Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

Dissertação de Mestrado

FORGE V8: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia

Charles Andryê Galvão Madeira

Recife/PE
20 de julho de 2001

FORGE V8: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia

Dissertação apresentada ao Centro de Informática da Universidade Federal de Pernambuco como requisito final para a obtenção do título de Mestre em Ciência da Computação.

Charles Andryê Galvão Madeira
(Autor)
cagm@cin.ufpe.br

Geber Lisboa Ramalho
(Orientador)
glr@cin.ufpe.br

Carlos André Guimarães Ferraz
(Co-orientador)
cagf@cin.ufpe.br

"É difícil dizer o que é impossível, pois a fantasia de ontem é a esperança de hoje e a realidade de amanhã".

Autor desconhecido

"Games are fascinating, and writing game playing programs perhaps even more so. We might say that game playing is to Artificial Intelligence as Grand Prix motor racing is to the car industry".

S. Russel & P. Norvig

Dedicação

À minha filha Júlia Luana.

Agradecimentos

A Deus, por tudo, e pela minha vida.

Aos meus pais, Gladston Madeira e Anilda Madeira, pelo incentivo, amizade, dedicação, e por me proporcionarem o ensino básico, abrindo as portas para que um dia eu conseguisse chegar até aqui.

À minha esposa, Sílvia Madeira, pelo amor, carinho, afeto e compreensão nos momentos mais difíceis de toda esta caminhada.

Ao meu professor e orientador Geber Ramalho, professor do Centro de Informática da Universidade Federal de Pernambuco, pela dedicação, confiança, orientação, e além de tudo, por sua amizade e compreensão.

Ao meu professor e co-orientador Carlos Ferraz, professor do Centro de Informática da Universidade Federal de Pernambuco, pela força depositada na idéia inicial deste projeto. Idéia esta, que surgiu num trabalho de sua disciplina, despertando o meu interesse pela área.

A todos aqueles que participam do grupo de desenvolvimento de jogos do Centro de Informática, pela motivação à ascensão desta nova área de pesquisa, e pela dedicação ao trabalho. Especialmente a Danielle Silva, Mauro Vieira, Talita Menezes, Alessandro Almeida e Mozart Filho.

Aos professores e funcionários do Centro de Informática, por terem me proporcionado um ambiente favorável à pesquisa, e consequentemente, bastante produtivo.

À Universidade Federal de Pernambuco e ao CNPq, pela ajuda financeira durante todo este período de curso.

E a todos aqueles, que de forma direta ou indireta, contribuíram nesses anos de curso, na confecção desta dissertação e no desenvolvimento do *FORGE V8*.

A todos, o meu

Muito Obrigado.

Resumo

Até o final desta última década, os jogos de computador eram desenvolvidos de maneira *ad hoc*, movidos pela prática dos *hackers* de construir aplicações a muito baixo nível, desconsiderando a reusabilidade e objetivando apenas a boa aparência e o bom desempenho. Recentemente, a indústria de jogos tem despertado para a utilização das ferramentas oferecidas pela engenharia de software, tentado buscar novas soluções através das técnicas e metodologias de desenvolvimento de projetos orientados a objetos. No entanto, devido a este atraso em relação à indústria de software em geral, muito trabalho resta a ser feito na melhoria da qualidade de desenvolvimento destas aplicações. Jogos são aplicações extremamente complexas que envolvem diversas áreas da computação e fora dela e que demandam a utilização de um *framework* para a sua implementação. Infelizmente, ainda não há uma definição clara e abrangente de um *framework* para jogos, nem tão pouco um estudo detalhado da aplicabilidade de padrões de projeto a tal *framework*. O presente projeto objetivou a concepção e implementação de um *framework*, denominado *FORGE V8*, capaz de viabilizar a construção de jogos de computador compatíveis com o padrão de mercado. O *FORGE V8* é um *framework* original e abrangente em cujo desenvolvimento foram aplicadas técnicas de engenharia de software orientada a objetos, entre as quais padrões de projeto. Além de facilitar a implementação de futuros jogos e garantir uma melhor qualidade deles, este projeto caracteriza o primeiro esforço do Centro de Informática da UFPE para dominar as tecnologias que estão realmente em voga na indústria de jogos.

Palavras-chave: Jogos de Computador, *Frameworks* e Padrões de Projeto.

Abstract

Until the end of 1990's, computer games were developed in an ad hoc way, based on hackers' method to build low level applications, disregarding reusability and focusing basically on appearance and performance. Only recently, the games' industry realized the importance of using software engineering tools, and sought for new solutions by means of the techniques and methodologies of object-oriented development. However, due to this lateness in relation to the software industry, a lot of work remains to be done in order to improve the quality of development of these applications. Games are extremely complex multidisciplinary applications which demand the use of a framework in their implementation. Unfortunately, there is not yet a clear and broad definition of a framework for games, nor a study of the applicability of design patterns in such framework. The goal of this project was the design and development of a framework, denominated *FORGE V8*, capable of making it possible the development of computer games compatible with the market's standard. The *FORGE V8* is an original and comprehensive framework developed on the basis of object-oriented software engineering techniques, including design patterns. Besides easing the development of future games and guaranteeing a better quality to this kind of application, this project is the first effort of the Center of Informatics of UFPE so as to master the technologies that are really being used in the games' industry.

Keywords: Computer Games, Frameworks and Design Patterns.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos últimos anos, jogos de computador e vídeo games têm evoluído de maneira muito rápida, acompanhando as novas tecnologias e impulsionando o aparecimento de outras. Com o avanço das redes de computadores e a popularização da Internet, grandes são os esforços para torná-los *on-line*, de forma que uma grande quantidade de usuários possam participar de uma mesma partida.

Atualmente, os jogos apresentam níveis excelentes de qualidade de apresentação e consequentemente altíssima complexidade no seu desenvolvimento. Esta complexidade se deve a necessidade de integração e interoperabilidade das novas tecnologias, além dos problemas clássicos intrínsecos a estas aplicações.

Este trabalho propõe o estudo destas tecnologias que compõem o estado da arte, e o desenvolvimento de um *framework* base para a construção de jogos de computador e aplicações multimídia. Este *framework* é conhecido no mundo dos jogos pela nomenclatura *motor*.

A seguir, serão apresentados os motivos que impulsionaram ao desenvolvimento deste trabalho, os objetivos a serem atingidos, e a estruturação desta dissertação.

1.1 Motivação

A indústria de desenvolvimento de jogos vem revolucionando o mercado da informática. Isso se deve ao fato da proliferação na venda e utilização de computadores de médio e grande porte, a revolução que a interface gráfica causou nesta última década, e a possibilidade de serem desenvolvidos para ambientes bastante atrativos e super difundidos, como o sistema operacional *Windows*. Além disso, o fato de diversas mídias poderem ser utilizadas em conjunto, e diversos usuários poderem participar de uma mesma partida ao mesmo tempo, possibilitam jogos cada

vez mais modernos, e principalmente, muito atrativos.

O mercado mundial de jogos é enorme, apresenta-se na faixa de dezenas de bilhões de dólares, e em contínuo crescimento [28]. Devido a este mercado promissor, empresas das áreas de computação e entretenimento, estão investindo pesadamente no desenvolvimento de técnicas computacionais sofisticadas, objetivando melhorar e facilitar o desenvolvimento de programas interativos, principalmente os que envolvam interfaces e animações gráficas complexas, além de vídeos, sons tridimensionais e ambientes multiusuários baseados na Internet.

Numa visão tecnológico-científica, poucas são as áreas da Ciência da Computação que utilizam os limites de hardware, de software, e das pessoas envolvidas tão quanto a de desenvolvimento de jogos. Por natureza, este tipo de atividade é multidisciplinar, envolvendo muitas áreas complexas que têm de trabalhar perfeitamente em conjunto para simular um mundo real, e em tempo real. Na computação, se inserem a Computação Gráfica, a Inteligência Artificial, a Engenharia de Software, as Redes de Computadores, algoritmos e estruturas de dados complexas, etc. Em artes, se inserem o projeto gráfico, a sonorização e trilhas musicais, os vídeos, o roteiro, a história, etc. Na modelagem, se inserem a Psicologia, a Sociologia, a Arte Militar, etc. Em entretenimento, se inserem o projeto da lógica do jogo, o projeto de níveis, o ambiente, os personagens, etc. E por último, mas não poderia faltar, a modelagem física, que ultimamente tem sido muito utilizada e proporcionado aos jogos implementações bastante próximas ao mundo real [25]. Por fim, para que uma aplicação deste tipo - limite para a computação - possa atingir um resultado satisfatório, é necessário um trabalho árduo de otimização. Devido a todos estes fatores, o desenvolvimento de jogos de computador pode ser caracterizado como um desafio acadêmico bastante interessante.

Dada a esta vasta área de pesquisa, e apesar de alguns preconceitos ainda encontrados no mundo acadêmico, existem esforços para a criação de disciplinas e até cursos inteiros dedicados aos jogos, além de revistas e conferências. Algumas universidades estrangeiras já apresentam como linha de pesquisa a área de jogos eletrônicos: University of North Texas[1] (EUA), Institut d'Informatique de Québec[2] (Canadá), Centre National d'Animation et de Design[3] (Canadá), Middlesex University School of Computing Science[4] (Reino Unido), Universisty of Teesside[5] (Reino

[1] http://larc.csci.unt.edu
[2] http://www.iiq.qc.ca
[3] http://www.nad.qc.ca
[4] http://www.cs.mdx.ac.uk/msc-cg
[5] http://www-scm.tees.ac.uk/courses/degree/compgames.html

Unido), University of Michigan[6] (Reino Unido), NgeeAnn Polytechnic[7] (Singapura). E outras duas universidades dos Estados Unidos já oferecem cursos completos de graduação: Full Sail Real World Education[8] e DigiPen Institute of Technology[9]. Todos esses esforços se devem ao objetivo de fortalecer a indústria de jogos e estreitar os laços com o mundo acadêmico.

No Centro de Informática (CIn) da UFPE existe um grupo de pesquisa, que já a algum tempo investe na área de jogos, apresentando uma experiência razoável e projetos em desenvolvimento. Além desta dissertação, duas outras dissertações de mestrado nesta mesma área já foram defendidas, referentes aos jogos *Batalha dos Guararapes* (Clauirton Siebra) e *Enigmas do Campus* (Danielle Silva). E duas novas disciplinas foram inseridas a estrutura curricular da graduação e pós-graduação - *Projeto e Implementação de Jogos* [60], e *Jogos Avançados*.

Em termos comerciais no Brasil, existem algumas empresas que já deslancharam no mercado nacional e até internacional, como é o caso da *Continuum Entertainment*[10] de Curitiba, com o jogo *Outlive*, e a *Jack in the Box Computing*[11] de Porto Alegre, com o jogo *Aquarius*. Em Recife, também já existem algumas empresas, como é o caso da *Jynx Playware*[12] que é incubada pelo Instituto de Tecnologia do Estado de Pernambuco (ITEP), e a *Art Voodoo Entertainment*[13] que já tomou vida própria.

1.2 Objetivos

Este trabalho de dissertação dá continuidade ao aprimoramento de uma idéia de projeto que nasceu em duas disciplinas do mestrado. O objetivo deste projeto inicial era criar um jogo de computador 2D multiusuário distribuído, denominado *NetMaze*. Apesar de ter rendido reflexões e publicações [45][46], o seu desenvolvimento foi baseado na utilização de tecnologias mais voltadas ao meio acadêmico, e o seu projeto era simples em relação aos jogos da atualidade. Portanto, decidiu-se por investir num novo jogo, com características gráficas 3D e projeto mais bem elaborado, utilizando-se das tecnologias de ponta da indústria de jogos. Este novo jogo

[6] http://bigfoot.eecs.umich.edu/∼soar/Classes/494
[7] http://www.np.edu.sg
[8] http://www.fullsail.com
[9] http://www.digipen.edu
[10] http://www.continuum.com.br
[11] http://www.jackbox.com
[12] http://www.jynx.com.br
[13] http://www.artvoodoo.com

recebeu o nome *Canyon*. Seu projeto foi desenvolvido baseado em idéias concebidas através de *brainstorms* realizados pelo grupo de desenvolvimento de jogos do CIn, como também através de algumas idéias extraídas de outros jogos.

Quando em sua fase de implementação, devido a enorme complexidade desta tarefa, chegamos a conclusão da grande necessidade de se utilizar um *framework* construído com base nos princípios de engenharia de software. Então alguns motores (termo como se trata um *framework* para jogos) [12][24][26] de código aberto que estavam em fase beta de desenvolvimento foram estudados para averiguar a possibilidade de serem reutilizados.

Motores são utilizados como peça chave na construção de jogos de computador, pois são responsáveis pelas características de mais baixo nível, essenciais para os jogos, concentrando todos os processamentos básicos necessários para o controle das mídias envolvidas. Um motor é nada mais que o coração de um jogo. A sua grande vantagem, é que, se construído baseado numa arquitetura geral e modular, pode ser reutilizado numa enorme variedade de jogos, e até em outras aplicações correlacionadas, de forma que estas utilizem apenas os módulos que forem necessários. Para isto, um motor deve oferecer algumas formas de configuração, de modo que cada projeto em específico consiga atingir as suas necessidades particulares, não necessitando modificação nele próprio. Desta forma, o desenvolvimento de jogos torna-se mais fácil, deixando os desenvolvedores voltados para os objetivos característicos da própria aplicação - lógica, arte, inteligência, entre outros - e não mais para os processamentos de baixo nível.

Entretanto, muitos problemas foram encontrados nos motores avaliados, arquitetura não modular, paradigma de orientação a objetos ignorado em muitos aspectos, pouca documentação, pouca reusabilidade, e baixo desempenho - alguns desses motores foram projetados para trabalhar com plataformas e/ou bibliotecas específicas.

Devido a esses fatores e a maioria destes motores serem específicos para jogos do tipo *DOOM*[14] (uso de ambientes fechados), o desenvolvimento do jogo *Canyon* (necessita de ambientes abertos) baseado em algum destes motores se tornou inviável.

Visto que nenhum dos motores satisfazia, então, objetivando adquirir mais experiência e acreditando que os princípios de engenharia de software deveriam ser utilizados extensamente, decidimos por projetar e implementar um *framework* original e abrangente para o desenvolvimento de jogos e aplicações multimídia. Este *framework*, denominado *FORGE V8*[15], deveria apresentar uma arquitetura modular

[14]http://www.idsoftware.com/archives/doomii.html

[15]Denominação concebida pela alta qualidade do motor tipo V8, acrescentado do poder ou energia de uma forja

e extensível que pudesse ser reutilizada em projetos futuros, possibilitando o desenvolvimento de jogos em 2D, 2 1/2D e 3D, diferentemente dos motores estudados, que eram apenas para jogos em 3D.

No entanto, um *framework* para jogos caracteriza um enorme desafio que normalmente não caberia em um projeto de dissertação de mestrado. A confirmação disso pode ser obtida pelo infeliz resultado do jogo *Golgotha* [26], que era um projeto comercial para o desenvolvimento de um jogo com motor proprietário 3D. Neste projeto, doze pessoas estiveram envolvidas durante dois anos de trabalho, e quinhentos mil dólares foram gastos, mas o projeto foi encerrado incompleto devido a sua não conclusão após o período preestabelecido, sendo então o código fonte liberado para o público. Isso demonstra o enorme esforço que deve ser empregado para tal tarefa, mesmo que sejam utilizados extensamente os princípios de engenharia de software. Em razão deste esforço requerido, apenas uma porção do *FORGE V8* pôde ser desenvolvida no período de mestrado. No entanto, além de facilitar a implementação de futuros jogos e garantir uma melhor qualidade deles, este projeto caracteriza o primeiro esforço do Centro de Informática da UFPE para dominar as tecnologias que estão realmente em voga na indústria de jogos, diferentemente de projetos anteriores que não eram tão ambiciosos.

1.3 Estruturação da dissertação

O restante desta dissertação está estruturado da seguinte forma. O capítulo 2 apresenta os trabalhos realizados anteriormente ao desenvolvimento do *FORGE V8*. O objetivo deste capítulo é mostrar alguns resultados de projetos anteriores e explicar todo o processo de decisões que foram levadas em conta até concluímos que a construção de um *framework* seria fundamental para qualquer projeto de desenvolvimento de jogos.

O capítulo 3 apresenta os requisitos necessários à motores utilizados no desenvolvimento de jogos, uma breve explanação das suas variações (2D, 2 1/2D e 3D), e uma avaliação de alguns motores de código aberto tratados neste projeto.

O capítulo 4 trata de apresentar a infra-estrutura necessária ao desenvolvimento de jogos: hardwares, bibliotecas multimídia, compiladores, ferramentas de modelagem, entre outros. O objetivo deste capítulo é mostrar o que há de mais recente para dar apoio ao desenvolvimento de jogos, ferramentas extremamente importantes que facilitam bastante o processo de desenvolvimento atual e elevam a qualidade das aplicações. Este capítulo é essencial para o entendimento da estruturação e

implementação do *FORGE V8*.

O capítulo 5 apresenta uma visão geral do estado da arte da Engenharia de Software, em especial, projetos orientados a objetos, com ênfase nos princípios de padrões de projeto e *frameworks*. Princípios estes, extremamente necessários ao desenvolvimento de grandes sistemas reusáveis, consequentemente ao desenvolvimento do *FORGE V8*. O objetivo deste capítulo é servir de base para o entendimento do capítulo 6, que apresenta a estruturação do *FORGE V8*.

O capítulo 6 expõe o *framework* proposto, o *FORGE V8*, detalhando a sua proposta, como desenvolvedores podem usá-lo, e os detalhes de projeto referentes as camadas e componentes que compõem a sua estruturação, com ênfase nos problemas de projeto e soluções conseguidas através do uso de padrões de projeto.

O capítulo 7 descreve os detalhes da implementação do *FORGE V8*, apresenta algumas decisões de projeto, e um estudo de caso para verificar os resultados obtidos por ele. A finalidade da implementação foi mostrar na prática como os componentes do *FORGE V8* podem ser implementados, e para servir como forma de validação.

Por último, no capítulo 8, segue-se então as conclusões e contribuições deste trabalho, e os possíveis trabalhos futuros.

Capítulo 2

Do *NetMaze* ao *Canyon*

Neste capítulo é apresentado todo o processo ocorrido desde a idéia inicial de concepção, desenvolvimento e resultados do jogo *NetMaze*. Também é mostrada a tentativa de dar continuidade a este jogo como projeto de dissertação, o que resultou na concepção do jogo *Canyon*. Do *Canyon*, jogo que deveria seguir os padrões industriais, surgiu a necessidade de se construir um *framework* para facilitar a implementação de jogos de computador em geral.

2.1 O jogo *NetMaze*

Tudo começou com uma idéia de projeto que nasceu numa disciplina do mestrado, entitulada *Sistemas Multimídia Distribuídos*. O objetivo deste projeto era criar uma aplicação multimídia e distribuída, em linguagem *Java*, que realizasse a comunicação entre os terminais envolvidos através da plataforma de distribuição *CORBA* (Common Object Request Broker Architecture). Então, foram identificadas algumas aplicações críticas, das quais decidiu-se pelo desenvolvimento de um jogo de ação multiusuário distribuído. Este tipo de aplicação foi escolhido devido aos diversos fatores que podem comprometer o seu desempenho em tempo real: consistência de dados, sincronização, interface gráfica de usuário, consumo de banda passante, tráfego de rede, interoperabilidade, entre outros [11].

Aproveitando um outro projeto de disciplina do mestrado, *Agentes Inteligentes*, um certo grau de inteligência foi atribuído aos personagens do jogo, possibilitando autonomia para a tomada de decisões das ações a serem realizadas.

Estes projetos de disciplina proporcionaram o desenvolvimento do *NetMaze*, um jogo 2D simples, mas com algumas características básicas dos jogos de alto nível (interface gráfica, sonorização, inteligência, multiusuário em rede, etc.), e resultaram

na aceitação da publicação de dois artigos [45][46].

2.1.1 Descrição do jogo

O *NetMaze* é um jogo 2D estilo *PacMan*, que consiste de personagens que se movimentam dentro de um labirinto. Os personagens são caracterizados por caçadores e por uma caça. Os caçadores têm como objetivo procurar e perseguir a caça no labirinto com a intenção de capturá-la, enquanto que a caça, tem como objetivo manter-se o maior tempo possível sem ser capturada.

Os usuários, ao se conectarem ao jogo, podem aparecer como caça ou como caçador, dependendo do estado da partida no momento da conexão. O estado do jogo é representado pelas posições dos jogadores no labirinto, pelo formato do labirinto atual - o jogo possui vários labirintos diferentes - e pelos eventos que são ocorridos. Estes eventos são colisões entre caçadores, de caçadores com a caça ou de jogadores com as paredes do labirinto. Sons específicos são executados para cada tipo de evento. O jogo possui cinco níveis, sendo cada um representado por um labirinto diferente. Os níveis se repetem na ordem de um a cinco continuamente, ou seja, após o nível cinco, o jogo retorna ao nível um. A finalização do jogo não é predefinida e fica a critério dos jogadores encerrá-lo. A passagem de um nível para outro ocorre apenas quando a caça é capturada. Logo após, o jogo é reiniciado passando para o nível seguinte, de maneira que outro jogador que era caçador passa a ser a caça. O vencedor é conhecido pelo acúmulo dos pontos obtidos durante os diversos níveis.

O *NetMaze* possui ainda uma outra funcionalidade, que é a de habilitar/desabilitar a inteligência dos seus personagens, onde o jogador pode optar por ter seus movimentos controlados ou não pelo computador. No caso do jogador ser controlado pelo computador (jogador autônomo), todas as ações são realizadas por um agente inteligente especializado [66]. O comportamento do agente depende do papel do jogador - se caça ou caçador - para qualquer nível do jogo.

2.1.2 Interface gráfica

A interface principal do jogo foi composta por três frames: no lado superior esquerdo, o frame de configurações (menu), no lado superior direito, o frame do cenário do jogo (labirinto), e na parte inferior, o frame de pontuação dos jogadores (ver figura 2.1). Existem algumas opções que os usuários podem configurar, como por exemplo, ligar e desligar o som de fundo, habilitar e desabilitar a autonomia do jogador, enviar mensagens, etc. Toda a interface é montada utilizando a técnica de *Double*

Buffering[1] para eliminar os reflexos de pisca-pisca.

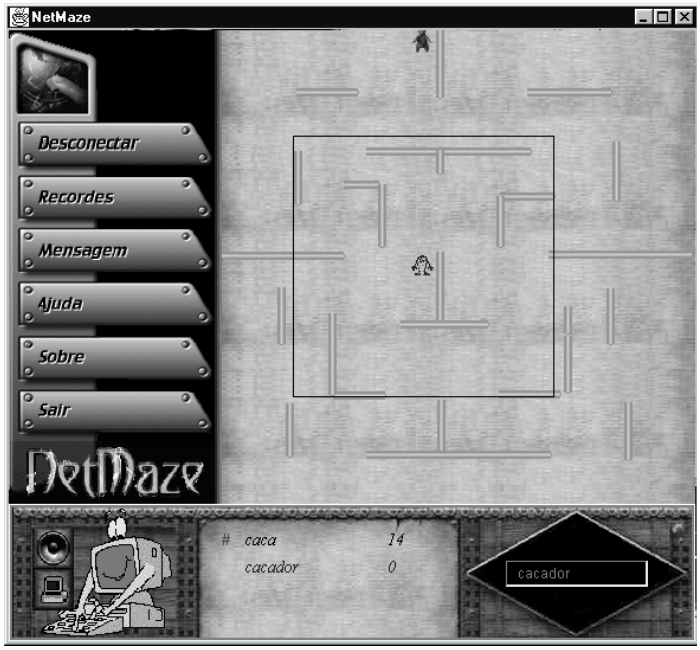


Figura 2.1: Interface principal do jogo *NetMaze*

Vários labirintos foram utilizados para diferenciar os níveis do jogo, cada um referenciando a um pano de fundo (*background*) específico. Em relação aos personagens, um urso representa a caça e um Taz (diabo da Tazmania, personagem da WB) o caçador. Os caçadores são diferenciados pela coloração da barriga do Taz.

Para facilitar a montagem da interface, foi utilizada a classe *MediaTracker* do Java, que tem o objetivo de armazenar as figuras capturadas, e criar índices para cada uma.

2.1.3 Arquitetura e implementação

O processamento do *NetMaze* foi baseado em um objeto distribuído que atua como servidor, controlando o estado do jogo, e em clientes que modificam este estado. A localização desse objeto servidor é efetuada pelos clientes através de requisições ao serviço de nomes da plataforma. Os jogadores podem se movimentar na horizontal, na vertical ou ficar parados. Essas requisições de movimentação, que representam a ação de uma tecla pressionada ou liberada, são enviadas para o servidor onde são processadas e o estado atual do jogo é atualizado.

[1] Processo de montagem de uma cena em um buffer secundário (*off-screen*) e cópia da cena montada para o buffer primário (*on-screen*) [41]

2.1.3.1 Modelos de distribuição

Todo o ciclo de comunicação, que engloba o envio de ações dos clientes para o objeto servidor, processamento, e envio de atualizações de estado do jogo do servidor para os clientes, foi projetado utilizando-se de três modelos de distribuição propostos, denominados *Cliente-Ativo*, *Call-Back* e *Canal de Eventos* [45][46], como mostrados a seguir.

No modelo *Cliente-Ativo*, os clientes enviam ao servidor seus pedidos de movimentação para serem processados e são os responsáveis por requisitar o estado atual do jogo (ver figura 2.2).

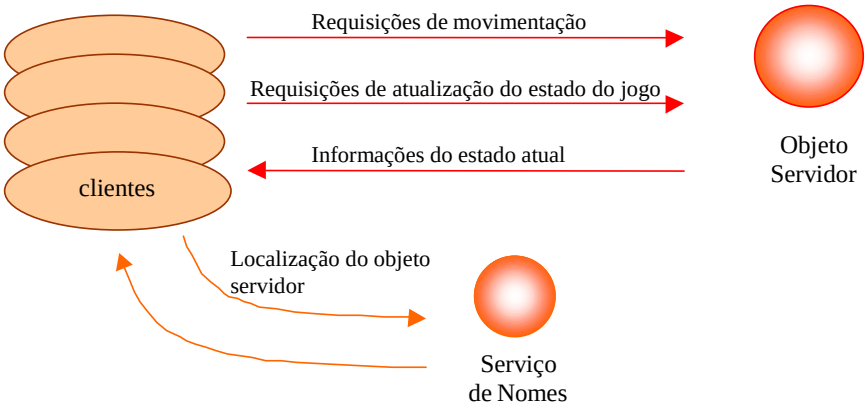


Figura 2.2: Modelo Cliente-Ativo

O objeto servidor possui uma interface com métodos responsáveis pelo envio de labirintos, posições dos jogadores e outros eventos que representam o estado atual do jogo. Esses serviços são requisitados pela aplicação cliente do jogador, através da interface do servidor, cabendo a este o retorno das informações solicitadas. As funções de apresentação de imagens, sons e texto são processadas no cliente com base nas informações de estado do jogo recebidas. A aplicação cliente faz requisições constantemente e apresenta na tela as atualizações no estado do jogo. Mesmo quando não há alterações no estado atual, os clientes têm suas telas atualizadas, gerando assim, um tráfego desnecessário na rede, bem como processamento local desnecessário para atualizar a tela.

No modelo *Call-Back*, figura 2.3, o objeto servidor passa a ser o responsável por informar aos clientes o estado atual do jogo, sem haver a necessidade de requisição pelos clientes. No entanto, essa informação só é passada quando há alteração nesse estado. Assim, quando um jogador qualquer se movimenta, após o processamento no objeto servidor, o estado do jogo é enviado pelo mesmo para todos os clientes

conectados.

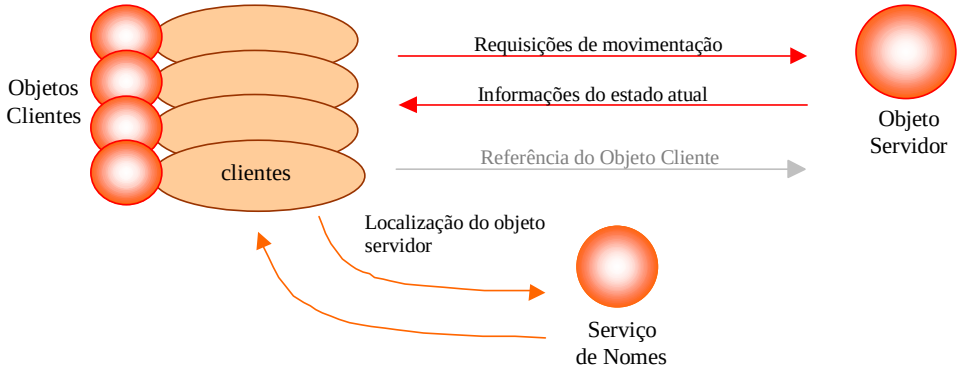


Figura 2.3: Modelo Call-Back

Uma nova interface (*callback*) é criada no cliente para que o servidor possa se comunicar com este. No início da conexão, a referência do objeto cliente é enviada para o servidor a fim de que a comunicação seja estabelecida nos dois sentidos. No servidor, é criado um novo *thread* para cada cliente conectado. Este *thread* é responsável por enviar o estado atual do jogo para seu respectivo cliente, sempre que há alguma alteração. Diferentemente do modelo *Cliente-Ativo*, no modelo *Call-Back* não há troca de mensagens entre o servidor e seus clientes se não houver mudança no estado do jogo. Assim, não haverá tráfego desnecessário de rede quando não houver movimentação dos jogadores, nem haverá atualização desnecessária de tela. As requisições de movimentação dos jogadores continuam sendo feitas à partir da aplicação cliente da mesma forma que no modelo anterior.

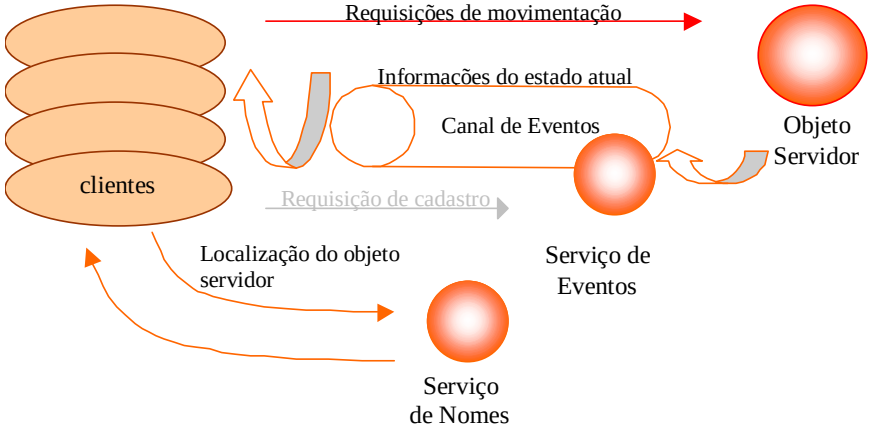


Figura 2.4: Modelo Canal de Eventos

No modelo *Canal de Eventos* (ver figura 2.4), o objeto servidor também é responsável por passar aos jogadores o estado atual do jogo, quando houver alteração

no mesmo. O serviço de eventos *CORBA* [78] é utilizado e um canal de eventos permanece aberto durante o jogo no qual os clientes se conectam para receber as informações atualizadas. Quando ocorre uma mudança de estado no jogo, o servidor gera um evento que é enviado a todos os clientes através de um *multicast* realizado pelo canal de eventos.

A principal diferença entre o modelo *Canal de Eventos* e o modelo *Call-Back* está na forma de como as atualizações são enviadas para os vários jogadores. No primeiro caso é realizado um *multicast* enquanto no segundo a comunicação se dá por conexões *unicast*.

2.1.3.2 Autonomia dos personagens

Para prover autonomia aos personagens do jogo, foram projetados e implementados agentes inteligentes, possibilitando aos personagens a percepção do ambiente através de sensores e a execução de ações através de efetuadores [66]. Para isso, os agentes possuem uma representação do mundo onde vivem, proporcionando a captura de informações do estado atual do mundo, e o que acontece a sua volta. Esta representação é codificada em termos de regras condições-ação em uma base de conhecimento. Esta base de conhecimento é executada por um motor de inferência[2].

A implementação de um agente inteligente é definida pelo seu ambiente de atuação e pelas suas funções de percepção e ação, realizando o mapeamento das seqüências de percepção para ações. As definições dos fatores usados para a elaboração dos agentes do *NetMaze* são mostradas na tabela 2.1.

Tabela 2.1: Tipos dos agentes inteligentes do *NetMaze*

Tipo	Ambiente	Objetivos	Percepções	Ações
Agente caça	Labirinto	Fugir dos caçadores	Choque c/ a parede, caçador próximo	Caminhar p/ sua direção atual, girar p/ qualquer direção
Agente caçador	Labirinto	Capturar a caça	Choque c/ a parede, choque c/ outro caçador, caça próxima, sussurro da caça	Caminhar p/ a sua direção atual, girar p/ qualquer direção

Estes agentes são classificados como cognitivos, e o seu ambiente é caracterizado por ser semi-acessível, não-determinístico, episódico, dinâmico e contínuo [66].

[2]Aplicação responsável por consultar regras de bases de conhecimento com o objetivo de executar as ações adequadas

No *NetMaze*, o processo de raciocínio dos agentes se divide em duas fases. A primeira utiliza as informações do jogo para verificar o posicionamento das paredes do labirinto e dos jogadores. A segunda decide as atitudes a serem tomadas utilizando inferência.

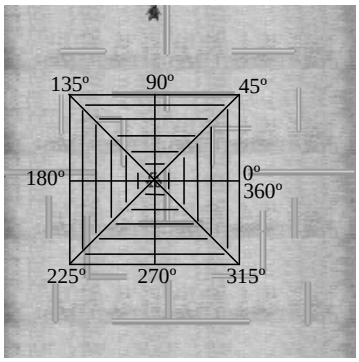


Figura 2.5: Raio de visão dividido em 8 partes de 45^o

Todas as ações dos agentes são realizadas de acordo com as percepções conseguidas através do ambiente (labirinto) dentro de um determinado alcance de visão em um raio constante (ver figura 2.5). O espaço de visão dos agentes é dividido em 8 partes de 45^o para facilitar as decisões a serem tomadas pelo motor de inferência. As regras da base de conhecimento verificam dentre os intervalos de 45^o, quais direções são livres ou não para os agentes caminharem. A figura 2.6 apresenta a arquitetura geral do *NetMaze* com personagens autônomos.

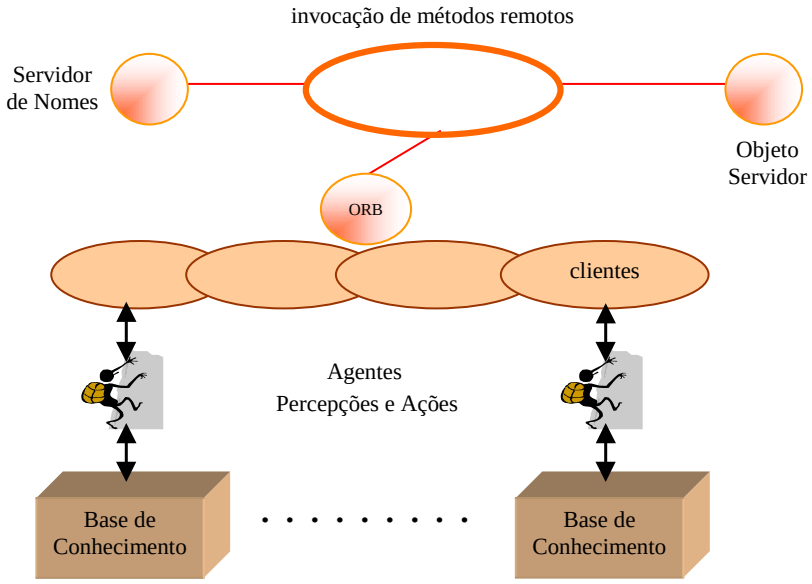


Figura 2.6: Arquitetura geral do *NetMaze* com personagens autônomos

O *NetMaze* utilizou-se da linguagem *Java* na sua implementação, acrescentada de dois pacotes extras a sua estruturação básica. O *JEOPS* [19] foi utilizado para automatização dos personagens do jogo e o *Visibroker* [78] para comunicação distribuída através de *CORBA*. Quanto à interface gráfica, à sonorização e aos dispositivos de entrada, não foram utilizadas bibliotecas extras à linguagem padrão.

2.1.4 Resultados

Com os diversos melhoramentos realizados no jogo, avaliações de tráfego de rede foram feitas em um laboratório montado de acordo com a topologia mostrada na figura 2.7. Estas avaliações tiveram o intuito de checar os resultados conseguidos pelo *NetMaze* em relação aos modelos de distribuição propostos e as capacidades dos personagens inteligentes.

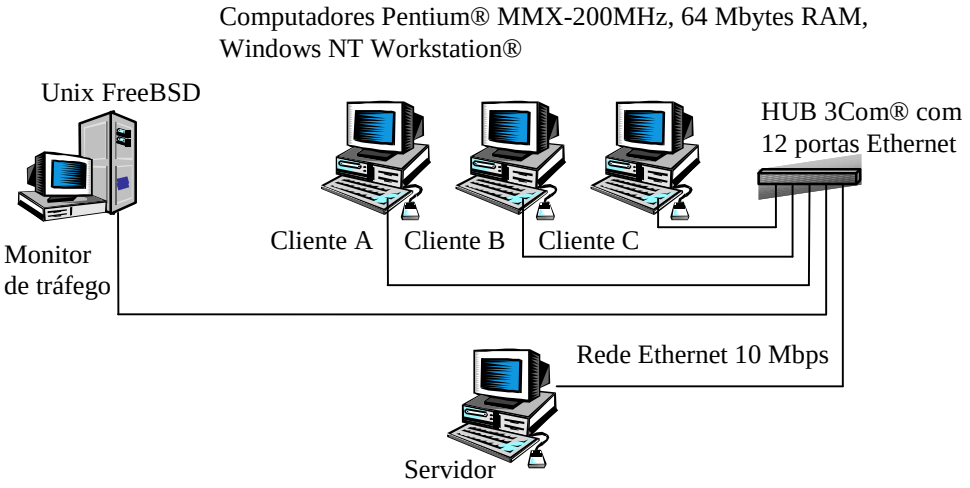


Figura 2.7: Topologia da rede

A medições foram efetuadas de acordo com as seguintes configurações: em um microcomputador servidor foi executado o ORB Visibroker v.3.4 [78], seu serviço de nomes, seu serviço de eventos (modelo *Canal de Eventos*), e a aplicação servidora do *NetMaze*; três outros computadores da rede (A, B e C), com Windows NT Workstation v.4.0, executaram a aplicação cliente do *NetMaze*; e em um quinto computador com sistema operacional Unix - FreeBSD v.3.2 foi executado o utilitário TCP-Dump v.3.4 [74], responsável por capturar as informações dos pacotes de dados na rede.

Todos os clientes se conectaram ao servidor com a condição jogador autônomo habilitada, de forma a gerar movimentos mais uniformes entre os diferentes jogadores. Esta uniformidade se deve à característica do agente inteligente conseguir

movimentos mais constantes que um jogador humano, devido ao *delay* que é causado pelo teclado.

Com o jogo em andamento, foram executadas diversas medições com duração de dois minutos cada. Nestas medições, levou-se em consideração a quantidade de pacotes na rede, o número de atualizações de estado enviadas pelo servidor em função do número de requisições de movimentação efetuadas pelos clientes e o número de eventos perdidos, ou seja, a quantidade de mudanças do estado do jogo que não foram recebidas pelos clientes, para apresentação na tela.

A rede utilizada foi isolada de qualquer tráfego externo, e pacotes não relacionados com as conexões da aplicação analisada (tráfego *NetBEUI*, *ICMP* e *ARP*) foram desconsiderados.

Os modelos *Cliente Ativo*, *Call-Back*, e *Canal de Eventos*, analisados neste trabalho, apresentaram aproximadamente o mesmo nível, qualidade de apresentação e sincronismo de movimentos verificados a olho nu na tela de todos os clientes.

2.1.4.1 Tráfego de pacotes

A primeira análise verificou o tráfego de rede em relação ao número de pacotes enviados e recebidos pelos clientes, sem levar em conta o tamanho do campo de dados de cada pacote.

Durante a execução do jogo não há tráfego entre clientes, apenas entre clientes e servidor. Os pacotes enviados pelos clientes podem ser divididos em dois tipos: requisições de movimentação e solicitações de atualização de estado. Já os pacotes recebidos pelos clientes correspondem às informações sobre o estado atual do jogo e contêm as atualizações de tela.

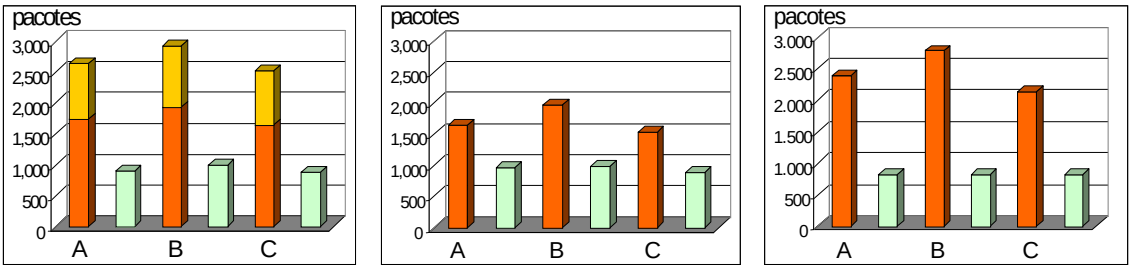


Figura 2.8: Gráficos do tráfego de pacotes nos clientes A, B e C referentes aos modelos de distribuição. A barra escura representa o número de pedidos de movimentação, e a clara, o número de atualizações recebidas

Os gráficos da figura 2.8 mostram os resultados comparativos entre os três modelos de distribuição avaliados.

O modelo *Cliente Ativo* apresentou um tráfego adicional no sentido cliente-servidor representado pelas solicitações de atualização de estado. Já o modelo *Call-Back*, apresentou uma redução de cerca de 34% no total de pacotes enviados no mesmo sentido uma vez que o processo de atualização de estado passou a ser controlado pelo servidor sem a necessidade de solicitações por parte dos clientes. O número de atualizações do estado do jogo enviadas do servidor para os clientes não apresentou diferença para os dois modelos. Para o modelo *Canal de Eventos*, os clientes realizaram mais pedidos de movimentação, enquanto houve uma pequena redução no número de atualizações de estado enviadas pelo servidor.

2.1.4.2 Perda de eventos

Em seguida analisamos o somatório do número de solicitações de movimentação dos três clientes e o número de atualizações de estado enviadas pelo servidor para cada cliente (ver figura 2.9).

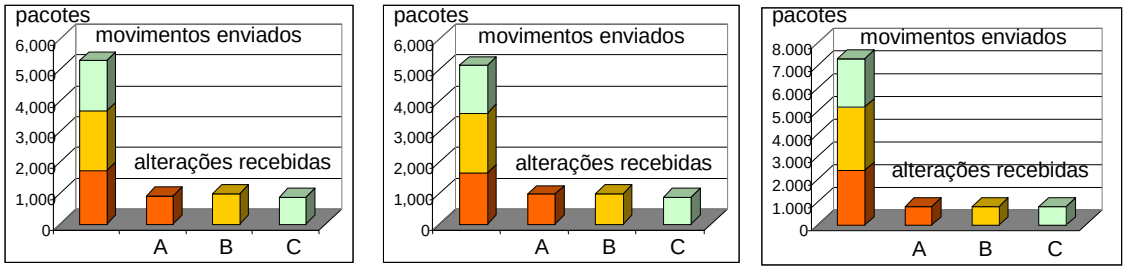


Figura 2.9: Gráficos da perda de eventos dos clientes A, B e C referentes aos modelos de distribuição. A barra de movimentos enviados apresenta três tonalidades referentes aos três clientes

Considerando que cada movimentação dos jogadores representa uma mudança no estado do jogo, poderia-se pensar que o número de respostas do servidor deveria corresponder ao número total de pedidos de movimentação dos jogadores. No entanto, como em um jogo de ação ocorrem freqüentes pedidos de movimentação de vários clientes em um mesmo instante, as mudanças no estado do jogo referentes a essas movimentações podem ser repassadas para os clientes em um único pacote.

Para avaliar a perda de eventos com base nisso, pode-se considerar o pior caso aquele que é representado pelo maior número de pedidos de movimentações enviados (cliente B) e pelo menor número de atualizações de estado recebidas (cliente C). A

perda de eventos, então, foi da ordem de 45%, na pior situação. Ainda assim, para o jogo utilizado, tal perda não foi significativa, já que não afetou a consistência de atualização de tela dos usuários verificada a olho nu. Isto pode ter ocorrido devido a simplicidade do cenário da aplicação. Estes valores foram alcançados após medições e ajustes. Uma maior taxa de recepção de atualizações de estado não afetou a qualidade de apresentação para o usuário, em virtude da não percepção visual desses movimentos adicionais.

Vale acrescentar que as medições apresentadas nos gráficos acima foram feitas com o jogo em sua capacidade máxima de atividade, ou seja, todos os clientes enviando pedidos constantes de movimentação. Em uma outra situação, onde os jogadores se movimentem mais lentamente, a taxa de perda de eventos será significativamente menor, podendo chegar a ser nula.

2.1.4.3 Consumo de banda passante

Para efeito de avaliação do consumo de banda exigido pelo jogo em questão, utilizando as mesmas medições anteriores, computou-se o tamanho, em bits, de todos os pacotes referentes à aplicação.

Os gráficos da figura 2.10 mostram os resultados obtidos, representados em Kbits por segundo (Kbps), para os dados enviados ou recebidos pelos clientes durante o tempo de medição.

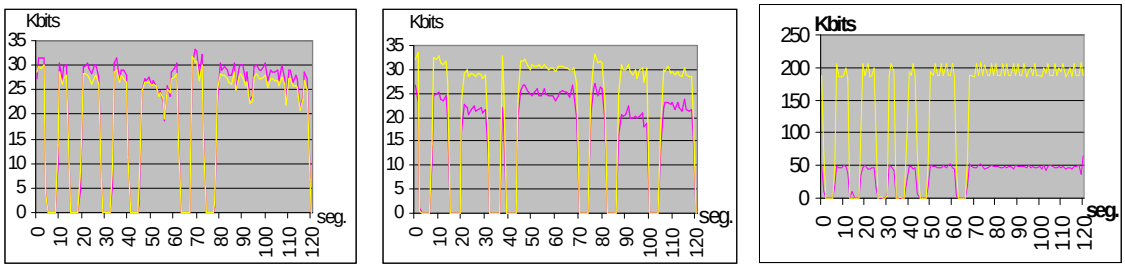


Figura 2.10: Gráficos de consumo de banda passante referentes aos modelos de distribuição. A linha escura representa as mensagens enviadas pelos clientes. A linha clara, as mensagens recebidas

Pode-se notar que, em relação ao consumo de banda, os dois primeiros modelos apresentaram, em média, o mesmo desempenho, apesar da redução significativa do número de mensagens enviadas no sentido do cliente para o servidor no modelo *Call-Back*, como mostrado anteriormente. Isto se dá em virtude do aumento de

mensagens do protocolo *GIOP* utilizado pelo *CORBA* para seu controle interno de conexões TCP/IP [78]. Como no modelo *Call-Back* é utilizado um maior número de objetos distribuídos que no *Cliente Ativo*, essas mensagens chegaram a ocupar a rede quase à mesma proporção - em termos de consumo de banda - que os pacotes que deixaram de ser enviados pelo cliente neste modelo. O modelo *Canal de Eventos* apresentou um tráfego bem maior que os outros modelos. A razão disto também se deve a quantidade de mensagens de *GIOP* que aumentaram enormemente.

Lembrando mais uma vez, esta situação apresentada vale apenas para o pior caso de consumo de banda, ou seja, com todos os clientes efetuando solicitações constantes de movimentação ao servidor, que por sua vez retorna atualizações de estado a uma taxa máxima aceitável para apresentação aos jogadores.

Os intervalos em que o tráfego cai para zero representam as paradas do jogo nos instantes em que a caça é capturada, o que se mantém por aproximadamente cinco segundos, indicando aos jogadores a finalização do nível.

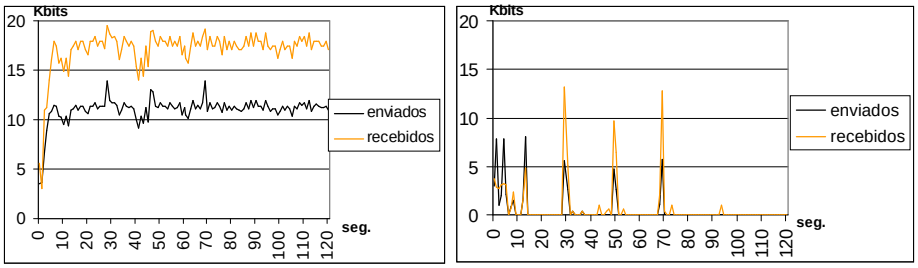
Levando-se em consideração apenas o tráfego no sentido do cliente para o servidor, a diminuição no consumo de banda no modelo *Call-Back* é claramente notada.

As medições realizadas mostraram que a taxa de transmissão máxima requerida pelo *NetMaze* é da ordem de, aproximadamente, 33 Kbps nos modelos *Cliente Ativo* e *Call-Back*, enquanto que no modelo *Canal de Eventos*, é da ordem de, aproximadamente 200 Kbps. Isto demonstra a possibilidade, nos dois primeiros modelos, dessa aplicação ser executada na Internet, inclusive através de uma conexão discada.

Para avaliação dos modelos propostos em uma situação diferente de comportamento dos jogadores, foram efetuadas as mesmas medições de tráfego executando o jogo com menor número de eventos enviados pelos clientes. Durante os dois minutos de teste, efetuou-se breves movimentações dos jogadores nos clientes A, B e C por volta dos instantes 30, 50 e 70 segundos, respectivamente, para que fosse medido o tráfego gerado. O intuito deste teste foi determinar o comportamento dos modelos *Cliente Ativo* e *Call-Back* em jogos com características de pouca ação. Os gráficos da figura 2.11 mostram os resultados obtidos.

Esta situação de pouca movimentação é típica em jogos baseados em turnos, como por exemplo o jogo de xadrez, onde não há pedidos constantes de movimentação e nem necessidade de atualizações de estado a cada instante. Por outro lado, uma mudança do estado do jogo deve ser informada imediatamente a todos os clientes.

Neste caso, pode-se notar claramente que o modelo *Call-Back* mostra-se mais indicado por não apresentar tráfego desnecessário na rede quando não há movi-



Cliente Ativo Call-Back

Figura 2.11: Gráficos de envio e recepção de pacotes em dois modelos propostos

mentação dos jogadores. Os picos de tráfego observados no modelo *Call-Back* no início das medições representam as mensagens *CORBA* de pedidos de localização e conexão entre objetos remotos na inicialização do jogo.

2.2 A continuidade do *NetMaze* como projeto de dissertação

Uma vez concluídos os projetos das disciplinas, passou-se a se pensar no *NetMaze* como um projeto de dissertação. Para isso, foram realizados diversos estudos no estado na arte do desenvolvimento de jogos. Nestes estudos detectou-se que o *NetMaze* utilizava de tecnologias muito mais acadêmicas que industriais para este tipo de aplicação, além de ser um jogo muito simples e sua *jogabilidade*[3] não ser boa e compatível com a realidade dos jogos modernos. Então, foi iniciado um processo de verificação de quais os melhoramentos que deveriam ser realizados a este jogo no intuito dele se tornar realmente competitivo, segundo o padrão do mercado.

Chegou-se a conclusão que seria necessário fazer uma completa modificação neste jogo, desde o seu projeto (tema, personagens, roteiro, etc.) até o seu processo de desenvolvimento e ferramentas utilizadas na implementação. Portanto, decidiu-se pelo desafio de se construir um novo jogo que proporcionasse uma melhor interação e apresentação ao usuário, constando de cenários maiores e mais bem definidos, personagens mais envolventes, ambiente e lógica mais realista, de maneira que os usuários sentissem um maior entusiasmo ao jogá-lo. Para este novo jogo, seria também essencial a utilização de tecnologias de ponta, bibliotecas multimídia padrão que compunham o estado da arte do desenvolvimento de jogos e aplicações multimídia, o

[3]Variável encontrada nos jogos referente ao nível de interação e entusiasmo por parte dos jogadores

que aumentaria enormemente a complexidade deste desafio, devido principalmente a não haver o domínio destas tecnologias no CIn na época.

2.3 O jogo *Canyon*

Para projetar um jogo, a criatividade é fundamental, idéias devem surgir através da inspiração a um determinado objetivo, sendo sintetizadas até chegar a uma convergência. Isto pode demandar tempo, mas é fundamental para o seu sucesso [65].

Os jogos apresentam uma classificação dividida em categorias - ação, aventura, estratégia, entre outras - que é uma forma de melhor entendimento da sua natureza e de identificação dos parâmetros necessários para a sua análise. Eles podem pertencer a uma única categoria ou apresentar propriedades que possam levá-los a serem classificados em mais de uma. Para a definição do tema do novo jogo, uma pesquisa foi realizada através de um estudo detalhado em diversos outros jogos que participassem de uma mesma categoria, a de jogos de ação. A finalidade deste estudo foi analisar as características e boas idéias já existentes, para incorporá-los ao novo jogo, como também, a definição dos seus personagens, e a sua própria história (principais fatos, causas e conseqüências). A partir daí, iniciou-se o processo de concepção deste novo projeto. Ao qual através de sessões de *brainstorms*, decisões críticas foram tomadas, orientando o jogo para uma melhor viabilidade e jogabilidade. Outras abordagens referentes a interface gráfica, sonorização, inteligência dos personagens e comunicação em rede também foram analisadas, para possibilitar a construção de um cenário bem definido, e composto por personagens dotados de um poder de decisão bastante evoluído.

Assim, bastante melhorado em relação ao *NetMaze*, e apresentando resultados muito mais aproximados aos jogos da atualidade, foi concebido o jogo *Canyon*. *Canyon* é um jogo de ação-inteligência, o qual seu período de concepção inicial durou cerca de três a quatro meses e continuou a evoluir a medida que surgiam novas idéias. O seu tema se baseia num acontecimento imaginário que relata uma batalha entre dois povos (*Zagotha* e *Xarix*) que lutam pela posse de um planeta (*Canyon*) rico em minérios energéticos. A energia destes minérios é quem garante a continuidade da espécie. Os personagens do jogo são caracterizados por aeronaves controladas pelo computador (*Bright*) e aeronaves controladas pelos jogadores humanos (*Force*). Os objetivos dos jogadores são impostos através de missões estabelecidas nos diversos níveis.

O projeto detalhado do *Canyon* encontra-se no apêndice A desta dissertação.

Seu projeto abordou os seguintes itens: a sua classificação baseada nas diversas categorias de jogos; o seu objetivo, estabelecendo as fantasias a serem suportadas e as emoções a serem produzidas nos jogadores; o seu tema (*backstory*), meio pelo qual se expressa o ambiente no qual o jogo se desenvolve; as pesquisas e decisões de projeto, responsáveis pelo detalhamento do tema e decisões ligadas à perspectiva, ao número de participantes, à temporização e ao nível de realismo; a sua estruturação, responsável pelo detalhamento do ambiente, personagens e regras gerais; e a sua interface gráfica de usuário, responsável pelas configurações de controle da aplicação, e pelo cenário.

2.4 A necessidade de um *framework* para o desenvolvimento de jogos

Com o projeto do *Canyon* definido, o seu processo de implementação pôde ser iniciado. Para isto, estudos foram realizados em diversas arquiteturas e implementações de jogos com o intuito de verificar se alguma destas arquiteturas poderia ser reutilizada de maneira satisfatória para o desenvolvimento do *Canyon*.

Para o desenvolvimento de um jogo, o ideal é projetar e implementar um *framework* que possa ser reutilizado por diversos projetos de jogos, ou utilizar um já desenvolvido por terceiros que se adapte as condições exigidas. Desta forma, descarta-se a possibilidade que uma grande porção de código seja rescrita diversas vezes, e ainda possibilita o controle geral e padronização de toda a aplicação. A grande meta deste tipo de *framework* é salvar o projeto do aparecimento de grandes problemas futuros e perda de tempo, principalmente no caso de um projeto muito extenso. Devido a todos esses fatores, pode-se caracterizar um *framework* como uma peça fundamental para jogos, do mesmo modo que em qualquer grande projeto de software de outra natureza.

Neste estudo, constatou-se que, ao contrário da indústria de software em geral, a indústria de jogos, despertou apenas no final da última década para a utilização das diversas técnicas, metodologias e processos oferecidos pela engenharia de software para a construção de software de melhor qualidade (modularidade, reusabilidade, extensibilidade, boa documentação, entre outros) [49]. Isso se deve em princípio, a indústria de jogos ter como seu objetivo principal apenas a qualidade da interface gráfica e o bom desempenho da aplicação, acarretando na geração de código de baixo nível e totalmente dependente [65].

No entanto, devido principalmente à concorrência acirrada do mercado de entre-

tenimento, a indústria de jogos tem buscado cada vez mais produzir exemplares de boa qualidade em períodos mais curtos de desenvolvimento. Isso não é tão simples dado que existe uma alta complexidade envolvida no desenvolvimento de jogos devido não só à necessidade de integração e interoperabilidade de novas tecnologias, mas também aos problemas clássicos referentes à distribuição, inteligência, sonorização, sincronização, construção e renderização[4] de cenários em tempo real, entre outras propriedades intrínsecas.

Dessa forma, os especialistas em projeto de jogos têm tentado buscar soluções em diversas áreas de conhecimento, dentre elas, a engenharia de software, a qual por muito tempo havia sido ignorada [65]. Os resultados desta "guinada" da indústria de jogos ainda são preliminares. Muito trabalho resta a ser feito. Por exemplo, ainda não há uma definição clara e abrangente de um *framework* geral para jogos, nem tão pouco um estudo da aplicabilidade de padrões de projeto a tal *framework*.

No mercado existem alguns *frameworks* (motores) *freeware* desta natureza em desenvolvimento, como por exemplo o *Genesis3D* [24], o *Crystal Space* [12], o *Golgotha* [26] e alguns outros comerciais voltados exclusivamente a atender funcionalidades específicas - gráficos, personagens, sons ou modelagem física. No entanto, segundo nossas avaliações (detalhadas no capítulo 3), sobretudo nos três primeiros motores, há uma série de problemas: arquitetura não modular, paradigma de orientação a objetos ignorado em muitos aspectos, pouca documentação, pouca reusabilidade, e baixo desempenho - alguns desses motores foram projetados para trabalhar com plataformas e/ou bibliotecas específicas.

Devido a esses fatores e a maioria destes *frameworks* serem específicos para jogos do tipo *DOOM* (uso de ambientes fechados), o desenvolvimento do jogo *Canyon* (necessita de ambientes abertos) baseado em algum destes *frameworks* se tornou inviável.

A avaliação destes *frameworks* demandou bastante tempo, mas ela serviu para extrair algumas boas idéias, e também para obter um fato muito importante: é praticamente inexistente o uso de padrões de projetos [23] na maioria destes *frameworks* mesmo sabendo que os mesmos apresentam situações próprias para seu uso.

Visto que nenhum dos *frameworks* satisfazia, e acreditando que os princípios de engenharia de software deveriam ser utilizados extensamente, decidimos por projetar e implementar um *framework* original e abrangente para o desenvolvimento de jogos e aplicações multimídia, denominado *FORGE V8* e detalhado no capítulo

[4] Processo do sistema de computação gráfica que é responsável por desenhar os polígonos de um modelo gráfico usando a sua informação de estado corrente

6. Esse *framework* tem como objetivo reduzir o tempo de desenvolvimento, riscos e complexidades exigidos a essas aplicações. A nossa intenção foi de fornecer aos desenvolvedores de jogos facilidade de uso, reusabilidade e modularidade - características essenciais para aplicações de boa qualidade.

Outros fatores importantes que nos levaram a esta decisão foram, o não conhecimento de artigos que resumam os principais padrões de projeto aplicáveis ao desenvolvimento de jogos, a necessidade de independência em relação aos *frameworks* existentes, e a intenção de adquirir uma boa experiência desenvolvendo um novo *framework*, e não apenas utilizando um já parcialmente desenvolvido. Além disso, os *framework* de código aberto ainda são apenas versões beta, não garantem estabilidade e não têm previsão da resolução dos problemas existentes. Um grande exemplo deste tipo de problema ocorreu recentemente com o motor *Crystal Space*, o mais bem estruturado e mais bem documentado dos citados acima, que modificou a base da sua arquitetura na sua última liberação de versão (beta), caracterizando uma modificação significativa nos projetos que o utilizavam.

Com este novo objetivo definido, a construção do *FORGE V8*, decidimos adiar a implementação o jogo *Canyon* para após a concretização deste *framework*.

2.5 Conclusão do capítulo

Os projetos desenvolvidos nas disciplinas que deram origem ao jogo *NetMaze* foram de grande valia, pois serviram de alavanca para a criação de novos projetos, e de motivação à pesquisas mais aprofundadas na área de jogos, como é o caso deste trabalho de dissertação. Além de resultarem na publicação de dois artigos, um nacional e um internacional.

A transição do jogo *NetMaze* para o jogo *Canyon* teve o objetivo de propiciar o desenvolvimento de um jogo realmente compatível com a indústria de jogos atual e competitivo de acordo com o padrão do mercado. Isto ainda não havia sido feito no Centro de Informática em projetos anteriores (jogos Batalha dos Guararapes & Enigmas do Campus) que se baseavam em *VRML* x *Java3D*.

O jogo *Canyon* foi concebido através do aproveitamento de boas idéias de outros jogos da sua mesma categoria e a inserção de novas características para a definição da sua própria identidade. Este processo de concepção demandou bastante tempo e esforço para atingir a sua primeira versão aceitável.

Para a implementação do *Canyon* precisávamos de um *framework* que facilitasse esta tarefa, descartando a necessidade de reescrever código, e apresentando um bom

nível de controle e padronização para o jogo. Mas nenhum dos *frameworks* estudados se mostrou plenamente satisfatório para as nossas necessidades, além de não utilizarem as técnicas de engenharia de software que garantem aplicações de boa qualidade.

Por estas razões, e em busca de adquirir mais experiência, aceitamos o desafio de construir nosso próprio *framework*, apesar disto ser uma tarefa complexa e árdua. Para que esta tarefa resulte em um bom produto, uma equipe de especialistas é normalmente envolvida no desenvolvimento.

Capítulo 3

Motores para Jogos de Computador

Jogos de computador são aplicações extremamente complexas. Elas necessitam de diversas ferramentas trabalhando em conjunto para obter um bom grau de qualidade e uma maior facilidade no seu desenvolvimento. Dentre estas ferramentas, os motores (*frameworks* para jogos) são os responsáveis por oferecer as funcionalidades básicas de um jogo fornecendo abstrações das características de mais baixo nível como as bibliotecas e APIs utilizadas no desenvolvimento. Além disso, como qualquer outro software, se exige de um motor níveis satisfatórios de desempenho, confiabilidade, robustez, modularidade, e reusabilidade de código [16]. Este capítulo apresenta como se dá a constituição e o funcionamento de um jogo de computador com o objetivo de apresentar como se enquadra e qual a importância de um motor em relação a um jogo. Em seguida, expõe a composição de motores para jogos e suas variações de acordo com as formas de representação gráfica (2D, 2 1/2D e 3D), e apresenta uma avaliação dos motores de código aberto que foram estudados com o objetivo de serem reutilizados para viabilizar o desenvolvimento do jogo *Canyon*.

3.1 A constituição de um jogo de computador

Jogos de computador são constituídos por três extensos componentes: o enredo, uma interface interativa e um motor. O sucesso de um jogo está associado a combinação perfeita destes componentes [41].

O enredo define o tema, a trama, e os objetivos do jogo, segundo quais os usuários devem se esforçar para realizar uma série de atividades. A definição da trama não envolve só criatividade e pesquisa sobre o assunto, mas a interação com pedagogos,

psicólogos, roteiristas, cineastas e especialistas no assunto a ser enfocado [33].

A interface interativa controla a comunicação entre o motor e o usuário, reportando a apresentação do jogo para este último [33]. O desenvolvimento da interface envolve aspectos artísticos, cognitivos e técnicos. O valor artístico de uma interface está na capacidade que ela tem de valorizar a apresentação do jogo, atraindo usuários e aumentando a sua satisfação ao jogar. O aspecto cognitivo está relacionado à correta interpretação da informação pelo usuário. O aspecto técnico envolve desempenho, portabilidade e a complexidade dos elementos [33]. Estes componentes dos jogos são construídos auxiliados por ferramentas específicas a cada tipo de mídia (modeladores de objetos, editores de imagens, editores de cenários, processadores de sons, etc.).

O motor (ou *framework*) de um jogo é o seu sistema de controle, o mecanismo que controla a reação do jogo em função das ações dos usuários. A implementação de um motor, envolve diversos aspectos computacionais, tais como, a escolha apropriada da linguagem de programação em função da sua facilidade de uso, eficiência e portabilidade; o desenvolvimento de algoritmos específicos, serviços para a gerência e renderização de cenários, controle de personagens e de mundos, gerência de janelas das aplicações, gerência de sons, suporte a rotinas matemáticas, estruturação e classificação de dados, mecanismos de comunicação, sincronização, modelo de interface com o usuário, dentre outros. A sua grande vantagem, é que, se construído baseado numa arquitetura geral e modular, pode ser reutilizado numa enorme variedade de jogos, e até em outras aplicações correlacionadas, de forma que estas utilizem apenas os módulos que forem necessários [65]. O relacionamento entre estes componentes essenciais aos jogos é ilustrado na figura 3.1.

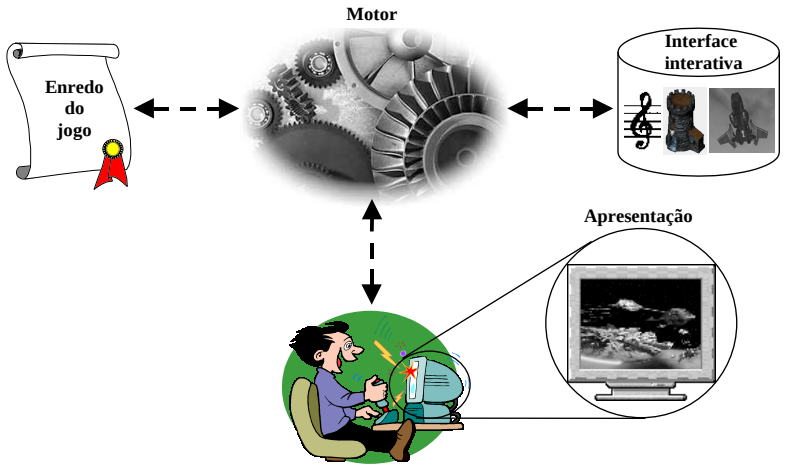


Figura 3.1: Relacionamento entre os componentes que constituem os jogos

Geralmente, um jogo de computador comercial demanda um grande tempo de desenvolvimento, aproximadamente de 2 a 3 anos para serem concluídos por uma equipe relativamente grande, e a complexidade de desenvolvimento cresce em função da categoria e gênero ao qual o jogo pertence [25].

Outro ponto importante é que na criação de novos jogos, surge sempre um novo conjunto de problemas a serem resolvidos devido, principalmente, à introdução de novas tecnologias e propriedades, como por exemplo, novas bibliotecas de renderização gráfica ou novas formas de modelagem física dos elementos do jogo. De uma maneira geral, os jogos devem fornecer aos jogadores, no mínimo, o mesmo desempenho e qualidade oferecidos pelos últimos jogos lançados no mercado, além de novas características. Naturalmente, isso tudo com a pretensão de atrair um maior público de jogadores.

Por estas razões, é extremamente relevante encontrar ferramentas adequadas a fim de agilizar o processo de desenvolvimento de jogos. Além disso, essas ferramentas devem ser de fácil extensão, reutilização e modularidade dado o grande dinamismo dessa área de software que exige sempre o suporte e adição de novas tecnologias e propriedades.

Em relação ao funcionamento, um jogo de computador se dá pela composição de um laço contínuo que executa uma determinada lógica e apresenta um resultado ao usuário, normalmente a uma taxa de 30 ou mais repetições por segundo [41]. Esse laço é responsável por gerenciar a execução dos diversos componentes responsáveis pelas tarefas da aplicação (motor + lógica do jogo) descritas a seguir e ilustradas pela figura 3.2.

Inicialização Responsável por executar as operações padrões de inicialização de um jogo. Alocação de memória, aquisição de recursos, leitura de dados do disco rígido, entre outros.

Início do laço A execução do código entra no laço principal. Local onde as ações se iniciam e continuam até que o usuário invoque a saída.

Entrada de dados do usuário As informações de entrada dos usuários são processadas e/ou armazenadas para serem utilizadas na seção de lógica e Inteligência Artificial (IA).

Execução da IA e lógica do jogo Esta seção contém a maior parte do código do jogo. A inteligência artificial, a modelagem física, e a lógica são executados.

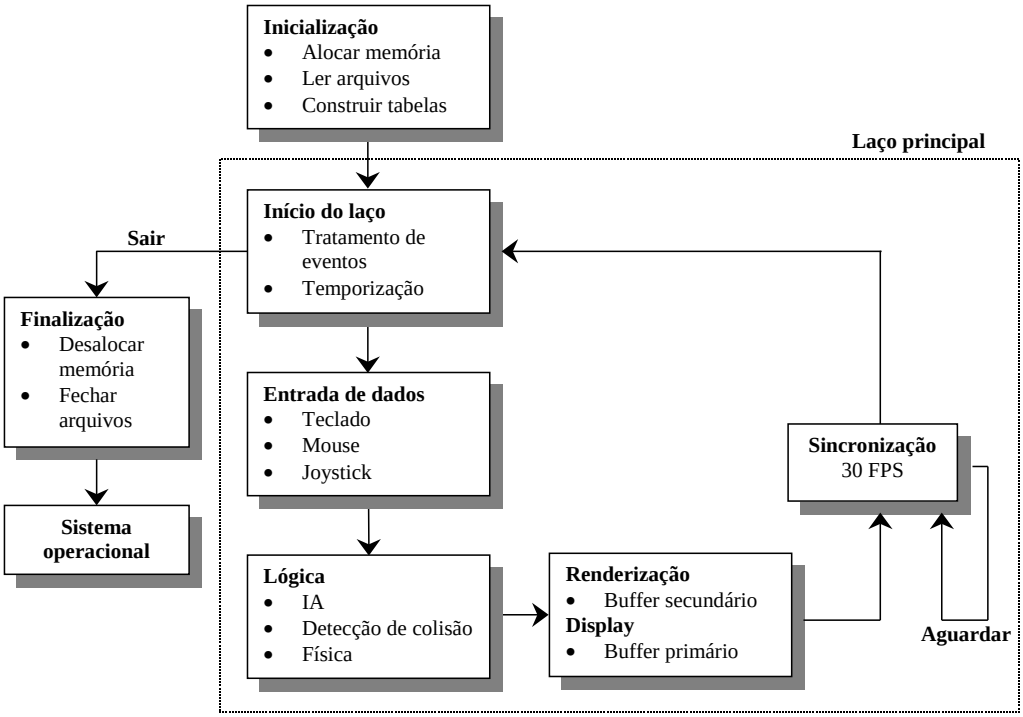


Figura 3.2: Composição do laço principal de um jogo

Renderização Os resultados das informações de entrada dos jogadores e execução da IA e lógica são utilizados para a apresentação ao usuário, gerando o próximo quadro de animação para o jogo.

Sincronização Devido a variação de desempenho apresentada pelos computadores, principalmente de acordo com a complexidade de jogo específico, este deve ser sincronizado, através de temporizadores, para uma taxa de apresentação satisfatória.

Laço Retorna ao início do laço e executa tudo novamente.

Saída Esta seção corresponde ao fim da aplicação, quando o usuário desejar sair do laço e retornar ao sistema operacional. No entanto, antes de finalizar a aplicação, todos os recursos alocados devem ser liberados para a limpeza do sistema.

Acrescentando a necessidade de controle de entrada e saída da aplicação por parte do usuário, a figura 3.3 apresenta um diagrama de transição de estados para um outro laço de mais alto nível que engloba a estrutura apresentada na figura 3.2. Neste diagrama, é atribuído ao usuário o controle do estado da aplicação.

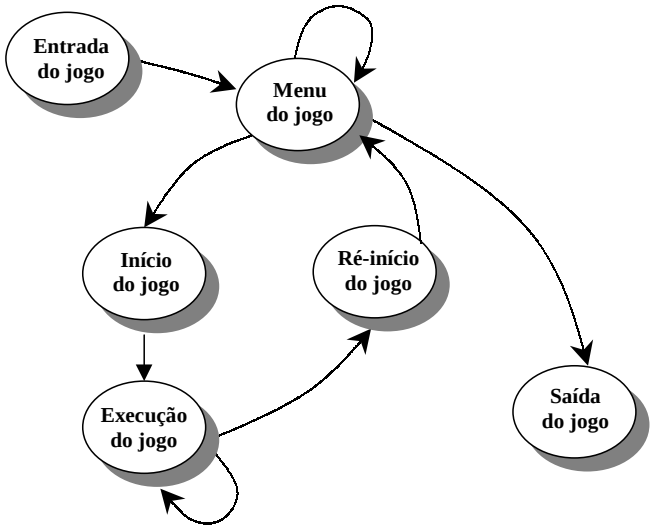


Figura 3.3: Diagrama de transição de estados para o controle do estado da aplicação

3.2 A composição de um motor

Motores são utilizados como peça chave na construção de jogos de computador, pois são responsáveis pelas características de mais baixo nível destas aplicações, concentrando todos os processamentos básicos necessários para o controle das mídias envolvidas. Desta forma, o desenvolvimento de jogos torna-se mais fácil, deixando os desenvolvedores voltados para os objetivos característicos da própria aplicação - lógica, arte, inteligência, entre outros - e não mais para os processamentos de baixo nível.

Do ponto de vista de programadores, um motor é uma ferramenta que permite novos jogos serem construídos a partir das interfaces e funcionalidades que ele provê, cabendo aos programadores implementar as peculiaridades de um jogo específico. Do ponto de vista de requisitos, espera-se que um motor assuma uma série de tarefas e disponibilize uma série de módulos de fácil utilização e extensão para cada novo jogo [65].

A arquitetura de um motor corresponde exatamente à definição dos seus módulos e como é realizada a interação entre eles. Tais módulos são responsáveis por identificar e gerenciar todas as mídias que são utilizadas e eventos que ocorrem durante um jogo, desde eventos gerados pelos usuários e personagens autônomos até aqueles gerados pelo sistema operacional e hardwares.

Um motor bem projetado deve apresentar dois módulos ou "fachadas" de interfaceamento: um módulo de abstração de hardware e um módulo de abstração de jogos [44]. O primeiro tem o objetivo de isolar todos os componentes que necessitam

de acesso à plataforma utilizada pelo usuário (hardwares e sistema operacional). O segundo, deve disponibilizar em alto nível as funcionalidades do motor para serem utilizadas na implementação de projetos de jogos. Estes módulos podem ser inseridos entre três camadas: baixo nível, médio nível e alto nível [44] (ver figura 3.4).

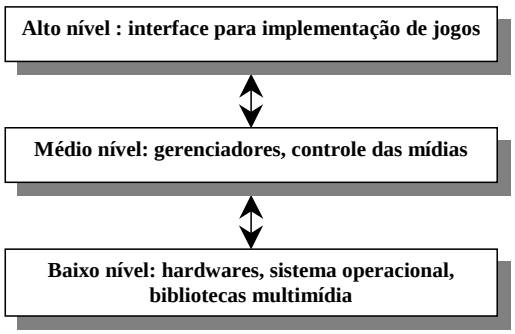


Figura 3.4: Camadas de um motor

A camada de baixo nível é extremamente importante. Os seus principais benefícios são o desenvolvimento de um ambiente homogêneo e de componentes reusáveis que permitam fácil configuração de hardware, tratando todas as complexidades e problemáticas a eles associadas. Esta camada é a que deve utilizar as bibliotecas multimídia para aceleração por hardware (apresentadas no capítulo 4) e tratar as peculiaridades dos diversos sistemas operacionais, abstraindo da camada de médio nível as complexidades intrínsecas às diversas plataformas. Desta forma, disponibilizando um único conjunto de funcionalidades, e proporcionando a independência de plataforma devido a possibilidade de serem construídas interfaces específicas para cada uma delas.

A camada de médio nível deve ser responsável pelo controle das mídias, objetos e cenários do jogo, garantindo facilidade para implementação do processamento lógico. Além disso, deve ser responsável por todo o controle de eventos, tratando as ações dos jogadores e entidades inteligentes no decorrer das partidas dos jogos.

A camada de alto nível é a responsável por oferecer aos desenvolvedores uma interface de fácil utilização do motor. Ela tem o objetivo de abstrair as tarefas que podem ser resolvidas automaticamente, e disponibilizar uma estrutura padronizada para a implementação da lógica do jogo.

Dentre estas camadas, existem diversos módulos responsáveis por tarefas de gerenciamento de um motor como, o módulo gráfico, o de sonorização, o de inteligência de personagens, entre outros. Dentre os quais, o de maior complexidade para ser de-

senvolvido e que pode alterar radicalmente a estruturação de um motor é o módulo gráfico [16]. Em função disto, será discutido na próxima seção as formas de representação que podem ser implementadas por este módulo.

3.3 Formas de representação gráfica para motores

Nesta seção são apresentadas as diferentes formas de como o módulo gráfico de motores para jogos pode ser projetado: baseado em representação gráfica 2D ou baseado em representação gráfica 3D.

3.3.1 Representação gráfica 2D

Um motor que trabalha com representação gráfica 2D, se caracteriza essencialmente por utilizar mapas de bits, através do uso preferencial de rasterização de *sprites* e de técnicas relacionadas, como *double-buffering* e *scrolling* [33]. Um *sprite* é composto de uma seqüência de imagens, usualmente retangulares, que definem a movimentação de uma determinada entidade [33]. Em função desta movimentação, as imagens são substituídas para apresentação em vídeo, e normalmente, contém áreas que devem ficar transparentes durante a sua exibição para permitir que não só elementos retangulares sejam mostrados através de um *sprite*.

Sprites são categorizados por diversos tipos: *sprites* que agem como repositório para outros (podem ser animados pela troca de imagens), *sprites* estáticos (contém uma única imagem) e *sprites* de texto.

Em cenários 2D, diversas técnicas de gerenciamento de *sprites* podem ser utilizadas, a mais elegante funciona baseada no controle das partes que sofrem modificação em um cenário, alterando apenas estas, resultando numa melhor taxa de apresentação de quadros. A detecção de colisão dos objetos nestes cenários é feita quando da colisão de dois *sprites*, retornando uma resposta a estas ações.

Cenários 2D podem ser extremamente grandes, e usualmente são compostos por *tiles*, células que representam uma textura para a composição de um cenário 2D, ou seja, representam uma simples localização no mapa. Na intenção de obter cenários bem projetados, boas estruturas de arquivos devem ser desenvolvidas para compor o mapa destes cenários. Um mapa deve permitir mudanças em um cenário sem causar modificações na estruturação do seu modelo.

O uso de *sprites* sobrepostos a um cenário de fundo, permite implementar rápidas e interessantes animações de entidades. Ao *sprite* se associa altura, largura, posição, estado de visibilidade, prioridade entre outros *sprites*, transformações de escala,

translação e rotação, detecção de colisão, etc [33]. *Sprites* podem ser 2D, que realizam movimentação no eixos x e y em cenários retangulares ou isométricos, ou 3D, que realizam movimentação nos eixos x, y e z em cenários isométricos ou 3D.

3.3.1.1 Cenário retangular ou puramente 2D

Cenários puramente 2D usualmente são baseados em *tiles* retangulares ou em porções do mapa que correspondem a retângulos do tamanho de uma tela cheia, realizando a rasterização destes retângulos. Para projetar as entidades do cenário, os *sprites* devem ser desenhados na ordem de cima para baixo da tela, sendo esta a ordem correta de apresentação [41]. A detecção de colisão pode ser realizada por sistemas de coordenadas também baseados em retângulos. A figura 3.5 ilustra exemplos deste tipo de cenário.

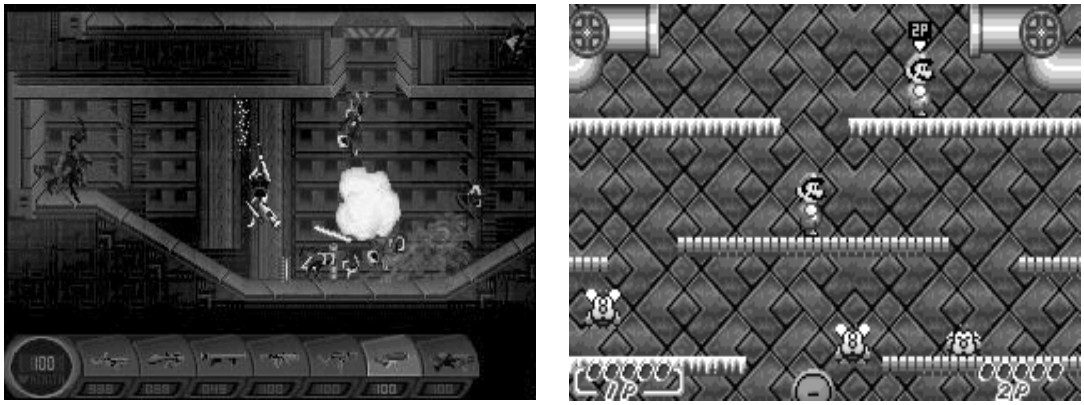


Figura 3.5: Exemplos de cenários retangulares - Abuse (esquerda) e Super Mario Brothers (direita)

3.3.1.2 Cenário isométrico ou 2D com perspectiva (2 1/2D)

Cenários isométricos apresentam um mesmo conjunto de características de cenários retangulares, acrescentados de outras características bastantes interessantes como, controle de relevo e espelhamento. A grande diferença destes cenários é que eles utilizam uma visão com ângulo em perspectiva, e ordenam os objetos de acordo com a visão da câmera, para uma correta rasterização. Este tipo de cenário é mais utilizado para jogos do tipo RPG e jogos de guerra. A sua alta popularidade vem do fato de que a sua ampla visão permite uma boa jogabilidade. Apresenta uma aparência muito melhor que em relação ao puramente 2D, e mantém uma boa aproximação do 3D [41].

Nos cenários isométricos, os *tiles* se apresentam em forma de losangos. Devido a utilização desta forma, eles dão a ilusão de profundidade e relevo mesmo usando gráficos 2D. Cenários 2 1/2D fazem uso extensivo de técnicas computacionalmente simples para simular uma cena 3D sem renderizações custosas. Esta técnica se divide em duas classes: *sprites* planares e geo-*sprites* [33]. *Sprites* planares fazem uso de texturas aplicadas a objetos 3D simples (planos), enquanto geo-*sprites* são descritos em termos de polígonos, mas ao invés de serem renderizados, eles são preenchidos com texturas predefinidas e armazenadas em memória.

O mapa de um cenário é provavelmente a mais importante estrutura de dados dos jogos isométricos. Ele serve como repositório central de quase todos os dados do jogo e é a fonte primária de informação usada para a lógica. Os *tiles* apresentam informações específicas ao jogo, e as diversas entidades, como exemplo os personagens inteligentes, dependem extensivamente do mapa.

Existem três abordagens de desenvolvimento para cenários isométricos: baseados em *tiles*, baseados em tela cheia e baseados em 3D [41].

Na abordagem baseada em *tiles*, deve-se ser decidido o ângulo de visão do cenário e então projetar toda a arte gráfica baseada neste ângulo. Usualmente, projeta-se tudo em tiles retangulares, e a rasterização é executada baseada em losangos. Para projetar o cenário, os objetos não podem ser desenhados numa ordem qualquer. No caso da arte ter sido criada numa visão com perspectiva para o sentido nordeste ou noroeste no mapa, a tela deve ser preenchida na ordem de cima para baixo, sendo esta a ordem correta de distância dos objetos. A detecção de colisão pode ser realizada por sistemas de coordenadas complexas baseados em losangos, hexágonos, ou octágonos.

Na abordagem baseada em tela cheia, o cenário é desenhado por um modelador 3D, ou por alguma ferramenta que permita projetar a informação necessária para cada tela. A tela pode ser de tamanho configurável e utiliza uma estrutura de dados secundária para conter informações sobre detecção de colisão. Usando esta técnica, pode-se simplesmente desenhar a tela de fundo toda de uma vez.

Uma grande gama de jogos tem sido desenvolvida baseados em cenários 2 1/2D, como exemplos *Age of Empires II* e *Outlive* ilustrados na figura 3.6.

Na abordagem baseada em 3D, as características são totalmente diferentes das anteriores, pois utiliza-se um modelo totalmente 3D (apresentado na próxima seção), aumentando bastante a sua complexidade em relação as abordagens 2D. Neste caso, apenas uma simples adaptação deve ser feita ao modelo 3D, ou seja, a câmera deve ser fixada numa visão isométrica. Como a visão estará sempre numa mesma perspec-

Figura 3.6: Exemplos de cenários isométricos - Age of Empires II (esquerda) e Outlive (direita)

tiva, certas otimizações e suposições podem ser feitas sobre a ordem de renderização e a complexidade das cenas. Esta é a abordagem que muitos jogos isométricos do *Sony Playstation I e II* utilizam (ver figura 3.7).

Figura 3.7: Exemplos de cenários isométricos construídos com renderização 3D - Age of Mythology (esquerda) e Warcraft 3 (direita)

3.3.2 Representação gráfica 3D

Um motor que trabalha com representação gráfica 3D, se caracteriza essencialmente por utilizar cenários constituídos de objetos representados por polígonos ou malhas de triângulo. Ele é passível de renderizar imagens realistas complexas em tempo real num ambiente tridimensional, o que é extremamente custoso em termos computacionais [33].

Objetos 3D são nada mais que um conjunto de equações matemáticas representadas por polígonos que indicam a forma, posicionamento e tamanho relativo de tal objeto. A estes objetos são sobrepostas texturas, apresentando uma sensação muito maior de realismo, e através do uso de transformações geométricas e interpolação se torna possível a implementação de animações. Além disso, cenários tridimensionais comportam o uso de *sprites* 3D através do uso da técnica de *billboarding*, responsável por executar rotações nos *sprites* de acordo com a visão da câmera, possibilitando sempre sua aparição frontal.

Para a construção de cenários 3D, um bom conjunto de características devem ser oferecidas pelo motor: superfícies curvas, iluminação dinâmica, neblina volumétrica, espelhamento e portais, *skyboxes*, preenchimento de vértices, sistemas de partículas, modelos de malhas estáticas e dinâmicas, detecção de colisão, etc [16].

Em cenários tridimensionais, diversas técnicas de gerenciamento de objetos podem ser utilizadas, uma das mais conhecidas e eficientes são as árvores BSP[1], responsáveis por precalcular a ordem de distância dos polígonos no mundo. Desta forma, facilita a renderização, e provê muitos benefícios em outras áreas como detecção de colisão e *culling* [70].

Apesar das CPUs serem capazes de criar imagens tridimensionais, trabalhando sozinhas elas não geram imagens de boa qualidade à alta velocidade, nem aplicam os principais efeitos 3D, devido ao grande número excessivo de cálculos necessários. Portanto, um motor 3D deve oferecer suporte a utilização de bibliotecas gráficas capazes de utilizar as características inerentes aos hardwares que oferecem aceleração 3D. Desta forma, o processamento do seu pipeline gráfico (transformação, iluminação e rasterização) se torna muito mais eficiente. Em favor disso, existem ainda técnicas para a renderização em conjunto de uma malha de triângulos (*fans*, *strips*), que aproveitam ainda mais as características de aceleração dos hardwares [16].

Jogos 3D já são a maioria nas produções da indústria de jogos atual (ver exemplos na figura 3.8), mas a construção de um motor com representação gráfica 3D é a mais difícil entre todas as formas de representação, pois a implementação dos algoritmos envolvidos é bastante complexa. A principal razão é a alta velocidade e o realismo visual requisitados pelas aplicações, os quais são basicamente limitados pelos recursos oferecidos pelos hardwares, exigindo do motor um alto nível de otimização.

[1] Binary Space Partitioning

Figura 3.8: Exemplos de cenários 3D - Rogue Squadron (esquerda) e Unreal II (direita)

3.4 Motores de código aberto

Como mencionado no capítulo 2, estudamos algumas ferramentas para viabilizar o desenvolvimento do jogo *Canyon*, dentre elas temos: o *Genesis3D* [24], o *Crystal Space* [12] e o *Golgotha* [26] que são apresentados nesta seção explicitando suas características, além de uma avaliação sucinta e críticas sobre os problemas encontrados.

3.4.1 Genesis3D

Genesis3D é um pacote de desenvolvimento que permite construir aplicações gráficas 3D. Ele é composto de bibliotecas, ferramentas para a construção de cenários, exemplos de código para demonstração das suas capacidades básicas, e uma documentação para ensino da sua utilização. O *Genesis3D* adota os princípios de software livre, disponibilizando ao público o código fonte das suas bibliotecas e ferramentas, com exceção de alguns módulos que são disponibilizados apenas em bibliotecas de ligação dinâmica precompiladas (DLLs).

O *Genesis3D* foi projetado primariamente para renderizar cenários de ambientes fechados com polígonos que apresentem quantidade moderada de vértices. Ele também pode ser usado para construir cenários abertos não extensos, mas com diversas restrições, e várias precauções devem ser tomadas para a apresentação destes cenários de forma correta. *Genesis3D* não é projetado para renderizar cenários compostos de terrenos abertos não limitados, e nem para executar *culling*[2] em ve-

[2]Processo pelo qual é determinado que um objeto não esta visível, logo não deve ser renderizado

locidade satisfatória para geometrias arbitrárias abertas. Não deve ser esperado uma alta taxa de quadros na construção de grandes cenários com alto número de polígonos e sem inibidor de visibilidade (paredes). A figura 3.9 apresenta exemplos de funcionamento deste motor.

Figura 3.9: Exemplos de execução do motor Genesis3D

Este motor disponibiliza subsistemas projetados para serem auto-suficientes, ou utilizarem o mínimo de suporte de outros. A implementação destes subsistemas se dá por bibliotecas de ligação dinâmica que devem ser utilizadas em sistema operacional *Windows*.

As principais características do *Genesis3D* são: mapeamento e animação de texturas; uso de texturas translúcidas; *morphing* de texturas; sombreamento dinâmico; iluminação em múltiplas cores; neblina; *morphing* de vértices; distorção de superfícies aquosas em tempo real; espelhamento; suporte a detecção de colisão; teste de visibilidade; suporte a simulação física de corpos rígidos integrados; hierarquia e disjunção de personagens poligonais *soft-skin*; suporte a arquivos de animação do 3D Studio Max; suporte a LOD[3]; uso de portais para delimitação de áreas permitindo renderização seletiva da geometria do ambiente; uso de renderização baseada em *Direct3D* (*DirectX* versão 6.0), *Glide* e rasterização por software; *scripting*; editor de cenários; posicionamento e atenuação de sons 3D; suporte a rede; e código fonte em C/C++.

3.4.2 Crystal Space

Crystal Space é um software livre que consiste de um pacote de componentes e bibliotecas que pode ser utilizado para a construção de jogos de computador 2D e

[3]Level of Detail

3D. Ele disponibiliza ferramentas para a construção de cenários, exemplos de código para demonstração das suas capacidades básicas, e uma boa documentação da sua estruturação e utilização. Uma das principais características do *Crystal Space* é que seus componentes e bibliotecas são independentes uns dos outros o quanto possível. Todos os componentes são projetados para serem funcionais com o mínimo de outros componentes em conjunto.

Crystal Space não é uma biblioteca monolítica simples, ele consiste de diversas bibliotecas e módulos de *plug-ins*[4]. Um módulo de *plug-in* é similar a uma biblioteca, porém tem a vantagem de poder ser acessado de forma comum pelas aplicações. Eles têm muitas interfaces bem definidas, e podem prover esta interfaces uniformes através de bibliotecas de ligação dinâmicas.

Da mesma maneira que o *Genesis3D*, o *Crystal Space* foi projetado inicialmente para renderizar cenários de ambientes fechados com polígonos que apresentem quantidade moderada de vértices, e na versão atual ainda funciona desta maneira. Mas, nas definições das etapas de projeto de versões futuras há a previsão de se poder construir cenários abertos, e então renderizar terrenos externos não limitados. A figura 3.10 apresenta exemplos de funcionamento deste motor.

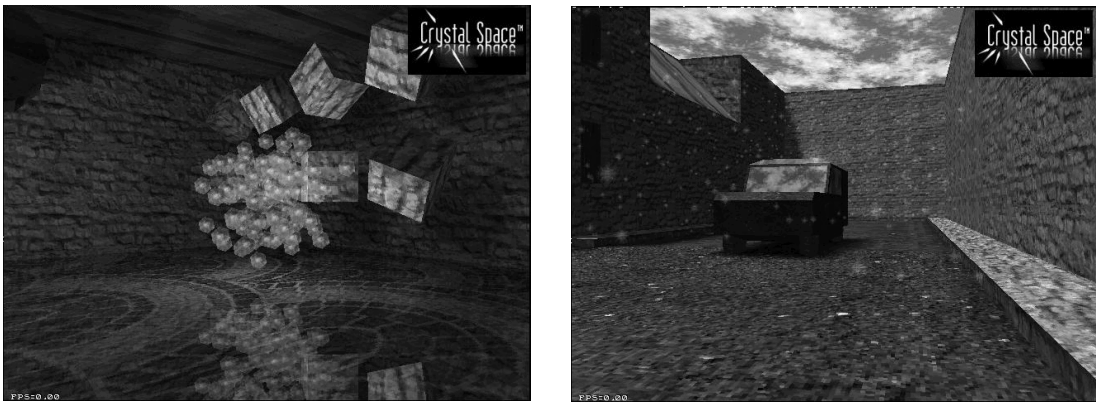


Figura 3.10: Exemplos de execução do motor Crystal Space

As principais características do *Crystal Space* são: mapeamento de texturas com perspectiva de sombreamento; *mipmapping*; precisão de *sub-pixel* e *sub-texel*; uso de portais para delimitação de áreas permitindo renderização seletiva da geometria do ambiente; iluminação estática com sombreamento precomputado; iluminação dinâmica colorida com sombreamento transparente e semitransparente; espelhamento; suporte a árvores BSP; suporte a superfícies curvas; suporte a correção de

[4]Módulos que realizam ligação entre duas entidades ou aplicações que não se comunicam

profundidade e neblina volumétrica colorida; sistema hierárquico de detecção de colisão; teste de visibilidade baseado em *coverage buffer*; suporte a sons 3D (*Windows* e *Linux*); movimentação de objetos e *scripts* para controle da movimentação; malha de triângulos com preenchimento *gouraud*; suporte a LOD; suporte a *Direct3D* e *DirectDraw* para Windows, *OpenGL* para Linux, BeOS, MacOS/9, OS/2, e Windows, e *Glide* para Linux e Windows; leitor para modelos do Quake II (MD2) e 3D Studio (3DS); suporte a rede; e código fonte em C++.

3.4.3 Golgotha

O *Golgotha* foi projetado para ser um motor comercial, desenvolvido pela *Crack dot Com*[5] com o intuito de construir um jogo 3D também denominado *Golgotha*. Mas após dois anos de desenvolvimento, devido ter passado o prazo de lançamento do jogo e não mais existir verbas para a continuidade, a *Crack dot Com* decidiu adotar os princípios de software livre e liberou o código fonte do *Golgotha* para o público. O motor *Golgotha* é composto por um conjunto de bibliotecas, ferramentas para a construção de cenários, e um exemplo de código do demo do jogo *Golgotha*.

O motor *Golgotha* executa a renderização de cenários de ambientes abertos (terrenos limitados ou não limitados) compostos por objetos (malha de polígonos) em seu formato proprietário. Os seus cenários podem ser constituídos por terrenos em diversos níveis de relevo, de modo que lagos, montanhas, e outras formações podem ser inseridas ao ambiente. Os objetos móveis possuem capacidades de modelagem física avançadas. O *Golgotha* utiliza renderização em profundidade e apresenta como resultado um desempenho excelente. Mas, em contraposição, não possui documentação, e sua estruturação visa um jogo específico, causando uma grande dificuldade para a reutilização das suas bibliotecas, além de apresentar uma padronização de código bastante confusa. A figura 3.11 apresenta exemplos de funcionamento deste motor.

As principais características do *Golgotha* são: mapeamento de texturas com perspectiva de sombreamento; filtragem de texturas; renderização de paisagens; uso de uma biblioteca de renderização portável com suporte a *Glide*, *Direct3D*, *OpenGL* e rasterização por software; suporte a sons 3D (*Windows* e *Linux*); uso de *plug-in* para importação/exportação de objetos 3D Studio Max; editor de cenários; uso de código portável para rede com suporte a UDP e TCP; interpretador LISP com coletor de lixo *threaded*; sistema de janelas e interface de usuário portável; sistema de gerenciamento de texturas JPEG, e apontadores de texturas 1024x1024 pelo uso de *cache*; LOD em terrenos; suporte a trilha musical em formato WAV e MP3;

[5] http://www.crack.com

Figura 3.11: Exemplos de execução do motor Golgotha

histogramas e quantização de imagens; otimizações em linguagem *Assembly* para rasterização por software em *Pentium* e *K6-3D*; e código fonte em C/C++ que pode ser executado em Windows (DirectX, Glide, software), Linux (Glide, OpenGL), e Mac (RAVE, Glide).

3.4.4 Avaliação e críticas

O nosso principal objetivo nesta avaliação se dá para o grau de qualidade apresentado pelos motores estudados, com o intuito de verificar se os princípios de desenvolvimento de software foram utilizados e quais os resultados obtidos. Para isto, a tabela 3.1 demonstra a comparação entre as diversas características avaliadas nos motores.

Tabela 3.1: Características relevantes avaliadas nos motores

	Genesis3D	**Crystal Space**	**Golgotha**
Grau de modularidade	médio	bom	fraco
Grau de reusabilidade	médio	bom	fraco
Extensibilidade	possível	possível	difícil
Portabilidade	não	sim	sim
Documentação do projeto	não tem	fraca	não tem
Documentação do usuário	fraca	média	não tem
Documentação do código	boa	excelente	muito fraca

	Genesis3D	**Crystal Space**	**Golgotha**
Linguagem de programação	C/C++	C++	C/C++
Suporte	bom	ótimo (possui lista de discussão bastante acessada)	não tem
Padronização do código	boa	boa	confusa
Continuidade	em andamento	em andamento	parado
Objetivo do projeto	código aberto	código aberto	comercial
Público que utiliza	amadores	amadores	não conhecido
Tipo de cenários	ambiente fechado	ambiente fechado	ambiente aberto
Grau de desempenho	médio	bom	excelente (muito rápido)
Velocidade de desenvolvimento	média	média	parado
Situação atual	versão beta	versão beta	versão demo não finalizada

O que se pode constatar pela avaliação apresentada na tabela acima, é que na comparação entre os motores, o *Crystal Space* é o mais bem desenvolvido, pois utiliza os princípios de projetos orientados a objetos para a construção de softwares modulares, reusáveis e portáveis, e apresenta boa documentação de código. Mas, apresenta desempenho razoável (necessita de otimização), não disponibiliza uma documentação detalhada do seu projeto, e o principal para o projeto do *Canyon*, não funciona para ambientes abertos.

O *Genesis3D* é o segundo melhor em termos de engenharia de software, mas é escrito particularmente para *Windows*, não apresenta boa documentação, não apresenta bom desempenho, e funciona apenas em ambientes fechados.

O *Golgotha* é fraco quanto a engenharia de software, pois não apresenta documentação nenhuma, o código é de difícil entendimento, não reusável, não possui suporte, e atualmente é um projeto morto. Mas em compensação, apresenta um desempenho extremamente superior aos outros motores e é o único projetado para funcionar em ambientes abertos.

Neste contexto, nenhum dos motores se apresentou satisfatório para o desenvolvimento do jogo *Canyon*, devido aos cenários deste jogo serem compostos de ambientes abertos onde aeronaves são os principais personagens, e também devido aos princípios de engenharia de software serem bastante importantes em tal desenvolvimento.

3.5 Conclusão do capítulo

Jogos de computador são aplicações extremamente complexas que consistem de uma série de fatores muito importantes para a sua boa aceitação: um bom enredo que resulte numa boa jogabilidade (sem isto, o restante é irrelevante); a construção de um conteúdo coerente e de boa qualidade (ambiente, objetos, texturas, sons, músicas); a construção de um motor para gerenciar os processamentos de baixo nível da aplicação em relação as mídias envolvidas; e a construção da aplicação do jogo para utilizar o conteúdo quando necessário e integrar a inteligência artificial para dar suporte a jogabilidade.

Um motor é um elemento essencial para os jogos, devido a necessidade de disponibilizar um ambiente que reduz enormemente a quantidade de código a ser escrito, e apresenta uma maior acessibilidade a programadores de linguagens de alto nível. Consequentemente, possibilitam uma grande economia de esforço, tempo, e custo em tal desenvolvimento. Logo, uma vez projetado, um motor pode ser reutilizado não só numa enorme variedade de jogos, mas também em outras aplicações correlacionadas. Mas para isso, um motor deve ser construído seguindo os padrões de engenharia de software que resultam em aplicações de boa qualidade, e devem apresentar a flexibilidade de serem personalizados para cada projeto específico a fim de atender necessidades particulares. Desta forma, o desenvolvimento de jogos torna-se mais fácil, deixando os desenvolvedores voltados para os objetivos característicos da própria aplicação, e não mais para os processamentos de baixo nível.

Construir um bom motor para jogos certamente requer muito conhecimento sobre computação gráfica, matemática, estruturas de dados, engenharia de software, programação de alto e baixo nível, processos concorrentes, entre outros. Portanto, é extremamente importante projetar um sistema modular, o qual seus módulos sejam integrados de maneira eficiente e possam ser reutilizados por outros projetos. Um motor para jogos é um grande *framework* que os princípios de projetos orientados a objetos devem ser fortemente aplicados na sua construção.

Capítulo 4

Infra-estrutura para Jogos de Computador

No capítulo anterior foi visto que o desenvolvimento de jogos de computador é uma tarefa complexa que necessita do apoio de um motor para facilitar a sua construção. Embora a utilização de motores para jogos seja essencial, existem outras ferramentas que dão suporte a esse desenvolvimento e devem ser levadas em consideração no projeto de qualquer *framework* para jogos. Neste capítulo, são apresentadas as principais e mais utilizadas destas ferramentas com a intenção de definir a infra-estrutura necessária para o desenvolvimento do *FORGE V8*. Estas ferramentas constam de placas aceleradoras gráficas, placas de som, bibliotecas multimídia, e outras ferramentas de suporte como, compiladores, modeladores gráficos, entre outros.

4.1 Placas aceleradoras gráficas

Nesta última década, a computação gráfica (especialmente gráfica 3D, e gráfica 3D interativa em particular) tem sido uma área bastante explorada da Ciência da Computação. Isto se deve em parte, a demanda por jogos de computador mais realistas e que suportam interação em tempo real. Um dos resultados dessa exploração tem sido o advento das placas gráficas aceleradoras que diminuem enormemente o processamento realizado pela CPU quando renderizando gráficos 3D.

As placas gráficas possuem processadores dedicados, cuja função é, unicamente, processar imagens, o que podem fazer com incrível rapidez, deixando a CPU livre para executar outras tarefas. Elas variam enormemente em complexidade, qualidade e preço, e se dividem em duas classes principais dentre as utilizadas atualmente [77]:

- Placas 3D: aceleram apenas o módulo de rasterização[1], correspondente ao último estágio do pipeline gráfico (transformação, iluminação e rasterização). O desempenho é acrescido enormemente por causa do muito trabalho que é requerido pela CPU para executar a renderização final de polígonos. Com esta carga de trabalho removida do processador, um jogo pode executar outras tarefas em paralelo com a renderização, como inteligência artificial ou física mais realista.

- Placas 3D TnL[2]: provêem um PC com capacidades comparáveis às estações de trabalho de altíssimo custo dedicadas à renderização gráfica que foram disponibilizadas no passado. Estas placas correspondem a mais nova geração, e aceleram todo o pipeline gráfico . No entanto, elas somente começaram a aparecer no mercado este ano, e irão significar a abertura de fronteiras para o desenvolvimento de projetos mais complexos devido a enorme velocidade apresentada.

Devido a essa evolução das placas gráficas, principalmente para o uso em jogos, as CPUs e placas mãe também evoluíram no suporte à operações típicas para os jogos tais como, mais rapidez no cálculo da divisão e da inversa da raiz quadrada (normalização de vetores), inserção de paralelismo para apoio à transformação de pontos e cálculo de produtos (*cross* e *dot*) [16], e a disponibilização de um barramento exclusivo para aceleração do processamento gráfico (AGP).

4.2 Placas de som

As placas de som, diferentemente das placas gráficas, não influenciam no desempenho dos jogos, mas sim no nível de qualidade apresentado para a reprodução de áudio [25]. Este nível de qualidade depende das tabelas compostas por notas musicais em formato digital que são disponibilizadas nestes *hardwares*. Estas tabelas variam de acordo com as gerações das placas de som (16, 32, 64 e 128 bits), da mais baixa para a mais alta qualidade respectivamente, e apresentam uma relação entre notas sintetizadas e gravações digitais para a reprodução das mídias.

Existem dois tipos de sons que estes hardwares podem reproduzir [41]: digital e sintetizado. Sons digitais são basicamente gravações de sons, enquanto sons

[1]Processo do sistema de computação gráfica que é responsável por calcular e desenhar os pixels na tela correspondentes a cada polígono fornecido pelo processo de renderização

[2]Transformação e iluminação

sintetizados são reproduções programadas de sons baseados em algoritmos e geradores de tons em hardware. Sons digitais (WAVs) são mais utilizados para efeitos e conversação, ocupam muito espaço em memória devido a serem gravações digitais, e a qualidade de reprodução não é influenciada pela placa de som, pois possuem suas próprias tabelas de notas musicais. Sons sintetizados (MIDIs - *Musical Instrument Digital Interface*) são utilizados para músicas, ocupam pouquíssimo espaço em memória devido a serem apenas representações de seqüências de notas musicais para instrumentos predefinidos, e a qualidade depende totalmente das placas de som, pois eles utilizam as tabelas de notas musicais em formato digital destas placas para a reprodução.

As placas de 64 e 128 bits apresentam ainda a possibilidade de reprodução de sons em 3D, gerando sons que parecem vir de todas as direções, mesmo usando caixas acústicas comuns. Este efeito é extremamente interessante nos jogos, pois dá uma sensação muito maior de realidade.

4.3 Bibliotecas multimídia

Um outro importantíssimo resultado da crescente demanda dos jogos é o desenvolvimento de bibliotecas multimídia destinadas a aumentar a portabilidade dos jogos sob os diversos hardwares e plataformas operacionais.

As bibliotecas multimídia eliminam a necessidade de escrever *drivers* distintos para cada plataforma ou hardware em que uma aplicação necessite executar. Esta padronização provê uma interface natural que permite os programadores escreverem código a mais alto nível. Além disso, as bibliotecas possuem a flexibilidade para acomodar extensões de novos subsistemas que apresentem novas operações, de maneira que estas possam ser providas sem causar problemas a interface original. Assim, promove-se a produção de jogos autoconfiguráveis, reduzindo a necessidade de escrever código de muito baixo nível, e desviando a responsabilidade do suporte a hardware para os próprios desenvolvedores de hardwares.

Dentre as várias bibliotecas existentes, as principais e mais popularmente conhecidas que provêem acesso direto a hardware e englobam uma grande quantidade de funções complexas implementadas, permitindo a construção de aplicações com qualidade e desempenho extremamente superiores são *OpenGL* [52] e *DirectX* [13].

4.3.1 OpenGL

OpenGL (Open Graphics Library) é uma biblioteca de baixo nível que provê um completo pipeline gráfico para a renderização de polígonos através de interfaces [52]. Ela foi originalmente desenvolvida pela *Silicon Graphics* e usada em aplicações de visualização de alto nível, como aplicações militares de simulação de vôo. Por esta razão, *OpenGL* é de fato um padrão para visualização de alto nível em estações de trabalho. Porém, também é bem suportada em PCs graças a grande importância de jogos como *Quake*, além dos muitos jogos populares da atualidade que a utilizam.

Ao contrário de ter que se comunicar diretamente com a placa de vídeo, a aplicação deve atribuir comandos através da biblioteca *OpenGL*, e então os processos de transformação, iluminação e rasterização são realizados. Neste sentido, *OpenGL* é uma grande caixa preta a qual pode se passar dados e comandos de desenho, e obter os resultados apresentados na tela. A execução do pipeline gráfico dessa biblioteca é independente do sistema de janelas do sistema operacional, o que permite ser incorporada em um sistema de janelas qualquer (*X Window/ Windows*). *OpenGL* pode eficientemente ser utilizada em diferentes hardwares gráficos, do básico (placas 2D) até os mais sofisticados subsistemas gráficos (placas 3D TnL). É uma excelente escolha para implementar aplicações gráficas 2D e 3D [77].

Uma grande vantagem de utilizar esta biblioteca é o seu aspecto de portabilidade, de modo que uma aplicação bem escrita em *OpenGL* pode ser recompilada para executar em qualquer sistema, do mesmo modo que um programa bem escrito em *ANSI C++*. Implementações de *OpenGL* existem para *Microsoft Windows*, *Macintosh*, *Linux*, *Silicon Graphics* e outras estações de trabalho, e novas versões estão sendo atualmente desenvolvidas para os consoles *X-Box* e *PlayStation2*.

OpenGL é dividida em três camadas: uma camada de aplicação, uma camada de software que se situa entre a aplicação e todo o sistema gráfico (API *OpenGL*), e o driver (*ICD*[3] *OpenGL*) que se comunica com o hardware gráfico, como mostra a figura 4.1.

A camada de *aplicação* é responsável por gerenciar os objetos e orientações no mundo, e a aparência destes objetos como malhas de polígonos. Ela passa os polígonos que definem estes objetos para *OpenGL* renderizar. Esta é a única camada que é utilizada pelos desenvolvedores, responsável por inserir todas as interfaces de *OpenGL* necessárias a aplicação.

A *API OpenGL* é um conjunto padrão de funções, que atribui às aplicações independência do subsistema gráfico em que seja executado. A API executa al-

[3]Independent Client Driver

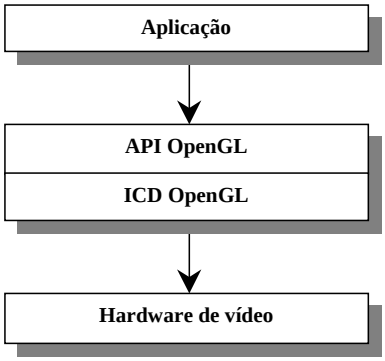


Figura 4.1: Camadas de hardware e software em uma aplicação *OpenGL*

guns processos nestes comandos, e então emitem comandos para o driver executar o desenho de polígonos. Independente de qual sistema esteja trabalhando, a API permanece idêntica.

O *IDC OpenGL* é o driver responsável por coletar a entrada da API e convertê-la em um formato adequado para o hardware gráfico. Dependendo das capacidades do hardware, o driver pode ser responsável por implementar parte ou todo o pipeline gráfico em software. O driver é provido pelo fabricante do hardware.

A camada da API representa a interface para *OpenGL*, e define a funcionalidade que deve estar presente nas camadas de mais baixo nível. Abaixo dela, o driver e a placa de vídeo implementam todas as funcionalidades. Isto significa que a maior parte do trabalho feito pelo driver varia de acordo com o sistema. Para o uso de uma placa 2D básica, todo o pipeline gráfico é implementado em software pelo driver. Ao contrário, em uma moderna placa de vídeo 3D TnL como uma *geForce*, o driver realiza pouco trabalho, e quase tudo de *OpenGL* é implementado em hardware.

OpenGL trata "apenas" a camada gráfica de uma aplicação, mas atualmente já está sendo testada (versão beta) uma nova biblioteca multiplataforma chamada *OpenAL* (Open Audio Library) [51] que tem o propósito de funcionar em conjunto com *OpenGL* para tratar a camada de sonorização das aplicações.

4.3.2 DirectX

DirectX é um conjunto de bibliotecas de baixo nível que trabalham juntas para formar um sistema integrado para a produção de jogos e aplicações multimídia que necessitam de alto desempenho e alta qualidade em ambiente *Windows* [13]. Estas aplicações podem incorporar gráficos 2D e 3D, sons stereo e 3D, músicas, vídeos, dispositivos de entrada, e suporte a rede para aplicações multiusuário. A sua versão

atual é a 8.0, e sua estruturação é dada pelos seguintes componentes: *DirectX Graphics*, *DirectX Audio*, *DirectInput*, *DirectShow* e *DirectPlay* (ver figura 4.2).

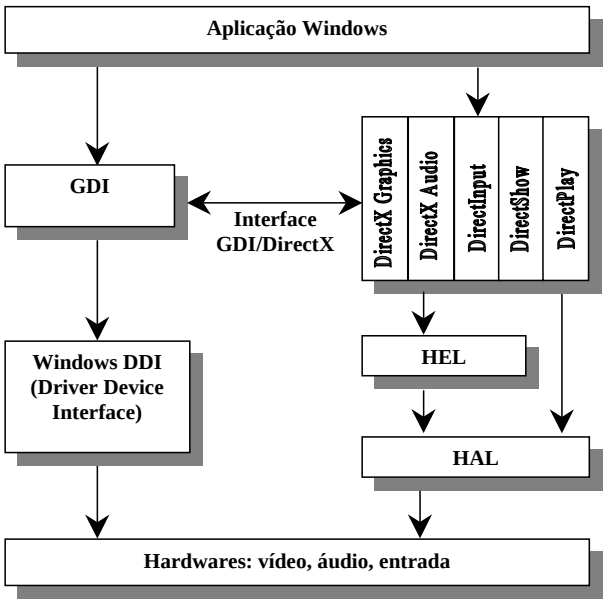


Figura 4.2: Arquitetura *DirectX* e seu relacionamento para o Windows

DirectX apresenta uma divisão em camadas que se aproxima a de *OpenGL*, uma camada de aplicação, uma camada da API *DirectX* (componentes) e uma camada de abstração de hardware (HAL[4]) que corresponde ao driver do fabricante do hardware para comunicação com este. Entre este último e a API existe a camada de emulação de hardware por software (HEL[5]), para casos onde uma placa aceleradora não esteja disponível. As diferenças que existem em relação a *OpenGL* são o acesso que *DirectX* possibilita aos drivers gráficos nativos do *Windows* através da GDI[6] e a quantidade maior de componentes em *DirectX* (*OpenGL* possui apenas o componente gráfico) [13].

4.3.2.1 DirectX Graphics

DirectX Graphics é o componente *DirectX* desenvolvido para prover suporte a gráficos 2D e 3D. No mercado moderno atual, os esforços despendidos no desenvolvimento de jogos têm sido bastante dirigidos às inovações em tecnologias de software 3D e projetos de aceleradores gráficos. *DirectX Graphics* é a junção dos componentes *DirectDraw* e *Direct3D*, separados em versões anteriores do *DirectX*, em um

[4]Hardware Abstraction Layer
[5]Hardware Emulation Layer
[6]Graphics Driver Interface

único componente que pode ser utilizado para qualquer modelo de programação gráfica. Neste processo de junção, um grande número de modificações foram feitas às interfaces e métodos destes componentes para suportar da forma mais eficiente possível os mais novos hardwares com aceleração 3D.

DirectX Graphics é uma interface de software que provê acesso direto para os dispositivos de vídeo enquanto gerencia a compatibilidade com a interface gráfica do Windows (GDI), provendo independência dos hardwares de forma transparente aos usuários, e proporcionando acesso a todas as características inerentes a estes.

O gerenciamento de *DirectX Graphics* é baseado em vértices, polígonos, e comandos para controlá-los. Este componente realiza acesso imediato para transformação, iluminação, e rasterização no seu pipeline gráfico (HAL), e oferece uma robusta emulação por software (HEL) para dispositivos gráficos sem aceleração 3D.

Quase todos os aceleradores gráficos do mercado são projetados para atuar em aplicações baseadas em *DirectX Graphics*. Recentemente, a *Microsoft* colaborou com a *nVidia* para desenvolver a próxima geração do acelerador gráfico *geForce* para uso no console *X-Box* (atualmente usa exclusivamente *DirectX*) [39]. Como resultado, quase todas as empresas de desenvolvimento de jogos têm inserido *DirectX Graphics* nos seus projetos, na ordem de produzir jogos capazes de explorar todas as características dos novos aceleradores gráficos.

4.3.2.2 DirectX Audio

DirectX Audio, da mesma maneira que o *DirectX Graphics*, é a junção dos componentes *DirectSound* e *DirectMusic* de versões anteriores do *DirectX* em um único componente. *DirectX Audio* pode ser utilizado para qualquer tipo de programação com áudio, inclusive áudio 3D, provendo um sistema completo para implementar uma trilha sonora dinâmica que tome vantagem de aceleração por hardware, *Downloadable Sounds* (DLS), *DirectX Media Objects* (DMOs), e efeitos avançados de posicionamento 3D.

DirectX Audio suporta a leitura e reprodução de arquivos ou recursos de sons em MIDI, Wave, ou no formato do *DirectMusic Producer*. *DirectX Audio* realiza execução de múltiplas fontes simultaneamente, localização de sons num ambiente 3D, e execução de diversos efeitos sonoros.

Além de apresentar facilidade para tarefas básicas, este componente também permite acesso a baixo nível, ou seja, ele pode ser estendido de forma que aplicações especializadas possam implementar novos objetos para formatos proprietários.

4.3.2.3 DirectInput

DirectInput é um componente de entrada virtual independente de hardware. Suporta uma enorme variedade de dispositivos de entrada, incluindo a tecnologia *force-feedback*[7] [13].

Este componente provê serviços para dispositivos que também não são suportados pelo *Windows*. Realiza comunicação diretamente com os drivers dos hardwares, proporcionando um acesso melhor e muito mais rápido aos dados de entrada que através do sistema padrão de mensagens do *Windows*. *DirectInput* pode habilitar uma aplicação para receber dados dos dispositivos de entrada até mesmo quando esta aplicação estiver fora de foco (*background*), se necessário. Isto se deve ao grau de cooperação entre aplicações que pode ser atribuído aos dispositivos. *DirectInput* possui a capacidade de enumerar todos os dispositivos de entrada conectados ao computador, e suas propriedades particulares. Através de mapeamento de ações, as aplicações podem recuperar os dados de entrada sem a necessidade de saber que tipo de dispositivo está sendo usado para gerá-lo.

A extensão dos serviços, e o ótimo desempenho de *DirectInput*, fazem deste componente, uma ferramenta de bastante valia para o desenvolvimento de jogos, simuladores, e outras aplicações interativas de tempo real.

4.3.2.4 DirectShow

DirectShow suporta captura e reprodução em alta qualidade de *streams* multimídia, incluindo uma ampla variedade de formatos, como *Advanced Streaming Format* (ASF), *Motion Picture Experts Group* (MPEG), *Audio-Video Interleaved* (AVI), *MPEG Audio Layer-3* (MP3), e Wave. *DirectShow* utiliza os dispositivos *Windows Driver Model* (WDM) ou os dispositivos de vídeo mais antigos do *Windows* para realizar a captura. Este componente automaticamente detecta e utiliza a aceleração por hardware disponibilizada para áudio e vídeo, e também suporta sistemas sem aceleração por hardware.

DirectShow simplifica as tarefas de reprodução, conversão de formato, e captura de mídias. Ao mesmo tempo, provê acesso a uma arquitetura de controle de *streams* para aplicações que requerem soluções personalizadas. Um componente *DirectShow* proprietário também pode ser criado para suportar novos formatos ou efeitos customizados. Diversos tipos de aplicações podem ser construídas, como: reprodutores

[7]Tecnologia encontrada nos dispositivos de entrada (joysticks e gamepads) para a recepção de sinais (fisicamente) das ações sofridas pelos jogadores

de DVD, editores de vídeo, conversores de AVI para ASF, reprodutores de MP3, e aplicações de captura de vídeo digital.

4.3.2.5 DirectPlay

DirectPlay é o componente *DirectX* que provê ferramentas para desenvolver aplicações multiusuário, como jogos e *chats*. Ele contém uma camada que isola a aplicação dos detalhes de rede, e apresenta características que simplificam o processo de implementação de aplicações multiusuário. *DirectPlay* suporta também a criação e gerencia de sessões ponto a ponto e cliente/servidor, gerência de usuários e grupos de uma sessão, e comunicação por voz.

4.4 Outras ferramentas de suporte

A alguns anos atrás, para desenvolver jogos não era necessário mais que um editor de texto e algum programa de desenhos simples. No entanto, atualmente, isso é completamente diferente. No mínimo, necessita-se de um compilador otimizado, um programa gráfico 2D robusto, e um processador de sons. Se a aplicação for 3D, também necessita-se de um modelador de objetos 3D. A seguir é apresentada uma breve categorização das aplicações de suporte mais utilizadas no desenvolvimento dos jogos atuais.

4.4.1 Compiladores

Atualmente, as linguagens padrão para o desenvolvimento de jogos são *C* e *C++*, e futuramente provavelmente prevalecerá o *C++*. Apesar de *Java* apostar fortemente no *Java3D*, ele não chega a ser competitivo devido ao seu baixo desempenho, principalmente quando se trata de tempo real.

Para o desenvolvimento em ambiente *Windows*, o compilador mais utilizado pela indústria de jogos é o *Microsoft Visual C++ 6.0* [65]. Ele apresenta todo um conjunto de características do sistema operacional, e os .EXEs gerados por ele são os mais rápidos. O *Borland C++ Builder 4.0* é um outro compilador que também trabalha bem no *Windows*, e ainda possui uma nova versão para *Linux* (ver exemplos na figura 4.3). Outros compiladores para as demais plataformas são o *Watcom C++* e o *GCC*.

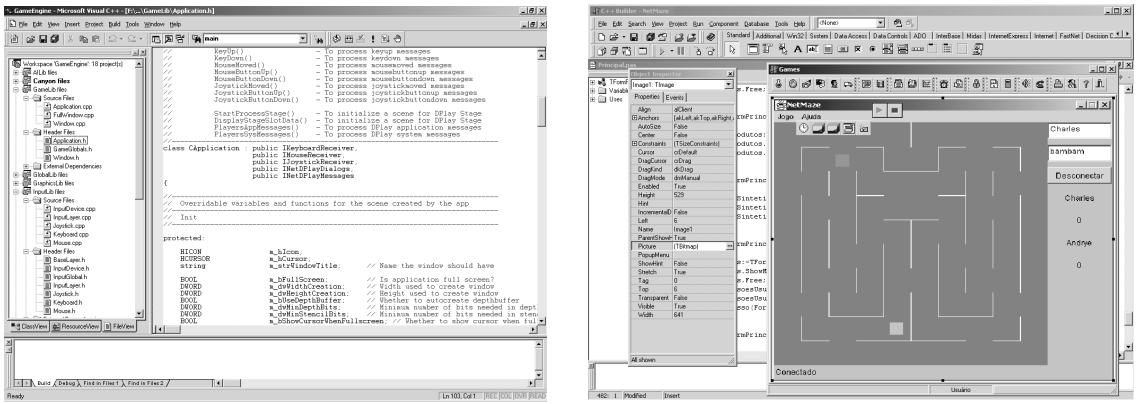


Figura 4.3: Exemplos de compiladores - Visual C++ (esquerda) e C++ Builder (direita)

4.4.2 Editores de imagens 2D

Estes editores são utilizados para desenhos e processamento de imagens. Eles permitem desenhar imagens pixel a pixel e então manipulá-las, editar fotos com alta qualidade, criar e otimizar gráficos para Web e desenhos artísticos, projetar animações, personalizar imagens, e aplicar efeitos especiais (ver figura 4.4). O pacote líder em relação ao custo versos desempenho é o *Jasc Paint Shop Pro 7*[8]. Um outro muito robusto e bastante adotado pelo mercado é o *Corel Photo-Paint 9.0*[9]. Mas o favorito do mercado é o *Adobe Photoshop 6.0*[10] que trabalha com produções em processamento de imagens.

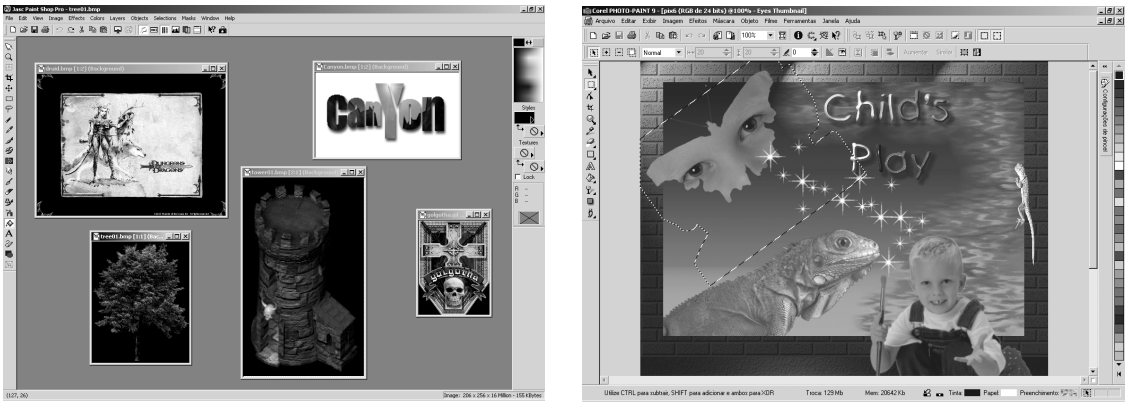


Figura 4.4: Exemplo de editores de imagens 2D - Paint Shop Pro (esquerda) e Photo-Paint (direita)

[8]http://www.jasc.com
[9]http://www3.corel.com
[10]http://www.adobe.com/products/photoshop/main.html

Estas ferramentas também possibilitam a criação de vetores de formatos editáveis e texto. Provêem operações para adição, subtração, limitação e inversão que combinam rapidamente vetores de formatos básicos em formatos complexos.

4.4.3 Modeladores 3D

Os modeladores de objetos 3D são ferramentas profissionais avançadas para criação e animação de caracteres e produção de efeitos visuais. Eles são ideais para criar o fotorealismo em 3D para a animação industrial. Os modeladores 3D apresentam resposta em tempo real de todas as operações avançadas e manipulação direta de interfaces de usuário. Eles contém um motor de renderização baseado em *Direct3D/OpenGL*, e são ferramentas eficientes e produtivas para o desenvolvimento de jogos, filmes e efeitos visuais de televisão (ver figura 4.5).

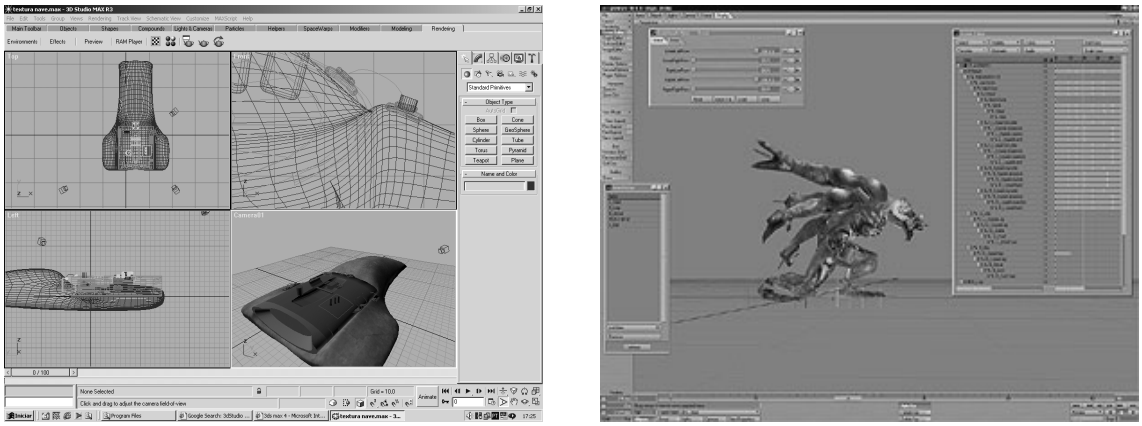


Figura 4.5: Exemplos de modeladores de objetos 3D - 3D Studio Max (esquerda) e TrueSpace (direita)

Estes modeladores são programas bastante complexos e de altíssimo custo. Podem chegar a custar dezenas de milhares de dólares. Recentemente, modeladores de mais baixo custo têm mostrado um alto poder de produção, como é o caso do *Caligari TrueSpace 5.1*[11]. O *3D Studio Max 4*[12] é o *best seller*, o mais utilizado pela indústria de jogos para modelagem 3D, apresenta alto poder de produção e absoluto fotorealismo. O *Maya 3*[13] e o *LightWave 6*[14] também são modeladores 3D bastante famosos que apresentam ótimos resultados.

[11] http://www.caligari.com/products/
[12] http://www2.discreet.com/products
[13] http://www.aliaswavefront.com/en/Home/homepage.html
[14] http://www.newtek.com/products/lightwave

4.4.4 Processadores de sons e músicas

Os processadores de sons e músicas são programas utilizados para a criação e edição de efeitos sonoros e trilhas musicais. Um dos melhores programas de processamento de sons digitais (WAVs) do mercado é o *Sound Forge*[15]. Ele apresenta uma enorme capacidade de processamento e é de fácil utilização.

No caso de sons baseados em MIDI, é necessário um programa de sequenciamento de pacotes. Um dos melhores e de custo razoável é o *CakeWalk*[16] (ver figura 4.6).

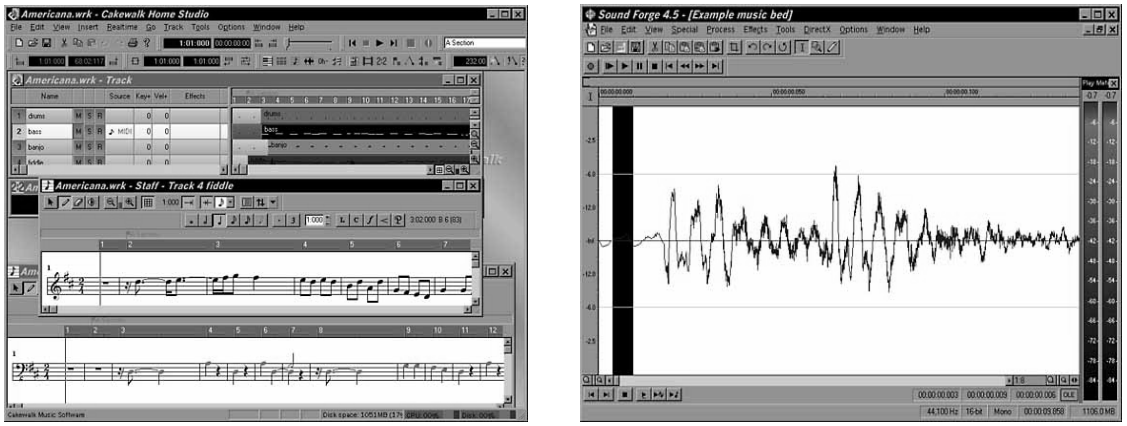


Figura 4.6: Exemplo de editores de som - CakeWalk (esquerda) e Sound Forge (direita)

4.5 Conclusão do capítulo

As placas aceleradoras e as bibliotecas multimídia são ferramentas de extrema importância para a qualidade e o bom desempenho de jogos de computador. Além disso, as bibliotecas multimídia facilitam enormemente a construção de jogos devido a independência de hardware que elas oferecem.

As ferramentas de edição de imagens e modelagem de objetos são fundamentais para elevar o nível de qualidade dos jogos, pois atualmente elas geram ambientes e personagens extremamente realistas, possibilitando a construção de cenários bastante complexos. Em resultado ao uso destas ferramentas, os projetistas de jogos tem melhorado cada vez mais o nível de realismo apresentado por seus produtos.

Este capítulo apresentou um resumo das ferramentas que compõem a infra-estrutura para o desenvolvimento de jogos de computador, enfocando a importância

[15]http://www.sonicfoundry.com
[16]http://www.cakewalk.com

do uso destas ferramentas nestas aplicações. Consequentemente, serviu para apresentar a infra-estrutura necessária para ser utilizada no desenvolvimento do *FORGE V8*, o qual sua estrutura é apresentada no capítulo 6 e sua implementação no capítulo 7.

Capítulo 5

A Engenharia de Software

Em capítulos anteriores, foi bastante enfocada a necessidade de uso dos princípios de engenharia de software para o desenvolvimento de software de boa qualidade, e mais especificamente, de projetos orientados a objetos com a intenção de aplicá-los a jogos de computador. Projetos orientados a objetos provêem suporte ao desenvolvimento particionado de sistemas, possibilitam modelagens baseadas em objetos reais, e ajudam a assegurar um bom nível de qualidade e gerenciamento de produtos e do processo de desenvolvimento. Este capítulo apresenta uma discussão sobre os princípios de projetos orientados a objetos que compõem o estado da arte da engenharia de software e são extensamente utilizados pela indústria software em geral. O objetivo deste capítulo é dar o embasamento necessário ao entendimento do *framework* proposto neste trabalho, que é exposto no capítulo 6.

5.1 Princípios básicos da Engenharia de Software

Projetos orientados a objetos têm apresentado sucesso na indústria de software em geral devido a melhor qualidade oferecida as aplicações e por permitir a construção de grandes sistemas melhor gerenciáveis [49].

Uma razão para o sucesso do paradigma orientado a objetos é que ele é bem mais voltado para a maneira como pensamos. As funcionalidades de um sistema podem ser corretamente particionadas em componentes e então empacotadas, isoladas de outros subsistemas de um projeto, o que é difícil conseguir com a programação procedural.

Os fatores de qualidade implicam na condução de propostas para a construção de arquiteturas flexíveis, produzidas por componentes de software autônomos e reusáveis (módulos). Componentes tem-se apresentado como uma excelente técnica

de desenvolvimento [49]. Na substituição de escrever um grandioso componente de software (desenvolvimento monolítico), escreve-se um conjunto de pequenos componentes que se comunicam entre si. Esta decomposição caracteriza um bom grau de independência que pode ser aplicado a estes componentes, de maneira que eles possam ser utilizados em diferentes projetos sem a necessidade de ajustes, além da possibilidade de realizar modificações e testes também de forma independente.

Estas técnicas aplicadas de forma ideal, provêem componentes que podem ser facilmente conectados para construir um novo sistema. Não é nada interessante que todo novo desenvolvimento deva partir do início. Deveria existir um catálogo de componentes de software, como existem catálogos de dispositivos de hardware, possibilitando a reutilização destes componentes, sem a necessidade de "reinvenção da roda". Consequentemente, diminuindo a quantidade de código a ser construído, e provavelmente gerando um resultado muito melhor [23].

A seguir são apresentados os principais conceitos e técnicas da engenharia de software relacionados ao desenvolvimento de projetos orientados a objetos, os quais podem ser largamente utilizados na construção de jogos de computador.

5.1.1 Modularidade

No desenvolvimento de um sistema de software, módulos são pacotes organizados que apresentam autonomia, coerência, e robustez. Eles devem ser projetados para serem extensíveis e portáveis de modo que possam ser plugados em outros sistemas. Existem cinco critérios que ajudam a caracterizar a modularidade, eles são [49]:

- Decomposibilidade: o problema deve ser decomposto em diversos subproblemas de maneira que sua solução possa ser encontrada separadamente;
- Composibilidade: o sistema suporta a produção de elementos de software que podem ser livremente combinados para produzir um novo sistema;
- Compreensibilidade: o sistema produz módulos que podem ser compreendidos separadamente, ou juntamente com uma pequena quantidade de outros módulos;
- Continuidade: uma modificação necessária na especificação de um problema resulta em alteração de apenas um (ou poucos) módulo(s);
- Proteção: o sistema apresenta uma arquitetura em que o efeito de condições anormais num determinado módulo, em tempo de execução, permanece restrito a ele próprio.

Estes cinco critérios apontam para cinco princípios que devem ser seguidos para atingir a modularidade. A relação entre eles é apresentada entre parênteses [16].

- Unidade lingüística modular: módulos devem corresponder à unidades sintáticas na linguagem utilizada (decomposibilidade, composibilidade, proteção);
- Poucas interfaces: todo módulo deve se comunicar com uma menor quantidade possível de outros módulos (continuidade, proteção);
- Pequenas interfaces: se dois módulos devem se comunicar, eles devem trocar um mínimo de informações possíveis (continuidade, proteção);
- Interfaces explícitas: sempre que dois módulos se comunicarem, eles devem se acoplar diretamente, ou seja, as informações trocadas devem ser restritas aos seus dados (decomposibilidade, composibilidade, continuidade, compreensibilidade);
- Ocultar informações: toda a informação sobre um módulo deve ser privada, a menos que seja extremamente necessário declará-la como pública (continuidade, não necessariamente proteção).

Módulos possuem duas maneiras de formatação, eles podem ser abertos ou fechados [49]. Um módulo aberto é caracterizado por ser disponibilizado para extensão. É possível adicionar novos campos a sua estrutura de dados ou adicionar novas funções para operar com estas estruturas. Um módulo fechado é caracterizado por ser disponibilizado apenas para ser usado por outros módulos. Ele deve apresentar uma interface estável e bem definida.

5.1.2 Reusabilidade em software

Reusabilidade é um problema básico de engenharia de software. Porque gastar tempo desenvolvendo algo quando este já existe? No entanto, esta questão não tem uma simples resposta. Vários fatores estão envolvidos neste problema [49]. Bibliotecas que apresentem um bom conjunto de características necessárias para o desenvolvimento de uma determinada aplicação podem ser encontradas, mas podem necessitar de licenciamento, adaptações, ou apresentarem problemas que não serão solucionados no intervalo de tempo requerido. Neste caso, quando componentes tiverem de ser desenvolvidos localmente, devem ser projetados de forma que o seu reuso seja maximizado. As duas principais formas de reusabilidade no desenvolvimento de software são reuso de código e reuso de projeto [65].

O reuso de código é apresentado pela produção de um conjunto de componentes e ferramentas que possam ser aproveitados por uma série de produtos. Desde o início do desenvolvimento, estes componentes e ferramentas devem ser projetados com a possibilidade de expansões futuras. Devido a necessidade de expansões, um conjunto de componentes (módulos) deve ser construído de forma que possa ser liberado de forma incremental, ou seja, deve permitir o funcionamento parcial de um sistema. Abaixo são apresentados alguns problemas inerentes às estruturas modulares, que devem ser resolvidos para produzir componentes reusáveis [49].

- Variação de tipos: um módulo deve ser aplicado a estruturas de diferentes tipos;
- Variação em estrutura de dados e algoritmos: como as ações executadas durante um algoritmo podem depender do tipo da estrutura de dados, o módulo deve permitir o direcionamento para variações de diferentes estruturas;
- Rotinas relacionadas: um módulo deve ter acesso as rotinas para manipular as estruturas de dados diferentes;
- Independência de representação: um módulo deve permitir ao usuário especificar uma operação sem saber como ela é implementada ou qual o tipo de estrutura de dados que está sendo utilizada;
- Comunidade com outros subgrupos: devem ser construídas interfaces abstratas que não exponham estruturas de dados, permitindo a sua implementação de diferentes formas, garantindo a não repetição de blocos de código similares.

Um grande feito da comunidade de desenvolvimento de software que tem resultado em um dos mais utilizados métodos de reuso, tem sido o reuso de projeto através de padrões, formando um catálogo de soluções para diversos problemas de projeto. Devido ao reuso de projeto ser o foco maior deste trabalho na estruturação do *FORGE V8*, ele é apresentado numa seção a parte.

5.2 Reuso de projeto - padrões de projeto

Padrões de projeto são bastante populares pela comunidade de orientação a objetos devido a proverem a reutilização das informações de bons projetos [37]. Um padrão descreve um problema que ocorre diversas vezes no ambiente, e a sua solução. Esta descrição é feita de uma maneira que possa ser utilizada infinitas vezes, sem a

necessidade de ser rescrita [23]. As soluções são expressas em termos de objetos e interfaces. Normalmente, um padrão tem quatro elementos essenciais: o seu nome, a descrição do problema em questão, a solução, e as possíveis conseqüências.

O *nome do padrão* é um identificador que é utilizado para descrever um problema, a sua solução, e quais suas conseqüências. O nome de um padrão incrementa o vocabulário de soluções de problemas em geral. Ele deve ser projetado com alto nível de abstração, fazendo-se interessante para possibilitar diálogos e documentações padronizadas, consequentemente aumentando a facilidade para pensar sobre projetos e prover melhor nível de comunicação. Utilizar bons nomes para padrões é necessário para uma atualização bem definida do seu catálogo.

A *descrição do problema* explica quando aplicar o padrão, mostrando a situação e seu contexto. Ela pode descrever cada problema de projeto específico e como representar algoritmos em objetos, e pode descrever estruturas de classes ou objetos que são sintomáticas de um projeto inflexível. Algumas vezes o problema poderá incluir uma lista de condições que devem ser verificadas antes que o padrão seja aplicado.

A *solução* descreve os elementos que constróem o projeto, seus relacionamentos, responsabilidades, e colaborações. A solução não descreve um projeto concreto ou implementação em particular, porque um padrão é como um *template*[1] que pode ser aplicado em muitas diferentes situações. Ela provê uma descrição abstrata de um problema de projeto e sua resolução através de uma configuração geral de elementos (classes e objetos).

As *conseqüências* descrevem os resultados da aplicação do padrão. Elas são críticas na disponibilização de alternativas de projeto em relação ao custo e benefício de sua aplicação. As conseqüências para o software freqüentemente se relacionam ao tempo gasto em se obter o resultado. Quando necessário, são descritos os problemas que podem ocorrer com aplicações em determinadas linguagens e implementações. As conseqüências de um padrão incluem o impacto dele na flexibilidade, extensibilidade ou portabilidade de um sistema.

O ponto de vista dos desenvolvedores pode afetar a interpretação do que é ou não é um padrão. Um padrão pessoal pode ser um bloco qualquer de construção primitiva. Porém, padrões de projeto têm um certo nível de abstração. Eles não são projetados como listas ligadas e tabelas *hash* que podem ser codificadas em classes e reutilizadas. Eles não são complexos, projetos de domínio específico para uma aplicação, ou subsistemas. Padrões de projeto são descrições de comunicação

[1]Parametrizador que realiza operações independente do tipo do objeto

de objetos e classes que são personalizadas para resolver um problema de projeto geral num contexto particular. Eles geram sistemas reusáveis, flexíveis, extensíveis, e portáveis [23].

Embora padrões apenas descrevam projetos, em orientação a objetos, eles são baseados em soluções práticas que têm sido extensamente implementadas nas principais linguagens de programação orientadas a objetos. A escolha da linguagem de programação é importante porque ela influencia no ponto de vista, e determina o que pode e não pode ser implementado facilmente. Atualmente, *Smalltalk*, *C++* e *Java* são linguagens características para a utilização de padrões de projeto [37].

Padrões de projeto fazem mais fácil o sucesso do reuso de projetos e arquiteturas, consequentemente, tornando mais fácil o desenvolvimento de novos sistemas. Neste contexto, se inserem os *frameworks*, que são grandes consumidores de padrões. Além de *frameworks* poderem utilizar padrões na sua arquitetura, eles também podem utilizar a linguagem de padrões para a sua documentação [36]. A documentação de um *framework* deve apresentar três diretrizes as quais a linguagem de padrões ajuda a descrever: a descrição da sua proposta, como desenvolvedores de aplicação podem usá-lo, e os seus detalhes de projeto (estruturação) [36]. A linguagem de padrões não é uma linguagem formal, mas sim uma coleção de padrões inter-relacionados, embora eles apresentem um vocabulário que fala sobre um problema particular. Ela provê um processo para resolução ordenada de problemas de desenvolvimento de software. Ambos padrões e a linguagem de padrões ajudam na documentação e gerenciamento de sistemas por fornecer uma especificação explícita de interações de classes e objetos, ajudam os desenvolvedores na comunicação do conhecimento e na aprendizagem de um novo paradigma de projeto ou estilo arquitetural, e ajudam novos desenvolvedores a fugir das armadilhas que têm tradicionalmente sido conhecidas somente com o alto custo da experiência [68].

5.3 Frameworks

Um *framework* é um projeto reusável que representa a estrutura principal de uma aplicação orientada a objetos - projeto de todo ou parte de um sistema de software que é representado por um conjunto de classes abstratas e uma maneira de interação com instâncias destas classes - que pode ser especializado para produzir aplicações personalizadas, a fim de reduzir o custo e aumentar a qualidade [17] (ver figura 5.1).

Um *framework* provê uma infra-estrutura para a solução de um problema. Ele pode conter diversas camadas de abstração de acordo com a complexidade do seu

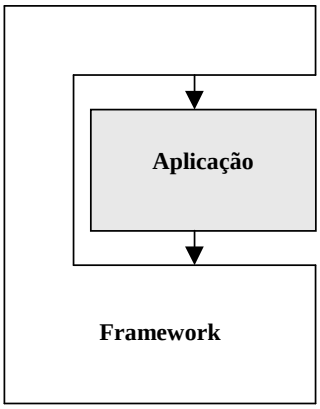


Figura 5.1: Diagrama de interação de um *framework*

modelo orientado a objetos. Isto ocorre devido a tratar detalhes de baixo nível e provê uma interface abstrata em alto nível, permitindo uma maior focalização dos usuários no conteúdo e comportamentos específicos das aplicações a serem desenvolvidas.

Frameworks vão além de simples bibliotecas de classes. Enquanto em uma biblioteca de classes básica os usuários simplesmente realizam chamadas para controlá-la, em um *framework*, os usuários codificam as chamadas para os componentes disponibilizados com a finalidade de tratar eventos e determinar comportamentos personalizados. Isto é feito pela derivação de classes providas pelo *framework* e sobrescrição de funções virtuais membros destas classes.

Os principais benefícios de *frameworks* orientados a objetos são a modularidade, reusabilidade, extensibilidade e controle que eles proporcionam aos desenvolvedores. *Frameworks* exibem a modularidade por encapsular detalhes de implementação através de interfaces estáveis, facilitando a localização das modificações necessárias ao seu projeto e implementação, e reduzindo o esforço requerido para o entendimento e gerenciamento [17]. Estas interfaces estáveis possibilitam a reusabilidade devido a definição de componentes genéricos que podem ser reaplicados para a criação de novas aplicações. O reuso de componentes de um *framework* pode elevar substancialmente a produtividade de programadores, aumentando a qualidade, o desempenho, a confiabilidade e a interoperabilidade do software [58]. A tabela 5.1 apresenta as vantagens e desvantagens do desenvolvimento de software baseado em *frameworks* com relação ao desenvolvimento monolítico.

Uma aplicação que utiliza um *framework* para ser desenvolvida deve se adaptar ao projeto e modelos de colaboração deste *framework*, o que causa automaticamente uma padronização nesta aplicação. Desde que um *framework* tenha sido utilizado

diversas vezes, ele também pode representar um tipo de padrão.

Tabela 5.1: Vantagens e desvantagens da utilização de um *framework*

Vantagens	**Desvantagens**
A média de tamanho dos projetos subseqüentes é encurtada	O primeiro projeto é muito longo
Realiza a independência de plataforma de maneira mais fácil	Bibliotecas que encapsulam os módulos devem ser desenvolvidas para cada plataforma
Apresenta código mais confiável	O código é mais genérico e mais difícil de desenvolver
Apresenta código reusável e gerenciável	Inicialmente, o código toma muito tempo para ser desenvolvido
O conhecimento é propagado entre todos os desenvolvedores	Novos desenvolvedores têm que aprender a lidar com bibliotecas não familiares
Aumenta a visibilidade de progresso do projeto	Necessita de melhor administração
Aumenta a especialização dos desenvolvedores	Menor flexibilidade

Diferentemente de padrões de projeto, *frameworks* não são apenas idéias, eles são implementações de idéias. Eles normalmente contém diversos padrões de projeto em um diferente nível de abstração, ou sejam, implementados na aplicação de um problema. Padrões de projeto são elementos micro-arquiteturais de *frameworks* [23]. O resultado deste conjunto de projeto reusável é que se faz possível construir uma aplicação mais rapidamente, possibilitando a sua evolução de forma independente [49].

Todos os benefícios e princípios de projeto, de diferentes *frameworks*, são em grande escala independentes do domínio em que eles são aplicados. Usualmente, existem três classes de *frameworks* [17]: *frameworks* de infra-estrutura, de integração, e de aplicação.

Os *frameworks* de infra-estrutura (sistema) são utilizados para o desenvolvimento de infra-estruturas de sistemas portáveis e eficientes tais como sistemas operacionais, sistemas de comunicação, e ferramentas de interfaces de usuário e de processamento de linguagens. Estes *frameworks* abstraem mecanismos tais como controle de processos, *threading*, dispositivos de entrada/saída, gerenciamento de memória, e comunicação de *threads*. Eles são principalmente usados internamente em empresas de desenvolvimento de software e não são vendidos de forma direta.

Os *frameworks* de integração (*middleware*) são utilizados para integrar aplicações distribuídas. Estes *frameworks* são projetados para melhorar a habilidade dos desenvolvedores para a sua infra-estrutura de software, na intenção de trabalhar num

ambiente distribuído. Eles representam um mercado promissor. Exemplos comuns incluem ORBs (Object Request Broker), *middlewares* orientados a mensagens, e bancos de dados transacionais.

Os *frameworks* de aplicação (empreendimento) são utilizados para atingir amplos domínios de aplicações, e geram grandes economias para as empresas [37]. Em relação aos outros modelos de *framework*, este apresenta custo muito elevado para o seu desenvolvimento e/ou compra. No entanto, *frameworks* de aplicação podem prover um retorno substancial ao seu investimento, desde que eles suportem o desenvolvimento de aplicações e produtos para os usuários finais. Como resultado, em relação ao custo-benefício, normalmente é mais indicado o desenvolvimento local de *frameworks* de aplicação, enquanto para os *frameworks* de infra-estrutura e de integração, é mais indicado a aquisição externa [37].

Independente do escopo, *frameworks* também podem ser classificados em função das técnicas utilizadas para sua extensão, conhecidas como caixa branca e caixa preta [17]. *Frameworks* de caixa branca dependem fortemente das características orientadas a objetos, como herança e encapsulamento dinâmico, na intenção de possibilitar a sua extensão. As funcionalidades existentes são reusadas e estendidas através de herança das classes bases do *framework*, sobrescrevendo métodos predefinidos utilizando padrões de projeto como o *Template Method* [23]. Estes *frameworks* requerem que os seus usuários, desenvolvedores de aplicação, tenham profundo conhecimento de toda sua estrutura interna. Eles são utilizados em grande escala, e tendem a produzir sistemas que são extremamente acoplados aos seus detalhes específicos de hierarquias de herança. Caixa branca é um estágio do ciclo de vida de um *framework*, similar a um adolescente humano. Neste estágio, ele está tentando determinar a sua razão de ser, e compartilha valores e individualidades. Ele é imperativo na continuação à execução de implementações adicionais (projetos baseados no *framework*), para somente com estas interações saber diferenciar a variabilidade da estabilidade, e o genérico do específico.

Em contraste, *frameworks* de caixa preta suportam a extensibilidade por definir interfaces para componentes que podem ser plugados via composição de objetos. As funcionalidades existentes são reutilizadas pela definição de componentes que se adaptam para uma interface particular, utilizando padrões de projeto como *Strategy* e *Functor* [23] para realizar esta integração. Eles apresentam o ganho da experiência de múltiplas implementações e diversos ciclos de refinamentos, resultando na sua maturidade. Como resultado, eles são geralmente mais fáceis de utilização e extensão que os de caixa branca. No entanto, são mais difíceis, levam mais tempo para serem

desenvolvidos e requerem uma definição das interfaces que antecipem um amplo conjunto dos casos de uso potenciais.

5.4 Conclusão do capítulo

Padrões de projeto são idéias arquiteturais aplicáveis para diversos domínios de aplicação, que têm feito importantes contribuições para o desenvolvimento da tecnologia de objetos. Como eles continuam a ser criados e publicados, ajudarão cada vez mais os desenvolvedores no benefício da experiência.

A focalização extensiva em *frameworks* de aplicação pela comunidade de orientação a objetos oferece aos desenvolvedores de software uma importante técnica para reuso, e meios para capturar a essência do sucesso de padrões de projeto, arquiteturas, componentes e mecanismos de programação.

Frameworks são caracterizados por serem tão bons quanto os seus construtores e usuários. Eles são complexos, difíceis de desenvolver e difíceis de aprender, portanto custam mais, e requerem melhor documentação, maior período de treinamento, e grupos de desenvolvimento com um maior grau de habilidade que em sistemas comuns. Mas, como eles apresentam ótimos resultados, então projetos complexos que queiram tomar vantagem desta técnica, devem pagar este preço.

Este capítulo teve o objetivo de apresentar um resumo do estado da arte de projetos orientados a objetos, com a intenção de mostrar que o uso de *frameworks* se torna o ferramental ideal para o desenvolvimento de jogos de computador, devido à alta complexidade envolvida nestas aplicações. Como também, que as técnicas de engenharia de software trazem excelentes benefícios ao desenvolvimento, possibilitando a produção de maneira muito mais rápida e com alto nível de qualidade, garantindo a construção de aplicações eficientes, de fácil gerenciamento, e confiáveis [49]. Assim, servindo de base para o *FORGE V8*, o qual é exposto no próximo capítulo.

Capítulo 6

FORGE V8- O Framework Proposto

Devido a um jogo de computador ser um sistema bastante complexo, e o propósito de *frameworks* orientados a objetos ser a simplificação de sistemas complexos, então a aplicação de *frameworks* para o desenvolvimento de jogos torna-se um processo natural. O fundamental do projeto de um *framework* é permitir que futuros desenvolvimentos de aplicações sejam realizados com menor dificuldade. Neste contexto, um *framework* deve incorporar flexibilidade, extensibilidade e fácil configuração. Padrões de projeto apresentam a base para arquitetura de *frameworks* flexíveis, encapsulando variações em classes ou componentes. Este encapsulamento permite interfaces e polimorfismo, provendo implementações individuais de componentes que expõem uma mesma interface. A intenção deste capítulo é apresentar em detalhes cada um dos módulos do *framework* proposto, o *FORGE V8*, e como eles se integram de forma a constituir uma interessante ferramenta para ser utilizada no desenvolvimento de jogos de computador e aplicações multimídia. Para a descrição deste *framework* foi utilizado o formato de linguagem de padrões [36], que tem o objetivo de definir a proposta do *framework*, como os usuários devem usá-lo, e a sua estruturação.

6.1 A proposta do *FORGE V8*

Para a proposta de um *framework* para jogos, é necessário apresentar uma idéia clara do que se deseja alcançar com ele. Quais os seus objetivos? Como será sua documentação para os usuários? Quais suas características? Estas questões são tratadas nesta seção.

6.1.1 Objetivos

Os objetivos de um *framework* afetam fortemente a sua implementação. A meta do *FORGE V8* é ser um motor genérico e portável para jogos e aplicações multimídia, apresentando suporte às diversas características essenciais para estas aplicações como, interface gráfica, sonorização, inteligência e modelagem física de personagens, e multiusuários em rede, além de ferramentas tais como editores de cenários.

O *FORGE V8* deve possibilitar a construção de jogos em 2D, 2 1/2D e 3D, utilizando uma mesma arquitetura de baixo nível para renderização em 3D, tanto para ambientes abertos quanto para ambientes fechados. No caso dos jogos em 2D e 2 1/2D, o uso de renderização será possível devido a *tiles* e *sprites* poderem ser construídos com a junção de dois triângulos. Desta forma aproveitando-se todas as características de aceleração por hardware, e possibilitando o uso de iluminação dinâmica. Esta técnica se torna ideal porque muitos hardwares aceleradores 3D são realmente aceleradores 2D, com a exceção dos que também executam transformação e iluminação. O que todos estes hardwares fazem é projetar texturas em triângulos - superfícies 2D. Então, é absolutamente natural usar a aceleração de uma placa de vídeo 3D para renderizar em 2D, principalmente por não haver a necessidade de envolver as dificuldades da matemática de matrizes tridimensionais [79].

Esta proposta genérica pretende ser conseguida com a construção de interfaces que possam ser implementadas para os diversos modelos, de maneira que os usuários possam configurar as suas aplicações para o modelo que lhe convier. A intenção é gerar componentes e bibliotecas com um bom grau de independência e apresentar um alto nível de integração para o desenvolvimento de jogos e aplicações multimídia. Além disso, bibliotecas de ligação dinâmicas poderão ser utilizadas para disponibilizar a implementação das interfaces e facilitar as extensões e liberações de versões.

6.1.2 Documentação

A intenção do *FORGE V8* é ser desenvolvido como um produto de fácil utilização para desenvolvedores de jogos em geral, portanto sua documentação precisa ser clara e bem projetada. Neste processo, deve-se utilizar um gerador de documentação para extrair comentários do código fonte, e seus comentários devem ser bem explicativos. O estilo de padronização e documentação de código do *FORGE V8* é apresentado no apêndice B desta dissertação. Além disso, neste capítulo é apresentada a documentação do projeto do *FORGE V8* construída com base na linguagem de padrões, que é uma maneira de mais alto nível dos usuários conhecerem a sua estruturação.

6.1.3 Características

As características almejadas pelo *FORGE V8* são apresentadas divididas em sete categorias: interface gráfica, sonorização, inteligência e modelagem física de personagens, controle de objetos, multiusuários em rede, editores de cenários e genéricos.

- Interface gráfica

  Suporte a renderização gráfica baseada em *DirectX Graphics* para Windows, e *OpenGL* para Linux, Unix, BeOS, MacOS/9, OS/2 e Windows; rasterização otimizada por software; correção de profundidade para ambientes 3D; superfícies curvas; iluminação estática e dinâmica; neblina volumétrica; espelhamento; *skyboxes*; sistemas de partículas; modelos de malhas estáticas e dinâmicas; *tiles* para ambientes 2D e 2 1/2D; *sprites* 2D e 3D; *billboarding*; mapeamento, animação, *morphing* e filtragem de texturas; sombreamento dinâmico; *mipmapping*; *morphing* de vértices; teste de visibilidade; fontes (textos); efeitos d'água; câmeras (fixa para ambientes 2D e 2 1/2D); *plugins* para importação/exportação de modelos 3D Studio Max; e sistema de gerenciamento de texturas [16].

- Sonorização

  Suporte a reprodução de sons para efeitos e vozes; músicas via *streaming*; posicionamento e atenuação de sons 3D; e sistema de gerenciamento de trilhas sonoras.

- Inteligência e modelagem física de personagens

  Suporte a tomada de decisões (raciocínio); *scripting*; simulação física de corpos rígidos integrados; simulação física de veículos; comportamento individual; comportamento em grupo; e comportamento híbrido.

- Controle de objetos

  LOD dinâmicos; portais para delimitação de áreas fechadas; detecção de colisão baseada em *tiles* para cenários 2D e 2 1/2D; sistema hierárquico de detecção de colisão para ambientes 3D abertos e fechados; e sistema de gerenciamento e ordenação de objetos.

- Multiusuários em rede

  Suporte a conexões TCP e UDP através de sessões ponto a ponto e cliente/servidor; e comunicação por voz.

- Editores de cenários

  Editor de cenários genérico para ambientes 2D e 2 1/2D; estruturação de mapas 2D e 2 1/2D; geração aleatória de mapas; controle de relevo; *morphing* de texturas entre *tiles*; editor de cenários genérico para ambientes 3D; estruturação de mundos 3D; e predefinição de rotas para personagens inteligentes.

- Genéricos

  Coletor de lixo; processador de cálculos matemáticos; sistema de gerenciamento de eventos; sistema de gerenciamento de exceções; sistema de gerenciamento de janelas e interface de usuário portável; controle de *looping*; e temporização.

6.2 Como usar o *FORGE V8*

O *FORGE V8* é composto por três camadas que são responsáveis desde os componentes que tratam problemas de baixo nível até componentes que trabalham em alto nível. Ele abstrai dos usuários diversas das dificuldades intrínsecas ao desenvolvimento de jogos, e possibilita uma maneira de fácil utilização por disponibilizar uma interface de alto nível. Esta interface é composta por classes abstratas que podem ser implementadas por aplicações específicas, e métodos que podem ser utilizados para a execução de tarefas implementadas pelo *framework*.

A versão inicial do *FORGE V8* é caracterizada por ser um *framework* de caixa branca. Isto implica que os seus usuários deverão usar herança para derivar novas variações das classes abstratas disponibilizadas. Para isso, necessita-se conhecer o funcionamento destas classes para poder implementar as aplicações específicas.

O *FORGE V8* disponibiliza um módulo principal, o módulo de aplicação (apresentado na seção de detalhes de projeto), que é responsável pelo gerenciamento das aplicações desenvolvidas com base neste *framework*. Este módulo integra e controla a inicialização, configuração, execução e finalização de todos os outros módulos deste *framework*.

Em versões futuras do *FORGE V8*, após diversas aplicações terem sido implementadas, e uma considerável gama de testes, refinamentos e melhoramentos terem sido realizados a sua estruturação, provavelmente se conseguirá atingir o estágio de maturidade de um *framework* de caixa preta. Assim, não mais necessitando que os usuários conheçam o seu funcionamento interno para poder utilizá-lo.

6.3 Estruturação do *FORGE V8*

Esta seção apresenta a estruturação do *FORGE V8* numa divisão em camadas, disponibilizando os componentes que desempenham as tarefas necessárias aos jogos, e as interações entre estes componentes. As camadas do *framework* são: camada de sistema, camada de gerenciadores e camada de aplicação (ver figura 6.1). Em concordância à linguagem de padrões, os componentes serão apresentados da seguinte maneira: nomenclatura, definição, detalhamento dos problemas de projeto, apresentação das soluções explicitando o uso de padrões de projeto, e diagrama de classes simplificado (se necessário).

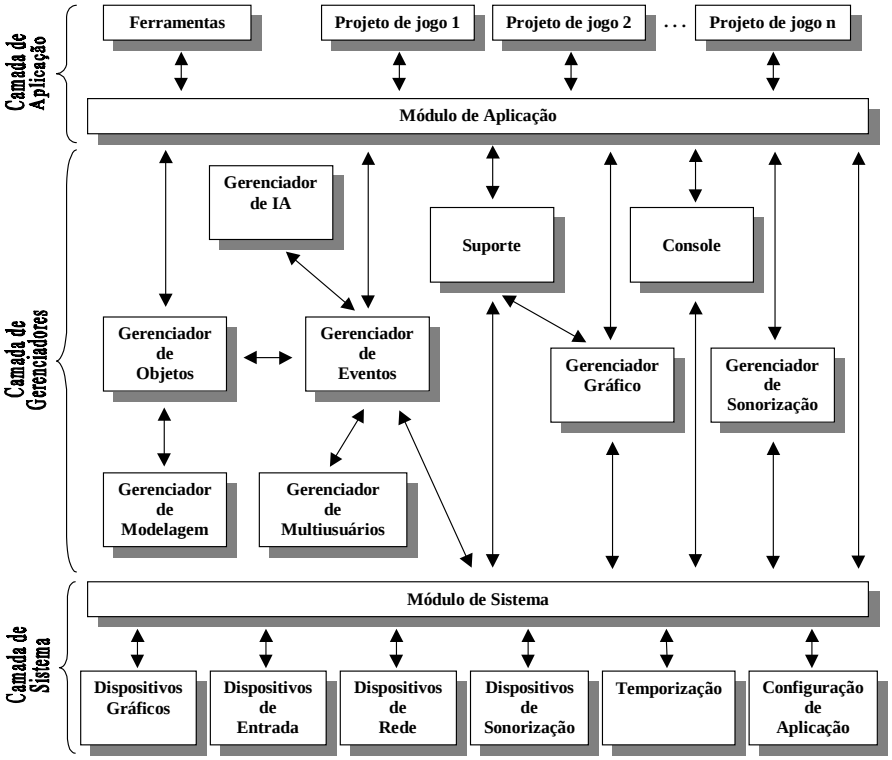


Figura 6.1: Camadas e componentes do *FORGE V8*

6.3.1 Camada de sistema

A camada de sistema é a camada de mais baixo nível, responsável por realizar toda comunicação com o hardware. Essa camada gera uma biblioteca de sistema que tem o objetivo de executar todas as operações dependentes de plataforma para as aplicações, e é a única que necessita de extensões caso se deseje portar o motor para diferentes plataformas. Ela contém diversos subsistemas que provêem funcionalida-

des específicas e gerais. Os subsistemas de funcionalidades específicas são: dispositivos gráficos, dispositivos de entrada, dispositivos de sonorização e dispositivos de rede. Os subsistemas de funcionalidades gerais são: temporização e configuração de aplicação, responsáveis pelo acesso ao sistema operacional. A camada de sistema contém ainda um módulo geral de acesso, chamado módulo de sistema, responsável pela inicialização, configuração e finalização de todos os seus subsistemas.

A camada de sistema deve possuir uma solução de como implementações devem ser combinadas de forma a minimizar as interdependências de código entre os diversos subsistemas, e entre implementações distintas de um mesmo subsistema. Estas implementações variam devido a utilização de diferentes APIs (bibliotecas multimídia), possibilitando as aplicações poderem ser configuradas para utilizar uma API qualquer que esteja implementada. Para isto, somente a porção de código que necessita conhecer as especificidades de uma API específica, que esteja selecionada para uso, deve realizar chamadas a métodos. A implementação desta camada deve ser disponibilizada através de interfaces para bibliotecas de ligação dinâmica, assim facilitando a liberação de novas versões.

6.3.1.1 Módulo de sistema

O modulo de sistema é responsável por realizar a comunicação entre as camadas de níveis acima com os subsistemas da camada de sistemas. Todos os métodos dos subsistemas são acessados apenas através deste módulo, e não de forma direta.

Problemas

1. O módulo de sistema deve ser inicializado e apresentar apenas uma única instância, sendo esta acessível por qualquer parte do *framework* a partir de um ponto conhecido. Este mesmo problema é apresentado pelos vários componentes que realizam gerenciamento em todo o *FORGE V8*;

2. O módulo de sistema deve oferecer todas as funcionalidades de baixo nível do *framework*, servindo de fachada para os diversos subsistemas. A camada de nível acima deve realizar chamadas a métodos dos subsistemas da camada de sistema sem necessitar saber qual das suas diversas implementações está selecionada para ser utilizada.

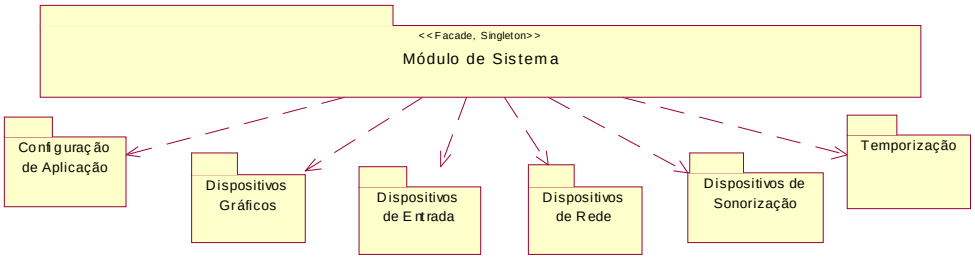
Soluções

1. Pode ser observado que o problema (1) é um caso típico da aplicabilidade do padrão **Singleton** [23]. Este padrão é provavelmente um dos mais familiares

em uso. Ele é necessário em diversas situações cotidianas e é fundamental para a construção de blocos requisitados por outros padrões mais complexos. O padrão *Singleton* garante que uma classe possuirá uma única instância, provendo um único ponto de acesso global a ela. Dessa forma, tal padrão é utilizado a fim de solucionar o respectivo problema;

2. Em relação ao problema (2), o padrão mais adequado para resolvê-lo é **Facade** [23]. Este padrão provê uma interface simplificada para um conjunto de classes relacionadas em um subsistema, elevando o nível de encapsulamento de código, reduzindo a complexidade, e resultando numa forma de mais fácil utilização. Portanto, permite a centralização das funcionalidades oferecidas pela camada de sistema do *FORGE V8*, além de unir as funcionalidades dos subsistemas desta camada.

Diagrama



6.3.1.2 Subsistema de dispositivos gráficos

Este subsistema é de mais alta complexidade da camada de sistema, responsável pelo acesso e configuração do dispositivo de vídeo, leitura e gerenciamento dos modelos gráficos e imagens, e pelo processamento de renderização de cenários. Atualmente, no desenvolvimento de motores para jogos em geral, este subsistema utiliza-se de bibliotecas gráficas padrão (*OpenGL* e *DirectX Graphics* são as mais utilizadas), produzindo um bom resultado devido a utilização das características inerentes aos hardwares aceleradores gráficos. Além disso, apresenta componentes otimizados de rasterização por software para casos onde uma placa aceleradora não esteja presente. Para que mais de uma biblioteca possa ser inserida a esse subsistema, interfaces devem ser projetadas de modo que diferentes implementações possam ser realizadas, disponibilizando aos usuários a possibilidade de escolha ou a realização de uma seleção automática. Dessa forma, provê uma maior compatibilidade e desempenho

com o hardware apresentado. Esta é uma tarefa um tanto trabalhosa, especialmente devido as bibliotecas gráficas não possuírem um mesmo conjunto de características.

Problemas

1. Deve existir uma única instância do subsistema de dispositivos gráficos, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

2. Este subsistema deve remover a interdependência de código por projetar uma interface para as suas chamadas. Neste contexto, deve existir classes base abstratas que contenham como membros dados comuns e funções puramente virtuais. Assim, será possível escrever subclasses para bibliotecas gráficas distintas, cada uma disponibilizando a sua implementação à interface descrita na classe abstrata;

3. Este subsistema deve disponibilizar uma forma de utilização genérica da atividade de renderização, e executar de acordo com sua configuração todos os casos especiais (2D, 2 1/2D e 3D).

Soluções

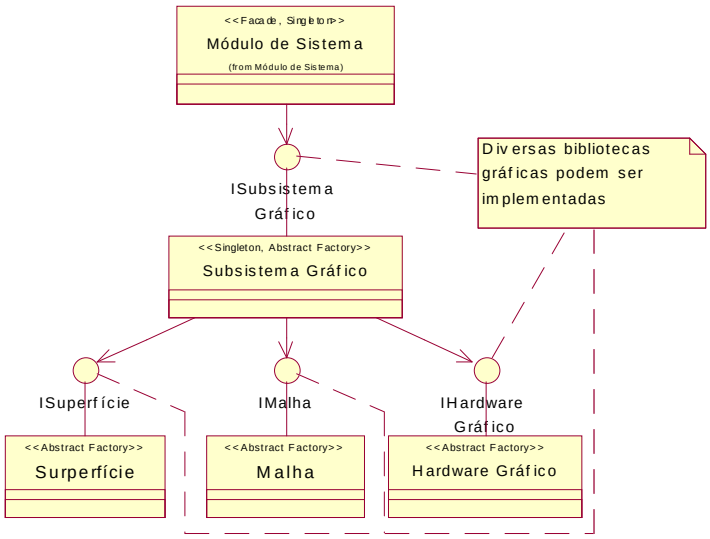
1. Tal como no módulo de sistema, o padrão **Singleton** é utilizado para a solução do problema (1);

2. Como solução para o problema (2), o padrão **Abstract Factory** [23] é utilizado, pois ele provê uma classe abstrata que descreve uma interface para encapsular cada tipo de objeto que deve ser gerado e retornado, de acordo com a API ou subclasses específicas de plataforma que implementam esta interface;

3. O problema (3) é resolvido pelo padrão **Render Delegation** [22] que trata todos os casos de visualização para um conjunto de subclasses de modelos de renderização.

Diagrama

- Este diagrama apresenta as principais interfaces deste subsistema que podem ser implementadas por diversas bibliotecas gráficas. A instanciação de um objeto referente a uma classe concreta, ou seja, de qualquer implementação destas interfaces, é realizada pelas fábricas. Desta forma, abstraindo das chamadas de métodos os detalhes específicos das implementações.



6.3.1.3 Subsistema de dispositivos de entrada

Esse subsistema disponibiliza serviços de teclado, *mouse*, *gamepad*, *joystick* e efeitos *force feedback* (quando existente) permitindo abstração do controle real usado pelo jogo. Ele deve realizar transparentemente checagem de todos os dispositivos instalados e identificar os eventos destes dispositivos, além de apresentar facilidade de configuração para os usuários, possibilitando vários dispositivos executarem uma mesma ação. As bibliotecas mais utilizadas para implementação deste subsistema são *GLUT* (tratamento baseado em mensagens do sistema operacional quando em uso de OpenGL) e *DirectInput*.

Problemas

1. Deve existir uma única instância do subsistema de dispositivos de entrada, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

2. Da mesma forma que acontece com o subsistema de dispositivos gráficos, este subsistema deve remover a interdependência de código por projetar uma interface base para as suas chamadas de métodos em bibliotecas distintas;

3. Este subsistema deve ser uma abstração dos dispositivos de entrada disponíveis, fornecendo funcionalidades para esse processamento no decorrer da execução das aplicações. Esta abstração acontece devido ao importante ser a ocorrência de eventos destes dispositivos;

4. Este subsistema deve tratar os eventos fornecidos pelos dispositivos.

Soluções

1. Tal como nos subsistemas descritos anteriormente, o padrão **Singleton** é utilizado na solução do problema (1);

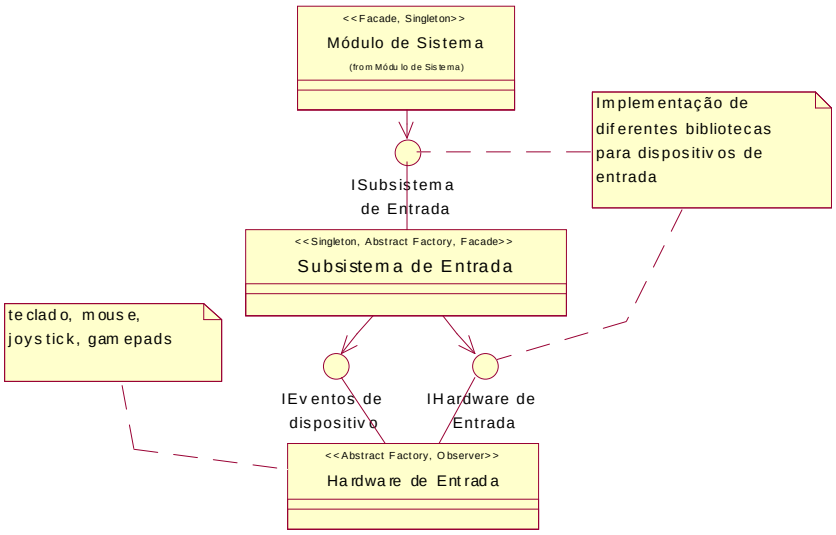
2. Como solução para o problema (2), o padrão **Abstract Factory** é utilizado tal como no subsistema de dispositivos gráficos;

3. A fim de solucionar o problema (3) é projetada uma fachada de acordo com o padrão **Facade**. Essa fachada tem a finalidade de fornecer as funcionalidades de todos os dispositivos de entrada, sem haver a necessidade de camadas de mais alto nível tratar detalhes específicos de cada um;

4. O problema (4) é resolvido pela utilização do padrão **Observer** [23], que é utilizado para "escutar" cada dispositivo. Assim, quando cada dispositivo observado sofrer modificação de alguma maneira, os seus observadores serão notificados de como e quando estas modificações ocorreram.

Diagrama

- Este diagrama apresenta as principais interfaces que podem ser implementadas para este subsistema. A instanciação de um objeto referente a uma classe concreta é realizada pelas fábricas, e as chamadas de métodos são abstraídas dos detalhes específicos das implementações.



6.3.1.4 Subsistema de dispositivos de sonorização

Esse subsistema é responsável pela leitura e gerenciamento de sons e músicas através de *streams* ou *buffers* estáticos, e a reprodução destes pela placa de som. Um requisito bastante importante adicionado a esse subsistema é o suporte a sons 3D. A biblioteca mais utilizada neste modelo de subsistema é *DirectX Audio*, mas *OpenAL* pode ser uma nova tendência.

Problemas

1. Deve existir uma única instância do subsistema de dispositivos de sonorização, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

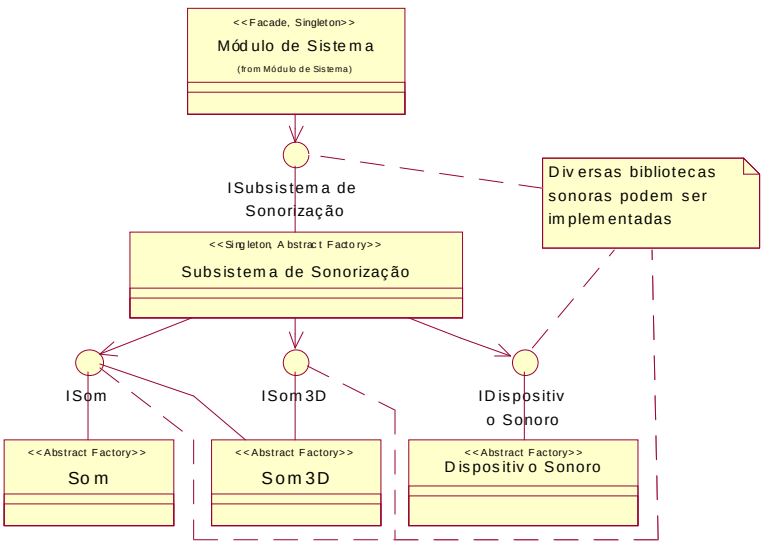
2. Este subsistema deve remover a interdependência de código por projetar uma interface base para as suas chamadas de métodos em bibliotecas distintas.

Soluções

1. Tal como nos subsistemas descritos anteriormente, o padrão **Singleton** é utilizado na solução do problema (1);

2. Como solução para o problema (2), o padrão **Abstract Factory** é utilizado tal como no subsistema de dispositivos gráficos.

Diagrama



- Este diagrama apresenta as principais interfaces que podem ser implementadas pelo subsistema de sonorização. A instanciação de um objeto referente a uma classe concreta é realizada pelas fábricas. As chamadas de métodos abstraem os detalhes específicos das diversas implementações.

6.3.1.5 Subsistema de dispositivos de rede

Este subsistema é responsável por realizar comunicação (conexões, formação de sessões de grupos, e envio e recepção de mensagens) através da rede. Ele pode prover suporte à comunicação por voz, e conexão via Internet e rede local através de modem ou placa de rede. O subsistema de dispositivos de rede pode utilizar bibliotecas como *DirectPlay* ou implementação de TCP e/ou UDP baseada em *sockets*, implementando arquiteturas ponto a ponto e cliente/servidor.

Problemas

1. Deve existir uma única instância do subsistema de dispositivos de rede, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;
2. Este subsistema deve remover a interdependência de código por projetar uma interface base para as suas chamadas de métodos em bibliotecas distintas;
3. Deve haver um tratamento de estado das conexões e sessões estabelecidas de acordo com o protocolo utilizado;
4. Este subsistema deve tratar as mensagens recebidas pelo dispositivo de rede, repassando-as para a camada de mais alto nível, e enviar as mensagens geradas pela camada de mais alto nível para o dispositivo de rede;
5. Este subsistema deve tratar que os clientes sejam perfeitamente sincronizados com o servidor, devido às diversas modificações de estado que podem ocorrer num ambiente.

Soluções

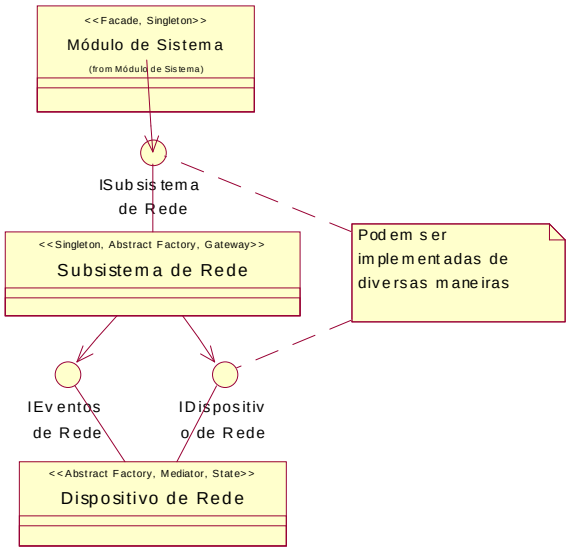
1. O padrão **Singleton** é utilizado na solução do problema (1);
2. Como solução para o problema (2), o padrão **Abstract Factory** é utilizado;
3. Para a solução do problema (3), o padrão **State** [23] é utilizado para o controle de comportamento das conexões de acordo com as mensagens recebidas do dispositivo de rede;

4. A solução para o problema (4) é apresentada pela utilização do padrão **Mediator** [23] que é responsável por concentrar e gerenciar as interações entre subsistemas através de comunicação bidirecional. Este padrão serve tanto para receber informações quanto para enviar respostas;

5. O padrão **Gateway** [22] é utilizado na solução do problema (5). Ele permite que múltiplas modificações no estado de um ambiente sejam tratadas simultaneamente através de um index de modificações para o repositório de objetos do mundo, permitindo uma melhor sincronização por transmitir apenas as informações necessárias (modificadas) do servidor para o cliente.

Diagrama

- Este diagrama apresenta as principais interfaces que podem ser implementadas pelo subsistema de dispositivos de rede.



6.3.1.6 Subsistema de temporização

O subsistema de temporização é responsável por tratar todas as informações ligadas ao tempo necessárias ao *framework*. Ele é utilizado principalmente devido à atualmente os jogos serem na sua maioria em tempo real, o que caracteriza a necessidade de rotinas de gerenciamento. Ele deve calcular o tempo de acordo com o hardware e sistema operacional utilizados no intuito de obter uma melhor aproximação. Este é o subsistema mais simples desta camada.

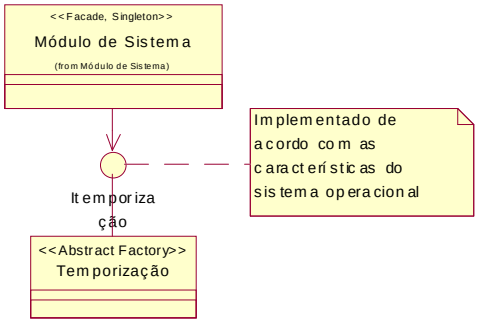
Problemas

1. Deve existir uma única instância do subsistema de temporização, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;
2. Este subsistema deve remover a interdependência de código por projetar uma interface base para as suas chamadas de métodos em sistemas operacionais distintos.

Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);
2. Como solução para o problema (2), o padrão **Abstract Factory** é utilizado.

Diagrama



6.3.1.7 Subsistema de configuração de aplicação

O subsistema de configuração de aplicação é responsável pelos ajustes necessários às aplicações de acordo com o sistema operacional utilizado. Ele provê leitura de arquivos de configuração para as aplicações, acesso a parâmetros de linha de comando, e tratamento de mensagens do sistema operacional. Estes processos são executados através de um laço principal que é responsável pelo controle geral de um jogo. Todos os outros subsistemas devem consultar este quando da inicialização, execução e finalização, proporcionando fácil configuração e seleção de bibliotecas a serem utilizadas. Desenvolver um *framework* com um bom suporte à configuração facilita os testes e permite que os usuários ajustem os parâmetros de acordo com as suas necessidades.

Problemas

1. Deve existir uma única instância do subsistema de configuração de aplicação, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

2. Este subsistema deve tratar os diversos sistemas de arquivos existentes, apresentando um sistema de arquivos virtual único para o *framework*;

3. Este subsistema deve suportar múltiplos sistemas de janelas de acordo com o sistema operacional;

4. Deve haver um tratamento de estado das aplicações em relação ao sistema operacional, no que se refere as suas condições de execução e mensagens de sistema a serem tratadas;

5. Os jogos necessitam que objetos do mundo possam ser criados e removidos dinamicamente de acordo com o decorrer de uma partida, provendo que cada objeto possa ser executado pela chamada do método virtual *update*, uma vez a cada *frame* do jogo.

Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);

2. Para a solução do problema (2), interfaces devem ser implementadas baseadas nas características de cada sistema de arquivos, através do uso do padrão **Abstract Factory** abstraindo a distinção deles da camada de nível superior;

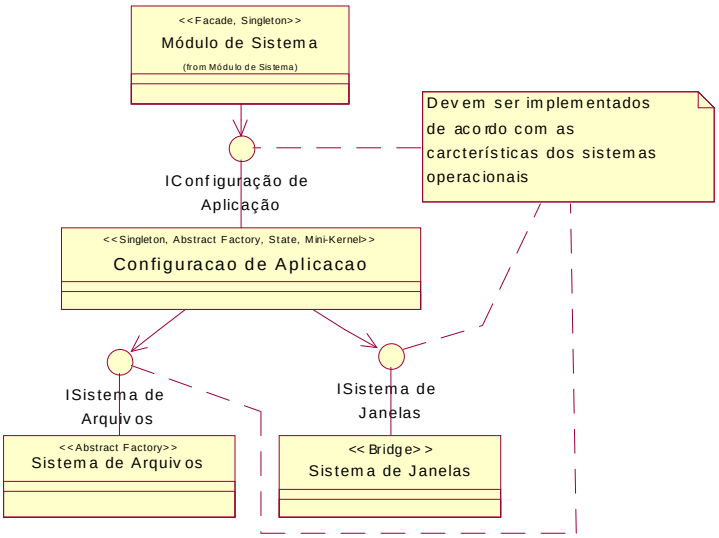
3. O problema (3) é solucionado através do uso do padrão **Bridge** [23] que utiliza abstração do sistema de janelas e implementações para plataformas específicas, através de hierarquias de classes separadas, disponibilizando essa abstração para ser utilizada pela camada de mais alto nível;

4. Para a solução do problema (4), o padrão **State** é utilizado para que as aplicações alterem seu comportamento de acordo com as mensagens recebidas do sistema operacional que afetem seu estado interno;

5. O problema (5) é solucionado por uma super classe de todos os controladores de objetos criada através do padrão **Mini-kernel** [22]. A cada *frame* do jogo, o *mini-kernel* possibilita a cada objeto controlado a chamada ao método virtual *update* pelo laço principal da aplicação, caracterizando um sistema multitarefas cooperativo.

Diagrama

- Este diagrama apresenta as principais interfaces que podem ser implementadas pelo subsistema de configuração de aplicação.



6.3.2 Camada de gerenciadores

A camada de gerenciadores é uma camada de nível médio, responsável pelo controle da execução da aplicação, tratando os processos "disparados" pela lógica do jogo. Ela contém diversos módulos que provêem funcionalidades específicas e gerais, do mesmo modo que a camada de sistema. Os módulos de funcionalidade específica são: gerenciador gráfico, gerenciador sonoro, gerenciador de objetos, gerenciador de eventos, gerenciador de IA, gerenciador de modelagem física e gerenciador de multiusuários. Os módulos de funcionalidades gerais são: console e suporte, que são utilizados pelos outros diversos módulos. Os módulos desta camada devem poder ser utilizados de forma independente por aplicações correlacionadas que necessitem de apenas um ou poucos gerenciadores.

6.3.2.1 Console

O console é o módulo do *framework* responsável por realizar as modificações de configuração necessárias ao jogo e ao *framework* sem a necessidade de reiniciá-lo, como também é uma maneira eficiente para apresentar a saída de informações de *debug* no estágio de desenvolvimento. Além disso, realiza o tratamento de erros em tempo de execução sem haver a necessidade de sair da aplicação, informando em uma

mensagem explicativa o erro ocorrido. Um console caracteriza o desenvolvimento de um mecanismo de verificação de exceções bastante robusto. Se não for de interesse que ele seja apresentado ao usuário final, facilmente poderá ser tornado invisível.

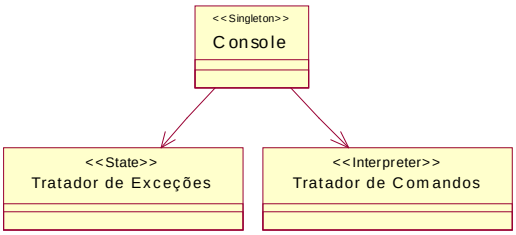
Problemas

1. Deve existir uma única instância do console, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;
2. Deve haver um tratamento de tipos de erros, e informações de saída;
3. Deve haver uma gramática para verificação de sintaxe dos comandos do console. Dada uma linguagem, definir a representação desta gramática.

Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);
2. O problema (2) é solucionado pelo padrão **State** que verifica o tipo do problema ocorrido para ser tratado;
3. O padrão utilizado para resolver o problema (3) é o **Interpreter** [23] que constrói um interpretador para as sentenças existentes na gramática.

Diagrama



6.3.2.2 Suporte

O módulo de suporte é utilizado pelos muitos outros módulos do *framework* e tem a função de oferecer serviços comuns e utilitários. Esse módulo inclui todas as rotinas matemáticas necessárias (vetores, planos, matrizes, transformações, etc.) para os processamentos gráficos, gerenciadores de memória, leitores de arquivos multimídia, tratamento de componentes de interface de usuário, ferramentas para compactação

e descompactação de arquivos multimídia proprietários, controle de *parser* para leitores de arquivos de configuração, *plug-ins* para modelos 3D, entre outros.

A matemática é o coração de todo jogo realístico. Ela dita como projetar um mundo tridimensional em uma tela bidimensional, descreve o comportamento de toda a virtualidade vista do ambiente, da trajetória de um objeto, até a reflexão de luzes e projeção de sombras. A utilização de cálculos matemáticos de forma otimizada é fundamental para o desenvolvimento de jogos realísticos em tempo real.

Problemas

1. Este subsistema deve oferecer uma estrutura hierárquica de componentes de interface de usuário (menu, botões, caixas de diálogo, caixas de texto, etc.) comum e portável que possa ser adaptada as necessidades das aplicações. A funcionalidade por trás desta estrutura é possibilitar a configuração e apresentação de objetos da aplicação, provendo um mecanismo uniforme para acesso;

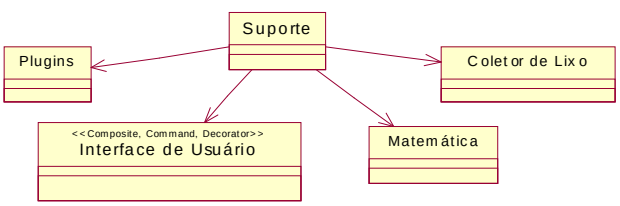
2. Este subsistema deve possibilitar que a aparência da interface gráfica de usuário (estilo de menus, botões, caixas de diálogo, etc.) possa ser configurada de acordo com as necessidades das aplicações.

Soluções

1. Para a solução do problema (1), dois padrões são utilizados, o **Composite** [23] que apresenta a essência da composição recursiva em orientação a objetos. Este padrão possibilita o tratamento de objetos individualmente e composições de objetos uniformemente. E o padrão **Command** [23] que encapsula comandos que podem ser passados como objetos, construindo uma biblioteca de interface de usuário que pode ser adaptada de acordo com as necessidades das aplicações;

2. O problema (2) é solucionado pelo uso do padrão **Decorator** [23] que é utilizado para definir a aparência de um ambiente através de um estilo de interface.

Diagrama



6.3.2.3 Gerenciador gráfico

Este módulo é responsável por controlar e processar os componentes utilizados na construção de cenários 2D, 2 1/2D e 3D, enviando os resultados para a renderização pelo subsistema de dispositivos gráficos da camada de sistema.

Os componentes que compõem o gerenciador gráfico são: visibilidade, detecção de colisão e resposta, animação, câmera, geometria estática, geometria dinâmica, sistemas de partículas, *billboarding*, malhas, iluminação, neblina, espelhamento, texturização, etc. [16]. Cada um destes componentes possui formas de configuração que facilmente permitem modificações para adaptação às características desejadas.

Em relação a composição das malhas, o processamento deste gerenciador deve gerar ao máximo listas de triângulos *fans* e *strips* para serem enviadas de uma só vez ao pipeline gráfico (não um triângulo por vez) [16]. Desta forma, diversos polígonos podem ser passados juntamente com a mesma iluminação, neblina, e código de preenchimento, possibilitando um melhor desempenho pois os efeitos atribuídos a um polígono poderão ser realizados simplesmente pela modificação do seu material e/ou textura.

Problemas

1. Deve existir uma única instância do gerenciador gráfico, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

2. Este subsistema deve possuir um repositório com busca otimizada de instâncias de texturas, *sprites* e malhas lidos no subsistema gráfico da camada de sistema;

3. Um mesmo objeto gráfico, como uma textura, deve poder ser compartilhado por mais de um *sprite* ou malha, sem a necessidade de ter mais de uma instância;

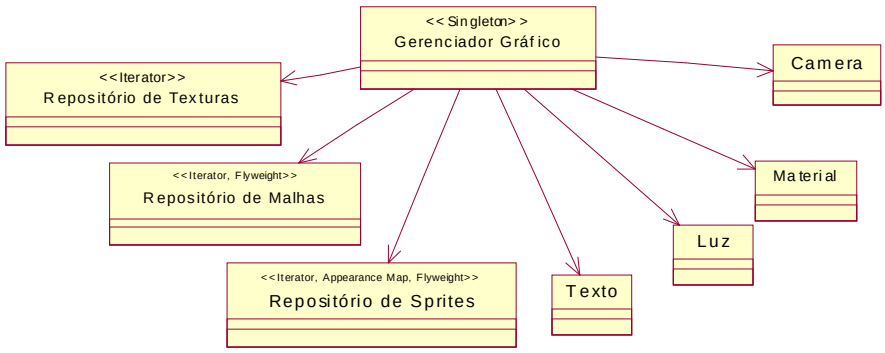
Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);

2. Para a solução do problema (2), o padrão **Iterator** [23] é utilizado para prover uma maneira rápida de acesso a elementos de uma seqüência de objetos agregados;

3. A solução do problema (3) é obtida pelo padrão **Flyweight** [23] que permite múltiplas instâncias de uma classe se referirem para um componente comum;

Diagrama



6.3.2.4 Gerenciador sonoro

O gerenciador sonoro é um módulo que provê uma interface de alto nível responsável por controlar o processo de execução de sons e músicas utilizadas na composição de cenários mais realistas, enviando os resultados para a reprodução pelo subsistema de dispositivos de sonorização da camada de sistema. Permite a mixagem de sons e músicas, e controla a execução de efeitos, volume e posicionamento (sons 3D).

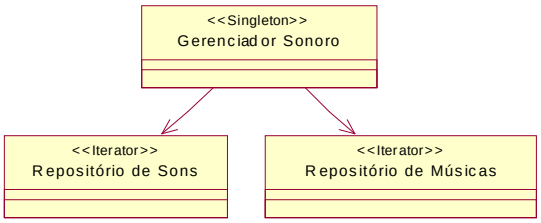
Problemas

1. Deve existir uma única instância do gerenciador sonoro, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;
2. Este subsistema deve possuir um repositório com busca otimizada de instâncias de sons e músicas, lidos no subsistema de sonorização da camada de sistema.

Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);
2. Para a solução do problema (2), o padrão **Iterator** é utilizado para prover uma maneira rápida de acesso a elementos de uma seqüência de sons agregados;

Diagrama



6.3.2.5 Gerenciador de objetos

O gerenciador de objetos é responsável pelo controle do ambiente (mundo) e entidades (personagens) de um jogo. Estas entidades devem ser armazenadas em estruturas de dados e controladas de acordo com o ciclo de vida de cada uma delas. Em geral, um jogo contém várias instâncias de diferentes entidades e todas as informações necessárias para que sejam gerenciadas. Esses dados envolvem, posicionamento, velocidade, dimensão, etc. Além disso, devem possuir métodos necessários para detecção de colisão com outros objetos.

Em geral, um editor de cenário se associa a este gerenciador para descrever o estado do mundo em cada nível do jogo. Tal associação resulta em delegações de tarefas para este gerenciador a fim de que ele inicie os objetos do mundo de acordo com a especificação inicial do cenário.

Mundos podem ser projetados de diversas maneiras: em jogos 3D, o mundo se refere a um ambiente 3D construído por malhas de polígonos; em jogos 2D, ambientes 2D são representados por vetores bidimensionais; e em jogos isométricos, ambientes 2D são rotacionados em 45^0, apresentando uma visão em perspectiva. O mundo é uma etapa de desenvolvimento bastante importante dos jogos de computador, pois dele pode se extrair a "mágica" do sucesso.

Problemas

1. Deve existir uma única instância do gerenciador de objetos, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

2. Os controladores de objetos deste gerenciador são freqüentemente máquinas de estado que realizam interações de uma maneira muito específica. É comum para esta interação, que objetos modifiquem a sua aparência, especialmente em controladores de animação. Desde que a arte possa mudar freqüentemente é interessante separar o estado do objeto da sua aparência;

3. Os jogos consistem de uma variedade de objetos que interagem. Em alguns casos, existem muitos tipos de objetos, em outros, poucos. No entanto, em todos os casos, cada objeto obtém seu estado de acordo com o progresso do jogo. As regras do jogo definem a transição desses estados;

4. Muitos jogos precisam tratar o estado de muitos objetos simultaneamente. Objetos agrupados em uma lista simplificam diversos dos processos necessários;

5. Este subsistema deve possuir busca otimizada de instâncias de objetos no seu repositório.

Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);

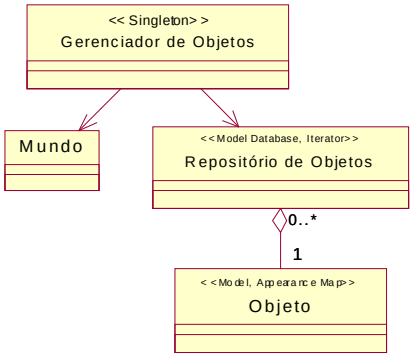
2. O padrão **Appearance Map** [22] resolve o problema (2) por verificar constantes de controle, e baseado nelas alterar a aparência do objeto. Tipicamente, isto é feito por uma tabela composta pela tradução de estado para o *frame* de aparência adequada, minimizando o impacto das modificações exigidos por estes controladores;

3. O padrão **Model** [22] é utilizado para resolver o problema (3) devido a tratar o estado de um objeto em relação ao mundo do jogo;

4. A solução do problema (4) se dá pela utilização do padrão **Model Database** [22] que agrega os objetos em um banco de dados de objetos (repositório);

5. Para a solução do problema (5), o padrão **Iterator** é utilizado para prover uma maneira rápida de acesso aos elementos do repositório.

Diagrama



6.3.2.6 Gerenciador de eventos

O gerenciador de eventos é responsável por rotear todos os eventos relacionados às aplicações e ao *framework*: eventos do sistema operacional, eventos de janela, eventos dos dispositivos de entrada, eventos do motor de inferência (IA), eventos do processamento multiusuário e eventos do controle de objetos do ambiente do jogo. Ele tem como função simplificar o manuseamento de eventos evitando que vários objetos tratem eventos diferentes separadamente, tornando assim a aplicação pouco flexível.

Para evitar o acoplamento dentro deste gerenciador, deve-se utilizar manipuladores de eventos (*handlers*). Alguns componentes do *framework*, tanto aqueles criados durante a execução do jogo quanto elementos que existem desde o inicio da aplicação, devem ser notificados da ocorrência de eventos que os interessem. Os componentes devem se subscrever para poder passar a observar os eventos de interesse.

Problemas

1. O gerenciador de eventos deve rotear todos os eventos possíveis necessários à aplicação: eventos do sistema operacional, eventos de janela, eventos dos dispositivos de entrada, eventos do motor de inferência, e eventos de processamento multiusuário;

2. Este gerenciador tem como função simplificar a manipulação de eventos evitando que vários objetos tratem eventos diferentes separadamente, tornando assim a aplicação pouco flexível;

3. Deve-se utilizar manipuladores de eventos (*handlers*) com o intuito de evitar o acoplamento dentro deste gerenciador;

4. Alguns componentes do *framework*, tanto aqueles criados durante a execução do jogo quanto elementos que existem desde o inicio da aplicação, devem ser notificados da ocorrência de eventos que os interessam.

5. Os componentes devem se subscrever para poder passar a observar os eventos de interesse.

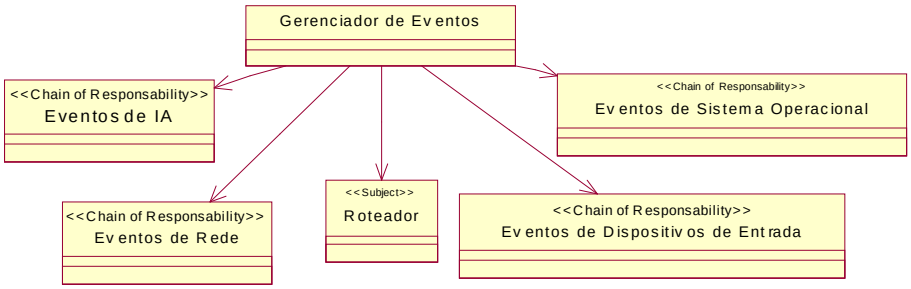
Soluções

1. Os problemas descrito nos itens (1) a (3) podem ser resolvidos com a utilização do padrão **Chain of Responsability** [23]. Este padrão trata da construção

de uma estrutura hierárquica que, neste caso, repassa a responsabilidade do tratamento de um determinado evento para o nível mais baixo da hierarquia até chegar a ser tratado ou por um dispositivo dos citados acima;

2. Em relação aos dois últimos problemas, pode-se notar uma dependência entre alguns componentes do *framework* e este gerenciador. Quando os dados armazenados no gerenciador de eventos são modificados devido a um novo evento, todos os seus observadores devem ser notificados. O padrão que resolve esse problema naturalmente é o **Observer**. No caso do *FORGE V8*, alguns subsistemas e gerenciadores correspondem ao padrão **Subject** [23] e os observadores deles ao padrão **Observer**.

Diagrama



6.3.2.7 Gerenciador de IA

Este gerenciador é responsável pelo comportamento de entidades autônomas (NPCs). Para isso, pode utilizar de diversas tecnologias (redes neurais, algoritmos genéticos, lógica *fuzzy*, motor de inferência, máquina de estados finitos (FSM)) que podem receber regras baseadas em *scripts* preestabelecidos pelas aplicações. Executa as ações de acordo com o estado atual da aplicação e as condições apresentadas nos *scripts*.

Este gerenciador contribui enormemente com o desenvolvimento de animações de computador realísticas em mídia interativa, proporcionando a resolução de diversos problemas complexos encontrados nestes tipos de aplicação [61].

O comportamento deste gerenciador é melhor compreendido através de uma divisão em etapas que tratam decisões de alto nível como planejamento e gerenciamento de tarefas, até o processamento de mais baixo nível relacionado a movimentação de objetos que pode ser executada em conjunto com a modelagem física (caso seja necessária).

Problemas

1. Deve existir uma única instância do gerenciador de IA, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

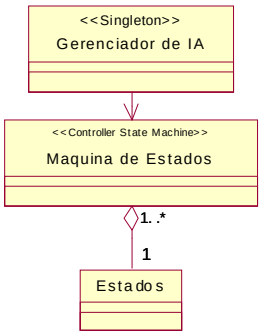
2. NPCs são máquinas de estado que envolvem transições como circunstâncias de progresso em resposta a eventos. A animação é um exemplo onde ambos tempo e informações de ações dos usuários afetam as transições de estado.

Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);

2. O padrão **Controller State Machine** [22] é utilizado para resolver o problema (2) através de uma subclasse controladora que contém a lista de todas as variáveis de estado.

Diagrama



6.3.2.8 Gerenciador de modelagem física

Este gerenciador é utilizado para modelagem dinâmica e simulação. Ele deve apresentar o foco para entidades dinâmicas, suportar cálculos de diversas propriedades físicas, e ter o objetivo de resolver equações de movimentação para estas entidades.

Os modelos físicos são baseados em física Newtoniana, que trabalha razoavelmente bem em movimentação de objetos que possuem limites racionais de tamanho e massa. Diferentes tipos de entidades devem ter diferentes equações de movimentação, as quais são usadas para alterar o posicionamento do objeto que representa a entidade a ser renderizada.

Problemas

1. Deve existir uma única instância do gerenciador de modelagem física, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);

6.3.2.9 Gerenciador de multiusuários

Este gerenciador tem o objetivo de permitir interação entre os vários usuários de uma aplicação. Controla o estado dos usuários referentes as sessões estabelecidas pelo subsistema de dispositivos de rede da camada de sistema.

As mensagens a serem enviadas para o agrupamento de uma sessão devem ser apenas referentes as informações essenciais entre os jogadores, fazendo um uso eficiente de ambos recursos de rede e CPU. Em alguns casos, o serviço de rede deve prover suporte a uma eficiente maneira de distribuição de dados agrupados - *multicast*. Em outros casos, jogos podem ser limitados para *broadcast* ou conexões ponto-a-ponto.

Agrupamentos podem dar apoio aos jogos em rede no sentido de aumentar a sua escala para a inclusão de grandes números de jogadores. A quantidade de jogadores pode variar de acordo com a necessidade de cada jogo.

Para facilitar mais ainda a integração entre os jogadores, já existem tecnologias que permitem comunicação por voz através da rede, como *Roger Wilco* [64] e *DirectPlay* [13]. Além disso, existem acordos com provedores de Internet para implantação do *PowerPlay* [59], um novo conjunto de padrões e protocolos que têm o objetivo de prover otimização do uso da Internet para os jogos, fazendo da grande rede uma melhor plataforma de entretenimento.

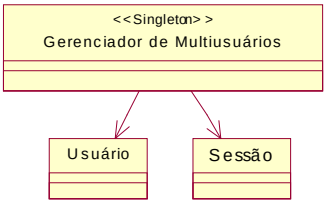
Problemas

1. Deve existir uma única instância do gerenciador de multiusuários, sendo ela acessível por qualquer parte do *framework* a partir de um ponto conhecido;

Soluções

1. O padrão **Singleton** é utilizado na solução do problema (1);

Diagrama



6.3.3 Camada de aplicação

A camada de aplicação é a camada de mais alto nível do *framework*, responsável pela interligação entre este e os projetos de aplicações e ferramentas. O ideal no desenvolvimento de um *framework* reusável é que as aplicações nunca tenham seus componentes inseridos a este *framework*. Portanto, esta camada contém o módulo de aplicação para atingir essa separação.

6.3.3.1 Módulo de aplicação

O módulo de aplicação é uma fachada de todo o *framework* que expõe toda a sua funcionalidade em alto nível para o desenvolvimento de jogos e aplicações multimídia. Representa o caminho de comunicação e acesso de projetos com todo o *framework*. Um importante objetivo deste módulo é abstrair ao máximo o processamento do *framework* e consequentemente facilitar o desenvolvimento de aplicações. Além de que, se uma determinada aplicação deve ser desenvolvida baseada neste *framework*, então esta aplicação deve implementar as interfaces e padrões oferecidos pelo módulo de aplicação, caracterizando uma padronização à aplicação. Para cada gerenciador do *framework* que tem uma propriedade dinâmica, o *framework*/módulo de aplicação provê uma interface para modificá-la.

Problemas

1. O gerenciador de aplicação deve possuir uma estrutura genérica do laço principal de um jogo (leitura e execução dos níveis do jogo, atualização dos eventos de entrada, atualização de estado e posicionamento dos objetos, e apresentação na tela do usuário).

Soluções

1. Para resolver o problema (1), o padrão **Template Method** [23] é utilizado. O laço principal de um jogo executa diversas funções, sendo cada uma delas representada por um **Template Method** que deve ser implementado de acordo com o jogo em questão.

6.3.3.2 Ferramentas

Durante o desenvolvimento de um jogo, diversos tipos de informações são necessárias. Por exemplo, não é fácil escrever arquivos que definam modelos de objetos 3D. Para isto, é necessário utilizar programas de modelagem e texturização gráfica. Certamente, muitas ferramentas são estritamente necessárias e algumas delas devem ser construídas num formato proprietário, como editores de cenários (níveis) e editores de personagens, para realizar as tarefas específicas do projeto. Neste caso, precisa-se desenvolver conversores ou *plug-ins* do formato utilizado pelas ferramentas de terceiros para o formato utilizado pelas ferramentas proprietárias. A maior certeza é que provavelmente se faz necessário tão quanto ou mais código de ferramentas que o próprio código do jogo.

A principal ferramenta a ser contruída para um jogo é o editor de cenários. Este editor é responsável por projetar ambientes ou mapas de mundos, adicionar objetos estáticos em suas posições fixas, adicionar NPCs em posições iniciais, editar propriedades dos objetos, determinar trajetórias dos NPCs, etc, para serem lidos em tempo de execução pelo jogo de modo a serem reconhecidas todas as propriedades editadas. Além disso, podem existir processamentos mais elaborados como a geração de mapas aleatórios.

A arquitetura geral do *FORGE V8* foi estruturada baseando-se nas avaliações dos motores de código aberto estudados, e na definição dos processos que se mostravam necessários a ela. Das avaliações dos motores foram extraídas algumas boas idéias, adaptando-as a realidade do nosso *framework*. Destas idéias extraídas, as principais foram: a utilização de uma camada de sistema para garantir a portabilidade e a implementação de novas versões de maneira facilitada (motor *Crystal Space*); a utilização de uma interface de usuário proprietária com possibilidade de adaptação à aplicação (motor *Crystal Space*); e a utilização de um sistema de controle geral de mensagens adaptável (motor *Golgotha*).

6.4 Conclusão do capítulo

Criar ferramentas reutilizáveis é uma das etapas fundamentais no desenvolvimento de jogos de computador, que necessita de um enorme esforço. O *FORGE V8* foi idealizado com o intuito de constituir uma interessante ferramenta, com a finalidade de apresentar uma maneira de como estruturar a arquitetura de jogos de computador, e tratar uma série de características gerais necessárias a estas aplicações permitindo que elas sejam desenvolvidas sem grandes dificuldades. Isto acontece devido a sua alta flexibilidade, facilidade de configuração, e a libertação que é atribuída aos usuários das obrigações de escrever código a baixo nível. A flexibilidade foi conseguida pelo uso de padrões de projeto que permitiram o encapsulamento de variações em classes, provendo implementações individuais de componentes que expõem uma mesma interface.

Padrões são uma boa maneira para descrever *frameworks* porque usualmente, os seus usuários estão primeiramente interessados em saber em alto nível como ele trabalha, para depois conhecê-los mais a fundo no caso de resolverem o problema em questão. A estrutura interna de um conjunto de padrões minimiza o montante que tem de ser lido para resolver um problema, e também provê a localização correta para inserir as informações do projeto. Quanto mais os padrões forem eficientes para descrever o uso de *frameworks*, mais eles apresentarão as necessidades dos usuários.

Capítulo 7

Implementação e Resultados

Este capítulo está dividido em duas etapas relacionadas a implementação e utilização do *FORGE V8*. Na primeira delas são apresentadas algumas decisões de projeto: a definição da linguagem de programação mais adequada, a escolha da biblioteca multimídia a ser incorporada inicialmente pelo *framework*, e a definição de um subconjunto de componentes a serem implementados. Na segunda etapa, é apresentado um relato sobre um estudo de caso realizado com base neste *framework*, e os resultados obtidos pela sua utilização.

7.1 Decisões de projeto

No desenvolvimento de qualquer projeto, algumas decisões críticas devem ser tomadas de acordo com o seu grau de complexidade, ambiente e ferramentas que serão utilizados, e o tempo disponibilizado para sua conclusão. Para o *FORGE V8*, abaixo estão discriminadas as decisões que influenciaram na sua implementação, referentes à linguagens de programação, bibliotecas multimídia, documentação de código, e implementação de componentes.

7.1.1 Linguagens de programação

As linguagens de programação seriamente consideradas para a implementação do *FORGE V8* foram Java e C++, devido a estas linguagens apresentarem as características intrínsecas ao paradigma de orientação a objetos, como modularidade, reusabilidade e robustez. Além disso, elas são compatíveis com diversas bibliotecas multimídia, são ideais para a implementação de padrões de projeto, e apresentam fácil portabilidade para inúmeras plataformas [29][65]. No entanto, dentre estas linguagens, apenas uma deveria ser escolhida, então foram verificados os prós e contras

de cada uma delas relacionados ao desenvolvimento de jogos.

Java é uma linguagem orientada a objetos, adicionada de diversas características de linguagens de alto nível, abstraindo dos programadores inúmeras complicações. C++ é uma linguagem orientada a objetos que também abstrai diversos problemas dos programadores, e ainda apresenta formas de programação a mais baixo nível, o que Java não possibilita. Para o desenvolvimento de jogos, essa característica de C++ se enquadra diretamente entre as funcionalidades de Java e C, sendo bastante interessante devido ao alto processamento envolvido nos jogos e a enorme necessidade de otimização.

Até poucos anos atrás, a idéia de usar C++ no desenvolvimento de jogos era extremamente desconsiderada por uma enorme gama de programadores desta área [65]. Devido a necessidade de alta velocidade, as linguagens C e *Assembly* eram escolhidas para o desenvolvimento destas aplicações. No entanto, com a aceitação universal das placas aceleradoras 3D de alto desempenho, e compiladores com alto grau de otimização, atualmente acredita-se que os benefícios de utilizar arquiteturas orientadas a objetos excede a pequena queda no desempenho que ela causa, e a elegância apresentada em projetos e implementações supera este problema.

Pelo motivo da programação para jogos necessitar de alto desempenho e um elevado grau de otimização, C e C++ são as linguagens padrão da indústria de jogos na atualidade [65]. Provavelmente, C++ será o padrão ainda por muito tempo para tal desenvolvimento. Então, devido a estas tendências, e experiências passadas de desempenho não satisfatório obtidas em projetos que utilizavam Java para jogos em 3D (projetos de dissertação de mestrado anteriores mencionados no capítulo 1), decidimos por utilizar C++ no desenvolvimento do *FORGE V8*.

7.1.2 Bibliotecas multimídia

A respeito das bibliotecas multimídia, duas foram seriamente consideradas para o desenvolvimento do *FORGE V8*: *OpenGL* e *DirectX*. *OpenGL* é o padrão para computação gráfica nas áreas de simulação e pesquisas acadêmicas. Ela é simples, elegante e veloz, mas requer cuidado com detalhes críticos na implementação. De fato, para se ter um *driver* de fácil utilização e totalmente compatível, cada fabricante de placas aceleradoras 3D devem implementá-los provendo suporte a todo o conjunto de características da biblioteca. Estes *drivers* são extremamente complicados de se implementar corretamente, e a performance do *hardware* esta totalmente vinculada a sua qualidade. *DirectX*, da *Microsoft*, tem a grande vantagem de ser capaz de se adaptar rapidamente as características de novos hardwares, pois, diferentemente de

OpenGL, suas modificações não precisam ser deliberadas por comitês. Além disso, *DirectX* implementa diversas funcionalidades com acesso direto a hardware (controle de dispositivos de entrada, sonorização, renderização de vídeo por *streaming* e controle de sessões multiusuário em rede) que *OpenGL* não dispõe. Por esses motivos, *DirectX* domina a maior fatia do mercado (90% dos jogos de computador são produzidos para esta biblioteca [28]).

O *FORGE V8* foi projetado para funcionar baseado em qualquer uma destas bibliotecas, de acordo com as implementações das interfaces da camada de sistema, como também baseado em qualquer outra biblioteca que seja necessária. Inicialmente, pensamos em implementar as interfaces para *DirectX* em ambiente *Windows*, e *OpenGL* para as outras plataformas. Mas, devido ao curto período disponível, chegamos a conclusão que seria possível implementar apenas uma abordagem. Logo, decidimos por utilizar *DirectX* (versão 8.0) em ambiente *Windows* para a primeira versão do *FORGE V8*. Em versões futuras, poderão ser realizadas contribuições para com outras plataformas.

7.1.3 Implementação de componentes

A implementação dos componentes do *FORGE V8* foi um processo custoso que utilizou de um modelo para organização e documentação de código, apresentado no apêndice B desta dissertação, com o objetivo de gerar código legível, de fácil manutenção e padronizado.

Por outro lado, apesar da importância de termos um protótipo do jogo *Canyon* em condições de ser utilizado e avaliado, e assim, podermos obter resultados, isso não foi possível devido a alta complexidade envolvida em tal desenvolvimento, e o curto período disponibilizado para um mestrado. Esta complexidade do jogo *Canyon* é caracterizada principalmente pelos seus cenários serem em 3D, cujo desenvolvimento é muito trabalhoso e demanda o conhecimento de especialistas. Além disso, a dificuldade foi ainda maior, devido ao fato de estarmos desenvolvendo um *framework* para uso geral, e não especificamente para o *Canyon*. Então, devido a todas as complicações envolvidas com cenários 3D, decidimos num primeiro momento, implementar apenas um subconjunto de componentes do *FORGE V8* com o objetivo de conseguirmos obter na prática um resultado concreto. Este subconjunto deveria fornecer os serviços essenciais para construção e execução de um jogo 2D. Assim, tornando possível a possibilidade de obtermos um resultado real.

Este subconjunto de componentes foi selecionado através de uma ordem de prioridade. Inicialmente, deveria existir uma aplicação básica, construída para compor

o laço principal do *framework* (explicado no capítulo 3). A partir daí, a criação dos gerenciadores de interface gráfica e sonorização baseados em *DirectX*, os gerenciadores de objetos e eventos para a utilização de personagens em ambiente 2D, e se possível, um editor de cenários 2D simplificado. Neste contexto, foram implementadas características específicas do sistema operacional *Windows* no que se refere ao tratamento de mensagens de sistema, temporização, sistema de gerenciamento de janelas, entre outros.

Essa primeira fase de implementação do *FORGE V8* durou cerca de 4 meses com o esforço de 3 engenheiros de software, 1 em tempo integral e 2 em tempo parcial, e gerou mais que 18000 linhas de código.

O maior tempo gasto com a implementação, foi com a incorporação de *DirectX* no *FORGE V8*. *DirectX* apresenta o seu funcionamento baseado num modelo de programação orientado a objetos conhecido como *Component Object Model* (COM). Portanto, foi necessário conhecer os princípios básicos e técnicas de programação em COM para obter um bom resultado na utilização de *DirectX*. A programação em COM pode ser realizada de duas maneiras distintas:

- Utilizando os objetos COM já existentes. Esta é maneira mais fácil, pois eles devem ser utilizados como objetos C++.

- Implementando um objeto COM proprietário. Esta pode ser uma tarefa complicada e extensa. A programação por este caminho atribui a COM uma má reputação devido a sua alta complexidade. É desta forma que *drivers* são implementados.

Na sua maioria, as aplicações que utilizam *DirectX* necessitam apenas dos objetos COM providos por *DirectX*, como é o caso do *FORGE V8*. Neste caso, necessita-se apenas saber como utilizar estes objetos.

Objetos COM são basicamente caixas pretas que podem ser usadas por aplicações para executar uma ou mais tarefas. Eles são comumente implementados por bibliotecas de ligação dinâmicas (DLLs). As DLLs provêem métodos implementados que as aplicações podem fazer chamadas para executar determinadas tarefas.

Na programação em COM, as interfaces podem sofrer alterações na sua nomenclatura devido a mudanças de versões, de modo que versões anteriores das interfaces continuam sendo disponibilizadas para garantir o funcionando de aplicações já implementadas. Isto acontece consequentemente em *DirectX*. Portanto, o *FORGE V8* deve tratar versões diferentes de *DirectX* de forma distinta, cabendo ao usuário se-

lecionar a versão desejada. A figura 7.1 apresenta um diagrama com os recursos necessários a um jogo implementado com base no *FORGE V8*.

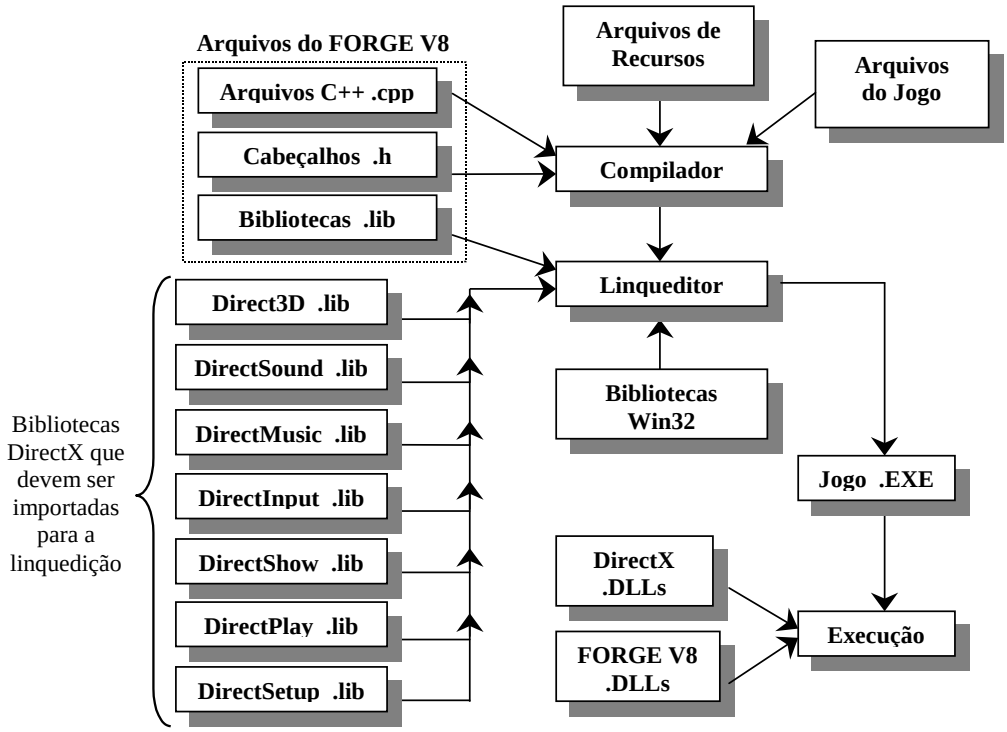


Figura 7.1: Recursos necessários para uma aplicação com base no *FORGE V8*

Todo o projeto e implementação do *FORGE V8* teve o objetivo de facilitar ao máximo o trabalho a ser desempenhado pelos seus usuários, através da definição de classes e métodos de controle que realizassem tarefas automaticamente sem ter a necessidade de intervenção externa. Claro que muitas destas tarefas são configuráveis, de modo que os usuários possam atingir as suas necessidades, mas esta configuração acontece apenas na inicialização da aplicação.

O *FORGE V8* disponibiliza uma classe base, denominada *CApplication*, responsável pelo seu interfaceamento com uma aplicação. Este interfaceamento corresponde ao módulo de nível mais alto da camada de aplicação deste *framework* (explicado no capítulo 6). Esta classe oferece a capacidade de controle de todos os gerenciadores do *framework*, e sua utilização por uma aplicação é realizada através de derivação (herança), forma como ocorre em *frameworks* de caixa branca. O exemplo 7.1 demonstra o modelo de código do *FORGE V8* para a implementação de uma aplicação derivada.

Exemplo 7.1: Exemplo de código de derivação da classe base do *FORGE V8*

```
/* ********************************************************
Name: Class CGameApplication
Desc: Define os métodos virtuais da classe base CApplication do FORGE V8 a
serem implementados pela nova aplicação, com o objetivo de controle específico
******************************************************** */
class CGameApplication : public CApplication
{
public:
    CGameApplication( HINSTANCE hInstance );
    virtual HRESULT SceneInit();
    virtual HRESULT SceneEnd();
    virtual HRESULT RenderFrame( LPDIRECT3DDEVICE8 pd3dDevice, float fDeltaTime );
    HRESULT InitDeviceObjects( LPDIRECT3DDEVICE8 pd3dDevice );
    HRESULT RestoreDeviceObjects( LPDIRECT3DDEVICE8 pd3dDevice );
    HRESULT InvalidateDeviceObjects( LPDIRECT3DDEVICE8 pd3dDevice );
    HRESULT DeleteDeviceObjects( LPDIRECT3DDEVICE8 pd3dDevice );
    HRESULT FinalCleanup();
//Input layer
    virtual VOID KeyUp( int iKey );
    virtual VOID KeyDown( int iKey );
//Net layer
    virtual void StartProcessStage( HWND &hDlg, int &nExitCode );
    virtual void DisplayStageSlotData( HWND hDlg, STAGEMSG_STAGEDATA*
                                       pStageDataMsg );
    virtual void PlayersAppMessages( HWND hDlg, GAMEMSG_GENERIC* pMsg,
                                     DWORD dwMsgSize, DPID idFrom,
                                     DPID idTo, HRESULT &hr );
    virtual void PlayersSysMessages( HWND hDlg, DPMSG_GENERIC* pMsg,
                                     DWORD dwMsgSize, DPID idFrom,
                                     DPID idTo, HRESULT &hr );
};
```

Na tabela 7.1, segue uma lista das principais classes implementadas nesta primeira versão do *FORGE V8*, constando os seguintes dados: nomenclatura, modo de utilização (derivação para a criação de novos modelos, configuração do *framework*, configuração da aplicação, ou gerenciamento) e a respectiva funcionalidade.

Tabela 7.1: Principais classes implementadas do *FORGE V8*

Classe	**Uso**	**Funcionalidade**
CApplication	derivação, configuração e gerenciamento	provê todas as funcionalidades do *FORGE V8*, permitindo a configuração de acordo com a aplicação, e servindo como ponto de acesso a todos os gerenciadores
CFullScreen	configuração interna	provê uma aplicação em tela cheia
CWindowedScreen	configuração interna	provê uma aplicação ajanelada
CCustomTime	gerenciamento	trata a temporização da aplicação e oferece a taxa de quadros/seg a cada *loop*
CError	gerenciamento	realiza o tratamento de exceções de todo o *framework*
CFile	gerenciamento e configuração	responsável pela leitura das mídias e configuração de cada gerenciador no intuito de validar a mídia utilizada
CVector2D	gerenciamento	oferece suporte aos cálculos matemáticos necessários a um ambiente 2D
CGraphicsLayer	gerenciamento e configuração	gerencia todo o processamento gráfico de mais alto nível (tratamento de *sprites*, texturas, etc.), além de enviar os resultados para ser renderizado pelo dispositivo gráfico
CRenderer2D	gerenciamento e configuração	responsável por renderizar os cenários 2D (atualmente implementado apenas para *DirectX Graphics*)
CCamera2D	gerenciamento	responsável por definir a porção do cenário a ser apresentada
CTexture	gerenciamento	trata as instâncias das imagens dos objetos do mundo
CSprite	gerenciamento	responsável por definir a imagem atual de um objeto, oferecendo métodos para troca de imagens, caracterizando a animação 2D
CTile	gerenciamento	responsável pela formação de um terreno correspondente a um cenário 2D, e controle de detecção de colisão
CFont	gerenciamento	responsável pela construção de textos para serem renderizados
CLight	derivação	responsável por gerar a iluminação de um cenário através de técnicas mais avançadas (não é obrigatório em um ambiente 2D)

Classe	**Uso**	**Funcionalidade**
CInputLayer	gerenciamento	controla os dispositivos de entrada, oferecendo os eventos gerados por eles e abstraindo o acesso a baixo nível (atualmente implementado apenas para *DirectInput*)
CInputDevices	gerenciamento e configuração	responsável por gerar uma instância de cada dispositivo de entrada encontrado e tratar suas características específicas
CSoundLayer	gerenciamento e configuração	responsável pelo controle da sonorização do ambiente, abstraindo o controle do dispositivo de som (atualmente implementado apenas para *DirectX Audio*)
CSound	gerenciamento	trata as instâncias dos sons e músicas através de *buffers* estáticos
CStreamingSound	gerenciamento	trata as instâncias dos sons e músicas através de *streaming*
CEventLayer	gerenciamento	responsável por gerenciar todos os eventos do *framework* referentes aos objetos do mundo, dispositivos de entrada, mensagens do sistema operacional, personagens inteligentes, etc. Este controle é feito com o auxílio do temporizador
CObjectLayer	gerenciamento	responsável pelo gerenciamento dos objetos no mundo e controle dos mapas correspondentes aos cenários
CWorldObject2D	derivação e configuração	responsável por tratar detalhes específicos dos mapas 2D (posicionamento de objetos estáticos, texturas dos *tiles*, dimensionamento, etc.)
CGameObject	derivação e configuração	oferece os métodos virtuais essenciais aos objetos do mundo. Esta classe deve ser derivada para criar objetos particulares
CObjectFactory	derivação e gerenciamento	responsável por criar instâncias de objetos dinamicamente
CGameObjectDatabase	gerenciamento	serve de repositório para guardar todas as instâncias de objetos contidos no mundo
CFSMState	derivação e configuração	responsável por instanciar os estados de comportamento das diversas classes de objetos NPCs
CFSMMachine	gerenciamento	máquina de estados responsável por oferecer a inteligência dos personagens NPCs

7.2 Estudo de caso: o jogo Super Tank

Uma vez definido o protótipo do *FORGE V8*, passou-se a desenvolver um jogo simples em versão demo com a finalidade de testar o referido *framework*. Esse jogo, denominado *Super Tank*, pertence à categoria ação e utiliza gráficos 2D isométricos. O jogador é representado por um tanque de guerra em um deserto completamente composto por planícies e repleto de tanques inimigos (NPCs). Essas planícies possuem grandes formações cristalinas impossíveis de serem destruídas por qualquer armamento conhecido, servindo, então, de ponto de defesa. Como em qualquer jogo do gênero, o objetivo principal do jogador, é a sobrevivência. Para isso ele deve eliminar seus inimigos, escapar de emboscadas, e obter energia e armamentos.

Super Tank foi desenvolvido para ser apresentado em tela cheia, e a sua interface de status, com a pontuação do jogador, energia de vida, energia de armamento, etc. se dá através de uma pequena interface na porção inferior da tela (ver figura 7.2).

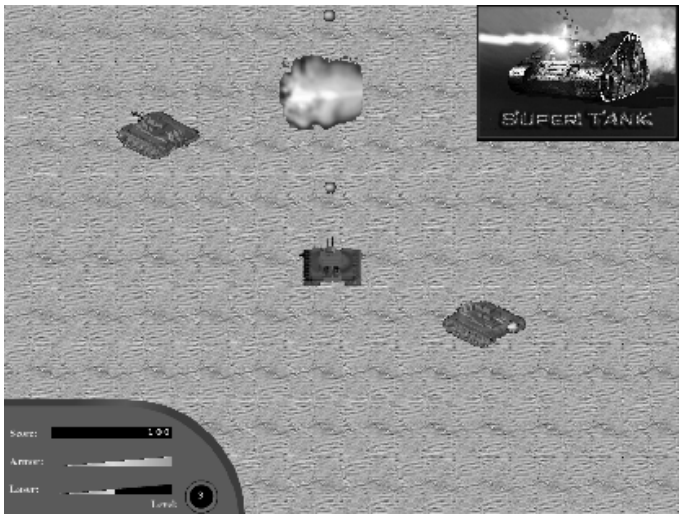


Figura 7.2: Tela principal de execução do Super Tank

Este jogo havia sido desenvolvido algum tempo atrás por alunos da graduação na disciplina Projeto e Implementação de Jogos do CIn. Este desenvolvimento foi realizado sem utilizar qualquer *framework* como base, teve a pretensão de estudar alguns problemas técnicos pontuais do desenvolvimento de jogos, e foi implementado utilizando *DirectX* em ambiente *Windows*. Uma vez o *FORGE V8* parcialmente desenvolvido, o mesmo jogo foi re-implementado. Esta nova implementação do *Super Tank* levou em consideração toda a estruturação original das suas classes, excluindo as funcionalidades já implementadas pelo *FORGE V8*, e aproveitando as funcionalidades específicas do jogo. Para obter uma melhor aproximação da versão original, o

código com as funcionalidades específicas do jogo foi aproveitado utilizando a mesma organização (espaçamentos, comentários, indentação, etc.). Desta maneira, pode-se ter uma comparação mais precisa entre os dois processos de desenvolvimento.

Com a utilização do *FORGE V8* como base, foram aproveitados do *Super Tank* original apenas a lógica da aplicação, e a estruturação dos objetos específicos do jogo (personagens). Todo o restante do código tratado como, gerenciador gráfico, gerenciador sonoro, controle de texturas, *sprites*, controle de eventos, interface com o sistema operacional, interface com *DirectX*, entre outros, foram tratados pelo próprio *FORGE V8*.

Com isso, esse estudo de caso pôde mostrar a facilidade de uso por parte dos desenvolvedores com a utilização do *FORGE V8*, e o desempenho final da aplicação gerada, além de verificar a corretude das suas funcionalidades. Alguns resultados são discriminados na tabela 7.2.

Tabela 7.2: Comparação dos resultados obtidos

Versão	**Linhas de** código	**Recursos envolvidos** (engenheiros de software)	**Desempenho** (frames/seg)
Super Tank 1.0 (versão antiga)	6.204	5	35
Super Tank 2.0 (versão FORGE V8)	3.597	3	32

Os resultados mostram que o *FORGE V8* diminuiu consideravelmente o esforço necessário a implementação do *Super Tank*, no que se refere ao número de linhas de código. Evidentemente, a implementação de uma aplicação pela segunda vez é mais fácil, mas neste caso também se insere o período de adaptação dos usuários com a aprendizagem do *FORGE V8*. A diminuição de esforço é caracterizada principalmente devido ao gerenciamento da aplicação e o acesso a *DirectX* ser realizado pelo *FORGE V8*, cabendo aos usuários o desenvolvimento apenas dos detalhes específicos do *Super Tank*. Como demonstração deste menor esforço, o exemplo 7.2 apresenta um trecho de código fonte necessário para o tratamento do mouse utilizando *DirectInput*, enquanto o exemplo 7.3 apresenta o código que o usuário deve inserir na sua aplicação para que o mouse seja tratado automaticamente pelo *FORGE V8*. Em relação ao desempenho, não houve grandes alterações, pois os testes foram realizados em um computador sem placa aceleradora gráfica, e mostrou que, mesmo sem otimização, o *FORGE V8* funciona bem.

Exemplo 7.2: Exemplo de código dividido em 4 métodos, necessário para implementar o tratamento do mouse através do DirectX

```
//-----------------------------------------------------------
// Name: Class CMouse
// Desc: Construtor
//-----------------------------------------------------------
CMouse::CMouse( LPDIRECTINPUT_GE pDI, HWND hWnd, BOOL bExclusive ) :
CInputDevice( pDI, hWnd, bExclusive ) {
    m_pTarget = NULL;
}

//-----------------------------------------------------------
// Name: InitializeObjects
// Desc: Inicializa o mouse
//-----------------------------------------------------------
HRESULT CMouse::InitializeObjects() {
    HRESULT hr;
    UnAcquire();
    FreeObjects();
// Cria o dispositivo
    hr = m_pDI → CreateDevice(GUID_SysMouse, &m_pDIDevice, NULL );
    if( FAILED( hr ) ) {
        CError( "CMouse", "O mouse não pode ser criado");
        return E_FAIL;
    }
// Seleciona o formato de dados para o padrão do mouse (predefinido pelo DirectX),
//com o objetivo de informar qual estrutura de dados deverá ser retornada por
//DirectInput
    hr = m_pDIDevice → SetDataFormat( &c_dfDIMouse_GE );
    if( FAILED( hr ) ) {
        FreeObjects();
        CError( "CMouse", "DirectInput não pode selecionar o formato do mouse");
        return E_FAIL;
    }
// Seleciona o nível de cooperação que DirectInput deverá interagir com o sistema
// e com outras aplicações que também o utilizem
    if( m_bExclusive ) {
        hr = m_pDIDevice → SetCooperativeLevel( m_hWnd, DISCL_EXCLUSIVE );
    }
    else {
        hr = m_pDIDevice → SetCooperativeLevel( m_hWnd, DISCL_NONEXCLUSIVE);
    }
```

```
    if( FAILED( hr ) ) {
        FreeObjects();
        CError( "CMouse", "DInput não pode selecionar o nível de cooperação");
        return E_FAIL;
    }
// Recebe o estado atual do mouse
    m_DIMouseState.lX = 0;
    m_DIMouseState.lY = 0;
    m_DIMouseState.lZ = 0;
    for( int i=0; i<8; i++ )
        m_DIMouseState.rgbButtons[i] = 0;
    return S_OK;
}

//-----------------------------------------------------------
// Name: Update
// Desc: Envia mensagens do mouse para o receptor
//-----------------------------------------------------------
HRESULT CMouse::Update() {
    HRESULT hr;
    DIMOUSESTATE_GE DINewState;
    if( NULL == m_pDIDevice ) {
        CError( "CMouse", "Mouse não inicializado");
        return E_FAIL;
    }
    ZeroMemory( &DINewState, sizeof(DINewState) );
    hr = m_pDIDevice → Poll();
    hr = m_pDIDevice → GetDeviceState( sizeof(DIMOUSESTATE_GE),
                                       (void*)&DINewState );
    if( FAILED( hr ) ) {
        hr = m_pDIDevice → Acquire();
        if( FAILED( hr ) )
            return E_FAIL;
        hr = m_pDIDevice → Poll();
        hr = m_pDIDevice → GetDeviceState( sizeof(DIMOUSESTATE_GE),
                                           (void*)&DINewState );
        if( FAILED( hr ) )
            return E_FAIL;
    }
    if( m_pTarget ) {
        long lX = DINewState.lX;
        long lY = DINewState.lY;
        long lZ = DINewState.lZ;
```

```
        if( lX || lY || lZ ) {
            m_pTarget → MouseMoved( lX, lY, lZ );
        }
        for( int i=0; i<8; i++ ) {
            if( DINewState.rgbButtons[i] & 0x80 ) {
                // Um botão foi pressionado
                m_pTarget → MouseButtonDown( i );
            }
            if( !( DINewState.rgbButtons[i] & 0x80 ) &&
                ( m_DIMouseState.rgbButtons[i] & 0x80 ) ) {
                // Um botão foi liberado
                m_pTarget → MouseButtonUp( i );
            }
        }
    }
    m_DIMouseState = DINewState;
    return S_OK;
}

//-----------------------------------------------------------------
// Name: SetReceiver
// Desc: Seleciona a classe responsável por receber as mensagens do mouse
//-----------------------------------------------------------------
VOID CMouse::SetReceiver( IMouseReceiver* pTarget ) {
    m_pTarget = pTarget;
}
```

Exemplo 7.3: Exemplo de código necessário para implementar o tratamento do mouse através do *FORGE V8* (inserido na implementação da classe base CApplication)

```
//-----------------------------------------------------------------
// Name: CGameApplication
// Desc: Construtor
//-----------------------------------------------------------------
CGameApplication::CGameApplication( HINSTANCE hInstance ) :
CApplication( hInstance ) {
    m_dwUseInput = DINPUT_USEMOUSE_NOEXCLUSIVE;
}
```

7.3 Conclusão do capítulo

Antes da implementação de uma aplicação, é necessário tomar decisões quanto ao uso das ferramentas que serão necessárias. Estas ferramentas devem ser selecionadas de forma a facilitar o processo de desenvolvimento. Neste capítulo foi apresentado tais ferramentas e discutido o porque das mesmas terem sido escolhidas para o *FORGE V8*.

Como forma de verificar a viabilidade deste *framework*, foi implementado um subconjunto dos seus componentes. Esta implementação foi inicialmente direcionada para a construção do jogo *Canyon*, mas devido a enorme dificuldade encontrada em trabalhar com ambientes 3D, decidimos por implementar primeiramente os componentes característicos de ambientes 2D retangulares e isométricos, o que resultou numa nova possibilidade de testes como o estudo de caso do jogo *Super Tank*.

Finalizando, o capítulo apresenta os resultados obtidos e conclusões tiradas do processo de implementação do *Super Tank*, cujo desenvolvimento tomou como base o subconjunto de componentes do *FORGE V8*. Isso mostra que estes componentes podem ser instanciados para dar suporte a jogos, facilitam características intrínsecas a todo o processo, e tratam detalhes relacionados a novas implementações.

Capítulo 8

Conclusões

Neste capítulo são apresentados as contribuições deste projeto de dissertação, as principais dificuldades encontradas no decorrer deste trabalho, propostas para trabalhos futuros e comentários finais.

8.1 Contribuições

Neste trabalho foi proposto o *FORGE V8*, um *framework* original para suporte ao desenvolvimento de jogos de computador e aplicações multimídia portáveis, que fornece os serviços básicos requisitados por aplicações dessa natureza, e tem a intenção de descartar a necessidade dos desenvolvedores escreverem código a baixo nível. O objetivo desse projeto foi facilitar o trabalho dos desenvolvedores de jogos, reduzindo o tempo, a complexidade e o custo envolvidos nesse processo. Além disso, devido à alta modularidade apresentada no *FORGE V8*, diversos gerenciadores podem ser utilizados independentemente para compor aplicações multimídia específicas, sem a necessidade da incorporação de todo o *framework* no projeto envolvido. Por outro lado, os gerenciadores são integrados através de uma interface de mais alto nível, que automatiza diversos dos requisitos necessários ao gerenciamento de um jogo.

Uma outra grande contribuição deste trabalho é que ele serve como base ou referência para aplicabilidade de padrões de projeto em *frameworks* para jogos, dado que não é do nosso conhecimento a existência de trabalhos relacionados, além de estar em concordância com os principais requisitos de engenharia de software aplicados a projetos orientados a objetos.

Este trabalho também serviu para enfatizar que o sucesso de um jogo é obtido pelo impacto que ele causa as pessoas, e principalmente, por ser "jogável". Certamente, a implementação deste tipo de aplicação é o maior problema devido a alta

complexidade envolvida, mas pensar em um jogo divertido e projetá-lo, é definitivamente muito importante. Os usuários apreciam os detalhes, portanto essa foi uma questão muito séria levada em conta no projeto do jogo *Canyon*, que teve o objetivo de implementar um jogo de ação/inteligência nos padrões da indústria atual, fazendo uso das tecnologias de ponta.

Para o uso das novas tecnologias no *FORGE V8* e consequentemente no jogo *Canyon*, este trabalho definiu a infra-estrutura necessária para o desenvolvimento dessas aplicações, apresentando um amplo conjunto de ferramentas extremamente importantes e suas características.

E uma última contribuição, mas não menos relevante, foi o estudo realizado nos motores de código aberto, apresentando as suas boas características, e também os seus problemas, os quais devem ser tratados no desenvolvimento de qualquer *framework* para jogos, e consequentemente foi tratado pelo *FORGE V8*.

8.2 Principais dificuldades

O pioneirismo deste trabalho indica a nossa falta de maturidade no desenvolvimento de jogos compatíveis com a indústria de jogos atual, principalmente devido ao conhecimento insuficiente sobre o desenvolvimento de jogos comerciais, computação gráfica 3D, e métodos de otimização específicos para estas aplicações. Isso ocorre devido as técnicas mais utilizadas no desenvolvimento de jogos somente agora estarem começando a ser publicadas.

Outra dificuldade, foi a falta de conhecimento do estilo de programação das bibliotecas multimídia *DirectX* e *OpenGL*, o que exigiu um período significativo para a aprendizagem, além das técnicas de programação em C++ no que se refere ao desenvolvimento para sistema operacional *Windows*.

Em relação ao projeto do jogo *Canyon*, a sua coerência com o mercado atual foi uma tarefa complicada devido a ter que apresentar uma boa jogabilidade, personagens inteligentes com um eficiente poder de decisão, e cenários tridimensionais complexos, todos com a intenção de atingir um nível aproximado aos jogos modernos e de prender ao máximo a atenção dos jogadores.

Em relação ao estudo realizado nos motores de código aberto, nos foi exigido um grande esforço devido a não estarem disponíveis documentação suficiente, principalmente no caso do *Golgotha*, que também não apresentou legibilidade de código.

Por último, se insere toda a complexidade encontrada no desenvolvimento de um *framework* para jogos, principalmente na intenção de prover um de solução geral, o

que requer mais tempo para projeto e testes. Além disso, um *framework* orientado a objetos pode introduzir alguma sobrecarga no desempenho de jogos, portanto necessita que seu projeto já seja desenvolvido com a pretensão de atingir um bom desempenho, para então poder obter um código mais otimizado.

8.3 Propostas para trabalhos futuros

Como continuação deste projeto, existe uma série de trabalhos futuros que podem ser desenvolvidos, os quais estão descritos a seguir:

- Implementação dos diversos componentes que ainda não foram adicionados ao *FORGE V8* (IA avançada, redes, gráfica tridimensional, modelagem física, editores de cenários, entre outros);
- Otimização de código visando atingir o melhor desempenho possível;
- Implementação de uma outra versão do *FORGE V8* para outros sistemas operacionais como *Unix* e *Linux*, utilizando outras bibliotecas gráficas como *OpenGL*;
- Verificação da eficiência do *framework* por meio de análise de desempenho de estudos de caso mais complexos que se baseiem principalmente em ambiente tridimensional;
- Comparação de desempenho entre as implementações de *OpenGL* e *DirectX*, e rede ponto a ponto e cliente/servidor aplicados ao *FORGE V8*;
- Desenvolvimento de um editor de cenários para o jogos *Canyon* e definição dos seus diversos níveis;
- Desenvolvimento de técnicas e algoritmos para a coordenação dos personagens inteligentes do jogo *Canyon* em relação as atividades de patrulhamento, exploração e perseguição;
- Implementação do jogo *Canyon* (lógica, cenários, interface gráfica, trilha sonora, entre outros) utilizando como motor o *FORGE V8*;

As possibilidades para a evolução do *FORGE V8* são intermináveis. Além de todas as peculiaridades relacionadas ao desenvolvimento de jogos, a indústria explora atualmente novas maneiras para a utilização destas tecnologias em outras aplicações,

como exemplo, em comércio eletrônico, em *plug-ins* para aplicações comerciais, em visualização científica, em análise de imagens médicas, e outros projetos tridimensionais auxiliados por computador [65].

8.4 Comentários finais

Nos últimos anos, a evolução dos jogos de computador tem sido espantosa, tornando-se aplicações limite para a computação, e passando a ditar regras para a indústria de hardware. Com o interesse em dominar o conhecimento envolvido nesta área de aplicação, o Centro de Informática (CIn) da UFPE se tornou pioneiro no país em pesquisas relacionadas a jogos, apresentando projetos em desenvolvimento e duas dissertações de mestrado já defendidas nesta área, além da inserção de duas disciplinas à estrutura curricular da graduação e pós-graduação. No entanto, este presente projeto foi o primeiro esforço do CIn com o objetivo de utilizar realmente as tecnologias de ponta da indústria de jogos, na intenção de construir ferramentas para viabilizar o desenvolvimento de jogos de computador compatíveis com o padrão de mercado, diferentemente de projetos anteriores que não eram tão ambiciosos.

Além disso, jogos são um ótimo laboratório para a prática e avaliação de estudos realizados em diversas áreas como computação gráfica, IA, modelagem física, redes, entre outras.

Como conseqüência, e em adição a todo o interesse comercial envolvido, o desenvolvimento do *FORGE V8* é uma alavanca para o impulsionamento do desenvolvimento de jogos no CIn, devido a maior facilidade atribuída aos projetos que trabalharem com este tipo de aplicação, e a todo o conhecimento que foi adquirido com o seu desenvolvimento. Em razão do *FORGE V8* ser um motor genérico que segue os princípios de engenharia de software, várias equipes poderão trabalhar em projetos distintos utilizando-o como base. Uma única equipe, especializada neste motor, poderá realizar todos os melhoramentos e atualizações que lhe forem necessários, repassando estas modificações para todas as outras equipes, sem haver a necessidade de adaptação ou interferência no projeto em andamento. Desta forma, é evitado ao máximo a programação, e possibilita que o conteúdo lógico do jogo fique totalmente separado dos módulos de baixo nível do motor, consequentemente, gerando código de alto nível, de fácil gerenciamento, e de muito mais fácil entendimento.

Em resultado a todo este trabalho, além das publicações relacionadas ao jogo *NetMaze*, outros dois artigos relacionados ao *FORGE V8* foram submetidos [47][48],

e estão em fase de avaliação. Em adição, as idéias deste trabalho proporcionaram a criação de novos projetos de mestrado, e a definição do meu próprio projeto de doutorado. Posteriormente, o projeto e o código fonte do *FORGE V8* poderam ainda ser divulgados na web em sites de projetos *Open Source*, como exemplo o *SourceForge*[1].

Por fim, fazer um bom trabalho em um *framework* requer um alto investimento de longo período, que salvará tempo somente em projetos futuros. Portanto, a experiência adquirida neste projeto mostra que deve-se estar bastante preparado para gastar tempo e cérebro no desenvolvimento de jogos de computador, tanto no desenvolvimento do conteúdo das aplicações como enredo, personagens, gráficos, níveis, sons, músicas, tão quanto no desenvolvimento das ferramentas necessárias como, editores de cenários e motor. De acordo com Michael Abrash no seu artigo [1], existem três segredos para que uma equipe possa obter sucesso em um jogo: "trabalhar intensivamente, trabalhar intensivamente, e trabalhar intensivamente".

[1] http://sourceforge.net

Apêndice A

O Projeto do Jogo *Canyon*

Canyon foi concebido para ser um jogo híbrido de ação-inteligência baseado num acontecimento imaginário. O seu período de concepção inicial durou cerca de três a quatro meses e continuou a evoluir a medida que surgiam novas idéias. A implementação deste jogo demanda tecnologia de ponta, necessitando da utilização de técnicas complexas para renderização de cenários, entidades com bom grau de inteligência e modelagem física realista.

Este apêndice apresenta o documento do projeto do *Canyon*. Este documento é responsável por especificar todas as características essenciais a construção desse jogo, o qual deve abordar os seguintes itens: a sua classificação baseada nas diversas categorias; o objetivo, estabelecendo as fantasias a serem suportadas e as emoções a serem produzidas nos jogadores; o tema (*backstory*), meio pelo qual se expressa o objetivo, ou seja, o ambiente no qual o jogo se desenvolve; pesquisa e decisões de projeto, responsável pelo detalhamento do tema e decisões ligadas à perspectiva, ao número de participantes, à temporização e ao nível de realismo; a estruturação, responsável pelo detalhamento do ambiente, personagens e regras gerais; e a interface gráfica de usuário, responsável pelas configurações de controle da aplicação, e pelo cenário do jogo. A seguir são detalhadas essas especificações.

A.1 Classificação

Canyon se classifica como um jogo de ação-inteligência. Jogos de ação se caracterizam por ser em tempo real, e apresentam uma maior ênfase na parte gráfica e na sonorização. Tais jogos primam pela habilidade dos seus jogadores no sentido deles terem um boa coordenação motora e rápidas reações no tempo. As categorias dentre este grupo de jogos em que o *Canyon* se enquadra são:

Jogos de combate apresentam confrontos diretos. A principal função do jogador é destruir os seus oponentes, que na sua maioria são controlados pelo computador (NPCs[1]). O desafio maior é posicionar-se de maneira a evitar que seja atingindo por um inimigo enquanto tenta acertá-lo.

Jogos de velocidade privilegiam a rapidez na realização dos objetivos. O objetivo do jogador é realizar uma determinada tarefa num menor tempo possível, ou num tempo predeterminado.

Jogos de inteligência, por sua vez, se caracterizam por enfatizar a capacidade intelectual e a paciência do jogador para planejar suas atividades. A preocupação maior está voltada para o comportamento dos personagens controlados pelo computador, e para o enredo do jogo. As categorias dentre este grupo de jogos em que o *Canyon* se enquadra são:

Jogos de aventura baseiam-se na idéia de que o jogador deve navegar através do ambiente acumulando recursos e dicas que o ajudem a alcançar o seu objetivo. O jogador deve utilizar a inteligência para juntar os fatos e saber o momento ideal de utilizar os recursos adquiridos.

Jogos de estratégia apresentam aos jogadores um problema complexo, que deve ser resolvido através dos recursos disponibilizados, que são geralmente escassos. As estratégias são as formas de como o usuário planeja utilizar os recursos com o intuito de solucionar o problema.

A.2 Objetivo

Como em qualquer jogo de ação, *Canyon* tem o objetivo de apresentar um ambiente onde um elevado grau de emoção dos jogadores é a sua principal razão de ser. Como em qualquer jogo de inteligência, também deve-se possibilitar que o planejamento e a tomada de decisões sejam características necessárias por parte destes jogadores. A junção destas categorias, provavelmente, é o que atribui ao *Canyon* o seu maior diferencial, principalmente por inserir bastante estratégia num jogo que poderia ser puramente de ação.

[1]Non-Player Characters, entidades que tomam decisões e realizam ações com autonomia

A.3 Tema (*Backstory*)

O jogo *Canyon* é caracterizado por seu ambiente se passar num planeta extremamente montanhoso, chamado *Canyon*, que há muitos séculos foi habitado por uma civilização bastante evoluída, e que por algum motivo desconhecido foi abandonado. Antes de partir deste planeta, a civilização deixou uma grande quantidade de construções bastante modernas as quais o tempo se encarregou de parcialmente deteriorá-las. *Canyon* possuia em abundância nas suas terras, minérios de grande valia (*Chumbex* e *Platon*) pois eram essenciais para a geração de energia. Muito tempo depois, um povo de um planeta bem próximo, chamado *Xarix*, descobriu *Canyon*, iniciando a partir daí um processo de exploração destes minérios. Mas em 2185, os governantes de um outro planeta muito distante, chamado *Zagotha*, que utilizavam espiões para buscar novas fontes de energia pelo universo devido a estas estarem se extinguindo nos planetas, recebe informações sobre a abundância de minérios no planeta *Canyon*. Logo, com a intenção de dominar este planeta e explorar suas riquezas, e devido a possuir um exército fortemente armado, elabora um plano estratégico de invasão à *Canyon*, mesmo sabendo que já existiam exploradores neste planeta, ou seja, incluindo no plano o aprisionamento ou expulsão deste povo. Pouco tempo depois, *Canyon* sofre a invasão de forma muito ofensiva por parte de *Zagotha*, e sem nenhuma chance de defesa, os cientistas e trabalhadores de *Xarix* que exploravam o planeta são facilmente aprisionados. Logo após, com *Canyon* em seu poder, o exército de *Zagotha* iniciou uma política para garantir de forma eficiente a segurança do planeta e dos seus exploradores.

Devido ao povo de *Xarix* ser bastante pacífico, quando a notícia destes acontecimentos chega ao seu governo, o 1^o ministro e demais governantes se reúnem para formar uma missão diplomática, na intenção de libertar os prisioneiros e propor uma divisão das riquezas de *Canyon* com o povo de *Zagotha*. Então esta comitiva é enviada a *Canyon* para realizar as negociações, mas ao governo de *Zagotha* não interessava divisão, e sim, ser o único explorador de *Canyon*, ocasionando a quebra das negociações e também o aprisionando os encarregados da pacificação, incluindo o 1^o ministro de *Xarix*.

Com esses aprisionamentos, e pela necessidade de energia para a sobrevivência do povo, restava apenas uma única alternativa ao governo de *Xarix*, decretar guerra a *Zagotha*. Mas como o exército de *Xarix* possuia armamentos muito antigos, então o seu governo financiou a compra de novos armamentos, fabricados por mercenários que buscavam minérios em planetas distantes, e ofereceu uma enorme recompensa aos pilotos do seu exército que conseguissem resgatar os prisioneiros e expulsar o

exército de *Zagotha* de *Canyon*.

O ponto principal das batalhas entre os exércitos de *Xarix* e *Zagotha* acontecem em *Canyon*, local onde estão encarcerados todos os prisioneiros. Devido a infantilidade do governo de *Xarix* de ter oferecido a enorme recompensa apenas para os pilotos que conseguissem resgatar os prisioneiros, e não para todo o grupo de pilotos, poderão haver disputas ilegais e até traições entre eles.

A.4 Pesquisa e Decisões de projeto

A pesquisa relacionada ao tema do jogo constituiu-se da realização de um estudo em diversos jogos de ação (*XWing Alliance*, *Rebel Assalt*, *Rogue Squadron* [43]) com o intuito de serem extraídas algumas das suas boas idéias (ambiente, personagens, fatos e conseqüências), como também para serem verificados os níveis de qualidade e complexidade apresentados por estes jogos. Além disso, sessões de *brainstorms* foram realizados afim de se discutir e propor novas características e objetivos que seriam essenciais e/ou inovadores ao jogo *Canyon*, assim compondo a sua própria identidade.

Uma série de decisões devem ser tomadas no início do desenvolvimento do projeto de um jogo. As decisões de projeto referentes ao *Canyon* e quais suas influências no processo de desenvolvimento deste jogo serão comentadas a seguir.

A.4.1 Perspectiva em 1^a pessoa X Perspectiva em 3^a pessoa

Canyon considera as duas formas de perspectiva com um mesmo grau de importância, devido ser apresentado num ambiente 3D, onde diversas câmeras com perspectivas diferentes poderão fazer parte do cenário. Deixando que o jogador selecione a que lhe for mais conveniente. Portanto, quando selecionada a opção 1^a pessoa, o jogador terá uma visão do ambiente que corresponde a visão real do seu personagem, já que o jogador controlará apenas um. E no caso de selecionada a opção 3^a pessoa, o jogador terá uma visão mais abrangente e com posicionamentos e direções diferenciadas. Contudo, o jogador continuará acompanhando o seu personagem como ponto de referência.

A.4.2 Monousuário X Multiusuário

Uma característica fundamental é que o *Canyon* seja multiusuário em rede, de forma que os jogadores possam controlar os seus personagens através de computadores

distintos. Isto possibilita que as batalhas ocorram entre os vários jogadores, e não só entre um jogador e personagens controlados pelo computador. Esta característica também é bastante interessante, se considerado que os jogadores poderão formar parcerias em busca de cooperação, e não só jogar de forma individual. Na sua versão preliminar, o *Canyon* comportará o máximo de oito jogadores.

A.4.3 Turnos X Tempo real

Tempo real é uma característica fundamental de todos os jogos de ação. E consequentemente, esta abordagem também será utilizada pelo *Canyon*. Desta forma, ocorre um aumento considerável no grau de complexidade da implementação devido ao grande número de mensagens trocadas entre os componentes do jogo, e também em relação ao tráfego de rede. Assim, gerando uma necessidade de melhor tratamento dos problemas relacionados a consistência de dados e desempenho final da aplicação.

A.4.4 Realismo X Subjetivismo

A intenção do *Canyon* é que seus personagens autônomos - NPCs, passem aos seus espectadores a ilusão de acontecimentos reais, ou seja, eles devem se comportar de tal forma que seja difícil distinguir se os mesmos estão sendo controlados pelo computador ou por outro jogador. As ações dos personagens devem ter significados lógicos e coerentes com a entidade que ele está representando. Desta forma, apresentando características que darão credibilidade a si próprio.

A.5 Estruturação

Nesta estruturação, estão especificados o ambiente, os personagens, e as regras gerais do jogo *Canyon*, que são descritos a seguir.

A.5.1 O ambiente

O ambiente do jogo é composto por enormes labirintos que são constituídos por formações rochosas - canyons - que apresentam grutas e portais de passagem que se ligam a cidades desabitadas. As vias de passagem destes labirintos apresentam espaçamentos diferenciados, de maneira que em determinados pontos estas podem alargar ou estreitar. O mundo do jogo possui diversos tamanhos predefinidos, que devem ser configurados de acordo com o grau de complexidade dos níveis e o número

de participantes de uma determinada partida. A visualização deste cenário por parte dos participantes será baseada numa pequena porção do seu interior, como mostra a figura A.1. Detalhes importantes podem ser inseridos ao ambiente para ampliar e proporcionar uma maior realidade ao cenário do jogo. Eles são flora, lagos, rios, cachoeiras, céu, etc., e alguns efeitos de modificação tais como estações climáticas (inverno, verão), períodos do dia (manhã, noite), ventanias, explosões, etc., todos com seus devidos efeitos sonoros.

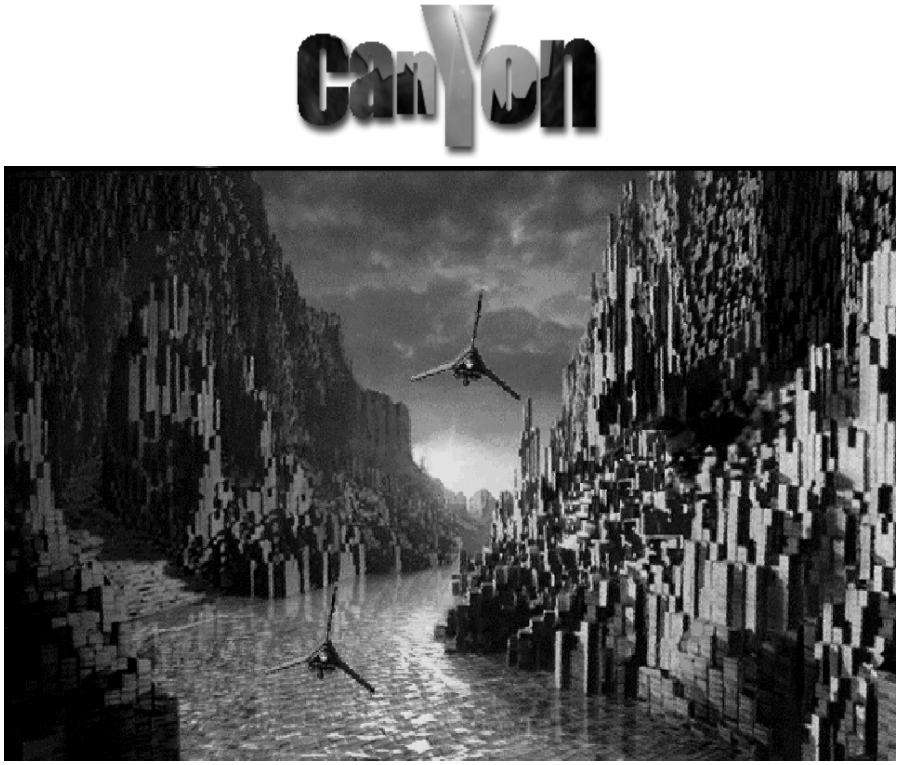


Figura A.1: Exemplo de cenário (3D Studio Max) - perspectiva em 3^a pessoa

A.5.2 Os personagens

De acordo com o tema imaginário deste jogo, os personagens que caracterizam as batalhas são aeronaves. Estas aeronaves podem ser classificadas em dois grupos. As aeronaves referentes ao exército de *Zagotha* - controladas pelo computador (NPCs) e entituladas *Bright*, e as aeronaves referentes ao exército de *Xarix* - controladas pelos jogadores e entituladas *Force*. Estas aeronaves são apresentadas na figura A.2.

Cada um destes grupos apresentam características diferentes. O grupo *Bright* é formado por aeronaves bastante modernas, possui quatro modelos característicos: *Bright Z15* que é constituída com alto poder de ataque e capacidade de manobras

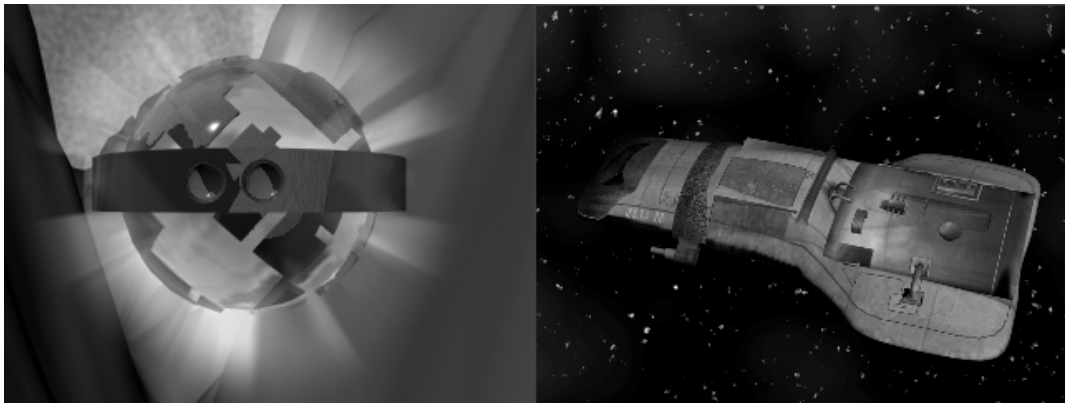


Figura A.2: Aeronaves *Bright* (esquerda) e *Force* (direita)

radicais; *Bright Z17* que possui como sua principal característica o fator defesa; *Bright Z19* que possui motor super turbo e alto poder de percepção; e *Bright V21* que possui poder de ataque e velocidade moderadas. O grupo *Force* é formado por aeronaves desgastadas (antigas). Possui três modelos: *Force X9* que é caracterizada pelo alto poder de defesa; *Force X11* que possui um ótimo motor super turbinado e capacidade de manobras radicais; e *Force X13* que é caracterizada pelo seu alto poder de ataque e capacidade de percepção. Estas aeronaves e suas características são detalhadas na tabela A.1.

Tabela A.1: Aeronaves e suas características - fator de 0 a 100%

Grupo	**Bright**				**Force**		
Modelo	**Z15**	**Z17**	**Z19**	**Z21**	**X9**	**X11**	**X13**
Ataque	100	40	40	80	50	60	100
Defesa	50	100	40	40	100	70	60
Velocidade	60	50	100	80	50	100	60
Percepção (radar)	50	50	100	50	50	50	100
Estabilidade	80	90	100	80	80	100	90
"Manobrabilidade"	100	70	60	70	60	100	50

As aeronaves devem obedecer a alguns diversos comandos dos jogadores: acelerar/freiar, atirar, ganhar/perder altitude, rotacionar na vertical e horizontal, selecionar armamento, ativar poderes especiais, e modificar a perspectiva de visão.

A.5.3 As regras gerais

O *Canyon* é baseado em missões que são atribuídas aos jogadores à cada nível de uma partida. Estas missões são caracterizadas por um determinado conjunto de objetivos a serem atingidos, tais como: resgatar prisioneiros, destruir inimigos e/ou bases inimigas, recolher recursos, adquirir bônus, recolher chaves de caminhos ou passagens obscuras, proteger e/ou escoltar aliados, e desvendar passagens escondidas. Em algumas missões, um menor intervalo de tempo para a conclusão será bastante importante, devido a possibilidade de uma maior recompensa.

O objetivo principal de uma partida em geral, é a libertação dos prisioneiros no mundo inimigo. Mas para isso, as diversas missões devem ser cumpridas, com bastante cautela, e rapidez quando necessário. O jogador deve apresentar um conjunto de habilidades que lhe assegure um comportamento satisfatório, ou seja, boa coordenação de movimentos para pilotagem (manobras, perseguição, fuga, mira, disparos, entre outros), além de saber decidir que melhoramentos devem ser realizados a sua aeronave nos intervalos entre as missões. No caso de parcerias entre os jogadores, as missões apresentarão suas dificuldades aumentadas para compensar os outros jogadores que não possuírem parceiros. Neste caso, a missão não será para o jogador, mas sim para o grupo. Isto significa que a junção dos objetivos conquistados é do grupo, mas os recursos continuam pertencendo apenas ao jogador que o adquiriu.

Os recursos que os jogadores podem recolher são os minérios de alto valor energético que são encontrados no ambiente (*Chumbex* e *Platon*), aos quais poderão ser utilizados no término de cada nível do jogo. Estes minérios são as principais moedas do jogo, e valem como fonte de melhoria para as condições das aeronaves dos jogadores: aquisição de armamentos, incremento da potência do motor, aquisição de radar de melhor alcance para visualização e detecção, conserto das avarias sofridas em níveis anteriores, entre outros. Todas as aeronaves do jogo possuem uma taxa de força que varia de 0 a 100%. No início do jogo, esta taxa é de 100%, pois as aeronaves encontram-se em perfeito estado. A cada ataque sofrido - aeronave atingida por armamentos inimigos, ou erro de manobra do piloto que resulte em colisão com o terreno ou montanhas - a aeronave sofrerá um grau de avaria de acordo com o acontecimento. As avarias sofridas e suas respectivas porcentagens de perda de força são mostradas na tabela A.2.

Ao chegar a 0% de força, a aeronave em questão explodirá totalmente, podendo finalizar o jogo para o jogador. Mas, este poderá continuar no jogo, se possuir uma aeronave reserva que tenha sido adquirida anteriormente durante as passagens

Tabela A.2: Taxa de avariação das ações sofridas pelas aeronaves

Ação sofrida	**Perda de força**
Explosão causada por colisão com outras aeronaves	1%
Explosão causada por armamentos	1% (laser/metralhadora) 2% (mísseis)
Explosão causada por colisão com o terreno ou montanhas	1%

de níveis. Ou se estiver em parceria com outro jogador que tenha adquirido uma aeronave reserva para o seu grupo, doando-a ao seu companheiro. No caso de não possuir uma aeronave reserva, continuará no jogo se seus recursos recolhidos forem suficientes para adquirir uma nova aeronave. Se não, não poderá continuar ativo, e o jogo estará finalizado para ele, mas poderá participar dos acontecimentos como um "fantasma" caso esteja jogando em grupo, não possuindo a capacidade de realizar qualquer ação que interfira no jogo. Desta maneira, estará invisível, e possuirá apenas a capacidade de locomoção para acompanhar as ações realizadas pelos outros jogadores, e comunicação com parceiros do seu grupo, se este for o caso.

A aquisição de novos armamentos, de poderes especiais, de novas aeronaves, e até mesmo a recuperação das aeronaves avariadas, é realizada numa nave muito poderosa, habitada por mercenários, que vaga continuamente no espaço. Esses mercenários, compram minérios por valores muito baixos em mundos distantes, fabricam poderosos armamentos, e vendem por valores bastante elevados em terras que apresentam conflitos.

A taxa de força das aeronaves poderá ser incrementada ao final de cada nível, conforme a escolha do jogador e a possibilidade para esta realização. Para ajustar as avarias das aeronaves, repor partes danificadas, e adquirir novos componentes, deve-se pagar estes serviços com uma determinada quantidade de recursos recolhidos. Os valores correspondentes ao ajuste/reposição para uma determinada porcentagem de avaria são mostrados na tabela A.3.

Tabela A.3: Valores para o incremento de força das aeronaves

Força da aeronave	**Valor**
10%	10 *Chumbex*
2%	10 *Platon*

No caso do jogador não possuir recursos, não poderá reajustar e/ou fazer melhoramentos na sua aeronave, nem adquirir armamentos, e iniciará o próximo nível com a aeronave avariada. As aeronaves NPCs também possuem taxa de força, mas apresentam uma importante diferença. Elas iniciam todos os níveis com força total, mesmo que tenham sido destruídas no nível anterior. A quantidade de NPCs encontrados no ambiente deve variar de acordo com o número de jogadores envolvidos e o grau de dificuldade do nível corrente.

No início de cada nível, os NPCs se encontram em *Canyon*, enquanto que as aeronaves controladas pelos jogadores (*Force*), encontram-se estacionadas no interior de uma nave de mercenários. No primeiro nível, as aeronaves possuem apenas armamentos básicos, e itens e pontuação zerados. Quando iniciado o nível, é liberada a decolagem, e as aeronaves *Force* devem partir para *Canyon*. Quando finalizado o nível, as aeronaves *Force* retornam a nave dos mercenários para receberem novas missões. Neste momento da passagem de nível, os jogadores obedecerão a um determinado tempo que lhes será concedido para que possam realizar os melhoramentos nas suas aeronaves, da maneira que preferirem, baseados nos recursos recolhidos nos níveis anteriores.

O funcionamento e os respectivos valores dos itens e poderes especiais que poderão ser adquiridos e incorporados às aeronaves para engrandecer as diversidades das batalhas, são apresentados na tabela A.4. Os valores são taxados em *Chumbex*, mas com a soma de *Platon* que se iguale ao mesmo valor de *Chumbex*, o jogador poderá realizar a aquisição. A referência entre os minérios é a seguinte: 1 *Chumbex* é equivalente a 5 *Platon*. A quantidade dos minérios espalhados pelo ambiente deve ser definida de acordo com os níveis do jogo.

Tabela A.4: Itens e poderes disponíveis para compra

Item/Poder	**Capacidade**	**Funcionamento**	**Valor**
Laser (básico)	Pleno	atinge um objeto qualquer a uma distância x, desintegra após essa distância	
Metralhadora	100 tiros	atinge um objeto qualquer a uma distância 2x, explode sem efeito algum após essa distância	20 *Chumbex*
Lança mísseis	5 mísseis	atinge um objeto qualquer a uma distância qualquer, explode	30 *Chumbex*

Item/Poder	Capacidade	Funcionamento	Valor
		quando atinge um objeto ou o terreno	
Alcance de radar	Pleno	Atribui a capacidade de enxergar outras aeronaves num raio de visão 3 vezes maior que o normal	40 *Chumbex*
Turbo para motor	Pleno	Aumenta velocidade do motor da aeronave	50 *Chumbex*
Invisibilidade	3 acionamentos	cada acionamento dura 30 segundos - a aeronave fica invisível para todas as outras	50 *Chumbex*
Teletransporte	5 acionamentos	acionamento é efetuado por um portal - a aeronave é transportada instantaneamente para um outro ponto no espaço	30 *Chumbex*
Escudo	10 acionamentos	cada acionamento dura 30 segundos - a aeronave fica totalmente protegida durante o tempo de acionamento	40 *Chumbex*
Submarino	5 acionamentos	cada acionamento dura 2 minutos - permite que a aeronave também navegue pela água (submersa ou não)	80 *Chumbex*
Miniaturização	3 acionamentos	cada acionamento dura 30 segundos - a aeronave se torna 50% menor	100 *Chumbex*
Aeronave *Force X9*	100% de força		100 *Chumbex*
Aeronave *Force X11*	100% de força		120 *Chumbex*
Aeronave *Force X13*	100% de força		150 *Chumbex*

Além da missão designada aos jogadores em cada nível, cada jogador recebe uma pontuação pelas suas ações. Estas ações são conseguidas pelas habilidades dos jogadores no decorrer dos níveis. A conquista do título de vencedor é conseguida de acordo a soma da pontuação e das missões realizadas pelos jogadores. As pontuações referentes a cada ação são mostradas na tabela A.5.

Tabela A.5: Pontuação das ações dos jogadores

Ação executada	**Pontuação**
Prisioneiro resgatado	100
***Bright* atingido por armamento**	25
***Force* atingido por armamento**	50
Conclusão de missão	1000
Bônus variável por conclusão de nível	de acordo c/ o tempo gasto e o número de jogadores

A.6 Interface Gráfica de Usuário

A interface gráfica de usuário do *Canyon* se divide em duas partes: a interface de configuração de controle da aplicação (seleção de dispositivos de entrada, número de jogadores, parcerias entre jogadores, sessões para partida multiusuário, volume de som, resolução de vídeo, entre outras) como exemplificado na figura A.3, e a interface do próprio cenário do jogo (figura A.1). Todas utilizando gráficos com alto grau de qualidade. A interface de configuração pode ser algo baseado numa mesclagem de *Caesar III* e *Age Of Empires II*.

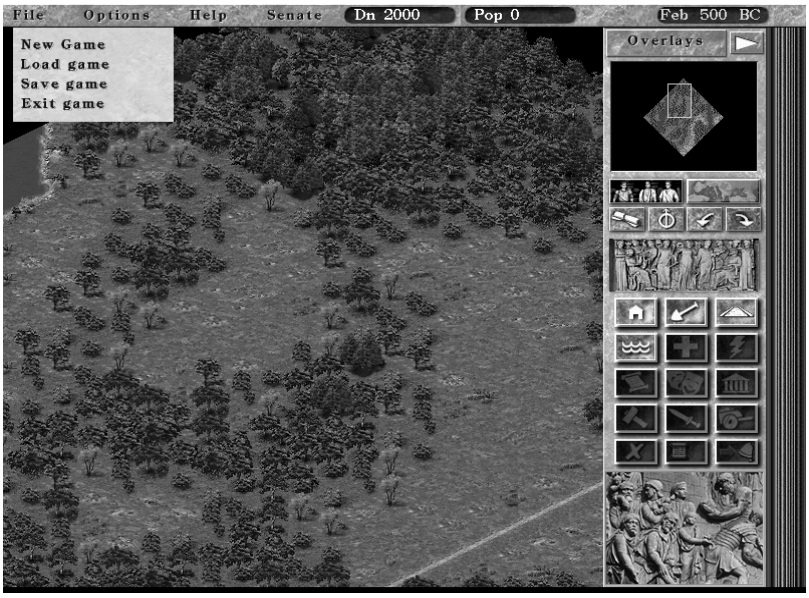


Figura A.3: Interface de configuração - Exemplo de *Caesar III*

Os comandos do jogo poderão ser acionados através de mouse, teclado, e/ou joystick. Um lado da interface de usuário, provavelmente numa porção lateral ou mais

baixa da tela, mostrará informações estatísticas para o jogador corrente (taxa de força da aeronave, número de prisioneiros resgatados, itens recolhidos, tempo decorrido no nível atual, situação dos armamentos e poderes especiais, entre outros), e um minimapa de todo o ambiente. A área correspondente a cena do ambiente apresentada na tela será focalizada no minimapa. Rotacionando a visualização desta cena para um jogador, acarretará também na rotação do foco no minimapa, indicando qual o sentido a que o jogador está tomando (norte, sul, leste, oeste). Também, na porção mais baixa da tela, o jogador poderá enviar mensagens aos outros jogadores, sendo estes seus adversários ou seu parceiro. Isto é muito interessante para haver uma troca de informações entre parceiros.

A.7 Considerações Finais

1. Alguns valores e porcentagens utilizados nestes projeto poderão ou deverão sofrer modificações na fase de implementação do jogo, devido a possibilidade de se conseguir um melhor resultado.

2. A especificação *Canyon* foi descrita de forma a apresentar uma visão geral de todo o jogo. Já, a descrição dos níveis, faz parte de um outro projeto que deve apresentar a composição dos cenários e as missões a serem atingidas.

3. *Canyon* está numa versão inicial de especificação do seu projeto, e com certeza ainda passará por diversas modificações até que possa atingir um nível satisfatório e compatível com a realidade atual dos jogos. Mas, já apresenta idéias que, após aprimoradas, poderão torná-lo um jogo de boa qualidade.

Apêndice B

Padronização e documentação de código para o *FORGE V8*

Neste apêndice são apresentados os requisitos de padronização e documentação de código exigidos ao *FORGE V8*. Esse processo teve o objetivo de melhorar a legibilidade e manutenção do código produzido, de forma a aumentar a qualidade do processo de codificação.

B.1 Convenções de nomenclatura

Nesta seção são apresentadas as convenções para definição de código utilizadas pelo *FORGE V8*.

B.1.1 Notação

Para a codificação do *framework*, foi utilizada a notação *húngara*, que é bastante conhecida e responsável por toda a nomenclatura de código de aplicações *Windows*. Esta notação adiciona um prefixo as variáveis com o seu tipo, ajudando a identificação do formato e utilização de cada variável pelas funções. Além disso, ela se estende para a nomenclatura de métodos/procedimentos/funções, parâmetros, classes, interfaces, etc. Na tabela B.1 são mostrados os prefixos usados por esta notação.

Tabela B.1: Notação húngara

Prefixo	**Tipo de dados**	**Prefixo**	**Tipo de dados**
b	boolean	by	byte ou unsigned char
c	char	f	float
cx/cy	short - usado como tamanho	dw	DWORD
fn	Função	h	handle
i	int	l	long
n	short int	p	ponteiro
str	string	sz	string terminada com nulo
w	word	x,y,z	short - usado como coordenadas
ui	unsigned int	hr	HRESULT
guid	GUID	id	DPID
ll	longlong	d	double

B.1.2 Definição de classes

Para a nomenclatura de classes de objetos, devem ser obedecidas as seguintes regras:

- Usar o prefixo C para designar o tipo <<classe>> seguido por nomes e frases nominais
- O nome da classe deve sempre ser iniciado com letra maiúscula
- Usar abreviações de fácil entendimento em nomes de classes extensas
- Não usar sublinhado
- Usar *PascalCasing*[1]
- Exemplos

  CApplication, CFullScreenWindow, CGameObjectNPC, CStreamingSound

B.1.3 Definição de interfaces

As interfaces devem seguir o padrão dado pelas seguintes regras:

- Usar o prefixo I para designar o tipo <<interface>>
- Usar abreviações de fácil entendimento em nomes de classes extensas

[1]Separação de nomes através de letras maiúsculas e minúsculas

- Não usar sublinhado
- Usar *PascalCasing*
- Nomear as interfaces através de nomes, frases nominais ou adjetivos que descrevam seu comportamento
- Usar nomes similares quando especificado um par classe/interface, em que a classe seja o padrão de implementação da interface:

  public interface IComponent {}
  public class Component: IComponent {}

- Exemplos

  IComponent, ICustomAttributeProvider, IPersistable

B.1.4 Definição de variáveis

Como explicado anteriormente, os identificadores devem seguir o padrão de convenção de nomes da notação *húngara*, assim, sendo estes identificadores formados pelas seguintes regras:

- Variáveis são sempre iniciadas por letras minúsculas
- Toda variável deve possuir um prefixo que designa seu tipo (ver tabela B.1)
- Todo prefixo é seguido por um qualificador
- Usar *PascalCasing*
- Exemplos

  pApplication, strObjectName, iSurfaceIndex

Variáveis globais

- Variáveis globais internas a um módulo ou classe iniciam com o prefixo m_ seguido pelo prefixo do tipo e qualificador:

  m_szFileName, m_iCurrentPlayState, m_pMediaControl, m_pBasicAudio

- Variáveis globais da aplicação iniciam com o prefixo g_ seguido pelo prefixo do tipo e qualificador:

  g_szFileName, g_iCurrentPlayState, g_pMediaControl, g_pBasicAudio

Variáveis locais

- Variáveis locais, internas aos métodos, iniciam com o prefixo l_ seguido pelo prefixo do tipo e qualificador:

  l_szFileName, l_iCurrentPlayState, l_pMediaControl, l_pBasicAudio

Variáveis estáticas

- Segue o mesmo padrão estabelecido para variáveis globais e locais:

  static CApplication* m_pGlobalApplication
  static CInputLayer* m_pGlobalInputLayer

B.1.5 Constantes

As constantes devem seguir o padrão dado pelas seguintes regras:

- Os nomes de constantes devem ser especificados através de letras maiúsculas
- Usar sublinhado para separar nomes
- Nomear as constantes através de nomes que descrevam seu objetivo
- Exemplos

  const int DEFAULT_SCREEN_WIDTH = 640;
  const int DEFAULT_SCREEN_HEIGHT = 480;
  const int DEFAULT_SCREEN_BPP = 16;

B.1.6 Enumerações

As enumerações devem seguir o padrão dado pelas seguintes regras:

- Os nomes devem ser em letras maiúsculas
- Usar abreviações de fácil entendimento para nomes extensos
- Usar sublinhado entre nomes
- Nomear as enumerações através de nomes que descrevam seu comportamento
- As constantes das enumerações devem ser atribuídas de novos valores, e não utilizar valores *default*

- Exemplo

  enum FILE_TYPES_SET {
  FILE_TYPES_WORLD = 1,
  FILE_TYPES_MESH = 2,
  FILE_TYPES_IMAGE = 4,
  FILE_TYPES_SOUND = 8,
  FILE_TYPES_MUSIC = 16,
  FILE_TYPES_MEDIA = 32
  };

B.1.7 Métodos/procedimentos/funções

Os métodos/procedimentos/funções devem seguir o padrão dado pelas seguintes regras:

- Nomear os métodos através de verbos ou frases verbais
- Os nomes dos métodos devem sempre iniciar com letra maiúsculas
- Usar abreviações de fácil entendimento para nomes extensos
- Usar *PascalCasing*
- Exemplos

  InitBuffer, FillBuffer, ReadRegisterKey, WriteRegisterKey

B.1.8 Parâmetros de métodos

Segue o mesmo padrão de variáveis.

B.2 Organização e apresentação dos arquivos

Para uma boa organização do código fonte do *FORGE V8*, foram exigidas uma série de considerações a respeito da estruturação dos seus arquivos.

B.2.1 Convenções gerais

- Evitar inserir comentários no final das linhas de código, no entanto, em caso necessário e apropriado, fazê-lo com uma indentação padrão;

- Evitar adicionar comentários supérfluos e ambíguos;
- É importante comentar de forma bem explicativa as estruturas de códigos, principalmente as que representam laços, pois geralmente essas estruturas são bastante importantes para a lógica da aplicação;
- Todos os comentários devem seguir a padronização utilizada pelo gerador de documentação *Doc++* [14], facilitando a geração da documentação final.

B.2.2 Comentários de classes

Todas as classes devem possuir no início de suas especificações a seguinte descrição:

```
///----------------------------------------------------------
/// Class: Nome da classe
/// Desc: Descrição da sua funcionalidade
///----------------------------------------------------------
```

B.2.3 Comentários de variáveis/constantes/enumerações

Preferencialmente, os comentários devem estar posicionados logo acima das variáveis/constantes/enu
Quando necessário posicionar os comentários ao lado dessas estruturas, fazê-lo com uma indentação padrão.

Comentários sobre variáveis devem seguir o padrão:

```
/// comentário sobre a variável
variável
```

B.2.4 Comentários de métodos/procedimentos/funções

Comentários de métodos/procedimentos/funções devem seguir o padrão a seguir:

```
///----------------------------------------------------------
/// Name: Nome do método ()
/// Desc: Descrição da sua funcionalidade
///----------------------------------------------------------
```

Comentários internos aos métodos/procedimentos/funções devem seguir o padrão:

```
/// comentário sobre a implementação
implementação
```

B.2.5 Tipos de arquivos

Os arquivos em C++ são divididos entre arquivos de cabeçalho (definem a estruturação das classes) e arquivos de implementação (implementam os métodos). Estes arquivos devem apresentar um cabeçalho, incluindo o seguinte comentário, antes de qualquer declaração:

/* **
File: Nome do arquivo.extensão
Desc: Descrição do seu objetivo
** */

B.2.5.1 Cabeçalhos (.h)

Os arquivos de cabeçalhos devem possuir uma descrição bem explicativa dos objetivos das classes e uma descrição resumida de seus métodos. Eles devem seguir o padrão:

/* **
Name: Class Nome da classe
Desc: Descrição bem explicativa dos objetivos

Método1() - descrição
Método2() - descrição
.....
MétodoN() - descrição
** */

B.2.5.2 Implementações (.cpp)

Os arquivos de implementações seguem o padrão de comentários de métodos, descrevendo as implementações das classes.

B.2.6 Indentação

A indentação de todo o código deve seguir o padrão de espaçamento 4 (<tab>)

Referências Bibliográficas

[1] Abrash, M., *It's Great to be Back! - Fast Code, Game Programming, and other Thoughts from 20 (Minus 2) Years in the Trenches*. Disponível em http://www.gamasutra.com/features/20010117/abrash_01.htm, 2001.

[2] Bäumer, Dirk, et all, *Framework Development for Large Systems*, Communications of the ACM, Vol.40, N^{o}10, October, 1997.

[3] Beck, K., Johnson, R., *Patterns Generate Architectures*, ECOOP, 1994.

[4] Ben-Natan R., *CORBA on the Web*, McGraw-Hill, United States, 1998.

[5] Bernt Habermeier, *Internet Based Client/Server Network Traffic Reduction*. Disponível em http://www.bolt-action.com, 2000.

[6] Bigus, J. P., Bigus, J., *Constructing Intelligent Agents with Java: a Programmer's Guide to Smarter Applications*, United States, John Wiley & Sons Inc., 1997.

[7] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[8] Castek, Johns E., *Framework, Frameworks, Everywhere...*, Castek Software Factory. Disponível http://www.cbd-hq.com, 2000.

[9] Castek, Johns E., *Framework Evolution! One Box, Two Box, White Box, Black Box*, Castek Software Factory. Disponível http://www.cbd-hq.com, 2000.

[10] Castek, Johns E., *Framework Architectures! Patterns usage for sucess*, Castek Software Factory. Disponível http://www.cbd-hq.com, 2000.

[11] Coulouris, G., Dollimore, J. & Kindberg, T., *Distributed Systems: Concepts and Design*, Second Edition, Addison-Wesley Pub. Co., 1995.

[12] *Crystal Space*. Disponível em http://crystal.linuxgames.com, 2001.

[13] *API Microsoft DirectX*. Disponível em http://www.microsoft.com/directx, 2001.

[14] *Doc++*. Disponível em http://www.doc++.com, 2000.

[15] D'Souza, D., Wills, A., *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1999.

[16] Eberly, D. H., *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann Publishers, 2001.

[17] Fayad, M., Schmidt, D., *Object-Oriented Application Frameworks*, Communications of the ACM, Vol.40, N^{o}10, October, 1997.

[18] Fayad, M., Schmidt, D., Johnson, R., *Object-Oriented Application Frameworks: Problems and Perspectives*, Wiley, New York, 1997.

[19] Figueira, C., JEOPS - *The Java Embedded Object Production System*. Disponível em http://www.di.ufpe.br/∼csff/jeops, UFPE, 2000.

[20] *Gamasutra*. Disponível em http://www.gamasutra.com, 2001.

[21] *Gamedev*. Disponível em http://www.gamedev.net, 2001.

[22] *Gamedev Patterns*. Disponível em http://www.gamedev.net/gdpatterns/patterns.asp, 2001.

[23] Gamma, Erich, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[24] *Genesis3D*. Disponível em http://www.genesis3d.com, 2001.

[25] *Game Developer Magazine*. Disponível em http://www.gdmag.com, 2001.

[26] *Golgotha*. Disponível em http://www.planetquake.com/golgotha, 2001.

[27] Hajnoczi, S., *WinSock2 for Games*. Disponível em http://www.gamedev.net/reference/articles/article1059.asp, 2000.

[28] Hendricks, S., *IDSA Report Video and Computer Games Industry's $16 Billion Contribution to the U.S. Economy*. Interactive Digital Software Association On-line. Disponível em http://www.idsa.com/releases/econ.html, 1998.

[29] Horstmann, C., Cornell, G., *Core Java 2 - Volume I - Fundamentals*, Sun Microsystems Inc., California, 1999.

[30] Howland, G., *A Practical Guide to Building a Complete Game AI: Volume II.* Lupine Games. Disponível em http://www.lupinegames.com/articles/prac_ai_2.html, 2000.

[31] Howland, G., *A Practical Guide to Building a Complete Game AI: Volume I.* Lupine Games. Disponível em http://www.lupinegames.com/articles/prac_ai.html, 2000.

[32] Iqbal, G., *Using Interfaces with Dlls.* Disponível em http://www.gamedev.net/reference/articles/article928.asp, 2000.

[33] Battaiola, A., *Jogos por Computador - Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação*, Anais da XIX Jornada de Atualização em Informática (JAI) - Volume 2, pág.83 à 122, SBC 2000.

[34] Jain, R., *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design*, Measurement, Simulation and Modeling, John Wiley & Sons Inc., 1991.

[35] *Java3D.* Disponível em http://www.sun.com/java3d, 2000.

[36] Johnson, Ralph E., *Documenting Frameworks using Patterns*, OOPSLA, 1992.

[37] Johnson, Ralph E., *Frameworks = (Components + Patterns)*, Communications of the ACM, Vol.40, Nº10, October, 1997.

[38] Kovach, Peter J., *Inside Direct3D*, Microsoft Press, 2000.

[39] Kovach, Peter, *3D Game Programming With Direct3D.* Game Institute Inc., 2001.

[40] Lajoie, R., Keller, R., *Design and Reuse in Object-Oriented Frameworks: Petterns, Contracts, and Motifs in Concert*, Proceedings of the 62nd Congress of the Association Cannadienne Française pour l'Avancement des Sciences (ACFAS), Montreal, Canada, 1994.

[41] Lamothe, A., *Tricks of the Windows Game Programming Gurus - Fundamentals of 2D and 3D Game Programming*, Sams, United States, 1999.

[42] Leite, J. C., Souza C. S., *Visão Geral do Projeto de Interfaces de Usuário*, UFRN, 1997.

[43] *LucasArts Entertainment Company LLC.* Disponível em http://www.lucasarts.com, 2001.

[44] Hodorowicz, L., *Elements Of A Game Engine.* Disponível em http://www.flipcode.com, 2001.

[45] Macedo, H., Araujo, A., Cavalcanti, D., Andrade, R., Madeira, C., Ferraz, C., *Evaluating Multi User Distributed Action Games Architectures on a CORBA Platform*, Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), p.1787-1792, Monte Carlo Resort, Las Vegas, Nevada, USA, June 26 - 29, 2000.

[46] Madeira, C., Araujo, A., Macedo, H., Andrade, R., Cavalcanti, D., Ferraz, C., Ramalho, G., *NetMaze: Um Jogo de Ação Multimídia Distribuído*, Anais do VI Simpósio Brasileiro de Sistemas Multimidia e Hipermídia (SBMídia 2000), p.129-139, Centro de Convenções, Natal/RN, 14 a 16 de Junho, 2000.

[47] Madeira, C., Almeida, A., Filho, M., Silva, D., Ramalho, G., Ferraz, C., *Framework e Padrões de Projeto para o desenvolvimento de Jogos de Computador*, The First Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2001), Rio de Janeiro/RJ, 3 a 5 de outubro, 2001.

[48] Madeira, C., Mauro, Talita, Silva, D., Ramalho, G., Ferraz, C., *FORGE V8: Um Framework para Jogos de Computador e Aplicações Multimídia*, VII Simpósio Brasileiro de Sistemas Multimidia e Hipermídia (SBMídia 2001), Florianópolis/SC, 15 a 18 de outubro, 2001.

[49] Meyer, B., *Object-Oriented Software Construction*, ISE Inc., California, USA, 1997.

[50] Myers, B. A., Rosson, M. B., *Survey on User Interface Programming*, IBM Research Report, RC-17624, 1992.

[51] *API OpenAL.* Disponível em http://www.openal.org, 2001.

[52] *API OpenGL.* Disponível em http://www.opengl.org, 2001.

[53] Orfali, R., Harkey, D., *Client/Server Programming with Java and CORBA*, John Wiley & Sons Inc., United States, 1998.

[54] Petzold, Charles, *Programming Windows, The Definitive Guide to the Win32 API*, Microsoft Press, 1998.

[55] Perez, A., *Advanced 3-D Game Programming Using DirectX 7.0* Wordware Publishing Inc, 2000.

[56] Pottinger, D., *Coordinated Unit Movement*, Game Developer Magazine, January 1999.

[57] Pottinger, D., *Implementing Coordinated Movement*, Game Developer Magazine, February 1999.

[58] Posnak, E., Lavender, R., Vin, H., *An Adaptive Framework for Developing Multimedia Software Components*, Communications of the ACM, Vol.40, N^{o}10, October, 1997.

[59] *PowerPlay - Enabling the Future of Entertainment on the Internet*. Disponível em http://www.powerplayinfo.com, 2001.

[60] Ramalho, G., *Projeto e Implementação de Jogos - CIn/UFPE*. Disponível em http://www.cin.ufpe.br/∼compint/jogos.html, 2001.

[61] Reynolds, C., *Steering Behaviors For Autonomous Characters*, Sony Computer Entertainment America, Disponível em http://www.red.com/cwr, 2000.

[62] Roberts, D., Johnson, R., *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, Proceedings of the PLoPD 1998, 1998.

[63] Rogerson, Dale E., *Inside COM (Programming Series)*, Microsoft Press, 1997.

[64] *Roger Wilco - Free Voice Chat for Games*. Disponível em http://www.rogerwilco.com, 2001.

[65] Rollings A., Morris D., *Game Architecture and Design*, The Coriolis Group LLC, 2000.

[66] Russel, S., Norvig, P., *Artificial Intelligence: A Modern Approach*, New Jersey, Prentice-Hall Inc., 1995.

[67] Salkin, S., *Design Patterns for Game Development*, Game Developer Magazine, July, 1998.

[68] Schmidt, D., Johnson, R., Fayad, M., *Software Patterns*, Communications of the ACM, Vol.39, N^{o}10, October, 1996.

[69] Schmidt, H., *Systematic Framework Design by Generalization*, Communications of the ACM, Vol.40, N^{o}10, October, 1997.

[70] Simmons, G., Hoult, A., *Advanced BSP*. Game Institute Inc., 2001.

[71] Siqueira, F., *A Framework for Distributed Multimedia Applications based on CORBA and Integrated Services Networks*. Disponível em http://www.cs.tcd.ie/Frank.Siqueira/PhD-Project/index.html, 2000.

[72] Sommerville, I., *Software Engineering*, Sixth Edition, Addison-Wesley, 2000.

[73] Stout, B., *Smart Moves: Inteligent Path-Finding*, Game Developer Magazine, October, 1996.

[74] *TCPDUMP* - Lawrence Berkeley National Laboratory, Network Research Group. Disponível em ftp://ftp.ee.lbl.gov/tcpdump.tar.Z, 2000.

[75] Tirakis, A. et al, *Distributed Multimedia Architectures - State-of-the-Art Report*. Disponível em http://viswiz.gmd.de/DVP/Public/deliv/deliv.222/act222.htm, 2000.

[76] Turner, Bryan, *Real-Time Dynamic Level of Detail Terrain Rendering*. Disponível em http://www.gamasutra.com/features/20000403/turner_01.htm, 2000.

[77] Turner, Bryan, *3D Graphics With OpenGL*. Game Institute Inc., 2001.

[78] *VisiBroker - Programmer's Guide*, Inprise Corporation Inc. Disponível em http://www.borland.com/techpubs/books/vbj/vbj40, 1999.

[79] Wolverson, H., *Using Direct3D For 2D Tile Rendering*, Disponível em http://www.gamedev.net/reference/articles/article935.asp, 2000.

[80] Woodcock, S., *Game AI: The State of the Industry*. Game Developer Magazine, November, 2000.