Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Ribeiro Poyart

IEngine – Uma interface abstrata para motores de jogos 3D

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática da PUC-Rio.

Orientador: Prof. Bruno Feijó

Rio de Janeiro

Julho de 2002

IENGINE – Uma Interface Abstrata para Motores de Jogos 3D

Dissertação de Mestrado apresentada por **Eduardo Ribeiro Poyart**, em 26 de abril de 2002, ao Departamento de Informática da PUC-Rio e aprovada pela Comissão Julgadora formada pelos seguintes professores:

Prof. **Bruno Feijó** (Orientador) - PUC-Rio

Prof. **Marcelo Dreux** - PUC-Rio

Prof. **Waldemar Celes Filho** - PUC-Rio

Visto e permitida a impressão,
Rio de Janeiro, 29 de julho de 2002.

Coordenador dos Programas de Pós-Graduação e
Pesquisa do Centro Técnico Científico

À minha família;

À Sanny.

Agradecimentos

À Vice Reitoria Acadêmica da PUC-Rio, pela bolsa de isenção concedida;

Ao Prof. Bruno Feijó, pelo incentivo e apoio;

Ao ICAD/IGames e a toda sua equipe, pela oportunidade de projetos usando IEngine;

Ao CNPq, pelo apoio concedido;

Aos novos colegas do ICAD, TeCGraf e da PUC em geral.

Resumo

Motores de jogos 3D (*3D game engines*) são APIs (*application programming interfaces*) que facilitam o desenvolvimento de jogos que possuem gráficos 3D em tempo real. É conveniente a utilização de um motor 3D quando se está desenvolvendo jogos, pois toda a funcionalidade de baixo nível já está construída. Porém, cada API de motor 3D é construída de uma forma diferente, tornando os motores incompatíveis entre si. O aprendizado também é complicado, pois em alguns motores a API não está bem estruturada. Este trabalho analisa esta questão, fazendo uma revisão crítica de alguns motores 3D existentes no mercado, e propõe uma camada de abstração sobre os motores 3D, chamada IEngine.

A camada IEngine dá uma API padronizada para os motores 3D. Uma aplicação construída sobre IEngine roda com qualquer dos motores 3D adaptados a ela, permitindo que a aplicação possa trocar de motor sem nenhuma alteração em seu código. A adaptação de um novo motor ao IEngine também é facilitada pela estruturação da API do IEngine em classes abstratas (interfaces), e é feita através de herança.

Neste trabalho é apresentada uma implementação de IEngine sobre o motor Crystal Space, que é *open source* e multiplataforma. É apresentado, também, um caminho para a implementação de IEngine sobre um segundo motor, o Fly3D. Por fim, é estudada a possibilidade de IEngine receber uma camada de interface com outra linguagem (por exemplo, Java ou Lua), de forma a permitir o rápido desenvolvimento de jogos de boa qualidade visual em 3D.

Abstract

3D game engines are APIs that help the development of games with real time 3D graphics. It is convenient to use a 3D game engine when developing games, because all the low-level functionality is already built. However, each 3D engine API is constructed in a different way, causing engines to be incompatible. The learning process is also difficult, because the APIs of some engines are not well designed. This work analyzes this issue by doing a critic survey of some existing 3D engines and proposes an abstraction layer over them, called IEngine.

The IEngine layer gives engines a standard API. An application built over IEngine can be compiled with any of the 3D engines adapted to IEngine, and that application could switch engines without any code change. The adaptation of a new engine to IEngine is also facilitated by IEngine's structure of abstract classes (interfaces) and is done by inheritance.

In this work, an implementation of IEngine over the open-source and multiplatform engine Crystal Space is presented. Also, it is presented a way to implement IEngine over a second engine, Fly3D. Finally, the possibility of IEngine receiving an interface layer with other programming languages such as Java and Lua is studied, in such a way to allow the quick development of good visual quality 3D games.

Índice

Índice de Figuras

Índice de Tabelas

"Peace is a word we teach

A place for us all to reach

Sing as it sings to you

As it sings to me

As I will always need you inside my heart"

Homeworld, do grupo Yes – trilha sonora do jogo de mesmo nome

1. Introdução

Motores de jogos 3D (*3D game engines*) são APIs (*application programming interfaces*) que facilitam o desenvolvimento de jogos que possuem gráficos 3D em tempo real [Watt&Policarpo 00] [Eberly 01]. Os motores permitem que o desenvolvedor de jogos, ao invés de utilizar diretamente uma API gráfica (por exemplo, OpenGL ou DirectX), utilize uma camada de mais alto nível. Os motores implementam algumas características que estão em um nível de abstração mais alto do que o das APIs gráficas, como por exemplo: oclusão por descarte rápido de áreas invisíveis, detecção e tratamento de colisão e leitura de um arquivo de modelo de um determinado formato do disco. Os motores de jogos 3D implementam também algumas características específicas para jogos: efeitos gráficos especiais, como halo em torno de fontes de luz, *lens flare*, neblina, etc. É conveniente utilizar um motor ao desenvolver um jogo 3D, pois o desenvolvedor pode concentrar-se em um nível de abstração mais alto sem se preocupar com os problemas que já são resolvidos pelos motores.

Porém, cada API de motor 3D é construída de uma forma diferente, tornando os motores incompatíveis entre si e o aprendizado muito complicado. Uma solução para isto seria a criação de uma camada de abstração para motores 3D, que permitisse que uma aplicação pudesse usar qualquer motor e que permitisse também um aprendizado mais fácil. A questão que surge é a seguinte: é possível a construção de uma camada de abstração que envolva as características de todos os motores?

Para analisar esta questão, este trabalho faz uma revisão crítica de alguns motores 3D existentes no mercado, tanto *open source* quanto comerciais. Os motores Crystal Space e Fly3D são estudados a fundo. São analisados com menos profundidade os motores Genesis3D e 3DGameStudio. Em seguida, este trabalho propõe uma camada de abstração em C++ para os motores 3D, chamada de IEngine. A camada IEngine tem como objetivos facilitar a programação por meio da adoção de boas técnicas de orientação a objetos e *design patterns* [Gamma 94] e permitir que uma aplicação seja facilmente portada para diversos motores 3D. Para a utilização da camada IEngine sobre um novo motor 3D é necessário adaptá-la, e esta adaptação também é facilitada por sua estruturação em interfaces abstratas e classes concretas.

Este trabalho mostra, também, detalhes sobre a implementação da camada IEngine sobre o motor Crystal Space [Crystal 01] e aborda introdutoriamente a sua implementação sobre o motor Fly3D [Fly3D 01]. Finalmente, testes são feitos a fim de medir o desempenho de uma aplicação que utiliza IEngine sobre Crystal Space, em comparação com uma que utiliza diretamente o motor Crystal Space.

Na conclusão, são feitos comentários sobre a portabilidade de jogos desenvolvidos sobre IEngine e sobre a possibilidade de, no futuro, esta linha de pesquisa gerar um padrão de camada de abstração para motores 3D.

2. Motores 3D

Este capítulo conceitua os motores 3D, e apresenta um breve histórico de sua utilização em jogos na seção 2.1. A seção 2.2 trata da arquitetura dos motores 3D, enfatizando os pontos que normalmente são comuns e os que normalmente diferem entre os vários motores 3D existentes. A seção 2.3 introduz os conceitos específicos dos motores Crystal Space e Fly3D.

2.1. Conceito

A construção de um jogo ou outra aplicação de visualização 3D em tempo real requer o uso de algoritmos eficientes de visualização, oclusão, detecção e tratamento de colisão e outros, de forma que o desempenho seja satisfatório. Em particular, no caso dos jogos, bom desempenho gráfico pode significar a possibilidade de imagens mais detalhados, com uma quantidade maior de polígonos, e também melhoras em outras áreas, como algoritmos mais sofisticados de inteligência artificial ou uma quantidade maior de efeitos especiais. Motores 3D (*3D game engines* ou simplesmente *engines*) são bibliotecas que implementam estes algoritmos e permitem a construção de jogos e aplicações trabalhando-se em alto nível.

Os motores 3D permitem que a camada de software que implementa visualização eficiente fique encapsulada do restante do jogo, tornando-se assim reutilizável. É comum, na indústria, o reuso de motores 3D e até mesmo a

comercialização dos mesmos como bibliotecas de software para o desenvolvimento de jogos.

O primeiro jogo lançado comercialmente com um motor 3D reutilizável foi o jogo Doom. Este jogo rodava em microcomputadores Intel 386 sem recursos de aceleração gráfica e obtinha, apesar disto, um desempenho muito bom para as máquinas da época: entre 10 e 30 quadros por segundo, em uma tela de 320x200 píxeis de resolução.

O motor de Doom baseia-se no algoritmo dos portais, com uma árvore BSP (*binary space partition*) como estrutura de dados que descreve o ambiente a ser visualizado. O motor de Doom possui uma rotina de *rendering* bastante eficiente, escrita em Assembly. Isto era necessário na época, devido à ausência de placas aceleradoras 3D. Em resumo, pode-se afirmar que o motor de Doom faz duas operações:

- limita o mundo a ser visualizado, fazendo oclusão baseada no algoritmo dos portais, descartando rapidamente áreas invisíveis;
- faz o *rendering* do ambiente, texturizando os polígonos na tela da forma mais eficiente possível.

Um motor 3D é necessário mesmo com o uso de placas gráficas que fazem aceleração 3D por hardware, principalmente as placas caseiras (público alvo dos jogos de computador). Estas placas têm um limite de número de polígonos por segundo que não comporta os ambientes mais complexos, como prédios com diversas salas ou ambientes abertos muito grandes, se estes ambientes forem

visualizados completamente, com todos os seus polígonos, a cada quadro. Além disso, há a questão da troca de texturas, operação ineficiente nas placas aceleradoras, que é otimizada pelos motores. Hoje em dia, portanto, motores 3D que se utilizam de placas aceleradoras 3D, mesmo não tratando do *rendering*, que é feito pelo hardware, fazem ainda a operação de oclusão de áreas invisíveis do mundo.

Como forma de melhorar a qualidade visual dos jogos, os motores 3D atuais, além da oclusão, dão suporte a alguns tipos de efeitos especiais. Estes motores possuem ainda algum nível de suporte à rede, para permitir jogos multiusuários, fato este que não é tratado no presente trabalho.

2.2. Arquitetura

Cada um dos diversos motores 3D existentes é projetado e implementado de uma forma diferente. Alguns dos principais aspectos da arquitetura dos motores 3D são apresentados a seguir.

2.2.1. Linguagem utilizada

A maioria dos motores 3D é implementada em C++. A linguagem C++ é suficientemente eficiente e permite que o motor ofereça uma interface orientada a objetos que facilita o desenvolvimento dos jogos. Além disso, a linguagem escolhida para a maioria dos jogos é o C++, tornando simples o interfaceamento

entre o jogo e o motor. Há alguns motores implementados em C. Motores implementados em outras linguagens, até o presente momento, não são utilizados para jogos comerciais.

2.2.2. Laço principal

Todo jogo deve possuir um laço (*loop*) principal de processamento, onde são efetuadas as seguintes ações (não necessariamente nesta ordem):

- Verificação de entrada (teclado, *joystick*, *mouse*) e tomada de ações correspondentes;
- Processamento de inteligência artificial;
- Atualização de posições de objetos de acordo com seus vetores velocidade e aceleração;
- *Rendering* de cena.

Cada volta deste laço faz o rendering de um quadro (*frame*) que é exibido no monitor.

A passagem de controle entre o motor e a aplicação do usuário pode ser implementada de duas maneiras diferentes: por chamada de funções ou por *callback*.

- Chamada de funções: a aplicação chama funções do motor para fazer o *rendering* de cada quadro, a detecção de colisão e todos os outros aspectos

necessários ao jogo. O laço principal está implementado na aplicação, neste caso.

- *Callback*: o motor possui o laço principal, e chama funções da aplicação (registradas previamente como funções de *callback*), que respondem a eventos, tais como: início do processo de *rendering* de um quadro, a movimentação ou o clique do *mouse* e eventos de teclado.

A escolha entre deixar o laço principal no motor ou na aplicação depende de diversos fatores, dentre eles o nível de controle que o jogo deve ter a cada quadro, a funcionalidade que o motor oferece e até mesmo o nível de experiência do desenvolvedor. Para desenvolvedores acostumados com interfaces gráficas orientadas a eventos, é mais fácil utilizar o paradigma de *callback*.

2.2.3. Modelagem do motor

Cada motor pode ser modelado de uma forma diferente, o que faz com que as diferenças entre os diversos motores se acentuem. A maneira de instanciar e controlar os objetos, detectar colisões ou até mesmo controlar a câmera varia muito, pois estas operações estão ligadas diretamente à estrutura interna dos motores. A inexistência de uma API padronizada faz com que cada desenvolvedor de motor siga suas próprias idéias. A câmera é um bom exemplo para ilustrar as diferenças que podem existir entre os diversos motores. Pode-se rotacionar uma câmera das seguintes maneiras: passando para a mesma uma matriz de rotação; ou passando-se um eixo e um ângulo de rotação; ou fixando-se um sistema local de coordenadas e fazendo a rotação em torno de um dos eixos deste sistema local.

Cada um desses métodos é mais adequado a um determinado propósito e menos adequado a outro. Que conjunto de métodos o motor implementa é uma decisão tomada por cada desenvolvedor de motor.

2.3. Crystal Space

Crystal Space [Crystal 01] é um motor 3D *open-source*, desenvolvido em C++. Este motor implementa o algoritmo dos portais para visualização de ambientes externos e um visualizador de terrenos em fase experimental. O Crystal Space tem módulo de *rendering* em software e possui suporte para OpenGL acelerado por hardware.

O Crystal Space começou a ser desenvolvindo antes do desenvolvimento das placas 3D para consumidor caseiro. Portanto, no início de seu desenvolvimento, teve como objetivo o *rendering* por software, tendo sido incluído posteriormente o *rendering* acelerado com OpenGL.

O Crystal Space é multiplataforma, podendo ser compilado em diversos processadores (por exemplo, Intel e Macintosh) e diversos sistemas operacionais (por exemplo, Windows, Linux e MacOS). É um produto desenvolvido sob a forma de *Open Source*, com cerca de 40 desenvolvedores.

Para se começar a modelagem de IEngine sobre Crystal Space, faz-se necessário enfocar os aspectos relevantes deste motor. Crystal Space, em sua versão 0.92, implementa tanto o modelo de laço principal externo quanto o

modelo de *callback*, embora os desenvolvedores estejam abandonando a linha de *callback*.

Outros aspectos importantes do Crystal Space são analisados na seção 3.2.3.

2.4. Fly3D

Fly3D [Fly3D 01], desenvolvido pela empresa brasileira Paralelo Computação, é um motor em C++ que requer uma placa 3D. Fly3D utiliza somente OpenGL para a visualização, o que torna sua utilização sem placa 3D muito ineficiente.

Fly3D é desenvolvido para Windows e está em sua versão 2.0. Fly3D é estruturado baseado no conceito de *frontend* e *plugins*, que serão explicados mais adiante.

Existe uma série de ferramentas para o motor Fly3D, que são distribuídas no próprio pacote do motor que pode ser baixado de seu site [Fly3D 01]. Dentre estas ferramentas, há um compilador de BSP (flyBuilder), um *frontend* para teste de aplicações e ajuste de parâmetros (flyEditor) e um programa para aplicação de texturas em objetos (flyShader).

Ao desenhar o ambiente, Fly3D encapsula completamente a camada gráfica OpenGL. Porém, ao desenhar os objetos (por exemplo, um personagem ou uma nave), o usuário deve escrever alguns comandos OpenGL para configurar as

matrizes de transformação antes de chamar o método *draw* da malha do objeto (`objmesh->draw()`). Isto, embora facilite a programação à primeira vista, é uma inconsistência de encapsulamento. O IEngine, portanto, deve sempre complementar este encapsulamento quando necessário, tornando todos os comandos (especificamente, neste caso, os comandos de desenho de objetos) independentes da camada gráfica utilizada e consistentes com a API IEngine.

As aplicações construídas com Fly3D são formadas por duas partes:

- *frontend* – é a aplicação principal que inicializa o motor e cuida do laço de mensagens do Windows. É nesta parte que fica o laço principal de *rendering*.
- *plugins* – são módulos que são compilados em DLLs e normalmente implementam uma classe de objetos usados no jogo: nave, tiro, explosão, câmera, *power-up*, etc.

Fly3D não implementa um laço principal interno. O laço principal fica no *frontend*. A sequência de passos que o laço do usuário deve fazer para renderizar uma cena é:

- chamar o método `step()` do motor, através da variável global g_flyengine (`g_flyengine->step()`). Este método atualiza o estado de todos os *plugins*, enviando para eles as mensagens necessárias.
- chamar o método `draw_frame()` do motor, através da variável global g_flyengine (`g_flyengine->draw_frame()`). Este método faz com

que o motor desenhe a cena. Mas a cena é desenhada indiretamente. O motor, na realidade, envia uma mensagem `FLY_MESSAGE_DRAWSCENE` para todos os *plugins*. Um deles deve, ao receber esta mensagem, desenhar a cena. E este desenho é feito chamando-se novamente um método do motor, `g_flyengine->draw_bsp()`.

Esta sequência de passos deve ficar em um laço no *frontend*. Observa-se que, no momento de desenhar uma cena, o controle passa do *frontend* para o motor, do motor para um *plugin*, e deste *plugin* de volta para o motor. Isto permite uma grande flexibilidade, tornando possível inclusive o rendering de múltiplos pontos de vista em áreas separadas da tela, apesar de tornar um pouco mais complicada a programação, principalmente para usuários inexperientes. Sendo um dos objetivos do IEngine facilitar a programação, o IEngine opta pela simplificação dos aspectos acima. Quando é necessário tomar uma decisão entre simplificar o uso ou dar poder ao programador, o IEngine opta por simplificar o uso. Como exemplo, na versão atual do IEngine, apenas um ponto de vista preenchendo toda a tela é permitido. Esta decisão reduz o poder de expressão do IEngine, mas facilita a programação.

2.5. Outros motores

Existem outros motores com características interessantes. Dois deles são o Genesis3D, motor *open source* escrito em C mas com uma organização semelhante à orientada a objetos, e o 3D Game Studio, motor comercial que

permite que aplicações sejam escritas em vários níveis de abstração: desde um nível alto utilizando-se objetos e comportamentos pré-definidos, sem a necessidade de programar, até um nível mais baixo utilizando-se uma API em C++. Estes dois motores são analisados a seguir.

2.5.1. Genesis3D

O motor Genesis3D [Genesis3D 01] foi desenvolvido pela empresa Eclipse Entertainment. A partir do ano de 1999, seu desenvolvimento passou a ser gerenciado pela comunidade *open source*, e o motor passou a ser distribuído gratuitamente sob este tipo de licença. O motor está em sua versão 1.1.

O motor Genesis3D foi desenvolvido para Windows, utilizando a linguagem C e a API DirectX. A modelagem do Genesis3D seguiu certas regras para trazer à sua API características de orientação a objetos, apesar da utilização da linguagem C. As funções foram divididas em grupos que tratam de "objetos" específicos. Desta forma, as funções são equivalentes a métodos aplicados aos objetos.

2.5.2. 3D Game Studio

O 3D Game Studio [GameStudio 02] é uma plataforma comercial de desenvolvimento de jogos para Windows. Possui editores de níveis, de modelos e de terrenos. Para a visualização em tempo real, utiliza um motor chamado A5.

O motor 3D Game Studio foi desenvolvido para permitir a criação de aplicações em três níveis de abstração:

- num nível mais alto, de dentro do editor de níveis, pode-se construir jogos simples com comportamentos pré-programados, apenas com cliques do mouse;
- num nível intermediário, pode-se programar o comportamento dos objetos com uma linguagem de *script* própria, semelhante a C;
- num nível mais baixo, pode-se utilizar o motor como uma API em C++.

2.6. Análise comparativa

Os motores 3D analisados possuem algumas características em comum e algumas diferenças. A plataforma utilizada geralmente é o Windows, por ser a plataforma mais comum para os jogos, que têm como público alvo os usuários caseiros. A linguagem é geralmente o C ou o C++, que são as mais utilizadas no desenvolvimento de jogos de grande porte. Não existem motores equivalente a estes (que implementam as principais funcionalidades) que suportem a linguagem Java. O motor Crystal Space permite o acesso a algumas de suas funções de baixo nível através da linguagem Lua, porém este acesso se dá de uma forma bastante específica ao Crystal Space.

Alguns motores comunicam-se com o hardware gráfico através do padrão OpenGL, outros utilizam o DirectX, da Microsoft. Esta questão, embora

polêmica, não deveria fazer sentido quando estamos tratando de uma camada de abstração mais elevada que a dos padrões de renderização 3D como OpenGL e DirectX. Estes padrões deveriam estar encapsulados e a aplicação deveria comunicar-se somente com o motor. Porém, em alguns casos, o desenvolvedor do motor incorpora, como parte do mecanismo de uso do motor, o uso da camada gráfica abaixo do mesmo, deixando-a não totalmente encapsulada. É o caso do motor Fly3D: para se fazer o desenho de um obeto, deve-se posicionar as matrizes de visualização de modelo fazendo-se chamadas ao OpenGL, para que o objeto seja desenhado na posição correta.

A Tabela 1 faz um resumo comparativo entre os motores estudados.

Motor	**Plataforma**	**Linguagem**	**Licença**	**Renderização**
Crystal Space	Multi	C++	Open Source	OpenGL; Software
Fly3D	Windows	C++	Uso livre; não é open source	OpenGL
Genesis3D	Windows	C	Open Source	DirectX; Software
3D Game Studio	Windows	C++	Comercial	DirectX; Software

Tabela 1 – Comparação entre alguns motores 3D

3. IEngine

Este capítulo apresenta a camada IEngine. A seção 3.1 trata do assunto de padronização de uma API para motores 3D, traçando um panorama atual sobre os motores 3D e mostrando os motivos que levam a esta padronização. A seção 3.2 mostra a modelagem de IEngine, mostrando também como pode ser feita a adaptação do IEngine a um motor existente, com os exemplos de IEngine-Crystal e IEngine-Fly.

3.1. Padronização de API para motores 3D

O enfoque dos motores 3D tende a se alterar gradativamente, devido à evolução da tecnologia e ao seu barateamento e acessibilidade pelo público consumidor. Há algum tempo, os motores 3D eram responsáveis por fazer o *rendering* texturizado, além da oclusão, transformação e iluminação dos polígonos. Estes primeiros motores possuíam várias rotinas escritas em Assembly para que o desempenho fosse satisfatório nos processadores existentes então.

Com o surgimento das placas 3D, os motores passaram a utilizá-las, e livravam-se da etapa de *rendering*, fazendo apenas oclusão, transformação e iluminação. Estes motores mantinham seu suporte ao *rendering*, para o caso do usuário não ter uma placa 3D, mas utilizavam-se dos recursos da placa 3D quando presente, melhorando a qualidade da imagem (devido à interpolação por

hardware que a placa faz na etapa de *rendering* e ao processo de *mipmapping*) e aumentando a taxa de quadros por segundo.

Surgiram então placas 3D de segunda geração, capazes de fazer transformação e iluminação por *hardware*. O número de polígonos por segundo que estas placas conseguem gerar vem aumentando rapidamente. Hoje em dia, são comuns placas 3D caseiras capazes de transformar, iluminar e renderizar 10 milhões de polígonos texturizados por segundo. E, paralelamente a isto, as CPUs dos microcomputadores caseiros também sofrem uma rápida evolução.

Esta evolução de *hardware* muda o enfoque dos motores 3D. Um exemplo é a etapa de oclusão. Com a melhora de desempenho das placas 3D, algoritmos mais simples de teste de oclusão podem ser usados, algoritmos estes que causam mais redesenho, mas que por outro lado consomem muito menos CPU, liberando-a para outras tarefas importantes nos jogos, como inteligência artificial. Os motores mais novos, portanto, são mais adequados para *hardwares* mais novos.

Não existe uma padronização nas APIs dos motores 3D. Portanto, os novos motores são construídos com uma API diferente dos motores antigos. Como consequência, jogos que foram construídos sobre os motores antigos não podem ser portados para motores mais novos. Os jogos e a API desenvolvida para cada motor tornam-se “descartáveis”, o que não aconteceria se esta API fosse padronizada. A padronização torna possível a evolução dos jogos juntamente com o avanço da tecnologia, sendo possível a simples troca do motor que está sob um jogo já desenvolvido, sem a necessidade de muitas adaptações.

Podemos citar como exemplo disto o jogo Doom. Este jogo utiliza baixa resolução (320x240) e não implementa alguns efeitos especiais hoje comuns, como brilho em luzes. Caso ele tivesse sido escrito sobre um motor padronizado, ele poderia ter sido facilmente portado para um motor mais moderno, que trabalha em resolução mais alta, utiliza a aceleração disponível em hardware e implementa o efeito de brilho de luzes. Teríamos então um Doom modernizado, o que prolongaria a vida útil do produto. Sendo o jogo e o ambiente descritos em alto nível e o motor tratando do baixo nível, novos motores podem explorar os novos recursos de hardware para fazer a visualização do que está expresso no nível mais acima. Em outras palavras, tudo que a API padronizada pode expressar evolui junto com o *hardware* à medida que aparecem motores novos.

A padronização da API pode ser feita como uma camada acima dos motores 3D. Uma camada abstrata modela a API e, abaixo desta, uma camada específica deve ser construída para cada motor, para adaptar a API ao motor utilizado. A camada abstrata que modela a API, no caso da utilização de C++, pode ser implementada como um conjunto de classes virtuais. A camada abaixo dela é composta de subclasses das classes virtuais, que implementam estas classes de forma específica para cada motor.

As camadas concretas podem ser construídas para adaptar a API a motores existentes, mas nada impede que novos motores venham a ser construídos já atendendo à API desde o início.

Programadores podem precisar programar jogos em outras linguagens, ou querer utilizá-las para controlar um motor (na verdade, para controlar um ambiente 3D). Uma linguagem que pode ser considerada de um nível de abstração mais alto que C++, por sua ausência do uso explícito de ponteiros e tipagem forte é a linguagem Java. Ao se pensar em programação 3D em Java, existe a tentação de usar o pacote Java3D [Bouvier 99]. Porém o Java3D não é um motor 3D. Ele implementa grafo de cena e permite visualização acoplada a OpenGL e DirectX, mas não possui tratamento eficiente de oclusão e colisão tal como os motores de jogos 3D. Portanto, seria interessante a utilização de motores de jogos 3D de dentro do ambiente Java a partir de uma interface padronizada.

Um resumo das vantagens que a padronização da API oferece pode ser o seguinte:

- Portabilidade dos jogos. Os jogos podem ser adaptados para funcionarem com outros motores 3D.
- Simplificação do interfaceamento com outras linguagens. Uma interface com uma outra linguagem construída em cima da API padronizada torna automaticamente disponível para esta linguagem todos os motores já adaptados à API ou construídos especificamente para a mesma. Podemos ter, por exemplo, uma interface com Java através de JNI [JNI 01], disponibilizando para programadores da linguagem Java todos os motores sob a API padronizada. Uma maneira de se interfacear C++ com Java é descrita em [Carasso&Staa 01].

- Facilidade de aprendizagem. O programador terá que aprender uma só API e poderá ter controle sobre diversos motores. A construção de uma API moderna, orientada a objetos e seguindo conceitos bem estruturados de *design patterns* contribui para a facilidade de aprendizagem.

Uma desvantagem da utilização de uma API padronizda é a potencial redução do poder de expressão do motor, uma vez que esta API provavelmente não deve abrangir todas as funcionalidades de todos os motores.

3.2. Modelagem

Esta seção apresenta a modelagem do IEngine. Na seção 3.2.1 é apresentada a técnica de modelagem e a implementação que foi seguida. Em seguida, é mostrada a camada comum do IEngine (seção 3.2.2 – IEngine), e posteriormente a camada de adaptação de IEngine para os motores específicos (seções 3.2.3 – IEngine-Crystal e 3.2.4 – IEngine-Fly). Por último, na seção 3.2.5 é mostrada a camada de aplicação mínima, que conceitualmente está acima da camada de interface e cujo propósito é permitir ao programador escrever rapidamente uma aplicação com comportamento padrão.

3.2.1. Técnica de modelagem e implementação

IEngine é uma camada que encapsula um motor 3D. Esta camada implementa uma API padronizada, de forma a tornar as aplicações independentes do motor utilizado.

IEngine é projetado de forma a permitir a troca de motor sem que seja necessário fazer nenhuma alteração nas aplicações já construídas. Para isto, dois tipos de classes distintos são definidos:

- Interfaces – são classes com funções virtuais, que serão usadas por programas-clientes para comunicação com os diversos módulos do motor 3D (rendering, câmera, eventos, etc.);
- Classes concretas – são classes que especializam as interfaces, implementando as funcionalidades e a comunicação com um determinado motor 3D (por exemplo, o motor Crystal Space).

Doravante, será adotada a convenção Java de classes e interfaces: uma *interface* é uma classe virtual que é uma interface com a aplicação e uma *classe* é uma classe concreta que é uma implementação de uma interface.

Neste trabalho, chama-se de "instância de IEngine" um conjunto de classes concretas que são filhas das classes de interface de IEngine, e que implementam todas as funcionalidades do IEngine utilizando um motor específico. A aplicação do usuário deve se comunicar apenas através das interfaces, que são as mesmas para qualquer motor 3D que venha a ser implementado sob o IEngine. As

interfaces consistem de classes com funções virtuais. As classes concretas (instâncias de IEngine) devem ser implementadas para cada motor específico, tendo-se desta forma um "driver" para o motor utilizado. As mesmas interfaces são implementadas de forma específica para cada motor, mas como as interfaces não mudam, uma aplicação que as utiliza fica independente de motor.

Uma instância do IEngine deve incluir as interfaces e também a camada que faz a ligação com um motor específico. Como exemplo, uma versão do IEngine escrita sobre o motor Crystal Space é chamada IEngine-Crystal e está estruturada conforme a Figura 1.

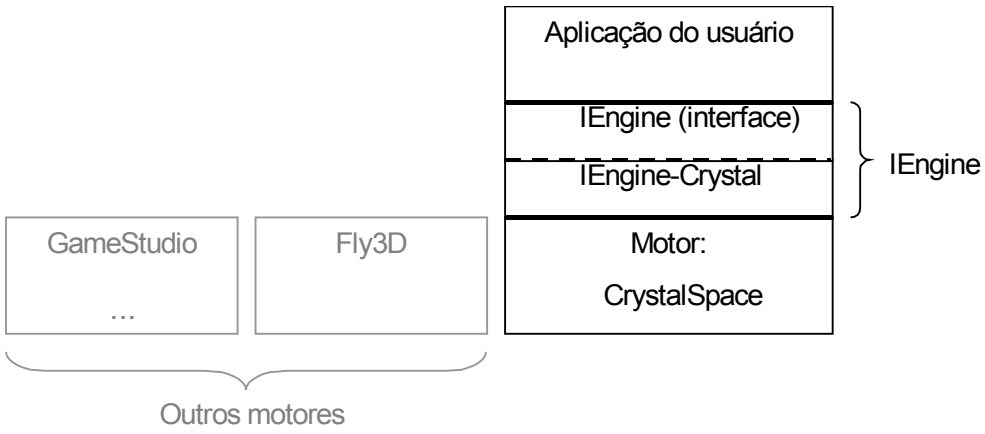


Figura 1 – Estrutura de Camadas do IEngine

A modelagem de classes da interface IEngine é definida como na Figura 2. A Figura 3 mostra a notação utilizada, baseada em [Staa 00].

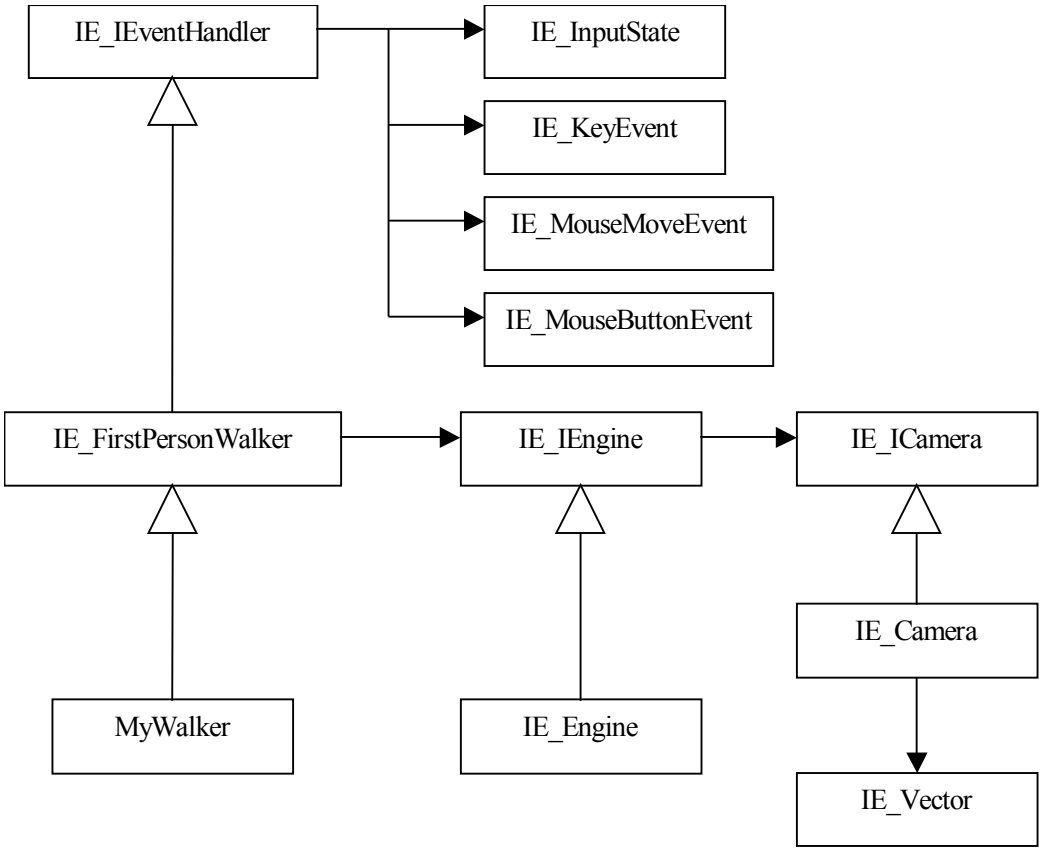


Figura 2 – Modelagem de classes da interface IEngine

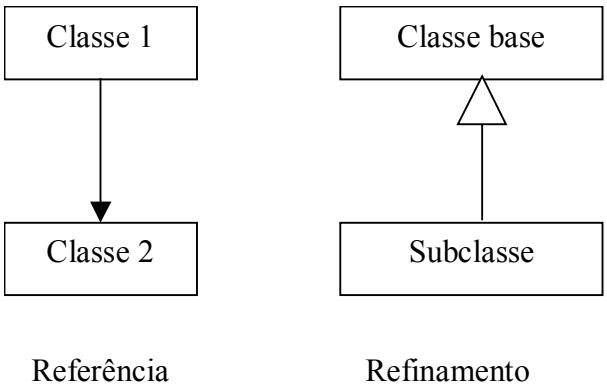


Figura 3 – Notação usada na modelagem de classes

A convenção para os nomes de classes é a seguinte:

- Todas as classes começam com IE_ (abreviatura de IEngine)
- As interfaces possuem a letra I após este prefixo.
- Em seguida, tem-se o nome da classe, com a primeira letra maiúscula e as demais minúsculas (isto torna possível a diferenciação entre uma interface e uma classe cujo nome começa por I).
- Para nomes de classes (e também para todos os objetos, variáveis e comentários do programa) é usada a língua inglesa.

Por exemplo, o nome da interface para uma câmera é: `IE_ICamera`. O nome da classe concreta que herda de `IE_ICamera` e implementa uma câmera é: `IE_Camera`.

Cada instanciação do IEngine sobre um motor deve adaptar toda a estrutura do motor que está abaixo dele para sua própria estrutura. Um caso comum de adaptação é o *callback*. O IEngine está modelado de forma a permitir o uso de *callback*, o que facilita a programação. Se o motor não implementar *callback*, o IEngine construído sobre este motor deve implementá-lo. No motor Fly3D, por exemplo, o usuário deve construir o laço principal e neste laço chamar os métodos `step()` (que atualiza o estado do motor para o próximo quadro) e `draw()` (que desenha um quadro na tela). Portanto, IEngine-Fly deve implementar o laço principal, chamar os métodos `step()` e `draw()` de Fly3D, tratar os eventos de teclado e *mouse* e chamar as funções de *callback* do usuário quando necessário. Isto acontece também com o motor CrystalSpace versão 0.9x, embora uma versão

anterior do Crystal Space implementasse um laço interno e funções de *callback* (esta linha foi abandonada pelos desenvolvedores do Crystal Space).

3.2.2. Camada de interface de IEngine

A camada superior do IEngine é a camada que contém as interfaces. As interfaces assumem dois papéis:

- são classes básicas para a estruturação de qualquer implementação específica do IEngine;
- são os pontos de comunicação do IEngine com a aplicação do usuário.

Esta seção apresenta um detalhamento da camada de interface do IEngine que pode ser lida com eventuais consultas ao código [IEngine 02], caso o leitor esteja interessado em se aprofundar no estudo do IEngine. Numa primeira leitura desta dissertação, o leitor pode se concentrar apenas nas simplificações e restrições da versão atual do IEngine mencionadas ao longo desta seção.

3.2.2.1. IE_IEngine

Esta interface modela o motor 3D utilizado. Através dela, os ambientes virtuais são carregados na memória, inicializados e o laço principal é chamado. Esta interface possui também uma câmera (classe `IE_ICamera`, seção 3.2.2.3). Seus principais métodos são os seguintes:

- `initialize()` – inicializa o motor, fazendo todas as preparações necessárias para o *rendering* em tempo real;
- `loadWorld(char *filename)` – carrega um ambiente virtual do disco, em um formato que é dependente do motor que está sendo utilizado;
- `setEventHandler(IE_IEventHandler *eventHandler)` – cadastra, no IEngine, o tratador de eventos a ser utilizado, que é um objeto da classe `IE_IEventHandler`;
- `loop()` – chama o laço principal de *rendering*. O motor então chama o tratador de eventos caso algum evento ocorra.

3.2.2.2. IE_EngineFactory

Esta classe é a maneira de o programa obter uma instância concreta de `IE_Engine`, moldada (*cast*) à sua interface. É um caso do *design pattern* denominado *Abstract Factory*. Seu papel é criar um objeto `IE_Engine` (do motor específico que está sendo utilizado), moldá-lo à interface `IE_IEngine` e retorná-lo através de seu único método, `createEngine(HINSTANCE hApp)`. Este parâmetro `HINSTANCE hApp` é um objeto existente no ambiente Windows, que será usado para permitir a criação de uma janela para o motor ou a ativação de tela-cheia. Este, atualmente, é o único ponto de dependência do IEngine com o ambiente Windows e foi implementado desta forma por questões de simplicidade.

A classe `IE_EngineFactory` deve saber sobre o motor que está sendo utilizado, portanto seu nome não recebe um I. Ela não é uma interface. Apesar disso, ela deve seguir este padrão, tendo sempre este nome e o método `createEngine`. Em outras palavras, o arquivo ".h" desta classe não muda, qualquer que seja a instância de IEngine utilizada. Desta forma, a questão de portabilidade entre diversos motores estará resolvida. Esta classe está sendo descrita aqui neste item (camada de interface) porque ela deve ser considerada como tal pelo programador, apesar de ter seu código dependente do motor.

3.2.2.3. IE_ICamera

Esta interface representa uma câmera, que pode sofrer translação e rotação de forma acumulativa através de seus métodos `move(double x, double y, double z)` e `turn(IE_Vector direction, double radians)`.

O desenvolvedor do jogo deve obter a câmera associada ao motor para poder movimentá-la. Isto é feito da seguinte forma:

- A classe concreta que representa o motor, `IE_Engine`, sempre contém uma câmera. Esta câmera é usada pelo motor como ponto de referência para o desenho do ambiente.
- Existe um método em `IE_IEngine` (e também, obrigatoriamente, em `IE_Engine`) que retorna um ponteiro para câmera. Este método, `getCamera()`, deve ser usado pelo desenvolvedor do jogo para obter o

ponteiro para a câmera e, através dele, movimentar a câmera. As movimentações são imediatamente "sentidas" pelo motor, uma vez que se trata de um ponteiro.

Nesta implementação de IE_Engine, somente uma câmera pode ser utilizada, a cada quadro, para fazer o *rendering* da cena, não sendo possível a utilização de múltiplos pontos de vista simultâneos na tela.

3.2.2.4. IE_IEventHandler

Esta interface representa um tratador de eventos. Será o ponto de contato entre o IEngine e o programa do usuário, no que se refere à passagem de eventos. Esta interface não deve ser refinada por uma instância de IEngine, e sim pelo usuário que constrói uma aplicação.

O usuário deve criar uma subclasse de IE_IEventHandler, implementando todos os seus métodos virtuais, e deve informar a IEngine que esta classe é o tratador de eventos que será utilizado. Isto é feito por meio do método `setEventHandler(IE_IEventHandler *eventHandler)` de `IE_IEngine`.

A partir do momento que a classe do usuário é registrada como tratadora de eventos, o IEngine chama diversos métodos desta classe quando estes eventos ocorrem, e passa um parâmetro a estes métodos. Os parâmetros passados também são classes definidas em IEngine: `IE_KeyEvent` (um evento de teclado),

`IE_MouseMoveEvent` (um evento de movimento de *mouse*) e `IE_MouseButtonEvent` (um evento de botão de *mouse*). Esta é a estratégia adotada nas interfaces orientadas a eventos AWT e Swing, em Java [Sun 02].

Além dos métodos de tratamento de eventos de teclado e *mouse*, existe um outro método importante que também é frequentemente chamado. Este método é o `step`, que será visto mais adiante.

Os métodos de teclado e *mouse* são descritos a seguir.

- `handleKeyDown` – é chamado quando uma tecla é pressionada no teclado. O parâmetro passado a este método indica que tecla foi pressionada.
- `handleKeyUp` – é chamado quando uma tecla é solta no teclado. O parâmetro passado a este método indica que tecla foi solta.
- `handleMouseMove` – é chamado quando o *mouse* é movido. O parâmetro passado a este método indica qual a distância percorrida nos eixos x e y.
- `handleMouseButtonDown` – é chamado quando um botão do *mouse* é pressionado. O parâmetro passado a este método indica qual botão foi pressionado.
- `handleMouseButtonUp` – é chamado quando um botão do *mouse* é solto. O parâmetro passado a este método indica qual botão foi solto.

O outro método de tratamento de eventos, `step`, é chamado antes que cada quadro seja apresentado na tela. Este evento deve ser usado para atualizar o estado da simulação: mover os personagens, atualizar o estado dos mecanismos de inteligência artificial (se necessário) e fazer qualquer outro processamento que deve ser feito a cada quadro durante o jogo. Em outras palavras, este é o método onde fica toda a lógica principal do jogo (a menos do tratamento de entrada por teclado e *mouse*). Portanto, pode-se considerar que a classe que o usuário constrói refinando `IE_IEventHandler` é a classe principal da sua aplicação.

3.2.3. IEngine-Crystal

Esta camada é a camada de adaptação do IEngine para o motor Crystal Space. É uma instância de IEngine. É composta de classes que refinam as interfaces de IEngine e outras classes auxiliares.

Outras instâncias de IEngine, como IEngine-Fly, têm características em comum com IEngine-Crystal. Os nomes das classes devem ser os mesmos, bem como a funcionalidade de cada método de cada classe. Apesar de estarem sendo usados os mesmos nomes de classes para todas as instâncias de IEngine, não ocorrem conflitos, pois um motor deve ser utilizado sozinho. Na prática, cada instância de IEngine é compilada em uma DLL ou LIB. Uma delas deve ser ligada à aplicação que o usuário está construindo, tendo-se desta forma o acesso a este motor que é encapsulado por esta instância de IEngine.

As classes de IEngine-Crystal são descritas a seguir. Valem aqui as mesmas observações feitas ao leitor no início da seção 3.2.2.

3.2.3.1. IE_Engine

Esta classe encapsula o motor utilizado. Ela é um refinamento de `IE_IEngine`, devendo implementar todos os seus métodos virtuais. É nesta classe que fica o código para:

- inicializar o motor – método `initialize`;
- carregar um mundo do disco – método `loadWorld`;
- cadastrar uma classe como sendo tratadora de eventos – método `setEventHandler`;
- transferir o controle para o loop principal de *rendering* – método `loop`.

Os arquivos de mundo, que contêm a geometria a ser visualizada, têm formatos específicos para cada motor. Apesar disto, pode existir um nível superior de compatibilidade no software de modelagem utilizado, o que permite que um mesmo mundo seja exportado para diversos motores, mantendo-se então o aspecto de portabilidade entre motores do IEngine. Este assunto é discutido mais detalhadamente na seção 5.3.

3.2.3.2. IE_Camera

Esta classe é o refinamento de `IE_ICamera`. Deve implementar os métodos `move` e `turn`, utilizando, no caso, comandos apropriados de Crystal Space.

3.2.3.3. IE_CrystalEventHandler

Esta classe é um tratador de eventos utilizado internamente por Crystal Space. Não deve ser confundida com a classe `IE_IEventHandler`; não há nenhuma relação entre elas. Enquanto que `IE_IEventHandler` deve ser refinada pelo usuário que está construindo uma aplicação para que ele possa tratar os eventos de teclado e mouse, `IE_CrystalEventHandler` é um refinamento da classe `iEventHandler`, uma classe de Crystal Space (que por sinal deve ficar encapsulada da camada superior do IEngine). Em outras palavras, enquanto `IE_IEventHandler` é o ponto de contato de IEngine com o mundo exterior, `IE_CrystalEventHandler` é o ponto de contato de Crystal Space com IEngine.

IE_CrystalEventHandler é uma classe construída para que os eventos de teclado e *mouse* possam ser passados do Crystal Space para o IEngine. Uma vez no IEngine, estes eventos têm seu formato convertido para um formato padrão e são passados para a camada superior da aplicação. Não é obrigatório o uso do motor para a captura de eventos de teclado e mouse, como foi feito com IEngine-

Crystal. Caso algum motor não possua uma maneira de tratar eventos de teclado e *mouse*, a instância de IEngine deve implementar esta captura utilizando funções do sistema operacional utilizado, converter os eventos para o formato padrão e passá-los para a camada superior, mantendo assim o encapsulamento.

3.2.4. IEngine-Fly

O motor Fly3D é conceitualmente diferente do motor Crystal Space. Conforme é visto na seção 2.4, o Fly3D trabalha com o conceito de *plug-ins* e *frontend*. Ao se escrever um programa utilizando puramente o Fly3D, o programador deve escrever:

- o *frontend*, que inicializa a janela gráfica e o motor e que contém o laço principal de mensagens do Windows;
- os *plug-ins*, que são DLLs que implementam o comportamento, a aparência e a inteligência de cada objeto do jogo (personagens, naves, tiros, etc).

Em geral, o *frontend* é simples e é sempre o mesmo. Existe um *frontend* mínimo no pacote do Fly3D, que faz as inicializações necessárias e possui um laço principal básico. É possível escrever um jogo utilizando-se este *frontend* e colocando-se toda a inteligência nos objetos. No IEngine, quem faz o papel do *frontend* é a classe `IE_Engine`, pois é esta classe que faz a inicialização do motor e possui o laço principal.

Quanto aos *plug*-ins, o conceito equivalente utilizado no IEngine é o conceito de herança de classes. Enquanto que programando-se no Fly3D pensa-se em *plug-ins*, programando-se no IEngine pensa-se em classes que implementam os objetos. Portanto, devem haver classes que implementam os principais *plug-ins*. Na versão atual do IEngine, que permite apenas uma caminhada de uma câmera por um ambiente virtual, existe um único *plug-in*: a câmera. Este *plug-in* de câmera fica encapsulado pela classe `IE_Camera` no IEngine e esta classe segue a padronização da interface `IE_ICamera`.

Desta forma, a classe `IE_Engine` abstrai os conceitos de *frontend* e a classe `IE_Camera` abstrai o conceito de *plug-in* de câmera do Fly3D. Para se incluir o suporte mais amplo que é planejado como um dos trabalhos futuros com o IEngine, deve-se escrever este suporte como classes que encapsulam *plug-ins*. Uma idéia interessante é incluir uma classe de objeto genérico na padronização do IEngine. Esta classe, no caso da instância Fly3D do IEngine, pode encapsular um *plug-in* genérico que permite ao programador criar qualquer tipo de objeto no jogo (vide seção 5.4 – Trabalhos Futuros).

A figura a seguir mostra o diagrama de camadas de IEngine-Fly3D. Pode-se notar o *plug-in* de câmera que está completamente encapsulado por IEngine-Fly3D.

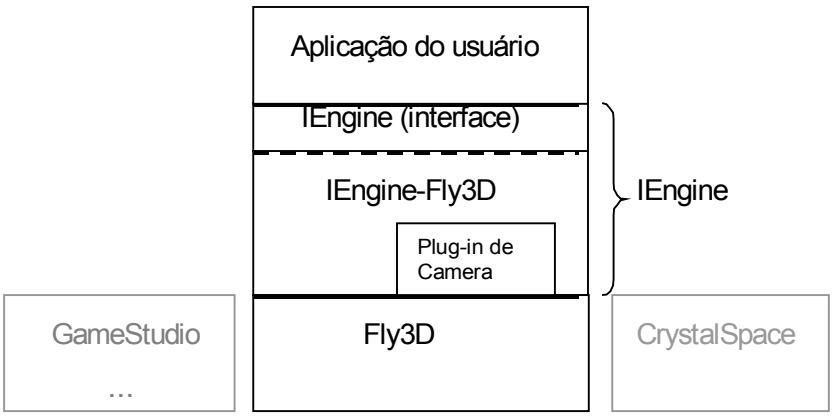


Figura 4 – Estrutura de Camadas do IEngine-Fly3D

3.2.5. Camada de aplicação mínima

O IEngine está construído de forma a permitir que uma aplicação mínima seja escrita com facilidade. Para isto, existe a camada de aplicação mínima, que está acima da camada de interface mas que deve fazer parte da distribuição padrão do IEngine. Seu principal objetivo é facilitar a programação, permitindo que o programador possa desenvolver uma aplicação simplesmente herdando de uma classe que implementa um comportamente padrão. É possível escrever uma aplicação sem se utilizar desta camada de aplicação mínima.

Existem alguns tipos padrão de jogos 3D. Por exemplo, jogos de personagens que caminham em primeira pessoa, jogos de personagens que caminham em terceira pessoa e jogos de naves. O objetivo desta classe é implementar estes tipos principais de jogos em classes que possam ser estendidas pelo programador, de forma que ele tenha todo o comportamento padrão (caminhada, vôo) pré-definido e possa sobreescrever métodos conforme

necessário. A única classe que está implementada no presente momento é a classe `IE_FirstPersonWalker`, que implementa uma caminhada em primeira pessoa.

A classe `IE_FirstPersonWalker` é filha de IE_IEventHandler, fazendo o papel tanto de classe inicializadora do motor como de classe que trata os eventos. Ela possui os seguintes métdos:

- `IE_FirstPersonWalker(HINSTANCE hApp)` – Construtor. Inicializa o motor e abre um arquivo de *log*, para fins de depuração do programa.
- `~IE_FirstPersonWalker()` – Destrutor. Destrói o objeto `IE_Engine`.
- `initializeEngine()` – Chama o método de inicialização de `IE_Engine` e configura a si própria como tratadora de eventos.
- `loadWorld(char *filename)` – Chama o método `loadWorld` de `IE_Engine` para carregar um mundo de um arquivo em disco.
- `loop()` – Chama o loop principal de `IE_Engine`.
- `step(long miliseconds, const IE_InputState& inputState)` – Método de *callback* que é chamado a cada quadro. É este método que faz funcionar a caminhada em primeira pessoa, por meio do tratamento de eventos de teclado.

- métodos `handle...` – métodos de tratamento de mouse e teclado.

Não são usados no presente momento. A movimentação é feita pelo teclado, utilizando-se não os eventos, mas o parâmetro `IE_InputState` passado a `step`.

4. Exemplo e Testes

Neste capítulo é descrita uma aplicação-exemplo e são mostrados resultados de testes de desempenho feitos com esta aplicação. Na seção 4.1 é apresentada a aplicação de testes MyWalker. Na seção 4.2 é feito um teste comparativo de desempenho entre a aplicação MyWalker, que utiliza a camanda IEngine sobre o motor Crystal Space, e uma aplicação semelhante que utiliza diretamente o motor Crystal Space.

4.1. Aplicação exemplo: MyWalker

Para exemplificar a utilização do IEngine, existe uma aplicação de teste, chamada MyWalker. Esta é a aplicação mais simples possível. Ela foi implementada como uma classe que estende a classe `IE_FirstPersonWalker` e utiliza o comportamento padrão desta classe, isto é, não sobreescreve nenhum de seus métodos, utilizando a caminhada padrão que esta classe permite fazer pelo mundo.

A construção de uma aplicação de caminhada desta forma é extremamente simples para o desenvolvedor. Em uma aplicação para Windows (deve-se lembrar que, devido ao exposto na seção 3.2.2.2, ainda existe uma dependência de IEngine com o ambiente Windows), os passos a serem seguidos são os seguintes:

- Deve ser criada uma classe (`MyWalker`) que estende `IE_FirstPersonWalker` e cujo construtor chama o construtor da classe mãe.
- Deve ser construído, então, o método `WinMain`, que é o método principal para aplicações Windows. Neste método, um objeto da classe `MyWalker` deve ser instanciado. Depois deve-se chamar o método `loadWorld` deste objeto e então o método `loop`.

Com estes passos, o método `loop` de `MyWalker` (herdado de `IE_FirstPersonWalker`) assume o controle e faz a movimentação interativa da câmera pelo mundo.

Na Listagem 1 é mostrado o código fonte de MyWalker.h e MyWalker.cpp, para ilustrar a simplicidade com que uma aplicação com comportamento padrão pode ser escrita.

```
// MyWalker.h: interface for the MyWalker class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_MYWALKER_H)
#define AFX_MYWALKER_H

#include "../iengine/iengine.h"

/**
A test application. Has minimum functionality. Leaves everything
to the default inherited from IE_FirstPersonWalker.
*/
class MyWalker : public IE_FirstPersonWalker
{
public:
     //  MyWalker's  constructor  only  calls  its  parent's
constructor.
     MyWalker(HINSTANCE hApp) : IE_FirstPersonWalker(hApp) {};
};

#endif // !defined(AFX_MYWALKER_H)
```

```
// MyWalker.cpp: implementation of the MyWalker class.
//
//////////////////////////////////////////////////////////////////////

#include "MyWalker.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>


/**
Main method. Instantiates a MyWalker class, loads a world
and enters the main loop.
*/
int WINAPI WinMain (HINSTANCE hApp, HINSTANCE prev, LPSTR
cmd, int show)
{
     // Instantiates MyWalker class and starts the engine
     MyWalker* myWalker = new MyWalker(hApp);

     // Loads a map from disk
     myWalker->loadWorld("world");

     // Starts the main loop. myWalker.handleEvent will be
called
     // on every event.
     myWalker->loop();

     // Cleans up
     delete myWalker;

     return 0;
}
```

Listagem 1 – MyWalker.h e MyWalker.cpp: aplicação de teste de IEngine

4.2. Testes

A preocupação com o desempenho reside no fato de que estamos introduzindo uma camada extra, que implica em chamadas extras de métodos. É necessário testar, então, o impacto que a introdução do IEngine causa no desempenho do motor.

A fim de comparar o desempenho de uma aplicação com o IEngine e uma aplicação sem o IEngine, foram utilizadas:

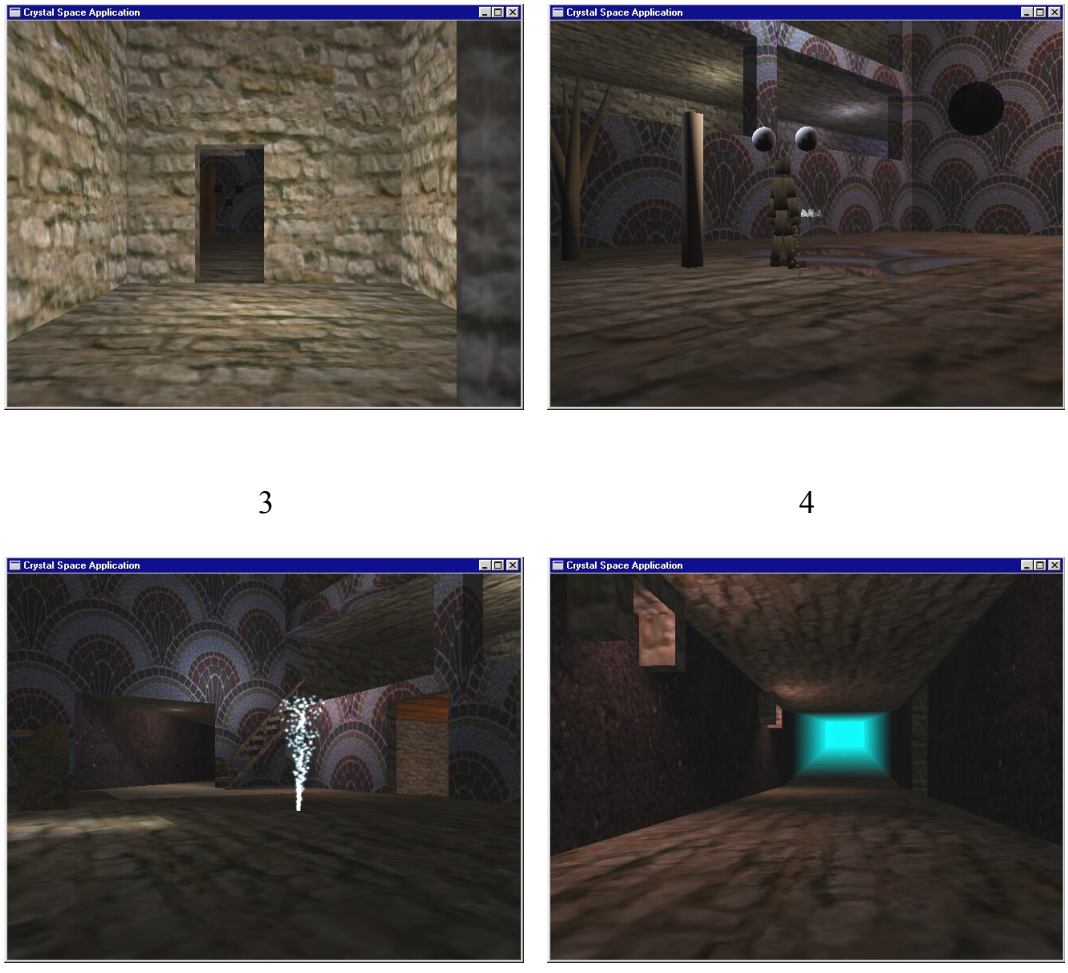
- *MyWalker* - a aplicação de testes do IEngine, que utiliza o arquivo de mundo "flarge" do Crystal Space;
- *simpmap* - uma aplicação de testes que vem com o motor Crystal Space, que utiliza o mesmo arquivo de mundo "flarge".

A aplicação *simpmap* tem o mesmo comportamento que a implementação atual da aplicação MyWalker sobre o IEngine: permite uma caminhada por um mundo carregado do disco, sem se preocupar com detecção de colisão. Esta aplicação é equivalente à aplicação MyWalker sem a camada IEngine. A única alteração que *simpmap* sofreu foi a inclusão de um arquivo de *log* para registrar a informação de tempo decorrido entre cada quadro.

Os testes foram feitos medindo-se o número de milissegundos entre cada quadro em quatro posições diferentes do mundo. Estas quatro posições estão

ilustradas na figura abaixo. Foi utilizada renderização com OpenGL acelerada por *hardware*. Isto faz com que o tempo de renderização de cada quadro seja menor e, conseqüentemente, a influência da camada IEngine no tempo total de processamento seja maior. Um total de aproximadamente 500 amostras seqüenciais de tempo decorrido entre quadros foi considerada em todos os casos.

1 2



3 4

Figura 5 – Posições de câmera utilizadas para teste de desempenho

Os resultados do teste de desempenho estão apresentados na Tabela 2. Os testes foram feitos em uma máquina com processador Pentium II de 400 MHz, e uma placa de vídeo ASUS GeForce 2.

Aplicação	**Posição**	**t_{min} (ms)**	**$t_{máx}$ (ms)**	**t_{med} (ms)**	**FPS**
MyWalker	1	14	35	14,56	68,67
MyWalker	2	20	58	20,93	47,77
MyWalker	3	22	43	22,85	43,75
MyWalker	4	5	15	13,21	75,70
simpmap	1	13	34	13,84	72,24
simpmap	2	20	41	20,70	48,30
simpmap	3	21	43	22,53	44,39
simpmap	4	5	17	13,21	75,70

Tabela 2 – Comparação de desempenho com (MyWalker) e sem (simpmap) o IEngine, relativa às posições da Figura 5.

Na Tabela 2, t_{min}, $t_{máx}$ e t_{med} são, respectivamente, o tempo mínimo, máximo e médio entre quadros, em milissegundos. FPS é o número médio de quadros por segundo, obtido pela fórmula $1000/t_{med}$.

A Tabela 3 mostra a quantidade de tempo acrescentada pelo IEngine em cada uma das 4 posições e a porcentagem de influência do IEngine no tempo total.

Posição	Acréscimo médio de IEngine (ms)	Porcentagem no tempo total
1	0,72	4,9
2	0,23	1,1
3	0,32	1,4
4	0,00	0,0

Tabela 3 – Influência do IEngine no caso testado

A porcentagem de influência de IEngine no tempo total de processamento manteve-se abaixo de 5%.

Não é o caso de se investigar a qualidade visual da imagem gerada com e sem o IEngine. De fato, como o IEngine é apenas uma camada intermediária de software, não interferindo com o funcionamento do motor, não deve haver alteração na qualidade da imagem. Uma simples inspeção visual mostra isto.

5. Conclusões e trabalhos futuros

Este capítulo apresenta as conclusões mais gerais sobre a proposta do IEngine e a portabilidade que pode ser conseguida com ele (seção 5.1) e tece um breve comentário sobre a facilidade de programação trazida por esta proposta (seção 5.2). Também são feitas observações sobre a portabilidade de arquivos de mundo, necessária para uma compatibilidade total entre aplicações (seção 5.3). Finalmente são apresentados possíveis trabalhos futuros utilizando ou estendendo o IEngine (seção 5.4).

5.1. Um caminho para a Portabilidade

Uma aplicação escrita sobre o IEngine é portátil desde que não faça nenhuma referência ao motor que está sendo utilizado, nem às classes concretas do IEngine. A aplicação deve referir-se às classes apenas através de suas interfaces. Se a aplicação é desenvolvida desta forma, pode-se trocar a instância de IEngine utilizada sem a necessidade de alterações no código da aplicação.

Ter-se uma interface que permite a portabilidade entre motores seria de grande utilidade. Apenas como um exemplo, podemos imaginar um jogo desenvolvido com o IEngine, que roda sobre os motores atuais. No futuro, a tendência é o surgimento de novas placas gráficas, com novas possibilidades para visualização 3D. Estas novas placas podem apresentar tanto uma melhor performance quanto uma melhor qualidade de imagem. Certamente devem surgir

novos motores, com funcionalidades específicas para estas novas placas 3D. Uma vez que estes novos motores sejam adaptados para a interface IEngine, os jogos já existentes podem rodar sobre estes novos motores, utilizando-se automaticamente das novas características das placas gráficas.

O autor da presente dissertação desconhece qualquer trabalho no mercado ou na literatura apresentando uma proposta de portabilidade de jogos entre motores 3D. Portanto, a proposta do IEngine, nesta dissertação, tem um caráter pioneiro e original. Há, entretanto, um longo caminho a ser percorrido até que esta portabilidade abranja um grande número de motores e as suas principais funcionalidades de uma forma inteligente e prática.

Apesar da crítica usual aos motores, por não ser possível a existência de um motor universal, o fato é que a utilização de motores 3D é bastante conveniente para os programadores de jogos. O programador fica livre de pensar no baixo nível, concentrando-se na lógica do jogo. Apesar dos motores serem adequados para tipos específicos de jogos, o programador que usa o IEngine necessita ganhar proficiência em apenas um mesmo ambiente. A meta planejada para o conjunto de classes do IEngine é englobar os principais tipos de jogos e as instâncias decidirem que motor utilizar. Na presente dissertação, apresentam-se evidências iniciais de que a proposta de tal camada abstrata geral é adequada e viável.

5.2. Facilidade de programação

Com a utilização do conceito de herança e a utilização dos métodos *default* introduzidos nas classes do IEngine, a programação torna-se bastante fácil para programadores minimamente experientes em C++. É possível escrever uma aplicação com comportamento padrão herdado dos métodos *default* e alterar este comportamento conforme necessário. A utilização do paradigma de *callback* torna o desenvolvimento familiar àqueles programadores acostumados com ambientes de janelas (Windows, Java AWT/Swing, etc) que seguem o mesmo paradigma.

5.3. Portabilidade dos arquivos de mundo

Com relação aos arquivos de mundo, existe uma observação importante. Cada motor trabalha com um arquivo diferente que representa o mundo a ser visualizado. Em geral estes arquivos não são compatíveis entre si. O arquivo de mundo do Crystal Space não é compatível com o arquivo de mundo do Fly3D. Então, como fazer para garantir que uma aplicação fique independente do motor utilizado?

Sendo os arquivos de dados incompatíveis, deve haver algum nível mais alto onde pode existir uma compatibilidade. Em geral, estes arquivos são gerados a partir de softwares de modelagem como o 3D Studio Max. O modelador, por exemplo, cria o mundo no 3D Studio Max e o exporta para um formato padrão, de onde um software que é parte integrante do motor compila um arquivo de mundo.

Ou em alguns casos, um *plug-in* de 3D Studio Max exporta o mundo diretamente para o formato que o motor entende. Então, pode-se considerar o software de modelagem como o ponto de contato entre motores diferentes, no que se refere ao arquivo de mundo. Para fazer uma aplicação funcionar em dois motores diferentes, continua-se tendo que ter dois arquivos de mundo diferentes, mas eles são gerados automaticamente a partir de um único modelo criado em um software de modelagem.

Um problema que surge ao se fazer isto é que alguns atributos específicos de um motor são incorporados à modelagem da cena. Por exemplo, os atributos de luzes dinâmicas devem ser definidos no momento da modelagem para que o motor possa tratar destas luzes posteriormente. O problema é que estes atributos podem ser diferentes para cada motor. Neste caso não temos uma adaptação tão imediata dos arquivos de mundo. É necessária uma adaptação manual ou a construção de *scripts* de tradução que atuam no nível do software de modelagem.

Como conclusão, uma aplicação escrita sobre o IEngine pode rodar facilmente sobre dois motores A e B, sem qualquer alteração, se os seguintes itens estão disponíveis:

- um arquivo de mundo em um formato tal que podemos exportá-lo para os motores A e B;
- e as instâncias IEngine-A e IEngine-B.

5.4. Trabalhos futuros

Esta linha de pesquisa pode, no futuro, gerar uma padronização de camada de abstração para motores 3D. A interface de IEngine ainda está em seu estado embrionário, apenas dando suporte a uma câmera que pode caminhar por um mundo estático, sem detecção de colisão.

O IEngine pode ser expandido de duas formas:

- "Horizontalmente": outras instâncias de IEngine podem ser construídas sobre outros motores. Em particular, a instância de IEngine sobre Fly3D encontra-se em estado de desenvolvimento parcial.
- "Verticalmente": mais funcionalidade pode ser acrescentada à interface IEngine. As mais importantes são: a detecção de colisão e o controle de objetos no mundo.

São feitos a seguir alguns comentários sobre o desenvolvimento "vertical" do IEngine.

A detecção de colisão pode facilmente ser implementada, pois é uma característica presente em todos os motores 3D, apenas com interfaces diferentes em cada caso. É necessário analisar estas diferentes interfaces e optar por uma boa modelagem, condizente com os paradigmas presentes no IEngine.

A questão do controle de objetos no mundo foi muito bem resolvida no motor Fly3D, com seu paradigma de *plug-ins* em bibliotecas dinâmicas (DLLs). Esta idéia deve ser adaptada para o modelo de herança de classes utilizado no IEngine. Objetos podem ser modelados como filhos de uma única classe (que pode se chamar `IE_IObject`). Pensando na instância IEngine-Fly3D, esta classe pode ser uma casca sobre um *plug-in* de Fly3D genérico. O comportamento deste *plug-in* deve ser deixado em aberto para ser implementado pelas classes que o programador irá desenvolver, ou seja, as filhas de `IE_IObject`.

A questão de jogos distribuídos também pode ser abordada nesta mesma linha. Os principais paradigmas de jogos distribuídos podem ser modelados sob a forma de interfaces do IEngine e implementados sobre motores específicos, instanciando-se assim estas interfaces de distribuição do IEngine sobre o motor utilizado. Quando um determinado paradigma não estiver disponível em um motor, pode-se implementar a sua funcionalidade diretamente como um módulo do IEngine, escrevendo-se todo o código em classes que implementam as interfaces de distribuição do IEngine.

Uma vez que a interface seja melhor elaborada (uma expansão "vertical" no nível de interface), é viável a criação de uma comunidade para o desenvolvimento "horizontal" desta interface sobre vários motores existentes. Com todas as principais funcionalidades necessárias modeladas sob a forma de interfaces, cada desenvolvedor pode, seguindo o padrão estabelecido, desenvolver a instância do IEngine sobre o motor que desejar. O mecanismo de herança de interfaces

garantirá a compatibilidade das aplicações existentes com as novas instâncias de IEngine.

Além da expansão "vertical" e "horizontal", o IEngine pode ser expandido também com a construção de uma API em outra linguagem (por exemplo, Java ou Lua). Tal expansão automaticamente torna possível, a partir de programas escritos nesta nova linguagem, a utilização de todos os motores 3D disponíveis sob o IEngine.

O uso de Lua [Lua 02] é especialmente atrativo, não apenas porque tem um reconhecimento internacional na indústria de jogos, mas também por sua elegância e flexibilidade. Lua substitui com vantagem as linguagens script encontradas em alguns motores. Alias, Lua atualmente já é uma opção de linguagem no motor Crystal Space. A vantagem do uso de Lua deve-se ao fato dela ser uma linguagem de configuração (ou, mais corretamente, uma linguagem de extensão) que integra construtores imperativos com facilidades reflexivas e facilidades de descrição de dados. Esta integração permite o uso de arrays associativos e dinâmicos, tipação dinâmica, manipulação dinâmica de exceções (em situações onde linguagens tipadas estaticamente dariam erro de compilação) e funções como valores de primeira ordem.

A incorporação de Lua ao IEngine pode ser feita através do encapsulamento de todas as funcionalidades da API do IEngine na linguagem Lua. Uma vez que todas as funcionalidades do IEngine sejam disponibilizadas sob a forma de uma API em Lua, pode-se escrever jogos em Lua utilizando uma instância do IEngine

sobre um motor 3D convencional. Estes jogos ficam independentes de motor, tal como qualquer outra aplicação escrita sobre o IEngine. Neste caso, é possível a troca de motor (trocando-se a instância de IEngine) sem que seja necessário fazer alterações na lógica do jogo em Lua.

Utilizando-se Lua, é possível a construção de jogos de forma rápida, simples e portátil. Os jogos podem inclusive ter uma complexidade bastante razoável, tanto gráfica quanto de inteligência e lógica de jogo. Neste caso, para problemas de baixo nível, como a visualização, dá-se um tratamento de baixo nível, nos motores. Para problemas de alto nível, como a lógica de jogo, dá-se um tratamento de alto nível, em Lua.

6. Bibliografia

[Bouvier 99] Bouvier, Dennis J., 1999 – *Getting Started with the Java3D API* – Sun Microsystems

[Carasso&Staa 01] Carasso, Felipe e Staa, Arndt Von, 2001 – *Utilizando JNI para adicionar implementação nativa C/C++ a Programas Java* – PUC-Rio (não publicado)

[Crystal 01] Crystal Space 3D Engine, 2001 – http://crystal.linuxgames.com/

[DeLoura 00] DeLoura, Mark A., 2000 – *Game Programming Gems* – Charles River Media

[Eberly 01] Eberly, David H., 2001 – *3D Game Engine Design* – Morgan Kaufmann

[Fly3D 01] Fly3D, 2001 – http://www.fly3d.com.br/

[Gamedev 01] Game Developer's Magazine, 2001 – http://www.gdmag.com/

[GameStudio 02] 3D Game Studio – http://www.conitec.net/a4info.htm

[Gamma 94] Erich Gamma et al., 1994 – *Design Patterns: Elements of Reusable Object-Oriented Software* – Addison-Wesley

[Genesis3D 01] Genesis 3D, 2001 – http://www.genesis3d.com/

[IEngine 02] ICAD/PUC-Rio, 2002 – *IEngine* – http://www.icad.puc-rio.br/iengine/

[JNI 01] Sun Microsystems, 2001 – *The Java Tutorial: Java Native Interface* – http://java.sun.com/docs/books/tutorial/native1.1/

[Karsten 00] Isakovic, Karsten, 2000 – *3D Engines List* – http://cg.cs.tu-berlin.de/~ki/engines.html

[Lua 02] TeCGraf/PUC-Rio, 2002 – *The Programming Language Lua* – http://www.lua.org

[Staa 00] Staa, Arndt von, 2000 – *Programação Modular* – Campus

[Stroustrup 00] Stroustrup, Bjarne, 2000 – *A linguagem de programação C++* – Bookman

[Sun 02] Sun Microsystems, 2002 – *Tutorial: Creating a GUI with JFC/Swing* – http://java.sun.com/docs/books/tutorial/uiswing/

[Watt&Policarpo 00] Watt, Alan & Policarpo, Fabio, 2000 – *3D Games, Volume 1: Real-time Rendering and Software Technology* – Addison-Welsey