Universidade Federal de Pernambuco

Centro de Informática

Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

Forge 16V: Um Framework para Desenvolvimento de Jogos Isométricos

Eduardo José Torres Sampaio Rocha

Recife,
Dezembro de 2003

Eduardo José Torres Sampaio Rocha

Forge 16V: *Um Framework para Desenvolvimento de Jogos Isométricos*

Dissertação apresentada à Coordenação da Pós-Graduação em Ciências da Computação do Centro de Informática, como parte dos requisitos para obtenção do título de Mestre em Ciências da Computação.

Orientador: Geber Lisboa Ramalho
Co-orientador: André Luís. M. Santos

Recife,
Dezembro 2003

AGRADECIMENTOS

Muito obrigado a Deus por ter-me dado condições e força para a realização deste trabalho.

Muito obrigado aos meus pais, Pedro e Sally, pelo apoio, dedicação, amizade e incentivo que me deram para chegar até aqui.

Muito obrigado a minha tia Suzy pelo apoio e amizade.

Muito obrigado a Talita por ter acompanhado todo o trabalho com paciência e por ter-me prestado grande ajuda nos momentos difíceis.

Muito obrigado ao meu professor e orientador Geber Ramalho pelo apoio, confiança, dedicação, orientação e, acima de tudo, pela amizade.

Muito obrigado ao meu professor e co-orientador André Santos pelo apoio dado durante boa parte da minha vida acadêmica e pela amizade.

Muito obrigado a André Furtado, André Amaral, Gustavo Andrade e Fernando Andrade pela implementação do jogo *Knock'em*.

Muito obrigado a Dante Torres, Fabiano Rolim, Marília Gama e Pablo Sampaio pela implementação do jogo *CInFarm*.

Muito obrigado a Georgia Albuquerque, Ivo Nascimento, Mauro Faccenda, Rodrigo Santos, Saulo Jansen e Tiago Barros pela implementação do jogo *Pirralhos*.

Muito obrigado aos funcionários e professores do Centro de Informática que, de alguma forma, contribuíram para a minha formação acadêmica.

Muito obrigado a todos que contribuíram em geral para a realização deste trabalho.

RESUMO

Nos últimos anos, com o aumento do poder computacional, a indústria de jogos passou a desenvolver projetos cada vez mais complexos que demandam um investimento cada vez maior. Para diminuir os riscos, essa indústria está crescentemente utilizando técnicas e metodologias da engenharia de software, em particular *frameworks,* chamados de motores de jogos. Entretanto, por questões de sigilo industrial, existe pouca documentação e compreensão sobre esses *frameworks* de jogos no meio acadêmico. Um dos esforços acadêmicos pioneiros a esse respeito foi a implementação do *Forge V8*, um framework para jogos e aplicações multimídia desenvolvido no Centro de Informática da UFPE. Contudo, como a maioria dos projetos pioneiros, alguns erros de projeto e implementação foram cometidos, o que praticamente inviabilizou seu uso prático em produtos. Por isso, projetamos e implementamos um novo *framework*, denominado *Forge 16V*, voltado para jogos em cenários isométricos. O *Forge 16V* foi testado por terceiros para o desenvolvimento de 3 protótipos de jogos, apresentando desempenho e facilidade de reuso satisfatórios. Além da implementação propriamente dita, foi dada continuidade ao estudo sobre *frameworks* e padrões de projeto no desenvolvimento de jogos iniciado no *Forge V8*.

ABSTRACT

In recent years, with the increasing computing power, the industry of computer games started to develop even more complex projects, demanding increasing investments. To lower risks, this industry is increasingly using software engineering techniques and methodologies, especially frameworks, called game engines. However, because of the game industry's secrecy about the implementation of game engines, there is very little documentation available and knowledge about these frameworks in academia. One of the pioneer academic projects related to this subject was the implementation of Forge V8, a framework for game and multimedia applications, developed at the Informatics Center of the Federal University of Pernambuco. As in the majority of the pioneer projects, some mistakes have been made, making difficult its practical adoption in commercial grade games. These problems lead to the implementation of a new framework, called Forge 16V, devoted to isometric scenarios. Forge 16V has been tested by third parties for the development of three game prototypes, showing good performance and reuse level. Apart from the implementation, the studies about frameworks and design patterns applied to game development, which was started with *Forge V8*, were continued and extended in this work.

.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

Neste capítulo serão apresentadas as principais motivações para realização deste trabalho. Serão também mostrados os objetivos e a estrutura deste trabalho.

Neste capítulo serão expostos as principais motivações e o contexto que levaram à realização deste trabalho, bem como os objetivos almejados. Por fim, será apresentada a estrutura do trabalho.

1.1 PESQUISA EM DESENVOLVIMENTO DE JOGOS

Existe uma vasta área de pesquisa em desenvolvimento de jogos. Mesmo para os jogos mais simples, o desenvolvimento dessas aplicações envolve a utilização de conceitos de várias áreas da Ciência da Computação, tais como Engenharia de Software, Computação Gráfica, Inteligência Artificial, Redes de Computadores e Computação Musical. Além disso, outras áreas como Educação, Psicologia, Artes e Estratégia Militar, também são envolvidas.

Apesar desta riqueza de desafios, somente há alguns anos o meio acadêmico começou a envolver-se diretamente e objetivamente com a área de jogos. Uma das principais dificuldades, além de um eventual preconceito da academia, é o sigilo industrial. Devido às grandes cifras que giram em torno dos projetos, muitas das empresas não divulgam seus métodos, arquiteturas e ferramentas, com medo que os resultados obtidos sejam usados por empresas concorrentes. Sem uma divulgação adequada, a pesquisa e o desenvolvimento dentro das universidades andam mais lentamente.

Mesmo com essas dificuldades, hoje já existem algumas universidades que dispõem de disciplinas ou mesmo cursos inteiros na área de jogos [1], inclusive no Brasil. Além disso, já existem várias revistas (e.g. *Game Developer Magazine* [2]) e congressos (e.g. WJogos [3] e *GameOn* [4]) que tratam especificamente do assunto.

No Centro de Informática (CIn) da UFPE já existem duas disciplinas na área de jogos: Projeto e Implementação de Jogos [5] e Jogos Avançados. Também existe um grupo de pesquisa que há cerca de cinco anos realiza pesquisas sobre o assunto, produzindo inclusive algumas dissertações de mestrado [6-10] e publicações.

No Recife também já existem algumas empresas envolvidas com jogos de computador, tais como, *Jynx Playware* [11] e Meantime/CESAR. Foi nesse contexto de dar suporte à criação de um pólo de desenvolvimento de jogos em Pernambuco e de ajudar nas

disciplinas existentes na área, que o CIn tem se dedicado a desenvolver ferramentas de código aberto que facilitem o desenvolvimento de jogos de computador.

1.2 FORGE V8: HISTÓRICO

Na década de oitenta, o baixo poder computacional das máquinas obrigava que os jogos fossem desenvolvidos em linguagens próximas da linguagem de máquina. Nessa época não existia nenhum processo para o desenvolvimento de um jogo. Somente em meados da década de noventa, essa realidade começou a mudar. Com o aumento do poder computacional dos computadores pessoais, a indústria de jogos começou a lançar jogos cada vez mais realistas e conseqüentemente mais complexos de serem desenvolvidos. Os altos orçamentos requeridos pelos projetos [12] levaram a indústria de jogos a investir pesadamente em pesquisa, e a utilizar conceitos que no restante da indústria de *software*.já, estavam consolidados havia algum tempo. A engenharia de *software* passou a ser fortemente usada nos projetos para diminuir o tempo de desenvolvimento e os riscos de falha inerentes às aplicações complexas.

Nesse contexto, o conceito de *frameworks* tornou-se chave no processo de desenvolvimento de um jogo. Como boa parte do código desenvolvido se repete de um jogo para o outro, o uso de um *framework* passou a ser muito comum nos projetos. Sendo desenvolvidos e testados por especialistas na área, os *motores* (como ficaram conhecidos os *frameworks* para o desenvolvimento de jogos) possibilitaram a diminuição do risco dos projetos darem errado, além de possibilitarem o desenvolvimento de mais de um jogo ao mesmo tempo. Outro ponto importante é que, com o uso de motores, é possível dedicar um maior tempo do desenvolvimento à lógica do jogo, visto que um *framework* abstrai grande parte dos detalhes de implementação de baixo nível.

Vendo a necessidade e a importância dos motores no processo de desenvolvimento de jogos, no início de 2000, o grupo de jogos do CIn da UFPE resolveu realizar uma pesquisa sobre o assunto . Devido aos altos custos dos motores comerciais (e.g. US$ 350.000 pelo *Unreal* da *Epic Games* [13]), o grupo dedicou-se então a analisar motores de código aberto. Após alguns meses de pesquisa e a análise de alguns motores (e.g. *Golgotha* [14] e *Crystal Space 3D* [15]), constatou-se na época a falta de maturidade desses projetos

[7] (pouca documentação e nenhum compromisso de compatibilidade com versões anteriores). Sentindo a crescente necessidade de dominar o processo de desenvolvimento de um jogo, decidiu-se então desenvolver um motor próprio.

O *Forge V8* (nome dado a este motor), além de ser um trabalho pioneiro na UFPE, possibilitaria acima de tudo o entendimento e domínio das tecnologias envolvidas no desenvolvimento de jogos. O principal objetivo do projeto era construir um motor que utilizasse a renderização gráfica 3D, possibilitando assim o desenvolvimento de jogos 3D, isométricos e 2D. Além disso, o motor deveria possibilitar que os jogos desenvolvidos pudessem rodar em várias plataformas (e.g. Windows, Linux, etc).

Devido à falta de experiência do grupo neste tipo de projeto na época, os objetivos traçados para o *Forge V8* foram superestimados, e o esforço para atingi-los subestimado. Dispondo de apenas quatro programadores, onde nenhum deles era especialista em computação gráfica tridimensional, o resultado foi um motor com uma arquitetura complexa, que tinha sérios problemas de performance e que no final ficou longe de ser terminado. Contudo, a avaliação do projeto foi positiva, pois realizou um grande estudo sobre os *frameworks* para o desenvolvimento de jogos e catalogou alguns dos principais problemas encontrados neste tipo de aplicação [7].

1.3 OBJETIVOS

Reaproveitando o esforço realizado e a experiência adquirida, e ainda dispondo da necessidade de um motor para a utilização nas disciplinas de jogos do Centro de Informática da UFPE, assim como para facilitar o nascimento de empresas do ramo no Recife, resolveu-se então criar um novo projeto, denominado de *Forge 16V,* que foi desenvolvido como parte deste trabalho de mestrado.

A principal meta traçada para o projeto do *Forge 16V* foi a de possuir um motor de jogos em estado funcional o mais rápido possível. Para conseguir isto, resolveu-se então reduzir as funcionalidades a serem desenvolvidas no projeto em relação ao *Forge V8*:

- o motor deve possibilitar o desenvolvimento de jogos com cenário 2D e isométrico (sem utilizar a tecnologia de renderização 3D);
- os jogos produzidos devem rodar apenas no Windows.

Além do motor em si, outro objetivo deste trabalho é o de continuar o trabalho iniciado no *Forge V8* de catalogar os principais problemas e as respectivas soluções em forma de padrões de projeto, encontrados no desenvolvimento deste tipo de aplicação.

1.4 ESTRUTURA DA DISSERTAÇÃO

Os assuntos abordados nesta dissertação foram divididos em sete capítulos. O capítulo 2 apresenta a importância da utilização de um *framework* no desenvolvimento de jogos. Para entender melhor o ambiente e a mentalidade existentes hoje na indústria de jogos, nesse capítulo também será apresentada evolução do processo de desenvolvimento desde a década de 80 até os dias atuais. Por fim, serão apresentadas as principais tecnologias envolvidas no desenvolvimento de um jogo, com o objetivo de facilitar o entendimento das escolhas e problemas enfrentados no projeto do *Forge 16V*.

O capítulo 3 mostra o *Forge V8* de maneira mais detalhada, apresentando os motivos que levaram ao desenvolvimento deste motor, e as principais virtudes e defeitos. Esse capítulo é importante para entender as razões que levaram à criação de um novo projeto, o *Forge 16V*, e a diferença de objetivos entre os dois.

O capítulo 4 apresenta o estudo realizado sobre os jogos isométricos, que foi de grande importância no desenvolvimento do *Forge 16V*.

O capítulo 5 expõe o *Forge 16V*, apresentando a sua arquitetura, principais módulos e as escolhas tomadas para conseguir os objetivos traçados. Além disso, esse capítulo também apresenta os principais problemas encontrados e respectivas soluções em forma de padrão de projeto, continuando assim o trabalho iniciado no *Forge V8* de criar um catálogo dos principais problemas encontrados nesse tipo de aplicação.

O capítulo 6 descreve os detalhes de implementação de cada módulo do motor que foi implementado. Também é mostrada a validação do *framework* através de três estudos de caso.

Por fim, no capítulo 7 encontram-se as conclusões, contribuições e possíveis trabalhos futuros.

Capítulo 2

Motores para Jogos

Neste capítulo será mostrada a importância de um framework no desenvolvimento de um jogo.

O objetivo principal deste capítulo é mostrar a necessidade da utilização de um *framework* no desenvolvimento de jogos de computador. Para entender melhor o estado da arte, será apresentada a evolução histórica do processo de desenvolvimento, enfatizando a mudança de mentalidade dos programadores desde a década de 80 aos dias atuais. O orçamento, a composição da equipe de desenvolvimento e as principais tecnologias envolvidas atualmente no projeto de um jogo também serão abordados. Por fim, um estudo dos *frameworks* aplicados no desenvolvimento de jogos, será apresentado.

2.1 HISTÓRIA DO DESENVOLVIMENTO DE JOGOS

Desde o início do desenvolvimento de jogos de computadores, o caminho trilhado pelos desenvolvedores de jogos foi diferente do desenvolvimento de software em geral. Enquanto a maioria dos softwares começou a ser desenvolvido em grandes *mainframes* com uma enorme quantidade de recursos (pelo menos para a época), o desenvolvimento de jogos era realizado com computadores pequenos, que tinham grandes limitações.

Na década de oitenta, os jogos eram desenvolvidos para computadores que tinham um processador de 8 bits rodando a aproximadamente 4 MHz e com 48 KB a 64 KB de memória [16]. Programar para esta plataforma envolvia uma eterna procura de superar as limitações físicas do *hardware* para se obter um melhor resultado. Velocidade e tamanho de código eram prioridades, obrigando a uma forte procura por otimizações no código.

Contudo, na época o *hardware* era bem definido. Isso significa que, tirando algumas raras exceções, a máquina que o usuário tinha em casa era a mesma máquina que o desenvolvedor tinha, similar ao que ocorre com os consoles nos dias de hoje. Isso evitava que o jogo produzido rodasse mais lento ou até mesmo não rodasse na casa do usuário final, caso a configuração da máquina utilizada no desenvolvimento fosse melhor que a do usuário final.

Nessa época, todos os jogos tinham que ser codificados em *Assembly*, pois não existiam bons compiladores C que gerassem um executável pequeno o suficiente para um jogo. Isso significa que o código não era portável entre diferetes plataformas. Além disso, depurar um código era um problema, pois geralmente o *debugger* não cabia na memória junto com o jogo [16].

Por mais de uma década, os programadores de jogos ficaram trabalhando em um ambiente onde o código mais rápido e o de menor tamanho eram o principal objetivo, e onde o programador tinha total controle de todo o código que estava rodando na máquina. Isso serviu para solidificar a mentalidade entre os programadores de jogos de que o código desenvolvido por terceiros é mais lento e não presta [16], indo completamente de encontro ao conceito de reusabilidade de *software*. Essa mentalidade ficou conhecida como NBH (*not built here*), e serviu como uma barreira na aplicação de conceitos de engenharia de *software* no desenvolvimento dos jogos durante muito anos.

Outra mentalidade que foi bastante difundida nesse período e que também ia de encontro aos conceitos da engenharia de *software* foi a que ficou conhecida como "*pedal to the metal*". Ela dizia que aplicações desenvolvidas em linguagens de baixo nível, como *Assembly* por exemplo, eram mais rápidas e menores e isso criou uma certa aversão às linguagens que não estavam próximas da linguagem de máquina.

Além disso, não existia nenhum processo de desenvolvimento de *software*. Para se ter uma idéia, a seguinte receita descreve como as equipes de desenvolvimento de jogos eram formadas e conduzidas na década de 80 [16]:

1. Encontre 5 programadores (Geral, IA, Gráficos, Som, etc.);
2. Eleja um *hacker* como líder;
3. Coloque-os numa sala pequena com alguns artistas à disposição;
4. Deixe cozinhar por 18 meses regando-os a pizza e coca-cola;
5. Dê um pouco mais de cozimento, e .... pronto!

O uso da linguagem C no desenvolvimento de jogos somente começou a acontecer a partir de 1993, quando a ID Software [17] lançou o jogo *Doom* (ver a Figura 2-1). Nessa época, os PCs já estavam populares, o *hardware* não era mais tão bem definido e, embora o Windows 3.1 já tivesse sido lançado, os jogos ainda eram programados para o DOS, pois o Windows fazia com que os jogos ficassem muito lentos. *Doom* [18]foi realmente um divisor de águas no desenvolvimento de jogos. Ele utilizou a tecnologia do compilador 32-bits da Watcom para DOS, o que acabou com a limitação dos 640 KB dos programas 16-bits. Na época, muitos programadores não acreditavam que o jogo não havia sido programado em *Assembly*, por acreditarem que nenhum compilador era capaz de gerar um código melhor que o desenvolvido por eles (mentalidade NBH).

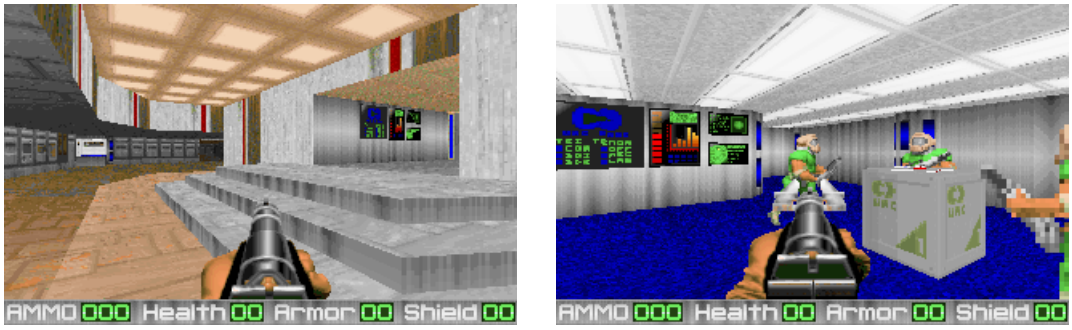

Figura 2-1 Screenshots do jogo Doom.[1]

Em 1995, a Microsoft lançou o Windows 95, um sistema operacional verdadeiramente 32-bits. Desde o Windows 3.1, a Microsoft promovia a política de abolir as aplicações para DOS. Com o Windows 95 essa política se acelerou. Todas as empresas de *softwares* passaram então a investir pesadamente em aplicações gráficas 32-bits, exceto a indústria de jogos que praticamente ignorou esse sistema operacional. O problema era que desenvolver uma aplicação para Windows implicava em não ter um acesso direto ao *hardware*, o que levava a uma performance sofrível dos jogos.

Tentando atrair a indústria de jogos, a Microsoft lançou a biblioteca DirectX [19] que permitia o desenvolvedor de jogos acessar diretamente o *hardware* sem a interferência do Windows e tirar proveito de qualquer aceleração que o *hardware* suportasse. Além disso, a biblioteca dava suporte a vários dispositivos (placas gráficas, placas de som, etc.), o que permitia que o desenvolvedor programasse sem se preocupar em produzir vários *drivers* para garantir uma maior compatibilidade do jogo. Pouco tempo depois, uma nova versão do Visual C++ [20] que possuía um compilador capaz de gerar um código menor e mais otimizado [16], foi lançado. Com tantos atrativos, aos poucos, a indústria de jogos começou a migrar das aplicações DOS para Windows 95.

Com o aumento do poder computacional das máquinas e com a existência de ferramentas que geram executáveis cada vez mais otimizados, a qualidade dos jogos produzidos após 1996 aumentou consideravelmente. Os programadores deixaram de se

[1] Figuras retiradas do Site http://www.rome.ro/lee_killough/

preocupar com detalhes de implementação de *drivers* para se preocupar com temas mais abstratos como Inteligência Artificial e Computação Gráfica. Aos poucos a mentalidade do NBH (*not built here*) e *pedal to the metal*, que por tanto tempo rondou a indústria de jogos, foi sendo superada e esquecida.

2.2 CRESCIMENTO COMERCIAL DA INDÚSTRIA DE JOGOS

Após 1995, a indústria de jogos recebeu uma grande ajuda com o crescimento da Internet e com a globalização da economia mundial. Com o passar dos anos, a distribuição dos jogos e consoles tornou-se cada vez mais abrangente, e isso levou a indústria a faturar valores cada vez mais astronômicos. A Tabela 2-1 abaixo mostra o faturamento com *software* de entretenimento nos Estados Unidos desde 1995. Em 2002, foram vendidos 6,9 bilhões de dólares em *software* nos EUA, mais que o dobro do valor arrecadado em 1995. Deste montante, 5,5 bilhões foram em software para console, e 1,4 bilhões em *software* para computador [21].

Hoje em dia, a indústria de jogos é uma das poucas que impulsionam o lançamento de novos dispositivos, como placas de vídeos, processadores e memórias mais rápidas. Cada novo jogo vem acompanhado do suporte a novas tecnologias que foram recentemente lançadas pelos fabricantes dos *hardwares*. Uma boa estimativa para quem joga em um computador é que a cada dois anos será preciso comprar novas placas para poder rodar os lançamentos de jogos de maneira satisfatória. Em 2002, a estimativa foi que a indústria chegaria a faturar mais de 30 bilhões de dólares vendendo *hardware* e *software* em todo o mundo [22].

Um outro fato que ajuda a explicar números tão altos é que, ao contrário do que ocorre nas demais indústrias de *software*, o consumidor tende a comprar mais de um jogo por ano. Este fato não ocorre, por exemplo, com um editor de imagens ou de texto.

Ano	***Faturamento (US$)***
1995	3,2 bilhões
1996	3,7 bilhões
1997	4,4 bilhões

1998	5,5 bilhões
1999	6,1 bilhões
2000	6,0 bilhões
2001	6,35 bilhões
2002	6,9 bilhões

Tabela 2-1 Faturamento com *Software* da Indústria de Jogos nos Estados Unidos[2]

2.3 EQUIPE DE DESENVOLVIMENTO

Para atingir o patamar de faturamento mostrado na seção anterior, a indústria teve que investir fortemente em pesquisa e desenvolvimento. Isto fez com que o padrão de qualidade da indústria crescesse, juntamente com a complexidade do desenvolvimento de um jogo.

Como conseqüência imediata, a equipe envolvida na produção do jogo também cresceu. Na década de oitenta, uma equipe típica de desenvolvimento de jogos era formada por cinco a dez programadores [16] onde um era eleito como líder e o projeto levava em torno de quinze meses para ser finalizado. No período pós 1995, essa realidade mudou completamente. As equipes cresceram e passaram a incorporar pessoas de outras áreas. Em 28 de Junho de 2000 a *Blizzard Entertainment* lançou o jogo *Diablo II* (ver a Figura 2-2), um sucesso de vendas na época. Para desenvolver este jogo, foram contratadas 40 pessoas que trabalharam em regime de tempo integral divididos em três grupos de pouco mais de 12 pessoas cada [23]: programação, arte dos personagens e arte dos cenários. O projeto durou aproximadamente três anos para ser concluído.

Como outro exemplo deste crescimento, podemos citar o jogo *Neverwinter Nights* da *Bioware* lançado em Junho de 2002. Este jogo levou aproximadamente cinco anos para ser concebido e tinha em média 160 pessoas envolvidas, trabalhando em regime de tempo integral [24]. Além destes desenvolvedores, 75 pessoas foram contratadas sazonalmente para realizar testes, traduções e compor sons do jogo.

[2] Informações retiradas do Site http://www.theesa.com/pressroom.html

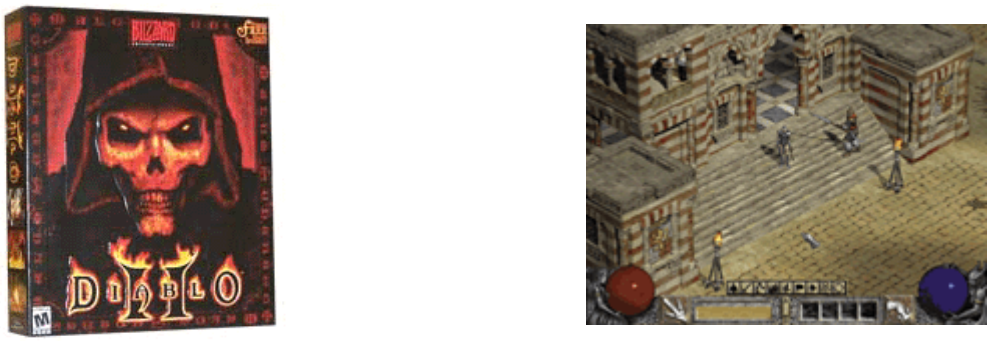

Figura 2-2 Diablo II da Blizzard Entertainment[3]

Devido ao grande custo envolvido nos projetos de jogos e ao longo tempo necessário para concluí-los [12] as empresas tiveram que aproveitar ao máximo o efetivo contratado e tentar assim diminuir o risco dos projetos. Com esse intuito, elas começaram a utilizar a idéia de fábrica de *software* onde cada pessoa tem um papel bem definido na empresa, e trabalha paralelamente em vários projetos [16]. Além disso, os projetos passaram a ser muito bem gerenciados e especificados para minimizar o risco de problemas em etapas futuras.

Andrew Rollings e Dave Morris [16] propuseram um modelo de divisão geral de papéis para as empresas de jogos . A Tabela 2-2 mostra um resumo do modelo onde existem cinco grandes divisões: Gerenciamento e Projeto, Programação, Arte, Música e Som e por fim, Suporte e Garantia de Qualidade. Os papéis apresentados no modelo não são posições que devem ser ocupadas pela mesma pessoa durante todo o projeto. Além disso, uma mesma pessoa pode ocupar mais de um papel no projeto e esse modelo não é um modelo que deve ser adotado do jeito que aí está. Cada empresa de jogos deve adequá-lo às suas necessidades.

A divisão de Gerenciamento e Projeto cuida dos vários níveis de gerenciamento que estão envolvidos com a produção do jogo. Neste contexto, o papel do planejador de *software* é o de quebrar o projeto do jogo em um conjunto bem detalhado de requisitos técnicos e estimar o tempo e esforço necessários para implementá-lo, trabalhando em conjunto com o *game designer* e o arquiteto líder. O arquiteto líder deve especificar os

[3] Figura retirada do Site http://www.gamasutra.com/features/20001025/schaefer_01.htm

módulos existentes no jogo a partir dos requisitos técnicos identificados pelo planejador de *software*. O gerente de projeto cuida das interações entre membros da equipe e produz um planejamento eficiente para o desenvolvimento do jogo. Já o papel do *game designer* é o de projetar os jogos que a equipe vai desenvolver. O *game designer* deve produzir um documento chamado *game design* que é o projeto detalhado do jogo.

Divisão	***Papel***
Gerenciamento e Projeto	Planejador de Software
	Arquiteto Líder
	Gerente de Projeto
	Game Designer
Programação	Programador Líder
	Programador
Arte	Artista Líder
	Artista
Música e Som	Músico
	Técnicos em Efeitos de Som
Garantia de Qualidade	Líder de Garantia de Qualidade
	Técnico de Garantia de Qualidade
	Testador de Jogos

Tabela 2-2 Divisões e Papéis principais de uma Empresa de Jogos segundo Andrew Rollings e Dave Morris

A divisão de Programação cuida do desenvolvimento propriamente dito do jogo e, quase sempre, esta equipe está alocada em um projeto por vez. Geralmente a equipe é composta de um programador líder que coordena a equipe e garante os prazos estipulados pelo gerente do projeto. O restante da equipe é composto por programadores que, na maioria dos casos, são especialistas em alguma área (e.g. Inteligência Artificial, Redes, etc.). Os programadores devem ser experientes, isto é, nenhum tempo deve ser desperdiçado em estudo de como implementar uma determinada característica.

A divisão de Arte é responsável por toda a arte gráfica do jogo. Geralmente este grupo trabalha em vários projetos da empresa ao mesmo tempo. O papel do artista líder é o de coordenar o resto da equipe e de interagir com o programador líder e com o *game designer* para verificar se a arte produzida está de acordo com o padrão exigido.

Na divisão de Música e Som, o papel do músico é o de criar a trilha sonora do jogo. Geralmente os músicos trabalham em separado da equipe. Entretanto, quando o jogo exige uma música iterativa, ou seja, uma música que muda conforme o que for acontecendo no jogo, o músico passa a interagir mais com o restante da equipe. Já o papel do técnico em efeitos de som é o de criar todos os efeitos sonoros que estão presentes no ambiente do jogo, como por exemplo, sons de balas, explosões, passarinhos, etc.

A divisão de Garantia de Qualidade é responsável por verificar a qualidade e a jogabilidade do jogo. O papel do líder de garantia de qualidade é o de coordenar a equipe, e de interagir com o *game designer* e o gerente de projeto para confirmar que o jogo foi totalmente testado. O técnico de garantia de qualidade deve testar o código produzido pela quipe de programação e verificar se todas as funcionalidades foram implementadas. O testador de jogos deve testar a jogabilidade do jogo. Numa fase inicial, esse papel pode ser ocupado pelos programadores e artistas. Numa fase mais avançada, os testadores geralmente são pessoas de fora do projeto que tenham experiência no ramo.

2.4 TECNOLOGIAS UTILIZADAS EM JOGOS PARA PC

Atualmente os jogos são desenvolvidos utilizando-se bibliotecas multimídia que facilitam o acesso ao *hardware* através do sistema operacional. Nesse contexto, os *frameworks* para jogos são construídos utilizando essas bibliotecas, provendo assim, mais uma camada de abstração (ver a Figura 2-3). Por esse motivo, é interessante tomar conhecimento das duas principais bibliotecas multimídias utilizadas no desenvolvimento de jogos para PC: o OpenGL e o DirectX.

Outro ponto importante no desenvolvimento de um jogo é a representação gráfica utilizada. Atualmente nos PCs, existem dois métodos possíveis para a renderização: a 3D e a bidimensional por transferência de *bits*. A 3D utiliza uma malha de triângulos para formar

o cenário. Já a bidimensional carrega uma imagem na memória e utiliza a transferência de *bits* de uma área para outra, a fim de formar uma animação.

Visando facilitar o entendimento das escolhas realizadas no desenvolvimento do *Forge 16V*, esta seção irá apresentar alguns detalhes do OpenGL e DirectX, bem como algumas informações da técnica da representação gráfica tridimensional e bidimensional.

Figura 2-3 Camadas de softwares para a execução de um jogo.

2.4.1 OpenGL e OpenAL

O OpenGL (*Open Graphics Library*) é um padrão de renderização e aceleração gráfica bidimensional e, principalmente, tridimensional. Associado ao padrão, existe a API que provê um método simples e bem definido de desenvolvimento de aplicações gráficas. Com sucessos comercias como Quake III, Half-Life e Decente 3 que utilizaram a API no módulo gráfico, o OpenGL provou estar consolidado na indústria de jogos.

Uma das principais características do OpenGL é a existência da implementação da API em várias plataformas. Assim, a menos alguma diferença entre compiladores, o código que utiliza a API do OpenGL pode ser facilmente portado para outras plataformas. Nos PCs, o OpenGL pode ser rodado nos ambientes Microsoft Windows, Linux e Mac OS.

Outra característica importante do OpenGL é boa documentação. Já existem inúmeros livros publicados sobre o assunto, além da documentação padrão. Um outro ponto importante é estabilidade. Implementações do OpenGL já estão disponíveis em várias plataformas há mais de sete anos. Além disso, cada nova característica a ser acrescentada passa pelo controle de um consórcio de empresas independentes chamado de ARB

(*Architecture Review Board*). Em Junho de 2002, treze empresas tinham força de voto no consórcio [25]: 3DLabs, Apple, ATI, Dell Computer, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Matrox, Microsoft, NVIDIA, SGI e Sun.

A Tabela 2-3 mostra as versões das últimas especificações e as respectivas datas de lançamento. Cada especificação contém características que a API deverá possuir e define padrões que os fabricantes de placas de vídeo deverão seguir para torná-las compatíveis com a especificação. A versão 1.2 foi a primeira versão a incluir características especificamente requisitadas pelos desenvolvedores de jogos, como por exemplo, *multitexturing*. Desde então, a indústria de jogos tem olhado com muita atenção para o OpenGL, principalmente pela possibilidade de desenvolver jogos para várias plataformas com certa facilidade.

Versão da Especificação	***Data de Lançamento***
1.2	03/1998
1.3	08/2001
1.4	07/2002

Tabela 2-3 Versões mais atuais da Especificação do OpenGL

O OpenGL trata apenas os aspectos gráficos de uma aplicação. Para se desenvolver jogos, falta tratar a sonorização. Para resolver esse problema, uma biblioteca chamada OpenAL [26] foi criada. Ela pode ser usada em conjunto com o OpenGL para produzir jogos e aplicações multimídia. A versão mais atual da especificação do OpenAL é a 1.0.

2.4.2 Microsoft DirectX

O DirectX [19] é um conjunto de bibliotecas de uso geral para desenvolvimento de jogos e aplicações multimídia no ambiente Windows, desenvolvido pela Microsoft. Ele foi lançado na tentativa de atrair a indústria de jogos que tinham como plataforma padrão na época o DOS, para a Plataforma Windows 32-bits. Com esse propósito, a Microsoft fez com que o DirectX criasse uma independência do *hardware* utilizado e, ao mesmo tempo, possibilitasse o uso de recursos implementados em *hardware* sem passar pela API do Windows. Além disso, o DirectX permite que as aplicações incorporem gráficos

bidimensionais e tridimensionais, vídeos, sons estéreos e tridimensionais, músicas, dispositivos de entrada, como teclado, mouse e *joysticks* e permite que as aplicações utilizem uma rede de computadores para o desenvolvimento de aplicações multiusuário.

Diferentemente do que ocorre com o OpenGL, a Microsoft lança uma versão com características novas pelo menos uma vez por ano. Para isso, a Microsoft mantém uma relação estreita com os principais fabricantes de *hardware*, principalmente os de placas de vídeo (e.g. NVIDIA e ATI). Assim é possível garantir um consenso nas novas características a serem lançadas. Além disso, cada nova versão é completamente compatível com a anterior, garantindo assim que quem possui a versão mais atual do DirectX possa rodar jogos desenvolvidos com versões anteriores. A versão mais atual do DirectX é a 9.0 que foi lançada esse ano e o mercado já está cheio de novas placas de vídeo que suportam esta versão.

A arquitetura do DirectX é dividida em camadas (ver a Figura 2-4): a de aplicação, a de API do DirectX e a de abstração do *hardware* (HAL[4]). A camada da API é dividida em vários componentes. Os principais são: *DirectInput*, *DirectSound*, *DirectXGraphics*, *DirectPlay* e *DirectShow*. Entre a camada de abstração do *hardware* e a API do DirectX, existe uma camada de emulação de *hardware* (HEL[5]) que permite que uma determinada característica seja emulada por *software* quando ela não existir no *hardware.* Existe ainda uma interface entre a biblioteca gráfica do Windows (GDI), que é muito lenta, e o DirectX.

O DirectX Graphics é o componente do DirectX responsável por gerenciar e exibir gráficos bidimensionais e tridimensionais. Isso é feito através de uma interface de *software* que provê acesso direto aos dispositivos gráficos e a todas características proporcionadas por elas. O gerenciamento é baseado em polígonos, vértices e comandos. Através do DirectX graphics é possível realizar transformações de espaço, iluminação e rasterização no *pipeline* gráfico.

[4] *Hardware Abstraction Layer*

[5] *Hardware Emulation Layer*

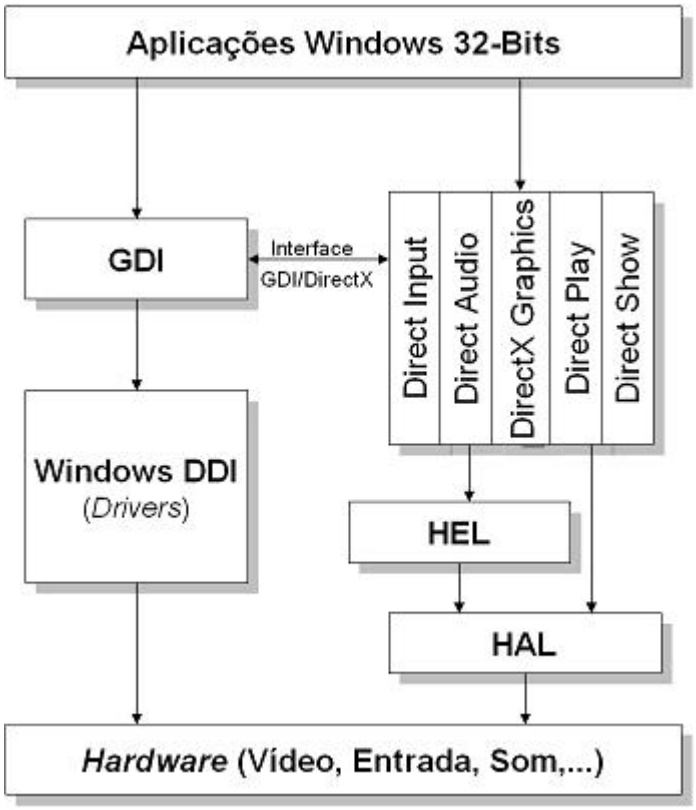

Figura 2-4 Arquitetura do DirectX

O *DirectInput* é o componente do DirectX que gerencia os dispositivos de entrada tais como teclados, mouses e *joysticks*. Para isso, ele realiza uma comunicação direta com os *drivers* dos dispositivos, proporcionando um acesso mais rápido aos dados de entrada do que o sistema padrão de mensagens do Windows. Em contrapartida, nenhum pós-processamento é realizado o que irá exigir um maior controle por parte do programador que está usando o *DirectInput*. Por exemplo, eventos do mouse são relativos à posição anterior, e para gerenciar a posição de forma absoluta (saber onde o mouse se encontra em um determinado instante levando-se em consideração algum sistema de coordenadas), um pós-processamento deve ser realizado.

O *DirectAudio* pode ser utilizado para incluir efeitos de som e música ao jogo. Com um suporte a áudio posicional (3D) e um sistema completo que facilita a implementação de trilha sonora dinâmica, o *DirectAudio* é perfeito para a criação de aplicações multimídia.

O *DirectShow* ajuda na captura, reprodução e conversão de formatos de *streams* multimídia. Embora vários formatos já sejam suportados, como por exemplo, ASF, MPEG,

AVI, MP3, o *DirectShow* permite a criação de novos componentes, possibilitando assim a reprodução de formatos proprietários. O *DirectShow* pode ser utilizado na criação de reprodutores de DVDs, editores de vídeo, reprodutores de MP3 e aplicações de captura de vídeo.

2.4.3 Formas de Representação Gráfica

Como foi mencionado, o módulo gráfico de um jogo atual pode utilizar uma das seguintes técnicas para a renderização gráfica: representação gráfica bidimensional e representação gráfica tridimensional. Nesta seção serão mostradas as principais características de cada uma delas, dando uma maior ênfase à representação gráfica bidimensional cujos conceitos envolvidos serão úteis nos próximos capítulos.

2.4.3.1 Representação Gráfica Bidimensional

A representação gráfica bidimensional se caracteriza essencialmente por transferência de mapa de bits entre áreas de memória, onde cada mapa de bits existente na memória do computador ou da placa de vídeo representa uma figura. Na placa de vídeo existe uma determinada área de memória que é varrida para formar o sinal enviado ao monitor. Para facilitar o entendimento, esta área de memória será chamada de "*área de exibição*" daqui pra frente. Quando se deseja exibir uma determinada figura que está na memória, basta transferi-la para a área de exibição na placa de vídeo. Esta cópia pode ser realizada a partir da memória convencional do computador (muito mais lenta pois existe uma série de operações ,como por exemplo, a obtenção do barramento) ou a partir de alguma outra área de memória da própria placa de vídeo (muito mais rápida). Embora a cópia realizada a partir da memória de vídeo seja muito mais rápida, algum tipo de gerenciamento deve ser realizado, pois a memória de vídeo é muito mais escassa que a memória do sistema. A Figura 2-5 demonstra a transferência de uma imagem na memória do computador para a *área de exibição* da placa de vídeo.

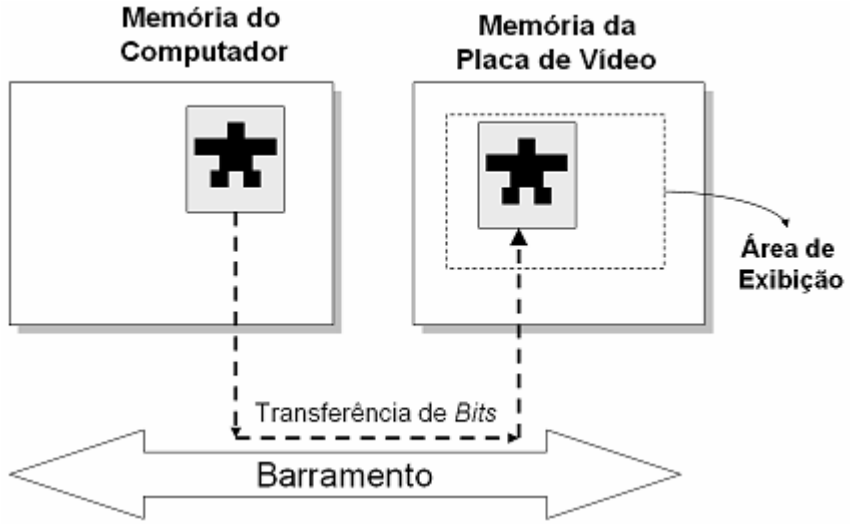

Figura 2-5 Transferência de bits entre a memória convencional do computador e a memória de vídeo.

Ao realizar a transferência de uma imagem de uma área de memória para a área de exibição da placa de vídeo, é importante observar se há espaço para toda a figura. Caso não haja, é necessário realizar uma operação de *clipping* e só transferir o que realmente vai ficar dentro da área de exibição. A Figura 2-6 demonstra o *clipping* realizado em uma figura que está sendo transferida para a área de exibição.

Realizando a transferência de bits a uma taxa de constante (e.g. 30 vezes por segundo), é possível reproduzir a sensação de animação igual ocorre com os desenhos animados. *Sprite* é o nome dado ao composto de uma seqüência de imagens, geralmente retangulares, que definem a animação de movimentação de alguma entidade [27]. A Figura 2-7 apresenta um exemplo de *sprite*. Certas propriedades, como por exemplo, altura, largura, posicionamento, velocidade e visibilidade estão diretamente associados aos *sprites* e servem para controlar o comportamento da entidade.

No entanto, quando várias imagens são transferidas diretamente para *área de exibição*, pode ocorrer o efeito colateral das imagens ficarem “piscando” no monitor [28]. Para resolver este problema, usa-se uma técnica conhecida como *double buffering* que consiste em reservar uma outra área de memória da placa de vídeo para se montar todo o quadro que se deseja exibir, e por fim, fazer uma única transferência para a área de exibição.

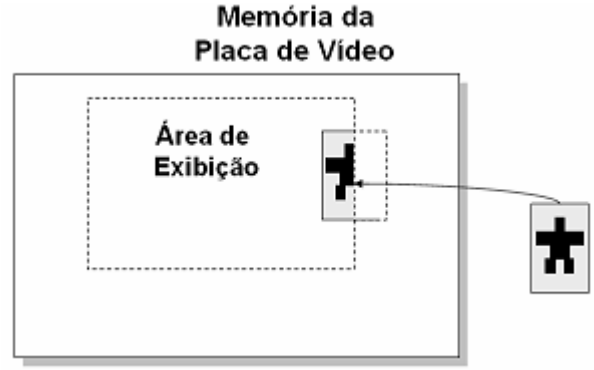

Figura 2-6 Clipping da figura.

Visando melhorar a performance do *double buffering*, algumas placas de vídeo possibilitam o uso de outra técnica conhecida como *flipping* que consiste em alterar a *área de exibição* da placa de vídeo. Com isso, pode-se montar o quadro que será exibido em uma área da memória de vídeo à parte, como no *double buffering*, mas ao invés de se fazer a transferência de todo o quadro montado para a área de exibição, altera-se esta área para onde o quadro foi montado, poupando-se assim uma transferência de bits. A Figura 2-8 demonstra o mecanismo utilizado no *flipping*.

Figura 2-7 Exemplo de *sprite*.

Outra técnica importante é a *source color keying* que possibilita a transparência. Quando uma transferência de bits é realizada, é possível estabelecer um conjunto de cores que não será transferida para o destino, dando assim a impressão de transparência. Esta técnica é muito importante, pois possibilita o uso de elementos não retangulares.

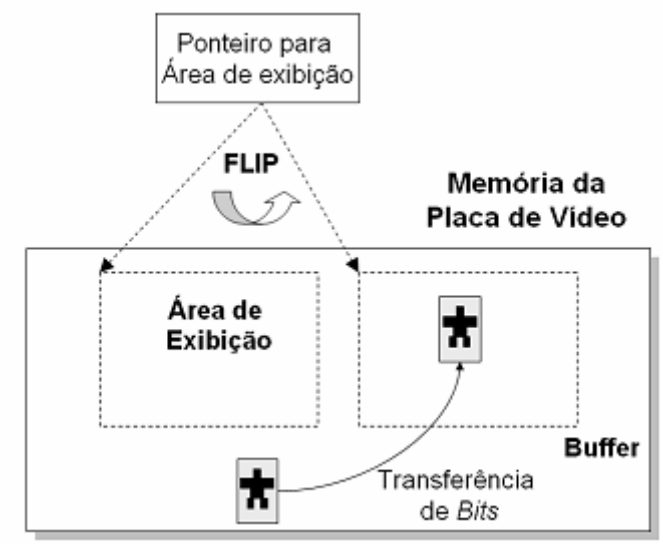

Figura 2-8 Flipping – Primeiro todas as imagens são transferidas para um *buffer*; em seguida o ponteiro que indica o início da área de exibição é trocado.

A representação gráfica bidimensional permite dois tipos de cenários: os cenários retangulares (2D) e os cenários isométricos. Os cenários retangulares podem ser baseados em textura ou em *tiles*. Os baseados em textura possuem uma figura em background, que pode ou não se movimentar dando a impressão de que o mundo está se mexendo (ver a Figura 2-9). Já os baseados em *tiles* dividem o mundo em pequenos retângulos, onde cada um deles cuida de exibir a figura de fundo e os *sprites* que estejam em seu domínio (ver a Figura 2-10). Para poupar recursos, do mesmo jeito que ocorre com os *sprites* de uma animação, é comum usar um arquivo denominado de *tileset* com um conjunto de *tiles* que estão presentes no jogo. Tanto nos cenários baseados em *tiles* quando nos baseados em textura, o sistema de coordenada é ortogonal e a detecção de colisão é geralmente baseada em retângulos.

Já os cenários isométricos utilizam uma visão em perspectiva, o que cria a impressão de profundidade no jogo e possibilita uma boa jogabilidade (ver Figura 2-11). Este tipo de cenário só é possível neste tipo de representação gráfica graças à transparência, que permite que figuras não retangulares sejam exibidas. Os jogos mais propícios a este tipo de cenário são os de estratégia e os de RPG (*role playing games*) [29] que demandam

uma boa visão geral do mundo. A representação gráfica tridimensional que será vista na próxima seção também pode gerar cenários isométricos.

Figura 2-9 Exemplo de cenário retangular – *Super Mario Bros Deluxe*

Com a representação gráfica bidimensional, é possível utilizar duas abordagens para conseguir produzir jogos isométricos: a abordagem de *tiles* e a de textura [28]. Na abordagem de *tiles*, o mundo é dividido por pequenos losangos ou hexágonos e cada *tile* é pintado seguindo uma ordem pré-estabelecida. O gerenciamento dos *tiles* é realizado através de uma estrutura de dados conhecida como *Tilemap* que mantém as informações de todos os *tiles* do cenário. Na abordagem de tela cheia, a figura de fundo é desenhada usando-se um modelador 3D ou qualquer ferramenta que possibilite gerar uma imagem do mundo no formato isométrico.

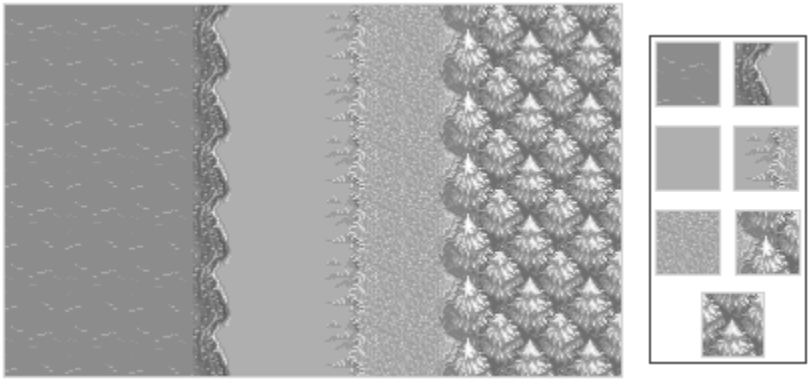

Figura 2-10 Exemplo de cenário retangular baseado em *tiles*.

Figura 2-11 Exemplo de Cenário Isométrico do Jogo *Age of Wonders* [30]

Um outro conceito muito importante é o de janela visível. Às vezes é necessário que o mundo seja bem maior que o que se pode exibir na tela. Para se fazer isto, é necessário que o jogo gerencie o estado do mundo como um todo mas desenhe apenas a parte que deve aparecer na tela. Através de um *scroll* é possível navegar por todo o mundo do jogo. A Figura 2-12 demonstra este conceito.

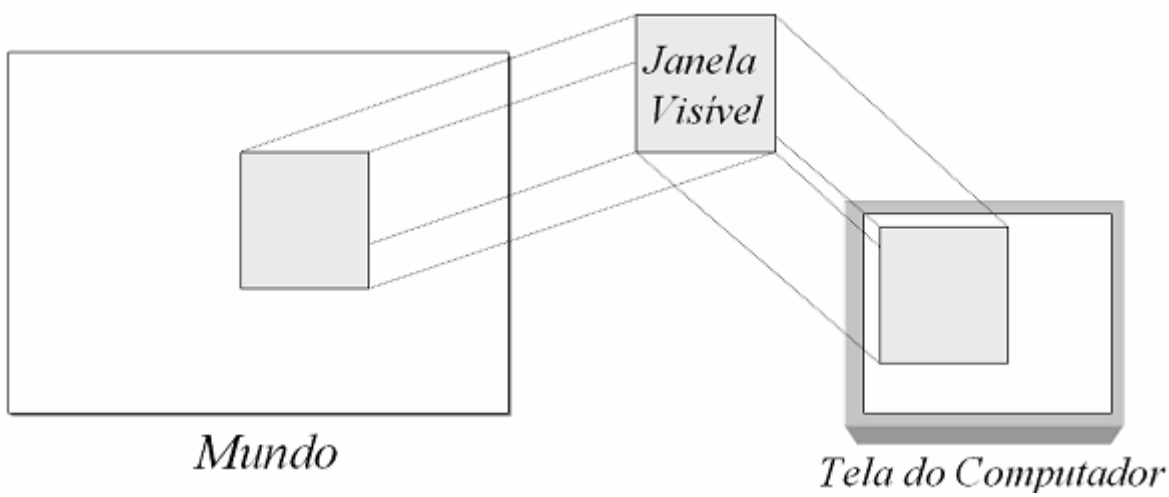

Figura 2-12 Janela visível – que parte do mundo deve ser renderizada na tela do computador.

2.4.3.2 Representação Gráfica Tridimensional

A representação gráfica tridimensional se caracteriza essencialmente por objetos representados por malhas de triângulos. Para cada triângulo, texturas podem ser mapeadas fazendo com que os objetos fiquem mais reais. Além disto, técnicas como iluminação dinâmica, neblina volumétrica, *skybox*, espelhamento e sistema de partículas tornam o mundo muito mais realista [28, 31, 32]. A representação gráfica tridimensional pode ser usada para produzir cenários puramente 3D ou cenários isométricos. Para desenvolver cenários isométricos, basta fixar a câmera num ponto de visualização isométrica.

Devido ao grande número de cálculos complexos necessários para gerar a representação gráfica tridimensional do mundo, as CPUs não são capazes de realizar esta tarefa a uma taxa aceitável para os jogos. Por isso, os jogos que utilizam a representação gráfica tridimensional exigem o uso de placas de vídeo aceleradoras 3D que possuem um *hardware* especializado para realizar as operações que envolvem cálculos 3D. Além disso, o código dos jogos tridimensionais é mais complexo que os 2D, pois envolvem técnicas complexas da matemática e demandam um alto grau de otimização para conseguir gerar o mundo 3D a uma taxa aceitável. A Figura 2-13 apresenta um exemplo de jogo que utiliza a representação gráfica 3D.

Figura 2-13 Exemplo de Cenário 3D – *Unreal 2*

2.5 USO DE FRAMEWORKS NO DESENVOLVIMENTO DE JOGOS

Nas seções anteriores, foi visto que a complexidade dos jogos cresceu no período pós 1995. Além disso, o orçamento destinado ao projeto aumentou juntamente com a equipe de desenvolvimento. Com um projeto que envolve cifras grandes e um tempo de desenvolvimento que varia de dois a cinco anos, o risco de problemas cresceu. Tentando minimizar este risco, a indústria de jogos investiu pesadamente em pesquisa e desenvolvimento e passou a utilizar técnicas da engenharia de *software* que antes não eram empregadas [16].

Tentando atingir um nível de reuso maior e tentando minimizar os riscos de problemas, a indústria de jogos começou cada vez mais a fazer uso dos *frameworks* orientados a objetos (que no meio ficaram conhecidos por *motores*) no desenvolvimento de jogos. Um *framework* pode ser visto como um conjunto de blocos de *software* pré-fabricados que podem ser usados como base de desenvolvimento para novas aplicações [33]. Roberts e Johnson [34] definem um *framework* como um projeto reusável de todo ou parte de um sistema de software, descrito por um conjunto de classes abstratas e pela forma como as instâncias dessas classes se relacionam entre si.

Com o uso dos motores para jogos, ficou mais fácil para uma empresa desenvolver mais de um jogo em paralelo e fazer com que a equipe focasse mais na lógica do jogo, devido ao maior nível de abstração e um maior grau de reuso, deixando de lado a maioria dos detalhes técnicos. Outro ponto positivo é que os motores diminuem o risco do projeto dar errado. Isto ocorre porque quem desenvolve os motores são especialistas em jogos, e o código e arquitetura resultante é bastante testado antes de entrar em produção, levando a um código mais robusto e otimizado. Além disso, o programador do jogo precisa escrever bem menos código, pois a estruturação do jogo é ditada, na maioria das vezes pela arquitetura estabelecida no motor.

Embora os motores utilizem as bibliotecas multimídia como base de desenvolvimento, uma das principais diferenças entre os motores e essas bibliotecas é que os motores provêem um comportamento padrão para a maioria das características e funcionalidades existentes. Assim, os desenvolvedores dos jogos não precisam saber quando chamar uma função, pois o motor possui um comportamento predefinido e sabe

chamar as funções certas no tempo certo. Isto significa que o motor é quem controla o fluxo de eventos e chama os elementos de código desenvolvidos. Por exemplo, é sabido que para o usuário se comunicar com o jogo ele precisa de alguma interface interativa, e em especial nos jogos para os PCs, o teclado é uma interface indispensável. Sabendo disto, o motor deve verificar a existência deste dispositivo e levantar um erro caso ele não esteja conectado. Já se o teclado estiver conectado, o motor deve inicializá-lo e pós-processar os eventos advindos do dispositivo para tratar, por exemplo, da utilização de acentos. Por fim, o motor deve enviar os eventos pós-processados a uma rotina escrita pelo desenvolvedor do jogo. Do ponto de vista do desenvolvedor do jogo, ele só precisa saber qual método deve sobrescrever para receber os eventos e como tratá-los, deixando de se preocupar com detalhes técnicos de como acessar o teclado.

Apesar dos motores proverem um controle de fluxo, eles ainda possibilitam que os desenvolvedores mais experientes modifiquem, acrescentem funcionalidades e alterem o fluxo de alguns eventos para adaptar o *framework* ao jogo que está sendo desenvolvido. Esta adaptabilidade possibilita que, na fase final do projeto, seja feito um ajuste fino no motor de acordo com as necessidades.

2.5.1 Metodologia de Desenvolvimento de Frameworks

O projeto de um *framework* é algo bastante complexo, pois é preciso chegar a uma abstração que atinja os principais problemas de um determinado domínio. Além disso, um *framework* precisa ser simples pra ser aprendido facilmente e ainda sim possuir funcionalidades suficientes pra poder ser utilizado na prática [34].

Para facilitar a criação de frameworks, Roberts e Johson [34] propuseram uma metodologia de desenvolvimento de frameworks através de uma linguagem de padrões (padrões que são relacionados entre si) que mostra os principais elementos envolvidos no desenvolvimento dos *frameworks*. O resto desta seção apresenta os padrões propostos por Roberts e Johson. Alguns desses padrões podem ocorrer ao mesmo tempo em algumas fases do desenvolvimento.

Para se desenvolver um *framework*, geralmente se pensa em abstrações a partir de exemplos concretos. Por isso o padrão denominado de “Três Exemplos” foi criado para ser o primeiro passo no desenvolvimento de um *framework*. A regra é simples: prototipa-se

três aplicações em seqüência na qual o *framework* deveria ajudar na criação se já estivesse pronto, sendo que uma um pouco diferente da outra. Com isso fica mais fácil a detecção de abstrações.

Quando se está desenvolvendo esses três protótipos, deve-se ter em mente que o objetivo final é o desenvolvimento de um *framework*, por isso é essencial tentar deixá-las flexível e estendível desde o primeiro protótipo través de mecanismos simples de orientação a objetos como, por exemplo, herança. Com isso se terá um *framework* de "Caixa Branca" (esse é o segundo padrão), pois o *framework* confiará fortemente no mecanismo de herança para se desenvolver novas aplicações.

Uma vez que se tenha o primeiro *framework* ao final do primeio protótipo, deve-se usá-lo para tentar descobrir o poderá mudar e o que provavelmente não precisará mudar no desenvolvimento de novas aplicações durante os outros dois protótipos. Toda vez que se precise de uma nova classe similar a uma que já foi desenvolvida no *framework*, cria-se uma subclasse e sobrescrevem-se os métodos diferentes. Após o desenvolvimento de algumas subclasses, será mais fácil identificar os métodos que não precisam ser sobrescritos. Nesse ponto, será possível fazer o *refactoring*[35] do *framework* criando-se classes abstratas que contém as partes comuns.

Também é importante identificar as principais funcionalidades que serão utilizadas no desenvolvimento das aplicações do domínio em questão. Por isso é importante começar a construir uma boa "Biblioteca de Componentes" (terceiro padrão).

Uma outra questão importante que se deve observar no desenvolvimento dos segundo e terceiro protótipos é a detecção dos pontos que mudam de uma aplicação para outra. Esses pontos são conhecidos como *Hot Spots* (quarto padrão). Coletando-se esses pontos em objetos conhecidos, fica mais fácil o processo de reuso do *framework* e mostra aos usuários do *framework* onde os projetistas esperam que o *framework* mude. Após coletar os *Hot Spots* em objetos conhecidos, usa-se a composição ao invés da herança para se conseguir a variação necessária de uma aplicação para outra.

Para facilitar a variação de código de uma aplicação para outra, geralmente são utilizados os "Objetos Plugáveis" (quinto padrão – objetos que se plugam através de composição tomando como base uma classe abstrata). Com eles a variação só precisa ser

conhecida no protocolo de inicialização, ou seja, durante o restante da execução do programa, essa variação fica transparente.

Outro padrão utilizado no tratamento de características variáveis são os “Objetos Especializados”. Toda vez que se encontrar uma classe que encapsule múltiplas características que podem variar de forma independente, é melhor criar múltiplas classes pra encapsular cada característica, criando-se assim, uma hierarquia mais especializada.

Uma vez que a biblioteca de componentes tenha sido organizada e especializada através da herança, usa-se a composição de componentes para a criação da aplicação. Com isso o *framework* torna-se um “*Framework* de Caixa Preta”, ou seja, o usuário não precisa ter pleno conhecimento da hierarquia do *framework* para utilizá-lo. Para ajudar a fazer a composição dos componentes, uma ferramenta visual que permite especificar de forma gráfica quais os componentes estão presentes na aplicação e como eles se relacionam, deve ser desenvolvida. A esta ferramenta dá-se o nome de “*Visual Builder”* e ela torna a utilização do *framework* muito mais fácil.

Uma vez criada a ferramenta visual, tem-se uma linguagem visual. E como toda linguagem, faz-se necessário à criação de “Ferramentas da Linguagem” que facilitam a depuração e a inspeção da mesma.

Vistos os principais elementos envolvidos na criação de um *framework*, na próxima seção serão apresentados os principais requisitos de um *framework* para jogos.

2.5.2 Principais Requisitos de um Motor para Jogos

2.5.2.1 Fácil Utilização

Pode parecer um requisito trivial, mas a existem vários motores que pouca documentação e que requerem um forte conhecimento da estrutura do *framework* para serem utilizados. O desenvolvimento de um jogo é um processo longo e que envolve o orçamento elevado e por isso a indústria optou por utilizar os *frameworks*. Se o motor não for bem documentado, grande parte do esforço será gasto no aprendizado de como utilizá-lo, aumentando assim o risco do projeto.

Por isso é muito importante uma boa documentação contendo exemplos de como usar, os principais pontos de variação de código que o projetista do *framework* espere que

mude de um jogo para o outro e os principais problemas conhecidos com a atual versão. Além disso, é essencial ter documentado as principais funcionalidades que o motor implementa. Assim fica mais fácil pra quem vai utilizar o motor decidir no início do projeto se ele atende ou não as características exigidas pelo jogo em questão.

2.5.2.2 Sistema de Renderização Gráfica Eficiente

Todo motor de jogos deve possuir um subsistema gráfico, seja ele 2D ou 3D. Como o tempo gasto para produzir um resultado visual na tela do computador consome mais da metade do tempo levado para produzir um quadro do jogo, ele deve ser o mais eficiente possível. Não adianta ter um motor cheio de funcionalidades se ele não tem performance para renderizar uma cena.

2.5.2.3 Garantir elementos que ajudem a Jogabilidade e Experiência do Jogador

Embora a jogabilidade seja uma questão do projetista do jogo, o motor deve garantir elementos que podem ser usados para dar uma melhor jogabilidade ao jogo como, por exemplo, o suporte a volantes, joysticks e mouses de três ou mais botões. Além desses, devem ser garantido os elementos que possam melhorar a experiência do jogador enquanto estiver jogando o jogo, como o suporte a som 3D que dá a impressão de imersão do jogador dentro do ambiente do jogo, e elementos de interface gráfica como janelas, botões etc.

2.5.2.4 Ferramentas que Auxiliem a criação dos Jogos

Um bom motor de jogos deve possuir ferramentas que ajudam a criação de novos jogos. Essas ferramentas devem ajudar desde o processo de codificação como, por exemplo, um subsistema que auxilie um a depuração do código, até o processo de construção do cenário do jogo, que é o caso do editor de cenários.

2.5.2.5 Suporte a Sons de Efeitos e Música.

Todo jogo precisa ter músicas e efeitos especiais para proporcionar uma melhor impressão aos jogadores. Por isso, é essencial ao motor de jogos provê um suporte para tocar músicas de fundo e efeitos durante o jogo. Assim a equipe que está criando um jogo só precisa se preocupar em compor a melhor melodia e em que horas ela deve ser tocada.

2.5.2.6 Suporte a Conectividade

Com o mundo conectado através da Internet, os jogos com suporte a conectividade estão se tornando cada vez mais comum. De fato, a grande maioria dos jogos lançados atualmente possibilita que vários jogadores joguem entre si usando uma rede de computadores, pois a iteratividade entre humanos geralmente propícia uma melhor diversão do que a iteratividade com a máquina. Por isso a conectividade tornou-se nos últimos anos um requisito dos motores de jogos pra computadores.

2.5.2.7 Suporte a uma Linguagem de Script

A grande parte dos jogos atuais possui eventos que são disparados quando uma determinada condição é satisfeita. Codificar esse comportamento na linguagem que o jogo foi desenvolvido geralmente é uma tarefa árdua para uma característica que pode mudar a qualquer momento. Por isso as linguagens de scripts que são bem mais simples que uma linguagem de programação como C++ e Java, por exemplo, passaram a fazer parte da maioria dos jogos atuais.

2.5.2.8 Implementação de Algoritmos Básicos de IA

Todo jogo usa algum conceito de inteligência artificial para controlar os personagens comandados pelo computador. Existem alguns algoritmos que são básicos e que podem ser colocados na biblioteca de componentes do motor como requisitos desejáveis. Um bom exemplo para isso é o algoritmo de *path-finding*. Quando um jogador, seja ele controlado pelo computador ou não, vai de uma posição a outra no cenário, é necessário descobrir o melhor caminho a partir de uma série de parâmetros, como por exemplo a menor distância.

2.5.2.9 Gerenciamento dos Objetos no Mundo

O gerenciamento de objetos no mundo é uma tarefa bastante complicada. Por isso é desejável que o motor possua algum suporte a esse gerenciamento. Quando o motor é 3D, fica mais fácil implementar um gerenciador do mundo mais complexo haja vista que o modelo usado poderá provavelmente ser usado pela grande maioria dos jogos. Com o 3D dá inclusive pra implementar no motor a detecção de colisão e garantir as leis físicas no mundo. Já quando o mundo não é 3D, fica mais complicado a implementação pois as

estruturas de dados ficam mais dependentes do jogo, o que torna esse requisito apenas desejável para um motor. Contudo é possível pelo menos implementar uma biblioteca com funções básicas que podem ser usadas para gerenciar os objetos, como por exemplo, um pool de objetos.

2.6 CONCLUSÕES

Neste capítulo foi mostrado que a engenharia de *software* não foi amplamente usada na indústria de jogos até 1995. No período pós 1995, a mentalidade começou a mudar quando o uso de rotinas em linguagem de baixo nível no desenvolvimento de jogos foi substituído por linguagens de alto nível. Isto levou a um aumento da complexidade dos projetos, fazendo com que a indústria de jogos investisse fortemente em pesquisa e desenvolvimento. As equipes de desenvolvimento cresceram, assim como o tempo de desenvolvimento que passou a ser de dois a cinco anos. Isto levou a indústria a perceber que o uso de *frameworks* (motores) no desenvolvimento de um jogo era imprescindível. Também foram mostrados os principais elementos do desenvolvimento de jogos e os principais requisitos que um motor para jogos deve satisfazer.

Além disso, foram mostradas as principais bibliotecas multimídia, a OpenGL e a DirectX, e apontadas as principais características de cada uma. Também foram apresentadas as principais técnicas de renderização gráficas utilizadas no desenvolvimento de jogos atualmente: a renderização tridimensional e a bidimensional.

Por fim, foi visto que os motores para jogos possibilitaram que vários jogos fossem desenvolvidos paralelamente dentro da mesma empresa. Além do mais, eles reduziram o risco de problemas nos projetos. Com os motores, a equipe de desenvolvimento voltou-se quase que totalmente para a lógica do jogo em questão. Hoje existem várias empresas especializadas no desenvolvimento de motores para jogos.

Capítulo 3

Forge V8

Neste capítulo serão expostas as principais motivações que levaram ao desenvolvimento do Forge V8, bem como a arquitetura proposta por ele. Além disso, serão mostrados os principais problemas encontrados no desenvolvimento do framework.

O *Forge V8* é um motor de jogos para PC e aplicações multimídia desenvolvido por Charles Madeira [7] em seu projeto de dissertação de Mestrado do Centro de Informática da Universidade Federal de Pernambuco. Por ser um projeto pioneiro no CIn-UFPE, foram cometidos alguns erros no seu desenvolvimento. Foi a partir da grande experiência adquirida pela construção do *Forge V8* que conseguimos melhorar a arquitetura e desenvolver um *framework* mais maduro, o *Forge 16V*. Por este motivo, um capítulo deste trabalho foi dedicado ao *Forge V8*, mostrando os motivos que levaram ao desenvolvimento de tal motor, e suas principais virtudes e defeitos.

3.1 HISTÓRICO

Tudo começou com o desenvolvimento de um jogo intitulado NetMaze como projeto de disciplina de mestrado de Charles. O NetMaze é um jogo bidimensional no estilo Pac-man, onde os personagens se movimentam dentro de um labirinto. Existem dois tipos de personagens: os caçadores e as caças. O objetivo dos caçadores é procurar e perseguir a caça e capturá-la. Já a caça deve ficar o maior tempo possível sem ser capturada. Cada personagem deve ser controlado remotamente usando a infra-estrutura de redes de computadores, podendo ser controlados por computador ou pelo próprio jogador.

Figura 3-1 Tela Principal do NetMaze

Na tentativa de dar continuidade ao NetMaze como projeto de dissertação, constatou-se que o jogo não estava compatível com a qualidade e tecnologia aplicadas aos jogos industriais. Resolveu-se então, projetar um novo jogo que estivesse à altura dos jogos comerciais da época: o *Canyon*. O *Canyon* foi concebido pra ser um jogo de ação projetado para usar um ambiente gráfico tridimensional. O tema gira em torno de uma batalha entre dois povos, os *Zagotha* e os *Xarix* que lutam pra obter a posse do planeta *Canyon*, rico em minerais energéticos. Sem a energia dos minérios, as espécies estariam fadadas à extinção. O Jogador assume o papel de piloto de uma nave que através de várias missões deve conquistar o planeta. O projeto do jogo pode ser encontrado no apêndice da dissertação de Madeira [7].

No processo de implementação, logo se verificou a necessidade de um *framework* no processo de desenvolvimento de um jogo. Após uma breve pesquisa, constatou-se que por questão de sigilo industrial, quase nenhuma documentação existia sobre o processo de desenvolvimento de *frameworks* para jogos. O estudo foi concentrado então, em encontrar um motor que atendesse a todas as necessidades de desenvolvimento. Nesta procura, vários motores, que na época eram de código aberto, foram analisados. Os principais foram: *Genesis 3D* [36], *Crystal Space* [15] e o *Golgotha* [14].

O *Genesis 3D* é um pacote de desenvolvimento de jogos tridimensionais. Na época, a principal dificuldade deste motor era na renderização de ambientes externos. Assim, a performance do motor em ambientes externos não era adequada, inviabilizando a utilização no jogo *Canyon*, que era composto basicamente de cenários externos. Além disso, a documentação não era boa.

Embora fosse o melhor projetado no que diz respeito à Engenharia de Software dos motores pesquisados, o Crystal Space não se adequava ao desenvolvimento do jogo *Canyon*, pois não provia, na época, nenhum suporte a renderização em ambientes abertos. A documentação do código era muito boa, entretanto a documentação do projeto ainda não era.

O *Golgotha* foi projetado para ser um motor comercial pela *Crack dot Com* como parte do desenvolvimento de um jogo que levava o mesmo nome. Após passar o prazo de entrega do projeto e sem verba para finalizar o motor, a empresa resolveu liberar o código para o público, tornando-o *software* livre. Este motor tinha uma performance muito

superior aos outros estudados. Além disso, ele funcionava muito bem em ambientes abertos. No entanto, observou-se que os principais conceitos da engenharia de software não foram aplicados no projeto de desenvolvimento do *framework*. O motor não possuía quase nenhuma documentação, o código era de difícil compreensão e a reusabilidade era questionável. Estes motivos inviabilizaram o uso do *Gogotha* no desenvolvimento do *Canyon.*

Tendo em vista que nenhum motor de código aberto estudado se apresentou satisfatório, começou-se então a se pesquisar na literatura, algo sobre o processo de desenvolvimento de um *framework* para jogos de computador. Contudo, devido ao sigilo comercial imposto pela indústria de jogos, pouco foi encontrado a respeito. Como a licença de um motor comercial era muito cara na época [13, 37, 38], optou-se então pelo desenvolvimento de um motor que atendesse a todos os requisitos do jogo *Canyon*. Assim, ao final do processo seria possível dominar o processo de desenvolvimento de *frameworks* para jogos e com isso, além de facilitar o desenvolvimento do *Canyon*, tornar mais fácil o apoio acadêmico a disciplinas de jogos na universidade e estimular um pólo emergente de empresas locais de jogos. Foi aí que o projeto do *Forge V8* tomou corpo.

A seguir serão mostrados os principais objetivos e características almejadas pelo *Forge V8*.

3.2 OBJETIVOS

Os principais objetivos da equipe eram de desenvolver um motor que fosse capaz de atender a todos os requisitos do jogo *Canyon* e principalmente o de dominar a e catalogar o processo de desenvolvimento de um *framework* para o desenvolvimento de jogos.

A proposta do *Forge V8* era ser um motor genérico para desenvolvimento de jogos e aplicações multimídia para PC. Com esse intuito, o *framework* foi projetado para criar uma certa independência da tecnologia utilizada, possibilitando assim a utilização de várias bibliotecas de desenvolvimento, tais como DirectX [19] e OpenGL [39]/ OpenAL [26]. Assim, seria possível desenvolver um *framework* que poderia rodar em vários sistemas operacionais, como o Windows e Linux. Como os jogos da indústria, por ser aplicações em

tempo real, exigem uma grande performance, resolveu-se utilizar a linguagem C++ [40] como base para o desenvolvimento do *framework.*

Como o *Canyon* era um jogo projetado para rodar em um ambiente tridimensional, o *Forge V8* foi estruturado para trabalhar com tal tecnologia. Contudo, visando a atender o maior domínio possível de jogos, o motor foi projetado para também dar suporte aos jogos bidimensionais e isométricos utilizando a tecnologia empregada nos jogos tridimensionais como base. No que diz respeito à sonorização, o motor facilitaria a reprodução de sons e efeitos em ambientes estéreo e com controle de parâmetros 3D. Além dessas características, o *Forge V8* deveria dar suporte ao desenvolvimento de jogos multiusuários (em rede), prover facilidade no desenvolvimento da Inteligência Artificial e da Modelagem Física e controle de objetos do jogo e disponibilizar um editor de cenários genérico para os jogos.

3.3 ESTRUTURA DO FORGE V8

Nesta seção será mostrada a estrutura inicial idealizada para o *Forge V8* e as principais características de cada módulo. Nas seções seguintes, veremos os principais problemas encontrados na implementação deste modelo.

Pensando em manter uma maior independência da tecnologia utilizada, o motor foi projetado para possuir três camadas (ver a Figura 3-2): *camada de sistema*, *camada dos gerenciadores* e *camada aplicação*.

A *camada de sistema* é responsável por realizar toda a comunicação com o *hardware*, criando assim uma independência de plataforma para as camadas superiores. Isso significa que esta é a única camada que precisa ser modificada, caso se deseje portar o motor para uma nova plataforma. Além disso, uma interface foi definida para servir de fachada às outras camadas, fazendo com que elas tenham acesso aos subsistemas sem precisar ter conhecimento qual implementação está sendo utilizada.

Como está exposta na Figura 3-2, a camada de sistema é composta de diversos subsistemas. O subsistema de dispositivos gráficos é o mais complexo deles. Ele é responsável pelo acesso ao dispositivo e pelo processo de renderização. Este subsistema utiliza outras bibliotecas gráficas que já estão bem fundamentadas, tais como DirectX [19] ou OpenGL [39], para garantir uma melhor qualidade gráfica.

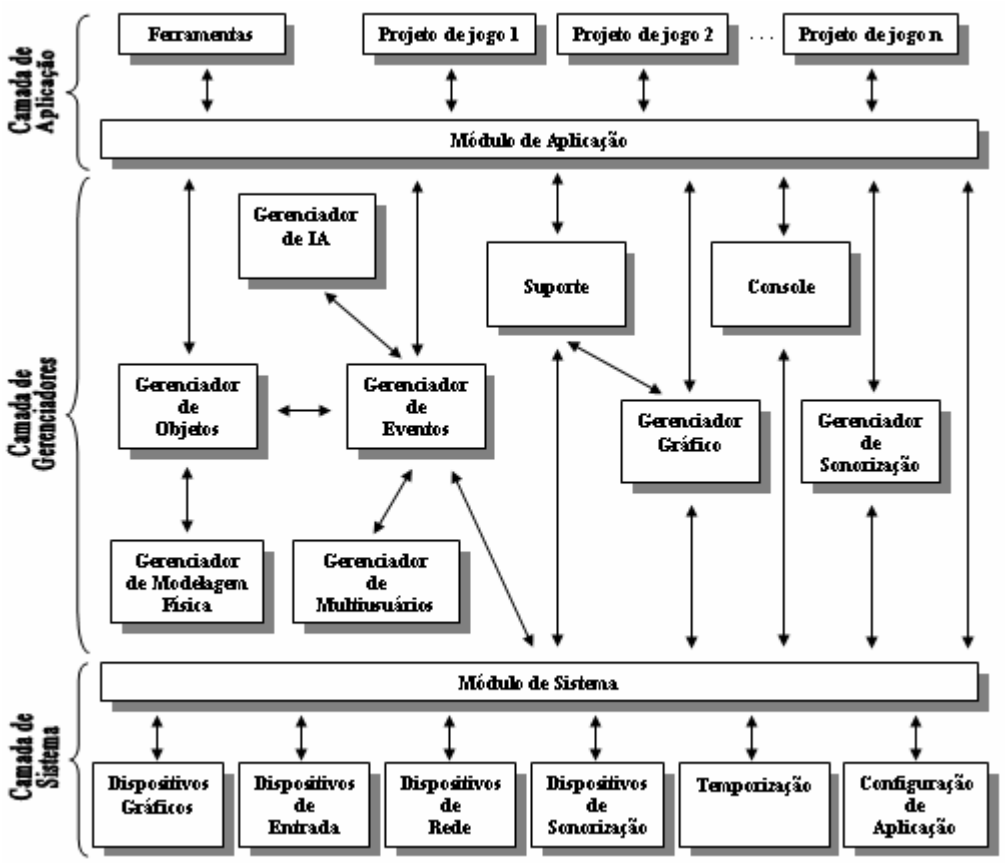

Figura 3-2 Camadas do *Forge V8* [7]

O *subsistema de dispositivos de entrada* provê a infra-estrutura necessária para que serviços de teclado, mouse e *joystick* sejam disponibilizados. Ele verifica a existência de tais dispositivos e identifica os eventos advindos deles.

O *subsistema de dispositivos de sonorização* cuida da leitura e gerenciamento de sons e músicas, e a reprodução deles na placa de som. Uma outra função bastante importante deste módulo é o suporte ao processamento de sons em ambientes tridimensionais. Isto garante uma boa qualidade sonora ao jogo, pois acrescenta realismo.

O *subsistema de dispositivos de rede* é responsável pela comunicação (envio e recepção de mensagens genéricas) através de uma rede de computadores. Já o subsistema de temporizarão é responsável por calcular o tempo com a melhor aproximação possível para um dado *hardware* e sistema operacional. Por fim, o subsistema de configuração de aplicação garante os ajustes necessários de acordo com o sistema operacional em uso.

A *camada de gerenciadores* é responsável pelo gerenciamento da execução da aplicação e ela também é composta de vários módulos. Os módulos de console e suporte são utilizados por outros módulos do motor. Os restantes dos módulos desta camada são independentes, podendo assim, a aplicação utilizar um ou mais deles.

Todo jogo, principalmente em sua fase de desenvolvimento, deve possuir um *console gráfico*. Ele serve para modificar as configurações do motor e a do jogo, sem precisar reiniciar a aplicação. Além disso, ele expõe os dados de *debug* na tela, o que torna o processo de depuração da aplicação mais fácil. No *Forge V8*, esta funcionalidade é implementada no *módulo de console*. O console gráfico sempre está presente nas aplicações, embora ele possa se tornar completamente transparente ao usuário da aplicação.

O *módulo de suporte* serve para oferecer serviços utilitários aos outros módulos do motor e ao jogo. Estes serviços incluem desde rotinas matemáticas até ferramentas de compactação e descompactação de arquivos proprietários.

O *gerenciador gráfico* é responsável por controlar o processo de criação de cenários bidimensionais, isométricos e tridimensionais, além de cuidar de todos os componentes associados ao processo. Este módulo se comunica indiretamente com o subsistema de dispositivos gráficos da camada de sistemas para realizar a renderização da cena. Sem dúvida nenhuma, este é o módulo mais complexo do motor, pois demanda um alto grau de otimização.

O *gerenciador sonoro* é o módulo responsável por controlar a execução de sons e músicas e controlar efeitos e parâmetros sonoros, como por exemplo, o volume e o posicionamento (no caso de sons tridimensionais). Este módulo utiliza o subsistema de dispositivos sonoros para envidar os sons à placa de som.

Já o *gerenciador de objetos* controla o mundo e os objetos de um jogo. O principal desafio deste módulo é ser genérico suficiente para atender o maior domínio de jogos possível, e ser flexível suficiente para permitir que cada jogo possua características particulares. Este módulo está fortemente ligado ao gerenciador de modelagem física no que diz respeito à detecção de colisão. Além deste, o editor de cenário, que é uma ferramenta gráfica utilizada para criar cenários, utiliza este módulo para descrever o estado do mundo em cada nível do jogo.

O *gerenciador de eventos* é responsável por controlar todo o fluxo de eventos da aplicação e do motor. Como exemplo de eventos, podemos citar os eventos do sistema operacional, eventos de rede, eventos de dispositivos de entrada e eventos gráficos.

O *gerenciador de multiusuários* controla todo o aspecto operacional da comunicação entre diversos usuários usando o subsistema de dispositivo de rede da camada de sistema. A principal virtude deste módulo é a de esconder detalhes de programação *multithread* do programador do jogo [29]. Com a abstração criada, o programador dos jogos vai simplesmente definir os parâmetros de comunicação, efetivar a conexão e começar a enviar e receber mensagens. Sendo assim, ele próprio pode definir o protocolo de comunicação que melhor convier ao jogo.

Para finalizar a da camada de gerenciadores, faltam serem mencionados o *gerenciador de Inteligência Artificial* e o *gerenciador de modelagem física*. O primeiro é responsável por controlar o comportamento de entidades autônomas (NPC) e toda a IA do jogo. Já o *gerenciador de modelagem física* é responsável pelo gerenciamento de propriedades físicas dos objetos do mundo. Estas propriedades físicas geralmente são baseadas na física Newtoniana que é bastante apropriada para este tipo de aplicação.

A camada de mais alto nível do motor é a *camada de aplicação*. Ela garante uma interface mais suave entre o motor e o jogo. Ela visa um maior desacoplamento entre o jogo desenvolvido e o *framework*, possibilitando que o motor possa ser utilizado no desenvolvimento de vários jogos.

3.4 POSTMORTEM

Esta seção apresentará as principais lições aprendidas na implementação do *Forge V8*, ou seja, os principais erros e acertos do projeto como um todo.

3.4.1 Principais Contribuições

Como foi mencionado antes, o *Forge V8* foi um trabalho pioneiro no Centro de Informática da UFPE. Não existia até então nenhuma pesquisa relacionada a motores de jogos. O grupo de pesquisa de jogos do CIn tinha noção de como desenvolver um jogo, mas nunca tinha feito nenhum trabalho relacionado ao desenvolvimento de um *framework* neste

domínio. A seguir serão mostradas as principais contribuições do desenvolvimento *Forge V8*.

3.4.1.1 Definição dos Principais Elementos de um Motor de Jogos

Pare se desenvolver o *Forge V8,* uma forte pesquisa teve que ser feita sobre os principais componentes que compõem um motor para jogos. Achar o limiar entre o que deve ser do motor e o que deve ser deixado para o jogo em si é uma tarefa difícil. Quanto mais se implementa um motor, mais se corre o risco de especializá-lo demais, e caso algo fique de fora, maior o risco de deixar tarefas importantes para o jogo em si.

Pode-se dizer que o trabalho realizado no *Forge V8* de identificação dos principais componentes e as interligações entre eles foi muito boa, tanto que serviu de referência para o desenvolvimento de outros trabalhos, como por exemplo, o wGEM[6] e o Forge 16V.

3.4.1.2 Estudo das Principais Tecnologias no Desenvolvimento de Jogos

Antes de desenvolver o *Forge V8*, o grupo de pesquisa de jogos do Centro de Informática só tinha conhecimento das tecnologias de desenvolvimento de jogos DirectX e Java 3D. *O Forge V8* serviu para aprender a tecnologia OpenGL, além de se aprofundar em cada uma delas a ponto de escolher o OpenGL e DirectX e propor uma arquitetura que possibilitasse a escolha uma delas por parte do desenvovedor do jogo.

3.4.1.3 Catálogo de Padrão de Projetos

Um outro ponto importante do *Forge V8* foi o estudo mais aprofundado da arquitetura dos motores pra jogos e o catálogo de padrões de projeto construído. Esse catálogo apresenta os principais padrões de projeto identificados durante a implementação do motor, e é o primeiro trabalho do gênero que se tem conhecimento. Com ele é possível identificar os principais problemas encontrados durante a fase de desenvolvimento, e como esses problemas foram resolvidos.

3.4.2 Principais Problemas

Parte destes problemas teve origem na inexperiência em desenvolver este tipo de sistema , o que levou o grupo a subestimar a tarefa a ser realizada. O motor idealizado deveria funcionar utilizando o DirectX e OpenGL e deveria atender aos jogos 2D,

isométricos e 3D. Isso, como será mostrado a seguir, aumentou muito a complexidade do projeto.

Contando com uma equipe de apenas quatro programadores (apenas dois em tempo integral) e gastando uma boa parte do tempo tentando resolver os problemas do módulo gráfico, parte do sistema inicialmente idealizado não chegou a ser implementado, ou o foi apenas parcialmente. Este é o caso do *gerenciador de inteligência artificial* e do *gerenciador de objetos*.

A seguir, serão apontados os três principais problemas encontrados durante o projeto.

3.4.2.1 Colisão de Nomes

O primeiro problema é simples, e se tivesse sido detectado na fase inicial do projeto, seria de fácil resolução. Ele diz respeito à colisão de nomes de classes. O *Forge V8* seguiu um padrão de codificação na implementação do sistema usando uma notação baseada na notação húngara [41], que é bastante utilizada em programação para Windows, para estabelecer a nomenclatura de classes, interfaces, métodos, parâmetros e variáveis. No que diz respeito às classes e interfaces, a notação estipula o seguinte padrão:

- O nome começa com o prefixo `C` para classes e `I` para interfaces, seguidos pelo nome da respectiva classe ou interface;
- O nome da classe ou interface deve começar com letra maiúscula;
- Usar uma separação de nomes através de letras maiúsculas e minúsculas. Exemplos: `CFile`, `CFileLoader`, `CJPEGFileLoader`, `IMouseListener`.

O problema é que outras bibliotecas usam a mesma notação. Isto quer dizer que a colisão do nome de classes ou interfaces do motor com outras bibliotecas ou mesmo com classes do jogo sendo desenvolvidas pode ocorrer, tendo como conseqüência um erro de compilação. Um bom exemplo disto é a biblioteca MFC da Microsoft [42]. Ela possui uma classe chamada `CFile` que ajuda na manipulação de arquivos. O *Forge V8* também possui uma classe com o mesmo nome e ela não pode ser usada quando a classe do MFC está em uso.

Existem duas soluções de fácil implementação para resolver este problema. A primeira utiliza uma construção da linguagem C++ chamada de *name space* [40]. Com ela é possível atribuir uma classe a um espaço com um nome predefinido. Com isso, a classe `CFile` do *Forge V8* passaria a ser chamada de `FV8::CFile`, evitando assim, qualquer problema de colisão. A segunda solução seria alterar um pouco a notação utilizada para inserir um prefixo `FV8` nas classes e interfaces para identificar que elas vieram do motor, evitando assim que seus nomes coincidam com os nomes das classes em outras bibliotecas. Assim, a classe `CFile` passaria a ser chamada de `CFV8File` resolvendo o problema.

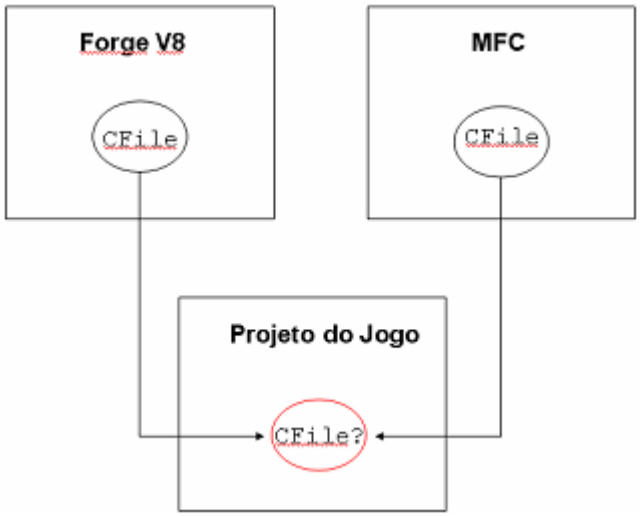

Figura 3-3 Colisão de nomes no *Forge V8*

Embora as soluções acima sejam de fácil implementação, a execução delas na fase em que o problema foi identificado (após a implementação de grande maioria do sistema) implicaria em modificar toda a nomenclatura (no caso da segunda solução) ou no mínimo, modificar todos os arquivos para inserir as classes em outro espaço de nomes.

3.4.2.2 Multiplataforma

Embora a proposta inicial fosse desenvolver um *framework* que criasse uma independência da tecnologia utilizada, permitindo assim o desenvolvimento baseado em várias tecnologias para o desenvolvimento para jogos (eg. DirectX e OpenGL/OpenAL) e possibilitando a criação de um motor para várias plataformas, na prática somente o

DirectX, que era aplicado em mais de 90% dos jogos para PC na época [7], foi utilizado. O desenvolvimento de um motor em paralelo com um jogo levou a algumas simplificações no projeto, pois era preciso ter um *framework* funcional para poder começar a implementar um jogo. Embora os principais módulos tenham ficado independentes dos demais, não foi possível conseguir uma transparência total da tecnologia utilizada. Uma boa prova disso é que alguns métodos das interfaces têm como parâmetros ponteiros para estruturas utilizadas no DirectX.

Nenhum estudo comparativo foi feito sobre as características providas pelas principais bibliotecas de desenvolvimento de jogos para PC, e qual o esforço em desenvolver um *framework* que se baseia em mais de uma tecnologia. O principal problema em se usar mais de uma tecnologia é que cada uma provê um conjunto de característica e funcionalidades, algumas em comum, outras não. As características em comum geralmente podem ser utilizadas sem maiores problemas, sendo talvez preciso apenas algumas adaptações no mecanismo de utilização. Contudo, as características presentes apenas em uma tecnologia geralmente não podem ser incorporadas diretamente ao motor, pois necessitam, por razões de performance, do acesso direto ao *hardware*, o que torna a implementação bastante complexa.

A Figura 3-4 apresenta uma ilustração simples do problema. A tecnologia “A” provê *Mesh*, *Billboarding* e Iluminação [31]; já a tecnologia “B” provê apenas *Billboarding* e Iluminação. Se o *Forge V8* desejasse disponibilizar todas essas funcionalidades em um nível de abstração mais alto, o motor precisaria implementar *Mesh* na implementação que utiliza a tecnologia “B”.

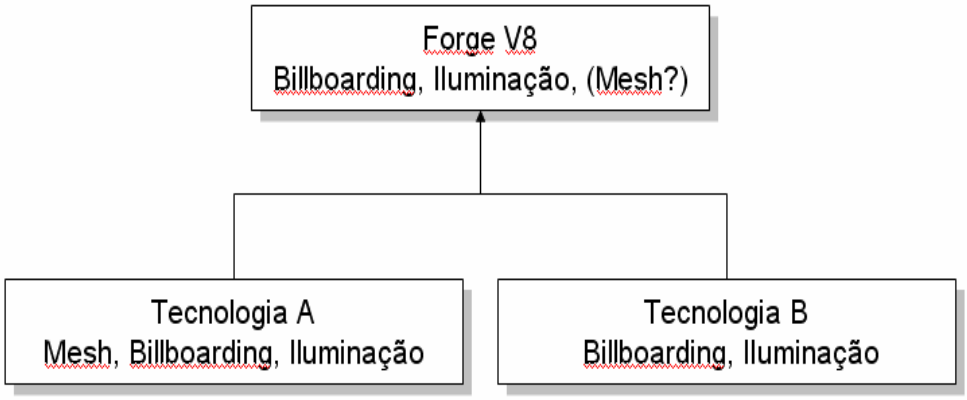

Figura 3-4 Exemplo de utilização de várias tecnologias.

Isto resultou em um motor com uma arquitetura complexa que, no meio do caminho, foi sendo simplificada para atender somente a implementação do DirectX.

3.4.2.3 O Módulo Gráfico

Como já foi apresentado antes, o objetivo principal do *Forge V8* era o de ser um *framework* de desenvolvimento de jogos para PC para ambientes bidimensionais, isométricos e tridimensionais. Isto seria implementado usando-se a tecnologia de renderização tridimensional, sendo possível assim aproveitar o *hardware* de aceleração 3D que existe em grande parte das placas de vídeo atuais.

Logo no início do processo de desenvolvimento, verificou-se que o desenvolvimento dos componentes do motor que serviriam para um jogo tridimensional seria muito complexo e demandaria um esforço muito grande devido à complexidade dos elementos que compõem um cenário 3D [7]. A falta de especialistas neste tipo de problema na equipe de desenvolvimento aumentaria o problema, e por isso, o projeto do jogo *Canyon* foi posto de lado juntamente com os componentes do motor específicos para este tipo de jogo.

Tendo em vista os problemas apresentados no desenvolvimento de ambientes tridimensionais e buscando atingir algum resultado concreto mais rapidamente, decidiu-se então, restringir momentaneamente o domínio dos jogos que poderiam ser desenvolvidos pelo *Forge V8*, para os jogos bidimensionais e isométricos. Entretanto, para aproveitar o

hardware de aceleração tridimensional que a maioria das placas vendidas no mercado possuíam, optou-se por continuar usando a tecnologia gráfica para desenvolvimento de jogos tridimensionais, adaptando-a ao desenvolvimento de jogos bidimensionais e isométricos. Pensava-se que, com esta decisão, seria mais fácil garantir uma boa performance aos jogos, pois a aceleração garantida pelas placas de vídeo impulsionaria o desempenho do jogo.

Este pensamento foi falsamente ratificado após o desenvolvimento de um jogo simples chamado de *Super Tank* (ver a Figura 3-5) utilizando-se o motor. O *Super Tank* pertence à categoria dos jogos isométricos e já havia sido implementado antes por alunos da disciplina de Projeto e Implementação de Jogos [5] do CIn-UFPE. Exatamente por ter sido implementado anteriormente sem o uso do *framework*, o *Super Tank* parecia ser um teste perfeito, pois seria possível compará-lo com a implementação sem a utilização do *Forge V8*. No entanto, o que passou despercebido foi que o *Super Tank* possuía poucos elementos gráficos. Isto dava a falsa impressão que motor possuía uma performance boa, conforme foi mostrado por Charles Madeira em sua dissertação de mestrado [7]. Comparada a versão anterior com a versão que utilizava o *Forge V8*, constatou-se um ganho significativo na produtividade com um pequeno custo na performance.

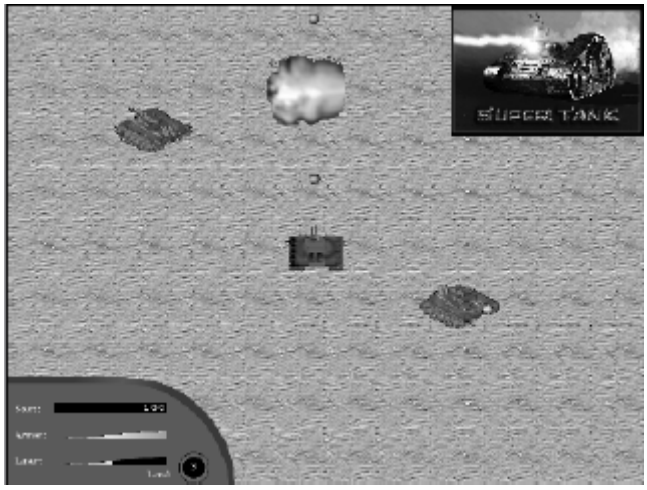

Figura 3-5 Tela do Jogo Super Tank

Posteriormente, testes com um maior número de elementos gráfico foram realizados e demonstraram que quanto maior o número de elementos gráficos, ou seja, o número de triângulos, mais drástica era a perda de performance. O que ficou claro foi que um forte trabalho de otimização do código gráfico 3D ainda se fazia necessário, embora o maior esforço de desenvolvimento tenha sido aplicado a esse módulo. Talvez isso tenha ocorrido pela falta de pessoas especializadas em códigos gráficos tridimensionais na equipe de desenvolvimento.

3.5 CONCLUSÕES

Este capítulo apresentou o *Forge V8*, um motor para desenvolvimento de jogos para PC que foi desenvolvido a partir da idéia de produzir um jogo chamado *Canyon*. O desenvolvimento de tal *framework* foi uma iniciativa pioneira no Centro de Informática da Universidade Federal de Pernambuco e em conseqüência disto, alguns erros de projetos foram cometidos. O problema mais sério encontrado foi a falta de performance e completude do módulo gráfico, resultante da grande complexidade envolvida em desenvolver código para jogos tridimensionais e da falta de pessoal especializado para o desenvolvimento de tal módulo. Embora os problemas encontrados tenham de certa forma comprometido o projeto, de maneira geral o projeto foi importante, pois catalogou as principais dificuldades encontradas no desenvolvimento de tal aplicação.

Como será visto nos próximos capítulos, um novo projeto para a construção de motor para o desenvolvimento de jogos para PC, denominado de *Forge 16V*, foi criado. No projeto do *Forge 16V*, todos os erros encontrados no projeto do *Forge V8* foram levados em consideração, assim como, o catálogo de problemas feito por Charles Madeira no *Forge V8* [7]

Capítulo 4

Conceitos Teóricos dos Jogos Isométricos

Este capítulo apresenta a teoria envolvida no desenvolvimento dos jogos isométricos.

Neste capítulo será mostrado o estudo realizado sobre o desenvolvimento de jogos isométricos. Este estudo foi realizado focando a representação gráfica bidimensional e um cenário baseado em *tiles*. Como será visto no próximo capítulo, estas foram as configurações utilizadas pelo *Forge 16V*.

4.1 PROJEÇÕES ISOMÉTRICAS PARA JOGOS

Para entender melhor os jogos isométricos, é necessário saber o que são projeções axonométricas e isométricas.

As projeções axonométricas são projeções do espaço 3D para o 2D que possuem as seguintes características [43]:

- A projeção no espaço 2D não possui "ponto de fuga" [44];
- Linhas paralelas no espaço 3D continuam paralelas no espaço 2D;
- Objetos que estão distantes possuem o mesmo tamanho de objetos que estão pertos.

Já as projeções isométricas são projeções axonométricas cuja métrica usada nos eixos *x*, *y* e *z* são as mesmas, ou seja, uma unidade no eixo *x* é, em comprimento, igual a uma unidade nos eixos *y* e *z*.

Existem várias projeções isométricas possíveis. Entretanto, os jogos de computadores isométricos são geralmente baseados em *tiles* e é imperativo fazer com que os *tiles* casem para poder formar um mapa de *tiles* [43, 45]. Por isso, geralmente a projeção isométrica utilizada é a conhecida 1:2 (ver Figura 4-1). Nela a altura e o comprimento do *tile* possuem uma razão de 1 para 2, conforme mostrado na superfície superior do cubo apresentado na Figura 4-1. Os tamanhos de *tiles* mais usados nos jogos de computadores são os de 16 *pixels* por 32 e o 32 *pixels* por 64.

Para conseguir exibir um *tile* isométrico que não possui uma forma retangular utilizando-se a representação gráfica bidimensional[1] que utiliza a transferência de mapa de bits retangulares, é necessário o uso da técnica de transparência[1]. Com ela é possível

[1] Maiores informações sobre esta tecnologia no capítulo 2.

realizar a cópia de figuras não retangulares sem que o restante do bloco retangular sobrescreva o conteúdo já existente no destino (ver Figura 4-2).

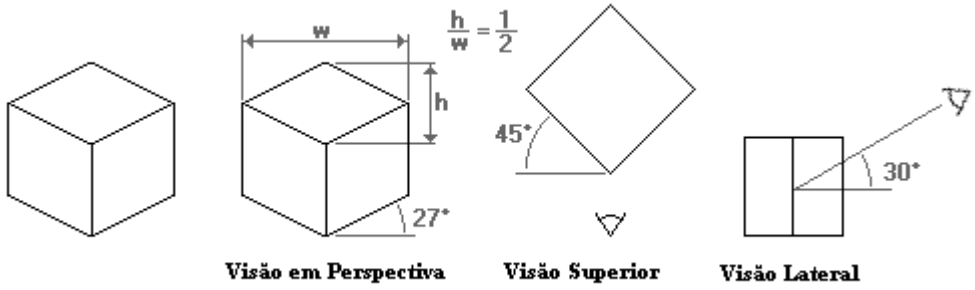

Figura 4-1 Projeção Isométrica 1:2 [2].

Outro ponto importante das projeções isométricas no desenvolvimento de jogos isométricos é o fato de objetos distantes possuírem o mesmo tamanho que objetos pertos. Isso possibilita que o mesmo *sprite* seja utilizado em todo o cenário sem que nenhuma operação de manipulação de imagem seja realizada, ou sem que um *sprite* possua várias versões para distâncias distintas.

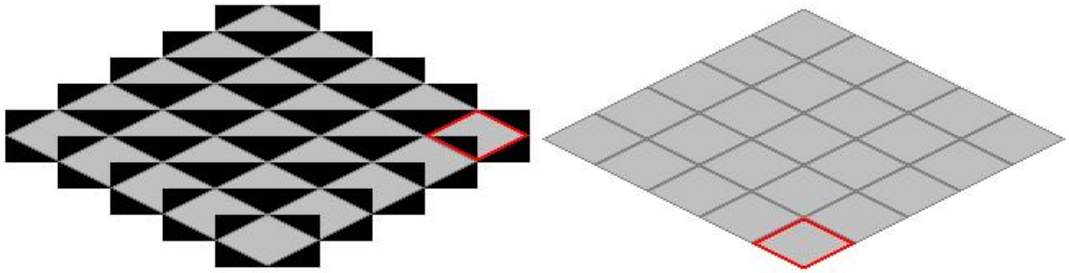

Figura 4-2 Necessidade do uso de transparência no mapa isométrico. À esquerda sem transparência e à direita com transparência.

4.2 TIPOS DE MAPAS ISOMÉTRICOS

Nesta seção serão apresentados os três principais tipos de mapas isométricos [46]:

[2] Figura retirada do *site* http://www.gamedev.net/reference/articles/article1269.asp

- Slide Maps;
- Staggered Maps;
- Diamond Maps.

4.2.1 Slide Maps

O *Slide Map* é possivelmente o mais fácil dos mapas isométricos de se navegar e de renderizar (ver a Figura 4-3). No entanto, ele possui uma aplicação prática limitada por ocupar um espaço muito grande na tela, e por isso poucos jogos utilizam esse tipo de mapa. Contudo, por ser um mapa fácil de lidar, ele é um estudo de casos perfeito para aprender a estabelecer um sistema de coordenadas, a movimentar unidades nos mapas isométricos e de descobrir a posição de um *tile* na tela.

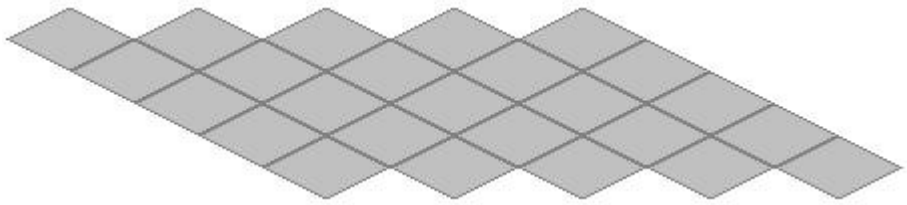

Figura 4-3 Mapa Isométrico do tipo *Slide.*

4.2.2 Staggered Maps

Os *Staggered Maps* (ver Figura 4-4) são mapas isométricos bastante utilizados nos jogos para PC. Jogos como *Civilization II*, *Alpha Centauri* e *Civilization: Call to Power* utilizam este tipo de mapa.

Os mapas do tipo *Staggered* são um dos mais complicados de manipular. No entanto, algumas características os tornam bastante atrativos. Primeiramente, pelo formato quase retangular, esse tipo de mapa é o que menos desperdiça espaço na tela. Existe ainda a possibilidade de cortar as “arestas” do mapa fazendo com que ele tome um formato totalmente retangular (ver Figura 4-5). E ainda, aproveitando o formato retangular apresentado na Figura 4-5, é possível fazer com que o *scroll* do mapa na tela, para a direita ou para a esquerda seja contínuo, onde a coluna de *tiles* mais à direita leva para a mais à esquerda e vice-versa, dando a impressão de um mapa cilíndrico. O mesmo vale para cima

e para baixo. Uma boa aplicação para os mapas cilíndricos é uma representação da Terra, como usado em *Civilization II*.

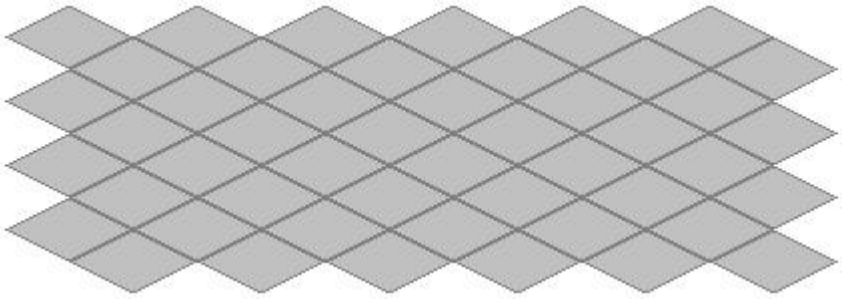

Figura 4-4 Mapa Isométrico do tipo *Staggered.*

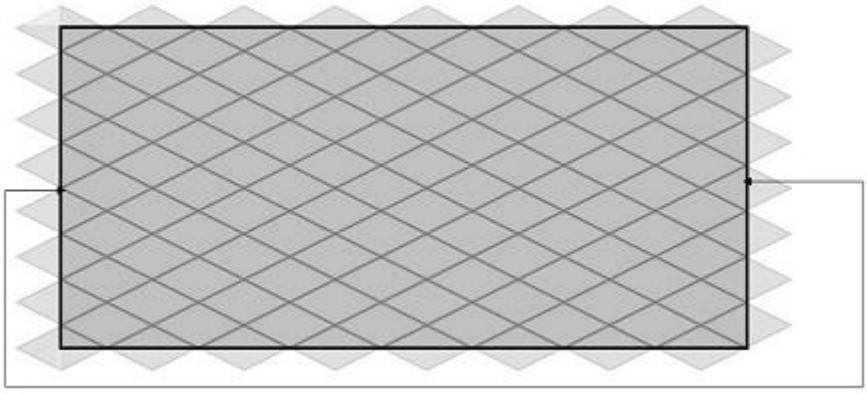

Figura 4-5 Cortando as "Arestas" dos Mapas tipo *Staggered* e tornando-o também um mapa cilíndrico.

4.2.3 Diamond Maps

Os mapas do tipo *Diamod* são um dos mapas isométricos mais utilizados, principalmente pelos jogos de estratégia em tempo real. Jogos clássicos como *Age of Empires*, *Sim City 2000/3000* e *The Sims* usam os mapas *Diamods* que têm este nome devido à forma gráfica(ver a Figura 4-6).

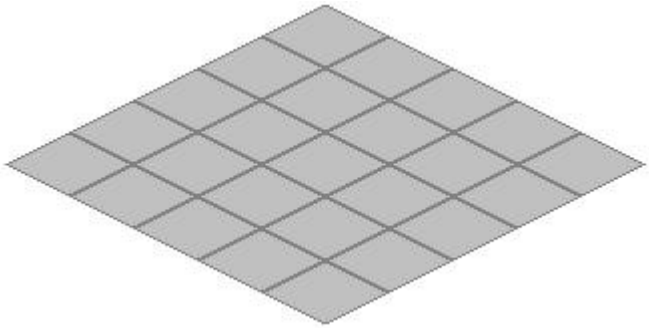

Figura 4-6 Mapa do tipo *Diamond.*

4.3 PRINCIPAIS PROBLEMAS DOS MAPAS ISOMÉTRICOS

Esta seção irá apontar os principais problemas encontrados na utilização dos mapas isométricos e as respectivas soluções. Esses problemas podem ser agrupados em duas categorias: os gerais cuja solução não depende diretamente do tipo de mapa utilizado, e os específicos cuja solução está diretamente ligada ao tipo de mapa. Inicialmente serão apresentados os problemas gerais, e logo a seguir, serão apresentados os problemas específicos para cada tipo de mapa.

A composição de vários *tiles* isométricos forma o que se chama de mapa de *tiles* isométricos. Da mesma forma como ocorre com os mapas retangulares, existe uma relação de 1:1 deste mapa com uma estrutura de dados bidimensional [47] que guarda as informações dos *tiles* e é conhecida como *TileMap*. Entretanto, existem alguns problemas que nos mapas de *tiles* retangulares possuem fácil solução e que no mapa de *tiles* isométricos são bem complicados. O primeiro deles é a ordem em que os *tiles* são plotados. Quando os *tiles* possuem objetos que vão além da fronteira, é necessário seguir uma ordem na renderização, senão a tela pode ficar inconsistente. Esta ordem pode ser descrita pelas seguintes regras [46]:

- *Tiles* precisam ser renderizados de forma que nenhum *tile* seja plotado após outro que está “à frente” dele;
- Se uma pequena porção da tela for atualizada, é necessário atualizar os *tiles* modificados e todos os vizinhos, obedecendo à regra anterior.

É claro que dependendo do tamanho dos objetos que estão nos *tiles*, talvez seja preciso atualizar vários "vizinhos" abaixo, acima e dos lados.

Outro problema dos mapas isométricos é saber como mapear um ponto na tela para uma posição no *TileMap*, ou seja, dado um ponto na tela, a que *tile* ele pertence. Este problema existe devido à forma não retangular dos *tiles* isométricos. Para resolvê-lo, é possível realizar cálculos matemáticos para efetuar a conversão de um espaço para o outro. Entretanto, existe uma solução muito mais prática e rápida ao custo de um pequeno uso de memória. Para um computador é muito fácil verificar em que retângulo de uma grade retangular um ponto está contido. A solução para o problema se baseia nisto. Primeiro divide-se o mundo em retângulos (ver Figura 4-7). Feito isto, fica fácil descobrir a que retângulo o ponto pertence. Descobrindo-se o retângulo e sabendo-se como andar nos *tiles* (problema específico do tipo de mapa - será visto logo a seguir), é fácil descobrir o *tile* central que está no retângulo. Agora o problema se resume a descobrir a qual dos cinco *tiles* que o retângulo cobre, o ponto pertence.

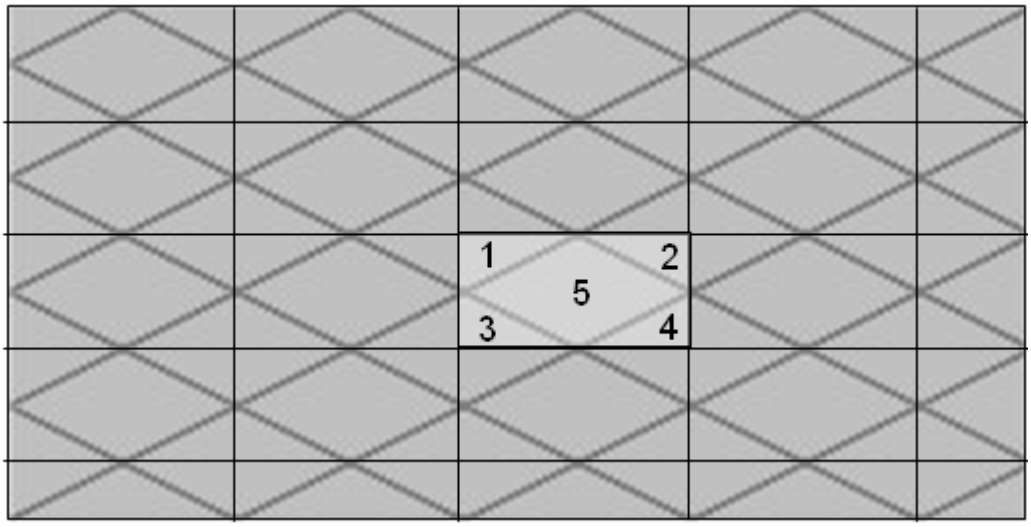

Figura 4-7 Mapa isométrico dividido em retângulos. Dentro de um retângulo existem cinco regiões, cada uma de um *tile* diferente.

Como os *tiles* possuem o mesmo tamanho, é possível construir uma figura externa do mesmo tamanho do retângulo construído anteriormente onde cada *tile* possua uma cor diferente. Assim, para descobrir a que *tile* o ponto pertence, basta mapear o ponto nas coordenadas do retângulo na figura e pegar a cor existente (ver Figura 4-8). Outra

abordagem equivalente para isto é construir no código, uma matriz onde uma posição na matriz equivale a um ponto na figura. Assim, o processo se resume a verificar um valor na matriz.

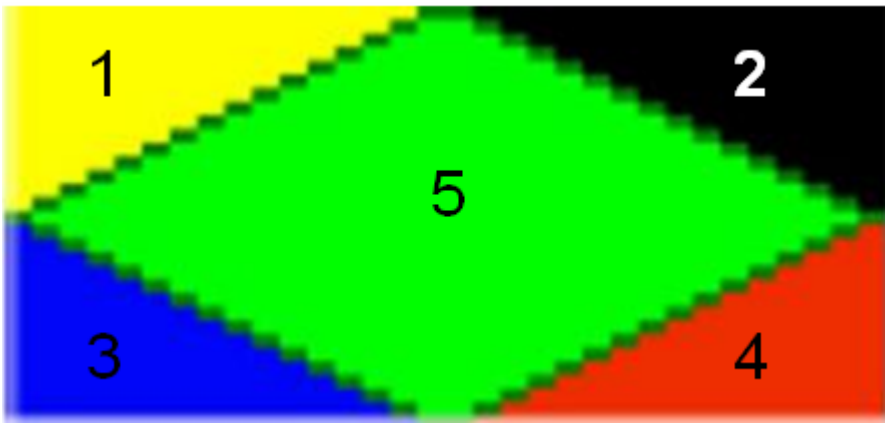

Figura 4-8 Figura que ajuda a descobrir a que *tile* um ponto pertence. Área 1 – Amarelo; Área 2 – Preto; Área 3 – Azul; Área 4 – Vermelho; Área 5 – Verde.

O restante dos problemas encontrados possuem soluções específicas para cada tipo de mapa. Para cada um deles serão mostrados como mapear uma posição do *TileMap* a uma posição na tela, e como se mover no *TileMap* usando uma direção do espaço da tela.

4.3.1 Slide Maps

Algumas variações neste tipo de mapa são possíveis. Contudo, elas são muito parecidas e por isso será mostrada apenas uma delas onde no sistema de coordenadas o *x* cresce para o leste e o *y* cresce para sudoeste. Este é exatamente o primeiro passo: definir o sistema de coordenadas. Nela, cada posição do *TileMap* está mapeada em um *tile* (ver a Figura 4-9.). Uma característica importante deste sistema de coordenadas é que ele facilita a manutenção da ordem de renderização dos *tiles* conforme as regras apresentadas na seção anterior, uma vez que o ponto (0,0) do *TileMap* está na linha superior do mapa.

Para cada tipo de mapa isométrico, existe um grande número de sistemas de coordenadas possíveis. A escolha de um deles influencia diretamente no mapeamento entre os espaços do *TileMap* e da tela e na navegação do mapa. Para não alongar muito o

trabalho, apenas uma possibilidade será adotada. Contudo, os conceitos que serão vistos podem ser aplicados a outras variações do sistema de coordenadas.

Uma vez definido o sistema de coordenadas dos *tiles*, é possível calcular a posição na tela (em *pixels*) dos *tiles* baseado na posição do *TileMap*, já que todos eles possuem a mesma dimensão. Assumindo que a posição (0,0) do *TileMap* esteja mapeada no pixel (0,0) e que *TileWidth e TileHeight* sejam respectivamente a largura e a altura do *tile* isométrico, é fácil verificar que o incremento de uma unidade no eixo *x* do *TileMap* implica no incremento de (*TileWidth*,0*) pixels* na tela (ver a Figura 4-9). Já o aumento de uma unidade no eixo *y* do *TileMap* implica em um aumento de (*TileWidth/2, TileHeight/2*) *pixels* no ponto na tela. Sendo (*MapX,MapY*) uma posição qualquer no *TileMap*, a Tabela 4-1 resume os cálculos feitos e apresenta a equação geral para o mapeamento do *TileMap* para um ponto na tela.

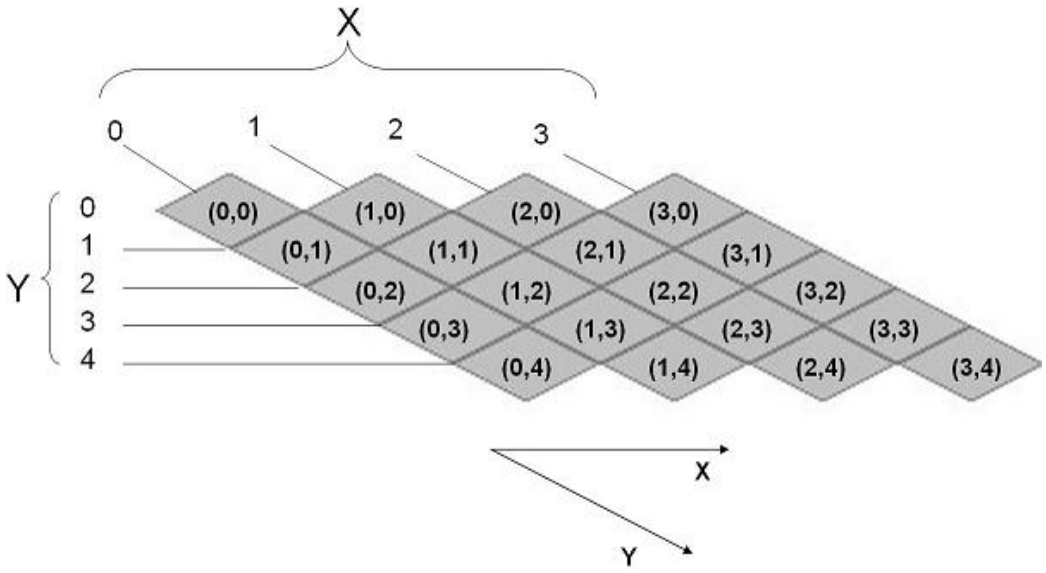

Figura 4-9 Um Sistema de Coordenadas para o *Slide Map.*

O próximo problema a ser resolvido diz respeito à movimentação de objetos na tela, principalmente quando eles não possuem movimento livre, ou seja, precisam se movimentar de um *tile* a outro. Supondo que exista uma unidade no *tile* (1,2) da Figura 4-9 e se deseje movê-la para o norte. Para que posição do *TileMap* deve-se movê-la?

Nos mapas isométricos, existem oito direções “regulares” [48] (ver a Figura 4-10 à esquerda). Para os mapas do tipo *Slide* que utilizam o sistema de coordenadas definido

anteriormente, mover-se para o Oeste, Leste, Sudeste e Noroeste é simples, pois estas direções acompanham os eixos do sistema de coordenadas e basta incrementar ou decrementar uma unidade no *x* ou *y* da posição no *TileMap*.

Pixel	***Incremento em 1 unidade de X do TileMap***	***Incremento em 1 unidade de Y do TileMap***	***Equação***
X	+TileWidth	+TileWidth/2	MapX*TileWidth + MapY*TileWidth/2
Y	0	+TileHeight/2	MapY*TileHeight/2

Tabela 4-1 Mapeando uma Coordenada do *TileMap* para a Tela no *Slide Map*.

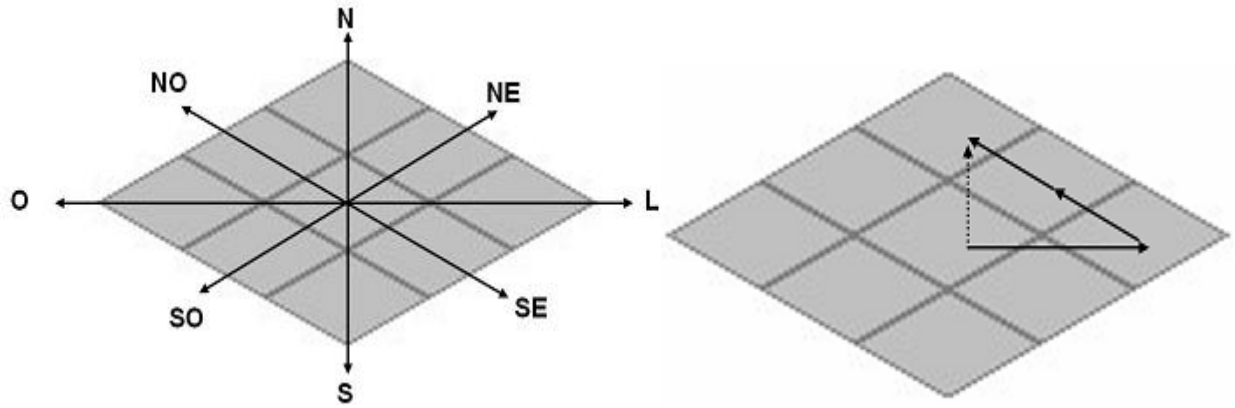

Figura 4-10 Direções Regulares nos Mapas Isométricos (figura à esquerda) e a Direção Norte sendo composta de Direções conhecidas (figura à direita).

Contudo, para se formar o restante das direções é necessário compor as direções já existentes. Por exemplo, para saber como chegar ao *tile* que fica ao norte, basta compor os movimentos para leste e duas vezes para noroeste conforme mostrado na Figura 4-10 à direita. O cálculo fica assim:

- Norte = Leste + Noroeste + Noroeste = (1,0) + (0,-1) + (0,-1) = (1,-2).

Portanto, o *tile* que fica ao norte do *tile* (x,y) é o tile (x+1,y-2). Verifique a Figura 4-9 para confirmar a veracidade dos cálculos. Como calcular o restante das direções é um tarefa repetitiva e fácil de ser realizada, a Tabela 4-2 já apresenta os resultados obtidos.

Direção	***Variação no X do TileMap***	***Variação no Y do TileMap***
Norte	***+1***	***-2***
Sul	***-1***	***+2***
Leste	***+1***	***0***
Oeste	***-1***	***0***
Nordeste	***+1***	***-1***
Noroeste	***0***	***-1***
Sudeste	***0***	***+1***
Sudoeste	***-1***	***+1***

Tabela 4-2 Variação na Coordenada de um *tile* nos *Slide Maps* segundo uma Orientação.

4.3.2 Staggered Maps

O sistema de coordenadas dos *staggered maps* é um dos mais complexos. O eixo *x* cresce para leste como nos *slide maps*, como foi visto anteriormente. A parte incomum é o eixo *y* que cresce para sudeste ou sudoeste dependendo do *tile* em que se esteja, fazendo um movimento em *zigzag* (ver a Figura 4-11). Quando o valor de *y* do *tile* é par, o incremento de *y* em uma unidade leva para sudeste. Já quando o valor de *y* do *tile* é impar, o incremento de *y* em uma unidade leva para sudoeste. Este comportamento garante a forma quase retangular do mapa.

Uma vez definido o sistema de coordenadas, é possível fazer o mapeamento de uma posição no *TileMap* em um ponto na tela. Assumindo-se que a posição (0,0) do *TileMap* está mapeada no *pixel* (0,0) da tela e que *TileWidth e TileHeight* são respectivamente a largura e a altura do *tile* isométrico, o incremento de uma unidade no eixo *x* do TileMap leva a um incremento de (*TileWidth*,0) nos pixels. Já no eixo *y*, o resultado do incremento de uma unidade vai depender da posição atual do *TileMap*. Se o *y* da posição for par, então

o incremento de uma unidade em *y* leva a um incremento de (*TileWidth*/2, *TileHeight*/2) nos *pixels*. Já quando o y da posição for ímpar, o incremento de uma unidade em y leva a um incremento de (-*TileWidth*/2, *TileHeight*/2) em relação à posição anterior. Sendo assim, sendo (*MapX*,*MapY*) uma posição qualquer no *TileMap*, o mapeamento para um *pixel* na tela é dado pela seguinte equação:

$$\begin{cases} \text{Se MapY for par} \rightarrow \text{PixelX} = \text{MapX}*\text{TileWidth} \\ \text{Se MapY for ímpar} \rightarrow \text{PixelX} = \text{MapX}*\text{TileWidth} + \text{TileWidth}/2 \\ \text{PixelY} = \text{MapY}*\text{TileHeight}/2 \end{cases}$$

O mapeamento pode ser verificado na Figura 4-11. A coordenada *y* na tela depende apenas da coordenada *y* no *TileMap*. A coordenada *x* na tela depende do *x* do *TileMap* e se a coordenada *y* é par ou ímpar.

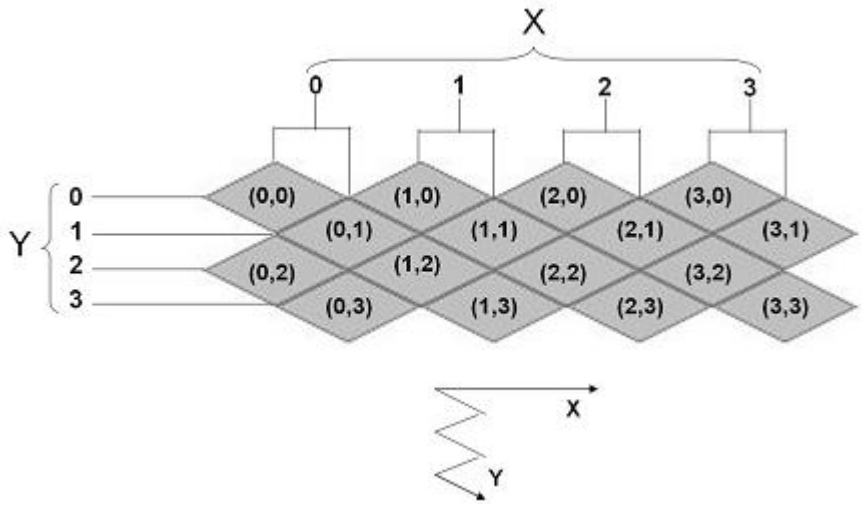

Figura 4-11 Sistema de Coordenadas do Mapa tipo *Staggered.*

A movimentação de objetos na tela também é mais complicada que nos *Slide Maps* e por isto um pouco mais de detalhes do processo será apresentado. O processo para determinar o *tile* é o mesmo, ou seja, compor as direções conhecidas para encontrar as demais. No entanto, uma outra variável deve ser levada em conta: se o *y* da coordenada do *TileMap* é par ou ímpar.

Seguindo o sistema de coordenada estabelecido, a movimentação para Leste e Oeste incrementa ou decrementa somente o valor de *x* na coordenada do *TileMap*, independente se o *y* é par ou ímpar (ver a Figura 4-11). Sendo o valor de *y* par, o incremento de *y* em uma unidade leva para sudeste. Sendo *y* ímpar, o incremento de *y* em uma unidade leva para sudoeste. Estas quatro direções são o ponto de partida para achar as demais.

Uma observação importante é que, quando incrementamos o valor de *y* em uma unidade, o *y* muda a paridade. Assim, quando o *y* é par e é incrementada uma unidade (indo para sudeste), ele passa a ser ímpar. Ou seja, quando ele é ímpar, é feito o movimento contrário que é ir para noroeste, basta decrementar uma unidade de *y*. Este raciocínio pode ser aplicado também para o caso do incremento em uma unidade de *y* quando ele é *ímpar*, o que leva para sudoeste. Neste caso, é fácil deduzir que quando *y* é par, o decremento de *y* leva para nordeste. A Tabela 4-3 apresenta as descobertas feitas até agora.

Direção	***Paridade do Y***	***Incremento em X***	***Incremento em Y***
Leste	**-**	***1***	***0***
Oeste	**-**	***-1***	***0***
Sudeste	***Par***	***0***	***1***
Sudoeste	***Ímpar***	***0***	***1***
Noroeste	***Ímpar***	***0***	***-1***
Nordeste	***Par***	***0***	***-1***

Tabela 4-3 Variação nas Coordenadas do *Tile* nos *Staggered Maps* seguindo uma Orientação – Valores encontrados até agora.

Com os dados da Tabela 4-3, é possível descobrir o restante das direções através da composição como foi usado nos mapa do tipo *Slide*. Para cada caso, faltam quatro direções a serem descobertas: Norte/Sul para ambos os casos, Noroeste/Sudoeste para o *y* par e Nordeste/Sudeste para o *y* ímpar. Como descobrir todos estes valores é uma tarefa repetitiva, será mostrada a solução apenas para o caso do *y* par. A solução para o *y* ímpar pode ser encontrada de maneira análoga.

A direção noroeste para o *y* par pode ser obtida pela composição das direções já conhecidas nordeste e oeste (ver a Figura 4-12 à esquerda). Assim:

- Noroeste = Nordeste + Oeste = (0,-1) + (-1,0) = (-1,-1).

Portanto, o *tile* que fica a noroeste do *tile* (x,y), onde y é par, é o (x-1,y-1). Já a direção sudoeste pode ser obtida compondo as direções sudeste e oeste (ver a Figura 4-12 à direita). Sendo assim:

- Sudoeste = Sudeste + Oeste = (0,1) + (-1,0) = (-1,1)

Portanto, o *tile* que fica a sudoeste do *tile* (x,y), onde y é par, é o (x-1,y+1). Por exemplo, para encontrar o tile que fica a noroeste do *tile* (1,4) na Figura 4-12, basta somar (-1,-1) à posição, ou seja o *tile* (0,3).

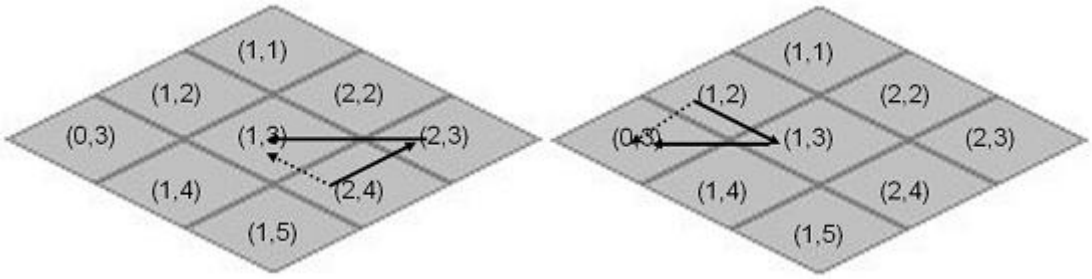

Figura 4-12 Direções Noroeste (à esquerda) e Sudoeste (à direita) nos *Staggered Maps* sendo compostas de direções já conhecidas.

As direções Norte/Sul, como as anteriores, também podem ser obtidas através da composição de direções conhecidas. Contudo, um pouco mais de cuidado é necessário. Por exemplo, a direção norte pode ser obtida com a composição das direções nordeste e noroeste. No entanto, a direção noroeste utilizada parte de um *y* ímpar (ver a Figura 4-13 à esquerda). Fazendo os cálculos:

- Norte = Nordeste + Noroeste(ímpar) = (0,-1)+(0,-1) = (0,-2)

Assim, o *tile* ao norte do tile (x,y), onde y é par, é o (x, y-2). Já a direção sul pode ser obtida através das direções sudeste e sudoeste. Mas, como no caso do norte, a direção sudoeste utilizada é a do *y* ímpar (ver a Figura 4-13 à direita). Então:

- Sul = Sudeste + Sudoeste (ímpar) = (0,1) + (0,1) = (0,2)

Portanto, o *tile* que fica ao sul do *tile* (x,y) onde y é par, é o *tile* (x, y+2). Por exemplo, na Figura 4-13, o *tile* que fica ao sul do *tile* (2,2) é o *tile* (2,4).

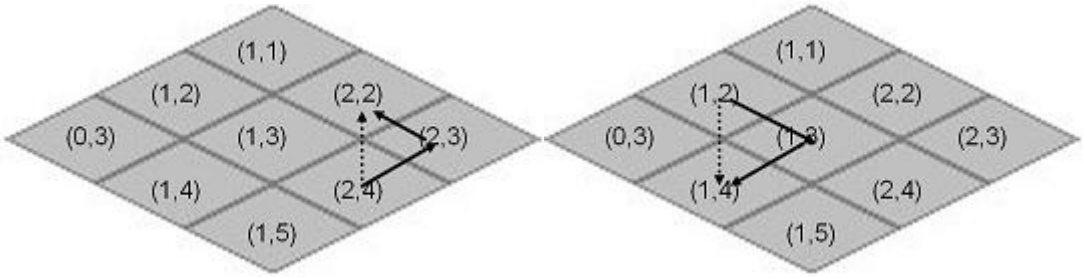

Figura 4-13 Direções Norte (à esquerda) e Sul (à direita) nos *Staggered Maps* sendo compostas de direções já conhecidas.

O restante das direções para um *y* ímpar podem ser obtidas de forma análoga ao caso par. A Tabela 4-4 apresenta o resto dos valores. Os resultados podem ser comprovados na Figura 4-11. Um fato interessante é que embora tenham sido obtidos por caminhos diferentes, os valores das direções Norte e Sul são os mesmos.

Direção	***Paridade do Y***	***Incremento em X***	***Incremento em Y***
Leste	***-***	***1***	***0***
Oeste	***-***	***-1***	***0***
Norte	***Par***	***0***	***-2***
Sul	***Par***	***0***	***2***
Nordeste	***Par***	***0***	***-1***
Noroeste	***Par***	***-1***	***-1***
Sudeste	***Par***	***0***	***1***
Sudoeste	***Par***	***-1***	***1***
Norte	***Ímpar***	***0***	***-2***
Sul	***Ímpar***	***0***	***2***
Nordeste	***Ímpar***	***1***	***-1***
Noroeste	***Ímpar***	***0***	***-1***
Sudeste	***Ímpar***	***1***	***1***
Sudoeste	***Ímpar***	***0***	***1***

Tabela 4-4 Variação nas Coordenadas do *tile* nos *Staggered Maps* seguindo uma Orientação.

4.3.3 Diamond Maps

Como nos mapas anteriores, existem várias possibilidades para o sistema de coordenadas. A que será adotada no restante deste documento tem origem no *tile* mais alto, o eixo *x* cresce para sudeste e o *y* cresce para sudoeste (ver a Figura 4-14).

Como nos outros tipos de mapas isométricos, uma vez definido o sistema de coordenadas, é possível realizar o mapeamento de um *tile* em um *pixel* na tela. Ao contrário dos mapas vistos até agora, os *diamods maps* possuem dois eixos diagonais. Assim, sendo *TileWidth e TileHeight* a largura e altura do *tile* isométrico, o aumento de uma unidade no eixo *x* da coordenada do *TileMap* significa um incremento de (*TileWidth/2,TileHeight/2)* no ponto na tela. Para o *y*, o aumento de uma unidade significa um aumento de (*-TileWidth/2*, *TileHeight/2*). Portanto, assumindo que (*MapX*,*MapY*) é uma posição no *TileMap*, a equação do mapeamento do *TileMap* para a tela é dado por:

- PixelX = (MapX-MapY)*TileWidth/2
- PixelY = (MapX+MapY)*TileHeight/2

É claro que esta equação mapeia o ponto (0,0) no pixel (0,0) e isto vai fazer com que parte do mapa fique fora da tela haja vista que a origem do sistema de coordenadas fica no centro. Isto pode ser corrigido com um *scroll* ou simplesmente adicionando uma constante à equação do eixo *x*.

O sistema de coordenadas proposto é importante, pois ajuda a manter a ordem que os *tiles* serão exibidos na tela. Nos mapas anteriores, o *x* crescia para leste o que facilitava muito manter as regras apresentadas na seção 4.2. Com os *diamonds maps* isto não ocorre. Por isto, manter a origem do sistema no ponto mais alto da tela ajuda a não violar as regras estabelecidas.

Ao contrário dos *staggered maps*, determinar qual *tile* está a uma determinada direção do *tile* atual é um problema relativamente simples nos *diamond maps*. Quatro direções são conhecidas: Nordeste, Sudeste, Noroeste e Sudoeste. Estas direções acompanham os eixos do sistema de coordenadas e as restantes podem ser encontradas pela composição delas. Para evitar a repetição, será mostrada apenas a solução para duas das quatro direções que faltam.

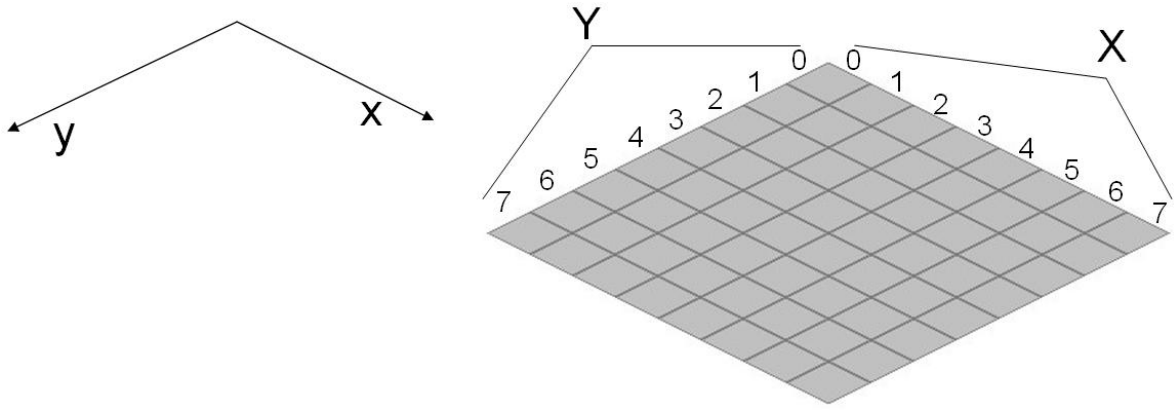

Figura 4-14 Sistema de Coordenadas do Mapa do tipo Diamond

A direção norte pode ser encontrada compondo-se as direções nordeste e noroeste (ver a Figura 4-15 à esquerda). Como as direções nordeste e noroeste seguem os eixos de coordenada, pode ser feito o seguinte cálculo:

- Norte = Nordeste + Noroeste = (0,-1) + (-1,0) = (-1,-1).

Ou seja, se (x,y) é uma posição qualquer no *TileMap*, o *tile* que fica imediatamente ao norte é o de posição (x-1,y-1).

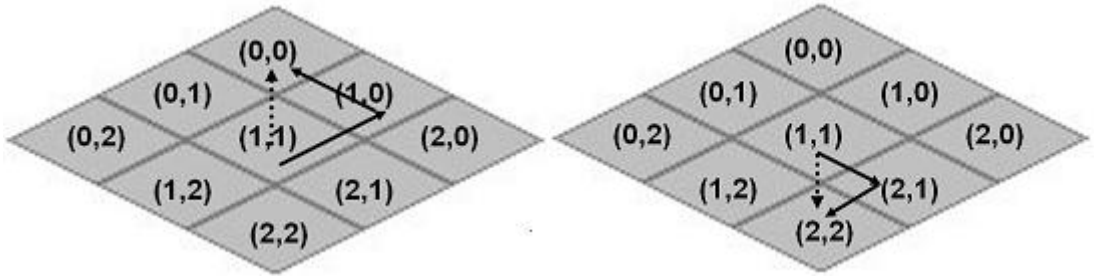

Figura 4-15 Direções Norte (à esquerda) e Sul (à direita) nos Diamond Maps sendo compostas de direções já conhecidas

Já a direção sul pode ser encontrada compondo-se as direções sudeste e sudoeste (ver a Figura 4-15 à direita). Assim:

- Sul = Sudeste + Sudoeste = (1,0) + (0,1) = (1,1)

Se (x,y) representa uma posição qualquer no *TileMap*, o *tile* que fica imediatamente ao sul é o (x+1,y+1). A Tabela 4-5 mostra o resultado para todas as direções.

Direção	***Variação em X***	***Variação em Y***
Sudeste	1	0
Sudoeste	0	1
Noroeste	-1	0
Nordeste	0	-1
Norte	-1	-1
Sul	1	1
Leste	1	-1
Oeste	-1	1

Tabela 4-5 Variação nas Coordenadas do *Tile* nos *Diamond Maps* seguindo uma orientação

4.4 CONCLUSÕES

Este capítulo apresentou a teoria estudada sobre os jogos isométricos para desenvolver o *Forge 16V*. Alguns dos principais problemas encontrados no desenvolvimento deste tipo de jogo foram mostrados, assim como as soluções para cada um deles. Também foram mostrados os principais tipos de mapas isométricos: *Slide*, *Staggered* e *Diamond*.

Capítulo 5

Forge 16V – O Framework

Este capítulo apresenta o Forge 16V, mostrando os módulos que compõem o sistema. Além disso, para cada módulo, serão apresentados os principais problemas encontrados.

Este capítulo apresentará o *Forge 16V*, um *framework* para o desenvolvimento de jogos isométricos. Primeiramente serão apresentadas as principais motivações que levaram ao desenvolvimento deste motor. Em seguida, serão apresentados os módulos que compõe o sistema apresentando a função de cada um deles no motor. Além disso, serão apontados os principais problemas encontrados em cada um dos módulos e as respectivas soluções utilizando-se padrões de projetos [49], continuando assim o trabalho original de enumeração de problemas iniciado no *Forge V8* [7].

5.1 MOTIVAÇÕES, OBJETIVOS E ESCOLHAS DE PROJETO

O projeto do *Forge 16V* surgiu do interesse em dominar a tecnologia de motores para o desenvolvimento de jogos para PC e da necessidade que o Centro de Informática da UFPE tinha de possuir um *framework* que pudesse ser utilizado na disciplina de Projeto e Desenvolvimento de Jogos e por empresas locais de jogos.

Como foi mostrado no capítulo 3, um primeiro esforço foi realizado pelo grupo de jogos do Centro de Informática com o projeto do *Forge V8* [7], um *framework* para o desenvolvimento de jogos e aplicações multimídia. Charles Madeira, o coordenador do *Forge V8*, realizou um ótimo trabalho de pesquisa dos problemas inerentes ao domínio de *frameworks* para jogos e catalogou as possíveis soluções em forma de padrões de projeto. Contudo, por ser um projeto pioneiro, problemas levaram ao desenvolvimento de um *framework* que tinha sérios problemas de performance e na prática não servia para a implementação de jogos que possuíam um grande número de objetos na tela.

Tentando não desperdiçar o trabalho realizado anteriormente, resolveu-se então iniciar um novo projeto, o *Forge 16V*, que levasse em consideração desde a concepção, todo o estudo já realizado anteriormente, a experiência adquirida anteriormente, e mesmo algumas soluções já adotadas no *Forge V8*. O principal objetivo do novo projeto era ter um motor que pudesse ser utilizado na prática e, além disso, pudesse continuar o estudo dos problemas na construção de *frameworks* iniciado no *Forge V8*.

Para conseguir atingir os objetivos traçados, algumas escolhas de projeto tiveram que ser feitas, possibilitando assim que o *Forge 16V* tomasse um caminho diferente do projeto antecessor. As principais escolhas foram:

- Desenvolvimento incremental;
- Desenvolver um motor para os jogos com cenários isométricos e bidimensionais;
- Desenvolver um motor utilizando transferências de *bits* (ver seção 2.4.3.1) ao invés de técnicas 3D;
- Usar somente a biblioteca multimídia DirectX, ou seja, desenvolver um motor para jogos que rodem somente na plataforma Windows;
- Não trabalhar com código *multithreaded* nesta primeira fase do projeto.

A primeira escolha e talvez a mais importante, diz respeito à política de desenvolvimento. Partindo de uma arquitetura modular, resolveu-se adotar a política do desenvolvimento incremental e da aprendizagem gradativa, ou seja, definir metas inicialmente mais simples para ter um motor em plenas condições de funcionamento o mais rápido possível e, a partir daí, incrementá-lo.

Um dos problemas do *Forge V8* era que o motor tentava atender a um grande número de tipos de jogos, ou seja, jogos bidimensionais, isométricos e tridimensionais. Visando possuir um produto final em um curto espaço de tempo e dispondo de apenas um desenvolvedor na fase atual do projeto, resolveu-se restringir o domínio de jogos que são possíveis desenvolver com o *Forge 16V* para os jogos isométricos e bidimensionais. Esta medida reduz bastante o esforço de programação e ainda assim possibilita que um grande número de jogos possa ser desenvolvido com o motor [46].

Um dos principais problemas do *Forge V8* foi a escolha de desenvolver o módulo gráfico utilizando a tecnologia tridimensional de renderização gráfica (ver o capítulo 2 para obter maiores informações sobre as tecnologias de renderização gráfica disponíveis), mesmo não dispondo de especialistas em computação gráfica na equipe de desenvolvimento. Isto, além de aumentar muito a complexidade do módulo gráfico, levou a problemas de performance que inviabilizaram a utilização do motor na prática. No *Forge 16V*, por dispor de apenas um programador que também não tinha experiência em computação gráfica, resolveu-se utilizar, na primeira fase do projeto, a tecnologia de renderização bidimensional, ou seja, transferência de bits entre áreas de memória. Esta tecnologia não possui uma complexidade tão grande quanto a tridimensional, possibilitando assim ter um produto final mais rápido e com uma maior facilidade de obter um melhor

desempenho gráfico (o código não necessita de tantas otimizações). No futuro, devido à arquitetura modular do *Forge 16V*, será possível incorporar a renderização tridimensional sem maiores problemas de integração.

Outro problema encontrado no *Forge V8* foi o desenvolvimento de uma arquitetura que possibilitasse a utilização de várias tecnologias, como por exemplo, o DirectX e o OpenGL. Enquanto do ponto de vista da engenharia de *software* isto parece atraente, na prática isto representou um passo muito grande para uma equipe que nunca tinha desenvolvido um *framework* para o desenvolvimento de jogos. O principal problema foi entender a fundo ambas as tecnologias e balancear as características providas por cada uma para chegar a um conjunto comum. No *Forge 16V* resolveu-se então utilizar apenas o DirectX, para simplificar o *framework*.

Por fim, resolveu-se não utilizar código *multithreaded* na primeira fase do projeto para simplificar a implementação, haja vista que problemas de concorrência teriam que ser tratados.

5.2 ESTRUTURAÇÃO DO FORGE 16V

Devido à grande dinâmica que envolve a área de jogos, o *Forge 16V* foi concebido com uma arquitetura modular possibilitando agregar novas funcionalidades ao motor no futuro.

Para compreender melhor a estrutura do motor, é necessário entender a estrutura do programa de um jogo em geral. Utilizando-se apenas um único *thread* [29] de execução, o programa do jogo é composto de um laço contínuo que executa uma determinada lógica e apresenta o resultado ao usuário [28]. Uma visão geral da estrutura de um jogo pode ser encontrada na Figura 5-1.

Na fase de inicialização são realizadas operações de preparação para executar o jogo, tais como, alocação de memória, aquisição de recursos e leitura de dados armazenados no disco. Já o laço principal é responsável pelo gerenciamento e execução dos diversos componentes da aplicação. A aplicação continua executando esse laço indefinidamente até que o usuário resolva sair do jogo. O tratamento de eventos é responsável por coletar os eventos advindos do sistema operacional e da rede de

computadores e disponibilizá-los para o jogo. A operação de entrada de dados é responsável por coletar e processar os eventos originados dos diversos dispositivos de entrada, como por exemplo, o teclado, *mouse* e *joystick*. No módulo de lógica está presente a maior parte do código variável de um jogo para o outro. Ele é responsável pela inteligência artificial (IA), modelagem física, detecção de colisão e as regras que governam o mundo contido no jogo propriamente dito. O módulo do mundo interage com o módulo de rede para sincronizar o mundo do jogo com os diversos participantes remotos. Já a renderização gráfica é responsável por apresentar uma representação gráfica para o estado do mundo naquele instante, gerando assim a cada iteração, um quadro que faz parte da animação do jogo. O módulo de som é responsável pelos efeitos sonoros e melodias, tornando o jogo mais realista. Por fim, devido à grande diferença do *hardware* que será utilizado para rodar o jogo, é necessário realizar uma operação de sincronização para garantir que a aplicação rodará a uma taxa satisfatória (geralmente 30 quadros por segundo) e constante. Quando o usuário deseja sair do jogo, é necessário que a aplicação libere todos os recursos que foram utilizados pelo jogo.

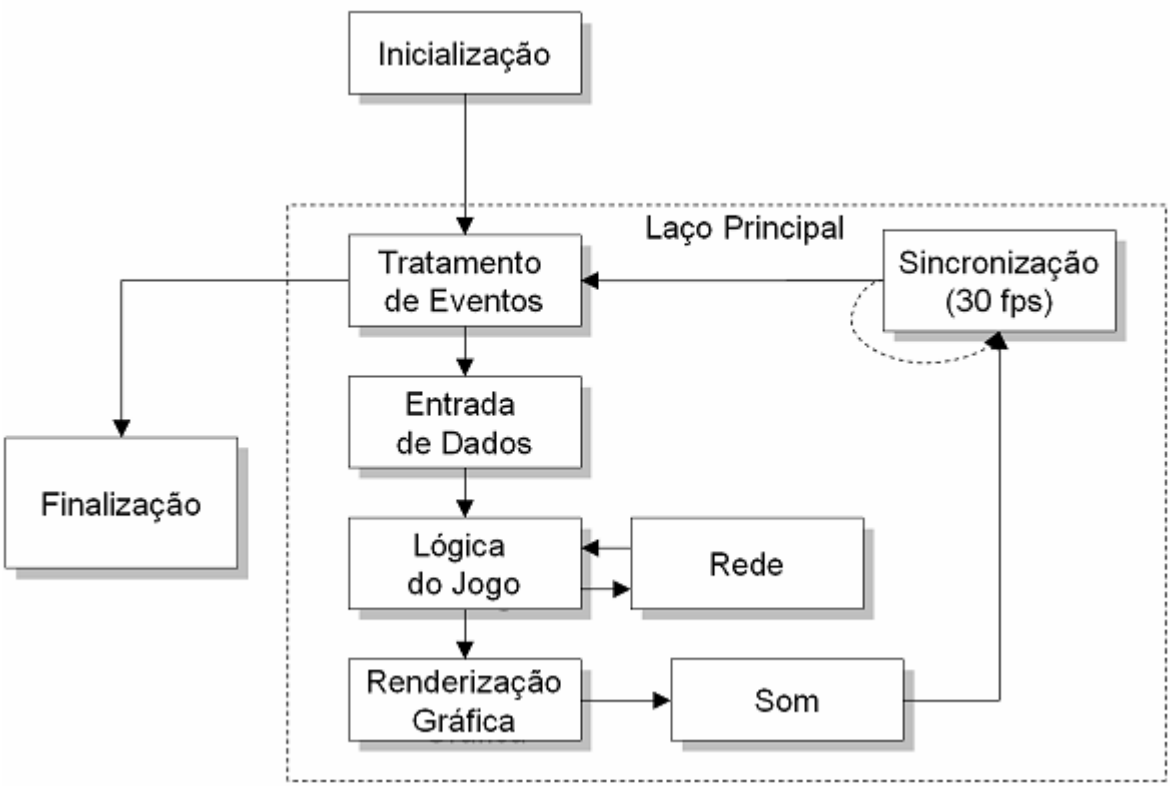

Figura 5-1 Estrutura Geral de um Jogo utilizando-se um único *thread* de execução

Quando se utilizam as mesmas tecnologias no desenvolvimento dos jogos, grande parte das operações citadas anteriormente se repete de um jogo para o outro. Por exemplo, o tratamento de eventos do sistema e dos dispositivos de entrada não muda de uma aplicação para outra. O mesmo acontece com a renderização gráfica quando a tecnologia utilizada, o tipo de cenário (ver o capítulo 2 para os tipos de cenários e tecnologias de renderização) e a representação do mundo são as mesmas. Observando isto, resolveu-se então categorizar as principais operações realizadas pelos jogos [50, 51]. e tomando como base o trabalho já realizado no *Forge V8* e no wGEM [6] (outro motor desenvolvido pelo grupo de jogos do CIn-UFPE, só que para dispositivos móveis), resolveu-se então propor a estrutura para o *Forge 16V* que aparece na Figura 5-2. Os seguintes módulos compõem o motor*:*

- Gerenciador Principal;
- Gerenciador Gráfico;
- Gerenciador de Entrada;
- Gerenciador de Som;
- Gerenciador de Multiusuários;
- Gerenciador de *Log*;
- Gerenciador do Mundo;
- Gerenciador de Modelagem Física;
- Gerenciador de IA;
- Editor de Cenários;

Embora esta estrutura tenha sido baseada nas do *Forge V8* e do wGEM [6], algumas funcionalidades não previstas anteriormente nestes motores foram acrescentadas, como por exemplo, um módulo de *log* que facilita a depuração da aplicação.

O restante desta seção descreve o papel de cada um dos módulos, mostrando também os principais problemas encontrados e as respectivas soluções em forma de padrões de projetos, ajudando assim, a melhor descrever o *framework* [49]. Este catálogo de problemas foi inicialmente realizado tomando-se como base os problemas já detectados no *Forge V8* [7] e que também apareciam no *Forge 16V*. À medida que a fase de implementação avançava e novos problemas iam sendo identificados, as soluções em forma

de padrões de projeto iam sendo incluídas neste catálogo, continuando assim o trabalho começado no *Forge V8*. Este catálogo é de grande valia pois existe pouca literatura sobre os problemas encontrados no desenvolvimento de motores para jogos, pelo menos que se tenha conhecimento. Além disso, uma das vantagens de possuir tal catálogo é que as soluções adotadas podem ser aproveitadas em outros projetos que pertençam ao mesmo domínio uma vez que os problemas tendem a ser muito parecidos.

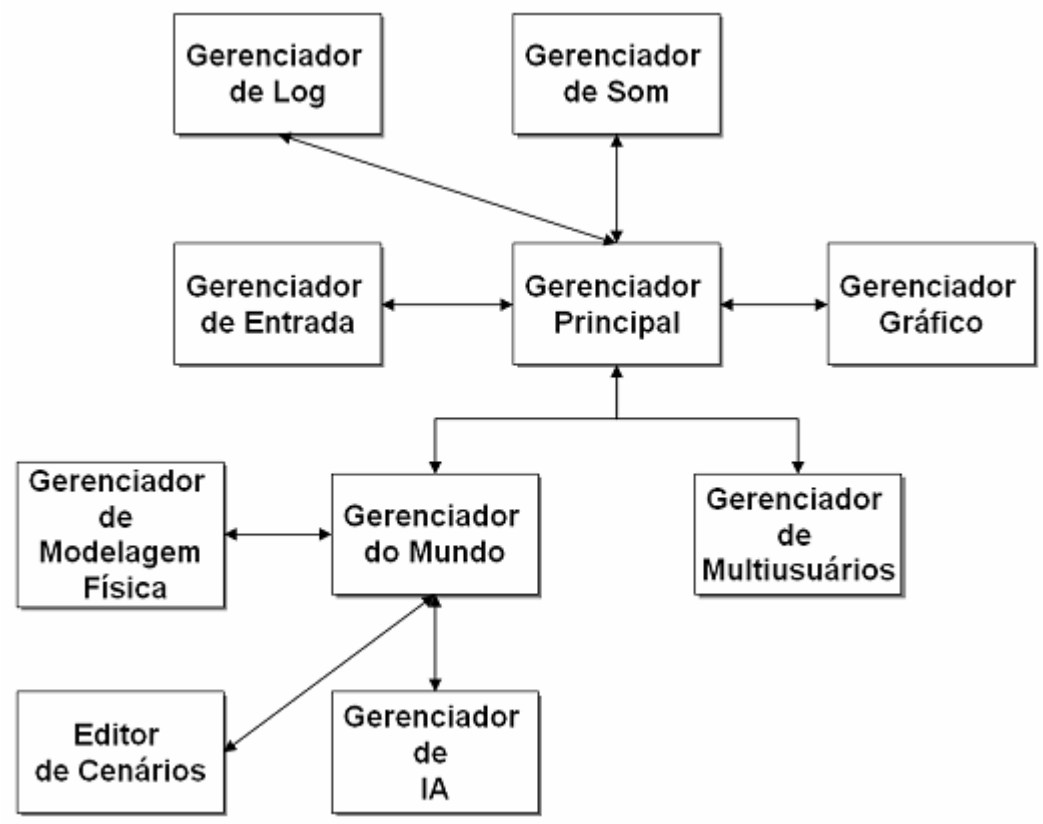

Figura 5-2 Estrutura do *Forge 16V*

Na listagem dos problemas de cada módulo, três situações podem ocorrer:

1. O problema já havia sido identificado pelo *Forge V8* em um módulo semelhante;
2. O problema já havia sido identificado pelo *Forge V8* mas em uma situação diferente;
3. O problema foi identificado por este trabalho.

Para distinguir o que realmente foi contribuição deste trabalho e o que foi contribuição do *Forge V8*, quando ocorrer a situação 1 será feita a referência ao trabalho do *Forge V8*, ao final do texto contendo o problema identificado.

5.2.1 Gerenciador Principal

O módulo gerenciador principal é encarregado de lidar com a instanciação e destruição dos demais módulos e recursos necessários ao motor. Além disso, o gerenciador principal é responsável pelo gerenciamento da execução de código dos outros módulos e pelo gerenciamento da temporização do laço principal (ver a Figura 5-1), com a finalidade de garantir que após a execução do código de cada módulo, o jogo apresentará uma taxa de execução constante e adequada para uma boa jogabilidade. Outra responsabilidade do módulo do gerenciador principal é a de tratar eventos advindos do sistema operacional para a aplicação, tais como, maximização e minimização da janela da aplicação.

Através do gerenciador principal é possível acessar todos os módulos do *Forge 16V*. Isto faz com que o gerenciador principal seja o local ideal para ser a principal interface entre o motor e o código de desenvolvimento do jogo.

Principais Problemas Encontrados

1. O módulo do gerenciador principal deve ser inicializado uma única vez e possuir uma única instância durante toda a execução do jogo. Esta instância deve estar acessível em qualquer parte do programa através de um ponto de acesso conhecido;
2. Ele deve redirecionar todos os eventos que vieram do sistema operacional para os demais módulos do sistema e seus respectivos manipuladores de eventos (*handlers)*, evitando ao máximo o acoplamento entre os módulos. Por exemplo, quando o módulo recebe um evento de um dispositivo de entrada como o *mouse*, ele não sabe como tratá-lo, então ele deve repassar o evento para outros membros da hierarquia. Quando o evento chegar ao módulo de entrada, ele será tratado [7];
3. Este módulo deve notificar o resto do sistema de modificações ocorridas no estado da aplicação, como por exemplo, se a aplicação perdeu o foco, os seja, o usuário passou a utilizar outra aplicação;

4. Em algumas máquinas existe um *hardware* para a temporização de alta resolução que permite uma manipulação do tempo mais precisa. Quando existir este hardware, o subsistema de temporização deve utilizá-lo. Para o módulo, isto deve ser transparente;
5. Como será visto no próximo capítulo, um jogo é composto de vários estados. Um bom exemplo disso é que a maioria dos jogos possui uma tela de menu principal onde se deseja ter um tratamento dos eventos de teclado diferente do que quando se está na tela do jogo propriamente dito. Como é este módulo que coordena o laço principal do jogo (ver a Figura 5-1), ele deve proporcionar uma forma do desenvolvedor especificar diferentes comportamentos dependendo do estado atual.

Soluções

1. O problema (1) pode ser resolvido com o padrão de projeto ***Singleton*** [52]. Este padrão é um dos mais usados no *Forge 16V,* e ele garante que uma determinada classe vai possuir uma única instância em toda a execução da aplicação. Além disso, o padrão provê um único ponto de acesso global à instância de uma determinada classe, possibilitando assim que ela esteja acessível em qualquer ponto de aplicação;
2. O problema (2) pode ser resolvido utilizando-se o padrão de projeto ***Chain of Responsability*** [52]. Este padrão usa uma estrutura hierárquica para passar a responsabilidade do tratamento do evento para os *handlers* que estão em níveis mais baixos na hierarquia, evitando assim o acoplamento do módulo principal aos demais módulos;
3. O problema (3) pode ser resolvido com o padrão de projeto ***Observer*** [52]. Os módulos que desejem ser notificados de mudanças no estado da aplicação devem se cadastrar. Quando alguma mudança ocorrer, o módulo principal envia uma mensagem aos módulos cadastrados, informando o evento;
4. O problema (4) pode ser resolvido utilizando-se o padrão de projeto ***Factory*** [52]. Assim é possível tornar transparente ao módulo, qual o subsistema de tempo está sendo utilizado;
5. O problema (5) pode ser resolvido com o padrão de projeto ***State*** [52] que atribui os diferentes comportamentos a objetos diferentes.

Diagramas

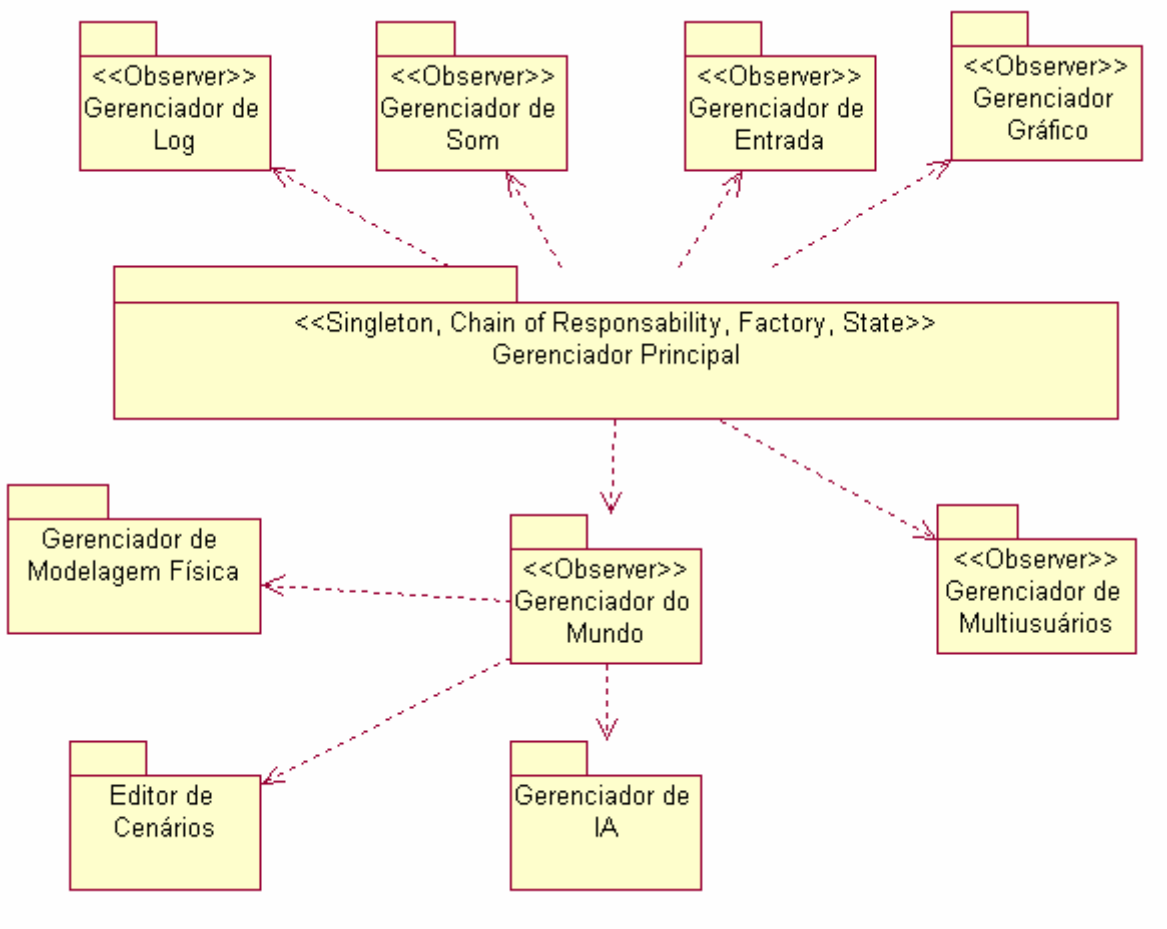

Figura 5-3 Diagrama do gerenciador principal.

5.2.2 Gerenciador Gráfico

O principal papel do gerenciador gráfico é o de verificar o estado atual do mundo e gerar uma representação gráfica para ele. Além disso, o gerenciador gráfico é o responsável por apresentar um conjunto de elementos gráficos de iteração com o usuário, como janelas, *textos, listboxes, comboboxes* [53](ver a Figura 5-4), etc. Quem controla quando o código do gerenciador gráfico deve ser executado para exibir na tela os elementos gráficos do jogo é o módulo do gerenciador principal.

Como foi mencionado na seção 5.1, o *Forge 16V* utiliza a tecnologia de renderização bidimensional, o que possibilita a construção de cenários 2D e isométricos.

Portanto, os principais componentes utilizados por este módulo são: *Surfaces* , *Sprites e Tilesets* [28]. É função deste módulo gerenciar os elementos gráficos utilizados na construção do cenário. Um problema típico que este módulo precisa gerenciar é a utilização da memória de vídeo. Uma figura pode ser armazenada na memória convencional do computador que, para operações gráficas, é muito lenta, ou na memória da placa de vídeo que é muito rápida, mas escassa na maioria dos computadores. Um bom *framework* deve tornar o processo de gerenciamento de memória dos objetos gráficos transparente do programador que está utilizando o motor. Para o programador, ele só precisa criar um objeto *Surface* e deixar o motor decidir aonde vai guardá-lo.

Figura 5-4 Elementos de Interface Gráfica do Jogo "Age of Wonders Shadow Magic" [30]

Principais Problemas Encontrados

1. O módulo do gerenciador gráfico deve ser inicializado uma única vez e possuir uma única instância durante toda a execução do jogo. Esta instância deve estar acessível em qualquer parte do programa através de um ponto de acesso conhecido;

2. Uma figura pode estar contida em mais de um *Sprite* que representa uma animação. Para poupar espaço em memória, é necessário garantir que um mesmo objeto gráfico possa ser compartilhado sem a necessidade de criar uma nova instância;
3. O módulo do gerenciador gráfico deve gerenciar a utilização da memória da placa de vídeo de forma a maximizar a performance do jogo;
4. O gerenciador gráfico deve permitir que o programador altere a aparência dos elementos de interface gráfica como por exemplo botões, listas e janela (*pluggable look and feel* [54]);
5. Deve existir uma hierarquia dos elementos gráficos para facilitar a estruturação. Por exemplo, uma janela é composta de painéis que por sua vez possuem botões, etc;
6. Este módulo deve lidar com diferentes tipos de imagens, como por exemplo, BMP, JPEG, PNG [55], etc.

Soluções

1. O problema (1) pode ser resolvido com o padrão de projeto ***Singleton*** [52];
2. O problema (2) pode ser resolvido aplicando-se o padrão de projeto ***Flyweight*** [52], que permite que múltiplas instâncias de uma classe se refira a um objeto comum;
3. O problema (3) pode ser resolvido com o padrão ***Cache Manager*** [56]. A idéia é fazer com que todas as figuras sejam armazenadas inicialmente na memória convencional do computador, e à medida que elas vão sendo renderizadas, uma cache é feita na memória de vídeo. E para isto, o padrão *Cachê Manager* se aplica;
4. O problema (4) pode ser resolvido com o padrão de projeto ***Abstract Factory***[52] que permite que a aparência seja trocada mais facilmente.;
5. O problema (5) pode ser resolvido com o padrão de projeto ***Composite*** [52];
6. O padrão de projeto ***Builder*** [52] mostrou-se apropriado para este problema. Com ele, foi possível separar a construção da *Surface* da representação que ela possui no arquivo.

Diagrama

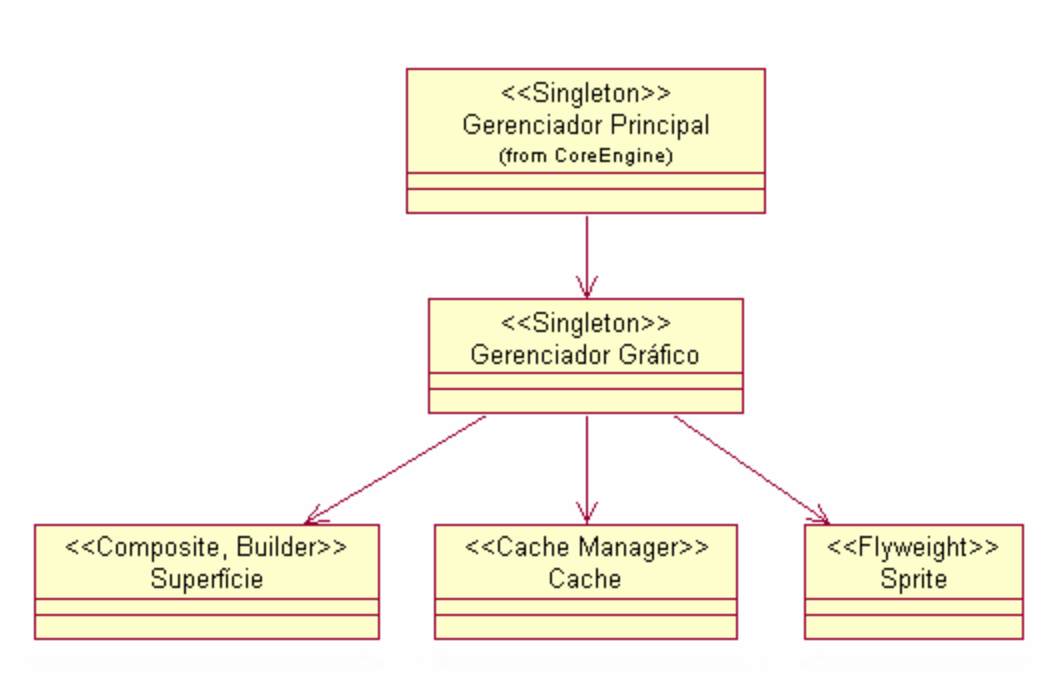

Figura 5-5 Diagrama de classes do gerenciador gráfico.

5.2.3 Gerenciador de Entrada

O módulo de gerenciamento de dispositivos foi criado para cuidar dos dispositivos que são utilizados na interação do usuário com o jogo. Atualmente, os dispositivos mais utilizados com esse intuito são o teclado, o *mouse* e várias variações de *joysticks* como por exemplo, *gamepads* e volantes. Este módulo é responsável por verificar a existência de tais dispositivos, facilitar a configuração deles para os usuários e processar todos os eventos gerados pelos dispositivos.

Principais Problemas Encontrados

1. Deve existir uma única instância do módulo de gerenciador dos dispositivos de entrada e ela deve estar acessível a partir de um ponto conhecido no motor;
2. Este módulo deve tratar os eventos gerados a partir de algum dispositivo de entrada e informar qualquer outro módulo que tenha interesse em saber das modificações realizadas.

Soluções

1. O problema (1) pode ser resolvido utilizando-se o padrão de projeto ***Singleton*** [52];
2. O problema (2) pode ser resolvido com o padrão de projeto ***Observer*** [52]. Assim, cada módulo que deseje receber notificações de mudança de estado nos dispositivos, basta se registrar.

Diagrama

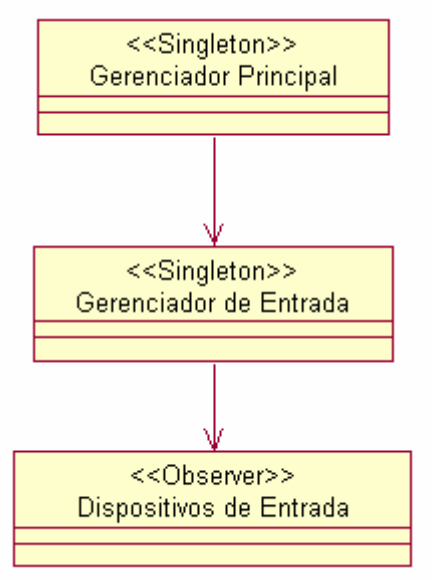

Figura 5-6 Diagrama do Gerenciador de Entrada.

5.2.4 Gerenciador de Log

Um bom sistema de depuração é essencial ao desenvolvimento de qualquer projeto. A principal função do módulo do gerenciador de *log* é exatamente esta, facilitar o monitoramento da aplicação através de mensagens inseridas no código fonte. Estas mensagens, ou algum erro não fatal na aplicação, podem ser enviadas para diferentes dispositivos, como por exemplo, impressoras, arquivos locais, console gráfico ou um computador remoto para que possam ser posteriormente analisadas. Além disso, cada mensagem pode ser categorizada, possibilitando assim estabelecer um nível de interesse para cada dispositivo. Isto possibilita, por exemplo, que sejam guardadas em um arquivo local todas as mensagens emitidas, enquanto que, para o computador remoto, sejam enviadas apenas as mensagens categorizadas como de notificação.

Além disso, através do console gráfico, que pode ser exibido em uma janela na tela do jogo, é possível realizar modificações de configurações tanto no motor quanto no próprio jogo, sem que seja preciso reiniciar a aplicação.

Principais Problemas Encontrados

1. Deve existir uma única instância do módulo de gerenciador de *log* e ela deve estar acessível a partir de um ponto conhecido no motor. Com isto, é possível enviar mensagens de qualquer ponto do código;
2. Cada dispositivo de *log* deve informar ao gerenciador que tipo de mensagem deseja receber.O gerenciador de *log* ao receber uma mensagem deve verificar os dispositivos cadastrados e repassar as mensagens aos dispositivos que desejam recebê-la;
3. No caso do console gráfico, deve existir apenas uma única instância na aplicação;
4. Deve existir uma gramática para verificar a sintaxe dos comandos que podem ser utilizados no console [7].

Soluções

1. Como nos módulos anteriores, os problemas (1) e (3) podem ser resolvidos utilizando-se o padrão de projeto ***Singleton*** [52];
2. O problema (2) pode ser resolvido com o padrão de projeto ***Observer*** [52];
3. O interpretador das sentenças do problema (4) pode ser construído utilizando-se o padrão de projeto ***Interpreter*** [52].

Diagrama

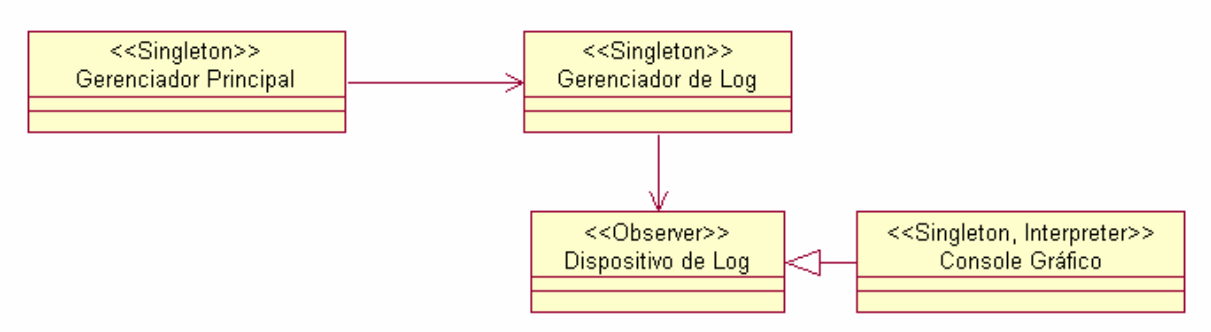

Figura 5-7 Diagrama do Gerenciador de Log.

5.2.5 O Gerenciador de Som

O módulo do gerenciador sonoro é responsável pela leitura e reprodução de sons e músicas através da placa de som. Este módulo também deve gerenciar mixagens de sons, executar efeitos sonoros e controlar parâmetros como por exemplo, volume e posicionamento (sons 3D).

Principais Problemas Encontrados

1. Deve existir uma única instância do módulo de gerenciador de som e ela deve estar acessível a partir de um ponto conhecido no motor;
2. É necessário manter um repositório de músicas e sons que já foram lidos do disco rígido, e na maioria dos casos manter seqüências de sons agregados [7];

Soluções

1. O problema (1) pode ser resolvido utilizando-se o padrão de projeto ***Singleton*** [52];
2. O problema (2) pode ser resolvido utilizando-se o padrão de projeto ***Iterator*** [52] que pode ser usado para prover uma maneira rápida de acessar uma seqüência de sons;

Diagrama

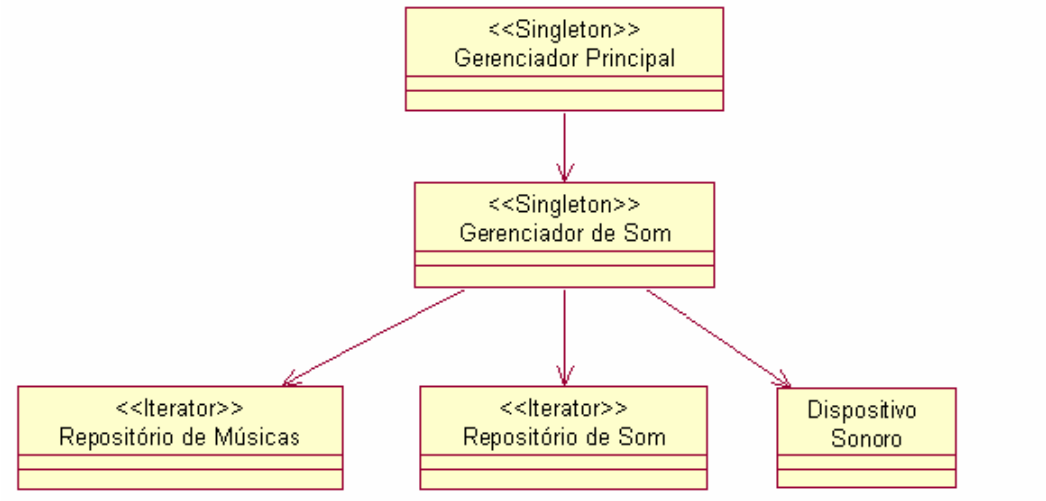

Figura 5-8 Diagrama do Gerenciador de Som.

5.2.6 Gerenciador de Multiusuários

Este módulo é responsável por gerenciar a comunicação através de uma rede de computadores, seja ela uma rede local ou a Internet, e permitir a interação entre vários usuários remotos de um jogo. A conexão entre os vários participantes pode ser estabelecida via modem ou via qualquer outro dispositivo de rede que interligue os computadores.

Este módulo também deve controlar a sessão do jogo e gerenciar a entrada ou a saída de participantes. Além disso, é responsabilidade do módulo multiusuário sincronizar o estado do mundo com o mundo dos demais participantes.

Principais Problemas Encontrados

1. Deve existir uma única instância do módulo de gerenciador multiusuário e ela deve estar acessível a partir de um ponto conhecido no motor ;
2. Deve existir um controle do estado das conexões e sessões estabelecidas [7];
3. Este módulo deve manter sincronizado o estado do mundo dos clientes (jogadores) com o do servidor [7];

Soluções

1. O problema (1) pode ser resolvido utilizando-se o padrão de projeto ***Singleton*** [52];

2. O problema (2) pode ser resolvido utilizando-se o padrão de projeto ***State*** [52] no controle do comportamento das conexões de acordo com a mensagem recebida;
3. O problema (3) pode ser resolvido usando o padrão de projeto ***Gateway*** [57]. Ele permite que uma sincronização entre clientes e servidor seja feita através de um índice de modificação do mundo. Com isso, fica mais fácil gerenciar as modificações realizadas;

Diagrama

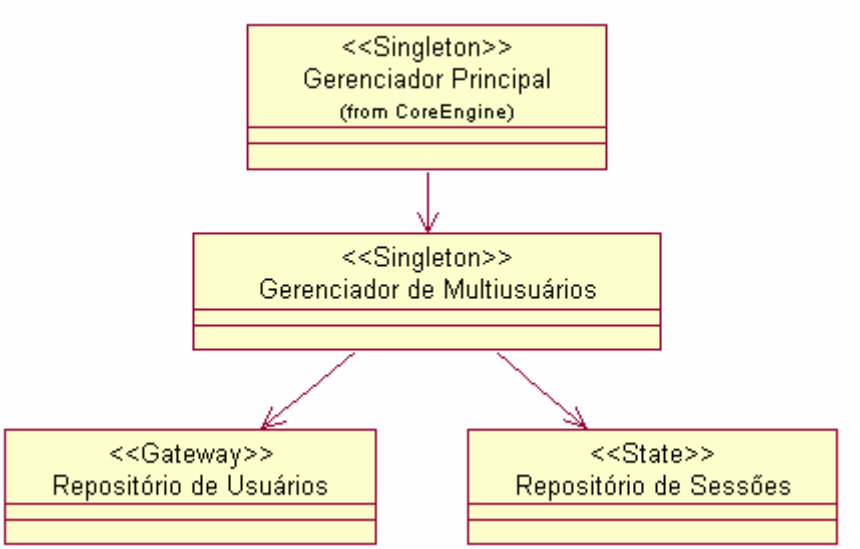

Figura 5-9 Diagrama de classes com o gerenciador de multiusuário.

5.2.7 Gerenciador do Mundo

O gerenciador de mundo é responsável pelo gerenciamento do ambiente e dos personagens do jogo. Este gerenciador deve garantir que o estado atual seja um estado válido, a partir de um conjunto de regras estabelecidas para cada jogo. Cada entidade existente no mundo é armazenada em estrutura de dados juntamente com as propriedades associadas a cada uma delas, como por exemplo, velocidade, posicionamento, etc.

A este módulo, estão associados o módulo de gerenciamento de modelagem física que garante que o estado atual do mundo não infringe nenhuma propriedade da física estabelecida, o gerenciador multiusuário que garante que o mundo na máquina local está sincronizado com os demais mundos dos outros usuários remotos e o editor de cenário que

é uma ferramenta que possibilita que vários cenários sejam projetados mais facilmente. Além disso, este módulo gerencia todos os objetos que estão presentes no jogo, sejam eles móveis ou fixos.

Principais Problemas Encontrados

1. Deve existir uma única instância deste módulo e ela deve estar acessível a partir de um ponto conhecido no motor ;
2. Cada objeto pode mudar a aparência de acordo com o estado do mundo ou do próprio objeto em um determinado momento do jogo. Por isso, é interessante poder separar o estado do objeto de sua animação [7];
3. O estado de um objeto no mundo depende do estado atual do jogo e quem define a mudança de estados dos objetos são as regras do jogo [7];
4. Geralmente os jogos precisam tratar os estados de vários objetos do mundo simultaneamente [7];
5. Deve existir uma busca otimizada de objetos do repositório [7].

Soluções

6. O problema (1) pode ser resolvido utilizando-se o padrão de projeto ***Singleton*** [52];
7. O problema (2) pode ser resolvido utilizando-se o padrão de projeto ***Appearance Map*** [58]. Com ele é possível verificar constantes de controle e baseado nelas alterar a aparência do objeto;
8. O problema (3) pode ser resolvido com o padrão de projeto ***Model***[59]. Com ele é possível tratar o estado do objeto em relação ao estado do mundo.
9. O problema (4) pode ser resolvido utilizando-se o padrão de projeto ***Model Database*** [60] que agrega objetos em um banco de dados (repositório);
10. O padrão de projeto ***Iterator*** [52] pode ser utilizado para resolver o problema (5).

Diagrama

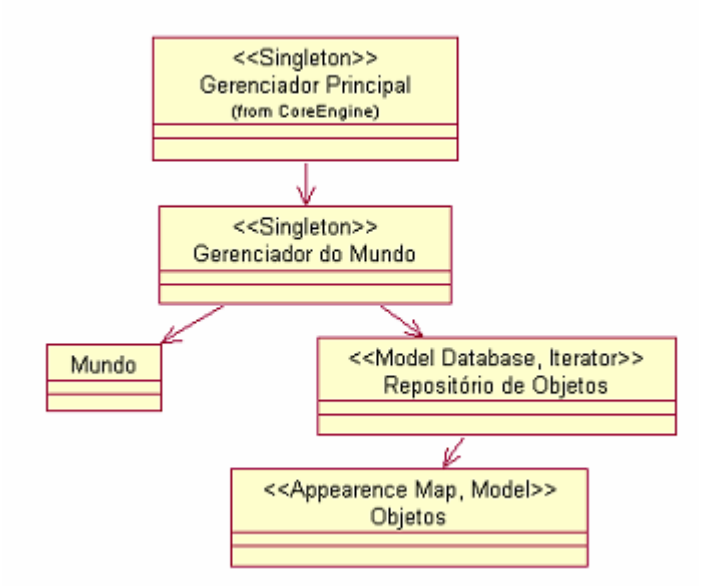

Figura 5-10 Diagrama de classes do Gerenciador do Mundo.

5.2.8 Gerenciador de Modelagem Física

Este módulo é responsável por gerenciar as regras físicas que se aplicam ao mundo em questão. Por se tratar de um código complexo e que precisa ser bastante otimizado, resolveu-se desenvolvê-lo como um módulo à parte do gerenciador do mundo.

Os modelos físicos são baseados na física Newtoniana [61] que trabalham bem para objetos que possuem limites razoáveis de massa e tamanho .

5.2.9 Gerenciador de IA

O módulo de gerenciamento de inteligência artificial (IA) é responsável pelo controle de entidades autônomas (NPC – *non player character*). Ele contribui muito para um jogo de sucesso pois possibilita uma maior interação do usuário com os demais componentes do jogo.

Diversas tecnologias podem ser utilizadas neste módulo. Dentre elas, podemos destacar: redes neurais, motores de inferências, lógica *fuzzy* e máquina de estados finitos (FSM).

Vale salientar que o estudo realizado neste módulo não está completo e que os problemas apresentados aqui são um esboço preliminar dos principais problemas que envolvem este módulo.

Principais Problemas Encontrados

1. Deve existir uma única instância deste módulo e ela deve estar acessível a partir de um ponto conhecido no motor [7];
2. Geralmente, os NPCs são máquinas que mudam de estado de acordo com os eventos do jogo e o estado do mundo[7];

Soluções

1. O problema (1) pode ser resolvido utilizando-se o padrão de projeto ***Singleton*** [52];
2. O problema (2) pode ser resolvido com o padrão de projeto ***Controller State Machine*** [62] através de uma subclasse controladora que contém uma lista de todas as variáveis de estado;

Diagrama

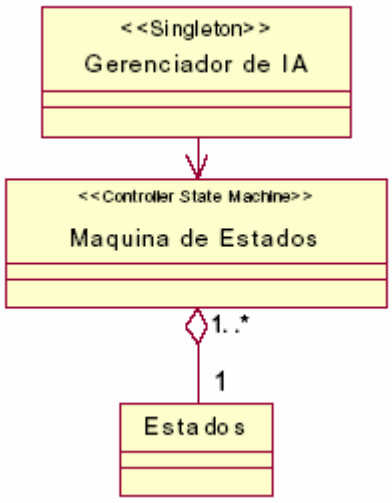

Figura 5-11 Diagrama de classes do gerenciador de IA.

5.2.10 Editor de Cenários

O editor de cenários é uma ferramenta bastante útil à criação de jogos [63]. Ele tem o papel de permitir, de forma simplificada e visual, o desenvolvimento de cenários para os jogos. Como cada jogo geralmente é composto de vários níveis onde cada nível possui um ou mais cenários, o editor de cenários torna o processo de concepção do jogo mais fácil, principalmente porque quem cria os cenários são projetistas gráficos que usualmente não possuem habilidades de programação.

O acoplamento entre o editor de cenários e o motor é um fator muito importante para que o cenário desenvolvido no editor possa ser importado no motor facilmente. Uma forma para se conseguir isto é desenvolver o editor na mesma linguagem de programação no motor e usar as mesmas estruturas de dados utilizadas na representação dos objetos do mundo. Com isso, o editor precisa apenas serializar os objetos que compõe o cenário para um determinado arquivo que será utilizado pelo motor para inicializar os objetos. A grande vantagem de se utilizar este mecanismo é a grande integração entre o motor e o editor, o que possibilita que o cenário possa ser simulado no próprio editor [6].

Outro mecanismo que pode ser utilizado é o de desenvolver o editor de cenários em uma linguagem de programação diferente da que foi utilizado no motor e utilizar uma linguagem descritiva que é entendido pelo editor e pelo motor para gerar o cenário. O arquivo gerado na linguagem descritiva será utilizado então pelo motor, para compor o cenário em tempo de execução do jogo. A principal vantagem desta abordagem é o desacoplamento do editor de cenários, o que possibilita que ele seja um projeto à parte do motor [6].

5.3 CONCLUSÕES

Neste capítulo foram apresentadas as principais motivações que levaram ao desenvolvimento do Forge *16V*. Também foi mostrado que um dos principais objetivos deste projeto é o desenvolvimento de um *framework* para o desenvolvimento de jogos que estivesse em estado funcional o mais rápido possível. Para conseguir isto, foram realizadas as seguintes escolhas de projeto:

- Desenvolvimento incremental;

- Desenvolver um motor para os jogos com cenários isométricos e bidimensionais;
- Desenvolver um motor utilizando transferências de *bits* (ver seção 2.4.3.1) ao invés de técnicas 3D;
- Usar somente a biblioteca multimídia DirectX, ou seja, desenvolver um motor para jogos que rodem somente na plataforma Windows;
- Não trabalhar com código *multithreaded* nesta primeira fase do projeto;

Também foram mostrados os módulos que compõem o *Forge 16V* e o papel de cada um deles no sistema. Além disso, foi apresentado um estudo que foi iniciado no projeto do *Forge V8* [7] e continuado por este trabalho sobre os principais problemas encontrados em um motor de jogos. A partir deste estudo foi feito um catálogo de padrões de projeto que poderá servir de guia para os futuros trabalhos na área.

Capítulo 6

Forge 16V - Implementação

Neste capítulo serão mostrados os detalhes de implementação do Forge 16V, bem como a validação do framework.

Neste capítulo serão mostrados os detalhes de implementação do *Forge 16V*, bem como as escolhas realizadas nesta fase. Além disso, será mostrado o estudo de casos realizado através da implementação de três protótipos de jogos.

Na atual fase de desenvolvimento, nem todos os padrões de projetos catalogados no capítulo anterior chegaram a ser utilizados.

6.1 AMBIENTE DE PROGRAMAÇÃO UTILIZADO

As linguagens utilizadas no desenvolvimento do *Forge 16V* foram o C++ [40] e, em alguns pontos que precisavam de um código bastante otimizado, a linguagem de montagem (*Assembly language*) da arquitetura Intel x86. A escolha pelo C++ como linguagem principal se deu devido à grande demanda de uma aplicação com um alto desempenho e, ao mesmo tempo, pela necessidade de se utilizar uma linguagem orientada a objetos que abstraísse diversos problemas dos programadores. Além do mais, o uso do C++ possibilitou a integração com linguagens de mais baixo nível, como *Assembly language*, por exemplo. Como a grande maioria dos jogos comerciais para PCs lançados são desenvolvidos em C++ [21], pode-se dizer que ela é a linguagem padrão da indústria de jogos hoje, o que justifica plenamente a escolha realizada.

O ambiente de desenvolvimento utilizado no projeto foi o Visual C++ 6.0 da Microsoft. O resultado da codificação do *Forge 16V* é uma biblioteca estática chamada `Forge16V.lib`. Essa biblioteca deve ser importada juntamente com os arquivos de cabeçalho do *Forge 16V*, pela aplicação que usa elementos do motor para poder gerar o executável (ver a Figura 6-1).

Como foi mencionado na seção 5.1, a biblioteca multimídia utilizada no desenvolvimento do *Forge 16V* foi o DirectX da Microsoft [19]. Embora o DirectX restrinja a aplicabilidade dos jogos à plataforma Windows, ele foi escolhido por ser um padrão da indústria de jogos para PC [21], e por implementar diversas funcionalidades essenciais ao desenvolvimento de jogos em uma única biblioteca, como por exemplo, sonorização, controle de dispositivos de entrada e renderização por *streaming* de vídeo, coisa que não ocorre com o OpenGL [39].

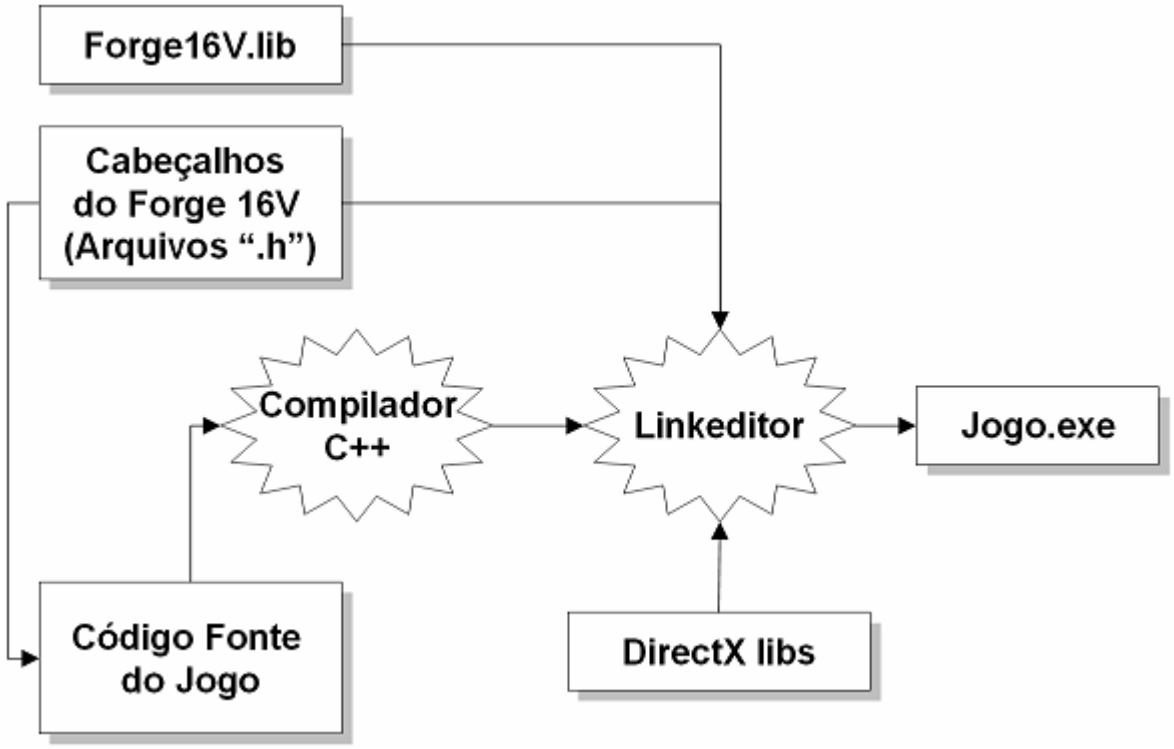

Figura 6-1 Recursos necessários para compilar um programa usando o *Forge 16V*.

6.2 MÓDULOS IMPLEMENTADOS

Por falta de tempo, não seria possível desenvolver por completo um motor como parte do trabalho de mestrado. Seguindo então a política mencionada na seção 5.1 de possuir um motor funcional o mais rápido possível, resolveu-se então que nesta primeira fase de implementação os seguintes módulos seriam desenvolvidos:

- Gerenciador Principal;
- Gerenciador Gráfico;
- Gerenciador de Entrada;
- Gerenciador de Som;
- Gerenciador de Log;
- Gerenciador de Mundo.

Atualmente estes módulos estão em estado funcional e possibilitaram assim o desenvolvimento de três protótipos de jogos por equipes que nada tinham a ver com o

projeto do *Forge 16V*. Estes jogos foram utilizados como estudo de caso para o motor e os resultados obtidos serão mostrados na seção 6.3 deste capítulo. A Figura 6-2 apresenta os módulos que estão em estado funcional, embora o módulo do gerenciador do mundo foi implementado apenas parcialmente, comparado ao que estava previsto, e por isso possui um destaque diferenciado.

O restante desta seção apresenta os detalhes de implementação de cada módulo já produzido e mostra como o programador que está desenvolvendo um jogo deve utilizar o *Forge 16V*.

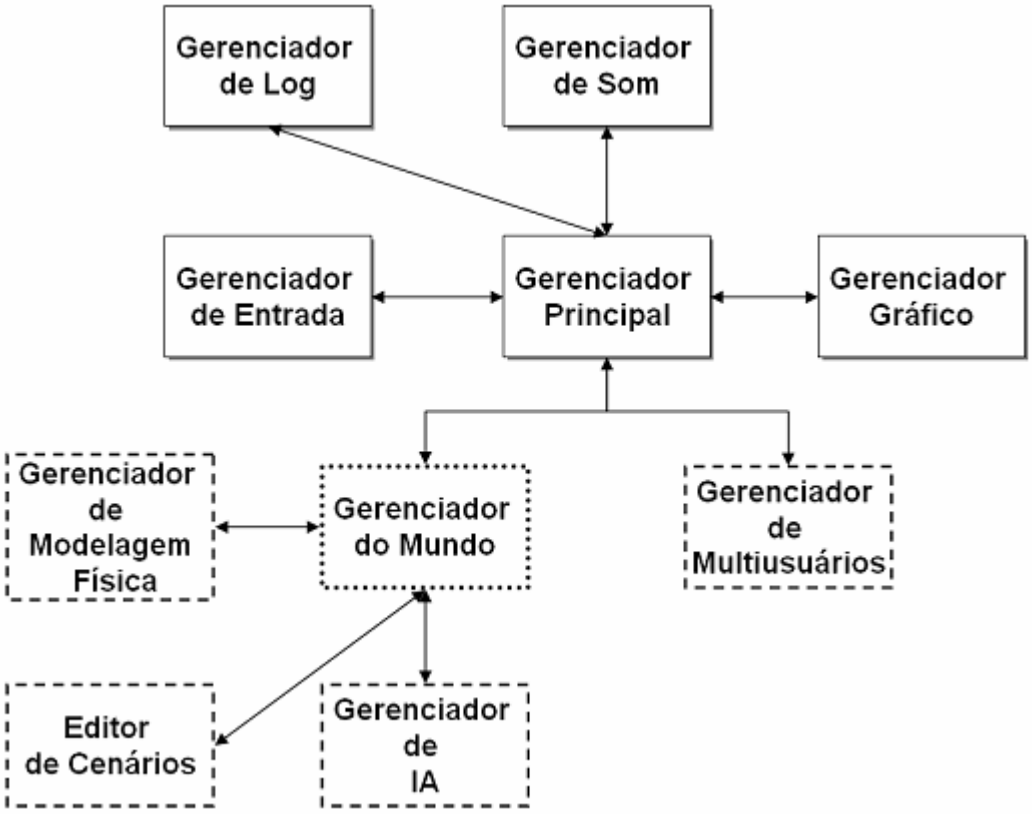

Figura 6-2 Módulos implementados (linha contínua), implementados parcialmente (bolinhas) e não implementados (tracejado).

6.2.1 Gerenciador Principal

Este módulo pode ser considerado o maestro do *Forge 16V*. É ele quem controla a instanciação dos demais módulos, a temporização do laço principal (ver a Figura 5-1) e

dispara a execução de código dos outros módulos. Além disso, ele provê a principal interface do *Forge 16V* com o código desenvolvido pelo programador do jogo.

Na fase atual, este módulo é composto de três classes principais (ver a Figura 6-3):

- CF16VGameManager;
- CF16VGameState;
- CF16VGameStateManager.

A classe `CF16VGameManager` é a classe principal deste módulo e obedece ao padrão de projeto *Singleton* [52]. Sendo assim, existe única instância desta classe na aplicação e ela pode ser acessada através do método de classe denominado `Instanciate`, que por sua vez, se encontra acessível em qualquer ponto do código da aplicação. Como em qualquer jogo os módulos gráficos e de entrada sempre têm que estar presentes, o *Forge 16V* inicializa estes módulos automaticamente na instanciação da classe `CF16VGameManager`. Os demais módulos podem ser inicializados através dos métodos `Init` da classe `CF16VGameManager` (e.g. `InitSoundEngine`, `InitNetEngine`, etc). A classe `CF16VGameManager` também provê acesso aos demais módulos através dos métodos `Get` (e.g. `GetSoundEngine`, `GetNetEngine`), possibilitando fácil acesso aos demais módulos do sistema.

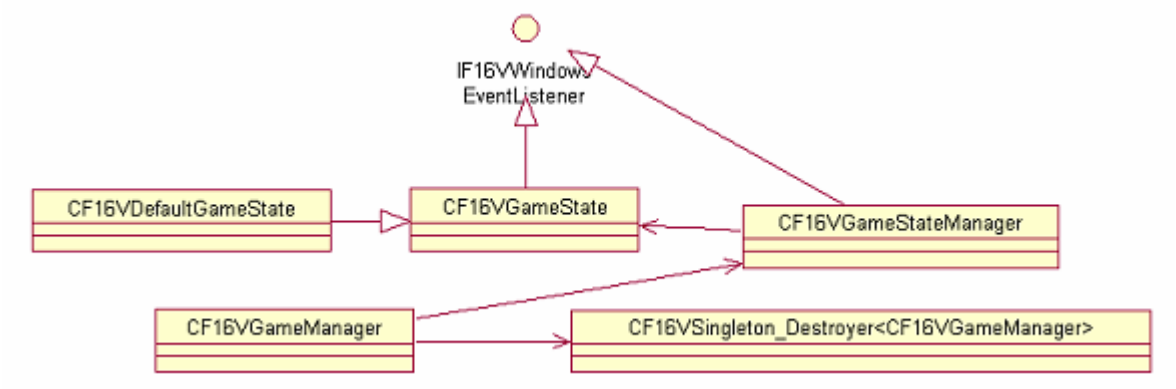

Figura 6-3 Diagrama de classes do módulo principal

Outra função importante da `CF16VGameManager` é o gerenciamento do laço principal e conseqüentemente da temporização do jogo. Através do método `SetMaxFrameRate`, é possível determinar uma taxa de quadros por segundo máxima

com a qual o jogo irá rodar, garantindo assim uma melhor jogabilidade. Este mecanismo é extremamente necessário por causa da grande diferença de poder computacional existente nas distintas máquinas em que o jogo irá rodar.

Este módulo também é responsável pela integração do *Forge 16V* com o código desenvolvido pelo programador do jogo. Observando os jogos já existentes, notou-se que em um jogo existem vários estados, e que a maioria dos módulos deve responder de forma diferenciada, dependendo do estado atual. Por exemplo, um jogo típico possui um filme de introdução que leva a um menu principal (ver a Figura 6-4). Do menu principal, o jogador pode ir a vários menus secundários ou ao jogo propriamente dito e cada elemento deste pode ser modelado como um estado onde o jogo deve agir de forma diferenciada aos eventos dos dispositivos de entrada e produzir elementos gráficos diferentes.

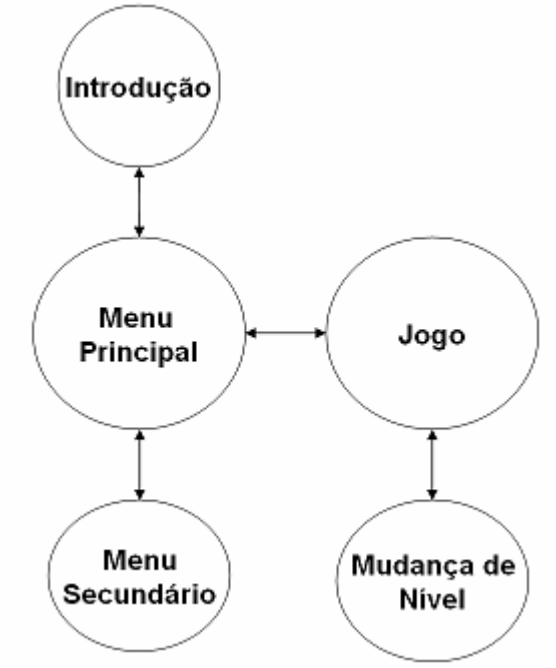

Figura 6-4 Exemplo de Estados que pode ser modelado no *Forge 16V*.

A solução padrão para este tipo de problema é a de implementar uma estrutura de escolha dentro de cada módulo da aplicação, como demonstra a Figura 6-5. Esta era a abordagem utilizada no *Forge V8*.

```
Do_graphics() {
    switch (state) {
      case INTRO:
        play_film();
        break;
      case GAME:
        build_scene();
      break;
    }
}

Do_Input(Event e) {
    switch (state) {
      case INTRO:
        if(e.kind == KEY_DOWN) skip_intro();
        break;
      case GAME:
        if(e.kind == KEY_DOWN && e.key == ESC) go_mainmenu();
        break;
    }

}
```

Figura 6-5 Pseudo código mostrando a solução padrão usada para os lidar com diversos estados de um jogo.

Visando melhorar o agrupamento de funcionalidades e facilitar a integração do motor com o código do programador, no *Forge 16V* foram criadas as classes `CF16VGameStateManager` e `CF16VGameState`. O papel da classe `CF16VGameStateManager` é o de gerenciar um conjunto de estados (modelados pela classe `CF16VGameState`), direcionando os eventos advindos de outros módulos, somente para o único estado que se encontra ativo em um determinado instante do jogo (ver a Figura 6-6), e possibilitando que o estado ativo interaja com o módulo gráfico. A classe `CF16VGameManager` contém uma instância da classe `CF16VGameStateManager` e ela não pode ser acessada diretamente pelo programador que utiliza o motor. O programador deve utilizar os métodos `AddGameState`, `RemoveGameState` e `SelectGameState` da classe `CF16VGameManager` para respectivamente adicionar, remover ou selecionar um estado como ativo.

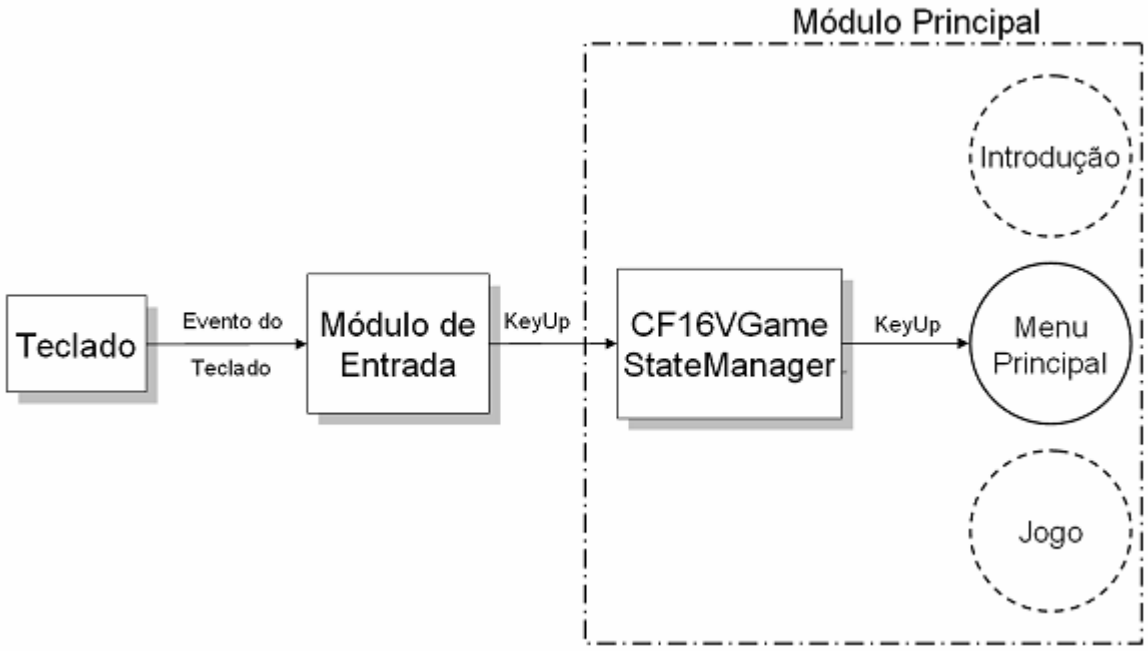

Figura 6-6 Fluxo de eventos gerenciado pela classe CF16VGameManager - somente o estado ativo recebe o evento.

A classe `CF16VGameState` corresponde a um estado no jogo. O programador que pretende utilizar o *Forge 16V*, deve usar o mecanismo de herança e implementar os estados do jogo que está sendo desenvolvido. Seguindo o exemplo da Figura 6-4, cinco classes que herdam de `CF16VGameState` devem ser implementadas e registradas através do método `AddGameState` da classe `CF16VGameManager`. A Tabela 6-1 apresenta os principais métodos da classe `CF16VGameState` que devem ser implementados por uma classe que a estenda.

Vistas as principais classes do módulo principal, os principais passos que devem ser seguidos para utilizar o *Forge 16V* são:

1. Criar uma instância da classe `CF16VGameManager`;
2. Criar uma nova classe que herde de `CF16VGameState`;
3. Registrar a nova classe com o `CF16VGameManager` através do método AddGameState;
4. Selecionar um modo de vídeo.

Método	Descrição
IncludeGameState	Este método é chamado toda vez que o estado se registra com a classe `CF16VGameManager`. Ele deve ser usado para inicializar os membros da classe que dependem do motor (e.g. elementos gráficos).
ActivatingGameState	Este método é chamado toda vez que o estado está sendo ativado.
DeactivatingGameState	Este método é chamado toda vez que o estado está sendo desativado.
RemoveGameState	Este método é chamado toda vez que o estado é removido da classe `CF16VGameManager`. Ele deve ser usado para destruir os membros da classe que dependem do motor.
ProcessFrame	Este método é chamado uma vez a cada iteração do *loop* principal do jogo quando este estado for o ativo, e deve ser utilizado para processar e apresentar elementos gráficos na tela.
WindowProc	Método usado para processar os eventos vindos do Windows quando este for o estado ativo.
KeyDown	Este método é chamado toda vez que uma tecla for pressionada e este for o estado ativo.
KeyUp	Este método é chamado toda vez que uma tecla for liberada e este for o estado ativo.
MouseMoved	Este método é chamado toda vez que o mouse se mover e este for o estado ativo.
MouseButtonDown	Este método é chamado quando um botão do mouse for pressionado e este for o estado ativo.
MouseButtonUp	Este método é chamado quando um botão do mouse for liberado e este for o estado ativo.
JoystickMoved	Este método é chamado toda vez que o “manche” principal do joystick se mover e este for o estado ativo.

JoystickRotated	Este método é chamado toda vez que houver uma rotação no joystick e este for o estado ativo.
JoystickButtonDown	Este método é chamado toda vez que um botão do joystick for pressionado e este for o estado ativo.
JoystickButtonUp	Este método é chamado toda vez que um botão do joystick for liberado e este for o estado ativo.

Tabela 6-1 Principais métodos que devem ser implementados na herança da classe CF16VGameState

A Figura 6-7 apresenta um código bem simples que, embora contenha elementos que não foram vistos ainda, demonstra os passos mencionados anteriormente. Primeiramente uma classe denominada `CF16Sample3GameState` que herda de `CF16VGameState` é declarada e os métodos que interessam (ver Tabela 6-1) são implementados. Na `WinMain`, que é o ponto de entrada da aplicação, ocorre a instanciação das classes `CF16VGameManager` e `CF16VSample3GameState`, o registro do novo estado com o gerenciador (único estado, logo passa a ser o estado ativo) e por fim ocorre a seleção do modo de vídeo que é onde o laço principal começa (ver a Figura 5-1). Uma vez iniciado o laço principal, os eventos dos dispositivos de entrada são enviados para o estado ativo. No exemplo, a classe `CF16VSample3GameState` só trata os eventos do teclado através do método `KeyDown(`tecla pressionada), que no caso, verifica se a tecla `ESC` foi pressionada e finaliza a aplicação. Outro ponto importante é que a cada iteração do laço principal, o método `ProcessFrame` do estado ativo é chamado, e no exemplo, esse método pinta uma figura na tela.

```
#include "Global\Global.h"
#include "GraphicEngine\VideoEngine.h"
#include "GameState.h"
#include "GraphicEngine\Layer.h"
#include "InputEngine\CommonStructs.h"
#include "InputEngine\Keyboard.h"

class CF16VSample3GameState : public CF16VGameState{
```

```
protected:
      CF16VLayer* m_layer;
      POINT             m_point;
      BOOL              m_loaded;
public:
      CF16VSample3GameState();
      virtual ~CF16VSample3GameState();
      virtual void      IncludeGameState();
      virtual void      RemoveGameState();
      virtual void KeyDown( int iKey );
      virtual HRESULT ProcessFrame(void);

};
CF16VSample3GameState::CF16VSample3GameState() {
      m_layer = new CF16VLayer();
      ReceiveKeyboardEvent(TRUE);
}
CF16VSample3GameState::~CF16VSample3GameState() {
      SAFE_DELETE(m_layer);
}
void  CF16VSample3GameState::IncludeGameState() {
      m_layer->Create(GetVideoEngine(),"Winter.jpg") ;
}
void  CF16VSample3GameState::RemoveGameState() {

}
void  CF16VSample3GameState::KeyDown( int iKey ) {
      if( iKey == F16VK_ESCAPE) {
            GetGameManager()->Exit(0);

      }
}
HRESULT CF16VSample3GameState::ProcessFrame(void) {

      CF16VSurface* back = GetVideoEngine()->GetBackBuffer();
      m_layer->Draw(back);
      return 0;
}
int WINAPI WinMain(HINSTANCE hinstance,
                           HINSTANCE hprevinstance,
                           LPSTR    lpcmdline,
                           int ncmdshow) {
      CF16VGameManager *gm = CF16VGameManager::Instanciate();
      CF16VSample3GameState state;
      gm->AddGameState(&state);
      gm->GetVideoEngine()->CreateFullScreen(800,600,16);
      return (0);
}
```

Figura 6-7 Exemplo de código de como usar o *Forge 16V*

6.2.2 Gerenciador Gráfico

Este módulo é um dos principais e também um dos mais complexos do *Forge 16V*. Como todo jogo deve utilizar um dispositivo gráfico, este módulo é instanciado automaticamente pela classe `CF16VGameManager`. Todo o gerenciamento do módulo gráfico é realizado através da classe `CF16VVideoEngine` que obedece ao padrão de projeto *Singleton* e pode ser acessada através do método `CF16VGameManeger::GetVideoEngine` (ver a Figura 6-7). É através da classe `CF16VVideoEngine` que é possível consultar os modos de vídeo disponíveis (com o método `GetVideoMode`) e escolher se a aplicação vai rodar no modo janela (com o método `CreateWindowed`), ou no modo tela cheia (com o método `CreateFullScreen`).

Tendo em vista todos os problemas ocorridos no *Forge V8* (ver Capítulo 3), resolveu-se utilizar, inicialmente neste projeto, a técnica de transferência de bits entre áreas de memória para realizar animações (ver seção 2.4.3.1), o que possibilitou que o módulo gráfico estivesse em estado funcional em pouco mais de três meses de desenvolvimento. Esta escolha limitou a princípio os tipos de cenários que podem ser produzidos utilizando-se o motor para os bidimensionais e isométricos. Contudo, como a arquitetura do motor foi idealizada criando-se uma certa independência entre os módulos, no futuro, uma nova implementação pode ser realizada utilizando-se a renderização tridimensional como base.

A seguir serão mostrados os melhoramentos realizados no motor para facilitar a manipulação de imagens.

6.2.2.1 Surface

O módulo do *DirectX* que possibilita a transferência de mapas de bits (imagens) entre áreas de memória é o *DirectDraw*. Na literatura, uma área de memória destinada a armazenar imagens ficou conhecida como *Surfaces*. O *DirectDraw* permite que *Surfaces* sejam criadas, tanto na memória de vídeo,. quanto na memória do sistema e além disso, permite o acesso e manipulação das imagens que estão armazenadas na memória.

Embora o *DirectDraw* disponibilize os mecanismos de alocação de uma área de memória e de transferência de bits com um certo grau de abstração do *hardware* utilizado (por se tratar de uma biblioteca de propósito geral), ela exige que o programador se preocupe com alguns detalhes de "baixo nível". Por exemplo, o programador fica

responsável por escolher em qual memória será reservada uma área para a transferência de uma imagem: na de vídeo (mais rápida mas, disponível em menor tamanho), ou na memória do computador (mais lenta, no entanto, mais abundante). Além disso, fica a cargo do programador o tratamento de falhas, como o corrompimento de uma determinada área da memória de vídeo por alguma outra aplicação que o usuário utilizou enquanto estava rodando o jogo (a memória de vídeo é compartilhada entre todas as aplicações da máquina que rodam em um determinado instante, e não existe um controle de alocação de memória de vídeo fora da aplicação o que acaba levando uma aplicação a escrever em uma área que estava sendo utilizada por outra aplicação).

Características	***Surface DirectDraw***	**CF16VSurface (*Forge 16V*)**
Compatibilidade com Arquivos Gráficos	Arquivos BMP através de funções do DirectX	Arquivos BMP, JPEG, PCX, TGA e PNG
Gerenciamento de melhor área de memória (vídeo ou sistema)	Manual – a escolha deve ser realizada pelo programador do jogo	Automática – o *Forge 16V* mantêm uma *cache* das figuras. O programador não precisa se preocupar com qual o melhor espaço a ser alocado.
Imagens corrompidas na área de memória.	Tratamento deve ser realizado pelo programador do jogo.	A classe CF16VSurface recarrega a imagem automaticamente.

Tabela 6-2 Comparativo entre as *Surfaces* do *Direct Draw* e do *Forge 16V*.

Visando diminuir este trabalho que precisaria ser realizado pelo programador a cada novo jogo desenvolvido, o *Forge 16V* criou mais uma camada de abstração através da classe `CF16VSurface` que é a classe base de todos os elementos gráficos no motor. Com ela o programador não precisa, por exemplo, se preocupar onde deve alocar espaço para guardar uma imagem que será apresentada na tela. A Tabela 6-2 apresenta algumas das

funcionalidades que foram desenvolvidas na classe `CF16VSurface` e apresenta um comparativo com o que o *DirectDraw* provê.

Para realizar a transferência da imagem localizada em uma área de memória para outra, basta utilizar o método `Draw` da classe `CF16VSurface`. Existe uma *Surface* no motor que representa a imagem que será exibida na tela do computador e ela pode ser acessada através do método `GetBackBuffer` da classe `CF16VVideoEngine`. Sendo assim, para renderizar uma imagem na tela basta utilizar o método Draw, passando como parâmetro a *surface* retornada pelo método `GetBackBuffer` (ver a Figura 6-8).

```
CF16VSurface* back = videoEngine()->GetBackBuffer();
imagem->Draw(back, X,Y);
```

Figura 6-8 Exemplo de como exibir uma imagem contida na *surface* "imagem" na tela.

Existe um grande número de elementos gráficos do *Forge 16V* que utiliza a classe CF16VSurface como base (ver a Figura 6-9). Por razão de performance, as imagens que compõem uma determinada animação são agrupadas em um único arquivo denominado de *tileset* (ver seção 2.4.3.1), evitando assim, a leitura de vários arquivos, o que poupa tempo e recursos. A classe `CF16VTileSet` é uma *surface* com facilidades para a obtenção de um determinado quadro da animação presente no *tileset*. Já a classe `CF16VSprite` possui uma *surface* e implementa funcionalidades para a execução de uma animação, como por exemplo o controle da taxa de exibição em quadros por segundo. A classe `CF16VLayer` representa uma imagem de fundo que pode se movimentar para qualquer lado, dando a impressão que é o cenário que está se movendo. Além destas classes, um grande número de elementos gráficos de interface, que até o momento não foram implementados, está previsto para ser construído utilizando uma *surface* como elemento base (e.g. Janelas, caixas de texto, etc).

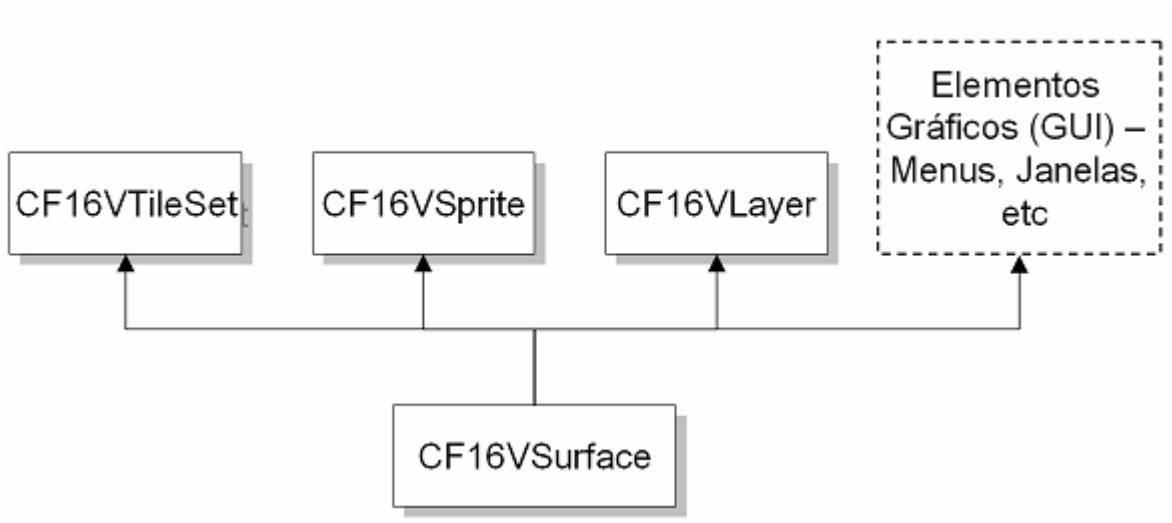

Figura 6-9 Elementos gráficos que usam a classe CF16VSurface como base. O bloco tracejado não foi implementado ainda.

6.2.2.2 Gerenciamento de Memória Gráfica

Para prover a abstração do local onde a *surface* deve ser criada, foi necessário construir uma *cache* de *surfaces*. Quando uma imagem é carregada do disco rígido, uma área da memória do computador é alocada. Somente quando uma determinada *surface* vai ser copiada para a área da memória de vídeo de onde é feita a exibição na tela (*Back buffer*), a imagem é copiada da memória do computador para a memória de vídeo, se ainda não estiver presente lá. Caso ela já esteja presente na placa de vídeo, a transferência ocorre a partir da cópia presente na placa de vídeo (muito mais rápido do que copiar a partir da memória convencional). Este processo se repete até que não seja mais possível alocar área na memória de vídeo. Quando isso ocorre, entra em cena uma política de substituição que escolhe qual *surface* deve ser removida da placa de vídeo. Atualmente existem duas políticas implementadas: a estática, que é modelada pela classe `CF16VStaticCache` e a LRU (*last referenced unit*) que é modelada pela classe `CF16VLRUCache`. A estática tem uma aplicação prática mais para testes. Ela aloca espaço na memória de vídeo até que ela seja preenchida, e quando isso ocorre, nenhuma outra *surface* é transferida para a placa de vídeo . Já a política LRU é a política padrão do *Forge 16V*. Ela remove a cópia da *surface* que foi pintada na tela há mais tempo. Com a cache utilizando a LRU, o motor implementa

a abstração de onde é o melhor lugar para guardar uma *surface*, tirando esta responsabilidade do programador.

Contudo, na fase de otimização do jogo (final do processo de desenvolvimento), pode ser necessário utilizar uma outra política que seja mais adequada às características da aplicação que está sendo desenvolvida, possibilitando assim um melhor desempenho. Isto pode ser feito implementando-se a interface `IF16VCache` e informando a classe `CF16VVideoEngine` da nova política através de método `SetCache`. A troca das políticas pode ser efetuada a qualquer momento na execução do programa.

Para finalizar esta seção, a Figura 6-10 apresenta um diagrama de classes do módulo gráfico dos elementos implementados até o momento.

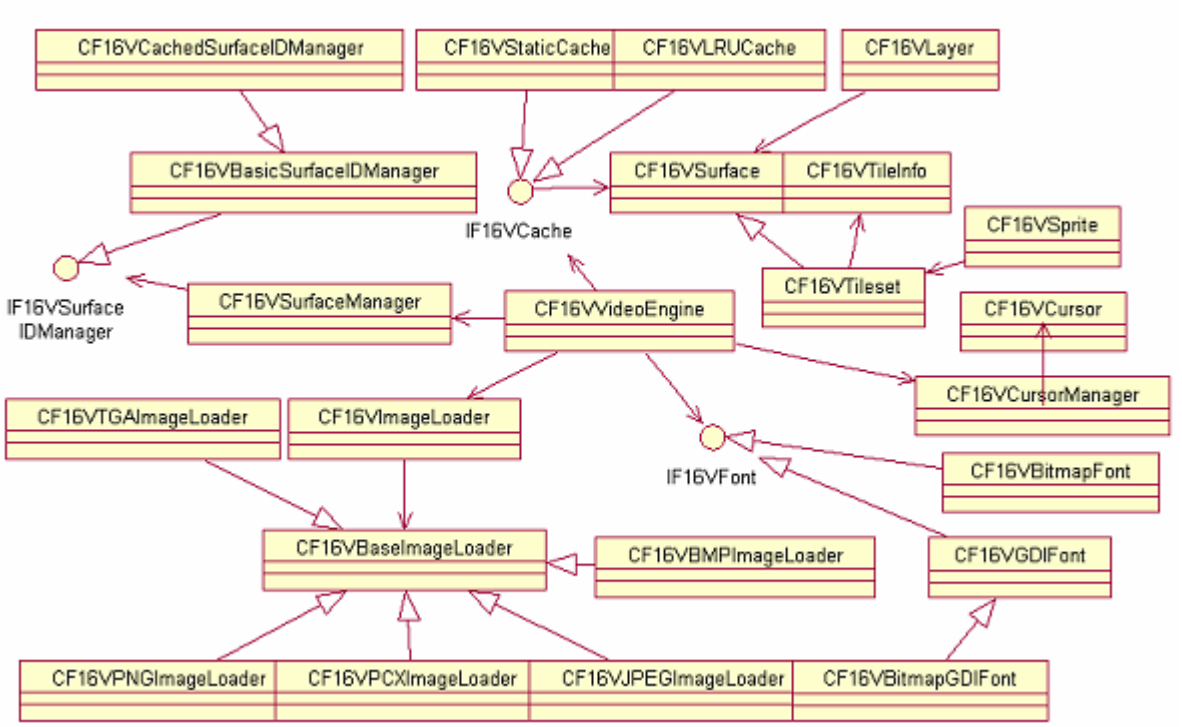

Figura 6-10 Diagrama de classes do módulo gráfico.

6.2.3 Gerenciador de Entrada

Este módulo é responsável pelo gerenciamento dos dispositivos de entrada. Na atual fase de implementação, este módulo é capaz de gerenciar um teclado, um *mouse* e até 128

joysticks. Como foi mencionado antes na seção 6.2.1, o gerenciador de entrada é inicializado automaticamente pela classe `CF16VGameManager`. As principais classes e interfaces existentes neste módulo atualmente são (ver a Figura 6-11):

- CF16VInputEngine;
- CF16VKeyboard;
- CF16VMouse;
- CF16VJoystick.

A classe `CF16VInputEngine` é a classe principal deste módulo e obedece ao padrão de projeto *Singleton* [52]. Ela pode ser acessada através do método `GetInputEngine` da classe `CF16VGameManager`. Uma das funções desta classe é a de verificar a existência do teclado, *mouse* e *joysticks* e criar as instâncias das respectivas classes `CF16VKeyboard`, `CF16VMouse` e `CF16VJoystick` para os dispositivos que existam. Estas três últimas classes controlam os dispositivos propriamente ditos, coletando os eventos originados pelos hardwares, pós-processando-os e disponibilizando-os para o restante do motor. As instâncias destas classes podem ser acessadas respectivamente através dos métodos `GetKeyboard`, `GetMouse` e `GetJoystick` da classe `CF16VInputEngine`.

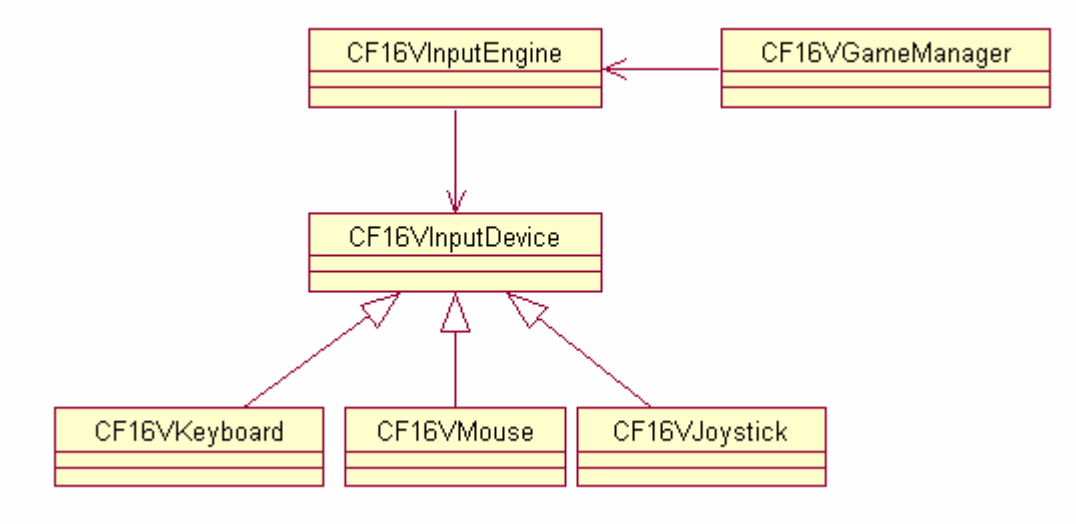

Figura 6-11 Diagrama de classes do módulo de entrada.

É possível obter informações sobre o estado dos dispositivos de duas formas:

- Consultando diretamente o estado dos dispositivos em um determinado instante (e.g. como está o botão esquerdo do *mouse*);
- Registrando-se com o dispositivo informando-o que gostaria de receber os eventos gerados por ele (padrão de projeto *Observer* [52]).

A primeira é a ideal para quando simplesmente se deseja saber o estado atual do dispositivo, sem se importar com o que ocorreu desde a última chamada, como por exemplo, quando se deseja saber em que posição o *mouse* se encontra no atual momento, sem precisar obter os deslocamentos desde a última chamada do método. Outro exemplo é quando se deseja saber se neste momento a tecla `ESC` está pressionada, sem precisar saber quantas vezes ela foi pressionada desde a última chamada do método. Esta forma de informação pode ser obtida através dos métodos `Gets` que existem nas classes `GetKeyBoard`, `GetMouse` e `GetJoystick`. O techo de código mostrado na Figura 6-12 demonstra o uso desta técnica.

```
gameManager->GetInputEngine()->GetKeyBoard()->GetKeyDown(F16VK_LMENU);
gameManager->GetInputEngine()->GetMouse()->GetMousePos(&x,&y);
gameManager->GetInputEngine()->GetJoystick(0)->GetJoystickPos(&x,&y);
```

Figura 6-12 Exemplo de código que acessa as informações do estado atual dos dispositivos de entrada.

A segunda forma de se obter informações dos dispositivos é baseado no padrão de projeto *Observer*. Por isto, basta implementar uma das seguintes interfaces:

- IF16VKeyboardEventListener;
- IF16VMouseEventListener;
- IF16VJoystickEventListenet;

Depois é só registrar a classe que implementa uma destas interfaces com o respectivo dispositivo e receber todos os eventos advindo dele. Este método, ao contrário do anterior, possibilita a análise de todos os eventos gerados pelos dispositivos, possibilitando, por exemplo, o total controle de quando e quantas vezes uma tecla ou botão foi pressionado. Este método é usado pelo módulo do gerenciador principal (ver a Figura

6-6), mais especificamente pela classe `CF16VGameStateManager`, possibilitando o redirecionamento de todos os eventos gerados pelos dispositivos de entrada para o estado ativo. A Figura 6-13 demonstra como registrar e remover classes que implementam as interfaces acima para passarem a receber os eventos dos dispositivos de entrada.

```
GetInputEngine()->GetKeyBoard()->AddKeyboardEventListener
((IF16VKeyboardEventListener*)m_stateManager);

GetInputEngine()->GetMouse()->RemoveMouseEventListener
((IF16VMouseEventListener*)m_stateManager);
```

Figura 6-13 Exemplo de código de como registrar e remover as classes que se interessam em receber os eventos dos dispositivos de entrada.

6.2.4 Gerenciador de Som

O módulo do gerenciador de som é responsável pela reprodução de músicas e sons do jogo. No *Forge 16V* é possível reproduzir vários sons que compõem os efeitos especiais (e.g. sons de pássaros) e a melodia do jogo ao mesmo tempo. Estes sons são mixados e enviados à placa de som para a reprodução. Além disso, é possível aplicar efeitos de posicionamento (som 3D), volume e reverberação. Na atual fase de implementação, somente os formatos WAV e MIDI dos arquivos de sons podem ser utilizados.

As principais classes (ver a Figura 6-14) deste módulo são:

- CF16VSoundEngine;
- CF16VSound;
- CF16VSoundEnvironment.

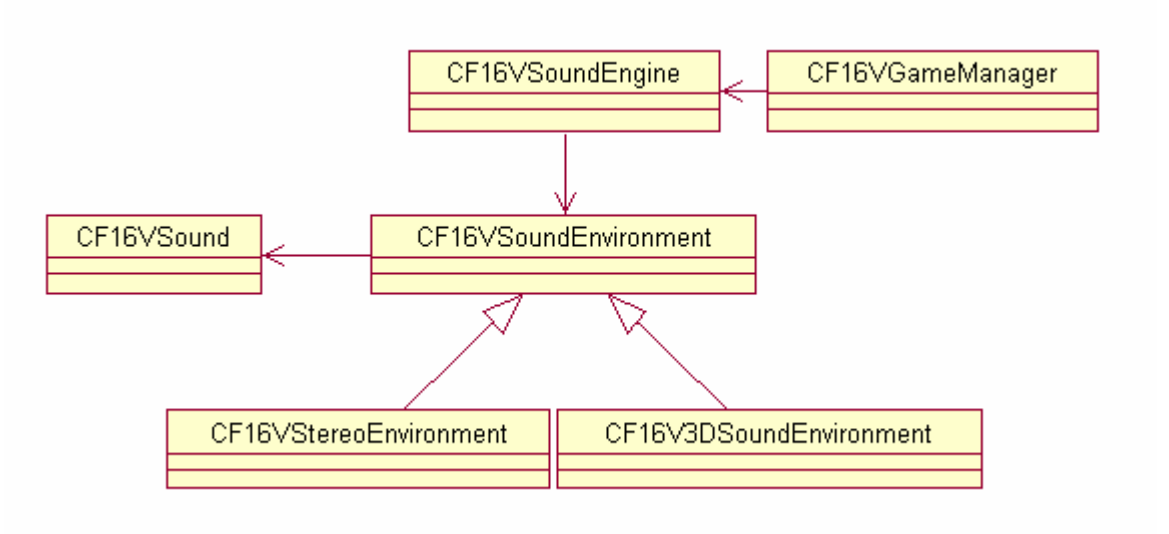

Figura 6-14 Diagrama de classes do módulo de som.

A classe CF16VSoundEngine é a classe principal do módulo de som e segue o padrão de projeto *Singleton*. Ela gerencia todos os elementos do módulo de som. Ao contrário do que ocorre com o módulo gráfico e com o gerenciador dos dispositivos de entrada, esta classe não é instanciada automaticamente pelo gerenciador principal, pois este módulo não é extremamente necessário a todos os jogos e é necessário economizar os recursos da máquina onde o jogo será rodado. Portanto, fica a cargo do programador esta inicialização, que pode ser feita usando o método `CF16VGameManager::InitSoundEngine`.

Uma vez inicializada a classe `CF16VSoundEngine`, o programador deve criar ao menos um ambiente onde os sons serão tocados. Um ambiente modela uma configuração sonora e é representado pela classe abstrata `CF16VSoundEnvironment`. Atualmente existem duas implementações de ambientes: a `CF16V3DSoundEnvironment` e a `CF16VStereoEnvironment`. A primeira possibilita que efeitos de posicionamento sejam aplicados ao som e a segunda modela um ambiente estéreo. Além disso, em cada jogo podem existir vários ambientes, e a configuração de cada ambiente pode ser mudada dinamicamente em tempo de execução, o que torna este esquema bastante flexível.

No *Forge 16V*, o som é modelado pela classe `CF16VSound`, que atualmente possibilita que um som no formato WAV seja carregado de um arquivo no disco rígido e tocado em algum ambiente (modelado pela classe `CF16VSoundEnvironment`) do jogo.

Cada som pode ser tocado em vários ambientes ao mesmo tempo e cada ambiente pode tocar vários sons (ver a Figura 6-15). Tudo isso é realizado de forma transparente pois a mixagem dos sons fica a cargo do módulo de som.

A Figura 6-16 demonstra como é simples incorporar a sonorização em um jogo utilizando o *Forge 16V*. O código em negrito representa o código referente ao módulo de som. Esse exemplo simplesmente toca um som ao pressionar a tecla "F" e este som fica tocando indefinidamente, até que a mesma tecla seja pressionada novamente. Note a inicialização da classe `CF16VSounEngine`.

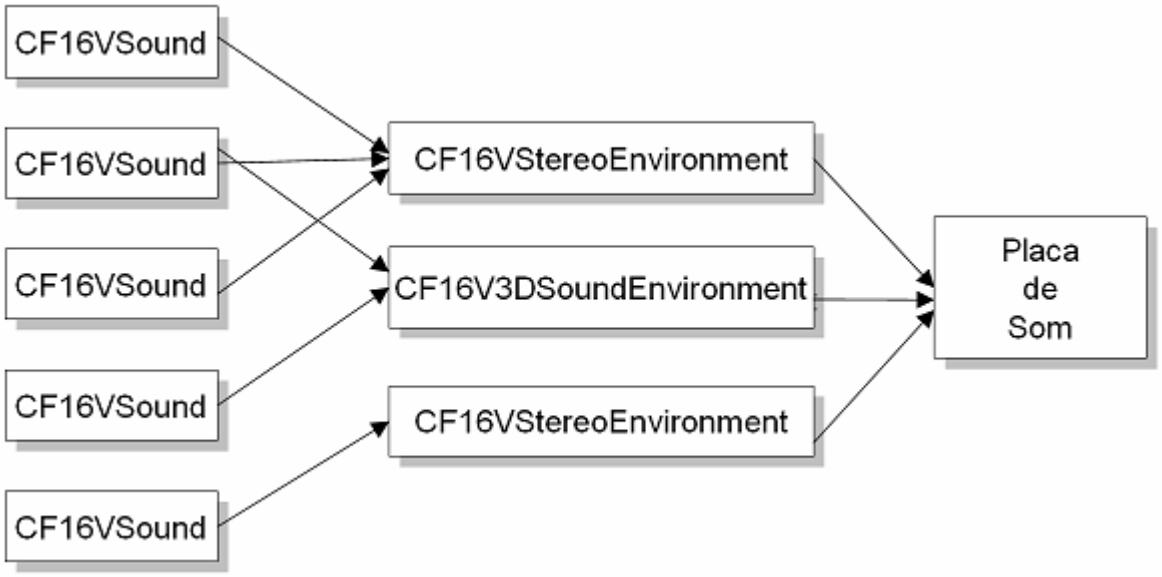

Figura 6-15 Fluxo no módulo de som.

```
#include "SoundEngine\SoundEngine.h"
#include "SoundEngine\SoundEnvironment.h"
//[resto dos includes do Forge 16V]

CF16VSurface*                 g_pbackground       = NULL;
CF16VStereoEnvironment*       g_pEnvironment      = NULL;
CF16VSound*                   g_pSound            = NULL;

class CF16VSample7GameState : public CF16VGameState{

public:
      CF16VSample7GameState();
      virtual ~CF16VSample7GameState();
      virtual void      IncludeGameState();
      virtual void      RemoveGameState();
```

```
      virtual void KeyDown( int iKey );
      virtual HRESULT ProcessFrame(void);
};
CF16VSample7GameState::CF16VSample7GameState() {
      ReceiveKeyboardEvent(TRUE);
}
CF16VSample7GameState::~CF16VSample7GameState() {
}
void  CF16VSample7GameState::IncludeGameState() {
      CF16VSoundEngine* soundEngine = GetGameManager()->
InitSoundEngine(); // inicializa o módulo de som
      if ( soundEngine) g_pEnvironment = soundEngine->
CreateStereoEnvironment(); // cria um ambiente
      if (g_pEnvironment) g_pSound = new CF16VSound(soundEngine);
      g_pSound->CreateSegmentFromFile("water.wav",false);
      g_pSound->SetRepeats(F16V_REPEAT_INFINITE);
      g_pbackground = new CF16VSurface();
      g_pbackground->Create(GetVideoEngine(),"bkgrnd.jpg");
}

void  CF16VSample7GameState::RemoveGameState() {
      SAFE_DELETE(g_pbackground);
      SAFE_DELETE(g_pSound);
}

void  CF16VSample7GameState::KeyDown( int iKey ) {
      if( iKey == F16VK_ESCAPE) {
            GetGameManager()->Exit(0);

      }
      if (iKey == F16VK_F) {
            if(g_pSound && g_pEnvironment) {
                  if (!g_pSound->IsPlaying()) g_pSound->
Play(g_pEnvironment);
                  else g_pSound->Stop();
            }
      }
}

HRESULT CF16VSample7GameState::ProcessFrame(void) {
      // código que imprime uma figura na tela
      // ver Figura 6-7
}
int WINAPI WinMain(HINSTANCE hinstance,
                           HINSTANCE hprevinstance,
                           LPSTR    lpcmdline,
                           int ncmdshow) {
      CF16VGameManager *gm = CF16VGameManager::Instanciate();
      CF16VSample7GameState state;
      gm->AddGameState(&state);
      gm->GetVideoEngine()->CreateWindowed(800,600);
      return 0;
}
```

Figura 6-16 Exemplo de código que inclui som no jogo.

6.2.5 Gerenciador de Log

No *Forge 16V*, foi criado o módulo de *log* com a tarefa de auxiliar a depuração de código do próprio *framework*, e do código dos jogos que estão sendo desenvolvidos utilizando o motor. Este módulo é inicializado automaticamente pela classe `CF16VGameManager` e as suas principais classes são (ver a Figura 6-17):

- CF16VSystemMessagerControler;
- CF16VMessageConsumer;

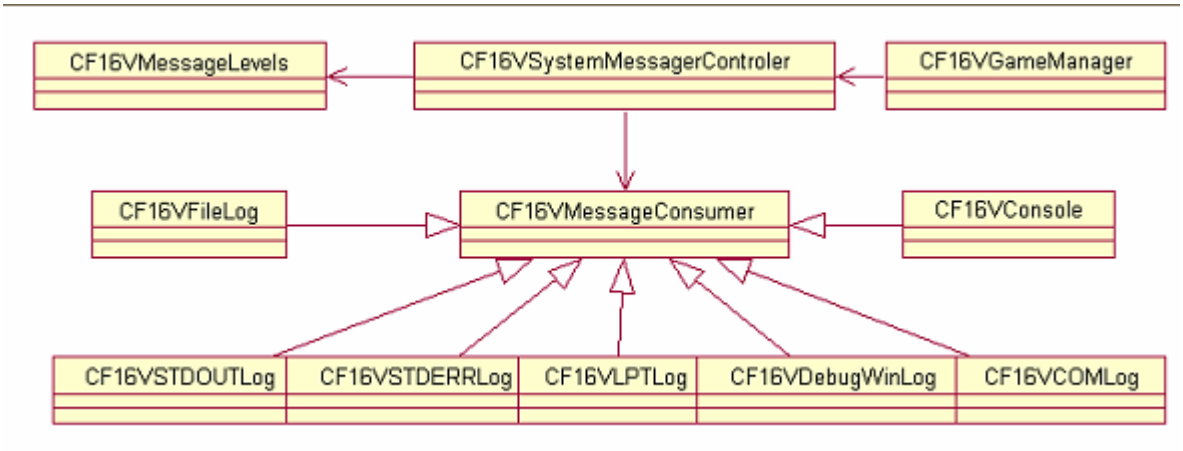

Figura 6-17 Diagrama de classes do módulo de log.

A principal função da classe `CF16VSystemMessagerControler` é receber todas as mensagens e roteá-las para um conjunto de consumidores que são modelados pela classe `CF16VMessageConsumer`. Assim, se uma classe deseja receber algum tipo de mensagem, basta que ele herde da classe `CF16VMessageConsumer` e se registre através do método `AddMessageConsumer` da classe `CF16VSystemMessagerControler`, seguindo o padrão de projeto *Observer* [52].

Tentando criar um maior grau de classificação das mensagens, foram criados quatro tipos de mensagens:

- Information;
- Warning;
- Error;
- Fatal.

Além disso, para cada tipo, foi criado outra divisão chamada de nível. Na atual implementação, existem dez níveis que vão do nível 1 ao 10, onde um menor número significa um maior grau nos problemas. Os níveis de 1 a 5 são reservados aos problemas do motor em si, ficando os níveis 6 a 10 para utilização do programador do jogo. Ao registrar um consumidor com a classe `CF16VSystemMessagerControler`, o programador deve informar para cada tipo de mensagem, qual o nível que deseja receber. Sendo assim, só serão repassadas as mensagens com um nível igual ou superior ao informado. Por exemplo, se ao registrar um determinado consumidor, o programador informou para o tipo de mensagem *information* o nível 6. Isto significa que quando uma mensagem do tipo *information* chegar, ela será repassada para o consumidor caso seja do nível 6 ou superior.

Na atual fase de implementação, já existem alguns consumidores implementados que possibilitam a saída de mensagens em um grande número de dispositivos. A Tabela 2-1 apresenta as classes já implementadas com as respectivas saídas.

Classe	***Saída***
CF16VCOMLog	Envia as mensagens para a porta serial do PC (COM)
CF16VLPTLog	Envia as mensagens para a porta paralela do PC (LPT)
CF16VSTDOUTLog	Envia as mensagens para um console DOS no modo normal (STDOUT)
CF16VSTDERRLog	Envia as mensagens para um console DOS no modo de erro (STDERR)
CF16VFileLog	Envia as mensagens para um arquivo.

Tabela 6-3 Consumidores de mensagens em estado funcional.

Além dos consumidores já implementados que foram apresentados na Tabela 6-3, ainda existe a classe `CF16VConsole` que implementa as funcionalidades de um console gráfico (ver seção 5.2.4 para maiores informações sobre console gráfico). Esta classe ainda

não se encontra em estado funcional e deve ser implementada na próxima fase de desenvolvimento.

Para facilitar o uso deste módulo, existe um atalho na classe principal do *Forge 16V*, a `CF16VGameManager`, para enviá-las de acordo com o tipo desejado. A Figura 6-18 apresenta um código que demonstra como enviar mensagens a partir do gerenciador principal.

```
gameManager->InfoMessage(“Mensagem de Debug”, MESSAGE_LEVEL_6);
gameManager->WarningMessage(“Mensagem de Debug”, MESSAGE_LEVEL_6);
gameManager->ErrorMessage(“Mensagem de Debug”, MESSAGE_LEVEL_6);
gameManager->FatalMessage(“Mensagem de Debug”, MESSAGE_LEVEL_6);
```

Figura 6-18 Exemplo de código de como enviar uma mensagem de acordo com o tipo.

6.2.6 Gerenciador de Mundo

O gerenciador do mundo ainda se encontra longe de possuir todas as funcionalidades previstas (e.g. gerenciamento de propriedades dos objetos – ver seção 5.2.7). Por isso o estado indicado na Figura 6-2 é o de parcialmente implementado. Na atual fase de implementação, somente o suporte a cenários baseados em *tiles*, tanto retangulares quanto isométricos, foi implementado. Ainda não existe nenhum controle dos objetos presentes no mundo, embora esta seja uma funcionalidade prevista. A Figura 6-19 apresenta as principais classes implementadas até o momento.

Para conseguir implementar este módulo, foi necessário utilizar os conceitos mostrados no capítulo 4 sobre jogos isométricos. Um dos principais problemas nesse tipo de cenário é realizar o mapeamento entre o espaço do mundo (baseada em tiles) e a tela do computador e vice-versa. Este problema pode ser dividido em três subproblemas:

1. Dado um *tile*, em que ponto na tela ele se localiza;
2. Dado um tile e uma determinada direção, qual o *tile*mais próximo na direção dada;
3. Dado um ponto na tela, a que *tile* ele pertence.

A solução destes três subproblemas é implementada respectivamente pelas classes `CF16VTilePlotter`, `CF16VTileWalker` e `CF16VMouseMap`. No entanto, o

programador do jogo pode criar a sua própria solução implementando as interfaces `IF16VTilePlotter`, `IF16VTileWalker` e `IF16VMouseMap` e as registrando com a classe `CF16VMap`.

A classe `CF16VMap` modela um mapa de *tiles* (ver seção 2.4.3.1) e é responsável por gerenciar quais os objetos que devem estar visíveis na tela do computador. Para isso, as classes que implementam as interfaces `IF16VTilePlotter`, `IF16VTileWalker` e `IF16VMouseMap` são usadas para fazer o mapeamento entre o espaço do mundo e o espaço da tela.

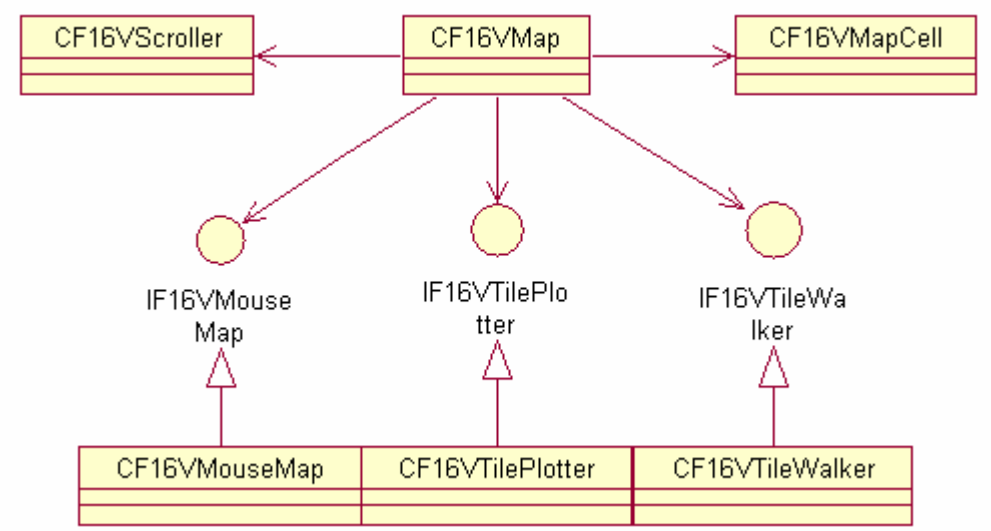

Figura 6-19 Diagrama de classes do gerenciador do mundo.

Além disso, a classe `CF16VMap` provê um sistema de *scroll* através da classe `CF16VScroller`, possibilitando a navegação por todo o espaço do mundo de forma transparente ao programador do jogo.

A classe `CF16VMapCell` representa os *tiles* no mapa de *tiles* (*TileMap*). É através desta classe que as operações envolvendo *tiles* serão realizadas, como por exemplo, a renderização.

Para fazer uso das classes acima, é necessário seguir os seguintes passos:

1. Criar uma instância da classe CF16VMap;
2. Declarar uma classe que herde de `CF16VMapCell` e implementar o método `PaintCell`;

3. Criar um *array* de objetos da classe que herdou de `CF16VMapCell`, tantos quantos forem os números de tiles no mapa;
4. Registrar as células no mapa através do método `CF16VMap::Create`;
5. Ajustar o tamanho do tile no mouse *map*;
6. No método `ProcessFrame` do estado corrente (ver seção 6.2.1), realizar o *scroll* e mandar o mapa renderizar a janela visível (ver seção 2.4.3.1 para maiores informações) na tela.

A Figura 6-20 apresenta uma demonstração simples de como usar este módulo. Os trechos de código que correspondem aos passos citados anteriormente estão em negrito.

```
#define MAPWIDTH 16
#define MAPHEIGHT 16


class Sample10MapCell;
CF16VMapCell**    g_mapCells;
CF16VTileset*  g_tiles;
CHAR              g_buffer[32];

CF16VMap          g_map; // PASSO 1

// PASSO 2
class Sample10MapCell : public CF16VMapCell {

public:
      virtual HRESULT    PaintCell(CF16VSurface *dest, LONG x, LONG y);
};

HRESULT      Sample10MapCell::PaintCell(CF16VSurface *dest, LONG x,
                                        LONG y) {
      if(dest) return g_tiles->DrawFast(dest, x, y, 3);
      else return -1;
}

class CF16VSample10GameState : public CF16VGameState{
public:
      CF16VSample10GameState();
      virtual ~CF16VSample10GameState();
      virtual void      IncludeGameState();
      virtual void      RemoveGameState();
      virtual void      KeyDown( int iKey );
      virtual void MouseButtonUp( int iButton );
      virtual HRESULT ProcessFrame(void);
};


void  CF16VSample10GameState::IncludeGameState() {
```

```
      // PASSO 3
      g_mapCells = new CF16VMapCell*[MAPWIDTH*MAPHEIGHT];
      for (int i = 0; i < MAPWIDTH*MAPHEIGHT; i++ ) {
            g_mapCells[i] = new Sample10MapCell;

      }
      ...
      g_tiles = new CF16VTileset();
      g_tiles->Create(CF16VGameManager::GetGameManager()-
>GetVideoEngine(),"Tiles.bmp");

      // PASSO 4
      g_map.Create(CF16VGameManager::GetGameManager()->GetVideoEngine(),
ISOMAP_SLIDE, MAPWIDTH,MAPHEIGHT, g_mapCells);
      // PASSO 5
      g_map.GetMouseMap()->Create(64,32);
}


HRESULT CF16VSample10GameState::ProcessFrame(void) {
      CF16VVideoEngine *ve = GetVideoEngine();
      CF16VSurface* back = ve->GetBackBuffer();

      POINT ptMouse, ptCursor;
      GetInputEngine()->GetMouse()->GetMousePos(&ptMouse.x, &ptMouse.y);
      // realizar scroll se o mouse estiver perto da borda da tela.
      LONG scrollx =0;
      LONG scrolly = 0;
      //left scroll?
      if(ptMouse.x<10) scrollx = ptMouse.x-10;

      //upward scroll?
      if(ptMouse.y<8) scrolly=ptMouse.y-8;

      //right scroll?
      if(ptMouse.x>=width-10) scrollx=ptMouse.x-(width-10);

      //downward scroll?
      if(ptMouse.y>=height-8) scrolly=ptMouse.y-(height-8);

      back->Fill(0);
      // PASSO 6
      g_map.ScrollFrame(scrollx, scrolly);
      g_map.UpdateFrame();

      return 0;
}

// O restante é a WinMain já visto anteriormente.
```

Figura 6-20 Código que demonstra como utilizar o módulo do mundo.

No passo 1 uma instância da classe `CF16VMap` é criada. No passo 2, uma nova classe chamada `Sample10MapCell` que herda de `CF16VMapCell` é declarada e um novo método `PaintCell` é implementado. Neste caso, o método `PaintCell` simplesmente desenha na tela um *tile*. Como este método é chamado para cada célula que estiver visível na tela, a renderização do cenário isométrico fica garantida. No passo 3 um *array* de células é criado e no passo 4 esse *array* é cadastrado no mapa através do método `Create` da classe `CF16VMap`. Além das células, o método `Create` recebe o tipo de mapa (slide, staggered, diamond ou retangular) a ser criado e as dimensões dele. No passo 5 o tamanho de um *tile* é ajustado no mouse *map* através do método `Create` da classe `CF16VMouseMap` cujos parâmetros são respectivamente a largura e a altura de um *tile*. Por fim, no passo 6 o *scroll* é realizado se o mouse se encontra na borda da tela e o mapa é renderizado através do método `UpdateFrame` da classe `CF16VMap`.

Na próxima seção será mostrada a validação do framework através da implementação de três protótipos de jogos.

6.3 ESTUDO DE CASOS

Validar um *framework* não é uma tarefa fácil. Uma possível abordagem para este problema seria a de realizar duas implementações para um único jogo. Uma utilizando o *Forge 16V* e outra não. Assim seria possível comparar os dois projetos e verificar a performance, tempo de desenvolvimento e as facilidades e limitações do uso do *Forge 16V*. Contudo, esta abordagem possui um número muito grande de variáveis, o que a torna não muito confiável. Por exemplo, é possível utilizar um mesmo grupo ou dois grupos distintos de desenvolvedores na implementação das duas versões. No primeiro caso, a implementação da aplicação uma segunda vez seria mais simples. Já no segundo caso, os códigos desenvolvidos poderiam ser tão diferentes que seria muito difícil compará-los.

Para diminuir a complexidade do processo, resolveu-se então validar o *Forge 16V* na disciplina de Projeto e Implementação de Jogos [5] do curso de Graduação em Ciência da Computação do Centro de Informática da UFPE, por intermédio do desenvolvimento de três jogos (apenas uma versão foi implementada e usava o motor): Knock'em, CinFarm e Pirralhos. Estes projetos foram desenvolvidos por três equipes distintas que não estiveram

envolvidas na implementação do *Forge 16V* e ao final de cada projeto, elas responderam a um questionário (ver Apêndice B) com perguntas sobre usabilidade, performance e utilidade do *Forge 16V*. Todos os projetos foram desenvolvidos num período de 2 meses, sem a dedicação exclusiva dos programadores.

Na seção a seguir serão mostrados os jogos desenvolvidos e, logo após, serão apresentadas as opiniões dos grupos a respeito do *Forge 16V*.

6.3.1 Jogos Desenvolvidos

6.3.1.1 Knock'em

O Knock'em [64] é um jogo 2D de luta no estilo dos antigos *arcades* [65]. No jogo, o usuário é um lutador que precisa ganhar dinheiro para poder treinar novos golpes e assim melhorar as chances de vitória. Para ganhar dinheiro, o jogador deve participar de torneios, e a cada vitória, uma soma é adicionada na conta do lutador. Para facilitar a vida do jogador, os primeiros torneios são bem fáceis. Com o passar do tempo, o jogador pode aprimorar-se e com isso participar de torneios mais difíceis.

No Knock'em existem dois modos de jogo: *modo combate* e *modo preparação*. Durante o modo combate, o jogador conduz seu lutador nos combates executando seus golpes através de comandos. Durante o modo preparação, o jogador define de quais torneios o lutador irá participar e gerencia sua evolução no jogo.

A diferença do Knock'em dos demais jogos de luta, está na evolução do personagem principal entre as lutas e torneios. Ao vencer lutas e torneios, o personagem adquire uma certa quantia em dinheiro. Esta quantia pode ser usada para pagar diferentes tipos de treinamentos. Cada treinamento melhora uma habilidade específica do personagem, como sua destreza, força ou resistência. Além de treinamentos físicos, existem alguns treinamentos espirituais que permitem ao jogador aprimorar certos golpes especiais chamados magias. Quanto mais complexo for o torneio a ser disputado pelo jogador, mais bem-preparados serão seus adversários. Dessa forma, o jogador deve preocupar-se com o planejamento da evolução de seu personagem, através dos treinamentos, na intenção de se equiparar com seus inimigos.

Figura 6-21 *Screenshot* do jogo Knock'em.

A implementação do jogo Knock'em utilizou os módulos gráfico, de som e entrada e fez bastante uso do gerenciador de estados do *Forge 16V*. Além disso, foi consenso entre os membros da equipe que o motor agilizou o processo de implementação, fazendo com que o protótipo ficasse pronto em menos de 2 meses.

O Knock'em proporcionou um bom teste de performance para o motor pois foram utilizadas mais de 400 imagens no total e o jogo rodou a uma taxa média de 35 quadros por segundo em um Pentium II 400 MHz com uma placa de vídeo GForce 2 da NVIDIA com 32 MB de memória, numa tela de 800x600 *pixels* com 16 *bits* de cores.

No que diz respeito ao som, o jogo utilizou arquivos MID e WAV para a trilha e efeitos sonoros, sem encontrar qualquer problema de usabilidade no módulo de som. Quanto ao gerenciador de entrada, apenas o teclado foi utilizado.

6.3.1.2 CinFarm

O CinFarm é um jogo isométrico que simula o gerenciamento de uma fazenda. Esse protótipo demonstra a viabilidade de desenvolver jogos isométricos usando-se o *Forge 16V*.

O usuário controla uma fazenda e nela pode cultivar plantações e criar gado (bovinos, eqüinos, suínos e caprinos). As plantações e o gado precisam ser tratados de

maneira adequada, para que gerem lucros a serem reinvestidos na fazenda. Também é possível a compra ou aluguel de máquinas (tratores, colhedeiras, etc.) para diversas funções. O usuário precisa gerenciar o gado (dando comida e água ou aplicando remédios) e as plantações (aplicação de inseticidas, fungicidas, fertilizantes, etc.) que ele escolher criar/ plantar. O grande desafio do jogo será expandir continuamente a fazenda sem perder controle sobre tudo que é criado nela. O usuário também poderá escolher alcançar prêmios em exposições de animais.

Figura 6-22 *Screenshots* do jogo CinFarm.

No que diz respeito à implementação, o CinFarm utilizou o módulo gráfico, o de som, o de entrada e o gerenciador do mundo do *Forge 16V*. A opinião geral do grupo de desenvolvimento foi que o motor realmente poupou trabalho, principalmente no que diz respeito da renderização isométrica.

A performance do motor foi avaliada usando-se um Pentium II 400 MHz com uma placa de vídeo GForce 2 de 32 MB. Em uma tela de 800x600 com 16 bits de cores, foram renderizados 500 objetos do mundo e o jogo apresentou uma performance média de 30 quadros por segundo, que é apropriada para a realização de animações [28].

A sonorização do jogo foi realizada com arquivos WAV e MIDI. Além disso, o jogo fez uso do teclado e do mouse.

6.3.1.3 Pirralhos

O jogo Pirralhos é uma mistura de combate e estratégia. O objetivo do jogo é destruir os personagens (bebês) dos outros jogadores em combates e com isso, ganhar mais pontos de experiência. Para isso, cada jogador dispõe de um conjunto de armas iniciais. À medida que os jogadores vão acumulando pontos, os personagens podem ser evoluídos com novas armas e acessórios para os combates futuros.

Figura 6-23 *Screenshot* do jogo Pirralhos.

O jogo utiliza cenários bidimensionais para formar as arenas de combate. Na implementação, os seguintes módulos do *Forge 16V* foram utilizados: gráfico, entrada (mouse e teclado) e som. O grupo de desenvolvimento relatou que o uso do motor abstraiu muito o uso do DirectX pois nenhuma referência à biblioteca da Microsoft foi encontrada nas interfaces do *Forge 16V*. Além disso, o grupo afirmou que o uso do motor agilizou muito o processo de desenvolvimento.

A performance do jogo foi de 37 quadros por segundo rodando em um Pentium II de 400 MHz, com uma GForce 2 de 32 MB de memória de vídeo, provando mais uma vez a boa performance do motor. Contudo, o grupo encontrou problemas de integração do módulo de som do *Forge 16V* com um módulo de rede escrito por eles (conflito na

inicialização do DirectX). Isso impossibilitou que este os módulos rodassem ao mesmo tempo.

6.3.2 Resultado da Avaliação

Como mencionado anteriormente, após a conclusão da disciplina, as equipes receberam um questionário de avaliação do motor. Esse questionário tinha intenção de identificar as qualidades e defeitos do *framework*, bem como servir de registro da usabilidade e performance conseguidas até então.

As equipes relataram que o uso do motor agilizou o processo de desenvolvimento do jogo, pois permitiu a concentração de esforços na implementação da lógica do jogo. No que diz respeito à performance, as equipes que desenvolveram os jogos com cenários bidimensionais, a classificaram como “muito boa”, enquanto a equipe de desenvolvimento do CinFarm (cenário isométrico) a classificou de “boa”. Uma pequena perda de performance na execução de jogos isométricos era esperada, devido ao grande número de cálculos feitos para mapear o sistema de coordenadas do mundo no sistema de coordenadas da tela.

Quanto à usabilidade, no geral as equipes acharam o *Forge 16V* fácil de usar. No entanto, cada equipe apresentou dificuldade em lidar com um módulo em particular. Por exemplo, a equipe que desenvolveu o jogo Pirralhos apresentou dificuldade para utilizar o módulo de som, enquanto as outras classificaram este módulo como fácil de usar.

Um outro ponto importante foi que duas das equipes acharam que o modelo de estados utilizados no módulo principal (ver seção 6.2.1) mostrou-se bastante útil no desenvolvimento dos jogos. Apenas a equipe do CinFarm não chegou a realmente fazer uso deste modelo (utilizou somente um estado).

Quando questionadas sobre os módulos considerados importantes de serem acrescentados ao motor, as equipes sugeriram módulo de rede, que permitiria jogos multiplayer, elementos de GUI (Graphical User Interface), módulo de inteligência artificial, além de modelagem física, entre outros.

Os questionários respondidos por cada uma das equipes podem ser encontrados na íntegra no Apêndice B desta dissertação.

6.4 REQUISITOS IMPLEMENTADOS

Na seção 2.5.2 foram apresentados os principais requisitos de um motor para jogos. Nesta seção serão vistos quais desses requisitos foram alcançados pelo *Forge 16V* na atual fase de desenvolvimento.

6.4.1 Fácil Utilização

Com o trabalho realizado para catalogar os principais problemas encontrados no motor sob a forma de padrões de projeto e com a documentação de código e exemplos, os grupos que desenvolveram os protótipos relataram que não tiveram muita dificuldade em usar o *Forge 16V*. De fato, eles elogiaram a facilidade de uso e reportaram que o uso do motor facilitou muito o desenvolvimento se comparado ao uso do DirectX [19].

6.4.2 Sistema de Renderização Gráfica Eficiente

A performance do rederizador gráfico mostrou-se bastante eficiente. Em todos os testes foi usado um Pentium II 400 MHz com uma placa de vídeo GForce 2 de 32MB de memória, o que significa uma configuração de uma máquina relativamente lenta para os padrões atuais. Mesmo nivelando por baixo o *hardware* utilizado, todos os grupos que desenvolveram os protótipos relataram uma performance satisfatória do motor.

6.4.3 Garantir elementos que ajudem a Jogabilidade e a Experiência do Jogador

No *Forge 16V* foi implementado o suporte de alguns dos principais elementos que auxiliam a jogabilidade e a experiência do jogador. Dentre esse podemos dar destaque ao suporte a joystick, mouses de três ou mais botões, volantes e o suporte a sons 3D. O destaque negativo foram os elementos que interface gráfica que não foram implementados.

6.4.4 Ferramentas que auxiliem a Criação de Jogos

O único elemento que auxilia a criação de novos jogos que foi criado pelo *Forge 16V* foi o subsistema de log que auxilia a depuração de código. Até a atual fase de

desenvolvimento não foi criado um editor de cenários para ajudar a construir as fases dos jogos.

6.4.5 Suporte a Sons de Efeitos e Música

Foi implementado no *Forge 16V* o suporte a sons Wave. Esses sons podem tocar ao mesmo tempo garantindo assim que os efeitos toque ao mesmo tempo em que uma música de background está tocando. Portanto, esse requisito foi totalmente satisfeito.

6.4.6 Suporte a Conectividade

Na atual fase de implementação, esse requisito não foi satisfeito pelo *Forge 16V*. Esse requisito e os que se seguem, podem ser adicionados na arquitetura do *Forge 16V* sem maiores problemas devido a sua natureza modular.

6.4.7 Suporte a uma Liguagem de Script

Esse requisito também não foi satisfeito pelo *Forge 16V*.

6.4.8 Implementação de Algoritmos Básicos de IA

Esse requisito também não foi satisfeito pelo *Forge 16V*.

6.4.9 Gerenciamento de Objetos no Mundo

Esse requisito também não foi satisfeito pelo *Forge 16V*.

6.5 CONCLUSÕES

Neste capítulo foram apresentadas as escolhas realizadas na fase de implementação do *Forge 16V*, como por exemplo, a utilização da linguagem C++ no desenvolvimento. Além disso, foram mostrados os detalhes de implementação de cada módulo que efetivamente chegou a ser implementado:

- Gerenciador Principal;
- Gerenciador Gráfico;
- Gerenciador de Entrada;
- Gerenciador de Som;

- Gerenciador de Log;
- Gerenciador de Mundo.

Por fim, foram mostrados os protótipos de jogos desenvolvido utilizando-se o *Forge 16V*:

- Knock'em;
- CinFarm;
- Pirralhos.

Estes jogos foram desenvolvidos por equipes que nada tinham a ver com o projeto do *Forge 16V*. Assim, foi possível avaliar a usabilidade e a performance do motor. Além disso, foi possível comprovar que o motor realmente auxiliou o processo de desenvolvimento, pois permitiu que os grupos se concentrassem mais no código relativo à lógica do jogo.

Por fim, foram vistos quais os requisitos de um motor de jogos foram satisfeitos com a atual implementação do *Forge 16V*.

Capítulo 7

Conclusões

Neste capítulo será mostrado um resumo do que foi visto neste trabalho e suas principais contribuições. Também serão apresentadas as principais dificuldades encontradas e algumas sugestões para trabalhos futuros.

Neste trabalho foi vista a evolução do processo desenvolvimento de jogos de computador desde a década de oitenta até os dias atuais, mostrando também a mudança de mentalidade dos desenvolvedores de jogos durante esse período.

Também foi mostrado que, devido à grande complexidade dos projetos atuais e aos altos orçamentos envolvidos, o conceito de *framework* tornou-se chave para o desenvolvimento de jogos, pois diminui os riscos do projeto e possibilita que grande parte do código seja reutilizada. Além disso, os *frameworks* possibilitam que os desenvolvedores de jogos não se preocupem com detalhes de baixo nível, reduzindo assim a complexidade, o tempo e o custo envolvidos no processo de desenvolvimento de um jogo.

Visando atender à crescente demanda acadêmica na área de desenvolvimento de jogos de computador e a dar apoio à indústria local de jogos, o Centro de Informática da UFPE criou um grupo de pesquisa para estudar o assunto. Um dos trabalhos que fez parte deste esforço foi o desenvolvimento de um *framework* denominado de *Forge V8*. Como foi mostrado, devido à inexperiência do grupo nesse tipo de aplicação, a tarefa foi subestimada e o projeto apresentou alguns problemas que levaram a um motor pouco útil na prática. Contudo, o trabalho foi um marco que possibilitou o entendimento do processo de desenvolvimento de um jogo e realizou uma série de estudos sobre o assunto.

Tentando não perder o esforço realizado no *Forge V8* e aproveitando da experiência adquirida, resolveu-se então propor neste trabalho um novo *framework* denominado de *Forge 16V*. Como foi apresentado, o principal objetivo do *Forge 16V* era dispor de um motor para jogos em estado funcional, o mais rápido possível. Para conseguir isto as seguintes escolhas foram tomadas:

- Desenvolvimento incremental;
- Desenvolver um motor para os jogos com cenários isométricos e bidimensionais;
- Desenvolver um motor utilizando transferências de *bits* ao invés de técnicas 3D;
- Usar somente a biblioteca multimídia DirectX, ou seja, desenvolver um motor para jogos que sejam exucultados somente na plataforma Windows;
- Não trabalhar com código *multithread* nesta primeira fase do projeto.

Além disso, estava previsto dar continuidade ao estudo de padrões de projeto no contexto de um framework para jogos, dada a pouca literatura disponível.

Para avaliar a usabilidade, performance, robustez e principalmente a utilidade do *Forge 16V*, três protótipos de jogos foram construídos: o Knock'em, o CinFarm e o Pirralhos. Estes projetos foram desenvolvidos por pessoas que nada tinham a ver com o *Forge 16V* e, como foi mostrado, a avaliação do motor foi bastante positiva.

No restante deste capítulo serão mostrados as principais contribuições desta pesquisa, as principais dificuldades encontradas no desenvolvimento do projeto e alguns possíveis trabalhos futuros.

7.1 CONTRIBUIÇÕES

Neste trabalho foi proposto um *framework* para a implementação de jogos isométricos e bidimensionais, denominado de *Forge 16V*. Além disso, uma outra contribuição importante foi a implementação deste *framework*. Embora o motor esteja parcialmente implementado, ele encontra-se em estado funcional e pôde ser utilizado na disciplina de Projeto e Implementação de Jogos [5] do CIn-UFPE, o que resultou no desenvolvimento de três protótipos de jogos utilizando o motor. Um outro ponto importante é que o código fonte do *Forge 16V* estará aberto para a utilização em outros projetos acadêmicos.

Por fim, este trabalho continuou a pesquisa iniciada no projeto do *Forge V8* [7] sobre os principais problemas encontrados no desenvolvimento de *frameworks* para jogos, em forma de padrão de projetos. Esta dissertação contribuiu com novos problemas encontrados e suas respectivas soluções. Este catálogo original de problemas e soluções pode ser usado futuramente como base para outros projetos que se destinem a desenvolver motores para jogos, ou mesmo, para os jogos propriamente ditos.

7.2 DIFICULDADES

Uma das principais dificuldades foi a pouca documentação encontrada sobre o desenvolvimento de jogos isométricos. Inicialmente só foram encontrados alguns artigos a

respeito do assunto na Internet. Posteriormente foi encontrado um livro [46] que cobria com maiores detalhes o funcionamento dos jogos isométricos.

Outra dificuldade encontrada foi a falta de conhecimento mais profundo no DirectX, o que demandou um tempo significativo de aprendizagem.

Por fim, a complexidade de um *framework*, principalmente para o desenvolvimento de jogos de computador que exigem um código altamente otimizado, representou um grande desafio.

7.3 TRABALHOS FUTUROS

Para dar continuidade a este projeto, existem alguns trabalhos que podem ser realizados:

- Implementação dos módulos que não foram ou que ficaram parcialmente implementados (Gerenciador de Multiusuários, Gerenciador do Mundo, Gerenciador de Modelagem Física, Gerenciador de IA e o Editor de Cenários);
- Levantamento mais detalhado dos principais problemas encontrados no uso de IA dos jogos. Até o momento o grupo de pesquisa de jogos do CIn só realizou uma pesquisa muito preliminar sobre o assunto;
- Utilizar mais de um *thread* de execução para separar a lógica do jogo e a renderização gráfica do restante do motor [66, 67], possibilitando assim uma melhor distribuição do tempo de cpu dado a cada módulo.
- Otimização de código para conseguir uma melhor performance do motor.
- Implementação de uma GUI (janelas, menus, listas, textos, *combobox*,etc.) para o módulo gráfico. Isto facilitaria significantemente a interação do o usuário com o jogo.

Além desses trabalhos já citados, a experiência adquirida neste trabalho pode ser utilizada para se desenvolver um motor para outras plataformas, por exemplo, XBox [68], Play Station 2 [69], ou mesmo para dispositivos móveis.

Apêndice A

Padrão de Codificação no Forge 16V

Neste apêndice será apresentado o padrão de codificação utilizado no Forge 16V.

Neste apêndice será apresentado o padrão de codificação utilizado no *Forge 16V*, que é baseado no padrão do *Forge V8* com as modificações necessárias para evitar o problema apresentado na seção 3.4.2.1.

Um padrão de codificação é importante pois melhora a legibilidade do código produzido e, conseqüentemente, facilita a manutenção.

A.1 CONVENÇÕES DE NOMENCLATURA

A seguir serão mostradas as regras utilizadas para a nomenclatura de classes, interfaces, variáveis, constantes, enumerações e métodos.

A.1.1 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 F16V, e por fim, seguido de 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, ou seja, "_";
- Usar *PascalCasing*[1].

Exemplos: `CF16VGameEngine`, `CF16VSprite`, `CF16VTileset`;

A.1.2 Definição de Interfaces

Para a nomenclatura das interfaces deve seguir as seguintes regras:

- Usar o prefixo I para designar o tipo <<interface>>, seguido por F16V, e por fim, seguido de nomes e frases nominais;
- O nome da interface deve sempre ser iniciado com letra maiúscula;
- Usar abreviações de fácil entendimento em nomes de interfaces extensas;
- Não usar sublinhado, ou seja, "_";

[1] Separar os nomes através de letras maiúsculas. Ex. MotorDeJogos.

- Usar *PascalCasing*;

Exemplos: IF16VCache, IF16VTilemap, IF16VMousemap.

A.1.3 Definição de Variáveis

O nome das variáveis devem seguir o padrão da notação húngara [41]. Sendo assim, as seguintes regras devem ser obedecidas:

- As variáveis devem ser iniciadas por letras minúsculas;
- Todas as variáveis devem ter um prefixo que informa o seu tipo. Os principais prefixos são: `b` para *boolean*, `c` para *char*, `i` para *inteiro*, `str` para *string*, `sz` para *string* terminada em *null*, `f` para *float*, `d` para *doublé* e `p` para ponteiro;
- Usar *PascalCasing*;

Exemplos: iNumDeSurfaces, pCelula, szNome.

O restante dessa seção apresenta outras regras baseado no escopo da variável.

A.1.3.1 Variáveis Globais

- Variáveis globais internas a um módulo ou classe iniciam com o prefixo "`m_`" seguido pelo prefixo do tipo e qualificador. Ex. `m_iNumTiles`, `m_szName`;
- Variáveis globais da aplicação iniciam com o prefixo "`g_`" seguido pelo prefixo do tipo e qualificador. Ex. `g_iScreenWidth`, `g_fSquareRoot`.

A.1.3.2 Variáveis Locais

- Variáveis locais, internas aos métodos, iniciam com o prefixo "`l_`" seguido pelo prefixo do tipo e qualificador. Ex. `l_iCount`, `l_szText`.

A.1.4 Constantes

As constantes devem seguir o padrão dado pelas seguintes regras:

- O nome das constantes devem ser precedido com o prefixo "`F16V`";

- 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: F16VDEFAULT_SCREEH_WIDTH, F16VDEFAULT_COLOR.

A.1.5 Enumerações

As enumerações devem seguir o seguinte padrão:

- O nome das enumerações devem ser precedido com o prefixo "`F16V`";
- 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 F16VIMAGE_TYPE {
        F16VIMAGE_TYPE_JPEG=1;
        F16VIMAGE_TYPE_PNG=2;
}
```

A.1.6 Métodos, funções e procedimentos

Os métodos, funções e procedimentos devem seguir as 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`, `WriteRegisterKey`;

A.1.7 Parâmetros

As variáveis devem seguir as mesmas regras aplicadas para as variáveis em geral.

A.2 COMENTÁRIOS

Todo arquivo de código do *Forge 16V* deve conter comentários a respeito da implementação. A seguir serão mostrados os padrões adotados para os principais elementos dos programas.

A.2.1 Comentários de Classes

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

```
/**
 * Desc: Descrição da sua funcionalidade
 * @author Autores
 */
```

A.2.2 Comentários de variáveis, constantes e enumerações

Os comentários devem estar posicionados logo acima das declarações de variáveis, constantes e enumerações. Exemplo:

```
// Comentário
int m_iCount;
```

A.2.3 Comentários de métodos, funções e procedimentos

Comentários de métodos, procedimentos e funções devem seguir o seguinte padrão:

```
/**
* Descrição do método.
* @param <nome do parâmetro> descrição
* @return descrição
*/
```

A.2.4 Cabeçalhos dos Arquivos

Os arquivos em C++ são divididos em arquivos de cabeçalho com extensão .h (definem a estruturação das classes) e arquivos de implementação com extensão .cpp (implementam os métodos). Esses arquivos devem apresentar um cabeçalho, incluindo o seguinte comentário, logo no início do arquivo:

```
/*******************************************
    File:  Nome do Arquivo
    Description: propósito do arquivo
    begin            : Wen May 29 2002
    copyright        : (C) 2002 by Nome do Autor
    email            : email do autor
*******************************************/
```

A.2.5 Identação

A identação do código deve ser de 4 espaços.

Apêndice B

Questionários Aplicados

Neste apêndice serão mostrados os questionários aplicados aos grupos que desenvolveram protótipos de jogos usando o Forge 16V.

Neste apêndice serão apresentados os questionários aplicados aos grupos que desenvolveram protótipos de jogos usando o *Forge 16V*. O intuito desses questionários era o de avaliar a usabilidade, performance e principalmente quanto o *Forge 16V* ajudou no desenvolvimento desses projetos.

B.1 KNOCK'EM

Grupo: André Furtado, André Amaral, Gustavo Andrade e Fernando Andrade.

1) Escreva como foi a sua experiência ao usar o *Forge 16V*.

Foi útil. O motor agilizou o processo de implementação do jogo, melhorando a produtividade do processo. Sem ele, possivelmente não teriam sido implementadas todas as funcionalidades do jogo, e a qualidade artística (*sprites* e sons) teria sido prejudicada em detrimento de necessidades de implementação.

2) Enumere os módulos efetivamente usados indicando o grau de facilidade (fácil, médio ou complicado). Acrecente as justificativas

Para o KnockEm, foram usados os módulos de vídeo (GraphicEngine), entrada de dados via teclado (InputEngine/Keyboard), som (SoundEngine) e o modo de gerenciamento de estados. O módulo de vídeo apresentou algumas dificuldades de manipulação, em virtude de problemas no motor (que foram corrigidas logo que detectadas), falta de experiência com os padrões do Windows, e a grande quantidade de *sprites* do jogo.

Os demais módulos foram usados com facilidade, não apresentando problemas.

3) Dos(as) módulos/carcterísticas propostos(as) no *Forge 16V*, indique quais foram "absolutamente fundamentais", "úteis" ou "irrelevantes". Acrescente alguma justificativa.

Os módulo de vídeo, entrada via teclado e gerenciamento de estados foram fundamentais para o KnockEm. Entretanto, nem todas as funcionalidades de cada módulo

foram utilizadas, como a manipulação de imagens JPEG, que poderiam prejudicar as informações de controle dos *sprites* com a compactação. O módulo de som foi útil, mas não fundamental para desenvolvimento do jogo. Os demais módulos foram irrelevantes.

4) O que o grupo achou do modelo de estados adotado (GameState) ?

Muito bom. O modelo de estados foi fundamental para o gerenciamento do jogo, uma vez que no Knock'em cada estado interpreta diferentemente as entradas do jogador e utiliza arquivos gráficos diferentes. Com o modelo de *GameState*, esses arquivos não precisam estar constantemente em memória, o que prejudicaria o desempenho do jogo. A título de exemplo, o KnockEm possui seis *GameStates*, como o estado de luta e o estado de treinamento.

5) Como foi a performance do motor?

Muito boa. O único problema de performance que ocorreu durante o desenvolvimento (relativo ao gerenciamento de memória/cache para os *sprites*) foi resolvido por novas versões do motor disponibilizadas logo que o problema foi detectado.

6) Que característica você gostaria de ver implementada no motor (pode ser mais de uma), indicando o quão ela seria importante para você (absolutamente fundamental ou útil) ?

O suporte a jogos multi-player seria muito útil (mas não fundamental) para o Knock'em, uma vez que o modo em rede eliminaria o desconforto atual de compartilhamento do teclado no modo de dois jogadores, sendo um diferencial em relação aos jogos atuais de luta.

Um módulo de Inteligência Artificial também seria útil para a implementação das características dos agentes sintéticos, permitindo que cada personagem tivesse um comportamento específico. Entretanto, este módulo precisaria ser genérico o suficiente para poder ser utilizado pelo Knock'em (não apenas implementar algoritmos padrões de IA, mas também suportar a "programação" de agentes)

B.2 CINFARM

Grupo: Dante Torres, Fabiano Rolim, Marília Gama e Pablo Sampaio.

1) Escreva como foi a sua experiência ao usar o *Forge 16V*.

O *Forge 16V* foi útil ao nosso projeto. O motor realmente poupou bastante trabalho nosso, principalmente no que se refere ao módulo de renderização isométrica, apesar de ter sido necessário tratar independentemente um problema desse módulo.

2) Enumere os módulos efetivamente usados indicando o grau de facilidade (fácil, médio ou complicado). Acrecente as justificativas.

Módulos	Uso	Justificativa/Sugestões
Isométrico	Fácil	O módulo oferece uma maneira muito fácil de renderizar um mapa isométrico de qualquer dos estilos (diamond, etc.), bem como de "caminhar" por ele. Oferece automaticamente a facilidade do scroll.
Motor gráfico geral	Fácil	Facilita a criação de conjuntos de imagens (Tileset) e de criação de aplicações *windowed* ou *fullscreen*. *Surfaces* também são muito úteis e fáceis de usar.
Motor de Som	Fácil	No entanto, apresenta alguns problemas estranhos.
Entrada	Médio	Nem sempre o tratamento da entrada é muito amigável. Deveria haver uma tradução direta entre clique do mouse e posição em *tiles*, se possível, através de variáveis de *callback*. Além disso, os métodos da classe que tratam o mouse estão um tanto despadronizados, ora recebendo POINT ora recebendo **x** e **y** separados como parâmetros.

3) Dos(as) módulos/carcterísticas propostos(as) no *Forge 16V*, indique quais foram "absolutamente fundamentais", "úteis" ou "irrelevantes". Acrescente alguma justificativa.

Módulos	Importância	Justificativa
Isométrico	Fundamental	Porque nosso jogo é isométrico, e esse módulo oferece muitas funcionalidades úteis para tais jogos.
Motor gráfico geral	Útil	Facilita trabalhar com modos gráficos e sprites.
Motor de Som	Útil	Oferece uma interface mais simplificada (do que DirectX) para trabalhar com sons.
Entrada	Fundamental	Nos poupa de trabalhar diretamente com a complicada biblioteca DirectInput.

4) O que o grupo achou do modelo de estados adotado (GameState) ?

Bom. A classe GameState se confunde um pouco com a classe GameManager. No nosso jogo havia um único GameState que era efetivamente o gerenciador principal do nosso jogo e a dependência dessa classe em relação GameManager era um tanto incômoda para nós. Por exemplo: no *callback* MouseButtonDown, era natural que precisássemos da posição do clique. No entanto, para obter tal posição era necessário fazer a chamada pouco amigável:

```
> CFV8iGameManager::GetGameManager()->GetInputEngine()->GetMouse()
```

Essa mesma crítica vale para todos os *inputs* e também para o uso do vídeo.

No geral, entretanto, GameState foi bastante útil para funcionar como gerenciador principal do nosso jogo, e os problemas apresentados acima foram facilmente resolvidos usando variáveis no motor para guardar as instâncias do mouse, do vídeo,

5) Como foi a performance do motor?

Boa.

6) Que característica você gostaria de ver implementada no motor (pode ser mais de uma), indicando o quão ela seria importante para você (absolutamente fundamental ou útil) ?

Uma funcionalidade que poderia ser útil seria a de criar várias camadas de *tiles* para dar mais flexibilidade ao motor.

Algo útil também poderia ser a fusão, ou talvez uma melhor integração entre as classes GameState e GameManager.

Prover elementos de GUI (Graphical User Interface) também seria algo útil.

Seria útil oferecer compactação de arquivos de arquivos binários (de preferência, que seja possível guardar vários arquivos num único arquivo compactado).

B.3 PIRRALHOS

Grupo: Georgia Albuquerque, Ivo Nascimento, Mauro Faccenda, Rodrigo Santos, Saulo Jansen e Tiago Barros.

1) Escreva como foi a sua experiência ao usar o *Forge 16V*.

Útil. O uso do motor permitiu concentrar esforços nas partes importantes do jogo, não precisando se preocupar com diversos detalhes de implementação e uso do DirectX (não foi encontrada nenhuma referência nas interfaces do motor do uso do DirectX).

2) Enumere os módulos efetivamente usados indicando o grau de facilidade (fácil, médio ou complicado). Acrecente as justificativas.

Módulo de VideoEngine - Fácil

Módulo de GameState - Fácil

Módulo de Tileset/Animation - Fácil

Módulo de Som - Médio

3) Dos(as) módulos/carcterísticas propostos(as) no *Forge 16V*, indique quais foram "absolutamente fundamentais", "úteis" ou "irrelevantes". Acrescente alguma justificativa.

Todos os módulos que usamos foram fundamentais para o desenvolvimento do jogo.

4) O que o grupo achou do modelo de estados adotado (GameState) ?

Muito bom. Permitiu uma melhor organização do jogo. A escolha do uso de GameStates fez com que tivéssemos que estruturar melhor o jogo o que permitiu distribuir melhor o desenvolvimento do jogo.

5) Como foi a performance do motor?

Muito boa.

6) Que característica você gostaria de ver implementada no motor (pode ser mais de uma), indicando o quão ela seria importante para você (absolutamente fundamental ou útil) ?

Módulo de Rede

Módulo de GUI (botões, editbox, janelas...)

Módulo de Inteligência Artificial

Módulo de modelagem física (movimentação, detecção de colisão, etc...)

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Gamedev - Schools, disponível em: http://www.gamedev.net/reference/list.asp?categoryid=77 (13/11/2003).

[2] Game Developer Magazine, disponível em: http://www.gdmag.com (12/11/2003).

[3] WJogos, disponível em: http://www.icad.puc-rio.br/wjogos/ (12/11/2003).

[4] GAME-ON 2002, disponível em: http://biomath.rug.ac.be/~scs/conf/gameon2002/ (01/12/2003).

[5] Ramalho, G. Projeto e Implementação de Jogos, disponível em: http://www.cin.ufpe.br/~game/ (03/11/2003).

[6] Pessoa, C.A.C. (2001). *wGEM: um Framework de Desenvolvimento de Jogos para Dispositivos Móveis*, Dissertação de Mestrado, Centro de Informática, Universidade Federal de Pernambuco, Recife.

[7] Madeira, C.A.G. (2001). *Forge V8: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia*, Dissertação de Mestrado, Centro de Informática, Universidade Federal de Pernambuco, Recife.

[8] Siebra, C.A. (2000). *Uma Arquitetura para Suporte de Atores Sintéticos em Ambientes Virtuais- uma aplicação a jogos de estratégia*, Dissertação de Mestrado, Centro de Informática, UFPE, Recife.

[9] Silva, D.R.D. (2000). *Atores Sintéticos em Jogos de Aventura: o Projeto enigma no Campus*, Dissertação de Mestrado, Centro de Informática, UFPE, Recife.

[10] Valadares, J.F. (2002). *Modelagem de Sistemas Complexos usando redes Bayesianas: Estudo de Caso do FutSim*, Dissertação de Mestrado, Centro de Informática, UFPE, Recife.

[11] Jynx Playware, disponível em: http://www.jynx.com.br (10/11/2003).

[12] Ahearn, L. (2001). Budgeting and Scheduling Your Game, disponível em: http://www.gamasutra.com/features/20010504/ahearn_03.htm (11/11/2003).

[13] Epic Games, disponível em: http://www.epicgames.com (16/07/2003).

[14] Golgotha, disponível em: http://sourceforge.net/projects/golgotha (14/05/2003).

[15] Crystal Space 3D, disponível em: http://crystal.sourceforge.net (14/05/2003).

[16] Rollings, A. & Morris, D. (2000). *Game Architecture and Design*, The Coriolis Group.

[17] id Software, disponível em: http://www.idsoftware.com/ (junho de 2003).

[18] Doom, id Software, disponível em: http://www.idsoftware.com/games/doom/ (29/07/2003).

[19] API Microsoft DirectX, Microsoft, disponível em: http://www.microsoft.com/directx (15/06/2003).

[20] Visual Studio Home, disponível em: http://msdn.microsoft.com/vstudio/ (18/10/2003).

[21] Entertainment Software Association, ESA, disponível em: http://www.theesa.com (11/11/2003).

[22] Video Games Sales to Top $31 Billion in 2002, disponível em: http://www.gamasutra.com/php-bin/industry_news_display.php?story=1245 (13/05/2003).

[23] Schaefer, E. (2000). Postmortem: Blizzard Entertainment's Diablo II, disponível em: http://www.gamasutra.com/features/20001025/schaefer_01.htm (20/05/2003).

[24] Spector, W. (2003). Postmortem: Ion Storm's Deus Ex, disponível em: http://www.gamasutra.com/features/20001206/spector_04.htm (02/06/2003).

[25] OpenGL Architecture Review Board - ARB, disponível em: http://www.opengl.org/developers/about/arb.html (06/2003).

[26] API OpenAL, disponível em: http://www.openal.org (15/05/2003).

[27] Battaiola, A. (2000). "Jogos por Computador - Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação". *Anais da XIX Jornada de Atualização em Informática (JAI)*. SBC 2000,Vol.2, (pp. 83-123).

[28] Lamothe, A. (1999). *Tricks of the Windows Game Programming Gurus - Fundamentals of 2D and 3D Game Programming*, Sams.

[29] Barron, T. (2001). *Multiplayer Game Programming*, Prima Tech.

[30] Age of Wonders Portal, Triumph Studios, disponível em: http://www.ageofwonders.com (13/08/2003).

[31] Perez, A. (2000). *Advanced 3-D game programming using DirectX 7.0*, Wordware Publishing, Inc.

[32] Bell, G. (1998). Creating Backgrounds for 3D Games, disponível em: http://www.gamasutra.com/features/19981023/bell_01.htm (23/09/2003).

[33] Aklecha, V. (1999). *Object-Oriented Frameworks Using C++ and Corba*, The Corolis Group.

[34] Roberts, D. & Johnson, R. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, (02/01/2004).

[35] Fowler, M., Beck, K., Brant, J. & Opdyke, W. (1999). *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

[36] Genesis 3D, disponível em: http://www.genesis3d.com (18/05/2003).

[37] Touchdown Entertainment - 3D game development technology, Touchdown Entertainment, disponível em: http://www.touchdownentertainment.com/ (09/07/2003).

[38] id Software Technology Licensing Program, disponível em: http://www.idsoftware.com/business/home/technology/techlicense.php#endnote4 (18/06/2003).

[39] API OpenGL, disponível em: http://www.opengl.org (15/06/2003).

[40] Stroupstrup, B. (1997). *C++ Programming Language*, Addison-Wesley.

[41] Simonyi, C. (1999). Hungarian Notation, disponível em: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsgen/html/hunganotat.asp.

[42] Micosoft Foundation Class Library, disponível em: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/mfchm.asp (25/05/2003).

[43] Riemersma, T. (2001). Axonometric Projections - A Technical Overview, disponível em: http://www.gamedev.net/reference/articles/article1269.asp (01/07/2003).

[44] Visualização em 3D - Projeções, disponível em: http://gbdi.icmc.sc.usp.br/documentacao/apostilas/cg/downloads/ap_p.pdf (14/11/2003).

[45] Adams, J. (1999). Introduction To Isometric Engines, disponível em: http://www.gamedev.net/reference/articles/article744.asp (07/2003).

[46] Pazera, E. (2001). *Isometric Game Programming with DirectX 7.0*, Prima Tech.

[47] Michael, D. (1999). Tile/Map-Based Game Techniques: Base Data Structures, disponível em: http://www.gamedev.net/reference/articles/article837.asp (15/06/2003).

[48] Pazera, E. (1999). Isometric 'n' Hexagonal Maps Part I, disponível em: http://www.gamedev.net/reference/articles/article747.asp (14/06/2003).

[49] Johnson, R.E. (1992). "Documenting Frameworks using Patterns". *OOPSLA*.

[50] Hodorowicz, L. (2001). Elements Of A Game Engine, disponível em: http://www.flipcode.com/tutorials/tut_el_engine.shtml (06/06/2003).

[51] Fan, J., Ries, E. & Tenitchi, C. (1996). *Black Art of Java Game Programming*, Waite Group Press.

[52] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[53] GUI design elements, disponível em: http://osr5doc.ca.caldera.com:457/cgi-bin/getnav/VTCLG/vtclgN.style_elements.html (14/11/2003).

[54] Explanation of the Pluggable Look and Feel, disponível em: http://java.sun.com/products/jfc/jaccess-1.2.2/doc/pluggable.html (13/11/2003).

[55] Introduction to Computer Graphic/Image File Formats, disponível em: http://ei.cs.vt.edu/~netinfo/backupcs1604/graphicformats.html (05/11/2003).

[56] Grand, M. (2003). Pattern Summaries 8: Cache Management, disponível em: http://www.developer.com/java/other/article.php/630481 (15/08/2003).

[57] Simpson, Z.B. (2000). Gateway, (14/08/2003).

[58] Simpson, Z.B. (2000). Appearance Map, disponível em: http://www.gamedev.net/reference/articles/article1398.asp (27/08/2003).

[59] Simpson, Z.B. (2000). Model, disponível em: http://www.gamedev.net/reference/articles/article1406.asp (28/08/2003).

[60] Simpson, Z.B. (2000). Model Database, disponível em: http://www.gamedev.net/reference/articles/article1407.asp (28/08/2003).

[61] Halliday, D. & Resnick, R. (1988). *Fundamento de Física (Mecânica)*, Livros Técnicos e Científicos Editora S.A.

[62] Simpson, Z.B. (2000). Controller State Machine, disponível em: http://www.gamedev.net/reference/articles/article1400.asp (19/08/2003).

[63] Filho, W.M.d.A., Menezes, T.R.d., Vieira, M.F. & Ramalho, G.L. (2002). "Level Editor Functional Requirements for 2D & 2½D Games". *Workshop in Games and Digital Entertainment 2002 (WJogos'02)*. Fortaleza, CE, Brazil. (October, 2002).

[64] Neto, F.C.A., Andrade, G.D., Leitão, A.R.G.A., Furtado, A.W.B. & Ramalho, G.L. (2003). "Knock'em: Um Estudo de Caso de Processamento Gráfico e Inteligência Artificial para Jogos de Luta". *II Workshop de Jogos e Entretenimento Digital*. (pp. 105-112), Salvador: Sociedade Brasileira de Computação.

[65] History of Arcade Games, disponível em: http://www.hut.fi/~eye/videogames/arcade.html (16/11/2003).

[66] Dawson, B. (2001). "Micro-Threads for Game Object AI". In DeLoura, M. (ed.), *Game Programming Gems*, Vol. 2, Charles River Media.

[67] Carter, S. (2001). "Managing AI with Micro-Threads". In DeLoura, M. (ed.), *Game Programming Gems*, Vol. 2, Charles River Media.

[68] Microsoft Xbox, disponível em: http://www.microsoft.com/xbox (17/11/2003).

[69] Sony Play Station 2, disponível em: http://www.us.playstation.com (17/11/2003).