jogos, esse tipo de prática tem se tornado inviável.

As bibliotecas, *toolkits*, *frameworks* e *engines* são formas de promover o reuso de funcionalidades existentes, em diferentes níveis de especialização. Este capítulo apresentou uma possível conceituação para esses tipos de ferramentas. Existe bastante contradição e polêmica em relação à definição e ao uso desses termos. Na prática, existe uma grande superposição entre eles.

4. *Framework* Guff

Este capítulo apresenta o *framework* Guff (games-uff), que foi desenvolvido para a aplicação de alguns dos conceitos pesquisados sobre desenvolvimento de jogos.

O *framework* possui duas partes principais: a camada de aplicação e um *toolkit*. A camada de aplicação determina as interfaces e a arquitetura dos programas que usam o *framework*. O *toolkit* representa um conjunto de classes auxiliares que podem ser usadas para desenvolver os programas.

Em linhas gerais, o *framework* Guff foi projetado para atender aos seguintes requisitos:

- Oferecer uma interface que modele uma possível arquitetura genérica de um jogo;
- Oferecer um conjunto de ferramentas para auxiliar no desenvolvimento dos programas;
- Funcionar em *Windows* e *Linux*;
- Gestão automática de todos os recursos;
- Facilidade de uso;
- Reuso de bibliotecas já existentes;
- Possibilitar que a configuração de parâmetros da aplicação seja determinada fora do código fonte.

O *framework* foi escrito em C++ com o auxílio de diversas bibliotecas. As seções deste capítulo discutem esses requisitos e apresentam as soluções oferecidas pelo *framework*.

4.1. Bibliotecas

Este trabalho reúne diversas bibliotecas para evitar a reimplementação de funcionalidades já disponíveis e cumprir o requisito de portabilidade. As bibliotecas usadas neste projeto estão resumidas na tabela 4.1. Essas bibliotecas são livres e estão disponíveis para vários sistemas operacionais.

Nome	URL
OpenGL	http://www.opengl.org
GLEW	http://glew.sourceforge.net
boost	http://www.boost.org
lib3ds	http://lib3ds.sourceforge.net
audiere	http://audiere.sourceforge.net
FTGL	http://homepages.paradise.net.nz/henryj/code/
Lua	http://www.lua.org
DevIL	http://www.imagelib.org
SDL	http://www.libsdl.org

Tabela 4.1: Bibliotecas usadas pelo *framework* Guff.

A SDL (*Simple DirectMedia Layer*) é uma biblioteca multimídia que oferece acesso de baixo nível a dispositivos de entrada, áudio e sistema de janelas, entre outros. Seu código é aberto e existem implementações para vários sistemas operacionais.

OpenGL é uma biblioteca para modelagem e visualização tridimensional em tempo real, que está disponível para vários sistemas operacionais. Foi desenvolvida inicialmente pela *Silicon Graphics* em 1992, e hoje é um dos padrões da indústria de Computação Gráfica.

A biblioteca GLEW (*OpenGL Extension Wrangler*) destina-se a auxiliar desenvolvedores na inicialização e uso das extensões oferecidas pelo *OpenGL,* de maneira automática. Foi desenvolvida por Milan Ikits e Marcelo Magallon.

A biblioteca *boost*, que foi criada originalmente por membros do Comitê de Padronização do C++, representa um conjunto de várias bibliotecas desenvolvidas por diversos colaboradores. Um de seus objetivos é estabelecer referências para o desenvolvimento de bibliotecas, de modo que seja possível integrá-las à implementação padrão do C++. Provê soluções para gestão de memória, programação concorrente, entrada e saída, programação genérica e Matemática, entre outras.

A *lib3ds* destina-se a manipulação de arquivos (formato 3ds) gerados originalmente pelo *software 3ds max*. Foi desenvolvida inicialmente por J. E. Hoffman.

A *audiere* é uma biblioteca para realização de operações de entrada e saída, decodificação e mixagem de áudio, desenvolvida inicialmente por Chad Austin.

A *FTGL* é uma biblioteca que foi desenvolvida por Henry Maddock cujo objetivo é possibilitar o uso de vários tipos de fonte de texto em aplicações que usem *OpenGL*.

Lua é uma linguagem para *script* (*scripting language*) projetada para estender aplicações. Pode ser usada como uma linguagem de propósito geral e também pode ser embutida nos programas (de modo a oferecer uma interface para *scripts* e trocar informações com a aplicação hospedeira). Foi projetada e desenvolvida por Roberto Ierusalimschy, Waldemar Celes e Luiz Henrique de Figueiredo no laboratório Tecgraf da PUC-Rio.

A *DevIL* (*Developer's Image Library*), desenvolvida por Denton Woods, é utilizada para manipulação de imagens em diversos formatos de arquivos.

4.2. Gestão Automatizada de Recursos

Um recurso pode ser definido como alguma coisa que exista no sistema operacional em quantidade limitada, de modo que para usá-lo é preciso fazer uma requisição ao sistema e ao término do uso é preciso devolvê-lo ao sistema (Valente e Conci, 2004).

A gestão de recursos é uma das tarefas mais corriqueiras que devem ser realizadas em uma aplicação. Tradicionalmente, a realização dessa tarefa é de responsabilidade do desenvolvedor. Caso essa tarefa não seja realizada corretamente, vários problemas podem ser manifestar, como perda de recursos, *deadlocks*, "*bugs* aleatórios" (um defeito difícil de se reproduzir), entre outros. Para apreciar essas consequências, têm-se este trecho de código:

```
void foo () {

  int * p = new int;
        p = new int;
}
```

Nesse trecho de código, dois blocos de memória são perdidos. O paradigma da gestão automatizada de recursos é uma alternativa para resolver esse tipo de problema, sendo aplicada extensivamente na implementação deste trabalho.

A gestão de recursos pode ser automatizada usando propriedades da linguagem C++, como escopo em blocos de código, construtores e destrutores de classes. Os construtores e destrutores são métodos especiais das classes cuja execução é garantida pelo projeto da linguagem C++. O construtor é executado na criação de uma instância de uma classe e o destrutor é executado quando a instância é destruída. Para qualquer classe utilizada em um programa, têm-se os seguintes casos:

- Um objeto declarado em um bloco de código será criado (terá o seu construtor executado) quando o fluxo de programa entrar no escopo do bloco. Quando o fluxo de programa deixar o bloco, o objeto será destruído (terá o seu destrutor executado);

- Um objeto A é declarado como campo de um outro objeto B. Nesse caso, o objeto A será construído quando o objeto B for contruído. Da mesma forma, o destrutor de A será executado pelo destrutor de B, quando este for destruído;

- Um objeto é declarado como uma variável ou constante global. A instância do objeto será criada no início da execução do programa e será destruída ao término da aplicação.

Ao utilizar os conceitos de construtores e destrutores combinados com o conceito de escopo, é possível gerir automaticamente os recursos seguindo-se a seguinte regra: alocar recursos nos construtores e devolver os recursos correspontes nos destrutores. Essa regra é definida em (Milewski, 2001) como Regra Primeira de Aquisição e como "*resource acquisition is initialization*" em (Stroustrup, 1997). Dessa forma, os compiladores podem automatizar o processo de gestão de recursos.

Um dos conceitos principais em gestão automatizada de recursos é a definição de propriedade sobre um recurso. O proprietário de um recurso é aquele que é responsável por sua devolução ao sistema. Um proprietário sobre um recurso também é conhecido como objeto automático.

Como exemplo, uma classe genérica projetada para a gestão de um recurso hipotético poderia ser declarada da seguinte forma (pseudo-código):

```
class AutoResource
{
  public:
     AutoResource () { Acquire (); }
    ~AutoResource () { Release (); }

  private:
    resource _r; // o recurso real
};
```

Entretanto, existe ainda um outro problema, que é a transferência de recursos. O exemplo a seguir pode ilustrar esse problema:

```
AutoResource r1;
AutoResource r2;
r2 = r1;
```

Nesse caso, para garantir que não existam recursos perdidos, o recurso alocado por `r2` deve ser devolvido ao sistema, e o recurso alocado previamente por `r1` deve ser transferido para `r2`. Nesse contexto, a operação de atribuição torna-se uma operação de transferência de recurso. Dessa forma, a propriedade sobre o recurso é garantida. Isso poderia ser implementado da seguinte forma:

```
class AutoResource
{
  public:
     AutoResource () { Acquire (); }
    ~AutoResource () { Release (); }

    void operator = (AutoResource & obj)
    {
       Release ();
       _r = obj.TransferResource ();
    }

     AutoResource (AutoResource & obj)
       : _r (obj.TransferResource () )
       {}

  private:
    resource  TransferResource ()
```

```
    {
      resource  tmp = _r;
      _r = referência nula;
      return tmp;
    }

   resource _r;   //  o recurso real
};
```

No exemplo anterior, “referência nula” quer dizer um valor neutro, como 0 seria caso o recurso gerido fosse memória.

A gestão automatizada de recursos também pode ser usada para o compartilhamento de recursos. Por exemplo, isso poderia ser implementado utilizando-se contadores de referências. Nesse caso, além do recurso a ser compartilhado, seria necessário gerir também o uso do contador de referências.

É importante observar que esse paradigma também funciona na presença de exceções geradas pelo programa. Isso se deve ao fato de que os destrutores dos objetos locais (automáticos) também são executados depois que o fluxo de execução deixa um bloco de código, em virtude de uma geração de exceção. Sendo assim, é possível realizar a gestão de recursos corretamente, mesmo que erros inesperados ocorram na aplicação.

4.3. Camada de Aplicação

A camada de aplicação é modelada como uma máquina de estados. A motivação para esse modelo é que um jogo pode ser decomposto em um conjunto de estados. A figura 4.1 ilustra um possível máquina de estados para um jogo hipotético, que possui uma apresentação, um menu principal, a parte principal (o jogo em si) e um menu especial que pode ser acessado dentro do jogo.

A idéia proposta neste trabalho tem como objetivo tratar como um estado da aplicação cada uma dessas etapas, de modo a facilitar a especificação e gestão delas.

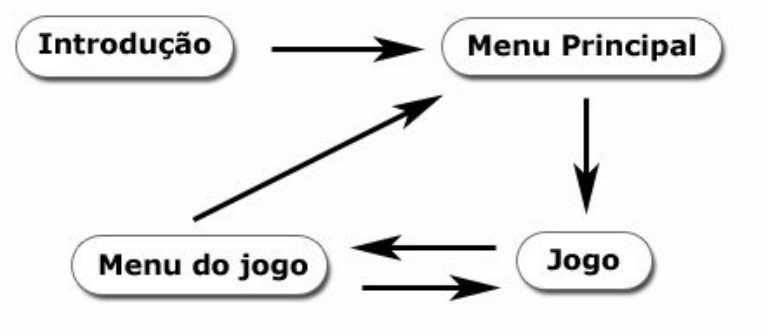

Figura 4.1: Representação das etapas de um jogo por estados.

O diagrama de classes para esta camada é ilustrado na figura 4.2. Esse modelo permite a existência de estados simples e estados compostos (grupos de estados).

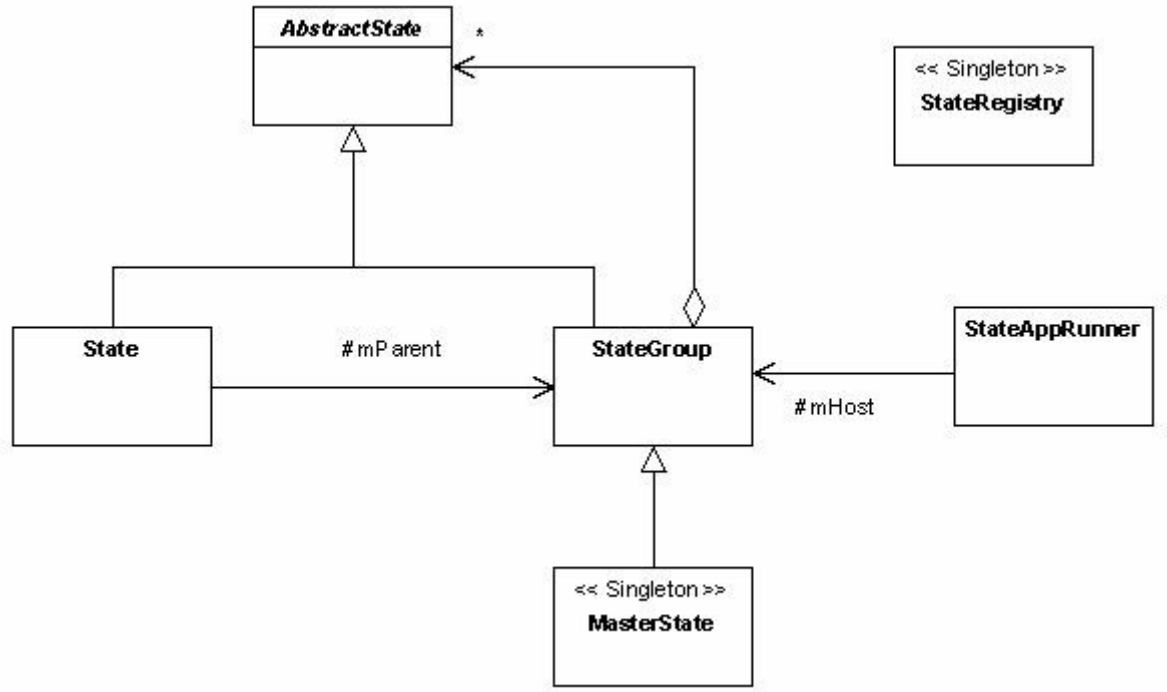

Figura 4.2: Diagrama de classes para a camada de aplicação.

Os estados simples são representados pela classe `State`, enquanto que os grupos de estados são representados pela classe `StateGroup`. Todos os estados possuem nomes que os identificam no *framework*. Por razões que ainda serão esclarecidas neste capítulo, esses nomes devem ser únicos.

Para facilitar o uso do *framework*, as associações entre estados pais e filhos são realizadas automaticamente. Todos os estados (simples ou compostos) definidos pela aplicação possuem um pai. O *framework* possui um estado, chamado de estado mestre, que

é o pai de todos os estados, por padrão. Esse estado é representado pela classe `MasterState`. Como exemplo, o seguinte trecho de código define um estado simples:

```
class Estado1 : public State
{
  public:
     Estado1 ()
     : State ("estado1") {}

  ...
};
```

Nesse exemplo, é declarado um estado cujo nome é "estado1". A hierarquia definida nesse exemplo é ilustrada na figura 4.3.

Figura 4.3: Hierarquia para estado1.

Caso se queira especificar um outro pai, é preciso informar seu nome, como neste trecho de código:

```
class Estado2 : public State
{
  public:
     Estado1 ()
     : State ("estado2", "pai") {}

  ...

};
```

Para que isso funcione, é preciso que o estado pai de `e2` tenha sido criado previamente. Esse problema pode ser resolvido facilmente usando composição de objetos. Dessa forma, quando o estado pai for criado, seus filho serão criados automaticamente. Uma hierarquia para esse exemplo pode ser representada pela figura 4.4.

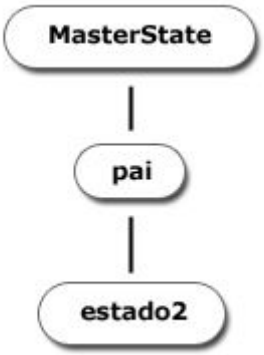

Figura 4.4: Hierarquia resultante.

Da mesma forma que os estado simples, os estados compostos possuem `MasterState` como pai, por padrão. Para estabelecer um novo pai, basta especificar o nome do grupo correspondente, conforme foi demonstrado no exemplo anterior.

4.3.1. Detalhes

O diagrama de classes dos estados foi modelado utilizando-se uma combinação de padrões de projeto. Padrões de projeto descrevem soluções para problemas que ocorrem repetidamente em sistemas diversos. Essas soluções são descritas de forma que possam ser reutilizadas em vários sistemas diferentes (Valente e Conci, 2004).

Os padrões de projeto aplicados nesse modelo foram *State* e *Composite*. Segundo (Gamma et al, 1995), o padrão *State* possui o seguinte propósito: Permitir que um objeto altere o seu comportamento quando seu estado interno sofrer mudanças. O objeto aparentará ter trocado de classe. O padrão *Composite* tem o seguinte propósito, de acordo com (Gamma et al, 1995): compor objetos em hierarquias de árvore. O padrão *Composite* permite que os clientes tratem os objetos individuais e os objetos compostos de maneira uniforme.

O padrão *State* é utilizado para permitir que cada estado do jogo seja representado por um objeto específico. Já o padrão *Composite* é aplicado para que a máquina de estados trate os estados individuais e os estados compostos sem distinção.

A interface comum para todos os estados é definida pela classe `AbstractState`. Essa classe é abstrata, portanto, não pode ser utilizada diretamente. O diagrama de classes para `AbstractState` é representado na figura 4.5.

Figura 4.5: Diagrama de classes para `AbstractState`.

O método `getId ()` permite que se recupere o nome do estado. Os outros métodos dessa classe correspondem a eventos que podem ocorrer na aplicação. Esses eventos podem ser de três tipos: eventos do sistema, eventos de estado e eventos do ciclo principal de execução.

Os eventos relacionados a estados são os seguintes:

- `OnInit`: É executado na inicialização da aplicação, para todos os estados. Seu propósito é permitir que os estados inicializem suas estruturas de dados e/ou executem tarefas necessárias para o seu processamento.

- `OnShutDown`: É executado ao término da aplicação, para todos os estados. Seu propósito é permitir que os estados liberem recursos alocados previamente. Como o *framework* possui gestão automática de recursos, normalmente não é necessário tratar esse evento.

- `OnEnter`: É executado para um estado qualquer quando este passa a ser o estado atual.
- `OnExit`: É executado quando o estado atual deixa de ser o corrente.

Os eventos do sistema que podem ser tratados são os seguintes:

- `OnWindowResize`: É executado quando a janela da aplicação sofre alteração de tamanho.
- `OnMouseDown`: É executado quando o usuário pressiona algum botão do *mouse*.
- `OnMouseUp`: É executado quando o usuário deixa de pressionar algum botão do *mouse*.
- `OnMouseMove`: É executado quando o usuário movimenta o *mouse*.
- `OnKeyDown`: É executado quando o usuário pressiona alguma tecla do teclado.
- `OnKeyUp`: É executado quando o usuário libera alguma tecla que estava pressionada.
- `OnActivate`: Executado quando a aplicação torna-se ativa (recebe o foco)
- `OnDeactivate`: Executado quando a aplicação torna-se inativa (por exemplo, quando a janela da aplicação é minimizada).

Para que seja possível descrever os eventos do ciclo principal de execução, é preciso entender como esse ciclo é implementado.

4.3.1.1. Execução da Aplicação

A aplicação é executada pela classe `StateAppRunner`. Um diagrama de classes que destaca esse componente é representado pela figura 4.6. As principais responsabilidades dessa classe são a execução do ciclo principal de jogo e a delegação do tratamento de eventos da aplicação.

Figura 4.6: Diagrama de classes para `StateAppRunner`.

A classe `StateAppRunner` possui apenas um método público, `go ()`, que é usado para iniciar a execução da aplicação. Esse método realiza três tarefas, nesta ordem: inicialização, execução do ciclo principal de jogo e finalização.

Na primeira etapa são realizadas a inicialização dos subsistemas e bibliotecas usadas pelo Guff. Na última etapa pode ser é possível devolver recursos ao sistema explicitamente, caso seja necessário.

O ciclo principal de jogo é implementado por um modelo com atualização em duas etapas (como exemplificado no capítulo 3), usando uma única *thread* para as operações de atulização e renderização. A figura 4.7 ilustra esse modelo.

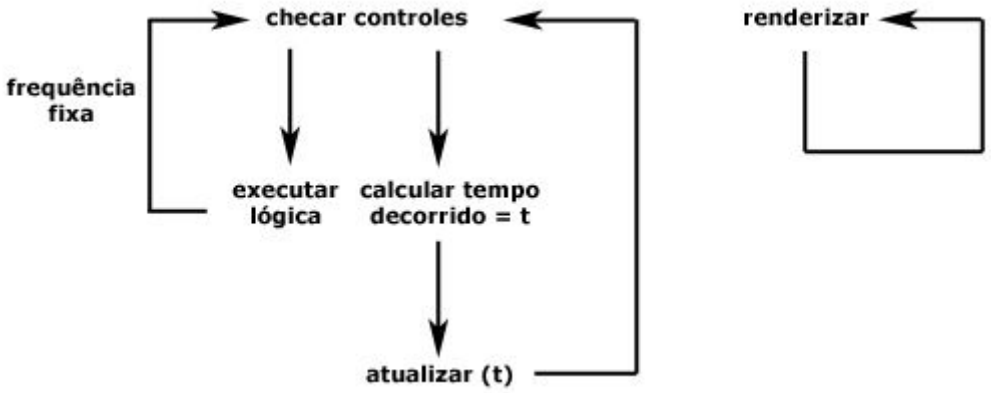

Figura 4.7: Modelo com atualização em duas etapas.

O ciclo é executado até que seja feita uma requisição para o término da aplicação. O objeto `mHost` (figura 4.6) delega o tratamento dos eventos para o estado atual da aplicação. Existem três eventos no ciclo principal de jogo que podem ser tratados:

- `OnFixedFrequencyUpdate`: É executado quando é necessário realizar a etapa de atualização de freqüência fixa. Esse tipo de evento pode ser desabilitado. A variável `mFixedFrequencyUpdateStep` representa o período de execução dessa etapa. Por exemplo, caso se queira que essa etapa seja executada 20 vezes por segundo, o valor de `mFixedFrequencyUpdateStep` corresponderá a 50 (milisegundos). A freqüência de execução desse evento é a mesma para todos os estados, sendo determinada por um arquivo de configuração.

- `OnUpdate`: Representa a etapa de atualização sem restrições de tempo. O parâmetro desse evento corresponde ao tempo decorrido desde a última execução dessa etapa, em segundos.

- `OnRender`: Representa a etapa onde são realizadas as operações de renderização. Ao final do ciclo ocorre automaticamente o *double-buffering*.

4.3.1.2. Registro de Estados

O registro dos estados no *framework* é realizado automaticamente com o uso da classe `StateRegistry`, que é um repositório de estados. Essa classe é implementada segundo o padrão de projeto *Singleton*. O padrão de projeto *Singleton* é usado para assegurar que exista somente uma única instância de uma classe em uma aplicação. A figura 4.8 ilustra o diagrama de classes para `StateRegistry`.

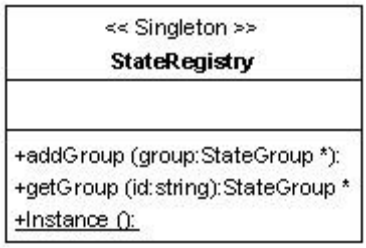

Figura 4.8: Diagrama de classes para `StateRegistry`.

A principal responsabilidade de `StateRegistry` é mapear nomes para instâncias de estados. Por essa razão, existe o pré-requisito de que os nomes dos estados sejam únicos no *framework*. A associação entre estados e grupos é realizada pelo construtor das classes `State` e `StateGroup`, com o auxílio da classe `StateRegistry`.

Durante a construção de uma instância qualquer de um estado, o `StateRegistry` é consultado para recuperar a referência para o pai do estado. Essa operação utiliza o identificador para pai do estado fornecida durante a chamada ao construtor. Dessa forma, é possível definir automaticamente hierarquias de estados no *framework*.

O único caso especial no registro refere-se ao estado mestre. Como o estado mestre não possui pai, o `StateRegistry` possui um identificador especial para definir estados nulos, que corresponde a uma cadeia de caracteres vazia.

4.3.1.3. Mudança de Estado

As operações de mudança de estado são requisitadas pelos próprios estados, a seus pais. Existem dois tipos de mudança de estado no *framework*: mudanças definitivas e mudanças temporárias.

A interface para essas operações é definida da seguinte forma:

```
class StateGroup
{
   ...

  public:

     // mudanças definitivas
     void changeState (const string & stateId);

     // mudanças temporárias
     void pushState   (const string & stateId);
     void popState    ();

   ...
};
```

As operações de mudança de estado aceitam como parâmetro o nome do estado para o qual se deseja mudar. Para exemplificar o uso dessas operações, será usada a máquina de estados representada na figura 4.9.

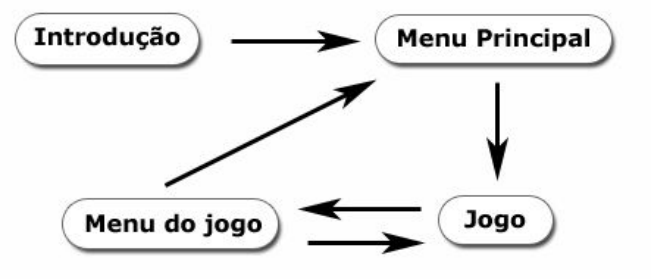

Figura 4.9: Exempo de máquina de estados .

A operação de mudança temporária de estados pode ser aplicada quando deseja-se restaurar o estado anterior após o uso de um determinado estado. Na figura 4.9 isso ocorreria caso o estado representado por "Menu do jogo" fosse invocado a qualquer momento. A operação de mudança definitiva de estados pode ser aplicada quando não é necessário guardar o estado corrente antes da operação ser invocada.

Internamente, os grupos organizam seus filhos com o uso de uma pilha. Dessa forma, quando uma operação de mudança temporária de estados é invocada (através do método `pushState ()`), o estado atual é empilhado e o novo estado passa a ser atual. Quando esse estado não é mais necessário, basta requisitar que o anterior seja restaurado (através do método `popState ()`) e o estado que estiver no topo da pilha passará a ser o estado corrente.

A implementação atual da máquina de estados permite apenas mudanças para estados irmãos, antecessores ou irmãos dos antecessores do estado. Por exemplo, na hierarquia definida na figura 4.10, não é possível mudar do estado "e" para o estado "h".

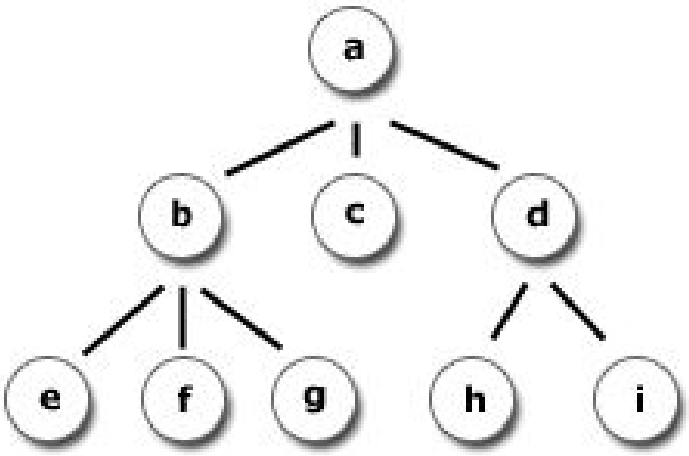

Figura 4.10: Exemplo de hierarquia onde não é possível mudar do estado “e” para o “h”.

Essa restrição deve-se ao fato de que o *framework* considera como estados principais da aplicação aqueles de nível 1 (filhos imediatos do estado mestre) e os outros como refinamentos desses estados. Isso simplifica a implementação das operações de mudanças de estado. O detalhamento dessas operações encontra-se em anexo.

4.3.1.4. Estados Simples

Essa classe não pode ser utilizada diretamente (pois seu construtor não é público), sendo obrigatória a definição de especializações. Todos os métodos dessa classe possuem uma implementação padrão, que é não realizar nenhuma ação. Dessa forma, as classes derivadas podem podem tratar apenas os eventos de seu interesse (ao contrário de `AbstractState`, que obriga as classes derivadas a implementar todos os métodos).

O diagrama de classes para esse tipo de estado pode ser visualizado na figura 4.11.

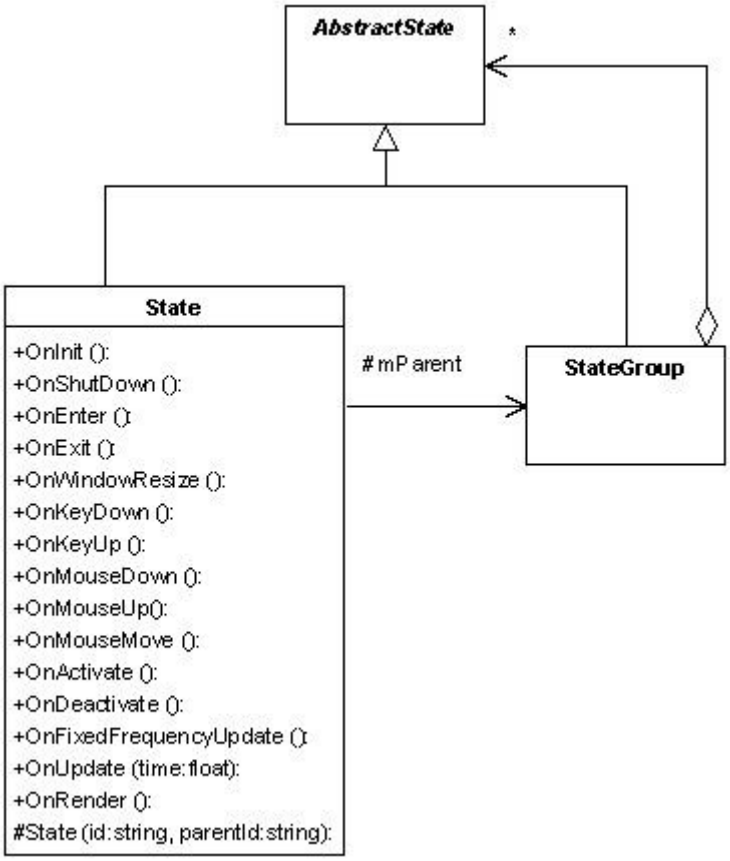

Figura 4.11: Diagrama de classes para `State`.

4.3.1.5. Estados Compostos

O diagrama de classes para os grupos de estados é representado na figura 4.12.

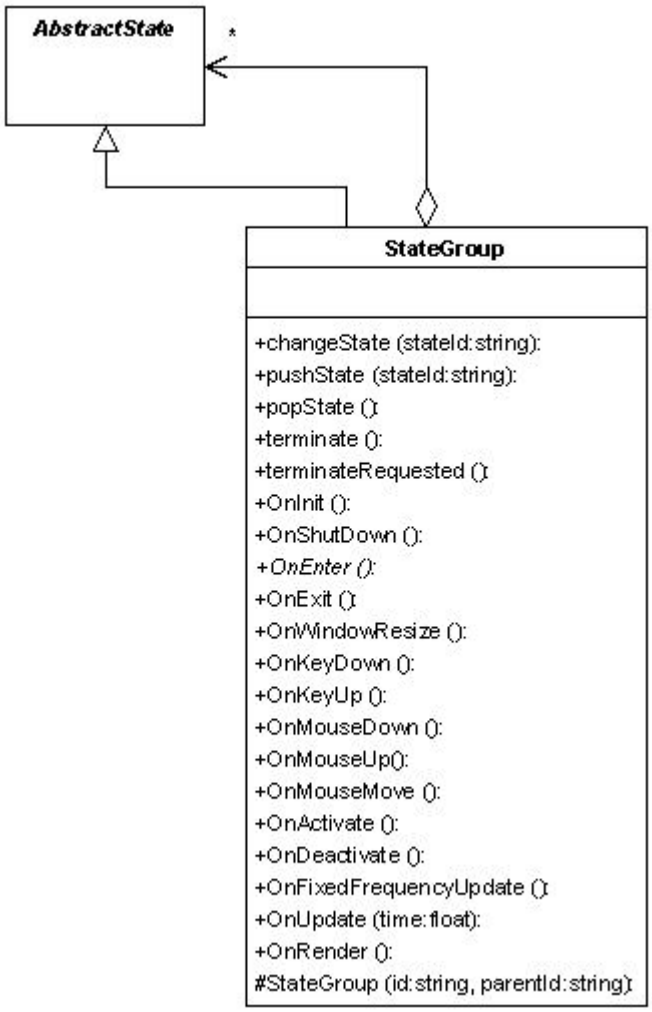

Figura 4.12: Diagrama de classes para `StateGroup`.

Em relação ao tratamento dos eventos, essa classe basicamente delega essa tarefa para o estado atual do grupo, exceto em três eventos: `OnInit ()`, `OnShutDown ()` e `OnEnter ()`.

Os métodos `OnInit ()` e `OnShutDown ()` executam para todos os estados as operações de inicialização e de finalização, respectivamente.

O método `OnEnter ()` não possui implementação definida, o que torna `StateGroup` uma classe abstrata. Conseqüentemente, não é possível usar essa classe diretamente, sendo obrigatória a definição de especializações. A motivação para essa abordagem é que esse método seja usado para definir o estado inicial do grupo quando a aplicação decidir que o grupo será o estado atual. Um exemplo de implementação é demonstrado neste trecho de código:

```
class Grupo : public StateGroup
{
  private:

     Estado1 e1;
     Estado2 e2;

  public:

     Grupo ()
     : StateGroup ("grupo") {}

     void OnEnter ()
        {
          // o estado e1 será o inicial deste grupo
          changeState (e1.getId () );
        }

};
```

A classe `StateGroup` define também outros métodos públicos: `getState ()`, `terminate ()` e `terminateRequested ()`.

O método `getState ()` pode ser usado para se recuperar uma instância de um estado pertencente ao grupo. O método `terminate ()` requisita que a aplicação seja encerrada. Essa requisição percorre a árvore de estados até chegar ao topo. O método `terminateRequested ()` é usado para verificar se alguma requisição de encerramento da aplicação foi feita. A classe `StateAppRunner` usa esse método para decidir se continua a executar o ciclo principal de jogo.

4.3.1.6. Estado Mestre

O estado mestre é implementado como um *Singleton*. Sua principal função é facilitar a operação de associação entre estados e grupos.

Por padrão, o estado mestre possui um estado filho pré-definido, que representa o estado vazio. O estado vazio é usado para que sempre exista um estado válido no *framework*. O único evento definido nesse estado é o `OnUpdate ()`, que requisita o término da aplicação caso seja executado.

Normalmente, as aplicações não precisam lidar diretamente com o estado mestre do Guff, nem estar cientes de sua existência.

4.4. Ferramentas (*toolkit*)

O *toolkit* do *framework* Guff oferece soluções para diversos módulos como visualização, Matemática, áudio e dispositivos de entrada. A implementação do *toolkit* está dividida em diversos *namespaces*, que são usados para agrupar soluções relacionadas. A figura 4.13 ilustra uma relação entre os módulos e os *namespaces* do *framework*.

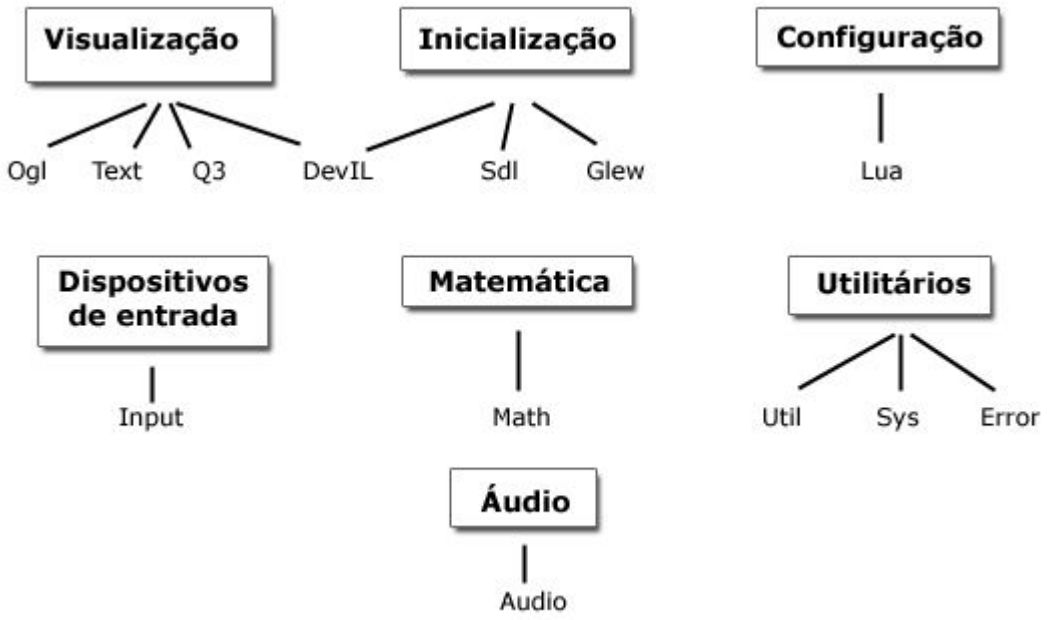

Figura 4.13: Relação entre módulos oferecidos pelo *framework* e seus *namespaces*.

4.4.1. Inicialização de Bibliotecas

Algumas das bibliotecas usadas pelo *framework* precisam ser inicializadas antes de serem utilizadas e finalizadas ao término da aplicação.

Esse problema representa um caso que pode ser resolvido através da aplicação do paradigma de gestão automática de recursos. O paradigma é aplicado segundo o padrão demonstrado neste trecho de código:

```
class Initializer
{
  public:

    Initializer ()
      {
        if (sWasInitialized == false)
          {
            inicializarBiblioteca ();

            if (inicialização falhou)
              // geração de uma exceção
          }

        sWasInitialized = true;
        sUseCount++;
      }


    Initializer::~Initializer ()
    {
      sUseCount--;

      if (sUseCount == 0)
        {
          finalizarBiblioteca ();
          sWasInitialized = false;
        }

    }

  private:
    static bool sWasInitialized;
    static long sUseCount;
};
```

Dessa forma, é possível inicializar e finalizar automaticamente as bibliotecas. Essa abordagem é aplicada para as bibliotecas SDL, GLEW e *DevIL*, em seus respectivos *namespaces*.

No *framework*, a inicialização de bibliotecas e subsistemas é realizada automaticamente pela classe `StateAppRunner`.

4.4.2. Configuração da Aplicação

Para a determinação da configuração da aplicação, o *framework* utiliza a linguagem Lua. O *namespace* `Lua` define uma classe para tratar contextos Lua como objetos automáticos. O diagrama de classes para esse *namespace* é representado na figura 4.14.

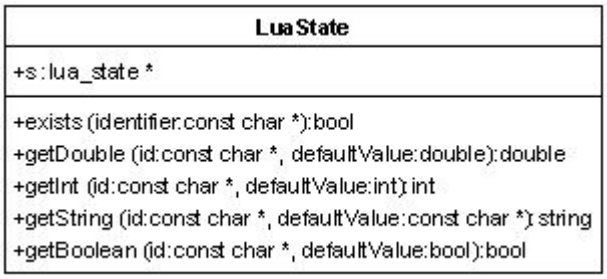

Figura 4.14: Diagrama de classes para o *namespace* `Lua`.

Os métodos definidos nessa classe permitem que sejam recuperados identificadores nos *scripts*. O comportamento comum a todos eles é retornar um valor padrão caso o identificador não exista no *script*. O contexto Lua é definido como público pela classe para não impor restrições quanto ao uso da linguagem Lua nas aplicações.

A configuração é definida através do *script* `config.lua`. Os parâmetros que podem ser alterados estão listados na tabela 4.2.

Parâmetro	Significado	Valor padrão	Obrigatório
`width`	largura da janela	640	não
`height`	altura da janela	480	não
`zbuffer`	precisão do *z-buffer* em *bits*	16	não
`bpp`	modo de cores em *bits*	16	não
`fullscreen`	usar modo gráfico em tela cheia	false	não
`vsync`	sincronizar com a taxa de atualização vertical do monitor	true[16]	não
`stencil`	precisão do *stencil buffer* em *bits*	0 (desabilitado)	não
`alphabits`	precisão do *alpha buffer* em *bits*	0 (desabilitado)	não
`state0`	nome do estado inicial da aplicação	-	**sim**
`gameTicks`	repetições por segundo do ciclo de freqüência fixa, caso seja 0 o ciclo é desabilitado	0	não
`appName`	nome da aplicação, que aparecerá como título da janela	Guff	não

Tabela 4.2: Parâmetros de configuração do *framework* Guff.

4.4.3. Módulo de Dispositivos de Entrada

O módulo de dispositivos de entrada está implementado no *namespace* `Input`. As classes definidas neste módulo representam abstrações para o teclado e o *mouse*. O diagrama de classes para esses dispositivos é demonstrado na figura 4.15.

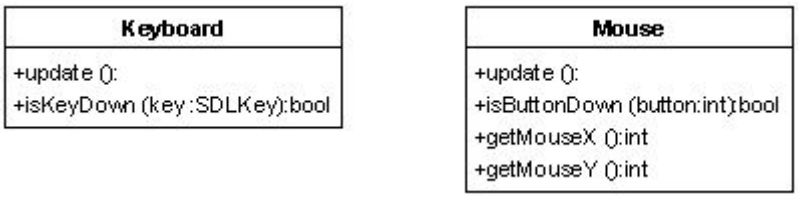

Figura 4.15: Diagrama de classes para o *namespace* `Input`.

Essas classes permitem acesso direto aos dispositivos.

16 Pode ser desligado se o *hardware* oferecer essa funcionalidade.

Na classe `Keyboard`, o método `update ()` permite que a classe seja atualizada com o estado atual do teclado. As consultas sobre quais teclas estão pressionadas (ou não) podem ser realizadas através do método `isKeyDown ()`.

O *mouse* é representado pela classe `Mouse`. Assim como na classe para teclados, o método `update ()` permite que a classe recupere as informações mais recentes sobre o *mouse*. O estado dos botões podem ser consultado com o método `isButtonDown ()` e a posição do *mouse* pode ser recuperada com os métodos `getMouseX ()` e `getMouseY ()`.

4.4.4. Módulo de Matemática

O módulo de Matemática está implementado no *namespace* `Math`, que define várias classes como vetores, planos, matrizes, *quaternions*, *frustums* e *bounding boxes*. Essas classes são representadas no diagrama da figura 4.16.

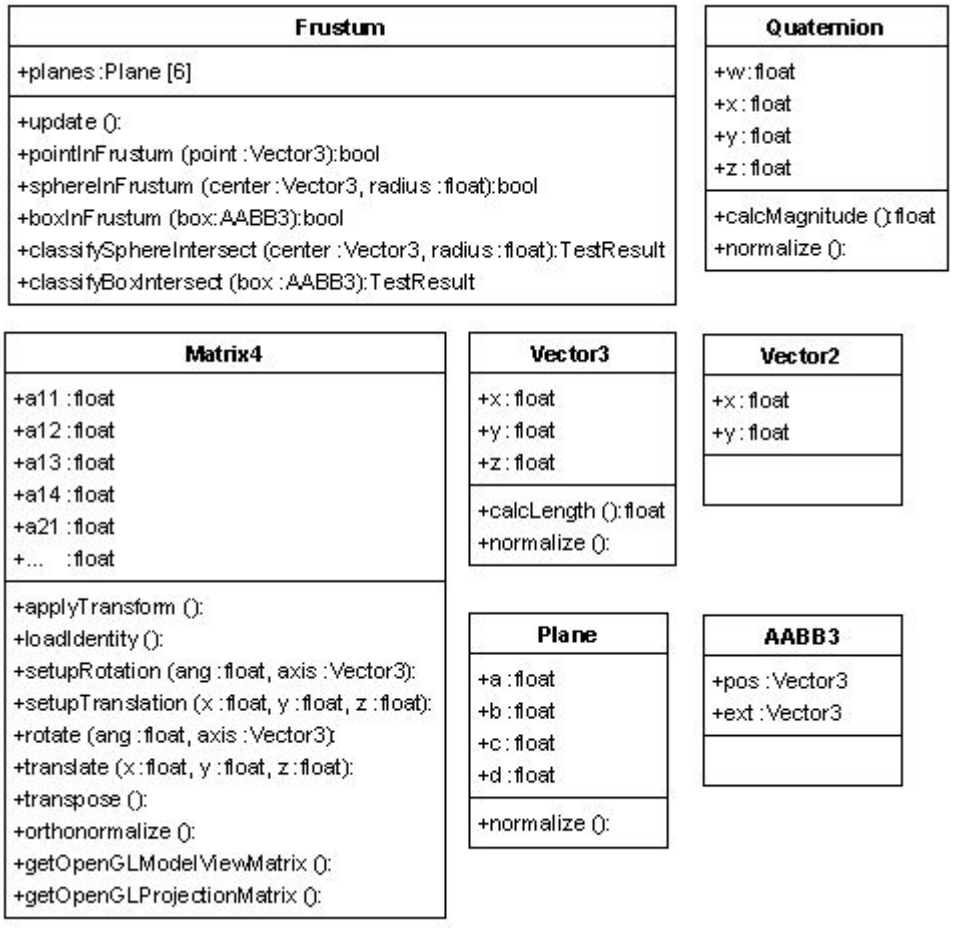

Figura 4.16: Diagrama de classes para o *namespace* `Math`.

A classe `AABB3` representa um *bounding box* 3D. Um *bounding box* é o menor paralelepípedo que envolve uma determinada geometria. Em particular, o AABB (*Axis Aligned Bounding Box*) representa um paralelepípedo que está alinhado com os eixos X, Y e Z globais. Essa configuração provê uma maneira simples de se verificar se dois paralelepípedos se sobrepõem, embora exista a limitação de que eles não podem sofrer rotações em qualquer um dos eixos (caso isso ocorra será necessário calcular um novo AABB).

A classe para matrizes (`Matrix4`) representa uma matriz 4x4. Essa classe foi projetada para ser usada com o *OpenGL,* e por isso segue as suas convenções de representação em memória e multiplicação. O *OpenGL* armazena as matrizes por coluna e realiza pós-multiplicação (multiplicação por vetores coluna).

A classe `Frustum` representa um volume de visão, que pode ser atualizada diretamente a partir do *OpenGL* através do método `update ()`.

Essa classe oferece vários tipos de teste que podem ser usados na implementação do *frustum culling*. Os testes `pointInFrustum ()`, `sphereInFrustum ()`, `cubeInFrustum ()` e `boxInFrustum ()` verificam se pontos, esferas, cubos e paralelepípedos estão ou não no interior do *frustum*, respectivamente. Os outros testes, `classifySphereIntersect ()` e `classifyBoxIntersect ()`, provêem um maior refinamento ao responder se o objeto está totalmente no interior do *frustum*, parcialmente contido no *frustum* ou totalmente fora do *frustum*.

As classes `Vector3`, `Vector2`, `Plane` e `Quaternion` representam vetores 3D, vetores 2D, planos e *quaternions*, respectivamente. Várias operações tradicionais como cálculo de produto escalar, produto vetorial e distância de um ponto ao plano, entre outras, também estão implementadas neste módulo.

4.4.5. Módulo de Áudio

O módulo de Áudio é representado pelo *namespace* `Audio`. O diagrama de classes para esse *namespace* é ilustrado pela figura 4.17.

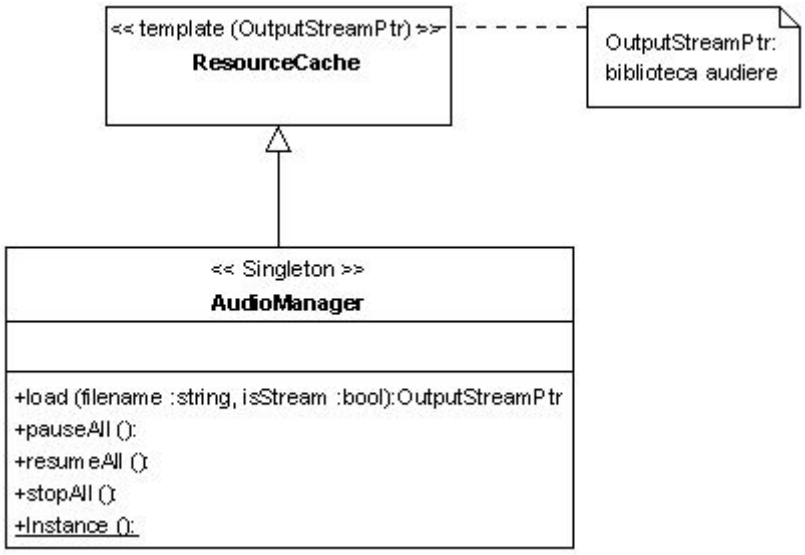

Figura 4.17: Diagrama de classes para o *namespace* `Audio`.

A leitura e reprodução de arquivos de áudio é realizada com a biblioteca *audiere*. É possível usar vários tipos de arquivos sonoros, como ogg, mp3 e wav.

A classe `AudioManager` representa um *cache* para arquivos de áudio. Assim como os outros gerentes do *framework*, sua principal responsabilidade é evitar que um arquivo de áudio qualquer seja carregado mais de uma vez. Essa classe é implementada como um *Singleton*.

A operação de leitura do arquivo de áudio é implementada pelo método `load ()`. O primeiro parâmetro corresponde ao nome do arquivo a ser lido, e o segundo especifica se o arquivo deve ser carregado com um *stream* ou não. Quando o arquivo é usado como um *stream*, significa que seus dados serão lidos sob demanda durante a reprodução do áudio (ou seja, o arquivo não será carregado de uma só vez para a memória). A classe `OutputStreamPtr` é um objeto automático definido pela biblioteca *audiere*, que representa um ponteiro com contador de referência.

O método `pauseAll ()` interrompe a reprodução de todos os arquivos de áudio em execução. Durante essa operação, esses objetos são salvos em uma fila para que possam ser restaurados mais tarde (pelo método `resumeAll ()`). No *framework*, o método `pauseAll ()` é invocado quando a aplicação torna-se inativa (por exemplo, quando o usuário minimiza a janela do programa) e o método `resumeAll ()` é usado quando a aplicação torna-se ativa.

A diferença entre `pauseAll ()` e `stopAll ()` é que este último não permite que a operação seja revertida.

Para conveniência do usuário, no *namespace* `Audio` está definida a função `LoadAudio ()` que invoca o gerente de áudio para a leitura dos arquivos em disco, que pode ser usada desta forma:

```
OutputStreamPtr audio = LoadAudio ("musica.mp3", true);
```

Os parâmetros são os mesmos do método `load ()` da classe `AudioManager`.

4.4.6. Módulo Utilitário

O módulo utilitário é definido pelos *namespaces* `Util`, `Error` e `Sys`.

O *namespace* `Sys` oferece ferramentas como relógios e repositório (*cache*) de recursos. O diagrama de classes para esse *namespace* é ilustrado na figura 4.18.

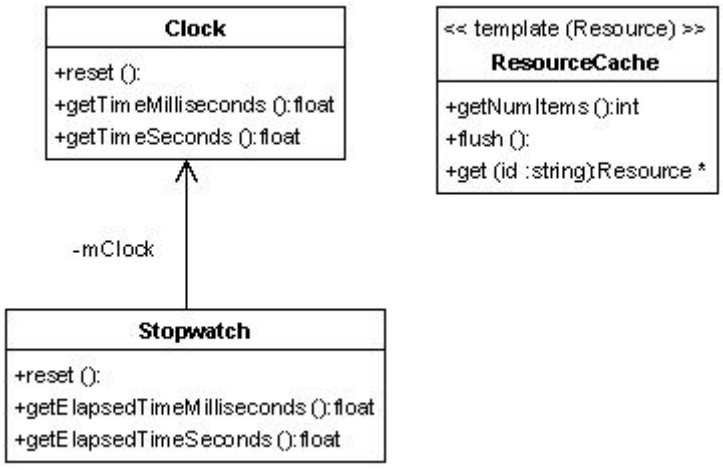

Figura 4.18: Diagrama de classes para o *namespace* `Sys`.

Os relógios possuem papel importante no *framework* já que várias funcionalidades dependem de tempo. A classe `Clock` possui métodos que recuperam o tempo decorrido desde que o relógio foi criado (ou reiniciado). A classe `Stopwatch` representa um cronômetro simples, e seus métodos recuperam o tempo decorrido desde a última consulta.

A classe `ResourceCache` implementa um *cache* para recursos quaisquer (é uma classe *template*). A principal responsabilidade dessa classe é impedir que existam duplicatas de recursos. Como o *framework* utiliza objetos automáticos, essa classe não tem responsabilidade de liberar os recursos explicitamente (embora isso possa ser feito com o método `flush ()`).

O método `getNumItems ()` permite recuperar o número de objetos armazenados no *cache* e o método `get ()` permite acesso direto a um determinado elemento do *cache*, sem cópias.

No *namespace* `Error`, está definida a classe usada como exceção padrão do *framework*, `Exception`. Além dessa classe, são definidas também macros para depuração e geração de exceções.

A única classe definida no *namespace* `Util` é `FrameCounter`, que representa um contador de quadros por segundo. Essa classe pode ser usada para medições de desempenho.

O diagrama de classes para esses dois *namespaces* está representado na figura 4.19.

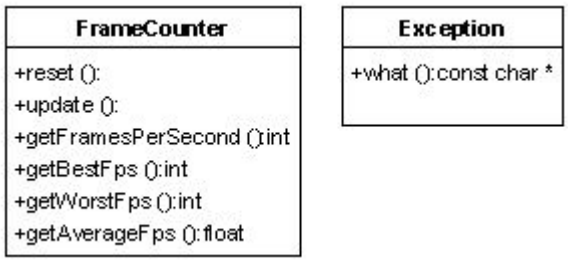

Figura 4.19: Diagrama de classes para os *namespaces* `Error` e `Util`.

4.4.7. Módulo de Visualização

O módulo de Visualização é baseado no *OpenGL*, sendo composto por três *namespaces*: `Ogl`, `Text` e `Q3`. Esses *namespaces* fornecem as classes principais para o módulo de Visualização, classes para renderização de texto e classes para lidar com cenários do jogo *Quake III*, respectivamente.

4.4.7.1. OpenGL

Um diagrama das classes principais para o *namespace* `Ogl` é representado na figura 4.20.

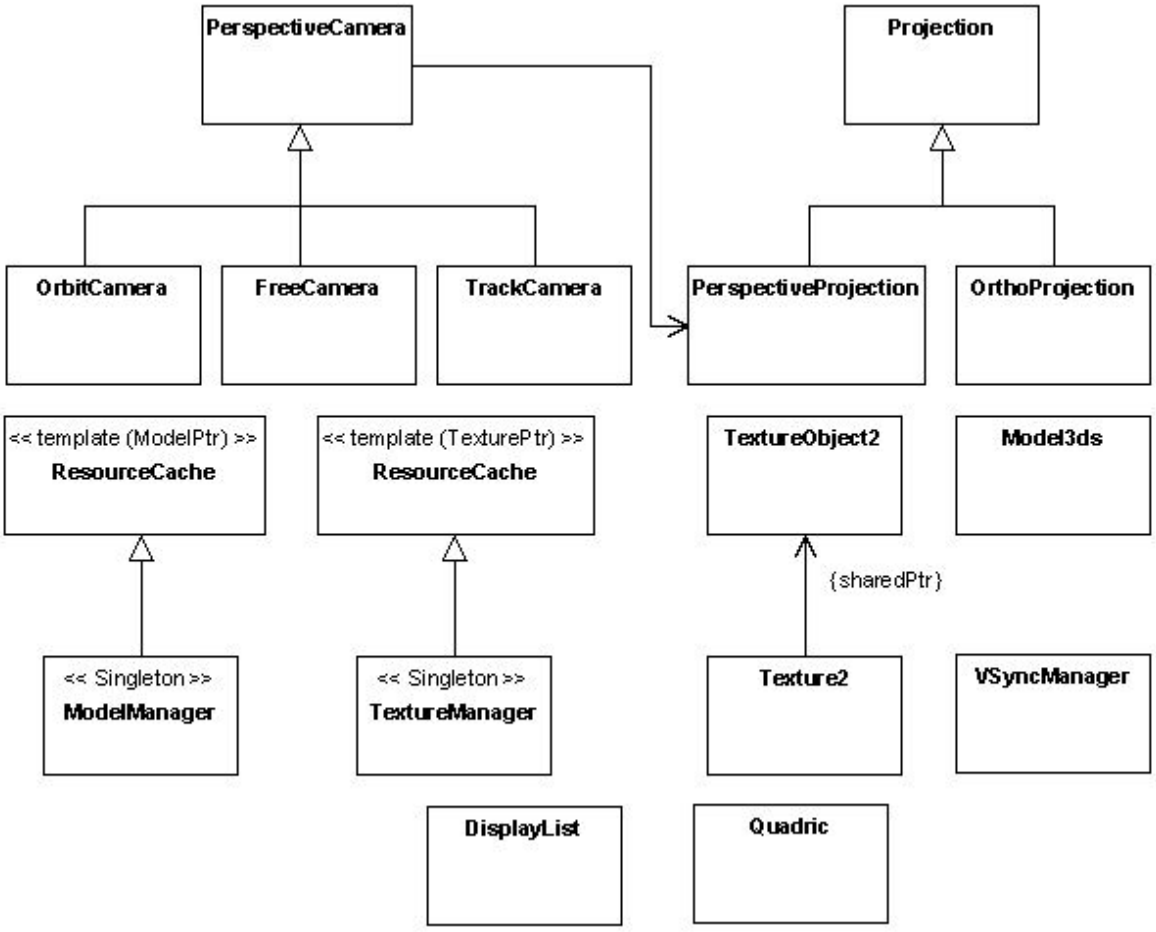

Figura 4.20: Diagrama das classes principais do *namespace* `Ogl`.

Várias dessas classes implementam objetos automáticos para recursos do *OpenGL*, como `Quadric`, `DisplayList` e `TextureObject2`.

Esse *namespace* define outros tipos de objetos como texturas, modelos tridimensionais e câmeras, que serão apresentados nesta seção.

4.4.7.1.1. Texturas

As classes relacionadas a texturas são `Texture2`, `TextureObject2` e `TextureManager`. Um diagrama mais detalhado dessas classes é representado na figura 4.21.

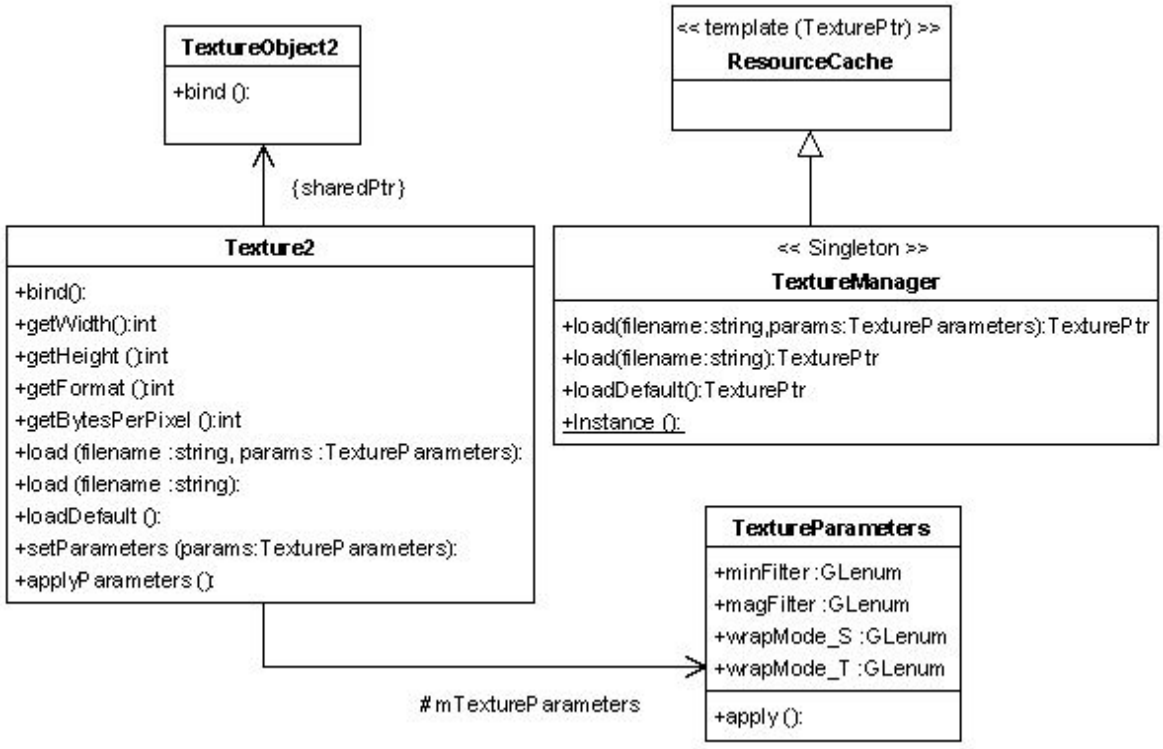

Figura 4.21: Diagrama de classes relacionadas a texturas.

A classe `TextureObject2` representa um objeto automático para o recurso "identificador de textura do *OpenGL*". No *OpenGL*, esse recurso representa uma abstração para um conjunto de informações sobre uma determinada textura (como seus *texels* e filtros aplicados). No construtor dessa classe, um identificador para texturas é requisitado ao *OpenGL*, e no destrutor esse identificador é devolvido ao sistema.

O método `bind ()` permite que a textura representada pelo identificador seja selecionada para uso.

A classe `TextureParameters` representa um conjunto de parâmetros que afetam a aparência da textura. Os campos definidos nessa classe estão listados na tabela 4.3.

Campo	Significado	Valor padrão
`minFilter`	Filtro de redução a se usar no mapeamento de textura	`GL_LINEAR_MIPMAP_NEAREST`
`magFilter`	Filtro de ampliação a se usar no mapeamento de textura	`GL_LINEAR`
`wrapMode_S`	Modo para *wrapping* de coordenadas de textura no eixo S da textura	`GL_REPEAT`
`wrapMode_T`	Modo para *wrapping* de coordenadas de textura no eixo T da textura	`GL_REPEAT`

Tabela 4.3: Campos da classe `TextureParameters`, seus significados e valores padrões.

O modo de *wrapping* (*wrap mode*) em texturas indica como ocorrerá o mapeamento de texturas quando as coordenadas de texturas especificadas estiverem fora da faixa [0, 1]. Por exemplo, o valor `GL_CLAMP` indica que a textura será truncada e o valor `GL_REPEAT` indica que a textura será replicada. A figura 4.22 ilustra esses dois modos.

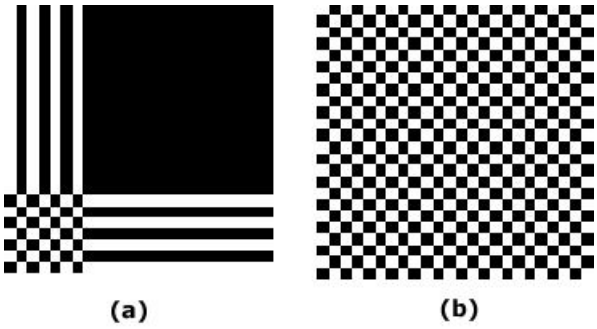

Figura 4.22: Modos para *wrapping* de texturas (Neider e Davis, 1997). (a) `GL_CLAMP`. (b) `GL_REPEAT`.

Os filtros de redução e ampliação possuem os mesmos valores definidos no *OpenGL*. Esses valores e seus significados estão definidos na tabela 4.4.

Nome do filtro	Descrição
GL_NEAREST	Utiliza o *texel* mais próximo (vizinho) do *pixel* que está sendo mapeado. É o filtro de melhor desempenho, porém é o de menor qualidade. Pode ser usado como filtro de ampliação ou redução.
GL_LINEAR	Utiliza interpolação linear. Provê melhores resultados visuais, mas é mais custoso em termos computacionais. Pode ser usado como filtro de redução e ampliação. Esse filtro é o selecionado por padrão.
GL_NEAREST_MIPMAP_NEAREST	Escolhe a *mipmap* com tamanho semelhante ao do polígono que está sendo mapeado e usa o mesmo critério que GL_NEAREST para escolher o *texel* da textura.
GL_NEAREST_MIPMAP_LINEAR	Escolhe a *mipmap* com tamanho semelhante ao do polígono que está sendo mapeado e usa o mesmo critério que GL_LINEAR para escolher o *texel* da textura.
GL_LINEAR_MIPMAP_NEAREST	Escolher duas *mipmaps* com tamanhos mais próximos ao do polígono que está sendo mapeado e usar o filtro GL_NEAREST para escolher o *texel* de cada textura. O valor final é uma média entre esses dois valores.
GL_LINEAR_MIPMAP_LINEAR	Escolher duas *mipmaps* com tamanhos mais próximos ao do polígono que está sendo mapeado e usar o filtro GL_LINEAR para escolher o *texel* de cada textura. O valor final é uma média entre esses dois valores.

Tabela 4.4: Filtros para ampliação e redução de texturas definidos no *OpenGL* (Valente, 2004b).

Os parâmetros definidos pela classe `TextureParameters` podem ser efetivados a qualquer momento utilizando-se o método `apply ()`.

A classe `Texture2` representa uma textura bidimensional. Essa classe armazena várias propriedades sobre a imagem, que podem ser recuperadas através dos métodos `get* ()`, que são listados na tabela 4.5.

Método	Significado
`getWidth`	Recuperar a largura da textura em *texels*
`getHeight`	Recuperar a altura da imagem em *texels*
`getBytesPerPixel`	Recuperar a quantidade de *bytes* por *pixel* da imagem
`getImageFormat`	Recuperar uma constante que indica como os dados internos da imagem são armazenados. Os valores são os mesmos definidos pelo *OpenGL*

Tabela 4.5: Métodos de recuperação de informações da classe `Texture2`.

Os dados da textura podem ser lidos do disco com o uso dos métodos `load (const string & filename)` e `load (const string & filename, const TextureParameters & parameters)`. A diferença entre eles é que o primeiro usa os valores padrões definidos pela classe `TextureParameters` e o segundo usa os parâmetros definidos pelo usuário.

A leitura dos dados da imagem em disco é realizada com o auxílio da biblioteca *DevIL*, através da classe `Image` do *namespace* `DevIL` (essa classe é outro exemplo da aplicação do paradigma de gestão automática de recursos). O uso da *DevIL* possibilita que sejam usados vários tipos de arquivos de imagem, como bmp, jpeg, tga e png, entre outros. A única restrição para as imagens é que suas dimensões devem ser potência de dois (64x64, 256x128, por exemplo). Essa é uma restrição do *OpenGL*[17]. Caso seja necessário, esses métodos contróem as *mipmaps* automaticamente, dependendo do valor dos filtros de redução.

Uma maneira alternativa para se definir a textura é usar o método `loadDefault ()`, que cria uma textura procedural "padrão". A imagem gerada é equivalente à figura 4.23.

Figura 4.23: Textura padrão.

17 A versão 2.0 do *OpenGL* possibilitará o uso de texturas de dimensões arbitrárias sem o uso de extensões (Segal e Akeley, 2004).

É interessante notar que a classe `Texture2`, por usar uma instância de `TextureObject` é, naturalmente, um objeto automático. Como exemplo, têm-se este trecho de código:

```
void foo ()
{
  Texture2 t1;                // t1 carrega a textura padrão
  Texture2 t2 ("abc.jpg");    // t2 carrega a textura abc.jpg

  t2 = t1;                    // t2 libera recursos alocados para abc.jpg
                              // e passa a referenciar a textura padrão

  t1.load ("xyz.tga");        // t1 carrega a textura xyz.tga e
                              // t2 continua com a referência para
                              // a textura padrão
}

// os recursos alocados para xyz.tga e para a textura padrão
// são liberados automaticamente ao término da função
```

A textura pode ser selecionada para uso com o método `bind ()`. Os parâmetros que afetam sua aparência podem ser especificados com o método `setParameteres ()` e aplicados com `applyParameters ()`.

A classe `TextureManager` representa um *cache* para texturas. Sua principal responsabilidade é gerir a leitura de arquivos de imagem de modo que uma determinada textura não seja carregada mais de uma vez. Essa classe é implementada como um *Singleton*. A forma de uso é bastante simples:

```
TexturePtr t = TextureManager::Instance ()->load ("imagem.bmp");
```

Os métodos para leitura de arquivos de imagem são equivalentes aos da classe `Texture2`. A operação de leitura de arquivos de imagem em disco, realizada pelo método `load ()`, é implementada desta forma: inicialmente, é feita uma consulta ao *cache* para verificar se o elemento está presente. Caso esteja, é retornada uma referência para esse elemento. Se o elemento não estiver presente no *cache*, é criada uma nova textura que é então adicionada ao *cache*.

Para conveniência do usuário, são definidas no *namespace* `Ogl` duas funções para a criação de texturas: `LoadTexture ()` e `LoadTextureDefault ()`. Essas funções invocam o *cache* de texturas automaticamente, possuindo parâmetros equivalentes aos métodos correspondentes da classe `Texture2`.

4.4.7.1.2. Modelos 3D

O diagrama das classes relacionadas com modelos 3D é representado na figura 4.24.

O *framework* Guff define uma classe (`Model3ds`) para representar modelos tridimensionais estáticos (sem animação). A geometria do modelo pode ser determinada através de arquivos 3ds, que são originários do programa de modelagem *3ds max*.

A única restrição em relação aos modelos é que cada malha que forma o modelo tenha apenas um material. Essa restrição tem como objetivos minimizar a troca de materiais durante a renderização e renderizar cada malha como um lote de geometria (enviar os dados de uma única vez para a placa gráfica).

Os materiais são representados pela classe `Material`. As propriedades definidas para os materiais são cores difusa, ambiente e especular, expoente especular e uma textura.

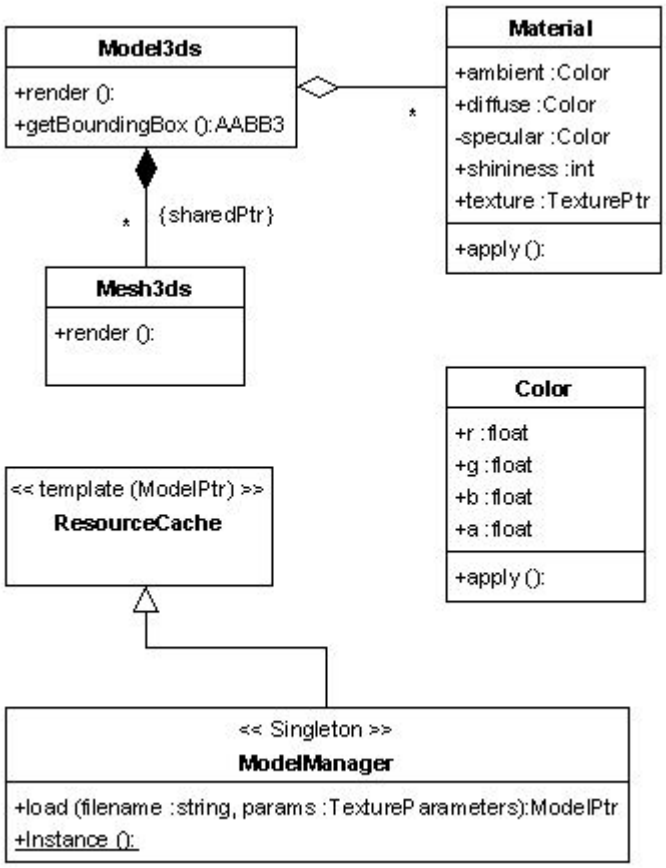

Figura 4.24: Diagrama de classes para modelos 3D.

O uso de modelos tridimensionais é demonstrado por este trecho de código:

```
Model3ds m ("abc.3ds");
m.load ("xyz.3ds");
```

O construtor de `Model3ds` aceita como parâmetros o nome do arquivo a ser carregado e os parâmetros de texturas que serão aplicados nas texturas do modelo. Caso os parâmetros de textura não sejam especificados (como no exemplo), serão usados os valores padrões definidos por `TextureParameters`. Muitas vezes pode não ser possível (ou desejável) carregar o modelo no momento da declaração do objeto. Para resolver esse problema (e evitar o uso de ponteiros tradicionais), pode ser usada a classe `ModelPtr`, que representa um ponteiro com contadores de referência. Essa classe é fornecida pela biblioteca *boost.*

```
typedef boost::shared_ptr <Model3ds> ModelPtr;
```

Essa classe pode ser usada desta forma:

```
// declaração
ModelPtr p;

...

// uso
p.reset (new Model3ds ("abc.3ds") );
```

A classe `ModelManager` representa um *cache* para modelos tridimensionais. Assim como os outros gerentes do *framework*, essa classe é implementada como um *Singleton* e possibilita que os modelos tridimensionais sejam carregados do disco uma só vez.

Assim como na classe `Model3ds`, o método `load ()` de `ModelManager` aceita o nome do arquivo a ser lido e os parâmetros de textura. Para conveniência de notação, a função `LoadModel3ds ()` está disponível para uso. Essa função invoca o *cache* de modelos 3D automaticamente, e seus parâmetros correspondem aos parâmetros do método `load ()` da classe `Model3ds`.

4.4.7.1.3. Projeções

Os tipos de projeção disponíveis no *framework* são projeções ortográficas e projeções em perspectiva.

O diagrama de classes para projeções está representado na figura 4.25.

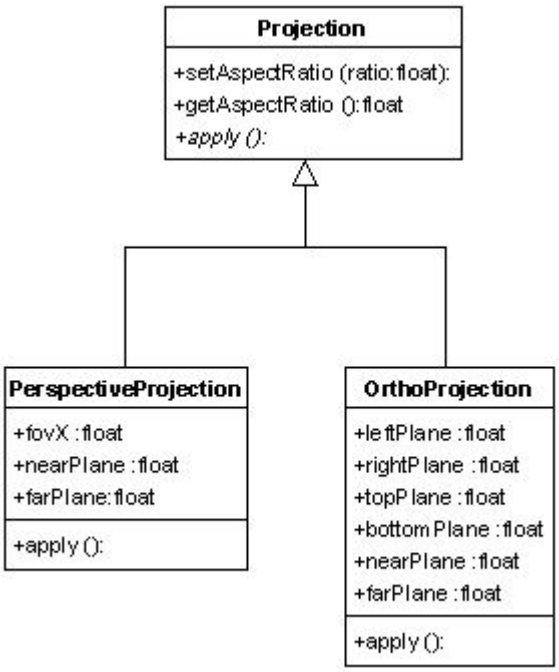

Figura 4.25: Diagrama de classes para projeções.

A interface comum para as projeções implementadas pelo *framework* é definida pela classe `Projection`. Essa classe é abstrata, e o método `apply ()` deve ser implementado pelas classes derivadas para que seja possível aplicar a projeção.

O método `setAspectRatio ()` é usado para se determinar a razão de aspecto da projeção. Esse valor corresponde à proporção entre a largura e a altura da janela (ou *viewport*) onde a projeção será aplicada. Caso as proporções da janela e da projeção não sejam as mesmas, a imagem final será distorcida.

A projeção ortográfica é representada pela classe `OrthoProjection`. Seus campos correspondem aos seis planos que definem o volume de visão.

A projeção em perspectiva é implementada pela classe `PerspectiveProjection`. Essa classe possui as propriedades campo de visão no eixo X (*field of view*) e planos delimitadores (*near* e *far clipping planes*).

4.4.7.1.4. Câmeras

As câmeras existentes no *framework* foram projetadas para serem usadas com projeções em perspectiva. A hierarquia de classes para câmeras é ilustrada na figura 4.26.

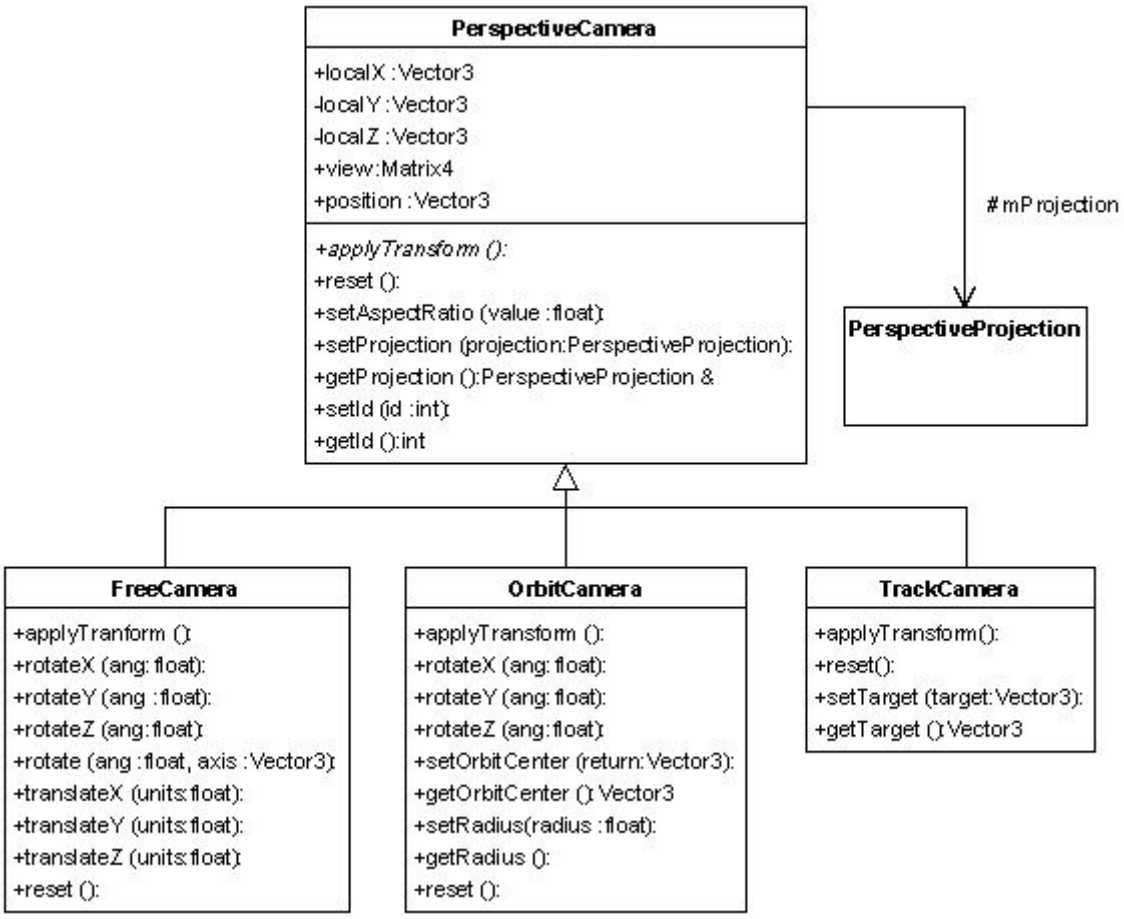

Figura 4.26: Diagrama de classes para as câmeras.

A classe base para câmeras, `PerspectiveCamera`, declara o método que é responsável por aplicar a a transformação de câmera. Esse método, `applyTransform ()`, precisa ser implementado pelas classes derivadas (o original é abstrato).

Internamente, as câmeras usam matrizes para representar a orientação. O objetivo dessa abordagem é evitar o *gimbal lock*. O *gimbal lock* é um fenômeno relacionado com a representação de orientação através de ângulos de Euler. Para estabelecer uma orientação com ângulos de Euler, é preciso efetuar rotações nos eixos X, Y e Z, seqüencialmente. O *gimbal lock* ocorre quando um ângulo de ± 90º para a segunda rotação da seqüência faz com que o primeiro e o último ângulos produzam rotações sobre um mesmo eixo (em vez de eixos independentes) (Parbery e Dunn, 2002).

Todas as câmeras possuem um método denominado `reset ()` que é responsável por restaurar a configuração inicial da câmera.

As classes que implementam câmeras concretas são `FreeCamera`, `OrbitCamera` e `TrackCamera`.

A classe `FreeCamera` representa uma câmera com seis graus de liberdade. Os métodos `rotateX ()`, `rotateY ()` e `rotateZ ()` ocasionam rotações em torno dos respectivos eixos locais da câmera. As rotações (especificadas em graus) são realizadas de acordo com a regra da mão esquerda. As translações são efetuadas de acordo com o sistema de coordenadas da mão direita.

A classe `OrbitCamera` representa uma câmera que orbita em torno de um ponto. Os métodos `setOrbitCenter ()` e `setRadius ()` podem ser usados para se determinar o centro e o raio de órbita, respectivamente. Essas propriedades podem ser consultadas a qualquer momento com os métodos `getOrbitCenter ()` e `getRadius ()`.

Os métodos `rotateX ()`, `rotateY ()` e `rotateZ ()` são usados para rotacionar a câmera em torno do centro de órbita. A tabela 4.6 resume o comportamento desses métodos.

Método	Comportamento quando ângulo (graus) é positivo
`rotateX (angle)`	A câmera é elevada em relação ao centro de órbita
`rotateY (angle)`	A câmera é rotacionada para a direita
`rotateZ (angle)`	A câmera gira no sentido anti-horário em torno do seu raio de órbita

Tabela 4.6: Comportamento dos métodos de rotação da classe `OrbitCamera`.

A classe `TrackCamera` pode ser usada quando se deseja acompanhar um determinado alvo, que pode ser determinado pelo método `setTarget ()`. A partir de então, a orientação da câmera será calculada de forma que a câmera irá apontar para o alvo e nunca estará de "cabeça para baixo".

4.4.7.1.5. Outras Classes

As outras classes definidas no *namespace* `Ogl` são: `Quadric`, `DisplayList`, `Info`, `LightSource`, `DirectionalLight`, `PointLight`, `Spotlight` e `VSyncManager`.

As classes `Quadric` e `DisplayList` representam objetos automáticos para as *quadrics* e *display lists* do *OpenGL*. As *quadrics* do *OpenGL* são abstrações usadas para a

renderização de superfícies quadráticas. As *display lists* são listas de comandos do *OpenGL* pré-processados.

A classe `Info` pode ser usada para recuperar informações sobre a implementação do *OpenGL*. Essas informações incluem o nome do fabricante, versão do *OpenGL*, nome do *driver* e os nomes das extensões do *OpenGL* que estão disponíveis.

A classe `LightSource` representa uma abstração para uma fonte de luz do *OpenGL*. As classes concretas são representadas por `DirectionalLight`, `PointLight`, `Spotlight`, que correspondem a fontes de luz direcionais, pontuais e *spot*, respectivamente.

A classe `VSyncManager` pode ser usada para se ligar ou desligar o uso da taxa de sincronismo vertical do monitor. Essa classe usa extensões do *OpenGL*, já que essa funcionalidade pode não estar presente em todas as implementações. Para verificar se o *hardware* oferece essa funcionalidade, basta usar o método `isSupported ()`. A partir de então, essa funcionalidade pode ser habilitada ou desabilitada através dos métodos `enableVSync ()` e `disableVSync ()`, respectivamente.

Essas classes estão representadas na figura 4.27.

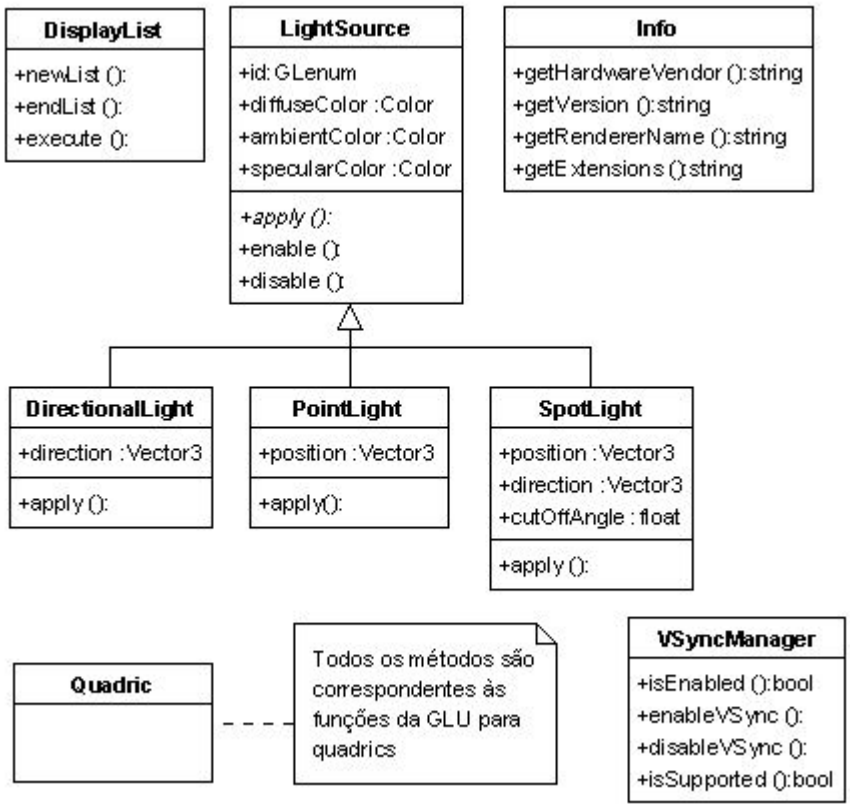

Figura 4.27: Diagrama para as classes restantes do *namespace* `Ogl`.

4.4.7.2. Texto

O *namespace* `Text` define funções e classes para uso de fontes. O diagrama de classes para esse *namespace* é representado pela figura 4.28.

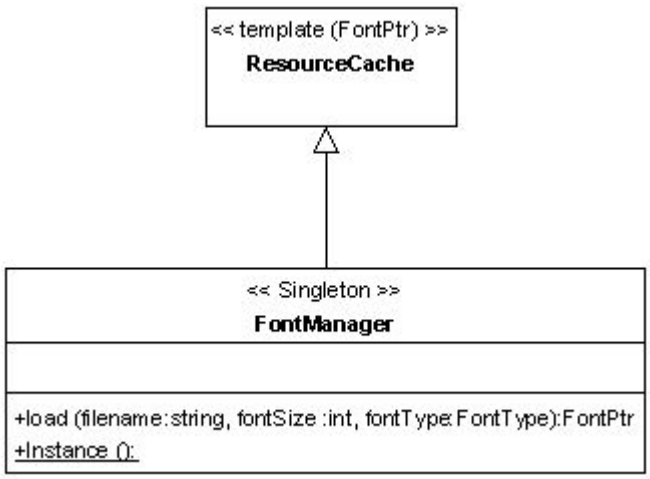

Figura 4.28: Diagrama de classes para o *namespace* `Text`.

As classes para fontes são implementadas pela biblioteca FTGL. Essa biblioteca possibilita o uso de fontes *true type*. A tabela 4.7 lista as classes da biblioteca, suas descrições e constantes para uso no *framework*.

Classe	Descrição	Constante correspondente
`FTGL`	Classe base da biblioteca, não pode ser instanciada diretamente	-
`FTGLTextureFont`	Usa um polígono simples com mapeamento de textura para renderizar cada caracter	`TEXTURE_FONT`
`FTGLBitmapFont`	Desenha os caracteres diretamente na tela usando *bitmaps* (imagens binárias)	`BITMAP_FONT`
`FTGLOutlineFont`	Desenha apenas o contorno de cada caracter, usando linhas	`OUTLINE_FONT`
`FTGLPixmapFont`	Desenha os caracteres diretamente na tela usando *pixmaps* (imagens não-binárias)	`PIXMAP_FONT`
`FTGLPolygonFont`	Uma uma malha de polígonos para representar cada caracter	`POLYGON_FONT`

Tabela 4.7: Classes da FTGL, descrições e constantes correspondentes no *framework*.

A classe `FontManager` é usada para se carregar as fontes em disco. Como os outros gerentes do framework, `FontManager` é implementada como um *Singleton*, é usada para se carregar um tipo de fonte apenas uma vez.

A operação de criação da fonte é realizada através do método `load ()`. Os parâmetros aceitos por esse método são o nome do arquivo de fonte, o tamanho da fonte (em pontos) e o tipo de fonte a ser gerado. As constantes que representam esses tipos estão definidas na tabela 4.7. A chave que identifica unicamente uma instância é uma combinação entre o nome do arquivo, tamanho e tipo. O objeto retornado representa uma referência para o elemento correspondente no *cache*. Esse objeto representa uma classe que implementa um ponteiro com contadores de referências.

O *namespace* `Text` define também algumas funções úteis para renderização de texto.

A função `Enter2D ()` é usada para estabelecer uma projeção ortográfica de modo que um *pixel* corresponda a uma unidade. O ponto (0,0) corresponde ao canto inferior esquerdo da janela. A função `Leave2D ()` é usada para restaurar a configuração anterior.

As funções `SetPosAbs ()` e `SetPosRel ()` são usadas para se determinar a posição inicial do texto. A diferença entre elas é que `SetPosAbs ()` aceita coordenadas absolutas e `SetPosRel ()` aceita coordenadas relativas em relação ao tamanho da janela. Por exemplo, `SetPosRel (0.5f, 0.5f)` indica que a posição será estabelecida no centro da janela.

4.4.7.3. Q3

No *namespace* `Q3` estão definidas classes que são usadas para lidar com cenários do jogo *Quake III*. As especificações sobre os cenários foram obtidas através de documentos não-oficiais como (Proudfoot, 2000) e (McGuire, 2003), pois não existe uma fonte oficial sobre o formato de arquivo.

A geometria dos cenários pode ser modelada com malhas de polígonos e superfícies quadráticas de Bèzier. Os cenários também podem conter objetos estáticos que são modelados com malhas de polígonos ou *billboards*. O *framework* possibilita a renderização de todos esses componentes exceto *billboards*.

O *framework* não possibilita a renderização de *sky boxes* nem o uso dos *shaders* do *Quake III*. Um *sky box* é um paralelepípedo desenhando em torno da posição onde se encontra o usuário, de modo a simular o céu. Os *shaders* são *scripts* de texto que são usados para definir a aparência das superfícies e outros efeitos especiais (Jaquays e Hook, 1999), não tendo relação os *shaders* para programação de placas gráficas atuais.

Os cenários do jogo *Quake III* são organizados em forma de uma árvore BSP[18]. A árvore BSP é utilizada para se determinar a posição do usuário na cena e nas operações de detecção de colisão. O renderizador de cenários do Guff usa *frustum culling* e o PVS[19] disponível nos arquivos de mapas para a determinação de visibilidade das cenas. O PVS é uma estrutura de dados que indica quais salas podem ser visualizadas a partir de uma outra

18 *Binary Space Partition.*
19 *Potentially Visible Set.*

sala desejada. Essa estrutura de dados é pré-calculada, pois sua determinação é bastante demorada.

As classes relacionadas aos cenários do *Quake III* e seus métodos principais estão representados no diagrama da figura 4.29.

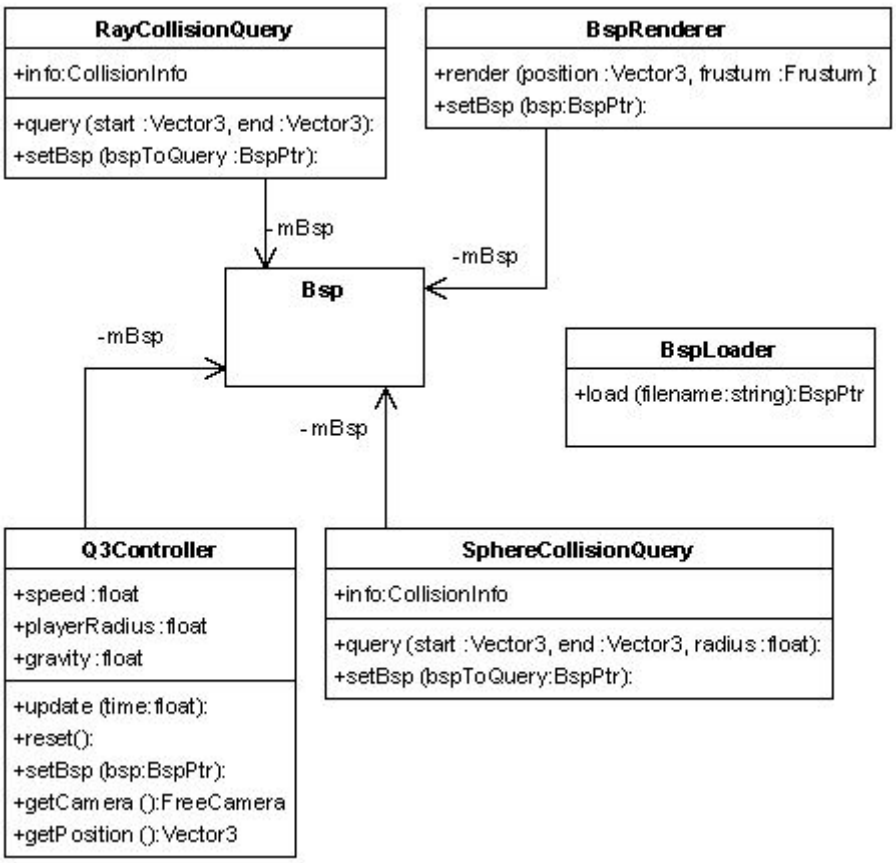

Figura 4.29: Diagrama de classes relacionadas a cenários do *Quake III*.

A classe `Bsp` é responsável por armazenar os dados do cenário. A leitura do arquivo bsp é realizada pela classe `BspLoader`.

A classe `BspRenderer` é responsável pela renderização do cenário. O método `setBsp ()` determina qual objeto `Bsp` será renderizado pela classe. A renderização é realizada pelo método `render ()`. Esse método aceita como parâmetros uma posição a partir da qual o cenário será renderizado e uma referência para o volume de visão. O volume de visão é usado para a implementação do f*rustum culling* (desconsideração de partes do cenário que o usuário não pode visualizar).

O *framework* implementa operações de detecção de colisão com o cenário através das classes `RayCollisionQuery` e `SphereCollisionQuery`. Essas classes são usadas para a verificação de colisões de raios e esferas contra o cenário, respectivamente.

A detecção de colisão pode ser realizada utilizando-se os métodos `query ()`. Em `SphereCollisionQuery`, o método `query ()` necessita dos pontos inicial e final da esfera (centro), e seu raio. Em `RayCollisionQuery`, os parâmetros correspondem aos pontos inicial e final do raio a ser disparado.

Após a realização da consulta, os resultados podem ser conferidos através do campo `info` dessas classes. Os campos da classe `CollisionInfo` estão listados na tabela 4.8.

Campo	Significado
`plane`	Plano contra o qual houve colisão
`brush`	*Brush* onde a colisão ocorreu
`endPoint`	Ponto final do movimento, já corrigido, caso exista uma colisão no caminho
`fraction`	Fração do movimento que foi percorrida. Por exemplo, o valor 1.0 significa que a movimentação desejada ocorreu totalmente, sem colisão
`startsOutsideOfBrush`	Indica se o ponto inicial do movimento está localizado fora de um *brush*
`stuckInsideOfBrush`	Indica se os pontos inicial e final do movimento estão no interior de um *brush*

Tabela 4.8: Campos da classe `CollisionInfo`.

No *Quake III*, um *brush* representa a primitiva usada para a criação dos cenários, sendo definida por quatro ou mais vértices de modo a formar um volume convexo sólido (Jaquays, 2000).

A classe `Q3Controller` pode ser usada para a interação com o usuário. A implementação dessa classe usa uma câmera em primeira pessoa e um estilo de movimentação encontrado em jogos do gênero FPS. Nesse gênero de jogo, é comum que o usuário utilize o *mouse* para orientar a câmera à sua volta e o teclado para se movimentar pelo cenário. Internamente essa classe implementa detecção e resposta a colisões de modo que o usuário permaneça dentro do cenário.

Para usar essa classe, é preciso estabelecer seus parâmetros iniciais, desta forma:

```
Q3Controller controller;
bsp = bspLoader.load ("cenario.bsp");
controller.setBsp (bsp);
controller.setStartPosition (bsp->getStartPosition (0) );
```

O método `setBsp ()` serve para informar ao controlador o cenário que será usado pela classe.

O método `getStartPosition ()` da classe `Bsp` retorna uma das posições iniciais definidas no arquivo bsp. A quantidade de posições iniciais definidas pode ser consultada através do método `getNumStartPositions ()`.

O estado do controlador pode ser atualizado com o método `update ()`. Esse método deve ser invocado periodicamente.

Durante a etapa de renderização, é necessário aplicar a transformação definida pela câmera do controlador, para que seja possível visualizar o cenário. Isso pode ser feito desta forma:

```
controller.getCamera ().applyTransform ();
frustum.update ();
bspRenderer.render (controller.getPosition (), frustum);
```

O usuário é representado por uma esfera no cenário. Essa esfera corresponde ao seu volume delimitador, cujo raio pode ser determinado pelo campo `playerRadius` da classe `Q3Controller`. O campo `speed` indica o módulo da velocidade do usuário, e o campo `gravity` representa um valor que é usado para simular a gravidade do ambiente.

4.5. Conclusão do Capítulo

Este capítulo apresentou o *framework* Guff, uma ferramenta que foi desenvolvida para a aplicação de alguns conceitos sobre desenvolvimento de jogos.

O *framework* é formado por duas partes principais: a camada de aplicação e o *toolkit*. A camada de aplicação é responsável por definir uma possível arquitetura para um jogo, utilizando uma máquina de estados para modelar suas as etapas de funcionamento. O *toolkit* oferece um conjunto de classes para diversos módulos, como visualização, Matemática, áudio e dispositivos de entrada, entre outros.

Várias bibliotecas são utilizadas na implementação do *framework* com o objetivo de evitar a síndrome "*not built here*". Em particular, a questão da portabilidade é tratada com o uso da biblioteca SDL.

O paradigma de gestão automatizada de recursos foi aplicado em diversas partes do *framework* de modo a facilitar seu uso e minimizar erros de programação relacionados a recursos.

5. Resultados Experimentais

Este capítulo apresenta uma programa de demonstração do *framework* e a avaliação do desempenho obtido pela aplicação.

O parâmetro usado para a medição de desempenho é a quantidade de quadros (*frames*) por segundo (FPS) geradas pela aplicação. Um quadro corresponde a um ciclo completo de execução.

A máquina de teste possui a seguinte configuração:

- Processador *Intel Pentium* 3 600E;
- Placa de vídeo *GeForce 4 MX* com 64 *megabytes* de memória, *driver* de vídeo versão 56.72 e velocidade 2X para o barramento AGP;
- 256 *megabytes* de memória RAM;
- Sistema Operacional *Windows XP Profissional*;
- Placa de som *Creative Labs SoundBlaster Live!*.

O programa de teste proporciona um passeio virtual por um cenário do jogo *Quake III,* com reprodução de uma música de fundo. Esse programa é constituído por dois estados: uma apresentação e o passeio virtual.

A aplicação possui vários parâmetros específicos que podem ser determinados através do arquivo `config.lua`. Esses parâmetros são listados na tabela 5.1.

Parâmetro	Significado	Valor usado no teste
`q3map`	Nome do arquivo bsp a ser usado.	data/pak0/maps/q3dm1.bsp
`maptess`	Fator usado para a subdivisão das superfícies paramétricas em malhas de polígonos. Quando maior o valor, mais suave será malha.	10
`mapgamma`	Fator usado para a modificação de luminosidade das l*ightmaps*. Quando maior o valor, mais claras serão as texturas.	2
`font`	Nome do arquivo de fonte a ser usado pelo programa.	data/chicken.ttf
`fontSize`	Tamanho da fonte em pontos.	24
`music`	Nome do arquivo de música de fundo a ser usado pelo programa.	data/audio.ogg
`camera_near`	Valor do *near plane* da câmera.	5
`camera_far`	Valor do *far plane* da câmera.	6000
`camera_fov`	Campo de visão da câmera, no eixo X.	80
`player_speed`	Velocidade do jogador.	300
`player_radius`	Raio da esfera que representa o jogador no cenário.	20
`player_gravity`	Valor para a gravidade do ambiente.	-1

Tabela 5.1: Parâmetros do programa de teste.

O cenário usado pelo programa é o `q3dm1.bsp`, que é fornecido pela versão[20] de demonstração do jogo *Quake III*.

Esse cenário possui as estatísticas listadas na tabela 5.2.

20 http://www.idsoftware.com/games/quake/quake3-gold/index.php?game_section=demo.

Tipo	Quantidade
Vértices	13978
Texturas	94
Lightmaps	9
Faces poligonais do cenário	1942
Faces dos modelos presentes no cenário	42
Faces geradas para as superfícies paramétricas	113

Tabela 5.2: Estatísticas do cenário `q3dm1`.

A figura 5.1 exibe uma tela da aplicação de teste.

Figura 5.1: Captura de tela da aplicação de teste.

A figura 5.2 exibe uma tela do jogo *Quake III*, na mesma cena.

Figura 5.2: Captura de tela do jogo *Quake III*.

A configuração comum para os testes é a seguinte: janela de dimensões 800x600, modo de cores 32 *bits* e *z-buffer* com precisão de 24 *bits*. A tabela 5.3 resume os resultados obtidos.

É importante ressaltar que não é possível comparar diretamente esta aplicação e o jogo *Quake III*, porque o jogo realiza uma série de tarefas (efeitos gráficos especiais, processamento de Inteligência Artificial e processamento de rede, entre outras) que não estão implementadas no programa de teste.

Configuração	**Melhor taxa**	**Pior taxa**	**Taxa média**
`fullscreen = false` `vsync = false`	135	57	84,7
`fullscreen = false` `vsync = true`	61	31	55,8
`fullscreen = true` `vsync = false`	242	76	125,8
`fullscreen = true` `vsync = true`	61	42	56,2

Tabela 5.3: Resultados obtidos.

6. Conclusão

Jogos para computador são aplicações complexas que requerem a aplicação de conhecimentos de diversas áreas da Computação, como Computação Gráfica, Engenharia de *Software*, Inteligência Artificial e Redes de Computadores, entre outras.

Este trabalho apresentou alguns fundamentos sobre desenvolvimento de jogos, como uma possível classificação dos gêneros de jogos e detalhamento de alguns módulos computacionais presentes em jogos para computador.

Não existe uma padronização em relação à classificações para os gêneros de jogos, que muitas vezes são subjetivas. As classificações podem ser realizadas segundo diversos critérios, como interface, número de usuários, objetivo e jogabilidade. A classificação apresentada neste trabalho é baseada nos critérios objetivo e jogabilidade.

Jogos para computador podem ser considerados como aplicações interativas de tempo real. Dessa forma, o desempenho é um dos fatores cruciais em jogos. Por essa razão e por limitações de *hardware*, os jogos mais antigos eram desenvolvidos com linguagens de baixo nível, onde as maiores preocupações eram a apresentação visual e a eficiência de execução de código. Embora as linguagens de programação e a capacidade do *hardware* tenham evoluído consideravelmente, essa prática perdurou durante vários anos. Uma das conseqüências desse fato é a chamada síndrome "*not built here*", que refletia a prática de se implementar novamente todas as funcionalidades necessárias para a criação de um jogo a cada projeto. Entretanto, com o aumento da complexidade dos projetos de jogos, esse tipo de prática tem se tornado inviável.

Diversos tipos de soluções para combater a síndrome "*not built here*" foram conceituadas neste trabalho, como bibliotecas, *toolkits*, *frameworks* e *engines*. A principal diferença entre esses tipos de soluções é o nível de especialização apresentado por elas. As bibliotecas são a solução menos especializada enquanto os *engines* são os mais especializados.

O *framework* Guff é o resultado da aplicação de alguns dos conceitos apresentados neste trabalho sobre desenvolvimento de jogos. Esse *framework* foi concebido em duas

partes: uma camada de aplicação, que define uma possível arquitetura para jogos e o *toolkit*, que representa um conjunto de classes auxiliares para a implementação dos programas. O *framework* utiliza uma máquina de estados para modelar as possíveis etapas de um jogo. O *toolkit* oferece ferramentas diversas para os módulos de visualização, áudio, dispositivos de entrada, Matemática e utilidades.

Entre outros aspectos a se destacar, encontram-se:

- Extenso uso de bibliotecas externas para evitar a implementação de funcionalidades já existentes. Essa abordagem também torna possível cumprir um dos requisitos para o *framework* que é a possibilidade de uso em *Windows* e *Linux*.

- O paradigma de gestão automática de recursos é um dos conceitos centrais aplicados ao *framework*. A utilização desse paradigma é uma maneira simples de se minimizar erros de programação relacionados a recursos, o que contribui para aumentar a confiabilidade e robustez de uma aplicação.

- O uso da máquina de estados facilita a especificação e gestão das diversas etapas de um jogo.

6.1. Dificuldades

Ferramentas como *frameworks* são difíceis de serem projetadas pois é preciso prever quais funcionalidades serão necessárias para as aplicações.

Em relação a *frameworks* para jogos, não existe uma padronização sobre os tipos de funcionalidades requeridas. Existem locais na *internet*, como em (3D Engines, 2005), onde estão cadastrados diversos *frameworks* e *engines*. Na última data de acesso a esse local, existiam 178 ferramentas catalogadas. Muitas dessas ferramentas oferecem funcionalidades semelhantes.

Definir um *framework* que possa ser usado para o desenvolvimento de jogos de diversos gêneros é uma tarefa complicada, pois vários algoritmos e técnicas foram criadas de modo a explorar certas características dos jogos. Por exemplo, árvores BSP e portais são técnicas aplicadas em jogos de tiro em primeira pessoa, pois são adequadas para cenários fechados (que é comum nesse tipo de jogo). Já jogos que se passam em cenários abertos

(como simuladores de vôo) utilizam diversas técnicas de nível de detalhe (LOD), de modo a oferecer uma grande riqueza de detalhes em regiões próximas ao observador e pouco detalhe em regiões que estão muito distantes para serem notadas.

6.2. Trabalhos Futuros

A proposta mais trivial de melhoria para o *framework* Guff seria a incorporação de outros módulos (como Redes e Inteligência Artificial) e refinamento daqueles já existentes. Por exemplo, o módulo de Visualização permite apenas o uso de modelos 3D estáticos. Seria interessante se fosse possível usar modelos 3D com animação.

Como já foi discutido, diferentes tipos de jogos possuem necessidades diferentes. Dessa forma, uma proposta interessante seria a incorporação de um sistema de *plugins*. O objetivo desse tipo de sistema é permitir que novas funcionalidades sejam acrescentadas ao *framework* de maneira independente. Por exemplo, essa abordagem poderia ser utilizada para acrescentar módulos para a gestão de diversos tipos de cenários (abertos ou fechados). Esse tipo de sistema também poderia ser usado para a incorporação de novas tecnologias, que avançam muito rapidamente.

Anexo A. Exemplos Complementares

Neste anexo, estão listados exemplos de utilização e implementação de várias partes do Guff.

A.1. Ciclo Principal de Execução

A classe `StateAppRunner` é responsável pela execução do ciclo principal de jogo. Esse ciclo é implementado por um modelo com atualização em duas etapas (como exemplificado no capítulo 3), usando uma única *thread* para as operações de atulização e renderização. A figura A.1 ilustra esse modelo.

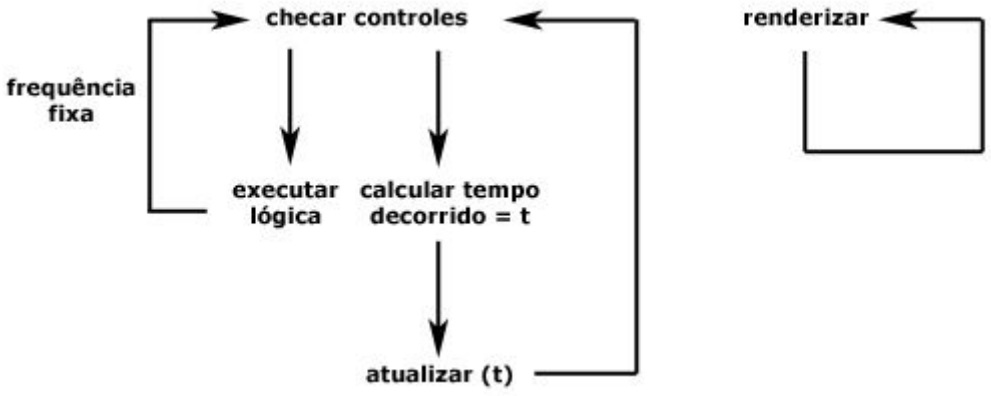

Figura A.1: Modelo com atualização em duas etapas.

A implementação é realizada segundo este trecho de código:

```
void StateAppRunner::mainLoop ()
{
  // Processar algum evento pendentes (como requisições para
  // encerramento do programa)
  processEvents ();

  // O ciclo será executado até que seja requisitado o encerramento da
  // aplicação
  float timeLastCall = mClock.getTimeMilliseconds ();

  while (mHost->terminateRequested () == false)
     {
        ...

        // verificar se é para executar a etapa de atualização de
```

```
        // freqüência fixa
        if (true == mIsFixedFrequencyUpdateEnabled)
          {
            if ((mClock.getTimeMilliseconds () - timeLastCall) >
                  mFixedFrequencyUpdateStep)
              {
                mHost->OnFixedFrequencyUpdate ();
                timeLastCall += mFixedFrequencyUpdateStep;
              }
          }

        // executar a etapa de atualização baseada em tempo
        mHost->OnUpdate (mFrameTimeStopwatch.getElapsedTimeSeconds () );

        // renderização
        mHost->OnRender ();

        ...

        // processamento de outros eventos
        processEvents ();
}
```

A execução do ciclo principal é realizada pelo método público `go ()` da classe `StateAppRunner`, que é definido desta forma:

```
void SimpleAppRunner::go ()
{
  init     ();
  mainLoop ();
  shutDown ();
}
```

O método `init ()` realiza a inicialização dos subsistemas e bibliotecas usados pelo Guff, e o método `shutDown ()`. Esse método foi implementado para dar chance às aplicações de realizar algum tipo de finalização personalizado.

Um exemplo de uso da classe `StateAppRunner` é demonstrado por este trecho de código:

```
int main ()
{
  ...
  Estado1 e1;          // declaração dos estados/grupos da aplicação
  Estado2 e2;

  StateAppRunner r;
  r.go ();
```

```
}
```

A.2. Registro de Estados

O registro dos estados no Guff é realizado pelos construtores das classes `State` e `StateGroup` com o auxílio da classe `StateRegistry`, de modo a automatizar esse processo. A associação entre estados e grupos é realizada pelo construtor dessas classes, da seguinte forma (tratamento de erros omitido):

```
State::State (const string & id, const string & parentId)
: AbstractState (id),
  mParent       (StateRegistry::Instance ()->getGroup (parentId))
{
  ...

  mParent->add (this);
}

StateGroup::StateGroup (const string & id, const string & parentId)
: AbstractState       (id),
  mParent             (StateRegistry::Instance ()->getGroup (parentId)),
  ...
{
  ...

  StateRegistry::Instance ()->addGroup (this);

  if (mParent != 0)
   mParent->add (this);
}
```

O objeto `mParent` representa uma referência para o pai do estado. Esse objeto é usado para acessar diversos serviços oferecidos pelos grupos de estados.

Durante a construção, o `StateRegistry` é consultado para recuperar a referência para o pai do estado. A seguir, é feita uma requisição ao pai para que o estado seja adicionado como seu filho. Caso o estado seja um grupo, este é adicionado no registro de estados.

A.3. Operações de Mudança de Estado

Esta seção descreve um exemplo de utilização das operações de mudança de estado (definitivas e temporárias) disponíveis no Guff.

A interface para essas operações é definida da seguinte forma:

```
class StateGroup
{
  ...

  public:

    // mudanças definitivas
    void changeState (const string & stateId);

    // mudanças temporárias
    void pushState   (const string & stateId);
    void popState    ();

  ...
};
```

As operações de mudança de estado aceitam como parâmetro o nome do estado para o qual se deseja mudar. Internamente, os grupos organizam seus filhos com o uso de uma pilha, que torna possível o uso temporário de estados. O estado atual do grupo é aquele que está no topo da pilha.

Para o restante desta seção, será usada como exemplo a hierarquia representada pela figura A.2.

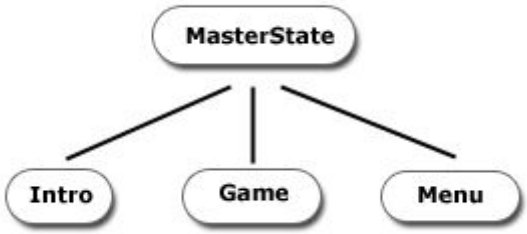

Figura A.2: Hierarquia-exemplo.

Nesse exemplo, assume-se que as classes que definem os estados possuem os nomes indicados na hierarquia. O estado “Menu” representa um estado que pode ser invocado a qualquer momento. Após o uso de “Menu”, é necessário restaurar o estado que estava em uso anteriormente. Um possível uso desses estados é demonstrado por este trecho de código:

```
void Intro::OnUpdate (float time)
{
  mTempoTotal += time;
```

```
  if (mTempoTotal > 10)
    { mParent->changeState ("Game"); return; }
}

void Game::OnUpdate (float time)
{
  comando = interpretarComandosDeEntrada ();

  if (comando == MENU)
    { mParent->pushState ("Menu"); return; }

  ...

}

void Menu::OnUpdate (float time)
{
  comando = interpretarComandosDeEntrada ();

  if (comando == MENU_SAIR)
    { mParent->popState (); return; }

  ...
}
```

O método `changeState ()` é implementado desta forma:

```
void StateGroup::changeState (const string & stateId)
{
  if (theStateIsMine (stateId) )
    doChangeState (stateId);
  else
    {
      if (mParent != 0)
        mParent->changeState (stateId);
      else
        // a transição é inválida, um erro é gerado
    }
}
```

Inicialmente, o grupo verifica se o estado requisitado é seu filho. Caso seja, a mudança de estado é realizada por `doChangeState ()`. Caso contrário, a requisição é delegada ao pai, recursivamente. Se essa requisição chegar ao topo da hierarquia e não for possível satisfazê-la, um erro será gerado.

Como foi descrito no capítulo 4, a implementação da máquina de estados permite apenas mudanças para estados irmãos, antecessores ou irmãos dos antecessores do estado. Essa restrição deve-se ao fato de que o *framework* considera como estados principais da

aplicação aqueles de nível 1 (filhos imediatos do estado mestre) e os outros como refinamentos desses estados. Isso simplifica a implementação das operações de mudanças de estado.

O método `doChangeState ()` é implementado segundo este algoritmo:

```
void StateGroup::doChangeState (const string & stateId)
{
  if (pilha não está vazia)
   (topo da pilha)->OnExit ();

  esvaziarPilha ();
  empilharEstado (stateId);

  (topo da pilha)->OnEnter ();
}
```

A operação de mudança temporária de estado é implementada de maneira semelhante à operação de mudança definitiva. Essa operação é definida assim:

```
void StateGroup::pushState (const string & stateId)
{
  if (theStateIsMine (stateId) )
    doPushState (stateId);
  else
    {
      if (mParent != 0)
        mParent->pushState (stateId);
      else
        // a transição é inválida, um erro é gerado
    }
}

void StateGroup::doPushState (const string & stateId)
{
  empilharEstado (stateId);

  (topo da pilha)->OnEnter ();

  ...
}
```

As principais diferenças entre essas operações é que a mudança de estado temporária não invoca o evento de saída do estado atual e preserva os estados empilhados anteriormente.

A operação de desempilhamento de estados, definida pelo método `popState ()`, é implementada com este algoritmo:

```
void StateGroup::popState ()
{
  // executa o evento de saída do estado e o desempilha
  (topo da pilha)->OnExit ();

  desempilhar ();

  ...
}
```

Ao contrário da operação de mudança de estado definitiva, o desempilhamento não invoca o evento de entrada para o estado anterior.

A.3. Encerramento da aplicação

Os métodos usados no processo de encerramento da aplicação são definidos por esta interface:

```
class StateGroup : public AbstractState
{
  ...

  public:

    void terminate ();
    bool terminateRequested ();

  ...
};
```

O método `terminate ()` requisita que a aplicação seja encerrada. Essa requisição percorre a árvore de estados até chegar ao topo. O método `terminateRequested ()` é usado para verificar se alguma requisição de encerramento da aplicação foi feita. A classe `StateAppRunner` usa esse método para decidir se continua a executar o ciclo principal de jogo. Um exemplo de uso dessas funções é demonstrado aqui:

```
void Estado1::OnUpdate (float time)
{
  if (condição == true)
    { mParent->terminate (); return; }
```

```
}

void StateAppRunner::mainLoop ()
{
  ...
  while (mHost->terminateRequested () == false)
     {
       // execução do ciclo
     }


  ...
}
```

Os métodos `terminate ()` e `terminateRequested ()` são implementados da seguinte forma:

```
void StateGroup::terminate ()
{
  mTerminateRequested = true;

  if (mParent)
    mParent->terminate ();
  else
   {
       // chegou ao estado raiz, então executa-se o evento de saída
       // para o estado corrente (recursivamente, os filhos
       // executarão também)
       OnExit ();
   }

}

bool StateGroup::terminateRequested ()
{
  return mTerminateRequested;
}
```

A.4. Estado Mestre

Por padrão, o estado mestre possui um estado filho pré-definido, que representa o estado vazio. O estado vazio é usado para que sempre exista um estado válido no *framework*. O único evento definido nesse estado é o `OnUpdate ()`, que requisita o término da aplicação caso seja executado.

O estado mestre e o estado vazio são implementados da seguinte forma:

```
class EmptyState : public State
{
  private:

     void OnUpdate (float time)
        {
          mParent->terminate ();
        }

  private:

     EmptyState ()
     : State (EMPTY_STATE_ID, MASTER_STATE_ID) {}


    friend class MasterState;
};


class MasterState : public StateGroup
{
   ...

  public:

    void OnEnter ()
      {
        changeState (mEmptyState.getId () );
      }


  private:

     MasterState ()
     : StateGroup (MASTER_STATE_ID, "") {}

  private:

     EmptyState mEmptyState;

};
```

A.5. Inicialização e Finalização Automáticas de Bibliotecas

Vários serviços oferecidos pelo Guff dependem de bibliotecas externas. Algumas dessas bibliotecas precisam ser inicializadas antes do uso e finalizadas quando não são necessárias. Para resolver esse problema, é utilizado o paradigma de Gestão Automatizada de Recursos. Nesse caso, o recurso a ser gerido é o estado de inicialização da biblioteca.

Uma possível implementação é descrita por este trecho de código:

```
class Initializer
{
  public:

     Initializer ()
        {
          if (sWasInitialized == false)
            {
              inicializarBiblioteca ();

              if (inicialização falhou)
                 // geração de uma exceção
            }

          sWasInitialized = true;
          sUseCount++;
        }


     Initializer::~Initializer ()
     {
        sUseCount--;

       if (sUseCount == 0)
          {
            finalizarBiblioteca ();
            sWasInitialized = false;
          }

     }

  private:
     static bool sWasInitialized;
     static long sUseCount;
};
```

Dessa forma, é possível inicializar e finalizar automaticamente as bibliotecas. O modo de uso é bastante simples, como demonstrado no seguinte trecho de código:

```
int main ()
{
  try
     {
        Initializer i;

        // uso da biblioteca

     }
  catch (...
     {  /* erro detectado; */ }

  return 0;
```

```
  // quando o fluxo de execução sair do bloco try/catch, a biblioteca
  //  será finalizada automaticamente, mesmo se exceções forem geradas
}
```

Existem diversas classes definidas de maneira semelhante a `Initializer` no Guff. A classe `StateAppRunner` utiliza várias delas para inicializar e finalizar automaticamente as bibliotecas externas.

A.6. Gestão Automatizada de Texturas

A gestão automatizada de texturas é realizada pela indiretamente pela classe `Texture2` e diretamente pela classe `TextureObject2`. A implementação é realizada desta forma:

```
class TextureObject2
{
  public:
     TextureObject2 ()
       : mId (0)
       { glGenTextures (1, & mId); }

     ~TextureObject2 ()
       { glDeleteTextures (1, & mId); }


     // seleção da textura para uso
     void bind () const
       { glBindTexture (GL_TEXTURE_2D, mId); }

  ...

  private:
     GLuint mId;  // identificador do OpenGL

};
```

A classe `Texture2`, por utilizar uma instância de `TextureObject2`, também é um objeto automático.

A leitura do arquivo de imagem em disco é implementada desta forma:

```
void Texture2::load (const std::string & filename)
{
   // leitura da imagem em disco
   DevIL::Image image (filename);
```

```
    // envio dos dados da imagem o OpenGL
    uploadImageToOpenGL (image);

    // recuperação de propriedades da imagem
    getPropetiesFromImage (image);

}
```

A leitura dos dados da imagem em disco é realizada com o auxílio da biblioteca *DevIL*, através da classe `Image` do *namespace* `DevIL` (essa classe é outro exemplo da aplicação do paradigma de gestão automática de recursos). O uso da *DevIL* possibilita que sejam usados vários tipos de arquivos de imagem, como bmp, jpeg, tga e png, entre outros. A única restrição para as imagens é que suas dimensões devem ser potência de dois (64x64, 256x128, por exemplo). Essa é uma restrição do *OpenGL*. O método `uploadImageToOpenGL ()` envia os dados da imagem para o *OpenGL*, construindo automaticamente as mipmaps dependendo do valor dos filtros de redução. O método `getPropetiesFromImage ()` recuperar várias propriedades da imagem, como altura, largura e número de *bytes* por *pixel*, entre outras.

Uma maneira alternativa para se carregar texturas em disco é utilizar o gerente de texturas. Os métodos disponíveis são equivalentes aos existentes na classe `Texture2`. Por exemplo, o método load é implementado desta forma:

```
TexturePtr TextureManager::load (const string & filename)
{
  if (isCached (filename) )
    return getFromCache (filename);
  else
    {
      // não existe no cache!
      TexturePtr t;

      ...

      t.reset (new Texture2 (filename) );

      addToCache (filename, t);


      return t;
    }
}
```

Inicialmente, é feita uma consulta ao *cache* para verificar se o elemento está presente. Caso esteja, é retornada uma referência para esse elemento. A classe `TexturePtr` simula um ponteiro com contadores de referência. Se o elemento não estiver presente no *cache*, é criada uma nova textura que é então adicionada ao *cache*.

Para conveniência do usuário, são definidas no *namespace* `Ogl` duas funções para a criação de texturas:

```
TexturePtr LoadTexture (const string & filename,
                        const TextureParameters & parameters =
                                                TextureParameters () ) )
{
  return TextureManager::Instance ()->load (filename, parameters);
}


TexturePtr LoadTextureDefault (const TextureParameters & parameters =
                                                  TextureParameters () )
{
  return TextureManager::Instance ()->loadDefault (parameters);
}
```

Nessas funções, um valor padrão para `TextureParameters` será usado caso o usuário não o especifique.

A.7. Detalhes de Implementação da classe `TrackCamera`

Esta seção detalha o algoritmo utilizado para a determinação da orientação da classe `TrackCamera`.

O método `applyTransform ()` é o responsável por definir a transformação de câmera. Na classe `TrackCamera`, esse método é implementado desta forma:

```
void TrackCamera::applyTransform ()
{
  // Inicialmente, os valores dos eixos locais são calculados
  // baseando-se na posição do alvo

  updateLocalAxis ();

  // A matriz que representa a transformação é preenchida
  // com os valores dos eixos locais. É utilizada a propriedade
  // das matrizes ortonormais em que as linhas da matriz representam
  // os valores dos valores locais X, Y e Z.
```

```
  view.a11 = localX.x;
  view.a12 = localX.y;
  view.a13 = localX.z;
  view.a14 = 0.0f;

  view.a21 = localY.x;
  view.a22 = localY.y;
  view.a23 = localY.z;
  view.a24 = 0.0f;

  view.a31 = localZ.x;
  view.a32 = localZ.y;
  view.a33 = localZ.z;
  view.a34 = 0.0f;

  view.a44 = 1.0f;

  view.applyTransform ();

  // aplica-se a transformação de translação para posicionar a
  // câmera
  glTranslatef (-position.x, -position.y, -position.z);

}
```

O método `updateLocalAxis ()` é o responsável por calcular a orientação da câmera. Os eixos locais da câmera forma um sistema de coordenadas da mão direita. Inicialmente, esse sistema de coordenadas está alinhado com o sistema de coordenadas global. Dessa forma, a orientação inicial da câmera corresponde a apontar no sentido do eixo Z negativo. O restante desta seção detalha a implementação desse método.

Como a câmera pode estar em qualquer lugar no espaço, é necessário inicialmente "traduzir" a localização desse ponto para o sistema de coordenadas da câmera. Isso quer dizer que é preciso calcular a posição do ponto em relação à câmera como se ela estivesse posicionada na origem. A partir de então, a idéia é usar uma projeção do eixo local Z no plano XZ e produtos vetoriais para calcular os outros dois eixos. Essa abordagem é usada para que a câmera não fique “de cabeça para baixo”.

```
// 1. Converter o alvo para o sistema de coordenadas da câmera. O eixo
//    Z local passará a apontar para o alvo

  localZ = mTarget;
  localZ -= position;
```

```
  localZ.normalize ();
```

A seguir, é calculada uma projeção do vetor de visão do plano XZ. A projeção nada mais é do que o vetor original com o componente y igual a zero.

```
// 2. Calcular a projeção do Z local no plano XZ

  Vector3 pxz (localZ.x, 0, localZ.z);
```

A seguir é preciso tratar alguns casos especiais. O primeiro caso ocorre quando a projeção calculada corresponde ao vetor nulo. Isso significa que o alvo está sobre o eixo Y. Nesse caso, a câmera apontará na direção do eixo Y global. Dessa forma, basta verificar qual é o sentido para o qual a câmera aponta e ajustar os outros dois eixos.

```
// 3. Verificar se a projeção é o vetor nulo (== Z local é paralelo
//    ao eixo Y padrão)

  if (pxz == Vector3::ZERO)
    {
       if (localZ.y > 0)
        {
          localZ.negate ();
          localY = Vector3::MINUS_UNIT_Z;
          localX = Vector3::MINUS_UNIT_X;
          return;
        }
       else
        {
          localZ.negate ();
          localY = Vector3::MINUS_UNIT_Z;
          localX = Vector3::UNIT_X;
          return;
        }
    }
```

O eixo local Z é invertido nesse passo porque na convenção estabelecida para a câmera, o usuário olha para o sentido negativo do eixo local Z.

O outro caso especial ocorre quando o eixo local Z está localizado no plano XZ global. Isso significa que o eixo local Y será igual ao eixo global Y. Dessa forma, o eixo local X pode ser calculado através de um produto vetorial.

```
// 4. Verificar se o Z local está no plano XZ (y = 0, ou seja,
```

```
//    Z local = projeção)

   if (localZ.y == 0)
    {
       localZ.negate ();
       localY = Vector3::UNIT_Y;
       localX = Vector3_Cross (localY, localZ);

       return;
    }
```

Novamente, o eixo local Z é invertido para que a convenção estabelecida para a câmera seja obedecida.

O caso genérico utiliza a projeção no plano XZ para calcular os eixos X e Y locais. Inicialmente, o eixo local Z é invertido para obedecer a convenção estabelecida para a câmera. A seguir, o eixo X local é calculado através de um produto vetorial entre a projeção e o eixo Z local. Para efetuar esse cálculo, é preciso saber se o eixo local Z aponta "para cima" ou "para baixo", em relação ao eixo Y global. Isso é necessário para que o cálculo do produto vetorial seja realizado de modo que câmera aponte "para cima". Logo depois, o eixo Y local é calculado através de um produto vetorial entre os eixos locais Z e X.

```
// 5. Caso genérico

   localZ.negate ();

  if (localZ.y > 0)
    localX = Vector3_Cross (pxz, localZ);
  else
    localX = Vector3_Cross (localZ, pxz);


  localX.normalize ();
  localY = Vector3_Cross (localZ, localX);

}
```

A.8. Fontes e Texto

O *namespace* `Text` do *toolkit* do Guff define algumas funções úteis para renderização de texto. Essas funções são demonstradas por este exemplo:

```
/* inicialização */

FontPtr f1 = FontManager::Instance ()->load ("arial.ttf",
                                             12,
                                             TEXTURE_FONT);

// maneira alternativa com uso de função utilitária, que invoca
// FontManager indiretamente
FontPtr f2 = LoadFont ("times.ttf", 14, OUTLINE_FONT);

...

/* renderização */

Enter2D ();

  SetPosAbs (0.0f, 0.0f);
  font1->Render ("Este texto está no canto inferior esquerdo");

  SetPosRel (0.5f, 0.5f);
  font2->Render ("A posição inicial deste texto é o centro da tela");

Leave2D ();


/* finalização */

// os recursos são liberados automaticamente
```

Referências Bibliográficas

(3D Engines, 2005) DevMaster.net. *3D Game and Graphics Engine Database*. Disponível em http://www.devmaster.net/engines/ (18/01/2005).

(3dsmax, 2004) Discreet. *Discreet 3ds max - overview*. Disponível em http://www.discreet.com/3dsmax/ (11/01/05).

(Adams, 2003) Adams, J. *Programming Role Playing Games with DirectX 8.0*, Premier Press, Boston, MA, 2003.

(Battaiola et al, 2001) Battaiola, A. L., Domingues, R. G., Feijó, B., Scwarcman, D., Clua, E. W. G., Kosovitz, L. E., Dreux, M., Pessoa, C. A., Ramalho, G. *Desenvolvimento de jogos em computadores e celulares*. Revista de Informática Teórica e Aplicada, Vol. 8, nº 2, pp. 7-46 (Outubro 2001).

(Blow, 2004) Blow, J. *Game development: Harder Than You Think*. ACM Queue, pp. 29-37 (February 2004).

(Capcom, 2004) Capcom. *Viewtiful Joe*. Disponível em http://www.capcom.com/v-joe/ (11/01/05).

(Dalmau, 2003) Dalmau, D. *Core Techniques and Algorithms in Game Programming*, New Riders, Indianapolis, IN, 2003.

(DirectX, 2004) Microsoft Corporation. *DirectX 9 SDK*, Documentação, 2004.

(Doom, 2004) Wikipedia.org. . Disponível em http://en.wikipedia.org/wiki/Doom (25/01/2005).

(Fly3D, 2004) Paralelo Computação. *Fly3D - Site Oficial*. Disponível em http://www.fly3d.com.br (11/01/05).

(Foley et al, 1990) Foley, J., van Dam, A., Feiner, S. K., Hughes, J. F. *Computer Graphics: Principle and Practice, 2nd Edition*, Addison-Wesley, Reading, MA, 1990.

(Gamma et al, 1995) Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, Reading, MA, 1995.

(Gomez, 1999) Gomez, M. *Simple Intersection Tests for Games*. Disponível em http://www.gamasutra.com/features/19991018/Gomez_1.htm (11/01/05).

(Hall, 2001) Hall, J., Loki Software Inc. *Programming Linux Games*, No Starch Press, San Francisco, CA, 2001.

(Hecker, 1996) Hecker, C. *Behind the Screen: Physics, The Next Frontier*. Game Developer, pp. 12-20 (October/November 1996).

(Jacobsen, 2003) Jacobsen, T. *Advanced Character Physics*. Disponível em http://www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml (11/01/05).

(Jaquays e Hook, 1999) Jaquays, P., Hook, B. *Quake III Arena Shader Manual Revision #12*, Documentação, 1999.

(Jaquays, 2000) Jaquays, P. *Q3Radiant Editor Manual*, Documentação, 2000.

(Jerz, 2004) Jerz, D. *Colossal Cave Adventure (c. 1975)*. Disponível em http://jerz.setonhill.edu/if/canon/Adventure.htm (11/01/05).

(LaMothe, 1999) LaMothe, A. *Tricks of the Windows Game Programming Gurus*, Sams, Indianapolis, IN, 1999.

(Lopes, 2002) Lopes, G. *Inteligência Artificial em Jogos 3D: Uma Estratégia de Busca de Caminhos no Espaço Dividido em Volumes Convexos*, Monografia de Conclusão de Curso, Universidade Federal Fluminense, Niterói, 2002.

(Lua, 2004) Lua.org. *The Programming Language Lua*. Disponível em http://www.lua.org (11/04/05).

(McCuskey, 2003) McCuskey, M. *Beginning Game Audio Programming*, Premier Press, Boston, MA, 2003.

(McGuire, 2003) McGuire, M. *Rendering Quake 3 Maps*. Disponível em http://graphics.cs.brown.edu/games/quake/quake3.html (11/01/05).

(Milewski, 2001) Milewski, B. *C++ In Action: Industrial Strength Programming Techniques - Web Edition*. Disponível em http://www.relisoft.com/book (04/03/2005).

(NBA Street, 2004) Eletronic Arts. *NBA Street Vol. 2*. Disponível em http://www.easportsbig.com/games/nbastreet2/ (11/01/05).

(Neider e Davis, 1997) Neider, J., Davis, T. *OpenGL Programming Guide, 2nd Edition*, Addison-Wesley, Reading, MA, 1997.

(OpenAL, 2000) Loki Software Inc. *OpenAL Specification and Reference*, Documentação, 2000.

(Parbery e Dunn, 2002) Parberry, I., Dunn, F. *3D Math Primer for Graphics and Game Development*, Wordware, Plano, TX, 2002.

(Perdersen, 2003) Pedersen, R. *Game Design Foundations*, Wordware, Plano, TX, 2003.

(Policarpo e Conci, 2001) Policarpo, F., Conci, A. *Real-Time Collision Detection and Response*. In Anais do SIBGRAPI 2001 p. 376, Florianópolis, 2001.

(Poyart, 2002) Poyart, E. IEngine - Uma interface abstrata para motores de jogos 3D, Dissertação de Mestrado, Dept. de Informática, PUC-Rio, Rio de Janeiro, 2002.

(Proudfoot, 2000) Proudfoot, K. *Unofficial Quake 3 Map Specs*. Disponível em http://graphics.stanford.edu/~kekoa/q3/ (11/01/05).

(Python, 2004) Python.org. *Python Programming Language*. Disponível em http://www.python.org (11/04/05).

(Quake 3, 2004) id Software. *Quake III Arena*. Disponível em http://www.idsoftware.com/games/quake/quake3-arena/ (25/01/2005).

(Quake, 2004) id Software. *Quake*. Disponível em http://www.idsoftware.com/games/quake/quake/ (25/01/2005).

(RealmForge, 2005) RealmForge Development Team. *RealmForge GDK: A .NET Game Development Platform*. Disponível em http://realmforge.com/ (02/03/2005).

(Rollings e Adams, 2003) Rollings, A., Adams, E. *Andrew Rollings and Ernest Adams on Game Design*, New Riders, Indianapolis, IN, 2003.

(Rollings e Morris, 2003) Rollings, A., Morris, D. *Game Architecture and Design - A New Edition*, New Riders, Indianapolis, IN, 2003.

(Segal e Akeley, 2004) Segal, M., Akeley, K. *The OpenGL Graphics System: A Specification - Version 2.0*, Documentação, 2004.

(Silberchatz e Galvin, 1998) Silberschatz, A., Galvin, P. *Operating Systems Concepts, 5th Edition*, Addison-Wesley, Reading, MA, 1998.

(Silva et al, 2004) Silva, A. R. V., Fernandes, C. E. C., Vale, S. L. C. *Um Sistema que Utiliza Técnicas de Inteligência Artificial e Comunicação via Redes de Computadores*, Monografia de Conclusão de Curso, Universidade Federal Fluminense, Niterói, 2004.

(Simpson, 2002) Simpson, J. *Game Engine Anatomy 101*. Disponível em http://www.extremetech.com/article2/0,1558,594,00.asp (11/01/05).

(Stroustrup, 1997) Stroustrup, B. *C++ Programming Language, 3rd Edition*, Addison-Wesley, Reading, MA, 1997.

(Unreal, 2004) Epic Games, Inc. *Unreal Technology*. Disponível em http://www.unrealtechnology.com/ (11/01/05).

(Valente e Conci, 2004) Valente, L., Conci, A. *Automatic Resource Management as a C++ Design Pattern*. In CD-ROM do Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital (I SBGames), Curitiba, 2004.

(Valente, 2002) Valente, L. *Um Modelo Orientado a Objetos para Jogos em Computador*, Monografia de Conclusão de Curso, Universidade Federal Fluminense, Niterói, 2002.

(Valente, 2004a) Valente, L. *Representação de cenas tridimensionais: Grafo de cenas*, Relatório interno de pesquisa RT-03/04, Universidade Federal Fluminense, Niterói, 2004.

(Valente, 2004b) Valente, L. *OpenGL - Um tutorial*, Relatório interno de pesquisa RT-01/05, Universidade Federal Fluminense, Niterói, 2004.

(Varanese, 2003) Varanese, A. *Game Scripting Mastery*, Premier Press, Boston, MA, 2003.

(Watt e Watt, 1992) Watt, A., Watt, M. *Advanced Animation and Rendering Techniques*, Addison-Wesley, Reading, MA, 1992.

(Wikipedia, 2004) Wikipedia.org. *Computer and video game genres*. Disponível em http://en.wikipedia.org/wiki/Computer_and_video_game_genres (11/01/05).

(Wolfenstein3D, 2004) Wolfenstein 3D Archive. *Wolfenstein 3D History and Technology Info*. Disponível em http://www.wolfenstein3d.org (11/01/05).

(Wright e Sweet, 1999) Wright, R., Sweet, M. *OpenGL Super Bible, 2nd Edition*, Waite Group Press, Indianapolis, IN, 1999.