Mozart de Siqueira Campos Araújo Filho

Um Editor de Cenários Urbanos para Aplicações de Realidade Virtual

Recife, PE

7 de Março de 2003 (2003)

Mozart de Siqueira Campos Araújo Filho

Um Editor de Cenários Urbanos para Aplicações de Realidade Virtual

Dissertação de Mestrado

Orientador:
Alejandro C. Frery

Centro de Informática
Universidade Federal de Pernambuco

Recife, PE

7 de Março de 2003 (2003)

Dissertação de Mestrado intitulada *"Um Editor de Cenários Urbanos para Aplicações de Realidade Virtual"*, defendida por Mozart de Siqueira Campos Araújo Filho, em 07 de Março de 2003, em Recife-PE, perante banca examinadora constituída pelos doutores:

Prof. Dr. Alejandro C. Frery
Orientador

Prof. Dr. Márcio Serolli Pinho
Pontifícia Universidade Católica do Rio
Grande do Sul

Profa. Dra. Judith Kelner
Universidade Federal de Pernambuco

Abstract

Any technology-aware person can verify that 3D applications have evolved and diversified in recent years. These technologies, now easily available, were traditionally employed in military and industrial simulation environments (to reduce desing and tests costs) and now they have reached the entertainment market. 3D tools are also available in edition platforms and in immersive environments. This spread is caused, among other factors, by new low-cost hardware devices and innovatives applications. This work studies and develops one of the most frequently used 3D application for PC: a scenario editor. Such applications allow the modelling of 3D objects and stages. A good editor, in this context, is intuitive and provides the user with interaction mechanisms he/she expects to deal with the specific information being handled. This kind of 3D application is complex from the developer's viewpoint, being a multiview 3D engine and hardware acceleration some of the critical aspects that have to be treated.

Structural and development patterns were adopted for this project, a detailed *framework* was developed and a case study of intrinsic problems to 3D applications was carried out. These patterns were proposed based on a case study of two specific scenario editors in the urban building and urban modelling context.

Keywords: Scenario Editor, 3D Programming, Computer Graphics, Software Engineering.

Resumo

Se observarmos os diversos tipos de aplicações tridimensionais (3D) nos últimos tempos, podemos ver claramente a evolução e a diversificação deste tipo de sistema e a sua inserção no mercado. Essas aplicações têm sido utilizadas nas mais diversas áreas: primeiramente em ambientes de simulações para reduzir o custo de projeto e teste de equipamentos militares ou industriais; posteriormente em jogos de computadores e em ferramentas de edição específicas e, atualmente, em plataformas imersivas. Entretanto, mesmo que esse tipo de aplicação tenha se desenvolvido bastante e que dispositivos de hardware estejam cada vez mais disponíveis ao usuário final, novas formas de utilização deste tipo de tecnologia estão aparecendo a cada instante.

O presente trabalho visa colocar em estudo, projetar e desenvolver um dos tipos de aplicações mais comuns dentro de ambientes 3D para PC: os editores de cenários. Um editor de cenários é um ambiente que permite a modelagem de objetos ou ambientes 3D. Um bom editor oferece o desafio de ser intuitivo ao usuário final, além de oferecer os mecanismos de interação que esse usuário espera de uma aplicação desse tipo. Esta é uma aplicação 3D bastante complexa dentro da ótica de programação, pois envolve o desenvolvimento de um motor de visualização 3D; além de tratar de desafios intrínsecos a esse tipo de aplicação, como fornecer, ao motor, mecanismos que o adaptem ao uso de múltiplas janelas de visualização (*viewports*) e ao uso de mecanismos de aceleração por hardware. O citado motor deverá tratar de eventos intrínsecos ao dispositivo de navegação dentro de uma *viewport*. Neste projeto foi adotado um padrão de desenvolvimento e estruturação, isto é, foi desenvolvido um *framework* detalhado, além de um estudo de caso de problemas intrínsecos a aplicações 3D. O padrão adotado foi firmados, com base, em um estudo de caso que consistiu na construção de dois editores de cenários específicos no âmbito de construção de edificações urbanas e modelagem urbana.

Palavras-Chave: Editor de Cenários, Programação 3D, Computação Gráfica, Engenharia de Software.

Agradecimentos

Inicialmente, agradeço ao professor Alejandro por, dentro da loucura que foi desenvolver o presente projeto de Mestrado, em razão da minha necessidade de defender esta tese em tão pouco tempo, ter sempre acreditado em meu esforço, no progresso e término deste trabalho.

À minha família, por ter me dado suporte nas noites e dias dedicados a este projeto, durante os cinco anos do meu curso de Ciência da Computação, e agora no mestrado, e sendo também os responsáveis por eu ser quem sou hoje: um vitorioso apesar de minha deficiência.

À professora Kátia que sempre atuou de maneira discreta mas impactante na minha vida acadêmica. Foi ela a professora da primeira cadeira pela qual me apaixonei, Algoritmos e Estruturas de Dados, cadeira na qual tive a oportunidade de prestar minha primeira monitoria. Foi também a responsável pelo meu envolvimento de quatro anos com o time de computação da UFPE. Por fim, ela pode até nem saber, mas foi a grande responsável pelo meu início de trabalho com o professor Alejandro.

À professora Judith, por me proporcionar a oportunidade de ministrar um curso de OpenGL na disciplina de Realidade Virtual e Multimídia, quando ainda me encontrava na graduação, o que me fez rever meus conceitos e gerou muitas idéias.

Ao Alex Ferrier (amigo on-line) que muito me ajudou com as dúvidas mais práticas de programação para Windows e MFC. Além de ser um bom conselheiro de OpenGL.

Agradeço ao Paulo Abadie Guedes por ter empurrado minha cadeira de rodas na rampa do Centro de Informática, no último dia para que eu marcasse a data da tese, pois a bateria estava se esvaindo. Agradeço não só a ele, mas a todas as pessoas que me ajudaram antes, em tantas outras vezes, quando tive de subir a rampa do Centro ou tive que ser levantado para subir escadas, à fim de assistir a uma simples palestra. Neste caso, principalmente, agradeço aos meus colegas de graduação, que sempre foram grandes companheiros. Acreditem pessoal! São nessas horas que sentimos quanto a natureza humana é boa, pois sempre tem alguém disposto a ajudar.

Mas ao mesmo tempo em que congratulo essas boas pessoas que me ajudaram durante minha vida, gostaria de registrar aqui neste parágrafo minha indignação. É válido lembrar que as dificuldades de locomoção que tive de enfrentar foram em grande parte decorrentes da insensibilidade de projetistas, que não tiveram a capacidade de se colocar no lugar das pessoas que utilizarão sua obra, ou porque algum burocrata egoísta se esqueceu que a Constituição garante a todos o direito de ir e vir, e resolveu desrespeitar o projeto do arquiteto. Porém, isso acontece porque não há leis que se cumpram neste pais, leis que

revivam a memória do arquiteto ou mexam com a consciência do burocrata, tudo por meio do bolso, claro.

Ao Centro de Informática, sem dúvida, a maior das surpresas quando comecei o curso de Computação. Encontrei aqui mais do que um curso com nível de excelência, reconhecido internacionalmente, mas uma família nos professores, colegas e funcionários, e desafios que tanto buscava para minha vida profissional. E pensar que tudo começou quando ganhei meu primeiro PC, há apenas dois anos do vestibular.

Acreditem, ou não, agradeço a Papoula, a tartaruga que recebi de presente. Antes de ganhá-la como um presente milagroso, tinha crises freqüentes de erisipela, havia um "Boato Popular" que dizia se alguém criasse um destes animais, deixaria de ficar doente. Pois é, desde então, não duvido de "Boatos Populares".

A Deus acima de tudo. Por estar vivo para poder estudar, aprender e realizar. Por ter dado a mim prazeres, desafios, amores e desafetos. Por ter construído em mim um humano com amor aos mais simples prazeres da vida, como o de dar um bom sorriso ou até mesmo o de sentir uma leve brisa passar por mim. Por ter feito de mim um guerreiro que encara desafios como um passo a mais para que se torne melhor e mais forte.

Ao amor no mais amplo dos sentidos; ao amor e honra de um amigo, ao amor incondicional da família e à capacidade de me fazer amar simples e puramente. Aos meus desafetos, por terem tornado minha vida mais difícil e terem feito despertar o guerreiro que está dentro de mim, por me colocarem em situações em que o guerreiro sozinho jamais venceria, para que eu pudesse contar com os amigos. O que eu posso dizer a vocês, sorrindo, é que vocês me fizeram melhor que nunca.

E por último agradeço ao destino, de uma maneira geral, que agiu de forma misteriosa, sempre que precisei de ajuda. Acredito que o destino operou quando ocorreu a última greve me ajudando a ganhar o tempo que não dispunha para concluir este Mestrado. Na minha opinião, nem sempre greve é perda de tempo, se você sabe usar bem o tempo perdido, principalmente quando existe um amigo que ajude com um trabalho ou um mestre que oriente. Esse destino maravilhoso que me reservou um 2002 formidável, em que encontrei novos amigos no pessoal do time de programação, que reservou lugar vitorioso nos meus desafios e me fez atingir as metas, que tinha em mente para este ano, que eu julgava ser mais um na minha vida. O mesmo destino que me arremessa em 2003, um ano cheio de promessas e desafios.

Lista de Figuras

1 Interface do AC3D Modeler . . . . p. 23

2 Interface do CiteMap Builder . . . . p. 24

3 Interface do Internet Space Builder. . . . . p. 25

4 Bibliotecas envolvidas no Editor de Prédios. . . . . p. 26

5 Bibliotecas envolvidas no Editor de Cidades. . . . . p. 27

6 Imagem de um mesmo objeto com e sem iluminação. . . . . p. 35

7 Uma normal por face e uma normal por vértice. FONTE: Game Institute (2002) . . . . p. 36

8 Esfera com uma normal por face e com normal por vértice. . . . . p. 36

9 Tipos de emissão de luzes. . . . . p. 37

10 Mecha e seus sub-componentes. . . . . p. 46

11 Utilizando a pilha de matrizes para organizar transformações. . . . . . p. 47

12 Rodovia: blocos de concreto mapeados sobre o terreno. . . . . p. 51

13 Quad Tree em um mapa de alturas. . . . . p. 59

14 Quad Tree com triangularização correspondente. . . . . p. 60

15 Erros dos pontos no mapa de altura devido à Quad Tree. . . . . p. 61

16 Árvore binária com triangularização correspondente a cada nível ℓ. . . . p. 62

17 Árvore binária mapeada no mapa de alturas. . . . . p. 62

18 Triangularizando o terreno por meio do ROAM. . . . . p. 63

19 Operações de divisão e junção de triângulos. . . . . p. 63

20 Operação de divisão sem triângulo base. . . . . p. 64

21 Erro inerente à não divisão do triângulo. . . . . p. 64

22 Textura do terreno. . . . . p. 71

23 Classes do Graphic Engine correspondentes ao terreno. . . . . p. 73

24 Estrutura do código da Tabela 22. . . . . p. 82

25 Objetos básicos de VRML modelados. p. 83

26 Estrutura simplificada para definição dos nós VRML. . . . . p. 84
27 Gerenciador de nós VRML. . . . . p. 85
28 Trecho de código VRML gerado pelo Building Editor. . . . . p. 86
29 Visualização do código mostrado na Figura 28 . . . . p. 87
30 Exemplo da arquitetura documento/visão. FONTE: MSDN Home (2002) p. 90
31 Documento na arquitetura Document/View. FONTE: MSDN Home (2002) p. 90
32 Building Editor, tela inicial. . . . . p. 104
33 Barra de ferramentas do Building Editor. . . . . p. 105
34 Menu view do Editor com os componentes grade e eixo. . . . . p. 105
35 Comando File do editor. . . . . p. 106
36 Exportando para VRML. . . . . p. 106
37 Editor com a grade e o eixo habilitados. . . . . p. 106
38 Painel de materiais. . . . . p. 107
39 Painel de texturas. . . . . p. 108
40 Painel do prédio editado. . . . . p. 109
41 Editor com um prédio gerado. . . . . p. 111
42 Visões do Building Editor. . . . . p. 111
43 Tela inicial do City Editor. . . . . p. 113
44 Opções do terreno para a aplicação. p. 113
45 Importação do terreno para a aplicação. . . . . p. 114
46 Definição das dimensões do mapa de alturas. . . . . p. 114
47 Visualização do ambiente com o terreno. . . . . p. 115
48 Tela de opções do terreno. . . . . p. 116
49 Menu de cores do terreno. . . . . p. 116
50 Menu detalhado de cores do terreno. p. 117
51 Ferramentas de navegação do *City Editor*. . . . . p. 117
52 Menu Organize. . . . . p. 117
53 Painel de importação do prédio. . . . . p. 119
54 Painel da Cidade. . . . . p. 120
55 Prédio em um trecho de terreno planificado. . . . . p. 121

56 Painel de Rodovias. . . . . p. 122

57 Ambiente com uma rodovia. . . . . p. 123

58 Exemplo de ambiente no City Editor. . . . . p. 124

59 Visão geral do Graphic Engine . . . . p. 134

60 Classes do VRMLSaver . . . . p. 142

Lista de Tabelas

1 Bibliotecas definidas pelos Editores . . . . p. 28

2 As classes e estruturas mais simples. . . . . p. 31

3 Modos de filtragem de uma textura para aproveitar mipmaping. . . . . p. 42

4 Objetos definidos pelo `CEiostream`. . . . . p. 80

5 Objetos definidos a partir de MFC para o editor de prédios e seu comportamento. . . . . p. 96

6 Objetos definidos a partir de MFC para o editor de prédios e seu comportamento. . . . . p. 96

7 Objetos definidos a partir de MFC e seu comportamento. . . . . p. 105

8 Objetos definidos a partir de MFC e seu comportamento. . . . . p. 118

Lista de Códigos

1 Estrutura do Objeto Base do Motor. . . . . . p. 34
2 Estruturas definidas para implementação de luz. . . . . . p. 38
3 Processo de renderização de um material. . . . . . p. 39
4 Estrutura para gerar a textura dentro de uma janela OpenGL. . . . . . p. 43
5 Renderização da Textura usando Handler. . . . . . p. 43
6 Interface da classe CTexture. . . . . . p. 45
7 Renderização da malha. . . . . . p. 47
8 Renderização da Sub-Malha. . . . . . p. 48
9 Criando uma display list. . . . . . p. 49
10 Renderizando a display list. . . . . . p. 49
11 Escopo da classe CIDFactory. . . . . . p. 52
12 Objeto básico do motor. . . . . . p. 54
13 Objeto básico com acesso aos estados. . . . . . p. 55
14 Uma classe que herda do `CObjectState`. . . . . . p. 56
15 Prédio processando seu estado. . . . . . p. 56
16 Renderização trivial do terreno. . . . . . p. 58
17 Gerando a Textura para OpenGL. . . . . . p. 69
18 Estrutura da textura do terreno. . . . . . p. 72
19 trechos de código do PrimitiveContainer. . . . . . p. 77
20 Escrita em formato proprietário de arquivos. . . . . . p. 78
21 Importando arquivo depois de feito o parser. . . . . . p. 79
22 Exemplo de código VRML. . . . . . p. 81
23 Uma MF structure. . . . . . p. 83
24 Shape com nós filhos de tipos genéricos. . . . . . p. 84

25 Usando DEF e USE para exportar VRML. . . . . p. 84

26 Comportamento de uma classe do `VRMLSaver`. . . . . p. 85

27 Adaptando a visão ortográfica ao redimensionamento da janela. . . . . p. 92

28 Redimensionando a visão em perspectiva . . . . p. 92

29 A nossa classe de documento. . . . . p. 94

30 Modelo do prédio a ser rederizado. p. 94

31 Modelo da cidade a ser rederizada. p. 94

32 Adicionando textura ao documento . . . . p. 95

33 Mapeando mensagens em MFC . . . . p. 97

34 Adicionando uma textura ao modelo . . . . p. 97

35 Prédios num arquivo City Editor . . . . p. 99

36 Fábrica de Flyweights do Editor de Cidades. . . . . p. 100

37 Fábrica de Flyweights criando instância a partir de um arquivo. . . . . p. 100

38 Fábrica de Flyweights do tipo prédio. . . . . p. 100

39 Método `CreateInstance` da classe `CBuildingFlyweightFactory`. . . . p. 101

40 ComboBox recorrendo ao Flyweight para importar/buscar um prédio. . p. 101

41 Importando um prédio para a cidade. . . . . p. 102

Sumário

1 Introdução p. 18

2 Editores de Cenários p. 22

2.1 AC3D Modeler . . . . p. 22

2.2 CiteMap Builder . . . . p. 23

2.3 Internet Space Builder . . . . p. 24

3 O Editor como uma série de componentes reutilizáveis p. 26

4 O Motor Gráfico p. 30

4.1 Estruturas Básicas . . . . p. 31

4.2 Objetos Principais . . . . p. 33

4.2.1 Objeto Padrão . . . . p. 34

4.2.2 Luzes . . . . p. 35

4.2.3 Materiais . . . . p. 37

4.2.4 Texturas . . . . p. 39

4.2.4.1 Lendo e Utilizando Texturas . . . . p. 39

4.2.4.2 Gerenciamento de Textura . . . . p. 43

4.2.5 Malhas . . . . p. 45

4.2.5.1 A classe CMesh . . . . p. 45

4.2.5.2 A classe CSubMesh . . . . p. 47

4.2.6 Prédios . . . . p. 49

4.2.7 Rodovias . . . . p. 50

4.3 Repositórios . . . . p. 51

4.3.1 IDFactory . . . p. 51
4.3.2 Iluminação . . . p. 52
4.3.3 Managers . . . p. 53
4.3.4 Mudando o estado dos objetos do motor . . . p. 55
4.4 Estruturas para Renderização do Terreno . . . p. 57
4.4.1 Algoritmo Trivial . . . p. 58
4.4.2 Algoritmo de Röttger . . . p. 59
4.4.3 Algoritmo ROAM . . . p. 61
4.4.4 Comparação de algoritmos de renderização de terreno . . . p. 64
4.4.4.1 Algoritmo Trivial . . . p. 65
4.4.4.2 Algoritmo Röttger . . . p. 66
4.4.4.3 Algoritmo ROAM . . . p. 67
4.4.5 Coloração, Textura e Iluminação . . . p. 68
4.4.6 Arquitetura das classes de terreno . . . p. 71

5 Usando Interpretadores para Entrada e Saída de Arquivo p. 75
5.1 Por que não usar MFC para escrever arquivos? . . . p. 75
5.2 BEiostream . . . p. 76
5.2.1 Estruturas de Importação . . . p. 77
5.3 CEiostream . . . p. 80
5.4 VRMLSaver . . . p. 81
5.4.1 VRML, uma visão rápida . . . p. 81
5.4.2 Gerando as classes VRML . . . p. 82
5.5 O padrão de projeto Interpreter . . . p. 86

6 Acoplamento com MFC p. 88
6.1 Por que MFC? . . . p. 88
6.2 A arquitetura documento/visão . . . p. 89
6.3 A criação de Viewports . . . p. 90

6.4 A nossa classe CDocument . . . . p. 93

6.5 Construindo eventos e interface gráfica . . . . p. 95

6.6 Flyweight e suas diversas aplicações . . . . p. 98

7 Guia do Usuário. p. 103

7.1 Instalação da aplicação . . . . p. 103

7.2 O *Building Editor* . . . . p. 103

7.2.1 Barra de Ferramentas e Menus . . . . p. 104

7.2.2 Formulários de criação de construções . . . . p. 107

7.2.2.1 Painel de materiais . . . . p. 107

7.2.2.2 Painel de Textura . . . . p. 108

7.2.2.3 Painel do Prédio Editado . . . . p. 108

7.2.3 Navegação nas Viewports . . . . p. 110

7.3 O City Editor . . . . p. 112

7.3.1 Funções do *City Editor* . . . . p. 112

7.3.1.1 Importando o terreno. p. 113

7.3.1.2 Definindo opções de renderização do terreno. . . . . . . p. 115

7.3.1.3 Navegação/Operação nas Viewports . . . . p. 117

7.3.1.4 Painel do Prédio Importado . . . . p. 119

7.3.1.5 Painel da Cidade . . . . p. 119

7.3.1.6 Painel de Rodovias . . . . p. 121

7.3.2 Barra de Ferramentas Principal . . . . p. 122

7.4 Comentários adicionais . . . . p. 124

8 Conclusão e Trabalhos Futuros p. 126

Apêndice A – Diagrama de Classes do Graphic Engine p. 133

Apêndice B – Detalhe das Classes do Graphic Engine p. 135

B.1 Diagrama 1 . . . . p. 135

B.2 Diagrama 2 . . . . p. 136

Apêndice C – Diagrama de Classes da Biblioteca MathLib p. 137

Apêndice D – Diagrama de Classes TextureLoader - A classe CGLBMP p. 138

Apêndice E – Diagrama de classes do BE_iostream p. 139

Apêndice F – Diagrama de classes do CE_iostream p. 140

Apêndice G – Diagrama de classes do VRMLSaver p. 141

Apêndice H – Diagrama de Classe do BuildEditor p. 143

Apêndice I – Diagrama de Classe do CityEditor p. 144

I.1 Diagrama 1 . . . . p. 144

I.2 Diagrama 2 . . . . p. 145

Apêndice J – Diagrama de Classes da Viewport p. 146

Apêndice K – Código para renderizar uma sub-malha p. 147

Apêndice L – Padrões de projetos utilizados p. 149

L.1 Singleton . . . . p. 149

L.1.1 Aplicabilidade . . . . p. 149

L.1.2 Estrutura . . . . p. 150

L.1.3 Participantes . . . . p. 150

L.2 Chain Of Responsibility . . . . p. 150

L.2.1 Aplicabilidade . . . . p. 150

L.2.2 Estrutura . . . . p. 151

L.2.3 Participantes . . . . p. 151

L.3 Interpreter . . . . p. 151

L.3.1 Aplicabilidade . . . . p. 152

L.3.2 Estrutura . . . . p. 152

L.3.3 Participantes . . . . p. 152

L.4 Flyweight . . . . p. 153

L.4.1 Aplicabilidade . . . . p. 153

L.4.2 Estrutura . . . . p. 154

L.4.3 Participantes . . . . p. 155

L.5 Composite . . . . p. 155

L.5.1 Aplicabilidade . . . . p. 156

L.5.2 Estrutura . . . . p. 156

L.5.3 Participantes . . . . p. 156

Apêndice M – Exemplo de código VRML gerado pelo Building Editor p. 158

Apêndice N – Gerando a textura do terreno p. 160

Apêndice O – Contribuições e Artigos publicados p. 162

1 Introdução

Se observarmos os tipos de aplicações 3D nos últimos tempos, podemos ver claramente uma evolução e diversificação deste tipo de produto e a sua inserção no mercado. Essa diversificação ocorre principalmente porque dispositivos de hardware e software se tornaram acessíveis ao usuário final, levando ao surgimento de várias idéias de novas aplicações que ainda não foram analisadas dentro deste contexto.

Antigamente o mercado de aplicações 3D era restrito a simuladores para forças armadas e indústrias de grande porte. Em ambos os casos, a intenção primordial era vender produtos que, pelo preço, só seriam acessíveis a grandes corporações que poderiam arcar com o custo de construção de um protótipo real ou com os custos e riscos de treinamento de pessoal com ferramentas reais.

Logo depois, em meados dos anos 90 a preocupação com um padrão em termos de compatibilidade hardware e software, para tornar esse tipo de tecnologia acessível ao usuário final, tomou o lugar da venda de produtos especializados. Foi quando surgiram duas bibliotecas padronizadas, que tinham como objetivo construir mecanismos para fixar padrões de programação compatíveis com os diversos hardwares disponíveis no mercado; são elas OpenGL (Silicon Graphics Inc, 2003) e DirectX (Microsoft, 2003).

Posteriormente vieram os jogos de computadores, que hoje se mostram cada vez mais realistas por usufruírem do suporte em hardware, como placas de vídeo cada vez mais eficientes que tomam para si cada vez maior responsabilidade no processo de renderização[1] e compatíveis com versões anteriores e posteriores de software. Esta compatibilidade é garantida por meio da utilização de drivers compatíveis, como OpenGL.

Por último, temos o momento que estamos atravessando agora, com uma verdadeira guerra sendo estabelecida pela busca de um padrão de uma linguagem 3D descritiva. Esta linguagem deverá possibilitar a construção de aplicações que possam mais do que ter o propósito de uma boa visualização, mas que permitam a distribuição de ambientes virtuais. Tais ambientes deverão oferecer aos seus usuários a possibilidade de serem com-

[1]Renderização – processo no qual as representações matemáticas, em forma de estrutura de dados, de uma imagem ou objeto são mapeadas para uma representação gráfica, quase sempre adaptadas a um plano.

pletados e personalizados, respeitando a funcionalidade e o aspecto específico originais.

Dentro deste contexto, podemos citar as Simulações Urbanas de Realidade Virtual como aplicações que estão se mostrando cada vez mais utilizadas no turismo, na construção de cidades planejadas e na criação de jogos e simuladores, dentre outras aplicações. Atualmente só pessoas que possuem um certo domínio de alguma linguagem de programação ou que são especialistas em algum ambiente específico para a construção de ambientes podem construir cenários urbanos com plena liberdade. Este fato é um problema que dificulta a disseminação da realidade virtual 3D.

Constatado o problema, este projeto tem como objetivo desenvolver um editor de cenários urbanos. O desenvolvimento desta ferramenta utilizou uma visão orientada a objetos, de modo que garantiu que o sistema venha a ser útil para a construção de futuras ferramentas, bem como a sua extensibilidade dentro de seu próprio contexto. Os cenários que serão criados com esta ferramenta podem ser visualizados utilizando plataformas de realidade virtual imersivas ou não, mas a ferramenta em si deverá aproveitar o ambiente imersivo disponível atualmente no Centro de Informática da UFPE, que é composto de um rastreador, um óculos que permite visualização 3D e um computador com os softwares *World Up* e *3D studio Max* (Sense 8, 2003; Autodesk, 2003). Isto foi feito a partir da compatibilidade da ferramenta com o formato VRML[2].

Neste trabalho serão descritos os dois editores que foram desenvolvidos: um editor de prédios e um editor de cenários urbanos. O primeiro editor, apesar de mais simples, possui toda a base gráfica necessária ao segundo editor que, por sua vez, prioriza mais a questão da interface com o usuário.

Mostraremos também aqui que, além de nos preocuparmos com a construção da aplicação em si, nos preocupamos também em utilizar ferramentas da engenharia de software para simplificar o desenvolvimento destes editores. As técnicas de engenharia de software utilizadas podem vir a definir um padrão de estruturação para aplicações deste tipo.

Mostraremos aqui também que o *framework*[3] proposto se adequou à estruturação da biblioteca OpenGL e que tenta em todos os aspectos criar mecanismos que englobem até os mecanismos de aceleração de hardware.

Além disso, a estrutura do software proposto também conseguiu se adequar aos

[2]VRML (*Virtual Reality Modeling Language*): Linguagem descritiva para ambientes 3D orientada também à Web que, por já estar consolidada desde 1997, é um padrão estabelecido na indústria de aplicações 3D. Esta linguagem é compatível com praticamente todos os editores disponíveis no mercado, tanto para importação quanto para exportação de mundos.

[3]Arquitetura padrão que descreve um esqueleto, o qual qualquer programador pode seguir, para construção de um tipo de aplicação especifica.

padrões de programação MFC[4], bem como gerou idéias e padrões a serem desenvolvidos para construção de aplicações 3D em MFC.

Mais a frente no documento veremos o estado da arte deste tipo de aplicação. Posteriormente, iremos exibir detalhes das tecnologias utilizadas. Também detalharemos o porquê da escolha de cada componente estrutural do trabalho em questão, chegando assim em uma conclusão sobre pontos comuns que podem ser utilizados em editores futuros.

Agora que já pusemos em pauta os focos principais do nosso projeto descreveremos, a seguir, as decisões de projetos tomadas durante o desenvolvimento do editor e mostraremos o porquê de cada uma das linguagens e mecanismos usados.

Pela necessidade de uma biblioteca gráfica que seja um padrão em termos de portabilidade e eficiência, foi escolhida a biblioteca OpenGL (HELIUM, 2001; WOO, 1999; WRIGHT JR; SWEET, 1999). OpenGL é um padrão já consolidado, implementado e otimizado tanto em nível de software quanto de hardware. Existem no mercado placas gráficas que permitem a aceleração por hardware do processamento das funções que fazem parte desta biblioteca.

Para o desenvolvimento desta aplicação foi escolhida a linguagem de programação orientada a objetos C++. Esta linguagem, além de ser plenamente compatível com OpenGL, que é feito originalmente em C, é ideal para o desenvolvimento de aplicações que requerem alto desempenho e grande capacidade de processamento, como é o caso de aplicações 3D. Pelo fato de ser uma linguagem orientada a objetos, ela permite um melhor uso dos conceitos e técnicas de engenharia de software.

A biblioteca MFC também foi utilizada por ser compatível com Windows, o sistema operacional da plataforma imersiva disponível aqui no Centro de Informática da UFPE, e por conta de ser uma biblioteca orientada a objetos compatível com C++. Com a escolha de MFC foi necessário trabalhar com o compilador Visual Studio da Microsoft, que oferece todo o suporte necessário a programação Windows.

Depois de escolhidas as ferramentas necessárias para a programação do editor de cenários, foram lidos diversos artigos e tutoriais sobre MFC e programação 3D em geral. Também foi feita uma revisão geral sobre a biblioteca OpenGL, antes do início do processo de codificação, já que ela não é abordada em nenhuma disciplina do curso oferecido pelo Centro de Informática.

Entre os artigos mais úteis, podemos citar *Creating 3D Tools with MFC* encontrado no site GameDev (2001), o único artigo que retrata como construir editores de cenários

[4]MFC (*Microsoft Foundation Classes*): pode ser descrita como "o framework da Microsoft para o visual C++" . Biblioteca definida para dar suporte à criação de janelas, aplicativos, aplicações distribuídas e eventos do Windows.

multi-viewport com MFC, além de definir um padrão para tratar eventos dentro dessas janelas ativas. Esse padrão é discutido e melhorado no nosso editor. Também foram usados artigos e tutoriais sobre programação MFC mais gerais. No início, artigos mais introdutórios, disponíveis na web como os da DevCentral (DevCentral, 2002a, 2002b), foram necessários. Posteriormente a biblioteca padrão MSDN – *Microsoft Developer Network* (ver a referência (MSDN Home, 2002)) foi usada quase que diariamente, por ser a mais completa para desenvolvimento em qualquer plataforma ou linguagem com suporte da Microsoft.

Neste trabalho analisamos ambientes de edição disponíveis na Home Page Web3D Consortium, uma home page padrão na área de Realidade Virtual, e três deles, que possuem características de interesse, foram analisados detalhadamente. Esta análise constitui-se na pesquisa sobre o "estado da arte" deste tipo de aplicação. Algumas das idéias dessas ferramentas foram aproveitadas nos nossos desenvolvimentos.

Um outro ponto importante no contexto da aplicação foi o fato de um motor 3D simplificado ter sido utilizado. Esse motor é do curso de OpenGL do Game Institute que foi feito pelo aluno nos últimos meses de 2001, e foi adaptado e otimizado para o nosso editor.

Um motor é um dispositivo que encapsula comandos de renderização 3D, disponíveis em bibliotecas como OpenGL. No nosso caso, muda o paradigma de programação de imperativo para o paradigma orientado a objetos.

Nossa ferramenta também foi completamente adaptada para exportação de arquivos em formato VRML. Esses arquivos são essenciais para compatibilidade da ferramenta desenvolvida com outras existentes no mercado, e principalmente com a plataforma disponível no Centro de Informática da UFPE.

Temos, então, duas aplicações funcionais que, mais do que atender exclusivamente ao propósito de desenvolver um trabalho sobre questões de interface homem máquina, fixa uma série de padrões no desenvolvimento deste tipo de aplicativo.

Finalmente, o texto foi preparado empregando LaTeX 2_ε, e utilizando o estilo ABNTeX (Código Livre, 2003), que é uma aproximação ao formato da ABNT (Associação Brasileira de Normas Técnicas) (ABNT, 2001; Universidade Federal do Paraná, 2000) para teses e dissertações. As referências bibliográficas foram administradas com o BiBTeX, empregando o estilo 'autor-ano' implementado como parte do ABNTeX. O documento final foi gerado com o uso de MiKTeX 2.2 (SourceForge.net, 2003a). As figuras originais foram desenhadas em Corel Draw 10 da (Corel Corporation, 2003), e exportadas para o formato *Encapsulated PostScript*.

2 *Editores de Cenários*

Conforme mencionado anteriormente, três editores de cenários foram analisados antes do inicio da implementação da nossa primeira versão do Editor de prédios. Nesta seção são apresentados os resultados da análise.

Os três editores foram obtidos da página Web3D Consortium, página central de boa parte dos projetos de Realidade Virtual orientados à Web do mundo inteiro. Com estes editores é possível visualizar o "estado da arte" destes projetos e tentar usar suas idéias no nosso editor.

Foi constatada uma grande quantidade de editores de mundos de realidade virtual, mas apenas três deles atendiam os requisitos de

- exportar para VRML,
- serem de domínio público.

Os editores, analisados a seguir, são o AC3D Modeler, o CiteMap Builder e o Internet Space Builder.

2.1 AC3D Modeler

Este é um exemplo de um editor 3D, mas completamente diferente daquilo que queremos. Ele é um editor de objetos simples e não de cenários, sua interface tem como objetivo criar objetos 3D a partir de primitivas como esferas, cubos e cilindros, e possibilitar a criação de objetos complexos por meio de extrusões e da definição de conjuntos de vértices. Esse editor não permite o mapeamento de texturas nos objetos construídos, estando disponível na referência AC3D Modeler.

Apesar de não corresponder ao nosso tipo de aplicação 3D, a ferramenta possui recursos interessantes como a visualização em várias janelas ativas. Esse paradigma de visualização foi adotado para o nosso projeto.

A Figura 1 mostra um exemplo de uma imagem do AC3D Modeler. É interessante observar a maneira como são dispostas as janelas ativas da aplicação, cada uma com uma visão de um ângulo diferente fornecendo, assim, uma quantidade maior de informações visuais ao usuário.

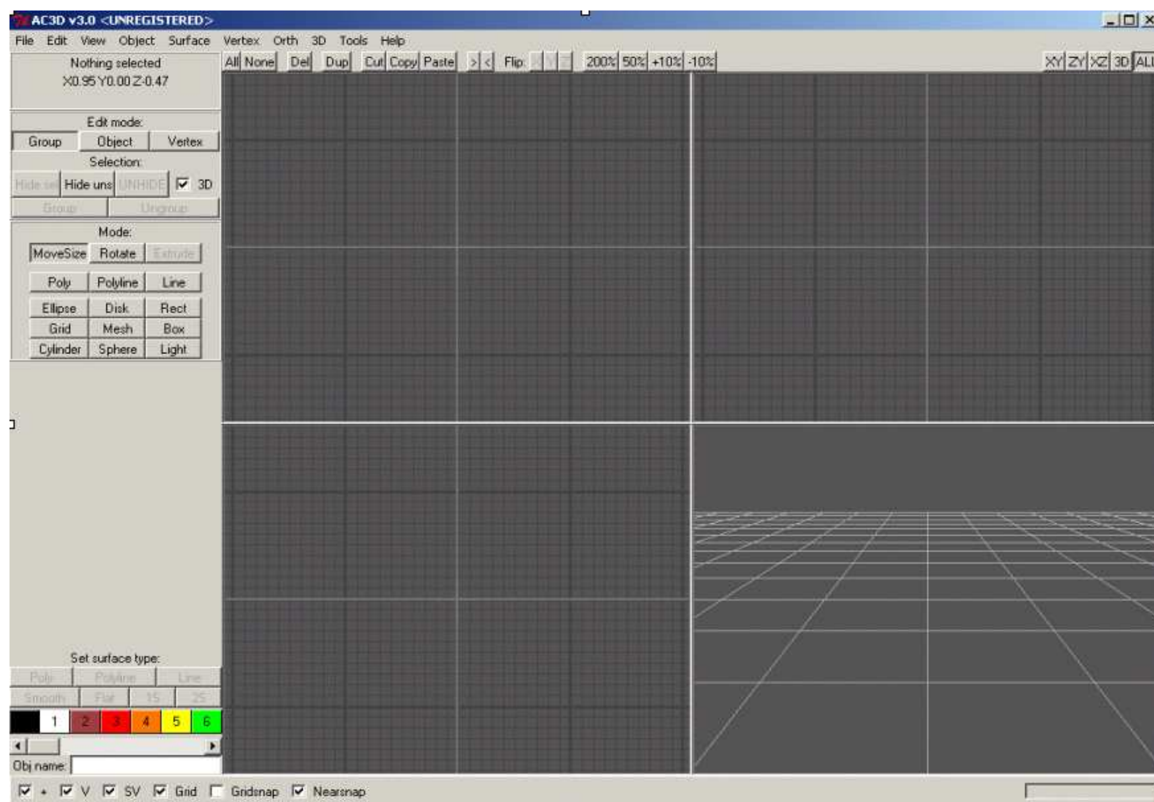


Figura 1: Interface do AC3D Modeler.

2.2 CiteMap Builder

Esta ferramenta é de fato a menos poderosa dentre as estudadas durante a pesquisa. No entanto, ela foi mencionada como uma aplicação bem resolvida, com interface simples, mas capaz de produzir resultados satisfatórios.

O CiteMap Builder é uma ferramenta desenvolvida em Java. Sua interface oferece uma visão isométrica[1] do cenário final. Esta visão é simplificada já que a qualidade visual é baixa, pois não possui nenhum componente 3D em sua interface. A citada ferramenta não dá suporte à importação de outros objetos definidos fora do editor, restringindo a

[1]Visão Isométrica: Visão localizada em uma perspectiva ou direção da câmera fixa em relação a um determinado objeto ou ambiente. Normalmente esta visão têm a característica de visualizar grandes ambientes de uma **visão superior**. Por suas características peculiares, uma visão isométrica tanto pode ser simulada em um ambiente 3D quanto por uma seqüencia de imagens 2D devidamente arranjadas ou animadas.

utilização aos objetos pré-definidos além de limitar-se a inserir objetos em uma grade isométrica de tamanho variável.

Nesse editor, apenas o resultado final é um ambiente 3D, na verdade, uma cena descrita em VRML.

A Figura 2 mostra a visualização do referido editor com a grade isométrica na posição central e com os objetos pré-definidos na barra lateral.

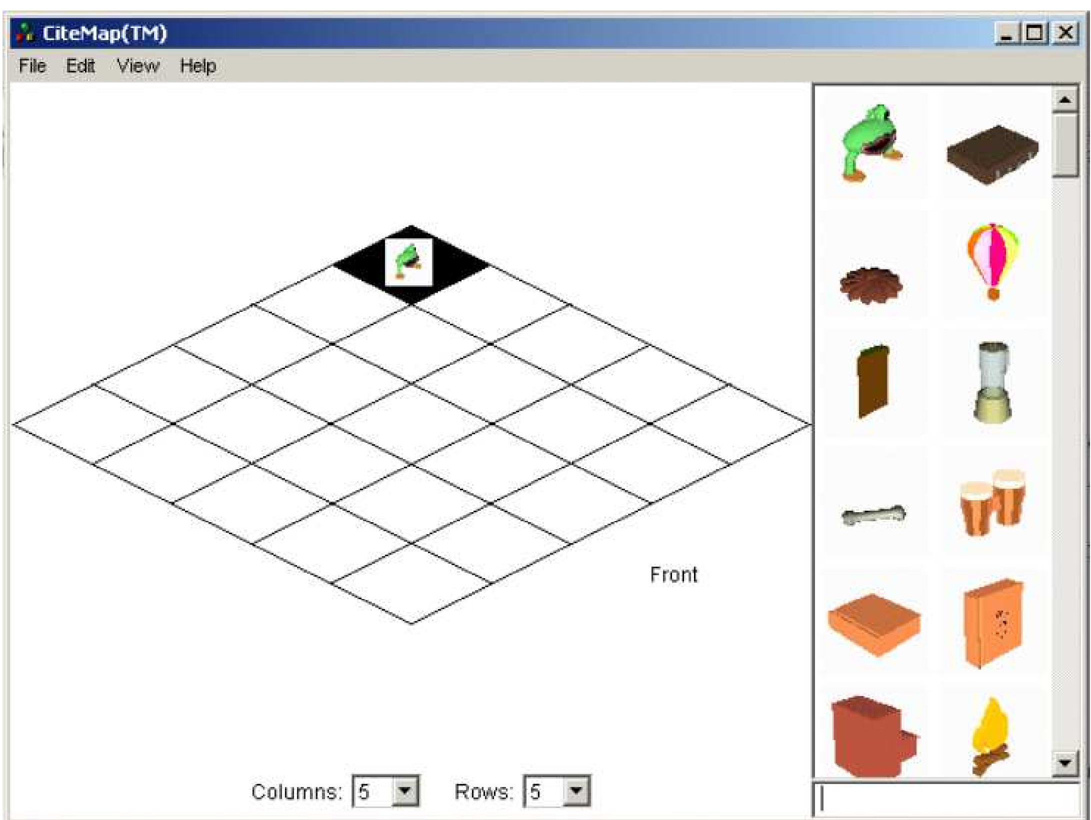


Figura 2: Interface do CiteMap Builder.

2.3 Internet Space Builder

É a mais completa das ferramentas estudadas para o nosso propósito específico, e lembra muito editores de cenários conhecidos para jogos de computador, só que seu objetivo é gerar mundos VRML.

O Internet Space Builder foi desenvolvido em C++ (STROUSTRUP, 1997), e permite a construção de cenários VRML bastante complexos. O editor possui funções intuitivas e avançadas para mapeamento de texturas, criação de formas complexas por meio de adição e subtração de sólidos, edição de imagens e modificação da posição da câmera para navegação e edição do cenário. Muitas das idéias presentes nesse navegador, como

interação para navegação e adicionamento de texturas, serão usadas no nosso editor de cenários. No entanto, certas funções para criação de sólidos mais complexos não são de nosso interesse, pois não fazem parte do escopo deste projeto.

Esse editor, apesar de bem mais completo que o do presente trabalho, pode ser visto como um referencial e tem muitas idéias que foram ou podem ser aproveitadas no nosso editor.

A Figura 3 apresenta o citado editor com um sólido, que é uma casa, totalmente construído nele. Observe a quantidade de ferramentas disponíveis no software, muitas das quais são avançadas demais para nosso escopo.

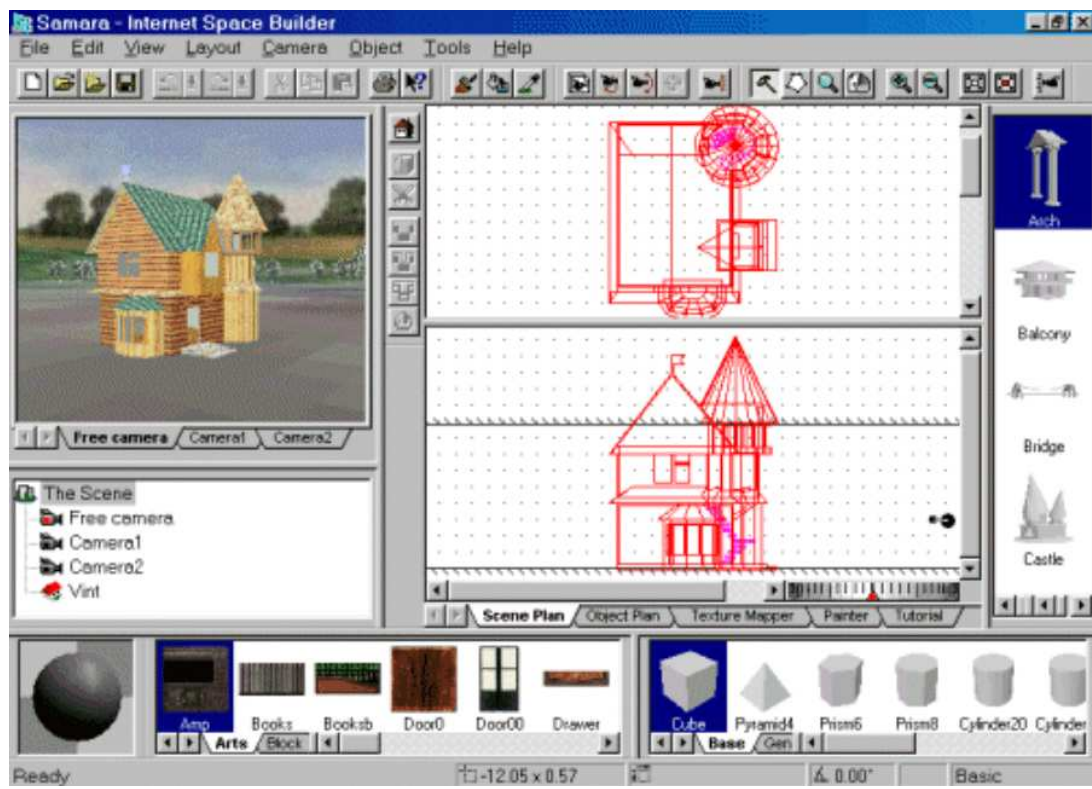


Figura 3: Interface do Internet Space Builder.

Com um estudo de caso já feito, faremos uma descrição detalhada, no próximo capítulo, da parte mais baixo nível do nosso editor, o seu motor. Esse motor será uma componente muito importante do nosso projeto por servir de base para os dois editores de cenários desenvolvidos.

3 O Editor como uma série de componentes reutilizáveis

Nesta seção daremos uma visão de como os editores aqui propostos e desenvolvidos são organizados, bem como das bibliotecas ou pacotes que foram utilizados e definidos na construção do editor. Desta maneira, temos aqui as principais componentes reunidas e como elas se comunicam entre si.

Começaremos então enumerando as componentes envolvidas no projeto do primeiro editor, o Editor de Prédios (*Building Editor*), Figura 4, e do segundo e último editor, o Editor de Cidades (*City Editor*), Figura 5. Em ambos os casos, as setas significam que componentes estão utilizando serviços de outra componente, sendo a origem da seta a componente usuária e o destino a componente que oferece o serviço.

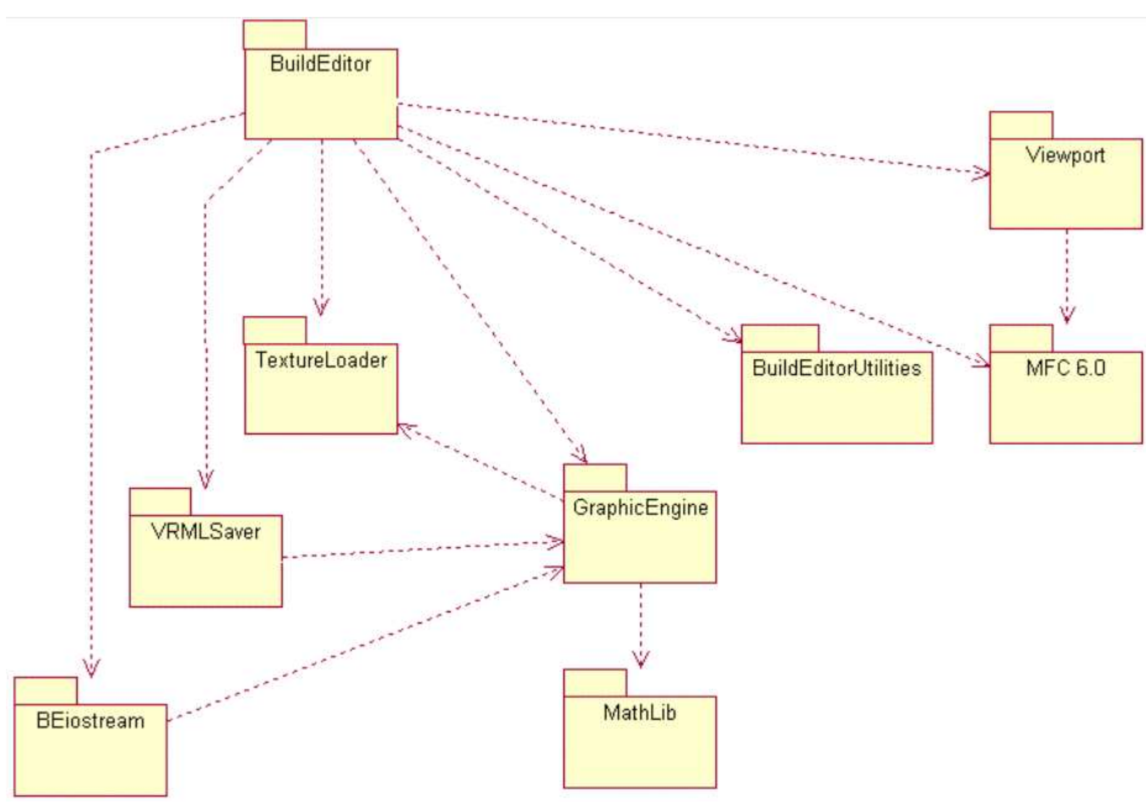


Figura 4: Bibliotecas envolvidas no Editor de Prédios.

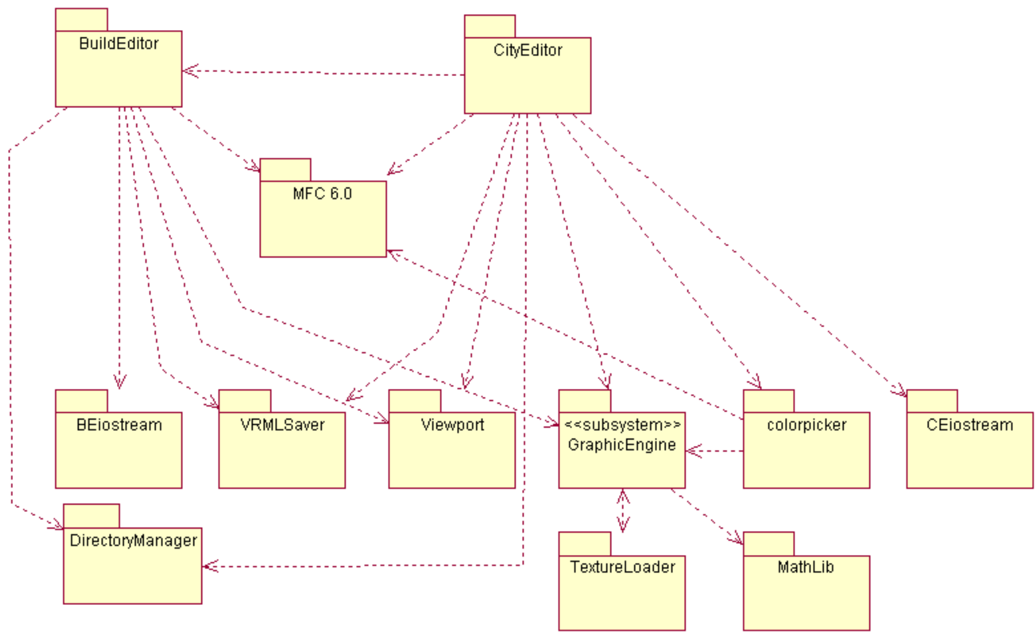


Figura 5: Bibliotecas envolvidas no Editor de Cidades.

Cada uma dessas bibliotecas é um projeto[1] dentro do Visual Studio.

As classes, no nosso caso, são agrupadas em projetos para facilitar a reutilização de componentes em versões do editor, e para organizar o código fonte de maneira intuitiva. Este tipo de organização dentro do compilador divide o código fonte em arquivos de extensão `.lib`[2] e `.dll`[3] o que reduz a carga e a complexidade associada à criação de um único arquivo executável padrão (extensão `.exe`).

No processo de criação de bibliotecas encontrou-se vários desafios intrínsecos à programação em plataforma Windows. O primeiro foi referente ao tipo de biblioteca que seria gerada: se uma biblioteca de ligação dinâmica (`dll`) ou uma biblioteca de ligação estática (`lib`). Escolheu-se a segunda porque na parte de gerenciamento de objetos teríamos que ter uma única instância desse objeto para cada aplicação que utilizasse o editor. Isto não é possível com o uso de `dlls`, já que neste caso existiria uma única instância da biblioteca rodando e que seria ligada com a aplicação dinamicamente. Com o uso de `libs` podemos contar com mais de uma instância de gerenciadores de objetos ao preço de uma aplicação

[1]Projetos são unidades de agrupamento de arquivos de cabeçalho (*header files* com extensão `.h`) com arquivos fonte (com extensão `.cpp`) dentro Visual Studio. Um projeto pode ter outros projetos incorporados, uma estrutura mais intuitiva e complexa de dependência entre essas unidades funcionais.

[2]Biblioteca de ligação estática – com o uso deste tipo de biblioteca a aplicação se torna mais pesada, pois seu conteúdo é carregado e duplicado em memória a cada vez que a aplicação é chamada. Além disso, a aplicação se torna maior com o uso de `libs` dentro do projeto.

[3]Bibliotecas de ligação dinâmica – carregada uma única vez em memória e compartilhada por todos os arquivos que a usam por meio de ligação dinâmica entre a aplicação e a biblioteca. Tem um contador de referências que permite descarregá-la quando nenhuma aplicação está mais interessada nela.

um pouco mais pesada.

Outro desafio bem interessante foi por conta da link-edição da aplicação final. Mesmo que as aplicações compilassem completamente em separado e em conjunto, durante o processo de link-edição havia choque entre bibliotecas padrão. Buscando mais a fundo informações dentro da biblioteca MSDN (MSDN Home, 2002) viu-se que as bibliotecas que se chocavam tinham relação com a escrita em arquivos, e que existiam versões de implementações diferentes de uma mesma biblioteca causando conflito. Descobriu-se posteriormente que este tipo de choque só existiria se aplicações fossem definidas como sistema de utilização de *Threads* diferentes, ou seja, definindo uma biblioteca como multi-thread e outra como single-thread. Dado que isso ocorreu, as bibliotecas foram compatibilizadas e tudo transcorreu normalmente.

A Tabela 1 apresenta a definição sucinta de cada uma das bibliotecas definidas no nosso editor com sua funcionalidade básica. Não entraremos aqui, no entanto, em maiores detalhes já que isso será feito nas seções seguintes.

Tabela 1: Descrição das bibliotecas definidas pelo Building Editor e pelo City Editor.

BuildEditor	Componente mais alto nível do *Building Editor*. É a parte do projeto que engloba o editor de prédios propriamente dito, é a única componente dentro do Building Editor feita em programação MFC para Windows e utiliza os pacotes descritos na Figura 4.
CityEditor	Componente mais alto nível do *City Editor*. Utiliza os pacotes definidos a seguir agrupando-os e utilizando seus serviços na versão final. É a parte do projeto que contém editor de cenários propriamente dito; é a única componente dentro do *City Editor* feita em programação MFC para Windows e utiliza os pacotes descritos a seguir.
Graphic Engine	Motor Inspirado no motor de Curso OpenGL do Game Institute, totalmente refeito para se adequar às aplicações necessárias, como suporte ao uso de estruturas OpenGL de renderização de hardware.
BuildEditorUtilities	Define as estruturas mais comuns de manipulação, por exemplo, para navegação em diretórios e para tratar conversão de caminhos absolutos para caminhos relativos. Oferece também mecanismos de escrita em arquivo de um *string* padrão C++ delimitado por aspas duplas.

continua na próxima página

continuando a página anterior	
BEiostream	Usada para leitura e escrita de informações no arquivo de formato padrão definido no projeto do *Building Editor*. Ela é totalmente integrada com o motor por meio de uma estrutura chamada *Importer*.
CEiostream	Usada para leitura e escrita de informações no arquivo que define uma cena do *City Editor*. Ela é totalmente integrada com o Modelo da cidade (classe `CModel`) por meio de uma estrutura chamada *CEImporter*.
VRMLSaver	Usada para salvar arquivos em formato VRML. Exigiu que boa parte dos nós VRML fossem modelados para ter um *parser* poderoso e compatível com futuras versões de editores. Ela é também integrada com o motor, só que por meio de uma estrutura interessada apenas na escrita dos arquivos.
TextureLoader	Responsável pelo carregamento de texturas nos formatos `BMP`, `PNG`, `JPEG` e `TGA` (MURRAY; RYPER, 1994). Ela é compatível com qualquer versão do sistema operacional Windows desde o Windows 95 até o momento.
Viewport	Usada como base para a criação de múltiplas viewports. Com isso oferece suporte a dispositivos de navegação, além de tratar de questões padrão de criação de ambientes OpenGL em MFC.
MathLib	Biblioteca de operações matemáticas retirada do Motor (Game Institute, 2002), não será muito abordada aqui neste documento por conta de sua pouca utilização no contexto deste trabalho.

4 O Motor Gráfico

Sem dúvida, o Motor Gráfico é a componente mais importante deste projeto. O referido motor é responsável por todo o processo de visualização (*renderização*, no jargão de computação gráfica). Na verdade, essa componente é uma estrutura que encapsula os comandos OpenGL padrão oferecidos, dando à programação um formato mais orientado a objetos. Tenta-se preservar, dentro do possível, no processo de construção de um motor a flexibilidade que o formato natural de OpenGL oferece. Algumas vezes, contudo, essa flexibilidade é comprometida por conta de algumas restrições impostas pelo paradigma de orientação a objetos. Por ser uma estrutura tão vital ara a aplicação final, o motor merece toda a atenção por parte do programador, e sua arquitetura deve tentar modelar os componentes mais genéricos oferecidos para a visualização, mais informações sobre com fazer um motor genérico estão disponíveis em Döllner e Hinrichs (2002).

Nesta seção descreveremos as estruturas mais importantes que foram realizadas no motor, e as otimizações que foram feitas em relação ao motor original obtido a partir de um curso on-line (Game Institute, 2002), discutiremos o porquê de algumas decisões tomadas. A figura presente no Apêndice A ilustra as interações e relacionamentos de todas as classes que compõem o motor.

A seguir descreveremos as principais componentes do motor, no intuito de ver como ele engloba as funções de OpenGL. Essas componentes serão divididas, para o melhor entendimento do contexto geral, nos seguintes grupos:

- Estruturas Básicas
- Objetos Principais
- Managers e IDFactory
- Estrutura de Importação de Mechas

4.1 Estruturas Básicas

Nesta seção trataremos das estruturas e objetos mais simples do Motor, objetos que, apesar de importantes, darão suporte às mais simples operações e que são facilmente entendidos no contexto individual. Objetos extremamente simplificados como Allocators, estruturas de dados da biblioteca padrão C++ que agrupam os objetos provendo mecanismos de acesso individual, não serão analisados aqui mas estão presentes no diagrama do Apêndice A.

As classes e estruturas mais relevantes desta categoria são mostradas na Tabela 2.

Tabela 2: As classes e estruturas mais simples.

OBJECT_ID	Identificador único do Objeto. Atualmente definido como um inteiro de 32 bits. Necessário nos *Managers* para permitir que somente uma instância de um determinado objeto com o mesmo ID exista.
COBJECTTYPE	Define o tipo do objeto do motor. Entre os tipos disponíveis podemos citar: • `UNDEFINED_TYPE` • `BUILDING` • `TEXTURE` • `MESH` • `MATERIAL` • `ROAD`

continua na próxima página

continuando a página anterior	
CBUILDINGTYPE	Responsável pela definição do tipo do prédio residencial ou comercial.
BELIGHTTYPE	Define qual o tipo da luz empregado: pontual (Point Light), de difusão cônica (Spot Light) ou direcional (Directional Light) (FOLEY, 1995; HILL JR., 2000).
CCamera	Oferece mecanismos necessários para a manipulação de câmeras. Em OpenGL as transformações de câmera e geométricas são representadas por uma matriz única, o que faz com que as transformações de câmera sejam a inversa das transformações geométricas. Esta classe possui os atributos GIGLVector, m_Position, m_Target e m_Up que são, respectivamente, o vetor posição, o vetor que mostra para onde a câmera está olhando e o vetor que aponta perpendicularmente para cima da câmera.
CColor	Vetor para representação das cores em quatro componentes de ponto flutuante (RGBA), síntese aditiva das componentes vermelha (*Red*), verde (*Green*), azul (*Blue*) e a componente Alpha que representa a transparência da cor.
CFrame	Responsável pela aplicação de transformações geométricas, neste caso somente translações e rotações são permitidas.
CMESHTYPE	Estrutura que define se a malha (ou mecha), tipo que será definido na Seção 4.2.5. É uma estrutura de agrupamento, uma instância de outra malha, ou se é uma malha folha.
GIGLVector	Vetor tridimensional padrão. Oferece estruturas para construção a partir de um número que é um ponto flutuante com precisão simples. Possui um método para que seja instanciado como um ponteiro para inteiro. Importantíssimo para a utilização de métodos OpenGL. Este vetor foi definido pelo motor original.
CTextCoord	Muito parecido com o GIGLVector, exceto pelo fato de possuir duas dimensões. Esta classe é responsável pelas coordenadas de texturas a serem mapeadas. Também está presente no motor original.

Algumas destas estruturas básicas, mais que meras ferramentas de programação, refletem na verdade os desejos ou objetivos que o usuário teria em mente ao usar as nossas ferramentas, os editores de cenários. Um exemplo disso são os tipos `CBUILDINGTYPE` e `BELIGHTTYPE` que definem se um prédio é residencial ou comercial e os tipos de luzes que serão usadas na aplicação.

Nem todos objetos discutidos são classes, alguns deles são de tipo enumerado de dados

e por isso não tem atributos. Para mais informações sobre os objetos que possuem algum tipo de codificação extra, o Apêndice B exibe suas interfaces por meio de diagramas de classes.

Nesta seção não ficaremos nos detendo a olhar alguns métodos das classes que tratam de renderização OpenGL. Esses métodos serão vistos nas seções seguintes conforme forem necessários.

Outro componente que define operações importantes de vetores é o pacote `MathLib` que só é referenciado pela biblioteca `GraphicEngine` (ver Figura 5 na página 27). Essa biblioteca possui operações comuns de vetores de duas, três e quatro dimensões como soma, subtração, multiplicação, produto interno e produto vetorial, além de operações para resolução de equações polinomiais e operações matriciais. Entretanto, a referida biblioteca praticamente não foi utilizada durante o projeto e, por isso, não será abordada aqui, porém suas classes estão retratadas no Apêndice C - Diagrama de Classes da Biblioteca `MathLib`.

4.2 Objetos Principais

Agora que temos suporte aos tipos básicos como cores, vetores de variadas dimensões e coordenadas de textura; podemos dar a devida atenção aos objetos mais relevantes do Motor. A parte dentro do contexto de orientação a objetos que iremos mostrar agora é uma estrutura simples que engloba as funcionalidades disponíveis no Driver OpenGL.

Seguiremos então com a abordagem dos casos mais interessantes. Os casos serão estudados na seguinte ordem:

1. Objeto Padrão
2. Luzes
3. Materiais
4. Texturas
5. Mechas
6. Prédios
7. Objetos de Estado

Os ítens citados serão estudados apenas de uma maneira simplificada. No entanto, exibiremos e comentaremos partes de código sempre que necessário. Nos Apêndices A e B o leitor encontrará informações que lhe permitirão analisar o escopo destes componentes e a maneira como eles se relacionam em um contexto geral.

4.2.1 Objeto Padrão

O objeto padrão, no nosso caso a classe CBuildEditorObject (ver Apêndice A), encarrega-se de ser uma componente que oferece uma *interface* comum a todos os objetos manipulados do Motor. Portanto, os mais complexos objetos do Motor herdam dessa classe.

Por trás deste objeto base está o fato de cada objeto do motor ter um OBJECT_ID e um *nome* que serão usados para identificá-lo na interface gráfica, por meio de cliques por exemplo ou por meio de estruturas como combo-boxes. Estes identificadores também são utilizados nos *Managers* (melhor explicados na Seção 4.3.3) para facilitar o manuseamento dos objetos. Além disso, este objeto tem um conjunto de *métodos virtuais* que torna possível, por exemplo, a implementação de mecanismos de aceleração por hardware dentro de OpenGL como *Display Lists*.

Display List é uma maneira de codificar uma seqüência de comandos OpenGL em um único comando, como seria uma macro de linguagem de programação. Essa seqüência de comandos pode ser tratada pelo driver OpenGL ou por hardware específico da placa de vídeo, de tal maneira que instruções redundantes ou que podem ser sintetizadas como uma única são otimizadas. Nas placas de hoje, o uso desse comando é ainda mais relevante, informações de vértices são passadas todas de uma só vez, o que nas placas de hoje é muito relevante já que elas podem tratar milhares de polígonos enquanto paralelamente outras operações de software são executadas pelo sistema.

Código 1: Estrutura do Objeto Base do Motor.

```
...
class CBuildEditorObject  {
    public:
        virtual void GenObject();
        virtual void Unload();
        virtual void Load();
        OBJECTTYPE GetObjectType() const;
        OBJECT_ID GetObjectID() const;
        string GetName() const;
        void SetObjectType(OBJECTTYPE nType);
        void SetObjectID(OBJECT_ID nID);
        virtual void SetName(const string & Name);
        virtual void Destroy() = 0;
        CBuildEditorObject();
        virtual ~CBuildEditorObject();
    private:
        OBJECT_ID m_ObjectID;
        OBJECTTYPE m_ObjectType;
        string m_Name;
    };
...
```

O trecho presente no Código 1 deixa claro que certos métodos podem ser implementados por outros objetos. No caso, os métodos Load/Unload e GenObject/Destroy são métodos que podem ser usados tanto na criação de Display Lists, quanto no mecanismo

Gerenciador de Textura padrão de OpenGL. Poderemos mostrar melhor o uso destes métodos nas seções seguintes.

4.2.2 Luzes

As luzes são o item mais importante na impressão de profundidade em ambientes 3D. No mundo real, o efeito da luz faz os objetos ficarem mais claros ou escuros. Por conta disso, o olho humano está adaptado a vincular variações de sombreamento dos objetos com a presença de luz.

OpenGL aproveita-se deste fato para construção de ambientes iluminados. Quando inserimos luzes em um ambiente, OpenGL não se preocupa em calcular o efeito de sombra de um objeto em outro. De fato, luz em OpenGL implica variação na cor de um objeto de acordo com a posição da luz. Na Figura 6 temos um exemplo dos efeitos da ausência e presença da luz no realismo de um ambiente 3D.

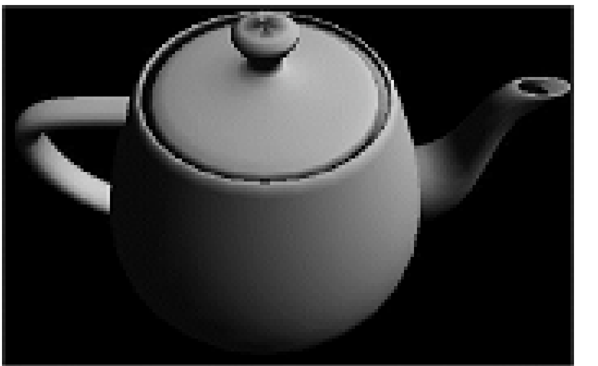


Figura 6: Imagem de um mesmo objeto com e sem iluminação.

O modo como a luz suaviza a superfície do bule, escondendo os polígonos que o formam, serve também para provar que esta etapa do processo de renderização é muito importante no realismo da aplicação final conseguida.

Para calcular o efeito das luzes em uma determinada superfície é necessário que haja definição de vetores normais, aqui chamados simplesmente de “normais”, à superfície a ser renderizada. Através das normais pode-se ter idéia se a superfície está virada de frente para a fonte de luz ou não, permitindo que quanto mais ela estiver de frente para a luz mais clara ela fique. Essas normais devem ser calculadas para cada elemento que descreve a superfície, em geral triângulos, e esse cômputo pode ser feito para cada vértice ou para cada face redundando em algoritmos de diferentes complexidade e realismo dos resultados obtidos (FOLEY, 1995; HILL JR., 2000). Na Figura 7 temos um exemplo de normais calculadas por faces (esquerda) e e por vértices (direita).

As normais por vértices dão um melhor resultado, pois produzem variações de ilu-

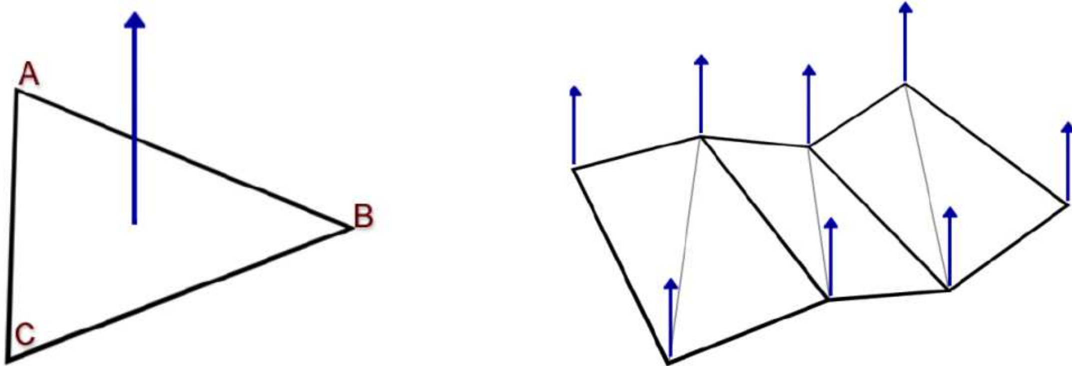


Figura 7: Uma normal por face e uma normal por vértice. FONTE: Game Institute (2002)

minação interpolando as cores disponíveis em cada vértice. As normais por face dão uma aparência mais lascada e discreta, permitindo que o usuário distinga as arestas e, consecutivamente, os polígonos que formam a superfície.

Na Figura 8 temos um exemplo de uma esfera com normais definidas para cada face (esquerda), com os polígonos perceptíveis. Na mesma figura, à direita, temos a mesma esfera com as normais calculadas por vértices e onde é aplicada uma variação de cor em cada vértice de modo a ter um efeito de sombreamento mais realista.

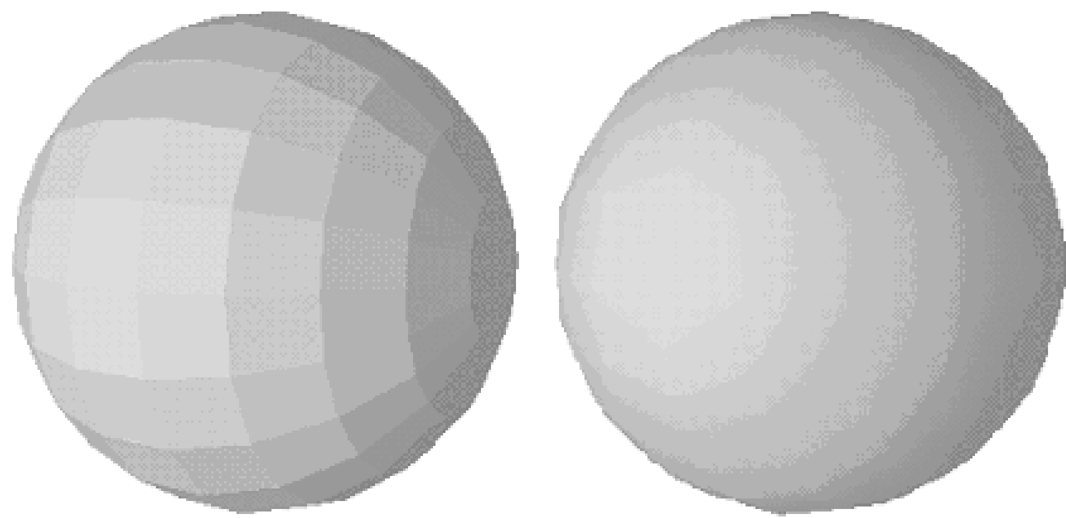


Figura 8: Esfera com uma normal por face e com normal por vértice.

Para modelar luzes temos a classe `CLight`. Esta classe é uma estrutura que engloba a criação dos mais diversos tipos de luzes. A classe possui métodos para criação de luz pontual (*Point Light*), luz de difusão cônica (*Spot Light*) e luz direcional (*Directional Light*). Na Figura 9, temos uma amostra de como se comportam estes tipos de emissão de luz.

Além disso, a classe `CLight` é responsável também por escolher o tipo de luz que

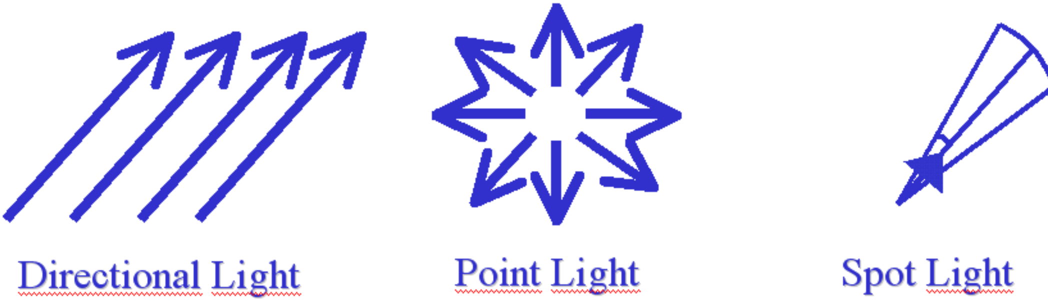


Figura 9: Tipos de emissão de luzes.

emana da fonte. Entre os tipos de luzes podemos enumerar:

Luz Ambiente: representa uma "luz onipresente" que possui a mesma intensidade em todo ponto e em toda direção da cena. Essa é a luz refletida por todos os objetos em um ambiente razoavelmente fechado.

Luz Difusa: é uma luz que ilumina, de uma direção particular, o objeto. Um exemplo desse tipo de luz é a luz do sol que, por estar tão distante, aparenta estar vindo de uma direção só de forma paralela.

Luz Emissiva: luz que emana do objeto propriamente dito.

Luz Especular: responsável pela luz reflexiva vista em superfícies altamente polidas como metais, por exemplo.

No Código 2 temos o escopo da classe encontrado no arquivo de cabeçalho. Nele podemos ter idéia da complexidade envolvida na manipulação da luz. É conveniente relembrar que este pedaço do código de iluminação foi desenvolvido no motor original, e que a parte OpenGL do código não será abordada aqui por conta da complexidade envolvida.

Observe que têm métodos para modificar todos os parâmetros discutidos até então. Uma vez modificados todos esses parâmetros, basta chamar o método `Enable()` para habilitar a iluminação e o método `PrepareRenderer()` para criar a luz em questão.

Para maiores informações sobre conceitos de iluminação gerais e iluminação em OpenGL veja as referências (FOLEY, 1995; FRERY, 2002; HELIUM, 2001; WOO, 1999; WRIGHT JR; SWEET, 1999; HILL JR., 2000).

4.2.3 Materiais

Os materiais são responsáveis pelas características de cor e reflexão de luz de um objeto. Contudo, a maneira como OpenGL obtém este efeito não é muito intuitiva. Para

Código 2: Estruturas definidas para implementação de luz.

```
...
enum BELIGHTTYPE{
    LIGHTTYPE_POINT,
    LIGHTTYPE_SPOT,
    LIGHTTYPE_DIRECTIONAL,
};
...
class CLight {
public:
    CLight(BELIGHTTYPE Type = LIGHTTYPE_DIRECTIONAL);
    void SetPosition(const GIGLVector & Position);
    void SetDirection(const GIGLVector & Direction);
    void SetAmbient(const CColor & Ambient);
    void SetDiffuse(const CColor & Diffuse);
    void SetSpecular(const CColor & Specular);
    void SetSpotParams(float Exponent, float Cutoff);
    void SetAttenuation(float Constant, float Linear, float Quadratic);
    void PrepareRenderer(LIGHTIDX LightIdx);
    void Enable()
    void Disable();
    bool IsEnabled();
private:... // atributos e métodos privados
```

entendermos melhor as componentes do nosso material, vejamos as partes envolvidas na definição do Material:

Cor ambiente e Cor difusa: Modulam respectivamente a cor da luz ambiente e difusa, de tal maneira que juntas formam a cor com que percebemos o objeto final. No caso desta aplicação, apesar do motor ter sido adaptado para suportar as duas cores separadamente, o padrão é escolher os valores dessas duas cores como sendo um só. Isso facilita a usabilidade da ferramenta final.

Cor emissiva: É a cor que é independente das luzes envolvidas, ela é originária da superfície do material por si só e não afeta outras superfícies.

Cor Especular: Modula a luz reflexiva. Como no mundo real, a propriedade especular é de reflexão da cor; normalmente essa cor é o branco total para habilitar a reflexão especular, ou preto para desabilitá-la por completo.

Brilho: trata quão fortemente uma cor especular se condensa na superfície do objeto, dando uma aparência mais metálica a ele.

No Código 3 temos o trecho do código de renderização do material, esse código se encontra na classe `CMaterial`. Observe que também há preocupação com a transparência dos materiais selecionados; essa transparência é feita por meio do `Blend` da cor origem disponível no fragmento inicial com a cor final do material.

O *blending* em programação 3D é o processo que mistura a cor do fragmento (que será um pixel) atual com alguma cor restante. Essa mistura pode ser linear, como é o caso

do efeito de transparência, ou quadrática, para obter efeitos como radiosidade (FOLEY, 1995; HILL JR., 2000).

Código 3: Processo de renderização de um material.

```
void CMaterial::PrepareRenderer(){
    CColor Diffuse = m_Diffuse;
    glEnable(GL_BLEND);
    Diffuse.a = m_Opacity;
    if (m_Opacity != 1.0f)
    {
        glDepthMask(GL_FALSE);
    }
    else
    {
        glDepthMask(GL_TRUE);
    }
    glMaterialfv(GL_FRONT, GL_AMBIENT, m_Ambient.GetGLfloatPtr());
    glMaterialfv(GL_FRONT, GL_DIFFUSE, m_Diffuse.GetGLfloatPtr());
    glMaterialfv(GL_FRONT, GL_SPECULAR, m_Specular.GetGLfloatPtr());
    glMaterialfv(GL_FRONT, GL_EMISSION, m_Emissive.GetGLfloatPtr());
    glMaterialf(GL_FRONT, GL_SHININESS, m_Shininess);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}
```

Esta classe também possui métodos `GenObject/Destroy` herdados da classe `CBuildditorObject`, usados para a criação e destruição de Display Lists respectivamente.

4.2.4 Texturas

Esta seção será dividida em duas partes: a primeira ensinando os mecanismos feitos para englobar as técnicas usadas para renderizar texturas em OpenGL; a segunda englobando mecanismos para gerenciar texturas na memória em OpenGL.

4.2.4.1 Lendo e Utilizando Texturas

Texturas, no contexto deste trabalho, são imagens que cobrem a superfície de objetos dando uma aparência mais real aos mesmos. Elas têm a mesma funcionalidade que pode ser observada nos bonequinhos de chumbo que eram feitos antigamente. A princípio, os bonecos eram opacos e sem vida e sua semelhança era apenas baseada na proximidade da forma física com a aparência de um soldado real. Mas, com o passar do tempo, esses bonecos foram pintados para assumir uma aparência mais condizente com o objeto real sendo representado. As texturas em ambientes 3D têm a mesma funcionalidade em objetos 3D da tinta no caso do boneco de chumbo.

Mas temos um problema, a textura é por si só um arranjo retangular de dados de cores da imagem como qualquer outro mapa de bits bidimensional. Assim sendo, como cobrir um objeto tridimensional com uma superfície? A solução está no uso de coordenadas de textura, que são funções que mapeiam os pontos dentro de uma textura em

vértices da superfície. Com o uso de coordenadas de textura, a textura (uma imagem) é deformada para que a sua área preencha os polígonos que formam a superfície do objeto completamente.

Tecnicamente, se $f\colon S \longrightarrow \mathcal{C}$ é a textura, onde $S = \{0,\ldots,m-1\} \times \{0,\ldots,n-1\}$ é o suporte da imagem e $\mathcal{C}$ é o espaço de cores empregado, e o sólido a ser ' forrado' ou 'embrulhado' pela textura possui uma superfície caracterizada pela função $h\colon \mathbb{R}^2 \longrightarrow \mathbb{R}^3$ então 'coordenadas de textura' são funções $\Upsilon\colon H \longrightarrow S$ que para cada ponto da superfície h devolvem um ponto de S, e onde $H \subset \mathbb{R}^3$ é o contradomínio da função h. Para mais informações sobre coordenadas de textura consultar uma das referências (FOLEY, 1995; ZONENSCHEIN, 1998; WRIGHT JR; SWEET, 1999; WOO, 1999; HILL JR., 2000).

Logo, temos desde já, dois papéis para o nosso objeto textura: o de carregar texturas a partir de imagens e o de mapear essa imagem em um objeto 3D por meio de coordenadas de textura.

Quanto ao papel de ler texturas de arquivos, foi procurada uma biblioteca compatível com todas as versões do Windows desde o Windows 95. Foram achadas diversas bibliotecas dependentes de DirectX, o que deixava muito a desejar dada a falta de compatibilidade de DirectX com o Windows NT, ainda largamente usado. Finalmente achou-se em New Page 1 (2001) um pequeno código que faz uso de duas `dlls`, disponíveis gratuitamente, para carregamento de arquivos nos formatos PNG (Libpng Home Page) e JPEG (Intel® JPEG Library); além do fato de já termos funções baseadas em bibliotecas padrão do Windows para carregamento de arquivos BMP e TGA. Mesmo que essa fonte estivesse disponível, alguns erros foram encontrados e tratados, e aquilo que era apenas um arquivo se tornou a biblioteca `TextureLoader`.

No entanto, as coisas em OpenGL não são tão simples... não basta apenas saber ler os mais diversos formatos de arquivo e depois renderizar. Antes disso, é necessário informar a OpenGL que a textura existe e carregá-la na memória de vídeo. Estas operações podem ser realizadas por meio de métodos padrão de OpenGL como o `glTexImage2D`, onde se passa um ponteiro para um *array* de caracteres que são os pixels da textura; além de informações como o número de bytes por pixel, o tipo de pixel, se é um pixel RGB ou um pixel RGBA etc.

Porém, esses métodos padrão têm limitações; o método `glTexImage2D` só suporta imagens 2D cujas dimensões sejam potência de dois, isso para tornar mais rápidos certos mecanismos de hardware e software. Esta limitação é muito restritiva ao nosso editor havendo, em princípio, três possibilidades de solução para o problema:

1. Exigir que o usuário só usasse texturas que são potência de dois; essa a princípio, seria minha opção preferida `:)`.

2. Completar a textura com dados brancos e definir as coordenadas da textura apropriadamente, tal que o usuário não visse os espaços em branco. Por exemplo: se fosse necessário renderizar uma textura de tamanho 80×80, então o programa editaria a textura em tempo de leitura para uma textura de 128×128 multiplicando, a seguir, todas as coordenadas de textura padrão por $\frac{80}{128}$, tal que a área em branco que foi adicionada nunca seja visível.

3. Deformar a imagem para que ela caiba em uma imagem potência de dois. Essa deformação pode ser feita por meio de um algoritmo próprio, ou pelas funções auxiliares `gluBuld2DMipmaps` e `gluScaleImage`.

A última opção foi a escolhida. Porém, existe um motivo peculiar que nos fez escolher esta opção. O fato é que existem vários defeitos que podem ser percebidos na visualização de uma textura muito de perto ou muito de longe. Isso decorre do fato da correspondência pixel do monitor-pixel da textura nem sempre ser de um para um.

Neste caso, podem ocorrer dois problemas conhecidos:

Texture Blockiness: Problema observado quando se aplica um zoom de aproximação (ampliação) em uma textura relativamente pequena. Uma textura 16×16 sendo aumentada vai parecer uma serie quadrados discretizados, tornando visível os pixels individuais.

Texture Simmering: Ocorre quando uma textura de tamanho 256×256 é reduzida, por exemplo, para uma textura de tamanho 8×8. Quando a imagem não fica trêmula (daí o nome *shimmering*), ela parece com uma série de pixels sem sentido.

Mas existem soluções para esse problema em OpenGL. A primeira solução é chamada de mipmaping. Essa técnica consiste em definir vários níveis de textura de tal maneira que as texturas sejam ordenadas em ordem de detalhe decrescente. Com isso, reduz-se a possibilidade do problema de *texture shimering.* Em OpenGL, essas texturas têm que ser definidas em todas as potências de dois possíveis. Por exemplo, para uma textura de 64 por 64 teríamos que definir as seguintes variações da textura original: 32×32, 16×16, 8×8, 4×4, 2×2 e 1×1.

Essa definição pode ser feita por meio de trabalho braçal com imagens separadas ou por meio da função da biblioteca auxiliar de OpenGL (gluBuild2DMipmaps) que oferece suporte à criação automática de todos os mipmaps e, além do mais, contorna a barreira de requerimento das dimensões da imagem serem potência de dois. Tudo isso é feito usando um filtro no bitmap que está um nível acima: o filtro *Box filtering.* O Box Filtering é um algoritmo que pega quatro pixels da imagem original e cria um novo pixel do bitmap destino. A imagem final possui metade das dimensões do bitmap anterior, e cada novo pixel

é a média dos quatro pixels acima. Este tipo de processamento é conhecido como *técnicas multiescala*, constituindo-se em uma ferramenta muito empregada em processamento de imagens (JAIN, 1989).

A segunda solução para o problema são algumas funções que permitem a filtragem em OpenGL. Essas funções habilitam ou desabilitam a função de filtro linear já disponível no OpenGL padrão. A filtragem pode ser ligada/desligada para eventos de maximização e minimização da textura padrão. Mais informações sobre tipos de filtro estão disponíveis em (FOLEY, 1995; HILL JR., 2000).

E para terminar, temos que OpenGL permite que as duas técnicas sejam usadas em conjunto. Para isso, o argumento do filtro tem que usar um dos parâmetros descritos na Tabela 4.2.4.1.

Tabela 3: Modos de filtragem de uma textura para aproveitar mipmaping.

Filtragem da Textura com mipmaps	**Ação**
`GL_NEAREST_MIPMAP_NEAREST`	Seleciona o texel (pixel da textura original) mais próximo do mipmap mais propício para o mapeamento de um para um com os pixels do monitor.
`GL_LINEAR_MIPMAP_NEAREST`	Seleciona quatro texels no melhor mipmap e filtra para produzir a cor resultante.
`GL_NEAREST_MIPMAP_LINEAR`	Seleciona o texel mais próximo dos dois melhores mipmaps, e interpola-os linearmente para produzir a cor resultante.
`GL_LINEAR_MIPMAP_LINEAR`	Seleciona os quatro texels mais próximos, dos dois mipmaps mais próximos, filtra os dois conjuntos de texels, e depois interpola linearmente as cores resultantes.

Uma visão complementar das técnicas descritas na Tabela 4.2.4.1 pode ser vista, no contexto de processamento de imagens, nas referências (JAIN, 1989; MASCARENHAS; VELASCO, 1989).

Analisemos, no Código 4 , o resultado da classe final `CGLBMP` para utilizar uma textura em OpenGL, apesar desse código estar no pacote `TextureLoader` ele é de autoria do aluno.

No Código 4 temos primeiramente a geração de um identificador e a alocação do identificador da textura dentro de uma estrutura OpenGL pelos métodos `glGenTextures` e `glBindTexture`. Neste caso, `m_magFilter` é o filtro de maximização e será inicialmente igual a `GL_LINEAR_MIPMAP_LINEAR`, e `m_minFilter` é o filtro de minimização e também será inicialmente igual a `GL_LINEAR_MIPMAP_LINEAR`; `m_pData` é a imagem já lida do arquivo e disponível em forma de *array* de bytes. Depois de definidos os filtros o mipmap é gerado.

Código 4: Estrutura para gerar a textura dentro de uma janela OpenGL.

```
void CGLBMP::GenTexture(){
    if (texID)
        glDeleteTextures(1, &texID);
    glGenTextures(1, &texID);
    glBindTexture(GL_TEXTURE_2D, texID); // aloca o id da textura
    ... //atribui parametros de repeticao de textura para coordenadas
        // de textura maiores que 1
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, m_magFilter);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, m_minFilter);
    gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, m_width, m_height, GL_RGBA,   GL_UNSIGNED_BYTE,
                    m_pData);
}
```

Após esta estrutura ter sido definida, basta criar o método de renderização da classe `CTexture` que está mostrado no Código 5.

Código 5: Renderização da Textura usando Handler.

```
void CTexture::PrepareRenderer(){
    if(m_Width!=0 && m_Height!=0) {
        glEnable(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, m_GLTextureObject);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    }
}
```

Para ver melhor o escopo de outros métodos da classe `CTexture`, ou como o processo de texturização é feito em conjunto com os outros objetos do motor, o leitor pode consultar os Apêndices A e B.

4.2.4.2 Gerenciamento de Textura

Outro ponto importante que está intrínseco nesta estrutura de textura é que por trás dos métodos de textura tem que existir uma estrutura de gerenciamento. Por conta de limitações de tempo inerentes ao hardware, uma textura na memória de sistema leva muito tempo para passar pelo barramento e chegar na memória de vídeo. Por isso, é interessante que as texturas mais usadas estejam sempre que possível na memória de vídeo, enquanto as menos usadas fiquem na memória de sistema caso não haja mais espaço para elas na memória de vídeo.

Para tratar esse tipo de evento OpenGL possui na sua implementação mecanismos de otimização de carregamento de texturas. Eles são usados de maneira muito transparente, e não são perceptíveis a um usuário leigo. O primeiro é um mecanismo que funciona diferente do método `glTexImage2D`, que sempre carrega a imagem da textura da memória de sistema para a memória de vídeo, mas oferece uma abstração de utilização por meio de *handles*. Ele foi usado nos códigos fontes anteriores, mas, por ser tão simples, não foi percebido.

No lugar de a cada chamada fazer uma nova transferência de textura pelo barramento, poderíamos deixar esta textura em memória de sistema ou de vídeo, e criar mecanismos que utilizassem informações de chamada freqüentes da textura para decidir qual textura deixar em memória de vídeo. Neste sentido, OpenGL tem implementado um mecanismo que lembra muito uma fila de *Round-Robin,* onde as texturas mais utilizadas ficam na memória de mais alta prioridade, no caso a memória de video. Para que este mecanismo funcione corretamente, o método glTexImage2D não deve ser usado, por isso não o usamos. Devem ser usados, mecanismos que permitam o referenciamento de textura com *handles.* Os *handles* ('alças'), como o próprio nome já diz, são manuseadores - no nosso caso, um ID único que referencia a textura dentro do contexto OpenGL. Eles oferecem um mecanismo de abstração que permitem ao driver otimizar o carregamento de textura da maneira mais confortável possível. Na listagem abaixo temos os métodos que foram usados em alguns dos códigos fornecidos anteriormente, mas que por seu uso ser meramente intuitivo não se sabia a funcionalidade deles a priori.

- `glGenTextures(GLsizei n, GLuint * textureNames);`
  - Aloca um número `n` de objetos textura e retorna o id dos handles no *array* apontado por `textureNames.`
- `glDeleteTextures(Glsizei n, Gluint * texturesNames);`
  - Deleta `n` texturas cujos *handles* estão no *array* apontado por `textureNames`
- `glBindTexture(GLenum Target, GLuint textureName);`
  - `Target` sempre será `GL_TEXTURE_2D`;
  - Se o *handle* não foi previamente usado, todas as operações subseqüentes em `glTexImage2D()` e `glTexParameteri()` serão armazenadas neste objeto;
  - Se o foi o objeto é completamente carregado;
  - Se `textureName` for zero, o uso dos objetos de textura é zero, e o uso de objetos de textura é desabilitado.

Com isso temos então a interface do objeto `CTexture` definida no Código 6.

Na Código 6, temos o escopo da classe `CTexture` com os métodos mais importantes `Load`, ,`Unload`, `GenObject` e `Destroy`. Esses métodos colocam em nossas mãos mecanismos de leitura e carregamento da textura. E são melhor detalhados na listagem a seguir:

`Load`: Cuida da leitura de arquivo e utiliza métodos de `TextureLoader`, para isso, armazenamento da textura em memória por meio de um *array* de bytes.

Código 6: Interface da classe CTexture.

```
class CTexture : public CBuildEditorObject{
public:
    CTexture(const BE_Texture& t);
    void Destroy(); string GetFileName() const;
    CTexture(string filename, string name = "", OBJECT_ID nTextureID = INVALID_TEXTURE_ID);
    void Load();
    void Unload();
    void GenObject();
    void PrepareRenderer();
... //atributos privados
}
```

`Unload`: Destrói esse *array*, normalmente depois de já termos informado e incorporado as texturas às viewports.

`GenObject`: O método `GenObject` serve para gerar o *handler* apropriado e passar as texturas para o gerenciador OpenGL

`Destroy`: Remove a textura do gerenciador de texturas de OpenGL.

Para finalizar, gostaríamos de salientar que essas classes foram totalmente construídas pelo aluno e, atualmente, estão completamente adaptadas ao uso de múltiplas viewports, além de tirarem proveito de situações de otimização de hardware, fato que torna texturização um ponto forte do motor.

4.2.5 Malhas

Por fim, chegamos às malhas, também conhecidas como mechas, que são um conjunto de pontos ligados de tal maneira que formam um objeto 3D propriamente dito. A malha, no nosso caso, além de um conjunto de polígonos, define também qual o material e texturas que serão aplicados, e possui mecanismos de agrupamento para que se possa definir uma estrutura hierárquica.

Dada a complexidade da malha, foi feita uma divisão no motor original (disponível em Game Institute (2002)) que simplificava a arquitetura. Esta divisão estava organizada em duas classes, que serão vistas a seguir.

4.2.5.1 A classe CMesh

Esta classe possui os pontos que definem a malha. Além disso, ela possui diversas *sub-malhas* que são o agrupamento dos materiais com a textura, com os polígonos aos quais estes materiais e texturas serão aplicados.

Na Figura 10, temos o relacionamento da malha com a sub-malha, além da estrutura da malha que será discutida nesta seção.

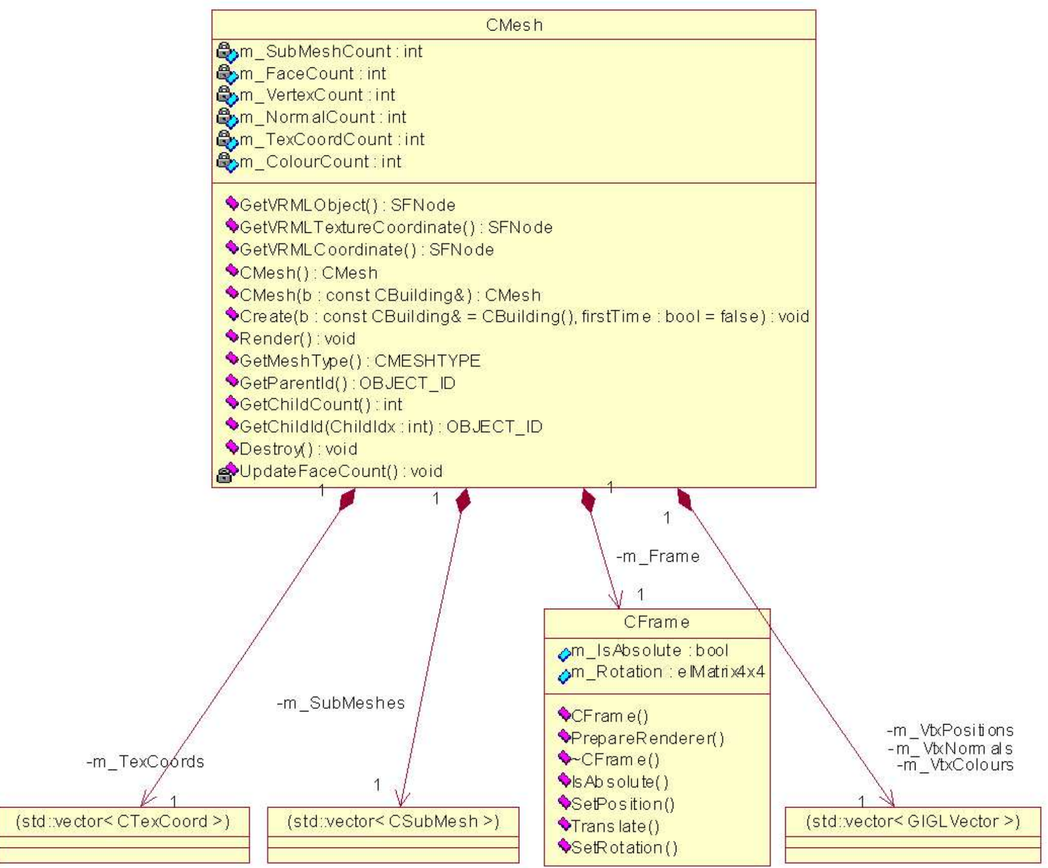


Figura 10: Mecha e seus sub-componentes.

Aqui vemos que uma malha é uma estrutura que possui várias outras sub-malhas, além de possuir informação de indexação de textura, de normais dos vértices armazenados e coordenadas de textura.

Apesar disso, o método de renderização da malha é extremamente simples: ele apenas chama as transformações geométricas relevantes e o método de renderização de cada uma das sub-malhas ou malhas filhas, sendo as sub-malhas as encarregadas de unir todas as informações de textura e material com a informação de vértices e normais presentes na malha.

No Código 7 temos a renderização da malha, bastante simples dentro do seu contexto e aproveitado do motor original:

Observa-se a existência de transformações geométricas colocadas em pilha pelo método `glPushMatrix` e `glPopMatrix`. Esses métodos de OpenGL permitem que estruturas lógicas de transformações geométricas sejam feitas em forma de árvore, pois através deles é que as matrizes de transformação se empilham. Novas transformações geométricas são feitas e, posteriormente, podem ser desempilhadas para voltar ao seu estado anterior (ver

Código 7: Renderização da malha.

```
void CMesh::Render(){
    glPushMatrix();
    m_Frame.PrepareRenderer();
    for (int i = 0; i < m_SubMeshCount; i++)
    {
        m_SubMeshes[i].Render(*this);
    }
    glPopMatrix();
}
```

Figura 11).

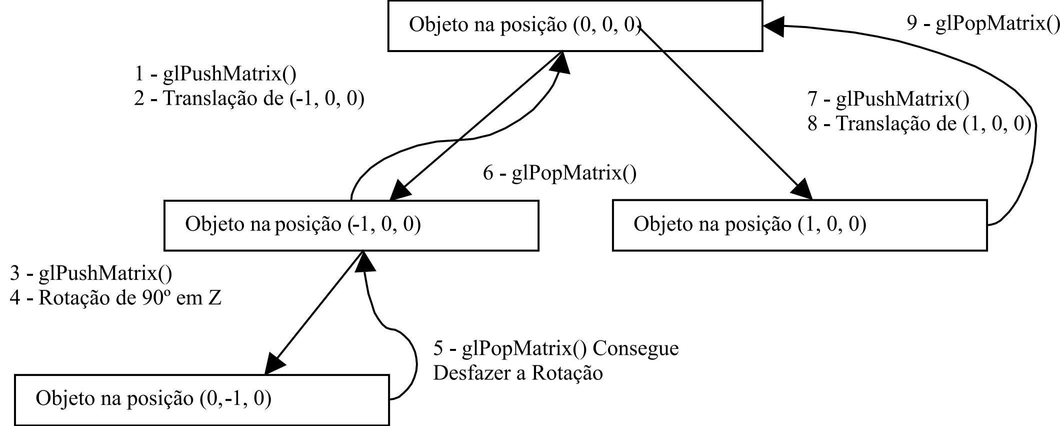


Figura 11: Utilizando a pilha de matrizes para organizar transformações.

O restante das operações de `CMesh`, referente à compatibilidade com OpenGL, será melhor abordado quando for mais conveniente.

4.2.5.2 A classe CSubMesh

A sub-malha é o objeto que engloba a maioria das operações referentes a OpenGL e, por conta disso, sua complexidade é muito grande. Este objeto utiliza *Display Lists* para garantir que o hardware possa tratar o código de maneira mais eficiente possível no código final.

Vejamos agora como funciona a sub-malha. Primeiramente vamos observar a função `PrepareRender()`, localizada no Código 8. Ela trata da seleção do material e das texturas apropriadas, para somente depois renderizar os polígonos. Para ter a textura e o material necessários ela utilza os managers adequados. Essa função é similar à função do antigo motor aproveitado.

Os objetos, que são chamados no Código 8 de `Managers`, serão melhor explicados nas seções seguintes. Os métodos de material e textura foram explicados na seção anterior e tornam o material e texturas apontados ativos para as funções de renderização seguintes,

Código 8: Renderização da Sub-Malha.

```
void CSubMesh::PrepareRenderer(){
    if (m_MaterialId != UNDEFINED_ID)
    {
        CMaterial * pMaterial;
        bool Ret = g_MaterialManager.FindBEObject(m_MaterialId, &pMaterial);
        pMaterial->PrepareRenderer();
    }
    if (m_TextureId != UNDEFINED_ID)
    {
        CTexture * pTexture;
        bool Ret = g_TextureManager.FindBEObject(m_TextureId, &pTexture);
        pTexture->PrepareRenderer();
    }
    else
    {
        glDisable(GL_TEXTURE_2D);
    }
}
```

onde serão dadas as coordenadas dos vértices e das texturas.

Dado que agora temos a textura e o material escolhidos, podemos facilmente renderizar o polígono em questão. Por ser um código extremamente grande e complexo, ele foi colocado no Apêndice K (“Código para renderizar uma sub-malha”, página 147 deste documento). Essencialmente, a função de renderização considera que a estrutura indexada na `CSubMesh` é compostas apenas de triângulos. Essa estrutura se chama `m_VertexIndices` e existe um laço para renderização de cada triângulo contido na malha por meio da chamada de cada um dos vértices do triângulo indexado.

Neste código também existe a possibilidade de uma coordenada de textura diferente ser alocada para cada sub-malha, ou da utilização do mesmo sistema de coordenadas de textura para a malha final, o que só é possível se a malha final tiver uma única textura. Outro aspecto importante deste código é a possibilidade de se renderizar uma malha com uma normal por vértice ou com uma normal por face; no entanto, por padrão, elas são tratadas como parte da malha.

Outro ponto importante na sub-malha é que ela é compatível com uma estrutura chamada *display list*. A criação de *display lists* da malha pode ser vista no código presente no Código 9 pelo método `CreateDisplayList`.

Com o Código 9, mesmo a malha mais complexa pode ser otimizada e suas sub-malhas, com milhares de polígonos, podem ser passadas para o driver por um único comando que usa o *handle* `m_DisplayList` para chamar uma display list já criada, vide o Código 10.

Com isso, cada estrutura toma conta da renderização; e, além de tudo, abstraímos do tipo do objeto criado, pois temos métodos `Load/Unload` somente para textura, mas que podem ser chamados para qualquer outro objeto sem efeito nenhum; e métodos `GenObject/Destroy` para todos os objetos possíveis e que se abstraem de estruturas

Código 9: Criando uma display list.

```
void CSubMesh::CreateDisplayList(CMesh & ParentMesh){
    m_DisplayList = glGenLists(1);
    glNewList(m_DisplayList, GL_COMPILE);
    Draw(ParentMesh);
    glEndList();
}

void CMesh::GenObject() {
    for (int i = 0; i < m_SubMeshCount; i++)
    {
        m_SubMeshes[i]. CreateDisplayList (*this);
    }
}
```

Código 10: Renderizando a display list.

```
void CSubMesh::Render(CMesh & ParentMesh){
    PrepareRenderer();
    glCallList(m_DisplayList);
}
```

de gerenciamento de textura e de `Display Lists`. Esta estrutura de herança e abstração de comportamento dos objetos por meio de um objeto base, no nosso caso `CBuildEditorObject`, caracteriza dentro do motor o padrão de projeto *Composite*, definido no Apêndice L, mais especificamente, na página 155.

4.2.6 Prédios

O prédio é uma estrutura razoavelmente simples, é um objeto com cinco faces retangulares que estão organizadas de forma a permitir o mapeamento de textura em cada face. Apesar de ser uma estrutura que se encaixaria melhor fora do motor ela foi incorporada a ele por querermos tratar com simplicidade a compatibilidade entre os editores.

Para reutilizar o código definido anteriormente no motor o prédio possui uma malha como atributo. E, sempre que necessário, o próprio prédio trata de criá-la de tal modo que ela seja adicionada ao `Manager` específico e seja removida do mesmo `Manager` sempre que necessário. Managers são estruturas de gerenciamento de objetos que serão estudadas na Seção 4.3.3.

Além disso, o prédio, que está presente na classe `CBuilding`, pode ser modificado em tempo real. Por conta disso, ele tem total acesso à malha que o compõe; isso é feito em C++, colocando as duas classes como amigas (`friends`).

Outro ponto importante foi a criação um construtor dentro da classe `CMesh` que constrói um prédio simplesmente olhando para os seus atributos. Essa função é extremamente importante para quando quisermos duplicar uma instância de um prédio, o que é feito no editor de cidades.

O prédio possui como atributos, além de suas dimensões, as texturas e materiais de cada face. Conforme são feitas alterações aos parâmetros, a malha é reconstruída apropriadamente. Justamente por esta constate reconstrução ser permitida, foi preferido não termos uma relação de herança entre o prédio e a malha correspondente.

O prédio também possui um atributo chamado `m_DefaultMaterial`, que é aplicado à malha cada vez que algum material de alguma face não é especificado, o que por default é verdade para qualquer prédio que é criado no *Building Editor*.

Um prédio também pode possuir um estado que define a presença ou ausência de calçadas, sendo que essas são nada mais que malhas definidas com base na dimensão da construção atual. A definição da calçada se encontra na classe `Sidewalk`. O escopo do objeto prédio é definido no Apêndice B, na página 136.

Por utilizar muito o objeto `CMesh`, por ser extremamente simples apesar do tamanho de seu código e por não utilizar técnicas diferentes das já discutidas aqui, o prédio não terá seu código exibido diretamente aqui, mas aquele que quiser olhar sua estrutura e a de qualquer outro componente deste trabalho pode recorrer à referência (ARAÚJO FILHO, 2003).

4.2.7 Rodovias

As rodovias, assim como os prédios, são objetos que possuem malhas como atributos por traz do seu funcionamento. No caso da criação da Rodovia, é aconselhado que haja o carregamento a priori do terreno, no qual esta rodovia será mapeada. Os terrenos são objetos que serão melhor explicados na seção 4.4.

O mapeamento de uma rodovia não é tão simples quanto aparenta, pois envolve uma série de cálculos matemáticos necessários para ver o grau de planaridade e inclinação do terreno nos pontos onde a rodovia passará. Afinal, não é qualquer malha que pode ser considerada uma rodovia.

A rodovia é representada pela classe `CRoad`, classe que possui uma malha e um material como atributos, material esse que é compartilhado entre todas as instâncias da classe, por ser um atributo estático. Este atributo define uma cor padrão para todas as rodovias a serem feitas.

Na nossa abordagem, para conseguirmos o efeito de mapeamento no terreno de uma rodovia, a malha que define a instância da classe `CRoad` é definida como se fosse uma seqüência de pequenas placas de concreto as quais devem ser colocadas uma ao lado da outra dando impressão de continuidade,uma ilustração deste processo é colocada na Figura 12. Esta seqüência de blocos tem suas arestas cruzadas com a grade do terreno

(vide seção 4.4) e suas posições no espaço são definidas com um certo grau de elevação sobre o terreno.

Para tirarmos ondulados da malha final, frutos de terrenos irregulares, um filtro linear de tamanho 3 é aplicado sobre a coordenada Z de placas de concreto vizinhas. Obtemos, então, após a aplicação do filtro, a rodovia final com ondulações e eventuais erros removidos. O escopo do objeto rodovia é definido no Apêndice B, na página 136.

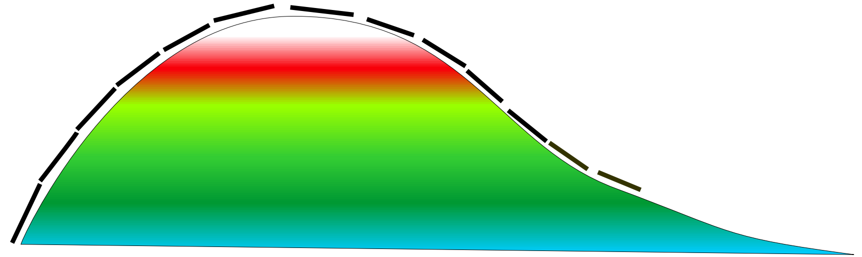


Figura 12: Rodovia: blocos de concreto mapeados sobre o terreno.

4.3 Repositórios

Nesta seção trataremos de objetos que servem para armazenar os objetos básicos, os repositórios. Entre os repositórios encontrados no nosso projeto podemos citar os `Managers` (gerenciadores mais gerais de objetos genéricos), o gerenciado de luzes e a fábrica de ID's. Os dois primeiros são estruturas de gerenciamento e o último é uma estrutura de criação de objetos. Por tratarem de aspectos diferentes, mas serem todos repositórios de objetos, eles serão tratados em uma mesma seção.

4.3.1 IDFactory

O objeto do tipo CIDFactory é extremamente importante em um sistema como um editor de cenários, onde objetos são criados freqüentemente e destruídos com a mesma naturalidade. Os objetos criados precisam de um identificador único que os acesse quando um evento de botão ocorrer na interface gráfica ou quando uma malha precisar de um material para ser renderizado.

No entanto, o CIDFactory não só fornece identificadores únicos, mas também trata para que identificadores já liberados sejam reutilizados por outros objetos.

Neste contexto, surge o problema de como garantir a unicidade de um identificador, ou seja, como fazer com que não exista a chance que outra fábrica de identificadores esteja sendo acessada de outro local. Mais que isso, como fornecer um mecanismo que permita

o acesso global a esta fábrica? Afinal, o identificador pode servir para um `Landscape`, `CRoad`, `Sidewalk`, `CBuilding`, `CMesh`, `CTexture` ou `CMaterial`.

A solução é o primeiro padrão de projeto (GAMMA, 1995) incorporado a este editor: o padrão `Singleton`, vide página 149. Este padrão garante que somente existirá uma única instância do objeto que o implementa, além de oferecer mecanismos de acesso global através de método estático. Para ter mais informações sobre os padrões de projeto utilizados, o leitor pode consultar o Apêndice L, página 149 deste documento.

A classe CIDFactory tem três métodos que são adicionadas ao padrão `Singleton`: os métodos `GetID`, `FreeID` e `Destory` que, respectivamente, obtêm um ID não utilizado, liberam um ID que não será mais usado e, por último, reiniciam toda a estrutura gerenciadora de identificadores deixando todos os IDs livres. No Código 11 temos o escopo da classe `CIDFactory`.

Código 11: Escopo da classe CIDFactory.

```
#define IDFACTORY CIDFactory::GetSingleton()
...
class CIDFactory
{
public:
    CIDFactory();
    virtual ~CIDFactory();
    static void Initialize();
    static CIDFactory & GetSingleton();
    static void Destroy();
    OBJECT_ID GetID();
    void FreeID(OBJECT_ID id);
private:
    int nNumIDs;
    int nAvailable;
    int firstID;
    int *nIDs;
    static CIDFactory * m_Singleton;
};
```

4.3.2 Iluminação

É uma estrutura bastante simples dentro de seu contexto e foi totalmente aproveitada do motor original.

Esta estrutura tenta garantir que o máximo de luzes aproveitadas sejam iguais ao máximo permitido pela placa de vídeo em uso. Por padrão, esse limite é de oito fontes de luzes em OpenGL, mas pode aumentar caso a extensão fornecida pelo driver permita.

Sua classe `CLighting` tem o papel de gerenciar as luzes por meio de identificadores locais, ou seja, não dependentes da classe `CIDFactory`, e permite que as luzes sejam habilitadas ou desabilitadas em tempo real.

Por conta de sua simplicidade não será discutida em profundidade aqui.

4.3.3 Managers

São estruturas que, em principio, têm que ser genéricas pois vão manusear todo objeto que herda de `CBuildEditorObject`, que é o objeto base para a construção de editor. Os Managers irão utilizar o `OBJECT_ID` para oferecer um mecanismo rápido de acesso aos objetos da cena. Esse acesso tem que ser eficiente porque, às vezes, é feito em tempo de renderização.

Essa estrutura funcionará também para impedir que se faça alocação dinâmica de memória por meio de ponteiros, visto que ponteiros são estruturas bem mais complexas que exigem bem mais atenção do programador tornando o programa mais sujeito a erro. Esta constatação de que ponteiros tornam a vida do programador mais difícil, se torna mais evidente durante o desenvolvimento de aplicações complexas como os editores de cenários, onde objetos podem ser criados e destruídos em qualquer parte do código podendo vir a ocasionar acesso indevido a memória, por exemplo.

A solução inicial de projeto que poderia ser tomada, seria construir um `Manager` genérico usando tipos polimórficos de dados.

Polimorfismo, no caso de linguagens orientadas a objeto, é a capacidade de um determinado método da classe ter o mesmo escopo de sua superclasse em um mecanismo de herança. Para um objeto externo, a visão da classe pai é transparente e independente, não sabendo se está lidando com uma classe filha ou com a classe original, dado que as duas classes têm um mesmo escopo por conta da herança.

No entanto, polimorfismo em linguagens de programação como C++, está disponível apenas a partir de acesso por ponteiros ou referência de objetos, o que não era desejado desde o início. Mas, além disso, o polimorfismo é computacionalmente muito caro, pois a classe e a instância do método do objeto são decididas dinamicamente a cada chamada de método.

A solução encontrada, oriunda do motor original, foi o uso de `Templates`: estruturas genéricas que são resolvidas em tempo de compilação e, por isso, são extremamente eficientes. `Templates` são estruturas que abstraem tipos de dados e classes na construção do código. São usados normalmente para definir estruturas lógicas genéricas como listas, conjuntos e vetores, que não dependem do tipo de dado usado. Em outras palavras, `Templates` são ideais para serem compatíveis com estruturas genéricas como os `Managers`.

Outro ponto forte dos `Templates` é a quantidade de linhas de código aproveitadas. Para redefinir uma estrutura baseada em `Templates` e vinculá-la a um determinado tipo de dados, basta uma linha de código. A seguir, temos a linha de código usada para gerar o tipo de dado `CMaterialManager` a partir do `CBEObjectManager`:

```
typedef CBEObjectManager<CMaterial> CMaterialManager;
```

Dado que já temos o porquê da estrutura dos gerenciadores terem sido feitos com o auxílio de `Templates`, veremos agora como eles se integram com as estruturas de aceleração que estão escondidas sob os métodos `Load/Unload` e `GenObject/Destroy`. Na Código 12 temos um código exemplo do cabeçalho da classe.

Código 12: Objeto básico do motor.

```
template<class T>class CBEObjectManager{
public:
    //! Return number of BEObjects currently in manager
    int GetBEObjectCount() const;
    //! Get a BEObject's ID from its name
    bool GetBEObjectId(const string & Name, OBJECT_ID * pId);
    //! Get a BEObject's name from it's ID
    bool GetBEObjectName(OBJECT_ID Id, string * pName);
    //! Test if a BEObject currently exists by Id
    bool ExistsBEObject(OBJECT_ID Id);
    //! Test if a BEObject currently exists by name
    bool ExistsBEObject(const string & Name);
    //! Find BEObject by name
    bool FindBEObject(const string & name, T ** ppBEObject);
    //! Find BEObject by Id bool FindBEObject(OBJECT_ID Id, T ** ppBEObject);
    //! Add a BEObject to the BEObject manager if it does not already exist
    bool AddBEObject(const T & NewBEObject);
    //! Deleting BEObjects.
    void DeleteBEObject(OBJECT_ID Id);
    //!deleta All BEObjects at once void DeleteAllBEObjects();
    //!Destroy all instances of BEObjects in a OpenGL Window
    void DestroyAllBEObjects();
    //!Load and Generate all BEObjecs
    void LoadAndGenAllBEObjects();
    //!Get a vector with all BEObject names in this managers
    void GetAllBEObjectsName(vector<string> & vs);
    typedef map<OBJECT_ID, T> CStlMapBEObjectIdToBEObject;
    typedef map<string, OBJECT_ID> CStlMapStringToBEObjectId;
    CStlMapBEObjectIdToBEObject m_BEObjects;
    CStlMapStringToBEObjectId m_BEObjectNamesToIds;
};
```

Como se vê, o `CBEObjectManager` tem a típica característica de um repositório genérico com métodos de acesso a estruturas por meio de `ID` ou nomes. Esse acesso tem que ser rápido, por isso são usados `Maps`: estruturas baseadas em árvores binárias, bem mais eficientes que *arrays* ou vetores. Na primeira versão, esses `maps` não eram utilizados corretamente, e a complexidade de acesso a um elemento por meio do seu nome era de $O(n)$, o que se demonstrou muito caro para nosso propósito. Atualmente, temos acesso a dados em tempo $O(\log n)$, onde n é o número de objetos presentes no manager.

Observa-se também que, além dos métodos tradicionais de busca, de adicionamento e remoção de objetos, temos métodos para destruição de todos os elemento do `Manager` e métodos para carregar todos os elementos dentro do contexto OpenGL, que são compatíveis com nossa arquitetura de transparência de mecanismos de aceleração por hardware, além de serem indispensáveis dentro do contexto de um editor de cenários onde, por exemplo, quando um documento é fechado, todos seus elementos têm que ser destruídos.

Os outros métodos da classe `CBEObjectManager` são razoavelmente intuitivos e, por isso, não serão abordados aqui.

4.3.4 Mudando o estado dos objetos do motor

Após terminado todo o processo de construção do motor definido nas seções anteriores, o *Building Editor* foi completamente construído sem maiores restrições. Contudo, durante o desenvolvimento do *City Editor*, foi notado que a partir do momento que certos objetos eram carregados no editor, outros objetos com características similares também precisavam ser criados em memória, o que rapidamente se tornou impraticável.

Para entendermos melhor o tipo de problema imaginemos uma situação onde temos um determinado objeto textura em memória e que esta textura tenha 1MB de tamanho. Agora, imaginemos que temos outro objeto do tipo textura, só que esta textura se trata exatamente da mesma imagem só que com coordenadas de textura diferentes. Como faríamos para lidar com estes dois objetos de uma maneira tal que tivéssemos como utilizar o mesmo objeto, mesmo que com coordenadas de textura diferentes? A resposta vem da permissão de uso de estados para renderizar os objetos do editor.

Para nós, 'estados' são certos objetos do editor que encapsulam as propriedades ou atributos mutáveis de um certo objeto, como é o caso das coordenadas de textura de um certo objeto do tipo textura. Para tornar os estados acessíveis a qualquer objeto do editor, o método de mudança de estado foi adicionado ao objeto base do editor. O novo escopo do objeto base do editor pode ser visto no Código 13.

Código 13: Objeto básico do motor com métodos para atualização do estado.

```
class CBuildEditorObject {
public:
    virtual void UndoState(CObjectState *p_Object);
    virtual void PrepareState(CObjectState *p_Object);
    string GetFileName() const;
    virtual void GenObject();
    virtual void Unload();
    virtual void Load();
    OBJECTTYPE GetObjectType() const;
    OBJECT_ID GetObjectID() const;
    string GetName() const;
    void SetObjectType(OBJECTTYPE nType);
    void SetObjectID(OBJECT_ID nID);
    virtual void SetName(const string & Name);
    virtual void Destroy() = 0;
    CBuildEditorObject();
    virtual ~CBuildEditorObject();

private:
    OBJECT_ID m_ObjectID;
    OBJECTTYPE m_ObjectType;
    string m_Name;
protected:
    string m_FileName;
};
```

Para mandar um estado para um objeto, basta enviar um `PrepareState` com algum objeto que herde de `CObjectState`. Para sermos mais precisos, vejamos o que acontece no *Editor de Cidades* quando temos mais de um prédio igual num terreno, como ocorre em um condomínio por exemplo. Primeiramente olhemos uma das classe que herda de `CObjectState`: a classe `CBuildingState`, presente no Código 14.

Código 14: Uma classe que herda do `CObjectState`.

```
class CBuildingState : public CObjectState {
public:
    float GetRotation();
    void GetPosition(float *pos);
    void SetRotation(float roty);
    virtual void SetPosition(float x, float y, float z);
    CBuildingState();
    virtual ~CBuildingState();
    void PrepareRenderer() {
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        m_Transform.PrepareRenderer();
    }
    void Destroy() {
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix();
    }
private:
    CFrame m_Transform;

};
```

A classe `CBuildingState` possui o propósito de guardar transformações geométricas a serem executadas sobre um prédio. Ou seja, a partir de agora, a posição e rotação de um prédio é um estado do objeto, porque assim podemos ter várias instâncias de um mesmo prédio sem descermos ao nível de motor para modificarmos a posição do prédio diretamente na malha, também passível de transformações geométricas. Para vermos quão simples é tratar um estado de um objeto especifico, veja o código presente no Código 15

Código 15: Prédio processando seu estado.

```
void CBuilding::PrepareState(CObjectState *p_Object) {
    CBuildingState *p;
    p = static_cast<CBuildingState *>(p_Object);
    p->PrepareRenderer();
}

void CBuilding::UndoState(CObjectState *pObject) {
    CBuildingState *p;
    p = static_cast<CBuildingState *>(p_Object);
    p->Destroy();
}
```

Um outro ponto interessante no uso de estados é que novos estados podem ser definidos externamente ao motor, e novos comportamentos podem se acrescentados à ferramenta sem o envolvimento direto com OpenGL. Assim simplificamos certas operações que são complexas de serem feitas externamente ao nosso motor, sem termos que mexer com estruturas peculiares como uma malha ou acessarmos atributos privados ou de baixo

nível. Isto facilita a integração do motor ao editor de cenários, bem como a outras possíveis estruturas.

4.4 Estruturas para Renderização do Terreno

No nosso motor sentimos a necessidade de uma estrutura especial de renderização de terrenos. Isso acontece porque terrenos são normalmente estruturas muito pesadas em um ambiente a céu aberto, por requererem um número muito grande de polígonos no ato da renderização.

Entretanto terrenos são estruturas especiais. Por causa do seu tamanho ocupam uma área muito grande do mundo, fazendo com que sua renderização possa ser adaptada à posição do observador. Por exemplo, para a representação de uma área próxima ao observador será utilizado um número maior de polígonos do que para representar uma área situada a centenas de metros do mesmo, a esta técnica se dá o nome de *refinamento coletivo*. Outra aproximação que pode ser usada é a *simplificação da malha*, que consiste em reduzir a quantidade de informação disponível no modelo original removendo pontos em excesso ou reduzindo a precisão da localização espacial dos pontos em questão. Para maiores referências sobre adaptação de malhas, ver Hoppe (1996).

Para representar um terreno no mundo discreto dos computadores foram usados mapas de alturas na forma de grades de elevação. Estas estruturas são descritas como um *array* bidimensional de valores, em que estes definem a altura do terreno naquele ponto, isto é, como funções do tipo $g\colon Z \longrightarrow \mathbb{R}$, onde Z é um produto cartesiano de dois intervalos em $\mathbb{Z}$, o conjunto dos inteiros. Esta estrutura, além de comumente usada na maioria dos artigos lidos, também é compatível com a estrutura (nó) `Elevation Grid` presente em VRML.

Para finalizar, artigos recentes (LINDSTROM; PASCUCCI, 2002, 2001) mostram que a renderização de terrenos envolve muitos aspectos não triviais como o conflito entre capacidade de armazenamento, eficiência de acesso e realismo do resultado alcançado. Este artigo mostra como aspectos comuns dos algoritmos de renderização de terreno podem ser empregados como critérios de comparação entre eles. Entre estas técnicas podemos citar:

- métrica de divisão dos polígonos;
- consistência da malha;
- estrutura de dados;
- memória por vértice;

- coerência entre quadros, dado que uma mudança muito brusca do ambiente entre os quadros comprometeria o realismo da aplicação.

Por conta da complexidade envolvida no assunto não entraremos aqui em detalhes maiores acerca da implementação de nossos algoritmos. Daremos ênfase a questões arquiteturais e a noções gerais dos algoritmos estudados.

4.4.1 Algoritmo Trivial

Esta, como o próprio nome sugere, é a implementação mais trivial entre os algoritmos feitos. Esta implementação não se adapta à localização da câmera ou ao formato do terreno em determinado tempo, por isso não pode ser definida como um algoritmo de *refinamento coletivo* e muito menos de *geomorphing*, conforme descrito em Hoppe (1996). Ela simplesmente renderiza o mapa de altura com uma certa precisão; como item de definição de precisão temos um determinado *passo* que nos faz apenas pegar uma amostragem do mapa de alturas percorrendo, por exemplo, apenas de 16 em 16 pixels deste mapa. Os vértices não percorridos são apenas descartados pelo algoritmo. Por conta disso o algoritmo pode ser classificado como algoritmo de *simplificação da malha*.

Devido a simplicidade do algoritmo mostraremos seu código, disponível no Código 16.

Código 16: Renderização trivial do terreno.

```
...
    for ( X = 0; X < MAP_SIZE; X += STEP_SIZE )
        for ( Y = 0; Y < MAP_SIZE; Y += STEP_SIZE )
        {
            // Color each quad based on the height of the landscape.
            UINT8 nHeight = Height( X, Y );
            ...
            // Output a vertex for each corner of the quad.
            glVertex3i( X,
                        Height( X, Y ),
                        Y );

            nHeight = Height( X, Y + STEP_SIZE );
            ...
            glVertex3i( X,
                        Height( X, Y + STEP_SIZE ),
                        Y + STEP_SIZE );

            nHeight = Height( X + STEP_SIZE, Y + STEP_SIZE );
            ...
            glVertex3i( X + STEP_SIZE,
                        Height( X + STEP_SIZE, Y + STEP_SIZE ),
                        Y + STEP_SIZE );

            nHeight = Height( X + STEP_SIZE, Y ) ;
            ...
            glVertex3i( X + STEP_SIZE,
                        Height( X + STEP_SIZE, Y ),
                        Y );
        }
```

4.4.2 Algoritmo de Röttger

Röttger é um algoritmo baseado em uma estrutura de dados chamada Quad Tree, na qual se define uma árvore onde cada nó tem, no máximo, quatro filhos. A Quad Tree é uma estrutura na qual colocamos o mapa de alturas. Ela é definida sobre este mapa de uma maneira tal que vai permitir a simplificação do mesmo, sempre que desejado. A Quad Tree é usada de maneira genérica para dividir o espaço em quatro sub-espaços, os quais também podem ser divididos recursivamente.

Para vermos como essa Quad Tree é definida dentro do nosso algoritmo, vejamos a Figura 13. Nesta figura temos os círculos como sendo valores dentro do mapa de alturas. O círculo de azul mais escuro define a raiz da árvore e as arestas em vermelho apontam para os filhos das arestas-pai e as arestas em preto apontam para a segunda hierarquia de arestas filhas.

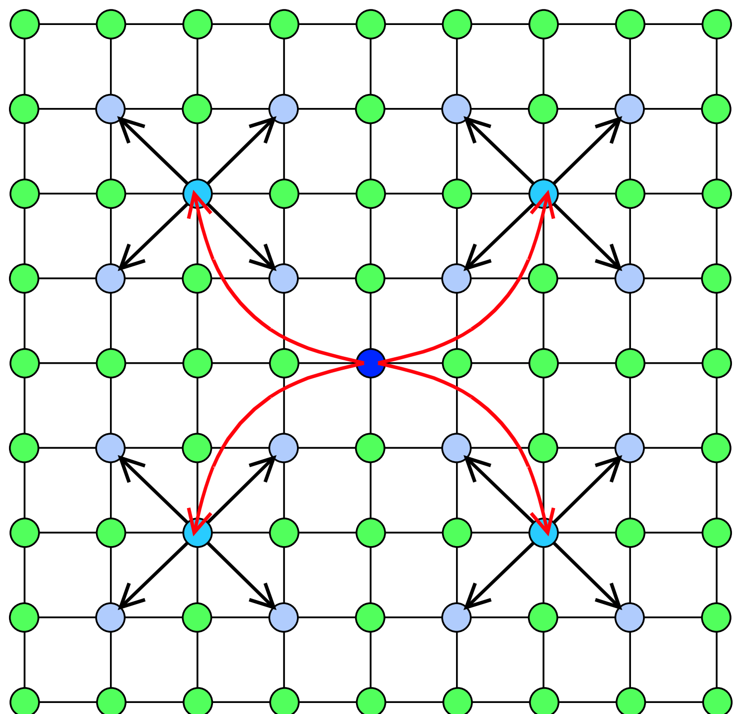


Figura 13: Quad Tree em um mapa de alturas.

A idéia do algoritmo de Röttger é dado um mapa de alturas expandir uma Quad Tree para um nível mais baixo na hierarquia sempre que necessário para que o realismo do terreno não seja afetado. Um exemplo de uma possível Quad Tree pode ser visto na Figura 14.

No algoritmo de Röttger a seguinte pergunta é feita à cada nível: **devo descer mais um nível da hierarquia da Quad Tree para dar mais precisão ao terreno?** Baseado nisso, o algoritmo sugere a análise da métrica de divisão dos polígonos.

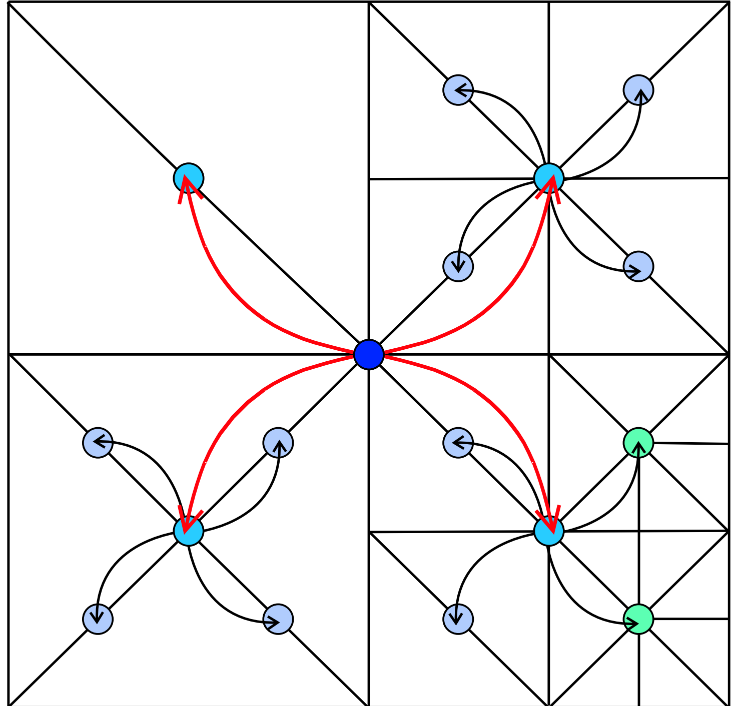


Figura 14: Quad Tree com triangularização correspondente.

A métrica de divisão dos polígonos no caso da Quad Tree, é dada pela equação (4.1):

$$f = \frac{\ell}{d \cdot C \cdot \max(c \cdot d_2, 1)}, \text{ subdivida se } f < 1, \tag{4.1}$$

onde

ℓ: Distancia do nó da Quad Tree até a câmera; quanto mais distante da câmera menor o detalhamento necessário do terreno.

d: Tamanho da aresta que forma esta sub-árvore.

C: Resolução global mínima desejada. Define uma idéia de quanta área uma sub-árvore pode ocupar na tela. C é definida à partir de ℓ e d como $\ell/d < C$

d_2: Medida de erro de um vértice em relação à posição em que ele deveria estar. Esta medida também leva em conta ℓ e d e considera os erros de todos os vértices filhos. Estes erros são conseqüencia das ramificações da Quad Tree atingirem profundidades diferentes. Estes erros são mostrados na Figura 15 com a notação dh_x.

c: Constante usada para ajustar o ambiente à taxa de quadros desejada para a máquina. Ou seja, baseado nesta constante é que o algoritmo aumenta ou diminui o limite de polígonos renderizados.

Maiores detalhes sobre esta técnica não serão dados aqui, um desses detalhes é como

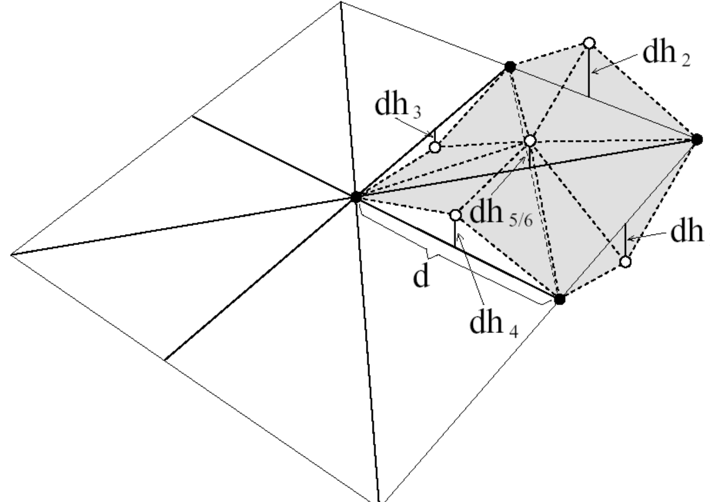


Figura 15: Erros dos pontos no mapa de altura devido à Quad Tree. FONTE: Röttger (1998).

esta técnica utiliza *Geomorphing* para reduzir o impacto visual que o terreno causa quando sua forma vai mudando conforme a camera se movimenta no ambiente, entretanto a referência (RÖTTGER, 1998) é a melhor fonte para quem deseja mais informações sobre esta técnica.

4.4.3 Algoritmo ROAM

Nesta seção falaremos do algoritmo ROAM, como é conhecido o algoritmo *Real-Time Optimally Adapting Meshes* disponível na referência Duchaineau (1997). Este algoritmo tem uma idéia muito parecida com a do de Röttger, ou seja, criar uma estrutura geométrica que se adapte à posição da câmera.

Uma das diferenças entre os algoritmos é que o de Röttger utiliza a estrutura chamada Quad Tree, enquanto o ROAM utiliza uma estrutura de *árvore binária de triângulos*. Esta estrutura pode ser facilmente notada na Figura 16. Na figura temos que ℓ simboliza a profundidade da árvore em questão, com $\ell = 0$ temos o triângulo original, com $\ell = 1$ temos os triângulos que na hierarquia seriam os filhos do triângulo original, com $\ell = 2$ temos o segundo nível da hierarquia com os netos do triângulo original, e assim sucessivamente para $\ell = 3, 4, \ldots$ até atingirmos a profundidade desejada na árvore binária.

Assim como o algoritmo visto na seção anterior, o objetivo deste algoritmo é, dada uma árvore binária completa (em que todos os ramos têm a mesma profundidade), reduzir a quantidade de triângulos necessários à renderização fazendo com que certos ramos não atinjam a profundidade máxima.

No nosso caso, a árvore corresponde ao mapa de alturas, como mostra a Figura 17.

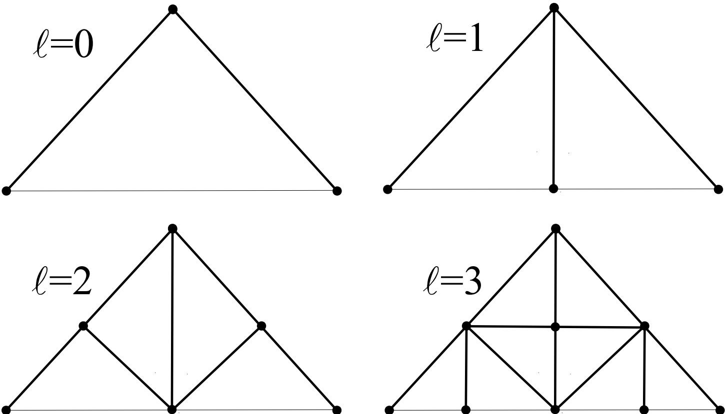


Figura 16: Árvore binária com triangularização correspondente a cada nível ℓ.

Inicialmente, ela possui 2 triângulos que representariam o terreno, os quais representariam o nível 0 de profundidade das duas arvores que irão retratar. Mas conforme fazemos a divisão dos triângulos chegamos ao nível 3 da hierarquia criando novos nós na árvore. Na Figura 17 temos uma representação dos nós do mapa das alturas, onde os nós usados pela árvore estão em preto e os nós não usados estão em azul.

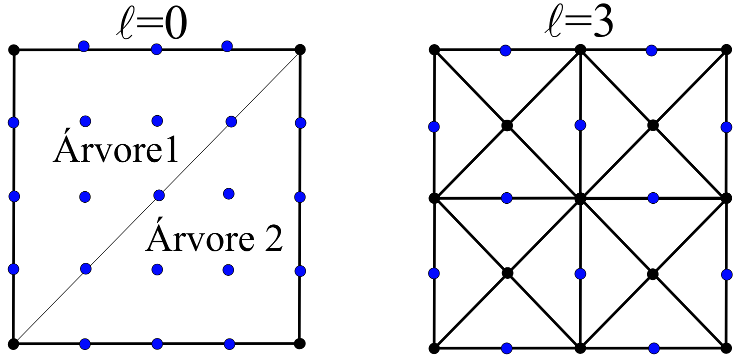


Figura 17: Árvore binária mapeada no mapa de alturas.

Vejamos como o algoritmo se comporta para definir a triangularização de uma árvore não completa, como mostra a Figura 18. Para definir uma árvore não completa o algoritmo tenta definir operações de junção e divisão dos triângulos da árvore binária de tal maneira que se impeça a descontinuidade da malha resultante.

Na Figura 19 temos um caso particular das operações de junção e união mencionadas anteriormente. Estas operações permitem que expandamos ou contraiamos a estrutura da árvore original, definindo novas árvores a partir dela. Um aspecto interessante a se observar nesta figura e, em todas as figuras anteriores, é o fato das operações estarem somente definidas para o maior lado do triângulo, definido no artigo original como lado *base*.

Entretanto, para definirmos estas operações temos que averiguar a existência de um triângulo vizinho pela base, como o da Figura 19. Caso o vizinho da base não exista,

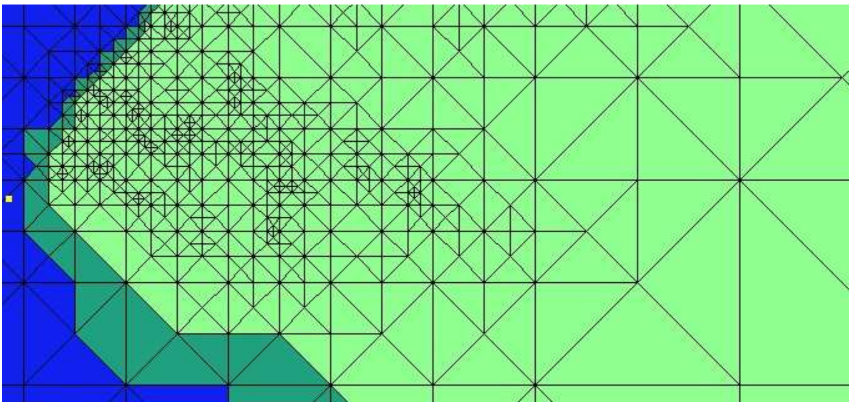


Figura 18: Triangularizando o terreno por meio do ROAM. FONTE: Duchaineau (1997).

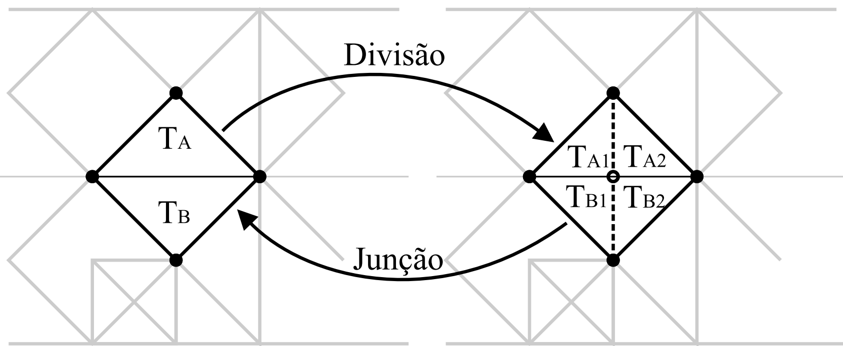


Figura 19: Operações de divisão e junção de triângulos.

temos que defini-lo recursivamente conforme mostra a Figura 20-a esta operação damos o nome de *divisão forçada*. Não entraremos em maiores detalhes a respeito de qual o algoritmo de divisão que decide qual dos numerosos triângulos são divididos para atingir um resultado satisfatório, pois ele envolve uma estrutura de duas filas de prioridades, o que o torna razoavelmente complexo.

Para finalizar, deve-se dar ao algoritmo uma *split metric* ou *métrica de divisão*, como a dada pela equação (4.1) na seção anterior. No caso do ROAM, esta métrica pode ser definida pela equação (4.2):

$$\text{métrica} = \frac{\text{ERRO_DO_NO} \bullet \text{TAM_MAPA} \bullet 2}{\text{DIST}}, \text{ subdivida se métrica} < \text{resolução}, \quad (4.2)$$

onde

ERRO_DO_NO: Medida de erro de um vértice em relação à posição em que ele deveria estar. Considera também os erros de todos os vértices filhos, considerando o máximo entre os erros deles e do vértice pai. Estes erros são conseqüência das ramificações da *árvore binária* atingirem profundidades diferentes da do mapa de alturas. Este

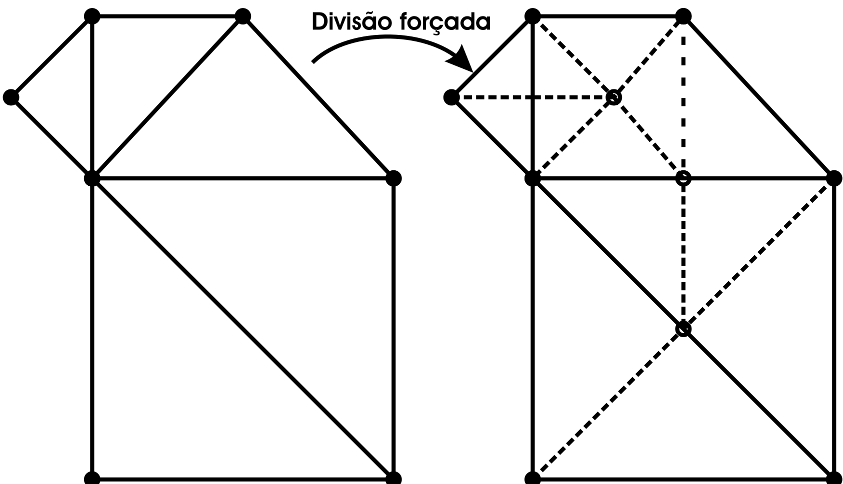


Figura 20: Operação de divisão sem triângulo base.

erro é mostrado na Figura 21.

TAM_MAPA: O tamanho do mapa, para tornar a distância à câmera uma medida no intervalo entre $[0, 1]$.

DIST: Distância do vértice até a câmera.

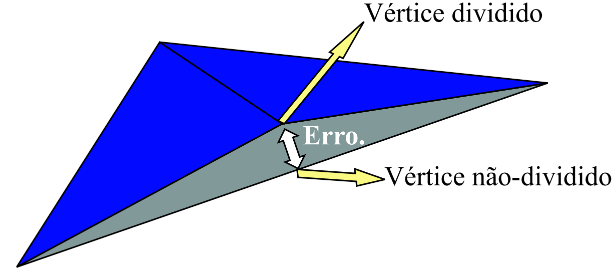


Figura 21: Erro inerente à não divisão do triângulo.

A origem da equação (4.2), que é na verdade a simplificação de uma equação que leva em conta a distorção de perspectiva e alguns detalhes da implementação de nosso algoritmo, estão disponíveis na referência (DUCHAINEAU, 1997). Estes detalhes não foram citados aqui por não se adequarem ao contexto deste trabalho.

4.4.4 Comparação de algoritmos de renderização de terreno

Como foi notado, o terreno é uma parte muito delicada na maioria de cenários abertos em ambientes 3D, como é o caso de nosso editor de cidades. Portanto a implementação de algoritmos eficientes se mostrou necessária neste contexto. Entretanto não foi abordado, até então, o porquê da implementação de três algoritmos diferentes que atendam ao mesmo propósito.

Nesta seção explanaremos o porquê da implementação de diversos algoritmos de renderização de terreno, analisando os prós e contras de cada uma das implementações e entrando em certos detalhes da implementação necessários ao nosso entendimento. Primeiramente, analisaremos o algoritmo que batizamos de *Trivial* e posteriormente os outros algoritmos mais complexos.

4.4.4.1 Algoritmo Trivial

Apesar de sua simplicidade vejamos nesta seção as características que, apesar de poucas e simples, tornam o algoritmo *Trivial* comparável com os outros algoritmos listados a seguir. Entre as virtudes do algoritmo trivial podemos citar:

- Simplicidade.
- Por não ter mudança da representação geométrica do terreno durante o processo de renderização, permite o uso total de mecanismos de aceleração por hardware.
- Permite, através do ajuste do passo no mapa, a sua adaptação às capacidades desejadas para a máquina.
- Recomendado para terrenos suaves, sem rugosidades.

Já entre os defeitos podemos enumerar:

- Falta da adaptibilidade ao terreno:
  - Não recomendado para terrenos não suaves, com regiões íngremes ou rugosas, pois tende a desprezar os detalhes por conta da quantidade de vértices ignorados pelo *passo do algoritmo*.
  - Se o *passo* for extremamente pequeno, como a unidade do mapa por exemplo, não reduz a quantidade de triângulos necessária à renderização, mas preserva os detalhes do mapa.
  - Terreno com regiões tanto suaves quanto outras regiões extremamente detalhadas não podem sequer ser adaptados a um *passo* específico.
  - Nenhum cálculo é feito em tempo de execução para adaptar o *passo* ao tipo de terreno.
- Não leva em consideração a posição do usuário no espaço e, por isso, não otimiza regiões distantes ou escondidas de um terreno grande.

4.4.4.2 Algoritmo Röttger

O algoritmo de Röttger, conforme já visto, é um algoritmo baseado em uma Quad Tree, assim como seu antecessor, o algoritmo de Lindstrom, disponível em (LINDSTROM, 1996). Esta referência define um *benchmark* para comparação de algoritmos de renderização de terreno. Contudo, a grande diferença entre os dois algoritmos é que o algoritmo de Röttger atinge a mesma eficiência do de Lindstrom, melhorando seus pontos fracos como a alta utilização de memória, pois emprega estruturas de dados bem mais enxutas, e a complexidade do algoritmo final.

Entre as virtudes deste algoritmo podemos enumerar:

- Redução do numero de polígonos a ser renderizado.
  - Esta redução leva em conta a complexidade do terreno, envolvendo questões como níveis aspereza e mudanças abruptas de altitude a exemplo do que ocorre em penhascos e desfiladeiros.
- Mudança contínua e suave entre vários níveis de detalhamento do terreno.
  - Este item é excepcionalmente importante para que se mantenha o realismo do resultado final, já que o usuário não deveria notar a mudança do terreno entre as mudanças da estrutura final a ser renderizada.
  - Isto só é possível por conta da fácil inclusão no algoritmo de técnicas de *Geomorphing*.
- Permite a geração de níveis de detalhamento do terreno em tempo real. Isto, entretanto, não foi inserido no nosso motor por não vir ao contexto do tipo de aplicação desejada.
- Pouca memória necessária para rodar o algoritmo. Só é necessário carregar uma matriz do tamanho do *mapa de alturas* e uma matriz do mesmo tamanho com elementos de 1 byte cada. Esta última matriz servirá para guardar o fator necessário para aferição do nível de profundidade da Quad Tree indispensável à adaptabilidade do algoritmo, bem como a aspereza do vértice atual.
- Permite aceleração por hardware com o uso de uma estrutura de OpenGL chamada *strip vertex*. Esta estrutura permite que uma certa quantidade de vértices seja mandada em cascata para o *pipeline* gráfico.

Entre os defeitos do algoritmo podemos citar:

- Não pode renderizar o terreno num detalhe maior do que sua matriz da Quad Tree. O que no nosso caso, não é um grande empecilho, uma vez que que nosso detalhe máximo é o do mapa de alturas.

- Apesar de termos anteriormente citado a memória como uma vantagem do algoritmo. Ela se perde quando adicionamos ao algoritmo características de *Geomorphing* ou *Propagação de erros* dos vértices pais para os filhos. Com estes dois mecanismos o gasto de memória do algoritmo é aumentado em três vezes, sendo que $\frac{1}{3}$ disso é memória gasta inutilmente em nome da eficiência.

Assim chegamos ao fim da explanação do algoritmo com sérias críticas mas também tornando claras certas características desejadas que ele oferece, no contexto do nosso motor.

4.4.4.3 Algoritmo ROAM

Por fim chegamos à análise algoritmo de renderização de terrenos ROAM. Este algoritmo atende a todos os requisitos de (LINDSTROM, 1996) e também oferece *Geomorphing*.

De uma maneira geral, as características do algoritmo podem ser descritas pelos seguintes prós e contras. Entre os prós podemos citar:

- Gera uma malha ótima com o menor número de triângulos possível no terreno.

- Imperfeições da malha podem ser concertadas sem nenhuma estrutura auxiliar, como é o caso no algoritmo de Röttger em que um bit é utilizado com este fim. Tudo é feito com a estrutura ligada de triângulos, a árvore binária.

- Adaptabilidade do algoritmo à taxas de frame desejadas; esta adaptação é feita em tempo real pelo algoritmo.

Após esta rápida avaliação, poderíamos, a princípio, pensar que este algoritmo é o melhor entre os algoritmos citados. Contudo isto não é real, pois o algoritmo ROAM apresenta os seguintes problemas:

- Implementação mais complexa que os algoritmos que são baseados em Quad Tree.

- Não é adaptável aos mecanismos de aceleração por hardware existentes. Isto acontece por que exigiria um tempo muito grande de CPU adaptá-lo à estruturas como *strip* de triângulos ou *fans*.

- Não é um algoritmo bem documentado, dado que toda sua teoria é fornecida de forma resumida em um pequeno artigo, com poucos detalhes da implementação.

Assim chegamos ao fim da análise do algoritmo ROAM, que apesar de ter suas limitações também tem suas qualidades, o que justifica nossa decisão de termos à disposição várias implementações diferentes de algoritmos de renderização de terreno no motor. Gostaríamos também de deixar claro que, por conta da complexidade de certos ítens sugeridos em (DUCHAINEAU, 1997), a implementação presente no motor é uma versão simplificada do algoritmo original.

4.4.5 Coloração, Textura e Iluminação

Agora que analisamos os algoritmos com maior detalhe, vejamos os aspectos de coloração do terreno, juntamente com questões como iluminação e mapeamento de textura. Estes ítens, apesar de aparentemente simples não o são; isto acontece porque a constante mudança da malha que retrata o terreno não se pode tornar perceptível ao usuário.

Vejamos, primeiramente, o exemplo de como a iluminação pode vir a afetar o resultado final, no caso de nosso editor e qualquer aplicação com terreno a céu aberto. Suponha que usemos o mecanismo padrão de OpenGL para darmos a noção de sombreamento ao terreno. Este mecanismo poderia vir a comprometer assustadoramente a qualidade do ambiente final, dado que a noção de sombreamento, como foi visto neste documento anteriormente, é obtida internamente a partir de cálculos matemáticos efetuados sobre a normal ao vértice de cada polígono. Sabemos agora que estes polígonos mudam insistentemente durante o tempo em que nosso ambiente é visualizado, o que nos faz supor que, com estas constantes mudanças, as sombras iriam se modificar imediatamente conforme os polígonos se dividissem ou se juntassem.

Este efeito é mais perceptível considerando um exemplo. Suponha que antes tivéssemos um único polígono que se estendia por vários metros, colocado sobre um único plano e cujo efeito de uma aproximação de câmera tenha tornado sua divisão iminente, aquele polígono que antes tinha apenas um sombreado unidirecional acabaria por criar novos detalhes de sombra necessários aos polígonos que surgirão, detalhes que deveriam anteriormente estar lá, já que possuíamos apenas uma luz unidirecional (o sol) que não se modificou enquanto caminhávamos.

Entretanto esta não é a única questão envolvida com o não uso de iluminação OpenGL por parte de nosso motor. Conforme analisado anteriormente, uma das preocupações dos algoritmos de renderização de terreno é a questão da memória, face a quantidade de informação que um terreno pode trazer, como ocorre, por exemplo, em um terreno definido sobre uma grade de tamanho 1024×1024. Agora imagine quando, além do mapa

Código 17: Gerando a Textura para OpenGL.

```
void CTerrainTexture::GenObject() {

    glGenTextures( 1, &m_GLTextureObject );
    glBindTexture(GL_TEXTURE_2D, m_GLTextureObject);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

    glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL );

    static GLfloat s_vector[4] = { 1.0f/(GLfloat)MAP_SIZE, 0, 0, 0 };
    static GLfloat t_vector[4] = { 0, 0, 1.0f/(GLfloat)MAP_SIZE, 0 };

    glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR );
    glTexGenfv( GL_S, GL_OBJECT_PLANE, s_vector );

    glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR );
    glTexGenfv( GL_T, GL_OBJECT_PLANE, t_vector );

    glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );


    gluBuild2DMipmaps(GL_TEXTURE_2D, 3, m_Width, m_Height, GL_RGB,
                      GL_UNSIGNED_BYTE, pTexture );

}
```

de alturas, sejamos obrigados a carregar normais aos vértices e coordenadas de texturas. Um custo que inicialmente seria de alguns MB de memória se transformaria rapidamente em dezenas de MB, o que seria inadimissível.

A questão é achar uma maneira de reduzir o problema de memória sem comprometer o realismo da aplicação final. Isso foi resolvido tomando como base a sugestão dada na referência (Game Institute, 2002). A sugestão consiste em simular o efeito de iluminação na textura clareando ou escurecendo o pixel da mesma que se encontra numa determinada inclinação do terreno. Por fim, no ato de renderização do terreno basta mapear a textura sobre toda a malha, sem para isso tenhamos sequer que usar coordenadas de textura, conforme mostrado pelos comandos `glTexEnvi`, `glTexGeni` e `glTexGenfv` do Código 17.

Um outro ponto forte desta técnica é que, além de reduzir nosso custo de memória, ela melhora a eficiência da renderização porque a geração da textura, já com efeito de iluminação, é feita somente durante o pré-processamento, depois disso. A textura é simplesmente carregada em memória. No ato da renderização efeitos de iluminação são desativados.

Para melhor entendermos o processo de geração de texturas, adaptado da referência (Game Institute, 2002), vejamos o código disposto no Apêndice N, página 160 deste documento. Neste código, vemos claramente que a textura gerada não suporta apenas os mecanismos de geração de textura descritos anteriormente. A textura gerada

possui as mesmas dimensões do terreno, conforme mostra a seguinte linha de código.

```
#define TEXTURE_SIZE (MAP_SIZE+1)
```

Em outras palavras, definimos que cada pixel do terreno deve ocupar uma unidade do mapa de alturas. Estes pixels, por sua vez, devem ter sua taxa de sombreamento adaptada, baseada na normal do triângulo de lado `TRI_SIZE` no qual eles se encontram. Esta normal tem seu produto interno feito com o vetor de luz direcional, representado pelas macros `LightX`, `LightY` e `LightZ`, as quais acessam uma variável estática que aponta para uma luz direcional que afeta todo o ambiente.

Entretanto este não é o único ponto de edição do terreno. A textura também deve trazer um pouco de vida e realismo ao terreno e é por isso que o algoritmo também trata de colorir e adicionar ruídos às texturas. O adicionamento de ruídos não tem como propósito adicionar um comportamento específico à textura, mas sim adicionar imperfeições sem nenhuma característica peculiar, mas que simulem bem areas como grama, por exemplo. Para isto, um gerador de seqüências pseudoaleatórias com distribuição uniforme no intervalo $[0, 1]$ atende às nossa necessidades e, por isso, nossa função utiliza a função padrão `rand()` de C++ para alterar a sombra adicionando algumas imperfeições ou pequenas pintas à textura. Isso é mostrado no código a seguir, onde definimos que esta imperfeição afeta apenas 20% do brilho final.

```
float shade=0.1f + (0.1f * rand())/RAND_MAX;
```

O último ponto relevante da textura é adicionar cores ou algo que melhore o efeito de realismo obtido com sua utilização, porém esta tarefa não é simples como parece. Em uma primeira impressão o uso de *tiles* ou pequenas texturas para gerar a textura final poderia parecer uma boa idéia, entretanto este tipo de procedimento tornaria a técnica de geração da textura final demasiadamente lenta, além de requerer um esforço muito grande na edição desta textura para gerar o ambiente final. Outro ponto forte contra a geração de texturas a serem editadas por meio de *tiles* é a necessidade de voltarmos a ter coordenadas de textura em nosso terreno final, o que, conforme comentado, aumentaria bastante o custo de memória envolvido em terrenos. Porém, mesmo com estas limitações, a técnica de *tiles* em terrenos 3D é usada em muitos jogos de computador, mas sempre com a câmera fixa e nunca com permissão de navegação em primeira pessoa, o que permite a fácil dedução da área visível do terreno, além do que estes terrenos de jogos são muito menores quando comparados aos terrenos de simulações realistas como as que desejamos aqui fazer.

A solução mais simples utilizada para o nosso editor foi utilizar uma cor correspondente para cada nível do mapa de altura, permitindo, assim, que retratemos questões

como neves nos picos mais altos de montes ou montanhas e níveis diferentes de verde de grama seguindo o aumento de altitude. Isto é incorporado no código disponível no Apêndice N, página 160 deste documento, por meio da variável `aLandColors`, que é um *array* responsável pelo detalhamento das cores dos níveis de altura desejados.

A Figura 22 mostra como a textura fica ao final dos passos mencionados acima, com as **sombras** sendo simuladas por meio do clareamento e escurecimento dos pixels de acordo com a inclinação do terreno no ponto em questão e de acordo, também, com a direção da **luz** ambiente. Esses fatores juntamente com a utilização de cores baseadas na altura do ponto correspondente ao pixel da textura, branco para mais alto, marrom para altura média e verde para mais baixo dão o efeito realista do terreno.

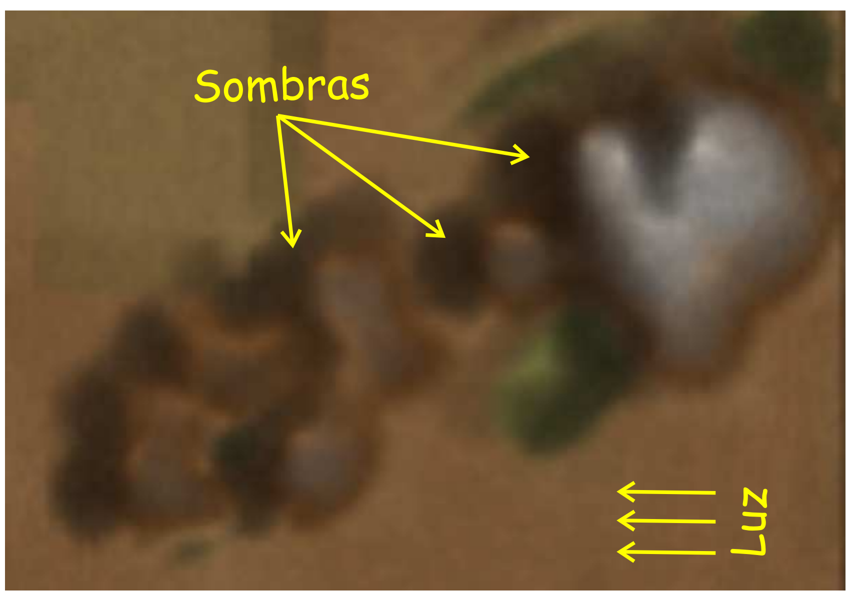


Figura 22: Textura do terreno.

Para mais informações sobre, técnicas mais realistas de texturização de terrenos o usuário deve consultar a referência Döllner, Baumann e Hinrichs (2000).

4.4.6 Arquitetura das classes de terreno

Agora que já tratamos questões a nível de implementação e descrição das nossas técnicas de renderização de terreno vejamos como foram abordadas questões arquiteturais, isto é, como as classes e objetos correspondentes ao terreno se comportam e relacionam entre si.

Como partimos do pressuposto de que todo o ambiente do nosso editor possui apenas um terreno, chegamos, posteriormente, à conclusão de que nossa textura de terreno é na verdade um Singleton (ver Apêndice L, página 149 deste documento). Assim, criamos a

estrutura mostrada no Código 18, que encapsula as propriedades de textura descrita na seção 4.2.4 bem como as propriedades do Singleton.

Código 18: Estrutura da textura do terreno.

```
#define TERRAIN_TEXTURE (CTerrainTexture::GetSingleton())

class CTerrainTexture: public CTexture {

public:
    void Load();
    void GenObject();
    void PrepareRenderer();
    static CTerrainTexture & GetSingleton();

protected:
    unsigned char *pTexture;

    int    m_Width;
    int    m_Height;

    int m_BPP; //Bytes per pixel
    float m_AmbientLight;

public:
    CTerrainTexture();
    GLuint m_GLTextureObject;
    static CTerrainTexture * m_Singleton;
    bool m_TextureDone;
};
```

A estrutura definida no Código 18, assim como na textura padrão, define os métodos `Load`, `Unload`, `GenObject`, `Destroy` e `PrepareRenderer`, entretanto os métodos `Load`, `GenObject`, e `PrepareRenderer` são adaptados para este tipo peculiar de textura, conforme o comentado na seção anterior.

Na Figura 23 mostramos que para cada um dos algoritmos de renderização de terreno definidos anteriormente temos uma classe correspondente. Isso facilita a integração de novos algoritmos ao motor, além de definir uma estrutura que se adequa a estratégia de qualquer algoritmo conhecido.

O fato de que os algoritmos sejam definidos herdando de uma classe em comum *Landscape*, que define os principais métodos de um objeto do tipo terreno, permite a abstração do tipo de algoritmo usado. Esta estrutura facilita o uso do motor, pois o programador pode utilizar os métodos da classe *Landscape* sem tem idéia de que tipo de algoritmo está fazendo o trabalho.

Mais do que simplesmente mostrar uma estrutura hierárquica, a Figura 23 define uma estrutura de cadastramento de terrenos criados. Os nomes que identificam estes terrenos são definidos no método construtor da classe, o `CreteLandscape`; este mesmo construtor convoca o método `RegisterLandscape` que os cadastra no *array* `m_aLandType`, que é um atributo estático e público da classe `Landscape`.

Outro atributo estático da classe `Landscape` é o tamanho do mapa `m_MapSize`, este

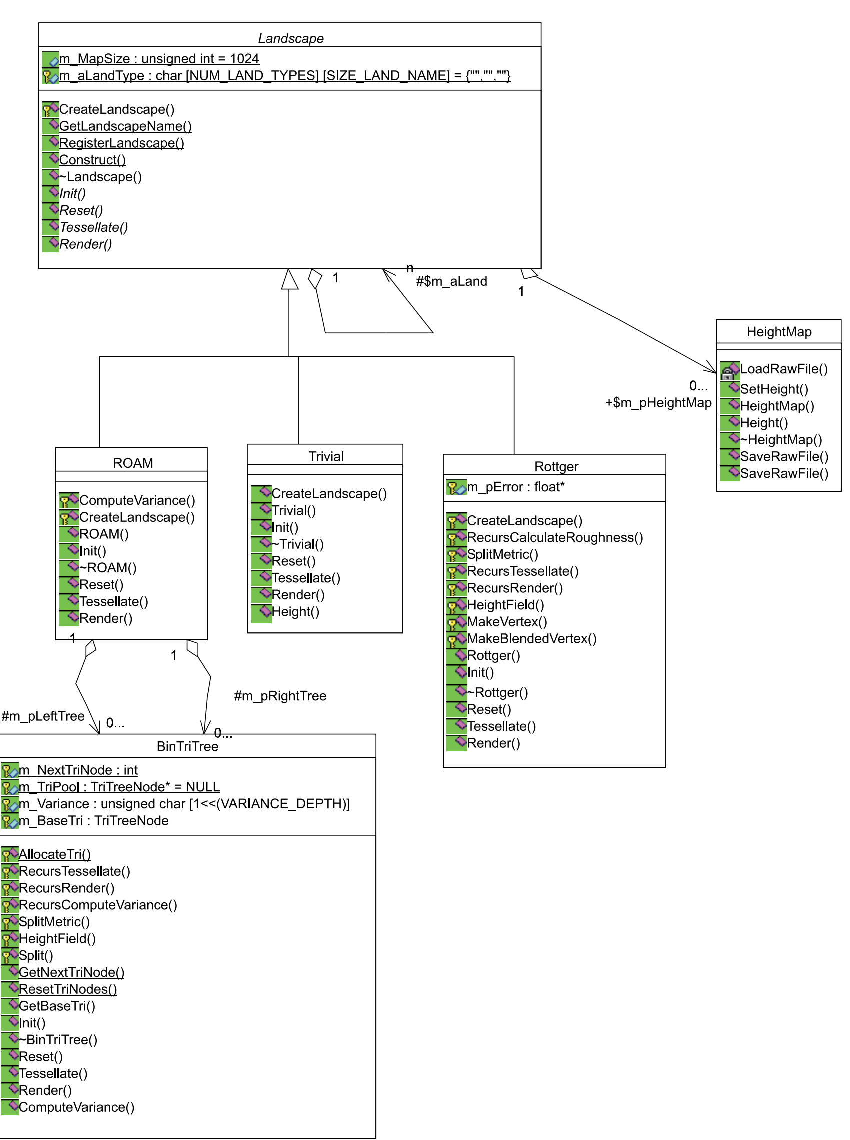


Figura 23: Classes do Graphic Engine correspondentes ao terreno.

atributo é estático porque para nós há somente um mapa por ambiente do motor, o que justifica a existência de apenas um tamanho de mapa entre as classes definidas pelo mesmo.

Um outro ponto que deve ser discutido na Figura 23 é o conjunto de métodos que são definidos num terreno: `Init`, `Reset`, `Tessellate` e `Render`. Estes métodos casam com a estrutura não só dos algoritmos discutidos anteriormente, mas com a quase totalidade dos algoritmos vigentes. O `Init` inicializa ou aloca memória para as estruturas de dados correspondentes ao nosso algoritmo, no nosso caso, Quad Tree ou *árvore binária*. O `Reset` define o estado inicial dessas estruturas de dados. O `Tessellate` modifica o estado dessa estrutura baseado na técnica do algoritmo a cada quadro. Por fim, o método `Render` renderiza o terreno.

Com isso, temos três algoritmos de edição de cenários que estão disponíveis no *City Editor*. Estes algoritmos são oferecidos, em versões simplificadas, na ferramenta pois a mesma possui também uma natureza educacional como foco. Assim poderão ser feitos testes e mecanismos de análise de desempenho em futuras versões do projeto. Além disso, o fato de termos várias versões disponíveis, nos permite escolher o algoritmo que melhor se adeque ao ambiente em questão.

Por meio dessas estruturas, definimos um ambiente simples e extensível para aplicações de edição de cenário com apenas um terreno. Podendo, agora, o trabalho se concentrar em aspectos de mais alto nível de interface homem máquina.

5 *Usando Interpretadores para Entrada e Saída de Arquivo*

Nesta seção, são definidos os mecanismos usados para alcançar compatibilidade entre o motor e os formatos de arquivo que o editor compreende. Esse mecanismo teria de ser o mais independente possível dos objetos do motor, além de ser facilmente acessível por parte do usuário e obedecer a mecanismos de orientação a objetos.

Foram feitos dois mecanismos, organizados em pacotes, que obedeceram a este princípio dentro do Building Editor e do City Editor. Veremos estes mecanismos a fundo e chegaremos a uma conclusão sobre o padrão aplicado dentro destes dois pacotes.

5.1 Por que não usar MFC para escrever arquivos?

MFC oferece mecanismos para integrar de maneira simples a leitura e escrita de arquivos com o código gerado pelo *Class Wizard*, mecanismo do compilador *Visual Studio* que facilita a criação e manutenção de classes. Desta maneira, seria razoavelmente simples gerar código para escrita de um determinado documento de uma aplicação MFC em arquivo, e, ainda, integrar esta escrita com a interface gráfica de maneira direta.

No entanto, isto não é aconselhável. Tornar uma parte tão vital de nossa aplicação dependente de uma API que pode ser mudada por qualquer outra em um futuro próximo não seria aconselhável.

A saída mais correta é, com certeza, usar funções de manipulação de arquivos da STL (OTTEWELL, 2002), biblioteca padrão de C++ (DEITEL; DEITEL, 2001; STROUSTRUP, 1997). Usando essas funções tornamos este código mais independente de plataformas e de API's usadas, além de tornarmos este código mais separado da interface gráfica. É por isso que nas subseções seguintes todo o código será dependente da biblioteca padrão.

5.2 BEiostream

É o primeiro pacote que trata de leitura e escrita em arquivos. Apesar do formato do arquivo escrito ser relativamente simples, o importante é notar que aqui temos uma estrutura que transforma uma leitura de arquivos em um intercâmbio de objetos já interpretados.

Vejamos, então, como se escreve em arquivo usando esta biblioteca. Para melhor entendimento do que estamos falando, consultar, sempre que necessário, o diagrama de classes desta biblioteca mostrado no Apêndice E (página 139 desta dissertação).

Primeiramente, temos uma `BuildingScene` que no nosso caso é definida como a cena que contém um prédio, além de todos os materiais e texturas a serem intercambiados com arquivos. No entanto, essa `BuildingScene`, como tudo mais no pacote `BE_iostream`, é parte de uma árvore (no caso de um prédio é uma árvore apenas, mas em ambientes mais complexos pode vir a ser um grafo) montada para englobar todos os objetos a serem manipulados do arquivo. De fato, `BuildingScene` é a raiz desta árvore e possui todas as informações em forma de atributos da classe.

Uma estrutura bastante interessante dentro deste contexto é o `PrimitiveContainer`, que é baseado em `templates`, além de ser uma estrutura rica em termos organizacionais. Como o próprio nome já diz, é um repositório de objetos, bem simplificado. De fato, sua estrutura tem por trás um `vector`, tipo de dado padrão de C++ que une os benefícios de acesso dinâmico de um *array* e a facilidade de inserção de uma lista, e todas as operações são feitas sobre esta estrutura.

O `PrimitiveContainer` tem todas as operações inerentes a qualquer classe desta biblioteca que sejam passíveis de serem lidas ou escritas de arquivo. Essas operações são `Import_From_File` e `Export_To_File` e servem para que todos os objetos armazenados no container sejam lidos ou escritos em arquivo respectivamente. Vejamos, então, a estrutura simplificada dessas operações dentro do `PrimitiveContainer`.

No Código 19, vê-se claramente que o objetivo do `PrimitiveContainer` é somente escrever e ler arquivos. Pode-se ver que, até no construtor da classe, um *stream* para leitura de arquivo é passado para que a seqüência de componentes que pertence a este container seja construída, ou seja, lida de arquivo, e o container em si seja formado. Este é o mecanismo de funcionamento de todos os objetos desta biblioteca.

Ao final de tudo, temos nossos tipos de *stream* padrão, o `BE_istream` e o `BE_ostream` (ver Apêndice E, página 139), baseados na biblioteca padrão de C++, a Standard Library ou STL. E temos também um conjunto de objetos que se constroem ou se escrevem de forma persistente a partir desses *streams*.

Código 19: trechos de código do PrimitiveContainer.

```
template <typename T> class PrimitiveContainer {
public:
... //attributes
    //constructors
    PrimitiveContainer(udword Number, BE_istream &BEFile)
    {
        Import_From_File(Number, BEFile);
    }
    PrimitiveContainer() {}
    //methods
    void Import_From_File(udword Number, BE_istream &BEFile)
    {
        Num_Items = Number;
        if (Num_Items > 0) {
            T::Verify_File(BEFile);
            for (int i = 0; i < Num_Items; i++) {
                Items.push_back(T(BEFile));
            }
        }
    }
    void Export_To_File(BE_ostream &BEFile) {
        if (Num_Items > 0) {
            T::Export_Verification(BEFile);
            for (int i = 0; i < Num_Items; i++) {
                Items[i].Export_To_File(BEFile);
            }
        }
    }
    ...//other methods
};
```

Para observar o comportamento de alguma classe inerpretadora, o Código 20 mostra parte da classe `BE_Texture`.

Como se pode ver, existem as estruturas padrão de importação e exportação de arquivo. O `BE_Texture` é um objeto que faz *parser*, ou seja trata leitura e escrita, no arquivo do nome e caminho da textura.

Com isso, essas classes formam uma estrutura de dados hierárquica que organiza toda a informação presente no arquivo. Dado que esta estrutura está toda montada, basta fazer nosso motor compreendê-la e construir seus objetos apropriadamente baseado nela.

5.2.1 Estruturas de Importação

Para o motor compreender a estrutura montada pelo `BEiostream` definimos uma classe chamada `CImporter`, essa classe é a última estrutura de importação propriamente dita que não foi abordada até então. É uma estrutura de complexidade relativamente simples e utiliza a biblioteca `BEiostream` para fazer o *parser* antes de importar os objetos já criados.

Além de importar os objetos já montados pela `BEiostream`, ela oferece mecanismos de busca dos objetos criados; isso é muito importante para construção dos objetos depois que o *parser* já foi feito, esta técnica será melhor detalhada a seguir no Código 21. Ou

Código 20: Escrita em formato proprietário de arquivos.

```
#include "BE_Texture.h"

BE_Texture::BE_Texture(const CTexture &t){
    Name = t.GetName();
    File = t.GetFileName();
}

BE_Texture::BE_Texture(BE_istream &BEFile){
    Import_From_File(BEFile);
}

void BE_Texture::Verify_File(BE_istream &BEFile){
    string  TEX;
    BEFile >> TEX;
    if (TEX.compare("TEXTURES:")!=0) {
        throw "Bad Texture signiture in bld file";
    }
}

void BE_Texture::Import_From_File(BE_istream &BEFile){
    BEFile >> delimString(Name) >> delimString(File);
}

void BE_Texture::Export_Verification(BE_ostream &BEFile){
    BEFile << "TEXTURES:" << endl;
}

void BE_Texture::Export_To_File(BE_ostream &BEFile){
    BEFile << delimString(Name) << endl;
    BEFile << delimString(File) << endl << endl;
}
```

seja, depois que já temos toda a estrutura de objetos equivalentes ao arquivo já criado; cabe ao motor transformar essa estrutura para o formato de seu entendimento.

O `CImporter` possui uma `BuildingScene` como argumento, e utiliza esta cena para importar um arquivo de tal maneira que fica transparente para o usuário a existência do arquivo. Os objetos, como o prédio `CBuilding` do `GraphicEngine`, tem um construtor simples que só recebe o `CImporter` e o objeto do tipo `BE_Building` e faz a importação dos materiais e texturas necessários para a existência do prédio e que ainda não estão nos seus respectivos `Managers` (`MaterialManager` e `TextureManager`). É importante salientar que se o ID ou nome de um dado objeto já existir no manager, este objeto será é importado para evitar a duplicação de objetos.

No Código 21 temos o construtor do prédio por meio de estrutura de importação. Este código será abordado como um exemplo mais geral, e nos permitiremos não exibir código de importação de um material ou textura por ser relativamente parecido com este.

Observe que o `CBuilding` só incorpora as informações que estão presentes na classe `BE_Building`, e cria os respectivos materiais e texturas a partir dos métodos `FindMaterial` e `FindTexture` do `CImporter`, quando eles ainda não existirem no respectivos `Managers`. Os métodos `FindMaterial` e `FindTexture` do `CImporter` fazem busca nos `PrimitiveContainers` que estão como atributos da classe `CBuildingScene`. Para

Código 21: Importando arquivo depois de feito o parser.

```
CBuilding::CBuilding(const CImporter &Importer, const BE_Building &Building){
    int i;
    BE_Material *pm;
    BE_Texture *pt;
    OBJECT_ID id;
    this->SetObjectType(BUILDING);
    this->SetObjectID(UNDEFINED_ID);
    this->m_MeshID=UNDEFINED_ID;
    this->m_pBuildingMesh=NULL;
    m_Modified=false;
    this->SetName(Building.Name);
    // see the default material properties
    if(!g_MaterialManager.ExistsBEObject(Building.DefaultMaterial)) {
        Importer.FindMaterial(Building.DefaultMaterial, &pm);
        CMaterial m(*pm);
        m.SetObjectID(IDFACTORY.GetID());
        g_MaterialManager.AddBEObject(m);
    }
    g_MaterialManager.GetBEObjectId(Building.DefaultMaterial, &id);
    this->m_DefaultMaterial=id;
    for(i=0;i<5;i++) {
        // see the material properties
        if(Building.Materials[i] == Building.DefaultMaterial) {
                this->m_MaterialID[i]=UNDEFINED_ID;
        }
        else {
            if(!g_MaterialManager.ExistsBEObject(Building.Materials[i])) {
                Importer.FindMaterial(Building.Materials[i], &pm);
                CMaterial m(*pm);
                m.SetObjectID(IDFACTORY.GetID());
                g_MaterialManager.AddBEObject(m);
            }
            g_MaterialManager.GetBEObjectId(Building.Materials[i], &id);
            this->m_MaterialID[i]=id;
        }
        // see the texture properties
        if(Building.Textures[i] == "NONE") {
                this->m_TexturesID[i]=UNDEFINED_ID;
        }
        else {
            if(!g_TextureManager.ExistsBEObject(Building.Textures[i])) {
                Importer.FindTexture(Building.Textures[i], &pt);
                CTexture t(*pt);
                t.SetObjectID(IDFACTORY.GetID());
                g_TextureManager.AddBEObject(t);
            }
            g_TextureManager.GetBEObjectId(Building.Textures[i], &id);
            this->m_TexturesID[i] = id;
        }
    }
    this->m_Length = Building.Length;
    this->m_Width =  Building.Width;
    this->m_Height = Building.Height;
    this->SetBuildingType((CBUILDINGTYPE)Building.LotType);
}
```

mais detalhes sobre a interação destes objetos, recorrer ao Apêndice E (página 139).

Com isso temos uma estrutura que organiza melhor o *parser* além de tornar a leitura de arquivos o mais independente possível da estruturação do motor.

5.3 CEiostream

Agora que já vimos como funciona a leitura e escrita em arquivo com a biblioteca `BEiostream` e com a estrutura `CImporter`, daremos uma explanada superficial nas classes que compõem a biblioteca `CEiostream`. Esta biblioteca também utiliza objetos que se escrevem em arquivo como uma estrutura hierárquica única.

Assim como o pacote `BEiostream` nosso pacote define streams próprios(`CE_istream`, e `CE_ostream`) que têm como objetivo validar nosso arquivos por meio de cabeçalhos próprios. Após a validação dos objetos, os streams tratam de escrever os arquivos por meio dos métodos `Import_From_File` e `Export_To_File`.

Por conta da semelhança deste pacote com o pacote `BEiostream`, definiremos os objetos que o compõem por meio da Tabela 4.

Tabela 4: Objetos definidos pelo `CEiostream`.

CE_Vector2D	Vetor 2D que é utilizado no ato de exportação para identificar posições bidimensionais no terreno.
CE_Vector3D	Vetor 3D que é utilizado no ato de exportação para identificar tridimensionais no espaço.
CE_Building	As informações sobre as instâncias do prédio no editor como posição, rotação e posição da calçada.
CE_Color3f	Vetor de cor em formato RGB onde os valores de R, G e B são valores ponto flutuante entre 0.0 e 1.0.
CE_Color3i	Vetor de cor em formato RGB onde os valores de R, G e B são inteiros entre 0 e 255.
CE_Landscape	Define o formato de exportação do prédio para arquivo com as cores que definem o mapa de alturas.
CE_PrimitiveContainer	Agrupa os prédios e estradas para escrever em arquivo.
CE_Road	Define o formato de escrita de uma rodovia em arquivo.
CitySceneMain	Define o número de prédios e rodovias em um arquivo.
CityScene	Agrupa todos os objetos que definem o modelo em um único objeto.

5.4 VRMLSaver

O `VRMLSaver` é o responsável pela compatibilidade do nosso software com VRML, o que torna o nosso software também compatível com a maioria das plataformas disponíveis no mercado.

Para entender o mecanismo usado para exportar o ambiente para VRML, vamos, primeiro entender como funciona VRML e, depois disso, explicar o mecanismo usado.

5.4.1 VRML, uma visão rápida

VRML (acrônimo de *Virtual Reality Modelling Language*), é uma linguagem descritiva de ambientes 3D. Ela é, hoje, um padrão em termos de descrição de ambientes 3D porque, desde 1997, houve um grande esforço da indústria no desenvolvimento de softwares de visualização no sentido de trabalhar com ambientes descritos neste formato.

A idéia de VRML é razoavelmente simples: desenvolver um grafo baseado em nós, onde cada nó é uma descrição de um objeto em um mundo virtual. Existem vários tipos de nós dentre os quais podemos enumerar nós para definição de texturas, materiais e coordenadas. Estes nós em conjunto formam o que chamamos de *Mundo*. No Código 22 temos um mundo VRML simples, incluindo o cabeçalho requerido para todos os arquivos deste padrão.

Código 22: Exemplo de código VRML.

```
#VRML V2.0 utf8
Shape {
    appearance Appearance {
        material Material {}
    }
    geometry Cylinder { }
}
```

O interessante deste código simples é que ele reflete totalmente a estrutura hierárquica de VRML com nós, como o Shape, que contém atributos que podem ser ou não outros nós. No caso do Código 22 a estrutura se torna o que está simbolizado na Figura 24.

O que temos que fazer, no caso de VRML, assim como fizemos com o `BEiostream` e `CEiostream` é construir uma estrutura organizacional que faça o trabalho de relacionar os nós de uma maneira condizente com a especificação da linguagem para, somente depois, se escrever a estrutura em arquivo.

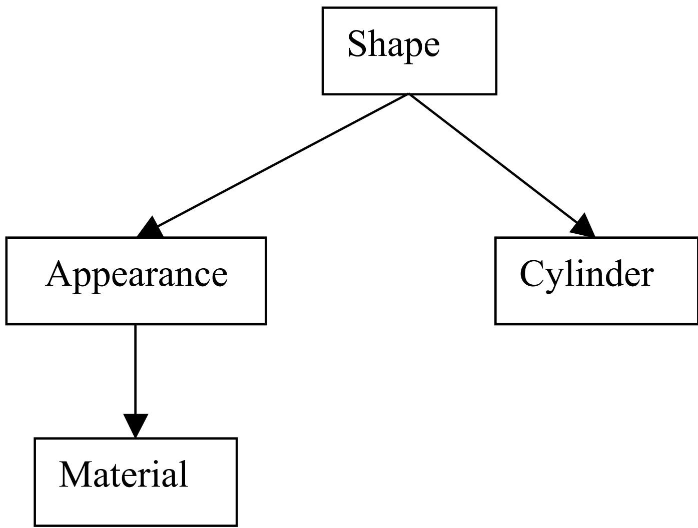


Figura 24: Estrutura do código da Tabela 22.

5.4.2 Gerando as classes VRML

Seguindo o mesmo formato da biblioteca `BEiostream` e `CEiostream`, precisamos, primeiramente, modelar os componentes de VRML que se adeqüam ao motor, no estado atual, e depois definir a escrita de arquivos sobre estes objetos já estruturados.

Para construir as classes de manipulação VRML, foi consultada a especificação da linguagem disponível no site do Web3D Consortium (2001).

Antes das classes básicas terem sido construídas, os tipos básicos de dados foram modelados para que tudo condissesse com a especificação. Os objetos básicos podem ser encontrados na Figura 25.

Observa-se que os tipos definidos como MF são, na especificação de VRML, conjuntos de objetos do tipo SF, e que todos os objetos do tipo MF são, no nosso caso, instâncias do `Template` de uma classe padrão definida: a `MFStructure`. Isso simplificou o código fonte final. No Código 23 temos o esqueleto da classe que define o comportamento da classe base para todo tipo de estrutura que em VRML seria MF.

Outra coisa interessante é a definição de `SFNode` no padrão VRML, que na verdade em VRML não é um tipo de dado, mas um ponteiro para o tipo de dado que seria o nó propriamente dito, aqui batizado de `VRMLNode`.

Depois de feita a definição dos objetos básicos, objetos mais complexos foram modelados de acordo com seu formato na especificação VRML.

É neste ponto que pomos algumas definições de VRML em questionamento. Dado que o padrão permite que não haja uma hierarquização dos nós por meio de herança para melhor delimitar a linguagem, propondo que todos os atributos de um nó sejam declarados

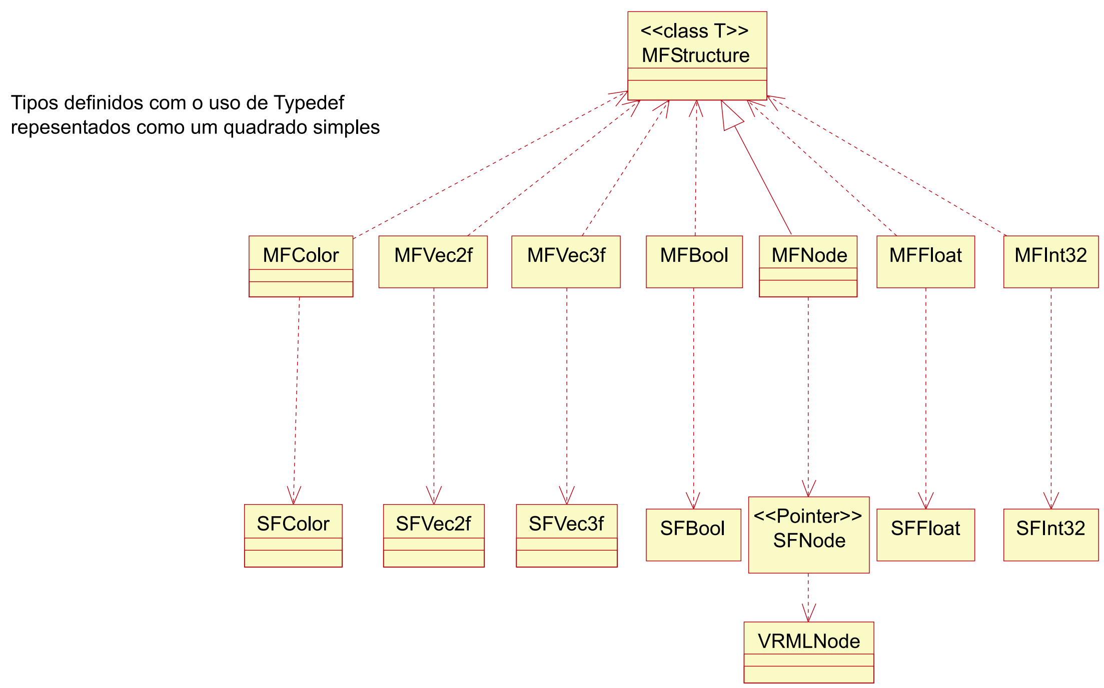


Figura 25: Objetos básicos de VRML modelados.

como `SFNode`, o que seria um absurdo em alguns casos, como mostrado no Código 24.

Esta definição foi retirada da página de especificação de VRML e mostra, a princípio, que a linguagem permite que coisas absurdas, como definir uma forma onde a aparência seria um nó `Geometry` e a geometria seria uma `Appearance`, sejam definidas. Cabe, então, ao visualizador de mundos VRML tratar deste tipo de situação.

Foi por isso que, mesmo baseando-nos na especificação, a definição final dos nós VRML usados terminou sendo simplificadamente estruturada da forma descrita na Figura 26.

Observa-se melhor o resultado final e a correlação de todos os nós VRML no diagrama

Código 23: Uma MF structure.

```
template<class T>
class MFStructure : public vector<T>
{
public:
    typedef vector<T>::const_iterator const_iterator;
    typedef vector<T>::iterator iterator;
    void print_list(ostream &File) const;
    virtual void remove_all();
protected:
    virtual void print_separator(ostream &File) const{
        File << ", ";
    }
    virtual void print_initial(ostream &os) const {
    }

};
```

Código 24: Shape com dois nós filhos de definição demasiadamente genérica.

```
Shape {
  exposedField SFNode appearance NULL
  exposedField SFNode geometry   NULL
}
```

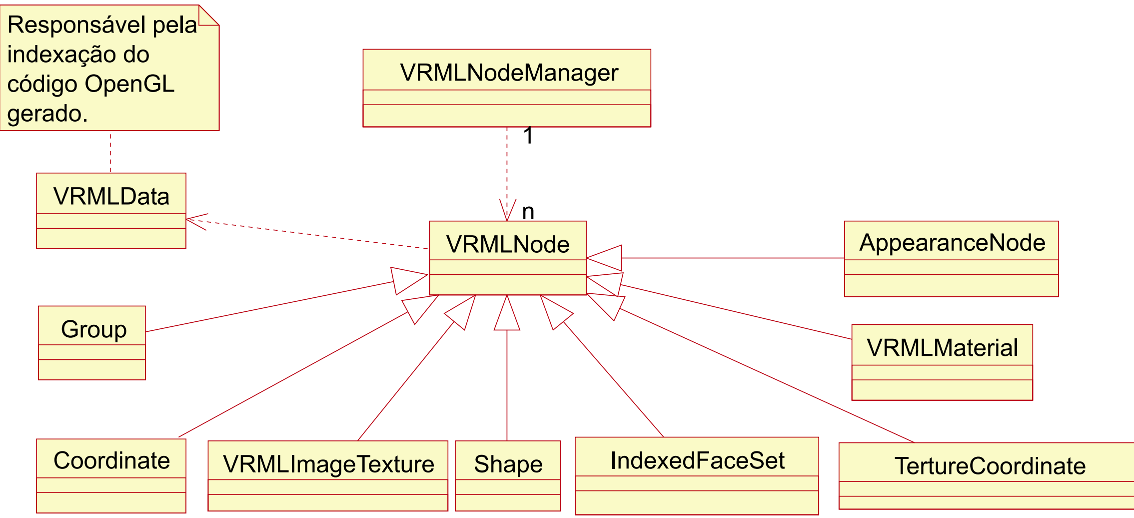


Figura 26: Estrutura simplificada para definição dos nós VRML.

de classes completo encontrado no Apêndice G (página 141 deste documento).

Outro ponto importante desta seção é a forma como os nós VRML são escritos em arquivo. O nó `VRMLNode` ajuda a identificar se um determinado nó já foi escrito no arquivo. Se isso já foi feito e estamos precisando escrever o nó novamente, ele usa as primitivas `DEF` e `USE` de VRML que permitem que macros sejam definidas e usadas. Estas macros possibilitam que um nó já renderizado seja referenciado apenas pelo seu nome, o que dá margem a certos mecanismos de aceleração por parte do visualizador VRML. No Código 25 temos o algoritmo que faz esse mecanismo.

Código 25: Usando DEF e USE para exportar VRML.

```
void VRMLNode::ExportVRML(ostream &File) const
{
    if(!alreadyExported) {
        if(this->name != "")
            File << "DEF "<< name << " ";
        File << typeName << " {";
        VRMLData::level++;
        VRMLData::FinishLine(File);
        WriteData(File);
        VRMLData::level--;
        VRMLData::FinishLine(File);
        File << "}";
        if(this->name != "")
            alreadyExported = true;
    }
    else {
        File << "USE " << name;
    }
}
```

A classe VRMLData fornece mecanismos de identação do código final e por isso é usada no exemplo. Além disso, vê-se aqui claramente que a única função das subclasses é sobrescrever os métodos WriteData para que os atributos sejam colocados dentro do esqueleto do nó, escrito com base no código do VRMLNode. No Código 26, temos o exemplo de como se comportaria a classe Shape que herda de VRMLNode.

Código 26: Comportamento de uma classe do VRMLSaver.

```
Shape::Shape(SFNode ap, SFNode ge) : VRMLNode("Shape")
{
    appearance = ap;
    geometry = ge;
}

Shape::~Shape()
{
    if(appearance!=NULL)
        delete appearance;
    if(geometry!=NULL)
        delete geometry;
}

void Shape::WriteData(ostream &File) const
{
    File << "appearance " << appearance;
    VRMLData::FinishLine(File);
    File << "geometry " << geometry;
}
```

Observe, no Código 26, que toda a complexidade envolvida para escrita em arquivo está no método WriteData, e tudo o que ele faz é escrever os argumentos do nó.

Outro ponto importante é que foi definido um gerenciador dos nós VRML que serão usados mais de uma vez, o VRMLNodeManager. Neste gerenciador os elementos gerenciados são ponteiros para nós, que são os elementos manipulados segundo a especificação de VRML. O VRMLNodeManager oferece mecanismos para que se busque uma instância de um nó já declarado. Logo, todo elemento que está neste gerenciador é na verdade um DEF, e esta estrutura, junto com o escopo definido para o VRMLNode, se torna, então, um gerenciador de DEF's e USE's. Na Figura 27, temos o escopo desta classe para que se tenha idéia do mecanismo.

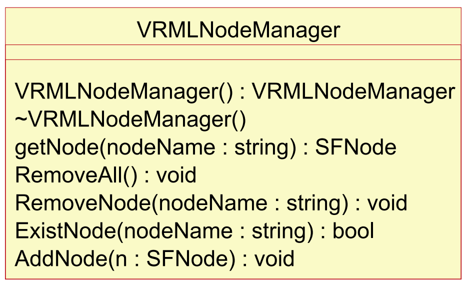


Figura 27: Gerenciador de nós VRML.

Não mostraremos aqui o código desta classe por conta da complexidade envolvida

neste código. Mas gostaríamos de lembrar que o código desenvolvido se encontra na referência (ARAÚJO FILHO, 2003).

Na Figura 28, temos um exemplo de trecho de um código VRML gerado pelo editor. Observe como é bem estruturado em termos de alinhamento, e como utiliza as primitivas DEF e USE para ter um código final menor e mais eficiente. O código completo de onde este trecho foi retirado está disponível no Apêndice L (página 149), a sua visualização se encontra na Figura 29.

```
#VRML V2.0 utf8
#############################################################
#      This VRML World is generated by Building Editor      #
#         Building Editor is autory of Mozart Filho         #
#     And a product of Federal University of Pernambuco     #
#############################################################
Group {
    children [
    Shape {
        appearance Appearance {
            material DEF _DEFAULT_MATERIAL_ Material {
                diffuseColor 0.8 0.8 0.8
                specularColor 0 0 0
                emissiveColor 0 0 0
                transparency 0

            }
            texture DEF CASA9 ImageTexture {
                url "casa-9.jpg"
            }
        }
        geometry IndexedFaceSet {
            coord Coordinate {
                point [0 0 0, 0 0 5, 5 0 5, 5 0 0, 0 6 0, 0 6 5, 5 6 5, 5 6 0]
            }
            coordIndex [4, 0, 1, -1, 4, 1, 5, -1]
            texCoord TextureCoordinate {
                point [0 1, 0 0, 1 0, 0 1, 1 0, 1 1]
            }
            texCoordIndex [0, 1, 2, -1, 3, 4, 5, -1]
        }
    }
    Shape {
        appearance Appearance {
            material USE _DEFAULT_MATERIAL_
            texture DEF CASA11 ImageTexture {
```

Figura 28: Trecho de código VRML gerado pelo Building Editor. Visualização em cores pelo editor VrmlPad (ParallelGraphics, 2003), feita para facilitar o trabalho dos constrututores de mundos virtuais em VRML

5.5 O padrão de projeto Interpreter

Agora que já terminamos as estruturas de leitura e escrita em arquivo, veremos que estas estruturas se encaixam em mais um padrão de projeto (GAMMA, 1995), o padrão *Interpreter*. Este padrão, por sua vez, é a organização de estruturas de tal maneira que *parsers* sejam feitos para transformar as instruções em objetos hierarquicamente dispostos.

A intenção primordial deste padrão é: dada uma linguagem, definir uma gramática junto com um interpretador que usa a representação orientada a objeto para interpretar a sentença na linguagem desejada.

Figura 29: Visualização do código mostrado na Figura 28

O padrão *Interpreter* oferece funções para representar ou avaliar a gramática em questão, o que acontece em todos os casos até então.

Por conta disso, nos pacotes `BEiostream`, `CEiostream` e `VRMLSaver` o padrão *Interpreter* foi utilizado desde o princípio.

A hierarquia definida pelo padrão *interpreter* é compatível com gramáticas de linguagens descritivas e, por isso, deveria ser usada na maioria dos editores de cenários conhecidos. Entretanto essa hierarquia deve ser levada em conta se a quantidade de objetos definidos por ela comprometer o requisito memória das máquinas conhecidas no mercado, como aconteceria se a usássemos para fazer o *parser* em linguagens de programação como C, C++, Java e Pascal. Neste caso, alguns analisadores léxicos e sintáticos são mais indicados, pois gerariam os dados direto para o formato final no lugar de gerar uma árvore ou grafo em um formato intermediário, como propõe o padrão.

Para mais informações sobre o padrão *Interpreter*, consultar o Apêndice L na página 151 deste documento.

6 *Acoplamento com MFC*

O presente capítulo aborda um item muito importante deste trabalho, que é a compatibilidade da arquitetura proposta com a biblioteca MFC – *Microsoft Foundation Classes*. Este item não é muito comentado na literatura e foi um grande desafio tornar a aplicação integrada com esta tecnologia.

6.1 Por que MFC?

Inicialmente estávamos procurando a biblioteca na qual trabalharíamos para definir a interface gráfica e os eventos do editor na plataforma Windows, plataforma na qual os nossos softwares de realidade virtual e de modelagem se baseiam. Procuramos uma opção que funcionasse dentro do contexto de orientação a objetos.

Durante a procura de bibliotecas de interface, houve também questionamento de utilizarmos bibliotecas independentes de plataforma como FLTK (SourceForge.net, 2003b), o *Fast Light Toolkit*, e GTK (GTK, 2003), o *GIMP Toolkit*.

Entretanto, na primeira opção, apesar do FLTK ser uma biblioteca bastante robusta, completa e com boa documentação, foi feita uma crítica com relação ao mecanismo de IDE, ou *interface de desenvolvimento*, disponível para a criação de ambientes nesta biblioteca, a FLUID ou Fast Light User Interface Designer, este mecanismo apesar de oferecer uma maneira rápida de construção de ambientes é muito arcaico quando comparado com o Visual Studio da Microsoft ou C++ Builder da Borland, por exemplo, o que exigiria do programador um tempo maior para desenvolvimento. Nada que não fosse contornável em um grande projeto, mas um problema de grande escala para um projeto feito por apenas um programador. Já o GTK, além de não possuir nenhum ambiente de desenvolvimento de interfaces gráficas, também possui uma documentação muito ruim com vários links faltando no único tutorial da página e com muitos dos ítens do manual de usuário precisando de voluntários para a documentação.

Por causa dos ítens enumerados acima, pela farta documentação existente a respeito da biblioteca MFC e da já familiaridade do programador com a biblioteca, MFC foi

escolhida.

Fazendo uma procura de artigos relacionados à construção de ferramentas 3D, encontramos muitas referências mostrando a possibilidade de construção de aplicações 3D usando MFC (ver GameDev; HELIUM, entre outras).

No entanto, estas fontes serviram apenas como primeiro passo dentro de nossa jornada de implementação. Outros artigos e home pages que estão relacionados somente com MFC foram lidos neste processo e ajudaram a esclarecer diversas dúvidas de implementação. Esses artigos podem ser vistos nas referências (C & MFC – Table of Contents; DevCentral; The Code Project).

6.2 A arquitetura documento/visão

A parte do *framework* MFC que é mais visível ao usuário final é a arquitetura documento/visão. Por sorte nossa, é também a parte mais interessante dentro do nosso contexto do editor de cenários. A parte mais trabalhosa dentro da definição de um *framework* MFC é justamente escrever as classes de Documento e Visão (Document/View).

A classe `CDocument` provê a funcionalidade básica para classes de documento definidas pelo programador. Um documento representa a unidade de dados que o usuário tipicamente abre com o comando **Abrir** no menu arquivo e salva no comando **Salvar** também no menu arquivo.

A classe `CView` provê as funcionalidades básicas para classes de visão definidas pelo programador. Uma visão é anexada a um documento e age como uma intermediária entre o documento e o usuário: a visão renderiza uma imagem do documento na tela e interpreta os eventos do usuário como operações no documento.

A Figura 30 mostra a relação entre o documento e sua visão.

A implementação documento/visão na biblioteca da classe separa os dados da sua representação gráfica e das operações dos usuários nos dados. Todas as mudanças dos dados são gerenciadas através da classe documento. A visão chama a interface documento para acessar e atualizar os dados.

A arquitetura documento/visão permite que um documento esteja estruturado de várias maneiras, tendo tantas visões quanto quisermos de um mesmo documento. Em uma determinada aplicação, como um editor HTML, podemos ter mutuamente a representação textual do código HTML e pode-se ter a representação da página graficamente, como mostra a Figura 31.

Essa vantagem de suporte a múltiplas visões é uma peça chave em vários tipos de

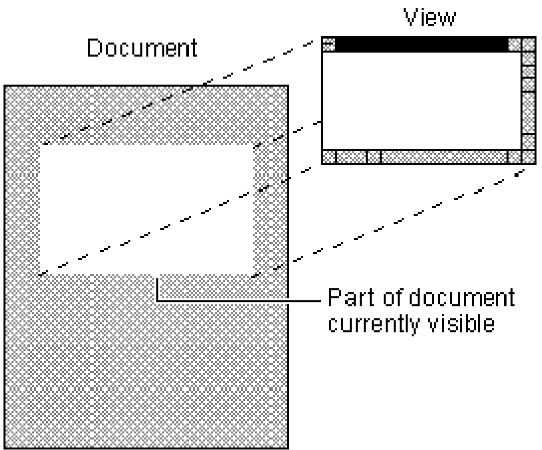


Figura 30: Exemplo da arquitetura documento/visão. FONTE: MSDN Home (2002)

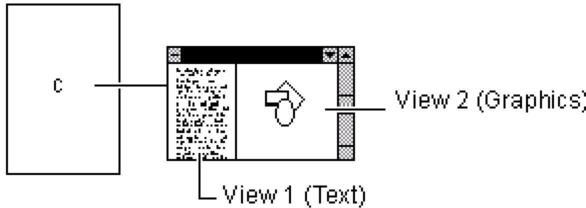


Figura 31: Documento na arquitetura Document/View. FONTE: MSDN Home (2002)

aplicação, incluindo a nossa. Para que todas as visões percebam a alteração de um documento em MFC é necessária uma chamada ao método `CDocument::UpdateAllViews`, que sincroniza todas as visões.

Este cenário de múltiplas visões seria bem mais complexo de codificar sem a separação entre os dados e a visão, particularmente se as visões guardassem os dados por si mesmas. Com Documento/Visão essa codificação é facilitada, já que o framework faz a maior parte do trabalho de coordenação.

6.3 A criação de Viewports

Uma vez apresentada a estrutura organizacional de MFC, vejamos como adaptar esta estrutura ao uso de uma aplicação *multi-viewport*. A primeira idéia, vista na minha opinião como a mais sensata, seria que cada viewport fosse uma visão diferente de um mesmo documento, que seria o ambiente 3D modelado. Logo definiríamos a classe Documento com o ambiente e uma visão para cada janela ativa da aplicação.

No entanto, definir uma visão para cada janela ativa da aplicação seria um trabalho exaustivo, além de extremamente redundante, dado que muitos métodos comuns existem para uma coleção de estruturas idênticas que se diferenciam apenas no âmbito comporta-

mental.

Foi pensando nisso, e se baseando em um artigo intitulado “Creating 3D tools with MFC” do site da GameDev (2001), que se construiu uma estrutura hierárquica que cuidasse da janela ativa independentemente do tipo de visão que se quisesse do ambiente.

Primeiramente, olhemos a estrutura como um todo que está modelada no Apêndice H (página 143). A primeira classe, a `OpenGLWnd` herda da `CView` padrão e, com isso, define que toda visão OpenGL é também uma visão MFC padrão a qual tem métodos para criar uma janela padrão e tratar eventos comuns como destruição e criação de janelas.

Entre os outros métodos padrão podemos escolher para o ambiente OpenGL uma determinada quantidade de bits de cores, uma determinada profundidade do *depth buffer*, responsável pela checagem de ordem dos polígonos de tal maneira que sabemos se devemos desenhar algum polígono posicionado na frente de um outro polígono, ou do *stencil buffer*, utilizado para simular sombra ou reflexão em um ambiente virtual.

No entanto, essas escolhas são genéricas para qualquer *viewport* criada dentro de uma mesma aplicação. E por isso, essas opções têm que ser pensadas para o pior caso dentro das *n viewports* que a aplicação possuir.

Vejamos, agora, segundo nível da hierarquia, onde são definidos dois tipos de visão: a visão ortográfica e a visão em perspectiva. A visão em perspectiva é aquela com que lidamos nos nosso dia a dia, ou seja, a que nos permite ter a noção de com que distância os objetos aparecem menores. Já na visão ortogonal, esta noção de perspectiva não funciona da mesma maneira, um objeto disposto a cem metros ou a um metro seria percebido como possuindo as mesmas dimensões.

OpenGL permite que, tanto na visão ortográfica, quanto na visão em perspectiva, sejam feitas operações de rotação dos sólidos de tal maneira que estes sólidos sejam visíveis de outro ângulo. Contudo, resolvemos desabilitar esta opção para nossa visão ortogonal padrão.

No Código 27 temos o método que adapta a visão ortográfica a cada redimensionamento da janela na aplicação dentro da classe `COrthographic`.

Observe que, no Código 27, o modo de projeção é habilitado por uma matriz; e a operação OpenGL para criar uma visão ortogonal é `glOrtho()`.

Vejamos também no Código 28 o método para cuidar do redimensionamento de uma janela OpenGL em perspectiva, a classe que cuida desta visão em perspectiva é a `CPerspective`. Uma visão em perspectiva é definida a partir do comando OpenGL `gluPerspective`.

Essas classes, além disso, tratam eventos de mouse dentro da janela ativa para na-

Código 27: Adaptando a visão ortográfica ao redimensionamento da janela.

```
void COrthographic::OnSize(UINT nType, int cx, int cy)
{
    COpenGLWnd::OnSize(nType, cx, cy);

    if ( 0 >= cx || 0 >= cy || nType == SIZE_MINIMIZED )
        return;

    SetContext();
    glViewport( 0, 0, cx, cy );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    //codigo extra...
    glOrtho( left, right,
        bottom, top, -400.0f, 200.0f );
    glMatrixMode( GL_MODELVIEW );
}
```

Código 28: Redimensionando a visão em perspectiva

```
void CPerspective::OnSize(UINT nType, int cx, int cy)
{
    COpenGLWnd::OnSize(nType, cx, cy);

    if ( 0 >= cx || 0 >= cy || nType == SIZE_MINIMIZED )
        return;

    SetContext();
    glViewport( 0, 0, cx, cy );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective(45.0f, (float)(cx)/(float)(cy), 1.0f,
                   100.0f);

    glMatrixMode( GL_MODELVIEW );
}
```

vegação. Os eventos de mouse em qualquer classe filha de `COrthographic` teriam que responder da mesma maneira, dado que os dispositivos de navegação seriam iguais para qualquer visão ortogonal existente. Maiores informações sobre o dispositivo de navegação estão disponíveis na Seção 7.

Assim temos uma estrutura que, baseada em mensagens padrão de MFC e do Windows, forma uma estrutura hierárquica de delegação de eventos. Essa estrutura lembra outro padrão de projeto Gamma (1995) até agora não usado, o *Chain of Responsibility*, que esta sucintamente explicado no Apêndice L (página 150).

Na *Chain of Responsibility* os eventos são tratados por um `switch` e depois os respectivos métodos pertencentes ao objeto são invocados. Aqui, aproveitamos o mecanismo de tratamento de eventos de MFC que relaciona um evento a um determinado método por meio do *Class Wizard* do *Visual Studio 6* para dispensar o uso de mecanismos computacionais mais simples como um `switch`, e fizemos apenas uma estrutura hierárquica que trata esses eventos de forma simples e descomplicada.

6.4 A nossa classe CDocument

A nossa classe `CBuildEditorDoc`, visualizada no Código 29, é feita para que o usuário tenha acesso ao ambiente virtual editado por meio de visualização e de navegação. Ela possui eventos de criação de documentos, eventos para salvar documentos e eventos para abrir documentos, respectivamente, `OnNewDocument`, `OnSaveDocument` e `OnOpenDocument`.

A classe `CBuildEditorDoc` classe é praticamente idêntica quando comparada com a classe `CCityEditorDoc`, pois os eventos de Salvar, Abrir e Criar um novo documento possuem o mesmo escopo de mapeamento de eventos bem como a mesma complexidade envolvida na escrita de arquivo por conta do padrão *interpreter* utilizado também no editor de cidades. Aqui, mais uma vez, nos deparamos com a simplicidade com que este padrão se adeqüa a MFC.

O nosso documento, em ambos os editores, tem um único atributo `m_currentModel` que é do tipo `CModel`, classe feita para conter um ambiente com luzes e prédios no editor de prédios e com prédios, luz, terreno e rodovias no editor de cidades. No Código 30 temos o esqueleto de `CModel` para um prédio só no editor de prédios e no Código 31 para o editor de cidades.

A partir destas tabelas vemos que a função do modelo é apenas possuir os objetos do nosso editor para a renderização. No caso do editor prédio possuímos apenas uma fonte de luz e um prédio, já no caso do editor de cidades possuímos todas as rodovias, com as instâncias dos prédios com as suas respectivas calçadas e a luz do ambiente chegando,

Código 29: A nossa classe de documento.

```
class CBuildEditorDoc : public Cdocument
{
protected: // create from serialization only
    CBuildEditorDoc();
    DECLARE_DYNCREATE(CBuildEditorDoc)

// Attributes
public:
    CModel *m_currentModel;
// Operations

public:
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CBuildEditorDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    virtual void DeleteContents();
    virtual BOOL OnSaveDocument(LPCTSTR lpszPathName);
    virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
    //}}AFX_VIRTUAL

// Implementation
public:
    bool OnExportVRMLDocument(const char *pcharFileName);
    virtual ~CBuildEditorDoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
};
```

Código 30: Modelo do prédio a ser rederizado.

```
class CModel
{
private:
    CBuilding * m_pBuilding;
    CLighting * m_pLighting;
public:
    void InitLights();
    CBuilding * GetBuilding();
    void SetBuilding(CBuilding *pB);
    void Render();
    CModel();
    virtual ~CModel();

};
```

Código 31: Modelo da cidade a ser rederizada.

```
class CModel { public:
    bool m_LandModified;
    string m_LandFile;
    static CRoadContainer m_RoadsInstance;
    static char * landType;
    static CBuildingsContainer m_BuildingsInstance;
    CLighting * m_pLighting;
    static void initTerrain();
    void InitLights();
    void Render();
    CModel();
    virtual ~CModel();

};
```

assim, ao nosso modelo final.

Logo uma visão OpenGL, no nosso caso a Viewport, precisa apenas renderizar o modelo `CModel` para que se tenha uma representação gráfica do mundo, desde que todo o modelo editado esteja com o seu contexto apontando para a viewport em questão. O contexto é um estado do sistema que determina qual a visão onde as operações estão fazendo efeito no momento.

6.5 Construindo eventos e interface gráfica

Aqui também, eventos de interface gráfica foram ligados a métodos por meio do *Class Wizard*. As classes que podem ser vistas no Apêndice G (página 141) formam a interface gráfica do editor de prédios final. Todas elas foram geradas a partir de um recurso do *Visual Studio* chamado *Resource Editor*. Esse editor permite vincular a descrição gráfica da interface, chamada por ele de *resource*, a um comportamento definido pela classe. Na Tabela 5 temos uma enumeração das classes do editor, descrevendo rapidamente a função de cada uma.

Para entendermos melhor como a estrutura de MFC se comunica com o documento e com as visões, vejamos no Código 32 o mecanismo de adicionar uma textura ao documento atualmente aberto.

Código 32: Adicionando textura ao documento

```
void CTextureTab::OnTextureAddButton()
{
    // TODO: Add your control notification handler code here
    if (UpdateData(TRUE)) {
        CTexture m(string((const char *)m_TextureFile), string((const char *)m_TextureName));
        m.Load();
        //Navegando pelas visões e gerando a textura em cada uma
        POSITION pos = doc->GetFirstViewPosition();
        doc->GetNextView(pos);
        while (pos != NULL)
        {
            COpenGLWnd* pView = (COpenGLWnd *)doc->GetNextView(pos);
            pView->SetContext();
            m.GenObject();

        }
        m.Unload();
        //adicionando texturas ao combobox e, indiretamente,
        //adicionando ao manager.
        m_TextureCombo.AddTexture(m);

        UpdateData(FALSE);

        doc->SetModifiedFlag(TRUE);
        doc->UpdateAllViews(NULL);
    }
}
```

Nesse código vemos claramente que para adicionar uma textura a cada visão OpenGL,

Tabela 5: Objetos definidos a partir de MFC para o editor de prédios e seu comportamento.

CBuildEditorApp	Responsável pela união do *documento* com a *visão*. Também trata dos eventos padrões de botões existentes na barra de ferramentas e em menus da interface principal.
CMainFrame	A janela onde a visão será colocada. Pode ser divida em outras sub-janelas, fornecer visões diferentes de um mesmo ambiente. No nosso caso para usar as viewports, a janela foi dividida em vários pedaços menores com a ajuda da classe `CSplitterWnd` da API MFC.
CFormView	A visão onde os botões e painéis do programa estão alocados. Recebe todos os eventos de `CDocument` e repassa-os quando necessário para os painéis.
CBEControlsSheet	Representa a componente onde estão colocados os painéis em forma de tabulação, e toma conta de eventos de redimensionamento dos painéis dentro do CFormView.
CBuildingTab	Painel que representa o prédio propriamente dito. Além de editar as dimensões do prédio, permite a junção de Texturas com Materiais para definir as faces do prédio.
CMaterialTab	Painel que fornece ferramentas ao usuário para criação, exclusão e modificação de um material para o modelo. Esse material depois de definido, pode ser mapeado em um prédio.
CTextureTab	Painel que fornece ferramentas ao usuário para importação de uma textura para o modelo. Esse material, depois de definido, pode ser mapeado em um prédio.
CMaterialComboBox	Estrutura que se integra com o *MaterialManager*, para enumerar os itens contidos nele em um *Combo Box*. Além disso, oferece operações de inserção e remoção que o integram com o *Manager*.
CTextureComboBox	Estrutura que se integra com o *TextureManager*, para enumerar os itens contidos nele em um *Combo Box*. Além disso, oferece operações de inserção e remoção que o integram com o *Manager*.

Tabela 6: Objetos definidos a partir de MFC para o editor de prédios e seu comportamento.

CCityEditorApp	Responsável pela união do *documento* com a *visão*. Assim como a aplicação do editor de prédios, trata dos eventos padrões de botões existentes na barra de ferramenta e em menus da interface principal.
CMainFrame	Idêntico ao que foi descrito na Tabela 5
CFormView	Idêntico ao que foi descrito na Tabela 5
CMainSheet	Representa a componente onde estão colocados os painéis em forma de tabulação. Toma conta de eventos de redimensionamento dos painéis dentro do CFormView.
CCityTab	Representa a componente onde estão colocados os painéis em forma de tabulação. Toma conta de eventos de redimensionamento dos painéis dentro do CFormView.

ou Viewport, precisamos navegar por cada uma delas gerando a textura, para depois podermos adicioná-la ao combo box e consecutivamente ao *TextureManager*.

Para que o método `OnTextureAddButton` seja chamado pelo evento do botão, temos que adicioná-lo aos `Messages Maps`, estruturas completamente fora do contexto de orientação a objeto proposto inicialmente pela API MFC. Essa tarefa precisa ser realizada pelo *Class Wizard*, pois fazer isto por código fonte gera erros pouco intuitivos e difíceis de serem concertados. No Código 33 temos um exemplo de *Message Map* que liga o identificador do botão a um método.

Código 33: Mapeando mensagens em MFC

```
BEGIN_MESSAGE_MAP(CTextureTab, CPropertyPage)
    //{{AFX_MSG_MAP(CTextureTab)
    ON_BN_CLICKED(IDC_TEXTURE_ADD_BUTTON, OnTextureAddButton)
    ON_BN_CLICKED(IDC_TEXTURE_REMOVE_BUTTON, OnTextureRemoveButton)
    ON_CBN_SELCHANGE(IDC_TEXTURES_COMBO, OnSelchangeTexturesCombo)
    ON_BN_CLICKED(IDC_TEXTURE_BROWSE, OnTextureBrowse)
    ON_WM_SHOWWINDOW()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Para observarmos como um determinado combo box está integrado ao Manager, vejamos no Código 34 o mecanismo que permite adicionar uma textura a um combo-box de textura.

Código 34: Adicionando uma textura ao modelo

```
void CTextureComboBox::AddTexture(CTexture m)
{
    OBJECT_ID id;
    if(m.GetName()!="NONE") {
        if(!g_TextureManager.ExistsBEObject(m.GetName())) {
            m.SetObjectID(IDFACTORY.GetID());
        }
        else {
            g_TextureManager.GetBEObjectId(m.GetName(), &id);
            m.SetObjectID(id);
            g_TextureManager.DeleteBEObject(id);
        }

        g_TextureManager.AddBEObject(m);
        // atualiza o conteudo da combo box baseado na informação do
        // manager que foi modificado.
        UpdateContent();
        this->SetWindowText(_T((m.GetName()).c_str()));
    }
}
END_MESSAGE_MAP()
```

No Código 34, primeiramente é verificada a existência prévia da textura. Se essa existência for confirmada, a identificação, ou ID, do objeto é dada à nova textura a ser adicionada, e a textura antiga é excluída fazendo com que qualquer referência a velha textura conste como sendo da nova.

Assim, só mostramos parte da complexidade envolvida no tratamento dos eventos

MFC dentro deste projeto. No entanto, pode se notar que a estruturação do código final tenta, mas não atinge, o objetivo de oferecer e garantir todos os mecanismos previstos pelo paradigma de orientação a objeto. Um exemplo bem claro disso é que a troca de mensangem é identificada por Messages maps, mecanismo puramente imperativo.

6.6 Flyweight e suas diversas aplicações

Agora que já analisamos os pacotes `BEiostream`, `CEiostream` e suas estruturas de importação e exportação de arquivo, vejamos agora uma estrutura que nos ajuda tanto no tratamento de eventos MFC, quanto na importação de ambientes de arquivo. Em um primeiro momento, poderíamos pensar que tendo o *Building Editor*e o *City Editor* fixado uma série de padrões, que facilitam a nossa programação da estrutura de leitura e escrita em arquivos, poderíamos simplesmente utilizar o padrão *Interpreter* juntamente com a estrutura *Importer* para definir o grafo de parser. Entretanto, foi visto que novas situações foram aparecendo e novos padrões de projeto foram sendo aplicados pouco a pouco.

O mais relevante desses padrões foi o padrão *Flyweight*, que estabelece um 'pool' de objetos comuns à aplicação. Estes objetos, apesar de redundantes à aplicação, não devem apenas ser referenciados, pois eles possuem o que chamamos de *estado*. O estado é normalmente algo que influenciará a visualização ou comportamento do objeto, mas que é altamente volátil ou mutável, tanto no contexto de uma única instância como no contexto das varias instâncias de um mesmo objeto.

Para entendermos melhor como funciona este padrão, vejamos o exemplo de um editor de texto. Uma única letra é referenciada de diversas formas e estilos como, por exemplo, negrito, itálico, estilo de título, estilo de sub-títulos etc. Enumerar todas as combinações possíveis, cada qual com seu objeto específico, seria impraticável, pois a complexidade de acesso envolvida no manuseamento de uma quantidade tão grande de tais objetos tornaria o programa excessivamente complexo. Adicionalmente, um programa desse tipo exigiria uma quantidade muito grande de memória para armazenar tais objetos.

O padrão *Flyweight* tem como objetivo reduzir esta complexidade e o custo de memória de uma maneira tal que a eficiência seja minimamente comprometida. Isto é feito através do uso de um *pool* de objetos e da "modificação" das instâncias por meio de seus estados.

Por isto, o padrão *Flyweight* tem uma aplicação tão clara em editores de cenários. Os objetos do *Flyweight* são comuns ao editor, como no nosso caso são certos prédios e rodovias, e suas instâncias podem ser diferenciadas com relação ao seu estado. Isto pode

vir a tornar o acesso aos objetos mais lento por conta da exigência de acesso ao pool a cada operação.

Além da necessidade deste padrão, o mesmo também se encaixa com a *técnica de estados*, que foi descrita na seção 4.3.4. Ou seja, não só o motor deve possuir dispositivos para permitir modificações de estado, como estes estados são essenciais para aplicação de um padrão como *Flyweight*.

Vejamos agora como este padrão se encaixa nos eventos de importação e exportação do modelo para o ambiente. Primeiramente, vejamos a importação. O nosso arquivo de descrição do ambiente representa as instâncias dos prédios presentes no ambiente conforme descrito no Código 35.

Código 35: Instância de prédios em um arquivo do City Editor.

```
BUILDING_INSTANCE: File_Name: "C:\\Mozart
Filho\\msc\\Editor\\BuildEditor\\b.bld" Has_Sidewalk: 1 3
Position: 3DVector 140 60.5 240 Rotation: 45

BUILDING_INSTANCE: File_Name: "C:\\Mozart
Filho\\msc\\Editor\\BuildEditor\\b.bld" Has_Sidewalk: 1 3
Position: 3DVector 140 60.5 230 Rotation: 45
```

No arquivo vemos claramente que a instância do prédio é descrita totalmente pelo arquivo para onde ela aponta, além de que suas propriedades como posição, orientação e a presença de calçadas. Uma situação que pode ocorrer facilmente é, no ato da leitura, termos que saber se um determinado prédio foi previamente importado para, então, termos que utilizar sua instância previamente existente. Isto é completamente transparente com o uso de *Flyweight*.

Para melhor entendermos, vejamos como está organizado o adicionamento de prédios ao modelo. Para isto vejamos primeiramente a classe `CFlyweightFactory`, presente no Código 36. Esta classe é um template para toda e qualquer classe que queira implementar o padrão *Flyweight*, seu único requisito é que o programador defina um manager (descrito na seção 4.3.3, página 53) para ser um *pool* de objetos dessa *Fábrica de Flyweight* e também um método para importar um *Flyweight* de arquivo caso ele não esteja no respectivo manager. Vemos também que esta Factory também oferece métodos genéricos de acesso aos seus objetos por meio de um `OBJECT_ID`, nome ou caminho para arquivo. Sendo que, no último caso, a importação retorna uma mensagem de erro, caso haja algum problema na leitura de arquivo; caso ocorra tudo bem com a importação, as texturas e materiais, que serão utilizados pelo novo prédio e que não estão disponíveis no manager, tornam-se disponíveis para serem tratados pelo recebedor da mensagem.

Para vermos como definimos o fábrica de *Flyweights* do prédio, vamos observar os Códigos 37, 38 e 39. No Código 37 fica claro que a criação, no caso de importarmos um prédio de um arquivo, só é necessária se o prédio não existir a priori. Caso o prédio exista,

Código 36: Fábrica de Flyweights do Editor de Cidades.

```
template <class T> class CFlyweightFactory { public:
    CFlyweightFactory();
    virtual ~CFlyweightFactory();

    bool GetFlyweightFromFile(string FileName, T ** ppObject, CImportFileMessage *message=NULL);
    bool GetFlyweight(OBJECT_ID nID, T ** ppObject);
    bool GetFlyweightFromName(string Name, T ** ppObject);
protected:
    CBEObjectManager<T> * m_pFlyweightManager;
    virtual bool CreateInstance(string FileName, T ** ppObject,
            CImportFileMessage *message=NULL)=0;

};
```

este é utilizado como uma única instância dentro do nosso editor. Os Códigos 38 e 39 nos mostram como uma *fábrica de flyweights* específica é criada a partir da estrutura genérica de templates citada anteriormente.

Código 37: Fábrica de Flyweights criando instância a partir de um arquivo.

```
template <class T> bool
CFlyweightFactory<T>::GetFlyweightFromFile(string FileName,
                    T ** ppObject, CImportFileMessage *message) {
    if(!m_pFlyweightManager->ExistsBEObjectFile(FileName)) {
        return CreateInstance(FileName,ppObject,message);
    } else {
        m_pFlyweightManager->FindBEFileObject(FileName,ppObject);
    }
    return false;
}
```

Código 38: Fábrica de Flyweights do tipo prédio.

```
class CBuildingFlyweightFactory : public
CFlyweightFactory<CBuilding> { public:
    CBuildingFlyweightFactory();
    virtual ~CBuildingFlyweightFactory();
protected:
    virtual bool CreateInstance(string FileName, CBuilding ** ppObject,
                                CImportFileMessage *message=NULL);

};
```

Com o padrão *Flyweight*, torna-se muito fácil a importação de prédios já que basta pedirmos à *fábrica* um determinado objeto que ela se encarregará de criá-lo ou de buscá-lo para que possamos utilizá-lo posteriormente, sem que o programador tome conhecimento do processo.

Para termos uma idéia de como este padrão funciona no mecanismo de importação de um prédio, vejamos o código de importação do prédio dentro do *City Editor*. Este código está acoplado à toda e qualquer *Combo Box* para que itens de interface gráfica possam ter acesso à busca, remoção e inserção de prédios à partir de um atributo local. Para isto, fizemos uma classe *CBuildingComboBox* ter acesso à *Fábrica de Flyweight* e, assim, temos o método de adicionamento encontrado no Código 40.

Código 39: Método `CreateInstance` da classe `CBuildingFlyweightFactory`.

```
bool CBuildingFlyweightFactory::CreateInstance(string FileName,
        CBuilding ** ppObject, CImportFileMessage *message) {
    CImporter importer;
    BE_Building *be;
    CBuildingImportFileMessage *bmessage;
    bool returnValue;
    importer.OpenScene((const char *) FileName.c_str());
    importer.GetBuilding(&be);
    CBuilding building(importer, *be);
    if(m_pFlyweightManager->ExistsBEObject(building.GetName())) {
        m_pFlyweightManager->FindBEObject(building.GetName(),ppObject);
        returnValue=false;
    } else if(m_pFlyweightManager->ExistsBEObjectFile(FileName)) {
        m_pFlyweightManager->FindBEFileObject(FileName,ppObject);
        returnValue=false;
    } else {
        building.SetObjectID(IDFACTORY.GetID());
        m_pFlyweightManager->AddBEObject(building);
        m_pFlyweightManager->FindBEFileObject(FileName,ppObject);
        returnValue=true;
        if(message) {
            bmessage=(CBuildingImportFileMessage *)message;
            for(int i=0;i<importer.imported_materials.size();i++) {
                if(importer.imported_materials[i]) {
                    bmessage->materials.push_back(be->Materials[i]);
                }
                if(importer.imported_textures[i]) {
                    bmessage->textures.push_back(be->Textures[i]);
                }
            }
        }

    }
    return returnValue;
}
```

Código 40: ComboBox recorrendo ao Flyweight para importar/buscar um prédio.

```
void CBuildingComboBox::AddBuilding(const char *fileName,
                                    CBuilding **b, CImportFileMessage *message) {
    CBuilding *p_building;
    if(g_BuildingManager.ExistsBEObjectFile(string(fileName))) {
        MessageBox("The refered building already exists. The desired building was not imported."
                ,"Warning!",MB_ICONINFORMATION);
        *b=NULL;
    }
    else {
        if(!g_BuildingFlyweightFactory.GetFlyweightFromFile(string(fileName),
                                                        &p_building,message)) {
            MessageBox("Impossible to import.\n There would be two buildings with the same name.
                        This version of the aplication doesn't support diferent files with the
                        same name.","Warning!",MB_ICONINFORMATION);
        }
        SelectBuilding(p_building->GetObjectID());
        (*b)=p_building;
    }
}
```

Após importado o prédio, os materiais e texturas que o compõem, mas que não estão definidos nos managers, são retornados por meio de uma mensagem para que façamos o seu carregamento nas respectivas *Viewports*. Este processo é mostrado no Código 41.

Código 41: Importando um prédio para a cidade.

```
void CBuildingTab::OnButtonAdd() {
    // TODO: Add your control notification handler code here
    CBuildingImportFileMessage *bmens;
    POSITION pos;
    int i;
    if (UpdateData(TRUE)) {
        CBuilding *b;
        bmens = new CBuildingImportFileMessage();
        m_BuildingCombo.AddBuilding((const char *) _T(m_BuildingFile),&b, bmens);


        for(i=0;i<bmens->textures.size();i++) {
            CTexture *t;
            g_TextureManager.FindBEObject(bmens->textures[i],&t);
            t->Load();
            pos = doc->GetFirstViewPosition();
            doc->GetNextView(pos);
            while (pos != NULL) {
                COpenGLWnd* pView = (COpenGLWnd *)doc->GetNextView(pos);
                pView->SetContext();
                t->GenObject();
            }
            t->Unload();
        }
        for(i=0;i<bmens->materials.size();i++) {
            CMaterial *m;
            g_MaterialManager.FindBEObject(bmens->materials[i],&m);
            m->Load();
            pos = doc->GetFirstViewPosition();
            doc->GetNextView(pos);
            while (pos != NULL) {
                COpenGLWnd* pView = (COpenGLWnd *)doc->GetNextView(pos);
                pView->SetContext();
                m->GenObject();
            }
            m->Unload();
        }
        pos = doc->GetFirstViewPosition();
        doc->GetNextView(pos);
        while (pos != NULL) {
            COpenGLWnd* pView = (COpenGLWnd *)doc->GetNextView(pos);
            pView->SetContext();
            b->GenObject();
        }
        delete bmens;
        m_BuildingCombo.UpdateContent();
    }
    UpdateData(FALSE);
}
```

Assim chegamos ao fim de nossa explanação sobre o padrão Flyweight e suas aplicações dentro de um editor de cenários. Para maiores detalhes sobre como usar este padrão o leitor pode recorrer ao Apêndice L na página 149 deste documento.

7 *Guia do Usuário.*

Agora que já foram analisados os aspectos de programação dos nossos editores, vejamos, através do manual do usuário, os mecanismos adotados para interface com o usuário. O detalhamento das componentes do editor serão debatidas juntamente como intuitividade da ferramenta e mecanismos adotados para proporcionar tal iteratividade.

Seria prudente também informar que este editor tem apenas o cunho educacional citado anteriormente nesta dissertação e que informações sobre eventuais bugs devem ser reportadas à *Mozart de Siqueira Campos Araújo Filho* através do email *msca@cin.ufpe.br*. Além disso, uma estrutura de suporte e manutenção destas aplicações não está montada, dado que estas duas aplicações foram desenvolvidas apenas pelo autor deste documento.

7.1 Instalação da aplicação

Para baixar os componentes de instalação de qualquer um dos editores, por favor, recorra a home page citada na referência Araújo Filho (2003).

Por enquanto, as ferramentas não dispõem de programas de instalação automática. Portanto, para instalá-las, copie os arquivos executáveis `‘‘.exe’’` e os de biblioteca estática `‘‘.lib’’` para um diretório a parte.

E, por último, copie o arquivo ijl15.dll para o diretório de sistema do seu Windows, no caso %System Root%\system.

Finalmente, agora você deveria ter o editor funcional rodando na sua máquina. Se por acaso ainda não tem esta versão do editor funcionando, procure se informar se o driver instalado para sua placa de vídeo tem suporte à OpenGL.

7.2 O *Building Editor*

Tentaremos, agora, dar ao usuário as principais instruções de uso de uma das ferramentas desenvolvida durante o progresso do trabalho o Building Editor, o editor de

construções. Conforme citado anteriormente, esta ferramenta se caracteriza mais como um editor de objetos, os prédios, do que como um editor de cenários propriamente dito. Entretanto, uma breve análise dos mecanismos oferecidos por este editor nos mostram que, apesar de sua simplicidade, ele oferece muito dos mecanismos usados no segundo editor.

As instruções dadas a seguir estão agrupadas de acordo com os componentes funcionais que compõem o editor. Todos os comentários acerca do editor estarão devidamente ilustrados sempre que a complexidade da explanação o exigir.

7.2.1 Barra de Ferramentas e Menus

Iniciaremos análise do *Building Editor* visualizando a Figura 32, na qual temos sua tela inicial sem nenhum prédio ainda desenvolvido. Esta tela tem quatro viewports navegáveis, uma barra lateral de ferramentas para edição do modelo, além de menus e barra de ferramenta presentes em uma aplicação Windows padrão.

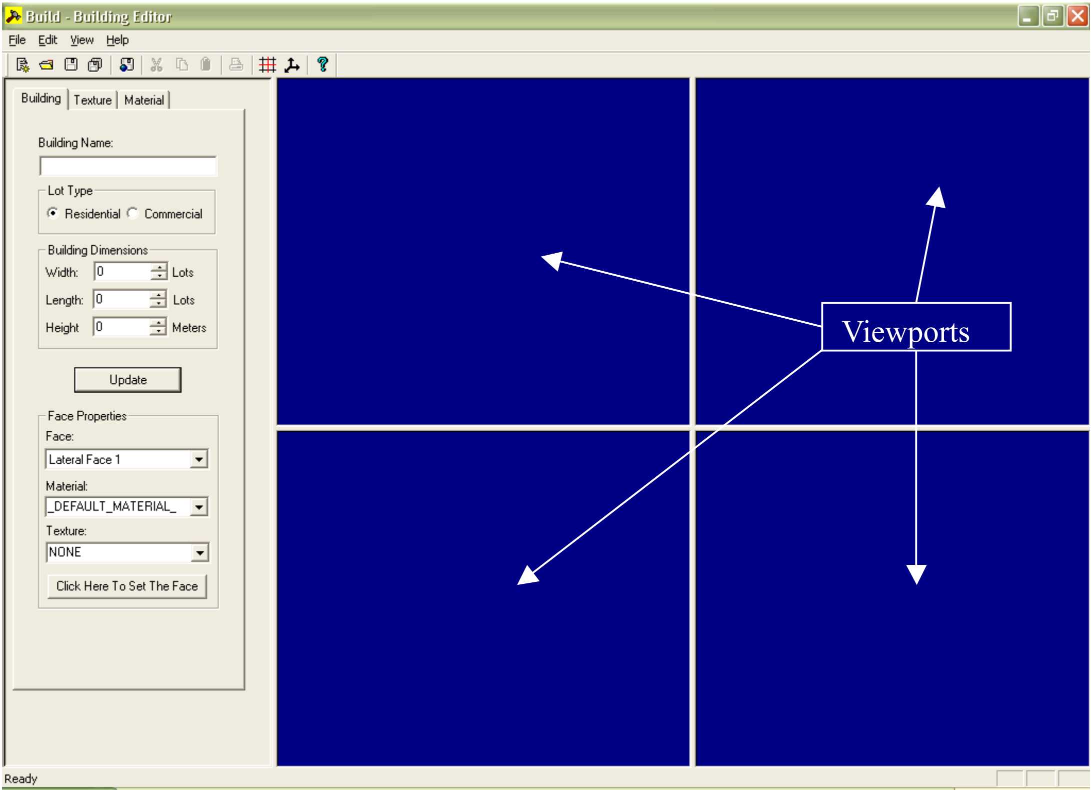


Figura 32: Building Editor, tela inicial.

Primeiramente, analisaremos os comandos conhecidos dentro do editor que estão contidos na barra de ferramentas da Figura 33.

Os ítens da barra de ferramentas e menus possuem, citadas em ordem correspondente aos botões da barra de ferramenta da Figura 33, as funcionalidades definidas na Tabela 7.

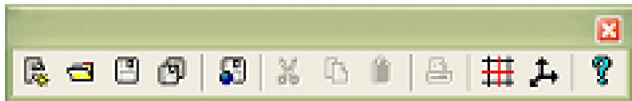


Figura 33: Barra de ferramentas do Building Editor.

Tabela 7: Objetos definidos a partir de MFC e seu comportamento.

	New: cria um novo ambiente sem nenhuma textura, material ou prédio incorporados a ele. A aparência de um novo documento é exatamente igual a aparência da Figura 32.
	Open: abre um arquivo de extensão ".bld" com um prédio salvado anteriormente.
	Save: salva o prédio atualmente em edição se este já tiver sido salvo anteriormente. Caso contrário, abre uma tela de salvar como.
	Save As: Permite que se salve uma cópia do modelo em edição. Para isso ele abre uma janela com navegador de diretórios padrão, que permite que escolhamos onde salvar o arquivo especificado.
	Cut, Copy e Paste: botões com funções padrão para se manipular as informações da área de transferência do Windows.
	Export to VRML: Botão para exportação de arquivo VRML. O arquivo VRML tem suas texturas exportadas como caminho relativo, para facilitar a compatibilidade com aplicações web. O comando de exportar VRML pode ser acessado através do menu como mostra a Figura 35 e pelo botão em alto relevo da Figura 36.
	Grid: permite a exibição de uma grade em todas as viewports. Esta grade ajuda a perceber as informações de dimensão e de posicionamento do prédio no espaço.
	Axis: Eixos Cartesianos X, Y e Z. Onde Z, inicialmente, está apontando para fora do monitor, X aponta para direita e Y aponta para cima. Na Figura 37 podemos ver a grade e o eixo em funcionamento, bem como, os penúltimo e antepenúltimo botões de Eixo e Grade habilitados. O acesso a grade e eixo pelo menu está disponível na Figura 34.

Figura 34: Menu view do Editor com os componentes grade e eixo.

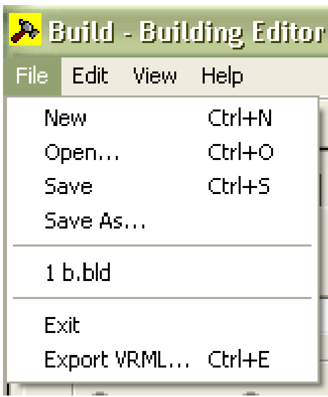


Figura 35: Comando File do editor.

Figura 36: Exportando para VRML.

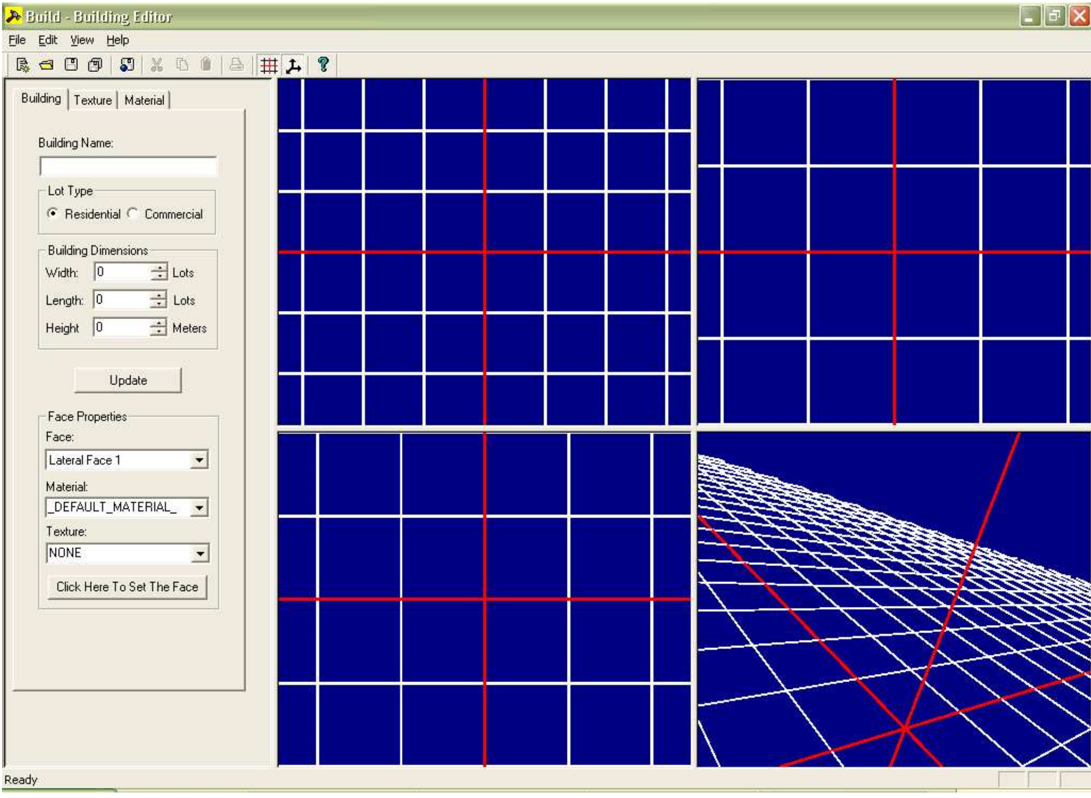


Figura 37: Editor com a grade e o eixo habilitados.

7.2.2 Formulários de criação de construções

Após a explanação das funcionalidades mais genéricas do editor, mostraremos o processo de criação de uma construção. Para criação de uma construção, as funcionalidades do *Building Editor* foram agrupadas em três painéis. Onde cada painel dentro da área lateral possui uma função específica que será discutida nas seções 7.2.2.1, 7.2.2.2 e 7.2.2.3.

7.2.2.1 Painel de materiais

Este painel, mostrado na Figura 38, está relacionado à criação de materiais que serão usados para definir os prédios. Estes materiais têm um nome e três cores distintas.

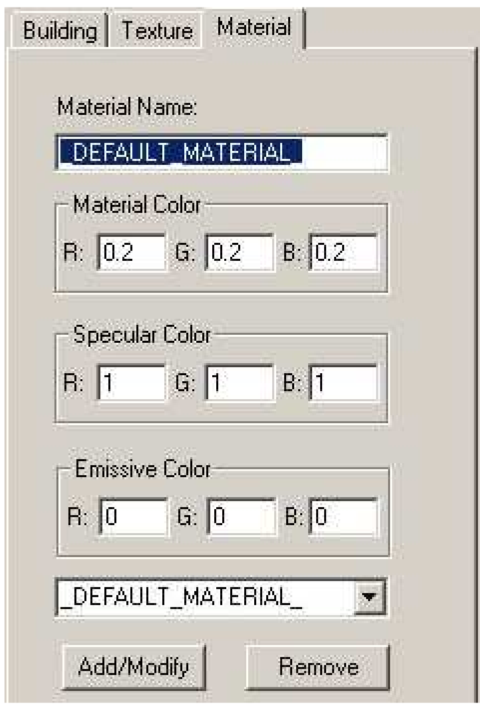


Figura 38: Painel de materiais.

O nome é uma seqüência qualquer de caracteres. Todavia, para que um ambiente VRML seja exportado corretamente, espaços e caracteres especiais como aspas e barra invertida não devem ser colocados, pois estes casos não são tratados pelo editor.

Primeiramente, temos a cor do material propriamente dito, definida com o nome de *Material Color* na figura ao lado.

A Cor Especular, definida na figura ao lado como *Specular Color*, define a cor da reflexão metálica que um material possui.

Por último, temos a propriedade *Emissive Color* - cor que irradia uniformemente do material propriamente dito, mas sem a necessidade de iluminação.

Após definidas as propriedades do material, ele pode ser adicionado ao ambiente por meio do botão *Add/Modify*. Este botão adiciona um material ao modelo se ele não havia sido definido antes, e modifica um material existente para um material atual se já houver um material com o mesmo nome.

O botão remover, identificado na figura pelo nome *Remove*, remove o material com o nome definido em *Material Name*, se ele existir. Caso não exista, nenhuma ação é tomada após o aperto do botão.

7.2.2.2 Painel de Textura

Este painel, mostrado na Figura 39, permite que uma textura seja incorporada ao modelo. Para incorporar uma textura, basta clicar no botão *Browse...* e localizar o arquivo desejado. Após feita a operação, é sugerido um nome para textura, que seria igual ao nome do arquivo. No caso da preferência por outro nome, modifique o campo *Texture Name* conforme desejado.

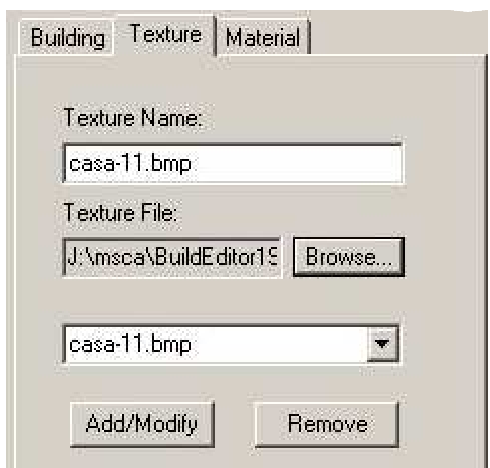


Figura 39: Painel de texturas.

Por fim, adicione ou modifique a textura usando o botão *Add/Modify*. Com o uso deste botão, caso uma textura que possua um mesmo nome já esteja definida, será feita a substituição dela pela textura atual.

Por último, temos o botão *Remove* que remove uma textura definida dentro de *Texture Name*.

7.2.2.3 Painel do Prédio Editado

Este é o último painel do nosso editor de prédios. Os componentes deste painel estão enumerados a seguir, e o painel é mostrado na Figura 40.

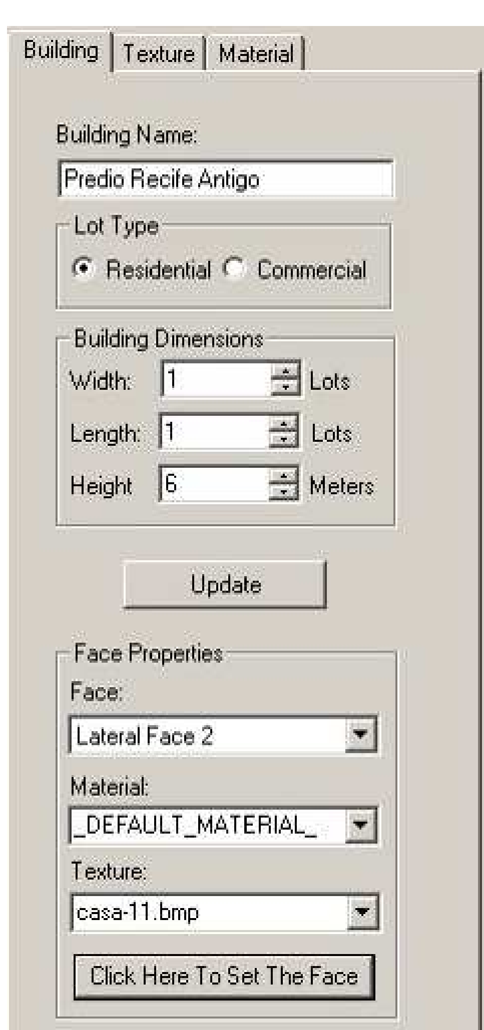


Figura 40: Painel do prédio editado.

Building Name: O nome do prédio em questão.

Lot Type: Define o tipo de lote que o prédio ocupa. Os tipos de lote são Residenciais e Comerciais, definidos como *Residential* e *Commercial* na Figura mostrada lateralmente. Os lotes comerciais possuem uma área quatro vezes maior que os residenciais, ou seja, um lote comercial é equivalente a 2X2 lotes comerciais. Atualmente, o lote residencial tem uma dimensão base de um por um metros. Esta abordagem apesar de não parecer muito intuitiva, dado que a maioria da pessoas consideraria mais simples uma abordagem direta em metros, torna possível uma discriminação de áreas residenciais e comerciais em um futuro próximo, no sentido do preço de lotes, cálculo do preço do imóvel, entre outros. Além disso, a maioria das ferramentas de autoria e linguagens descritivas como VRML não possuem unidades em suas dimensões mas sim medidas numéricas quaisquer.

Building Dimensions: Define as dimensões de um prédio. A largura e o comprimento são dados em função do lote base que foi escolhido no item *Lot Type*. A altura é dada em metros. Após as dimensões terem sido definidas aperte o botão *Update* e você verá o escopo do prédio a ser definido.

Face Properties: Neste caso, pode-se definir a textura e material de cada face do prédio, definidas pelo campo *texture* e *material*, respectivamente. Para modificar uma determinada face, basta primeiramente escolher a face, através da opção *Face*, e depois escolher o material e a textura entre os que fazem parte do modelo. Na opção face, as faces laterais (*Lateral Face*) são numeradas de 1 a 4, onde 1 significa a face frontal, 2 significa a face lateral direita, 3 a face traseira e 4 a face lateral esquerda, para um observador que esteja olhando na direção do eixo Z negativo.

Agora que já vimos as opções de edição fornecidas pelo editor temos disponível na Figura 41 uma amostra do resultado final do editor com um prédio editado nele.

7.2.3 Navegação nas Viewports

O último componente que merece atenção para se usar este editor é o mecanismo utilizado para navegar nas viewports.

Antes de entender o mecanismo de navegação, vejamos melhor as viewports contidas nele. Essas viewports estão organizadas de tal modo que há três visões ortogonais que são a visão lateral, visão frontal e visão superior. Além das visões ortogonais há uma visão em perspectiva, que permite rotações no ato da navegação. A Figura 42 mostra o que cada viewport do nosso editor significa.

O dispositivo de navegação funciona da seguinte maneira:

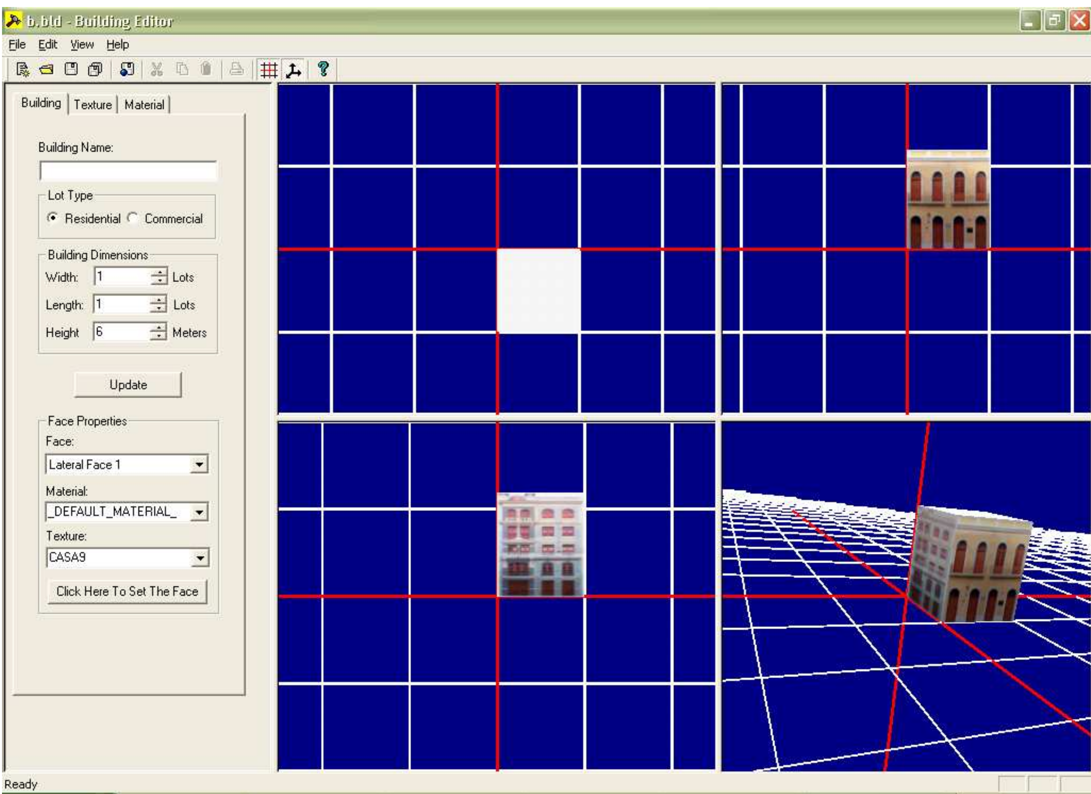


Figura 41: Editor com um prédio gerado.

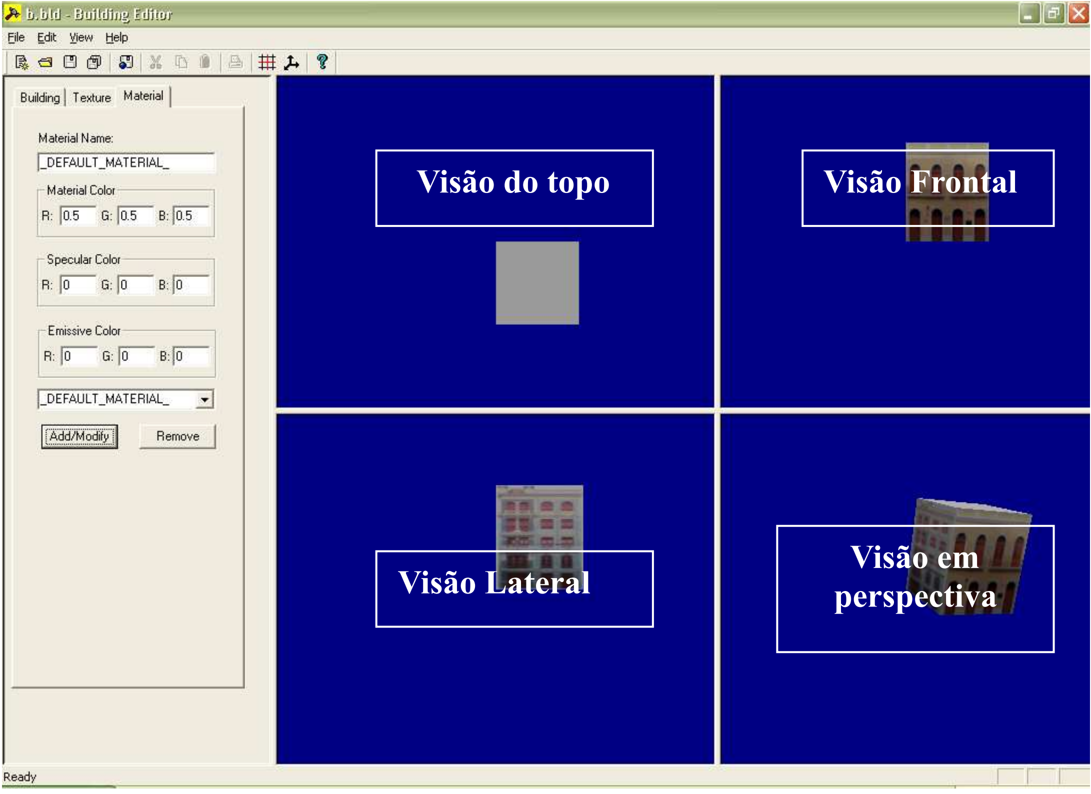


Figura 42: Visões do Building Editor.

- Ctrl + Botão esquerdo do mouse: permite que, conforme você mexa o mouse dentro da respectiva viewport, o objeto sofra rotações.
- Ctrl + Botão direito do mouse: permite que se ande paralelamente ao mecanismo de visão, movendo-se para esquerda, para direita, para cima e para baixo, conforme a movimentação do mouse na mesma direção dentro do viewport.
- Ctrl + botão central do mouse: aumentar o zoom ou diminuir o zoom conforme o cursor se move para cima ou para baixo.

A opção de rotação não está disponível nas viewports ortogonais, pois o ângulo destas visões é fixo. Em compensação, na visão em perspectiva a liberdade é total.

7.3 O City Editor

Dado que avaliamos o *Building Editor*, podemos começar a discutir o segundo editor, o City Editor, o qual é a aplicação principal deste trabalho, e para a qual o *Building Editor* foi criado desde o princípio. O *City Editor* é uma aplicação com o propósito de construir ambientes simulados de cidades. Onde a cidade é na verdade um ambiente que jaz sobre um terreno a ser importado pelo usuário.

As instruções de uso do editor dadas a seguir estão agrupadas em seções que englobam as várias opções disponíveis no nosso editor.

7.3.1 Funções do *City Editor*

Antes de entrarmos em detalhes sobre as funções disponíveis no *City Editor* falaremos um pouco sobre qual a idéia envolvida na utilização deste software específico.

O *City Editor*, cuja tela inicial é mostrada na Figura 43, permite-nos importar qualquer prédio definido pelo *Building Editor*. Estes prédios são, no caso do *City Editor*, objetos básicos que devem ser posicionados no ambiente. Assim como temos os prédios, temos também rodovias, objetos básicos que podem ser colocados em qualquer localização do terreno. Estes são os únicos objetos suportados pelo editor; entretanto, mais objetos básicos, como postes e árvores, podem ser adicionados às versões posteriores do editor, dado o grau de extensibilidade com que o motor e o editor foram feitos.

Visando um melhor entendimento do editor por pate do leitor, as funções do *City Editor* foram agrupadas em termos de funcionalidade, e são enumeradas nas seções seguintes.

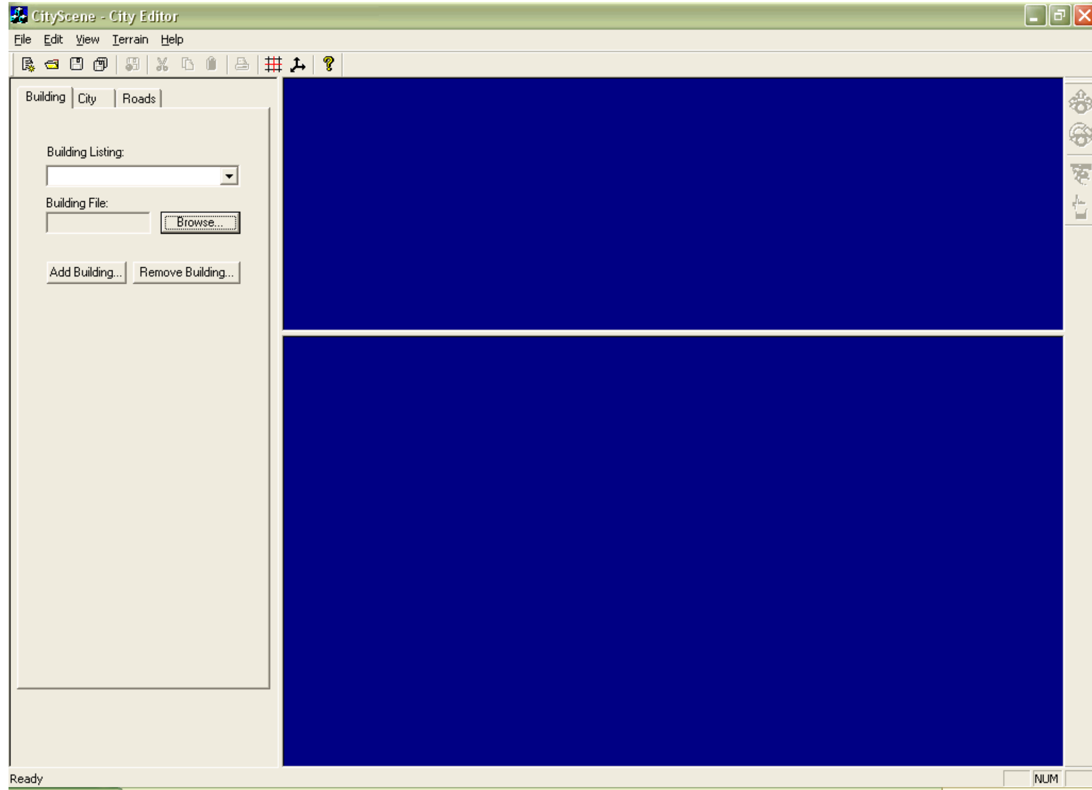


Figura 43: Tela inicial do City Editor.

7.3.1.1 Importando o terreno.

Conforme mostrado na Figura 43 o *CityEditor* possui inicialmente uma tela vazia. Tela esta que deve ser preenchida inicialmente com um terreno, assim como deveria ser intuitivamente constatado pelo usuário.

Para importar o terreno, devemos utilizar o comando "*Terrain – Import Terrain ...*" do editor, mostrado na Figura 44. Esta opção abrirá uma janela de abertura de arquivo mostrada na Figura 45. Nesta janela, temos disponíveis a opção de importação de arquivos brutos (`.RAW`), os quais são nada mais do que imagens disponíveis em escala de cinza sem cabeçalho. Os valores de pixel disponíveis pela imagem variam em escala de cinza de preto absoluto até branco, sendo o preto considerado o nível zero de altura e o branco o nível máximo de elevação do terreno. As dimensões da imagem tem que ser dadas na janela que sucederá a escolha do arquivo, mostrada na Figura 46.

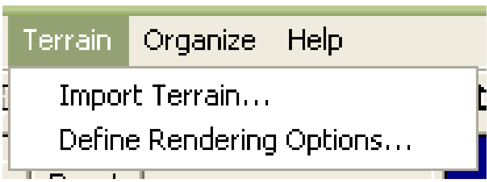


Figura 44: Opções do terreno para a aplicação.

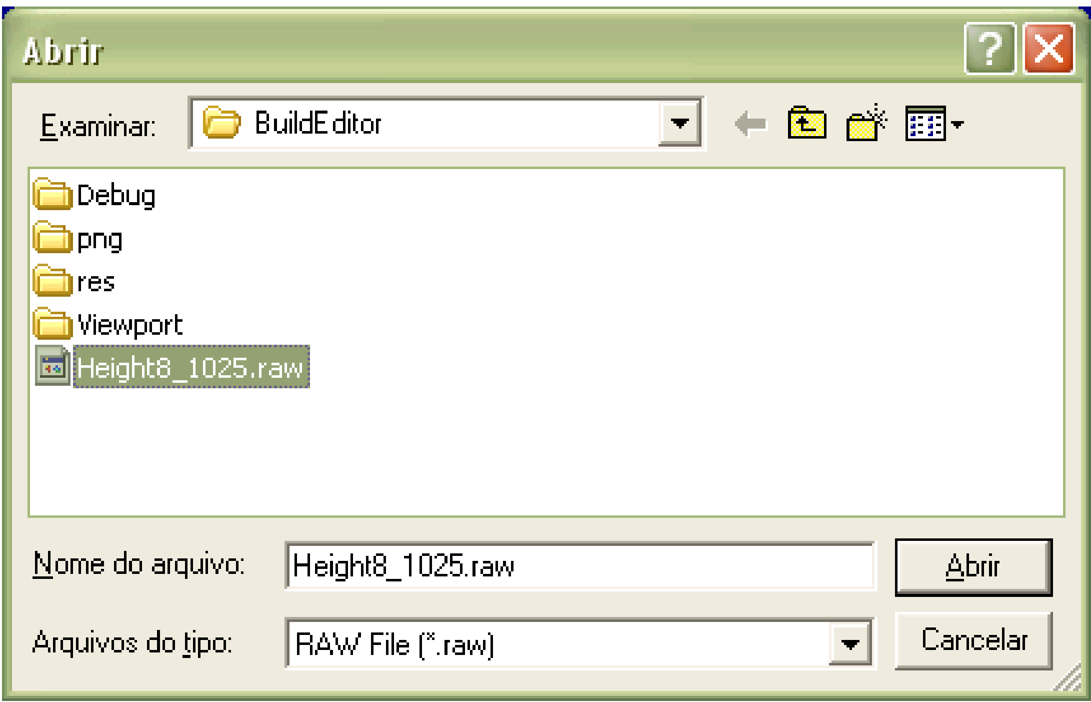


Figura 45: Importação do terreno para a aplicação.

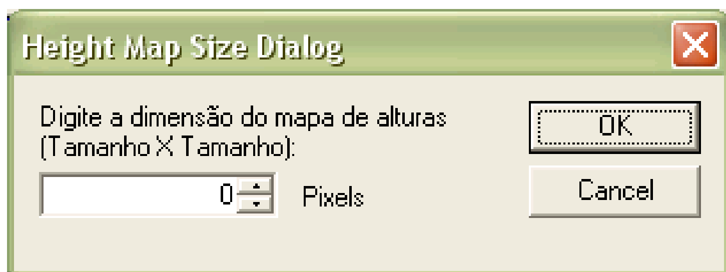


Figura 46: Definição das dimensões do mapa de alturas.

Após importado o terreno, o editor terá disponível sua visualização conforme mostrado na Figura 47.

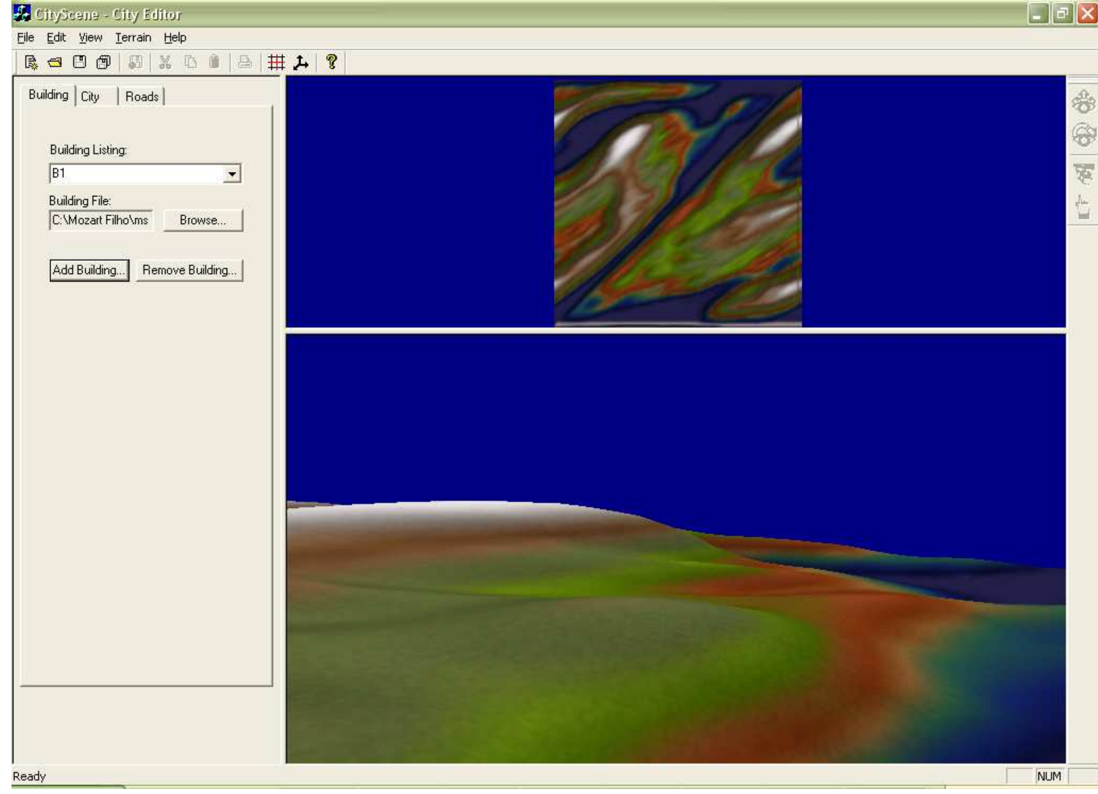


Figura 47: Visualização do ambiente com o terreno.

7.3.1.2 Definindo opções de renderização do terreno.

Outras opções oferecidas para o terreno já importado no nosso editor estão disponíveis no menu "*Terrain – Define Rendering Options...*". Entre essas opções, temos a escolha do algoritmo de renderização juntamente com a escolha coloração assumida pelo terreno de acordo com a altura, todas visíveis por meio da Figura 48.

As opções de escolha de algoritmos são triviais, consistindo apenas de três *radio buttons* disponíveis no grupo "*Available Algorithms*", cada qual correspondente à um dos algoritmos definidos na seção 4.4.

Além disso, está disponível para o usuário a possibilidade de editar as cores do terreno, discretizadas em 16 valores possíveis, todos dependentes da altura envolvida. Essa opção de edição das cores do terreno está disponível na opção "*Height Colors*" da Figura 48. As cores podem ser editadas tomando como base o menu de cores mostrado na Figura 49, ou recorrendo à opção custom deste menu e selecionando a cor através da janela mostrada na Figura 50.

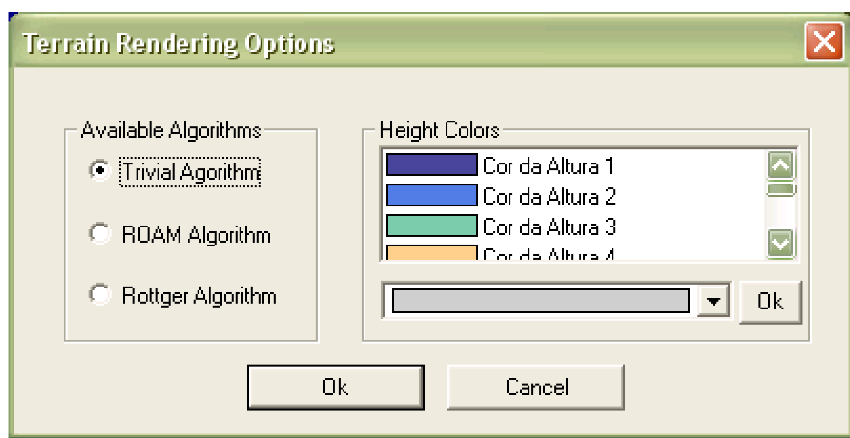


Figura 48: Tela de opções do terreno.

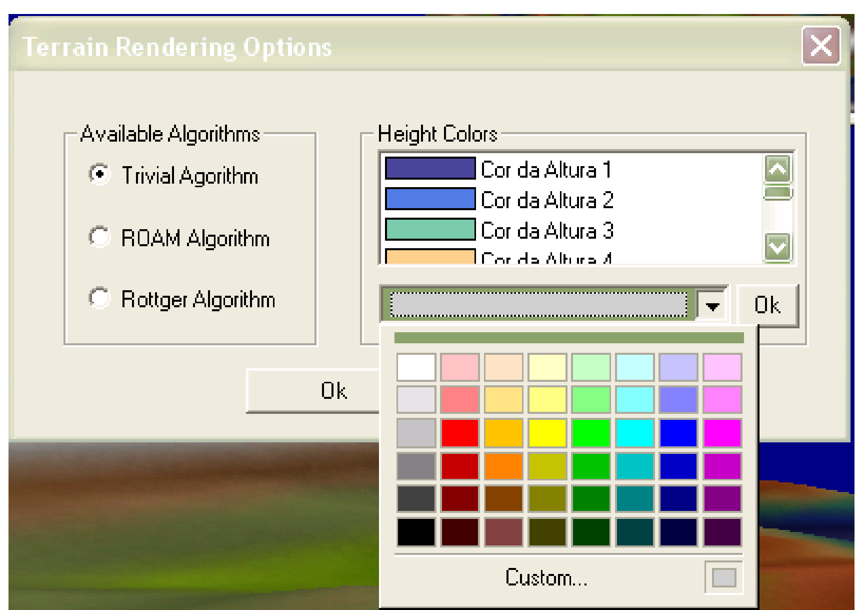


Figura 49: Menu de cores do terreno.

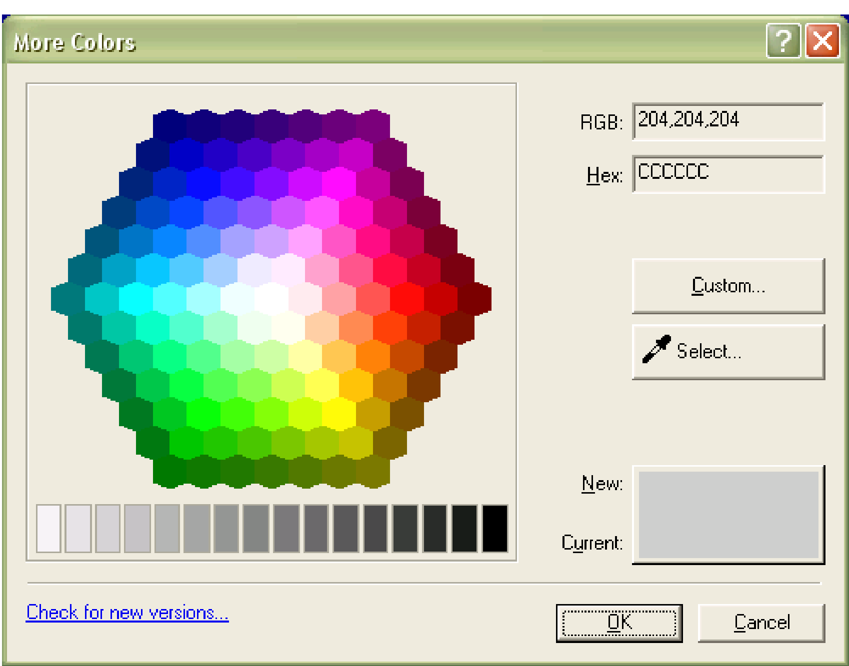


Figura 50: Menu detalhado de cores do terreno.

7.3.1.3 Navegação/Operação nas Viewports

Outro ponto relevante na nossa aplicação é a maneira como é feita a navegação de viewport. Neste editor, ao contrário do anterior, a navegação é feita por meio de botões e não mais de teclas especiais.

Os botões necessários à navegação estão, no caso do nosso editor, sempre localizados à direita da janela principal e têm o aspecto mostrado na Figura 51. Ou seja, define os quatro botões enumerados na Tabela 8 como opções de navegação/operação. A navegação/operação do botão selecionado é habilitada com um clique do mouse na viewport, no caso dos botões de navegação/inserção e seleção. No caso da opção *Fly*, apenas um clique no ícone já é suficiente para ativar a opção em questão.

Figura 51: Ferramentas de navegação do *City Editor*.

Figura 52: Menu Organize.

Tabela 8: Objetos definidos a partir de MFC e seu comportamento.

	Pan: Ativa o modo de pan no ambiente. Este modo permite que o usuário caminhe no plano que coincide com o do monitor(daí o nome **Pan**). Com isso, pode-se ir para direita ou para esquerda sem que se vire para o lado, numa visão em Perspectiva. Na visão ortogonal, permite que o usuário navegue em todas as direções permitidas, norte, sul, leste e oeste.
	Turn and Zoom: Ativa o modo que permite que o usuário se vire em qualquer direção, inclusive não paralelas ao plano XZ, mas desabilita a caminhada em qualquer direção da visão em perspectiva. Caso o seja pressionado na visão ortogonal, o modo se torna o modo de zoom, em que é possível fazer o ambiente ser dimensionado para aumentar ou diminuir a área de visão que temos no ambiente dentro da viewport.
	Fly: Habilita o vôo, ou seja, a navegação paralela ao plano que corta o meio da tela. Esta navegação, além de permitir caminhar para frente e para trás no ar, também permite que sejam feitas curvas para esquerda e direita.
	Select/Insert: Seleciona ou Insere prédios ou Rodovias em um determinado ponto que o mouse foi clicado. Os prédios ou rodovias a serem inseridos são definidos a partir da opção selecionada em um dos formulários localizados nas páginas 119 e 121. A seleção está disponível nas visões ortogonal e perspectiva, seleções múltiplas são ativadas com a tecla Ctrl e agrupamentos são permitidos através do menu mostrado na Figura 52. Entretanto, a inserção está disponível apenas na visão ortogonal, por conta de sua melhor referência espacial em relação ao plano XZ.

7.3.1.4 Painel do Prédio Importado

Por fim, chegamos aos formulários definidos pelo *City Editor*. O primeiro dos formulários definidos é o de importação de prédios, mostrado na Figura 53.

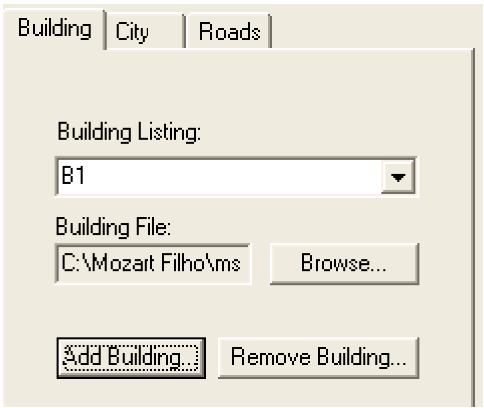


Figura 53: Painel de importação do prédio.

Para importar um prédio, basta clicar no botão "*Browse...*" e localizar o arquivo desejado. Este arquivo é importado e terá o nome idêntico ao que foi definido durante sua criação por parte do *Building Editor*.

Por fim, adicione ou modifique o prédio usando o botão "*Add Building...*". Com o uso deste botão, caso um prédio que possua um mesmo nome já esteja definido, será feita a substituição dele pela prédio atualmente importado.

Por último, temos o botão "*Remove Building...*", que remove um prédio definido dentro de "*Building Listing*".

Uma pergunta que pode surgir com relação ao prédio e sua importação é: Por que a função de importação do prédio não está no mesmo *Painel da Cidade*, onde fazemos a instanciação (veja seção 7.3.1.5)? A resposta está no fato do *Painel da Cidade* já estar excessivamente sobrecarregado com informações sobre a instanciação do prédio, resolvemos separar as funções de importação e instanciação para facilitar a amigabilidade da ferramenta.

7.3.1.5 Painel da Cidade

Neste formulário, mostrado na Figura 54, iremos criar instâncias dos prédios importados. Este mecanismo de instância serve para definir múltiplas versões de um mesmo prédio. Isto é muito importante quando se trata de lotes residenciais, em que prédios iguais estão espalhados nas diversas posições do terreno.

A seguir, daremos a descrição dos ítens que compõem este painel:

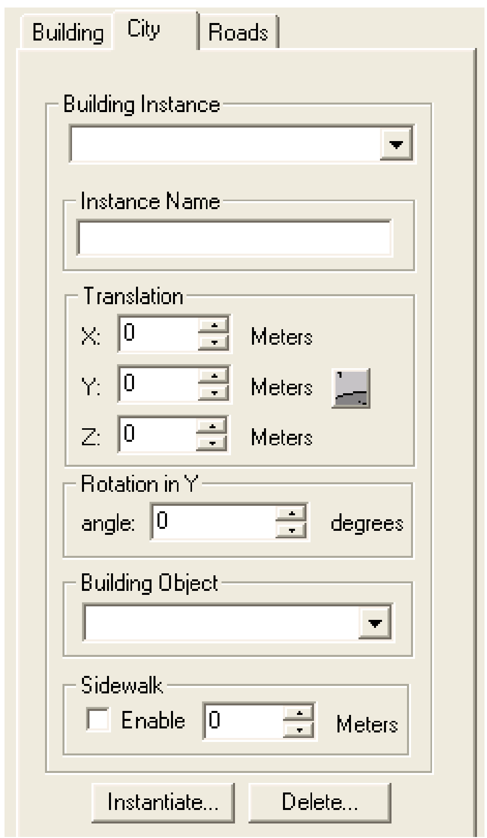


Figura 54: Painel da Cidade.

Building Instance: Listagem de todas as instâncias criadas.

Instance Name: Define o nome da instância de prédio que está sendo editado atualmente.

Translation: Define a posição no espaço do prédio atual. Esta posição é dada em coordenadas cartesianas fornecida em metros. O botão do lado da componente Y da translação define um comportamento que calcula automaticamente a posição vertical do prédio para que ele fique estabelecido sobre o terreno, sem flutuar ou afundar nele. Além disso, este botão habilita a planificação do terreno onde o prédio estará localizado. Um exemplo de planificação para um único prédio pode ser visto na Figura 55, onde vemos o editor com a visão superior, na viewport mais acima, e em perspectiva, na viewport abaixo, do mesmo ambiente. Esta planificação é feita baseada na área ocupada pela calçada do prédio.

Rotation in Y: Oferece um mecanismo de rotação do prédio no eixo Y. Este mecanismo só é oferecido neste eixo por querermos evitar situações onde o prédio fique inclinado como a torre de pizza, por exemplo.

Building Object: O objeto que foi importado anteriormente e que terá sua instância criada.

Sidewalk: Habilita ou desabilita a visualização de calçadas nas laterais do prédio, e fornece seu tamanho.

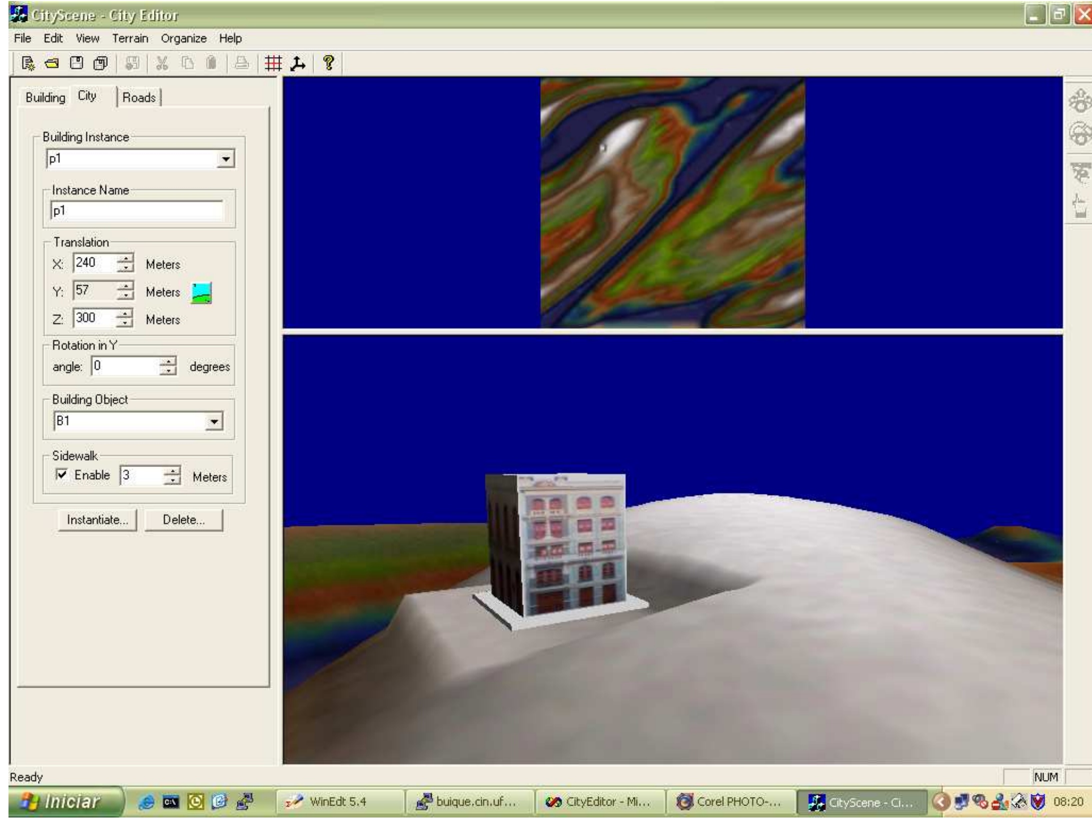


Figura 55: Prédio em um trecho de terreno planificado.

Após preenchido o formulário, devemos apertar o botão *Instantiate*, o qual cria a instância que passa a ser visível na tela ou substitui uma instância com o mesmo nome, se já existir. O Botão *Remove*, por sua vez, remove uma instância de prédio cujo nome está definido em *Instance Name*.

7.3.1.6 Painel de Rodovias

Por fim, chegamos ao último formulário do nosso editor: o formulário de rodovias, vide Figura 56, que para nós são estruturas que sempre acompanham o terreno e funcionam em linha reta. As rodovias são feitas de estruturas que lembram placas de concreto, estruturas que são por definição mono espaçadas de 1 metro.

Para melhor entendemos a funcionalidade da rodovia, vejamos os ítens que compõem o seu formulário, mostrado na figura ao lado.

***Roads listing*:** Listagem de todas as rodovias criadas.

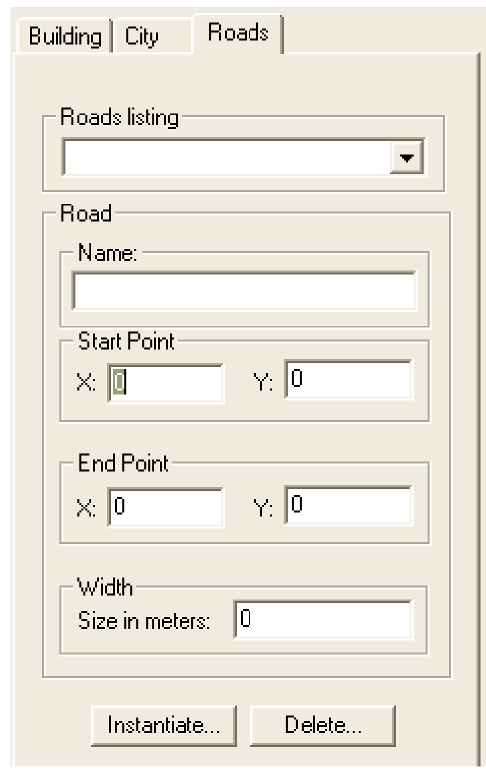


Figura 56: Painel de Rodovias.

***Name*:** Define o nome da rodovia.

***Start Point*:** Define um ponto onde a rodovia começará. O ponto $(0, 0)$ é na visão ortogonal do ponto mais superior esquerdo do terreno importado.

***End Point*:** Define um ponto onde a rodovia terminará. Tanto o *Start Point* quanto o *End Point* definem distâncias em metros a partir da origem.

***Width*:** Largura em metros da rodovia.

Após preenchido o formulário, devemos apertar no botão *Instantiate*, o qual cria a rodovia que passa a ser visível na tela ou substitui uma rodovia com o mesmo nome, se já existir. O Botão *Remove*, por sua vez, remove uma rodovia cujo nome está definido em *Name*. Na Figura 57 podemos ver uma rodovia em um ambiente aberto.

7.3.2 Barra de Ferramentas Principal

Para não deixarmos um dos ítens do editor de cidades sem explanação, podemos facilmente dizer que esta barra de ferramentas tem as mesmas opções oferecidas pelo *Building Editor*. Portanto, aqueles interessados em ver o comportamento de algum ítem específico desta barra de ferramentas devem olhar a seção 7.2.1.

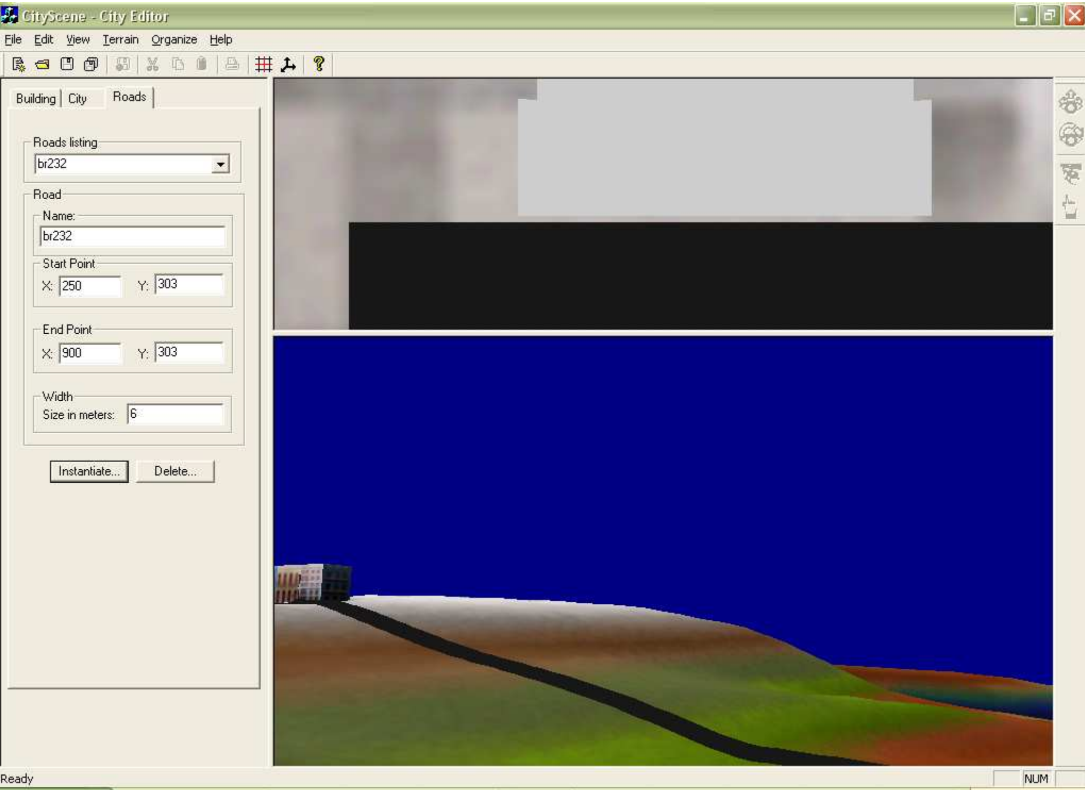


Figura 57: Ambiente com uma rodovia.

Por fim, chegamos ao fim do guia de usuário deste editor, com um possível resultado final sendo mostrado na Figura 58.

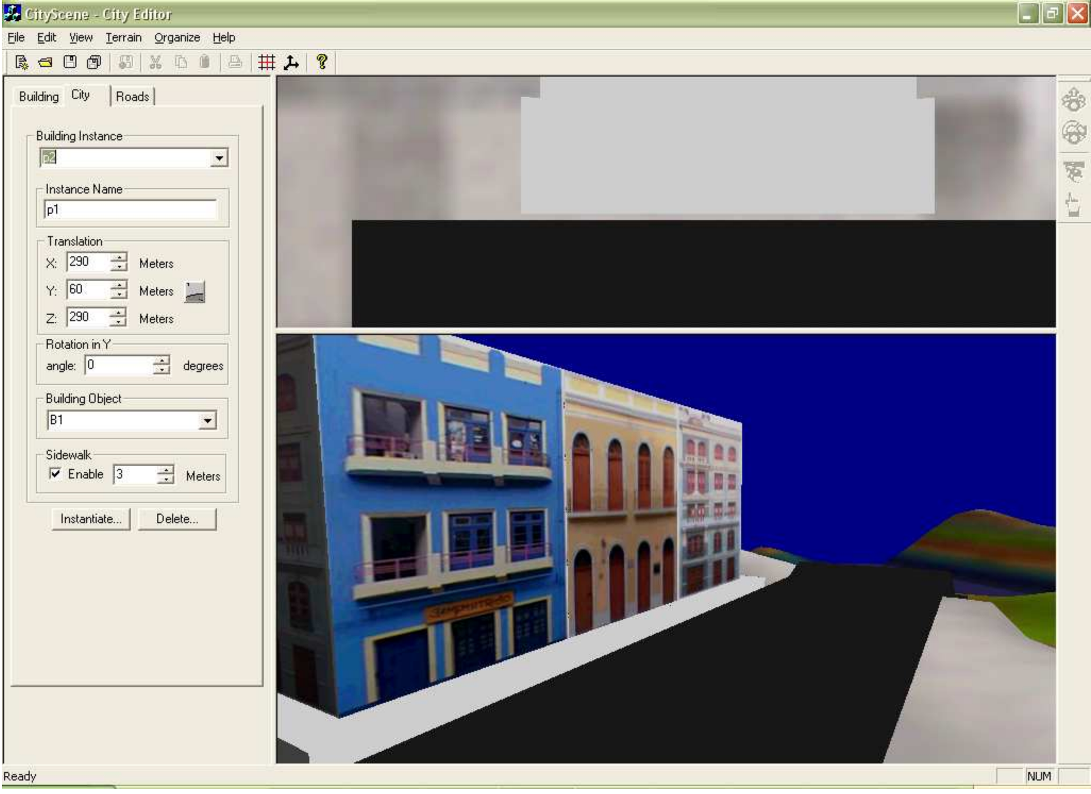


Figura 58: Exemplo de ambiente no City Editor.

7.4 Comentários adicionais

Agora que já sabemos da funcionalidade dos editores, justificaremos algumas decisões de funcionamento da interface gráfica tomadas. Primeiramente, analisaremos o porque dos comandos estarem disponíveis por meio de formulários.

A opção de formulários, além de definir uma maneira direta de agrupar ítens de uma mesma primitiva, também oferece a possibilidade da troca eficiente de primitivas sem muitas janelas a serem abertas e fechadas. Outro fator é importante de se notar é que os formulários não são a única maneira de fazermos a inserção de primitivas no ambiente do *City Editor*, esta inserção também é possível através do botão "*Select/Insert*", descrito na seção 7.2.3 (ver Tabela 8 na página 118), opção bem mais intuitiva a um usuário leigo. Entretanto o formulário é uma opção disponível para aqueles que querem maior precisão no fornecimento de coordenadas espaciais.

Outra simplificação foi feita na escolha das cores correspondentes ao terreno. Adotamos a convenção de fornecer apenas 16 tipos de cores, onde originalmente poderiam haver 256 valores possíveis de serem usados no motor. Esta redução de espaço do número de cores possíveis foi feita com o intuito de aumentar a amigabilidade da ferramenta final e não reduzir a qualidade final do ambiente resultante, dado que novas cores são geradas a partir da interpolação linear das componentes RGB (FOLEY, 1995), entre cores vizinhas

na amostra de 16 cores inicial.

Outro ponto que pode ser observado nos dois editores é o mecanismo usado na navegação. A navegação por botão é usada em muitos *plugins* de visualização de formatos 3D como VRML, mas também é usada em muitas ferramentas de autoria como o World-Up (Sense 8, 2003), por exemplo. Normalmente, esta opção de navegação se adequa bem às necessidades do usuário por ser uma maneira mais intuitiva de navegação. Entretanto, o *Building Editor* oferece um outro mecanismo de navegação, caracterizado pelo uso de teclado intercalado com mouse, aqui resumidamente chamado de mecanismo de *teclas intercaladas*. Esse mecanismo é conhecido entre os jogos de primeira geração como *Quake* (id Software, 2003), ou jogos modernos como *Unreal Tournment* (Unreal Tournament, 2003). Este último mecanismo, apesar de menos intuitivo, caracteriza-se por permitir uma navegação menos cansativa, mais rápida e direta do que o mecanismo de botões.

Ambos os mecanismos foram implementados nos dois editores para permitir um melhor estudo e análise deles. Além de permitir-nos visualizar a implantação diferenciada desses mecanismos numa Janela-Ativa (*Viewport*), estes mecanismos nos permitem observar também as diferenças na navegação inerentes a editores de objetos, como o *Building Editor*, onde o foco está no objeto localizado na origem e todas as transformações são feitas tendo a origem ou objeto como base, e de cenários, como o *City Editor*, no qual as transformações são feitas tendo a câmera como base e todas as transformações são feitas em torno da posição e orientação do usuário no espaço.

Assim, acabamos nossa discussão caracterizando o porque de cada um dos mecanismos colocados nos dois editores. Mostramos que, apesar de sua simplicidade, os dois editores levantam questões relevantes no que diz respeito a interface homem-máquina.

8 Conclusão e Trabalhos Futuros

Assim, chegamos ao fim de nosso trabalho com dois editores de cenários funcionais. Um que se apresenta como editor de objetos, os nossos prédios, e outro que é o editor de cenários propriamente dito, o editor de cidades. A construção destes editores levantou questões sobre a maneira correta de se integrar eventos gerados pelo usuário com objetos, primitivas e objetos fornecidos pelo motor, que por sua vez também tem que oferecer uma abstração de alto nível ao programador.

Neste trabalho também abordamos questões fundamentais de programação 3D que estão intrínsecas a qualquer tipo de aplicação deste tipo e não somente a editores de cenários em OpenGL e podem iniciar a construção de um motor OpenGL mais robusto e versátil.

Mostramos, também, durante o desenvolvimento desta aplicação técnicas de engenharia de software, como padrões de projeto, podem e devem ser usadas na construção de uma aplicação de qualidade. Estes padrões de projeto, consolidaram com muita facilidade, muitas das questões arquiteturais do projeto que foram levantadas durante o desenvolvimento.

Encaramos um verdadeiro desafio no aspecto de programação para Windows em MFC, que oferece mecanismos que se assemelham à orientação a objetos em C++, mas não atendem completamente a este propósito. Desafio aparentemente simples, mas que foi um dos ítens mais trabalhosos do projeto. Para que conseguíssemos possuir uma integração de MFC com as outras ferramentas e bibliotecas desenvolvidas em um paradigma orientado a objetos e no padrão ANSI, foi necessária muita reflexão e estudo.

O desafio de programação se torna ainda maior quando questões de interface Homem-Máquina entram no escopo do projeto. Questões que foram analisadas e traduzidas na forma de padrões que facilitam a construção de futuras ferramentas.

Entre os ítens correspondentes à trabalhos futuros, pode-se sugerir:

- Melhoramento do motor que pode contar com diversos aperfeiçoamentos. Um exemplo que pode ser dado para aperfeiçoamento do motor é a separação da malha em três classes, uma para transformações geométricas com o uso de pilha de matrizes,

outra para agrupamento com sub-malhas e outra para agrupamento de outras malhas. Este aperfeiçoamento, apesar de simples, ajudaria muito na legibilidade do motor por parte de um programador externo.

- Permitir ao motor não somente a exportação mas também a importação de objetos VRML.
- A construção de um editor mais complexo que permita construção de cenários mais realistas e com maior nível de interatividade. Realismo realmente foi um ítem que não foi buscado durante o desenvolvimento desse projeto por ser um trabalho muito minucioso e de pouca relevância acadêmica para um trabalho com um escopo como o do nosso.
- Uma análise mais profunda da questão de interface homem-máquina que traga novas sugestões de interação para aplicações deste tipo.

Com estas mudanças teríamos uma ou mais aplicações em nível de competição em mercado e que, além disso, abririam um leque acadêmico de publicações dentro do desenvolvimento deste tipo de aplicação. Uma área que ainda está em crescimento e que tem muito a dar.

Referências Bibliográficas

AC3D Modeler. 2001. Site AC3D - 3D Graphics modeller. Disponível em: <http://www.ac3d.org>. Acesso em: 19 setembro 2001.

ALMEIDA, A. L.; ARAÚJO FILHO, M. S. C.; TIMES, V. C. Algoritmos de roteamento para o desenvolvimento de aplicações de páginas amarelas para web. In: *VIII CONIC - Congresso de Iniciação Científica*. Universidade Federal de Pernambuco, Recife-PE: Editora Universitária UFPE, 2000.

ARAÚJO FILHO, M. S. C. *Iniciação Científica*. 2001. Site Home Page de Mozart Filho. Disponível em: <http://www.cin.ufpe.br/∼msca>. Acesso em: 10 de Junho de 2001.

ARAÚJO FILHO, M. S. C. *Tese de Mestrado - Editor de Cenários Urbanos*. 2003. Site Home Page de Mozart Filho. Disponível em: <http://www.cin.ufpe.br/∼msca/msc/>. Acesso em: 5 de Janeiro de 2003.

ARAÚJO FILHO, M. S. C.; ALMEIDA, A. L.; TIMES, V. C. Uma arquitetura independente de sig para o desenvolvimanto de aplicações de páginas amarelas para web. In: *VIII CONIC - Congresso de Iniciação Científica*. Universidade Federal de Pernambuco, Recife-PE: Editora Universitária UFPE, 2000.

ARAÚJO FILHO, M. S. C.; COSTA, M. A. S.; PESSOA, B. D. S. Visualização científica e análise de dados: Mundos virtuais e cyber arquitetura. In: *VI CONIC - Congresso de Iniciação Científica*. Universidade Federal de Pernambuco, Recife-PE: Editora Universitária UFPE, 1998.

ARAÚJO FILHO, M. S. C.; FRERY, A. C. Visualização científica e análise de dados: Mundos virtuais e cyber arquitetura. In: *VII CONIC - Congresso de Iniciação Científica*. Universidade Federal de Pernambuco, Recife-PE: Editora Universitária UFPE, 1999.

ARAÚJO FILHO, M. S. C.; FRERY, A. C. Visualização científica e análise de dados: Mundos virtuais e cyber arquitetura. In: *VIII CONIC - Congresso de Iniciação Científica*. Universidade Federal de Pernambuco, Recife-PE: Editora Universitária UFPE, 2000.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 10520*: Informação e documentação — apresentação de citações em documentos. Rio de Janeiro, jul. 2001. 4 p.

Autodesk. 2003. Site discreet.com > Welcome to Discreet. Disponível em: <http://www.discreet.com>. Acesso em: 13 de Janeiro de 2003.

Blaxxun. 2003. Site blaxxun interactive. Disponível em: <http://www.blaxxun.com/>. Acesso em: 13 de Fevereiro de 2003.

C & MFC – Table of Contents. 2002. Site CodeGuru. Disponível em: <http://www.codeguru.com/cpp_mfc/index.shtml>. Acesso em: 24 fevereiro 2002.

Código Livre. 2003. Site CodigoLivre: Informações sobre Projeto – abnTeX. Disponível em: <http://codigolivre.org.br/projects/abntex/>. Acesso em: 6 de Janeiro de 2003.

CiteMap Builder. 2002. Site Trivista Technologies, Inc. Disponível em: <http://www.trivista.com/products/citemap/citemapbuilder.htm>. Acesso em: 19 de Setembro de 2002.

Corel Corporation. 2003. Site Corel Corporation Landing Page. Disponível em: <http://www.corel.com/>. Acesso em: 04 de Janeiro de 2003.

COSTA, M. A. S.; ARAÚJO FILHO, M. S. C.; PESSOA, B. D. S. Visualização científica e análise de dados: Visualização de informações multidimensionais. In: *VI CONIC - Congresso de Iniciação Científica*. Universidade Federal de Pernambuco, Recife-PE: Editora Universitária UFPE, 1998.

DEITEL, H. M.; DEITEL, P. J. C. *C++ Como Programar*. 3 ed. Porto Alegre, SC: Prentice Hall, 2001.

DevCentral. *Introduction to MFC Programming with Visual C++ 6.X*. 2002. Site DevCentral Home. Disponível em: <http://devcentral.iftech.com/articles/MFC/vc6_mfc/default.php>. Acesso em: 15 de Fevereiro de 2002.

DevCentral. *Understanding the AppWizard and ClassWizard in Visual C++ Version 6.X*. 2002. Site DevCentral Home. Disponível em: <http://devcentral.iftech.com/articles/MFC/vc6_tools/default.php>. Acesso em: 15 de Fevereiro de 2002.

DÖLLNER, J.; BAUMANN, K.; HINRICHS, K. Texturing techniques for terrain visualization. In: *IEEE Visualization 2000*. Salt Lake City, UT: IEEE Press, 2000. p. 227–234. ISBN 0-7803-6478-3.

DÖLLNER, J.; HINRICHS, K. A generic rendering system. *IEEE Visualization and Computer Graphics*, v. 8, n. 2, p. 99–118, April-June 2002.

DUCHAINEAU, M.; WOLINSKY, M.; SIGETI, D.; MILLER, M.; ALDRICH, C.; MINEEV-WEINSTEIN, M. ROAMing terrain: Real-time optimally adapting meshes. In: IEEE. *Procedings of 8th IEEE Visualization '97 Conference*. Phoenix, AZ, 1997. p. 81–88.

FOLEY, J. D.; VAN DAM, A.; FEINER, S. K.; HUGHES, J. F. *Computer Graphics: Principles and Practice in C*. 2 ed. Reading, MA: Addison-Wesley, 1995.

FRERY, A.; GROTH, B.; JÄHNICHEN, S.; KELNER, J.; DUDZIAK, T.; TEICHRIEB, V.; ARAÚJO FILHO, M. S. C.; COSTA, M. A. S.; PESSOA, B. D. S. Convira - conceptual navigation in virtual reality applications. 5th *German - Brazilian And German - Argentinean Workshop On Information Technology*, GMD FIRST, Königswinter. Berlin, 1999.

FRERY, A. C. *Computação Gráfica*. 2002. Site Centro de Informática da Universidade Federal de Pernambuco. Disponível em: <http://www.cin.ufpe.br/∼if291>. Acesso em: 4 de Dezembro de 2002.

FRERY, A. C. *Computação Gráfica*. 2003. Site Computação Gráfica - Centro de Informática da Universidade Federal de Pernambuco. Disponível em: <http://www.cin.ufpe.br/∼if291/galeria/galeria.htm>. Acesso em: 08 de Fevereiro de 2003.

FRERY, A. C.; KELNER, J. *Realidade Virtual e Multimídia*. 2002. Site Centro de Informática da Universidade Federal de Pernambuco. Disponível em: <http://www.cin.ufpe.br/∼if124>. Acesso em: 4 de Dezembro de 2002.

FRERY, A. C.; KELNER, J.; MOREIRA, J. R.; PESSOA, B. D. S.; ALHEIROS, D. M.; ARAÚJO FILHO, M. S. C.; TEICHRIEB, V. Desktop virtual reality in the assessment of tidal effect. In: IEEE. *International Geoscience and Remote Sensing Symposium: The Role of Remote Sensing in Managing the Global Environment*. Hawaii: IEEE Press, 2000. CD-ROM.

Game Institute. 2002. Site Game Institute: Game Programming Education. Disponível em: <http://www.gameinstitute.com>. Acesso em: 17 de Abril de 2002.

GameDev. 2001. Site GameDev.net - all your game development needs. Disponível em: <http://www.gamedev.net>. Acesso em: 20 de Setembro de 2001.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.; BOOCH, G. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

GTK. 2003. Site GTK+ - The GIMP Toolkit. Disponível em: <http://www.gtk.org>. Acesso em: 07 de Fevereiro de 2003.

HELIUM, N. *NeHe Productions (OpenGL)*. 2001. Site GameDev.net - all your game development needs. Disponível em: <http://nehe.gamedev.net/>. Acesso em: 15 de Março 2001.

HILL JR., F. S. *Computer Graphics Using Open GL*. 2 ed. Upper Saddle River, New Jersey: Prentice Hall, 2000.

HOPPE, H. Progressive meshes. In: *Proceedings of SIGGRAPH 96*. New Orleans, Louisiana: ACM SIGGRAPH / Addison Wesley, 1996. (Computer Graphics Proceedings, Annual Conference Series), p. 99–108.

id Software. 2003. Site id Software. Disponível em: <http://www.idsoftware.com/>. Acesso em: 6 de Janeiro de 2003.

Intel® JPEG Library. 2002. Site Intel Corporation - Welcome to Intel.com. Disponível em: <http://www.intel.com/software/products/perflib/ijl/index.htm>. Acesso em: 19 de Março de 2002.

Internet Space Builder. 2001. Site Parallel Graphic(a 3D VRML Company). Disponível em: <http://www.parallelgraphics.com/products/isb>. Acesso em: 19 de Setembro de 2001.

JAIN, A. K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall International Editions, 1989.

KELNER, J.; FRERY, A. C.; TEICHRIEB, V.; ARAÚJO FILHO, M. S. C. Simulation and assessment of flooding with desktop virtual reality. In: SBC. *IV Simpósio de Realidade Virtual*. Florianópolis, SC, 2001. p. 56–66.

KELNER, J.; FRERY, A. C.; TEICHRIEB, V.; VASCONCELOS, G. P.; ARAÚJO FILHO, M. S. C.; BARROS, P. G. Modelagem virtual urbana: Perspectiva histórica e estudo de caso. In: *V Symposium on Virtual Reality*. Fortaleza, CE: SBC, 2002. p. 317–328.

Libpng Home Page. 2002. Site PNG(Portable Network Graphics) Home Site. Disponível em: <http://www.libpng.org/pub/png/libpng.html>. Acesso em: 19 de Março de 2002.

LINDSTROM, P.; KOLLER, D.; RIBARSKY, W.; HUGHES, L. F.; FAUST, N.; TURNER, G. Real-time, continuous level of detail rendering of height fields. In: *Proceedings of SIGGRAPH 96*. New Orleans, Louisiana: ACM SIGGRAPH / Addison Wesley, 1996. (Computer Graphics Proceedings, Annual Conference Series), p. 109–118.

LINDSTROM, P.; PASCUCCI, V. Visualization of large terrains made easy. In: IEEE. *IEEE Visualization 2001*. San Diego, CA, 2001. v. 12, p. 363–370. ISBN 0-7803-7200-x.

LINDSTROM, P.; PASCUCCI, V. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, v. 8, n. 3, p. 239–254, July - September 2002. ISSN 1077-2626.

MASCARENHAS, N. D. A.; VELASCO, F. R. D. *Processamento Digital de Imagens*. Buenos Aires: Kapelusz, 1989. IV Escola Brasileiro-Argentina de Informática.

Microsoft. 2003. Site DirectX Home Page. Disponível em: <http://www.microsoft.com/directx/>. Acesso em: 04 de Janeiro de 2003.

MSDN Home. 2002. Site Welcome to the Microsoft Corporate WebSite. Disponível em: <http://msdn.microsoft.com/>. Acesso em: 10 de Janeiro de 2002.

MURRAY, J. E.; RYPER, W. V. *Encyclopedia of Graphic File Formats*. Cambridge: O'Reilly, 1994.

New Page 1. 2001. Site MFC & OpenGL. Disponível em: <http://pws.prserv.net/mfcogl/bitmaps.%20MFC%20&%20OpenGL.htm>. Acesso em: 15 de Março de 2001.

OTTEWELL, P. *Phil Ottewell's STL Tutorial*. 2002. Site Phil Ottewell's Home Page. Disponível em: <http://www.yrl.co.uk/ phil/stl/stl.htmlx>. Acesso em: 15 de Maio de 2002.

ParallelGraphics. 2003. Site Parallel Graphics (a 3D VRML Company). Disponível em: <http://www.parallelgraphics.com/>. Acesso em: 04 de Janeiro de 2003.

PESSOA, B. D. S.; ARAÚJO FILHO, M. S. C.; COSTA, M. A. S. Visualização científica e análise de dados: Organização espacial de dados para a internet. In: *VI CONIC - Congresso de Iniciação Científica*. Universidade Federal de Pernambuco, Recife-PE: Editora Universitária UFPE, 1998.

RÖTTGER, S.; HEIDRICH, W.; SLUSALLEK, P.; SEIDEL, H.-P. Real-time generation of continuous levels of detail for height fields. In: SKALA, V. (Ed.). *Procedings WSCG '98*. Campus Bory, Plzen - Bory, Czech Republic, 1998. p. 315–322.

SANTOS, W. P.; ARAÚJO FILHO, M. S. C.; BARROS NETO, A. J.; SILVA, C. K. R.; SILVA, P. V.; SILVA, J. D.; FRERY, A. C. Uma nova técnica de classificação de imagens baseada em lógica nebulosa não iterativa. *XI Simpósio Brasileiro de Sensoriamento Remoto*, Belo Horizonte, Abril 2003.

Sense 8. 2003. Site Sense 8 Virtual Reality 3D Software. Disponível em: <http://www.sense8.com>. Acesso em: 6 de Janeiro de 2003.

Silicon Graphics Inc. 2003. Site OpenGL - Hight Performance 2D/3D Graphics. Disponível em: <http://www.opengl.org/>. Acesso em: 04 de Janeiro de 2003.

SourceForge.net. 2003. Site MiKTeXProject Page. Disponível em: <http://www.miktex.org/>. Acesso em: 04 de Janeiro de 2003.

SourceForge.net. 2003. Site The Fast Light Toolkit Home Page. Disponível em: <http://www.fltk.org>. Acesso em: 07 de Fevereiro de 2003.

STROUSTRUP, B. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1997.

Sun Microsystems, Inc. 2003. Site The Source for Java (TM) Technology. Disponível em: <http://java.sun.com/>. Acesso em: 13 de Fevereiro de 2003.

The Code Project. 2002. Site The Code Project – Free Source Code and Tutorials. Disponível em: <http://www.codeproject.com/cpp/#Utilities>. Acesso em: 23 de Maio de 2002.

Universidade Federal do Paraná. *Sistemas de Bibliotecas. Normas para apresentacao de documentos cientificos.* Curitiba: Ed. da UFPR, 2000.

Unreal Tournament. 2003. Site Unreal Tournament. Disponível em: <http://www.unrealtournament.com/>. Acesso em: 6 de Janeiro de 2003.

Web3D Consortium. 2001. Site WEB3D CONSORTIUM. Disponível em: <http://www.web3d.org>. Acesso em: 20 de Setembro de 2001.

WOO, M.; NEIDER, J.; DAVIS, T.; SHREINER, D. *OpenGL(r) 1.2 Programming Guide: The Official Guide to Learning OpenGL*. Reading, MA: Addison-Wesley, 1999.

WRIGHT JR, R. S.; SWEET, M. *OpenGL Superbible*. Corte Madera, CA: Waite Group Pr, 1999.

ZONENSCHEIN, R.; GOMES, J.; VELHO, L.; FIGUEIREDO, L. H. Controlling texture mapping onto implicit surfaces with particle systems. In: BLOOMENTHAL, J.; SAUPE, D. (Ed.). *Proceedings of the Third International Workshop on Implicit Surfaces*. Seattle, USA: Eurographics & ACM SIGGRAPH, 1998. p. 131–138.

APÊNDICE A – Diagrama de Classes do Graphic Engine

Neste diagramas temos todas as estruturas que participam do motor, entretanto sem maiores detalhes sobre a coleção de métodos disponíveis. Este diagrama pode ser visto na Figura 59 e está isolado em uma página a parte para permitir sua melhor visualização.

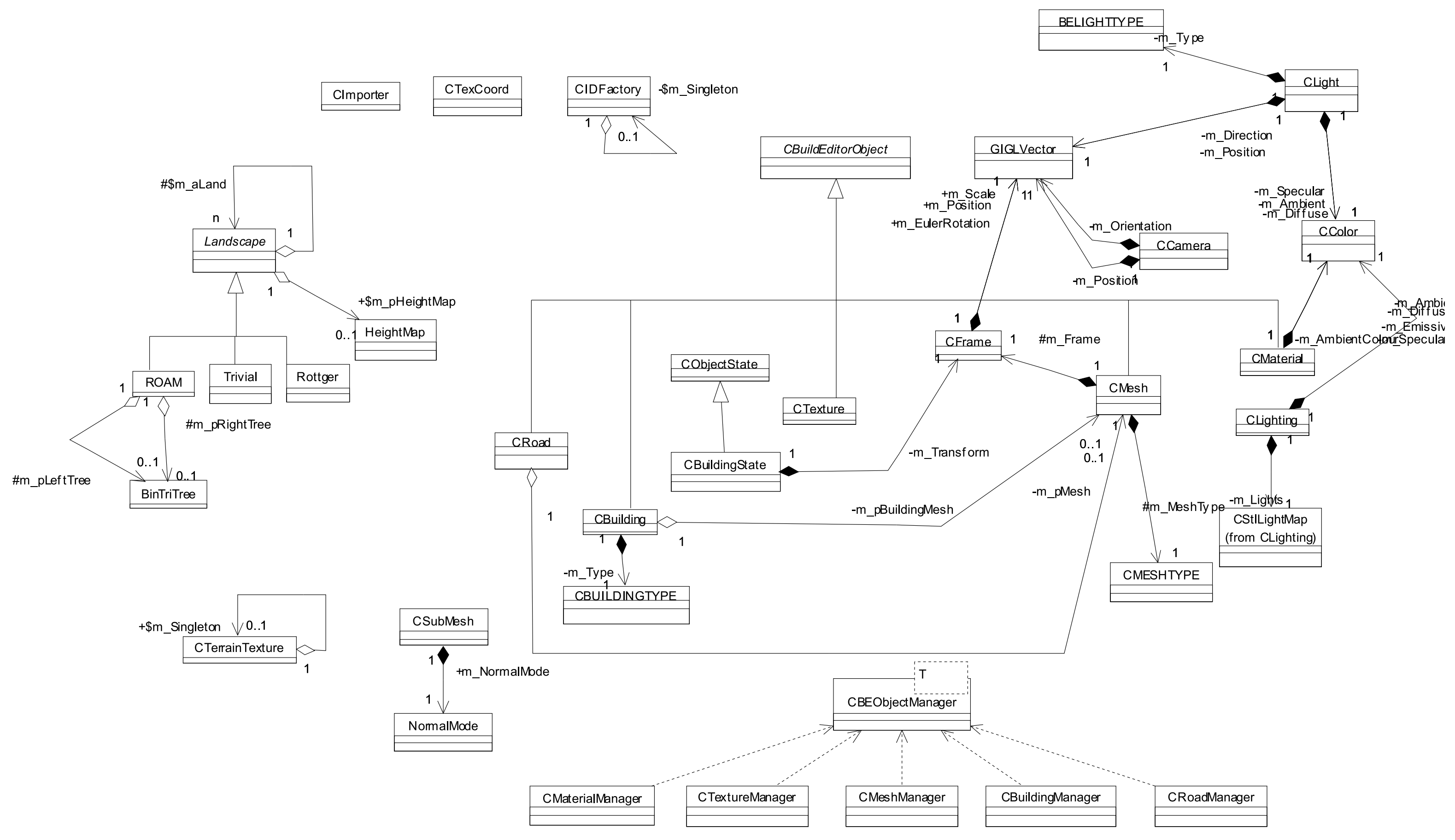
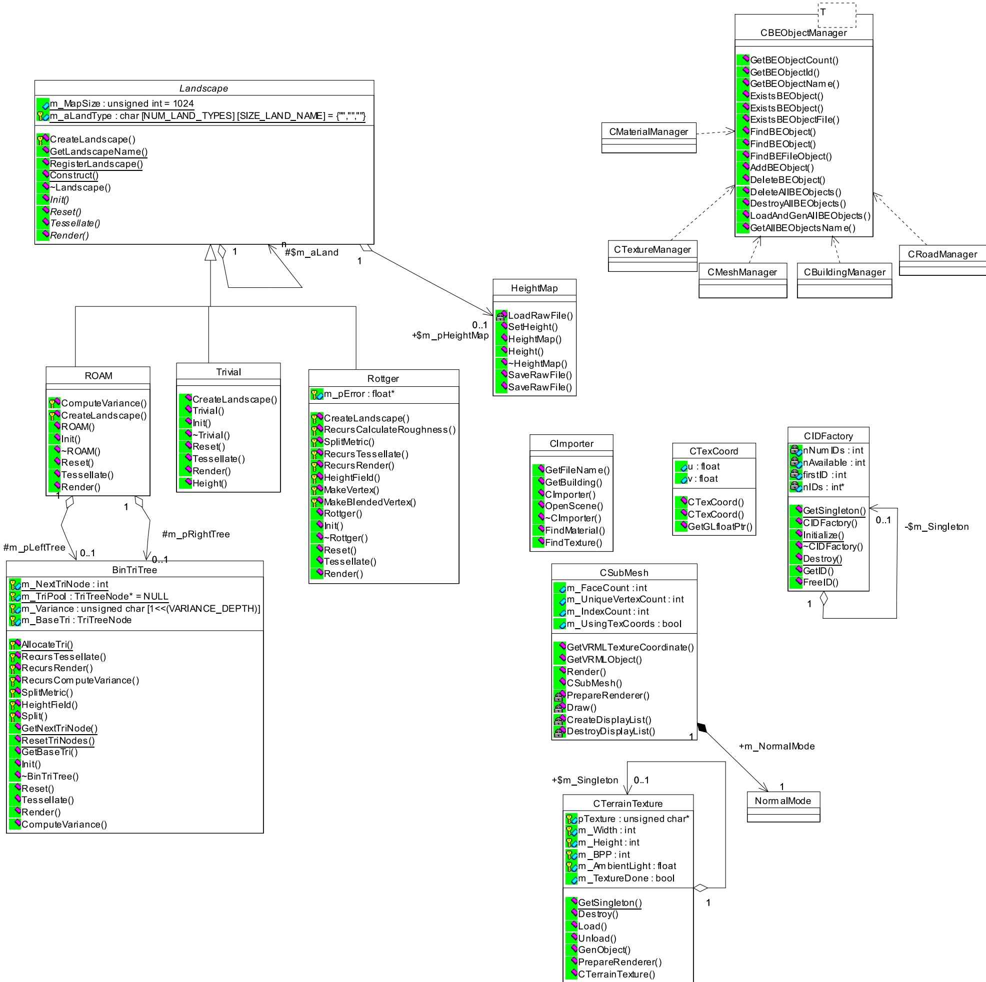


Figura 59: Visão geral do Graphic Engine

APÊNDICE B – Detalhe das Classes do Graphic Engine

Temos aqui os detalhes das estruturas presentes no motor. Detalhes que estão ao nível de enumeração de métodos e atributos das classes dadas.

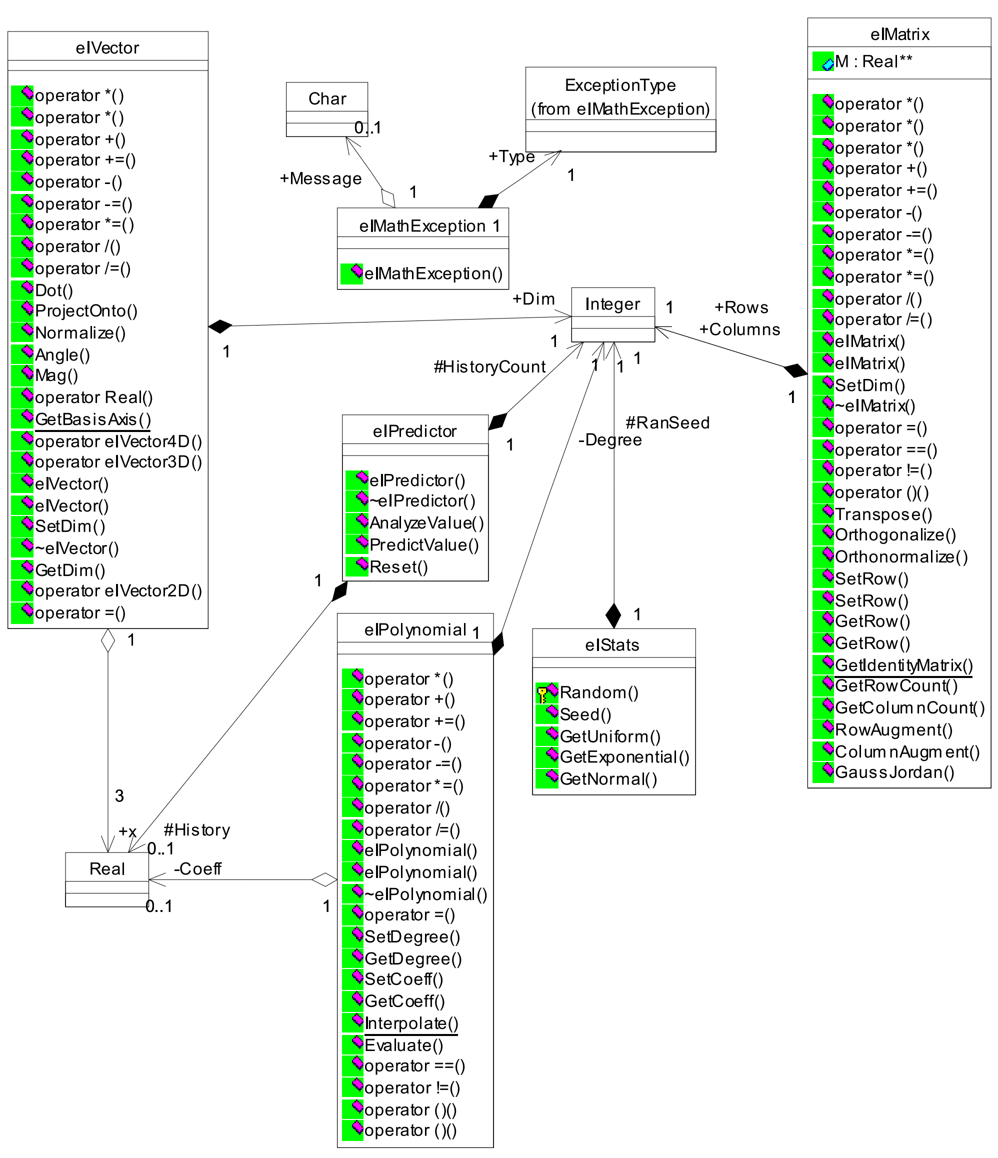
B.1 Diagrama 1



B.2 Diagrama 2

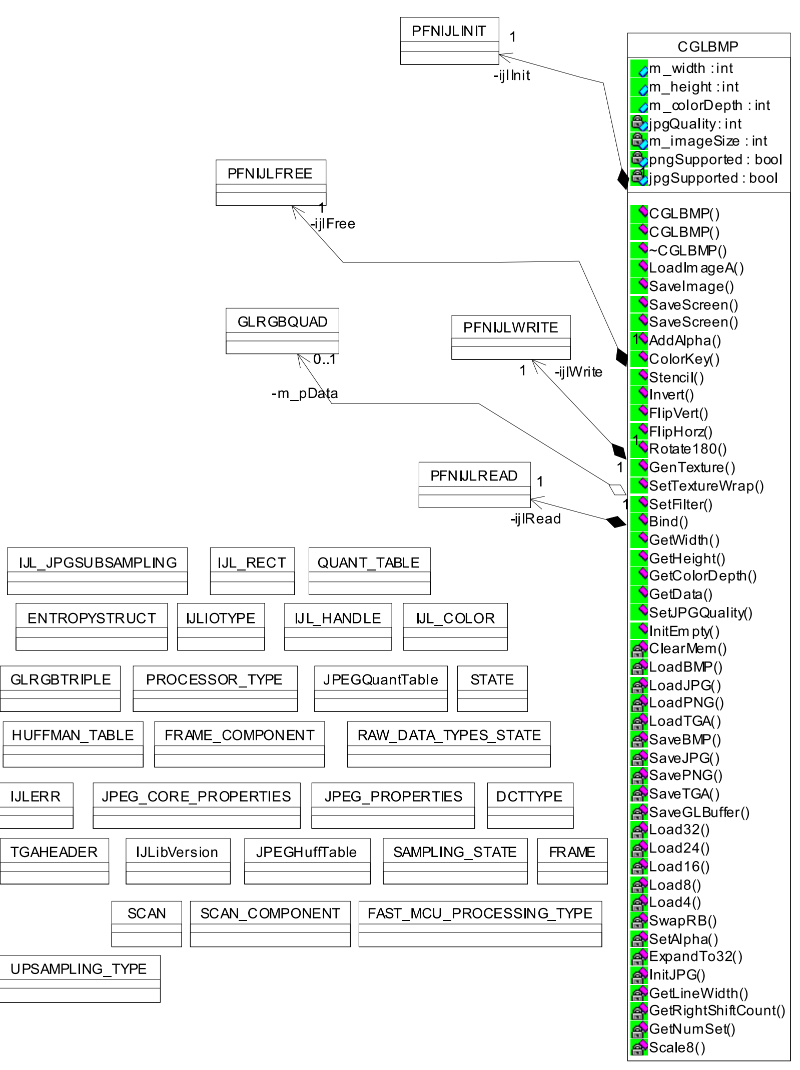
APÊNDICE C – Diagrama de Classes da Biblioteca MathLib

Classes definidas pela biblioteca matemática.



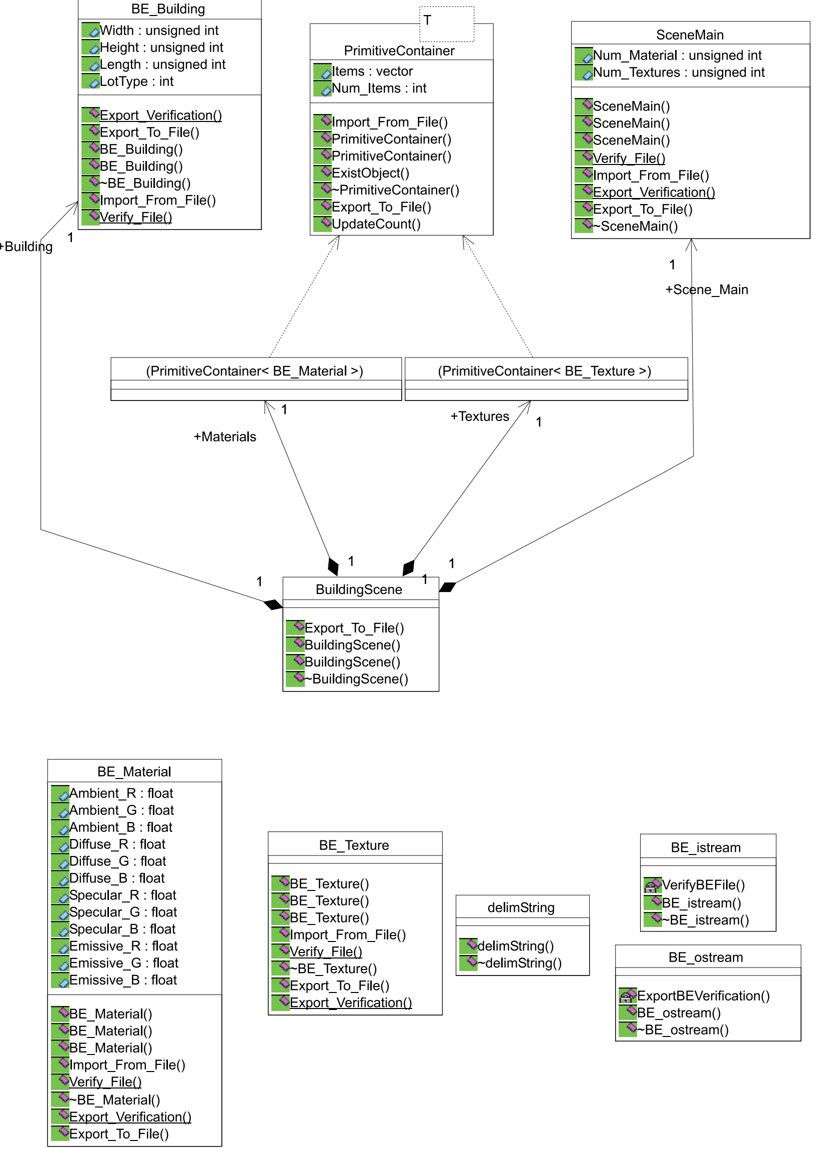
APÊNDICE D – Diagrama de Classes TextureLoader - A classe CGLBMP

Detalhes das classes e estruturas responsáveis pelo carregamento de texturas.



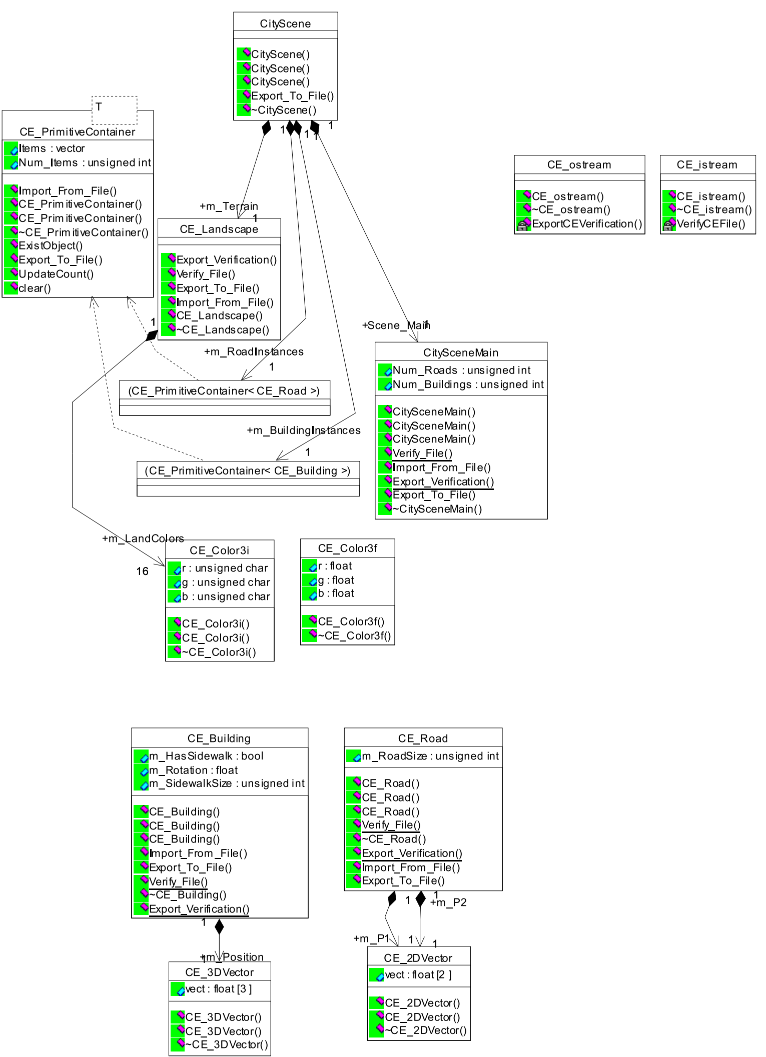
APÊNDICE E – Diagrama de classes do BE_iostream

Estrutura gramatical definida com nós em forma de objetos para ler e escrever um prédio em arquivo.



APÊNDICE F – Diagrama de classes do CE_iostream

Estrutura gramatical definida com nós em forma de objetos para ler e escrever uma cidade em arquivo.



APÊNDICE G – Diagrama de classes do VRMLSaver

Estrutura montada para permitir exportação para VRML. Este diagrama pode ser visto na Figura 60 e está isolado em uma página a parte para permitir sua melhor visualização.

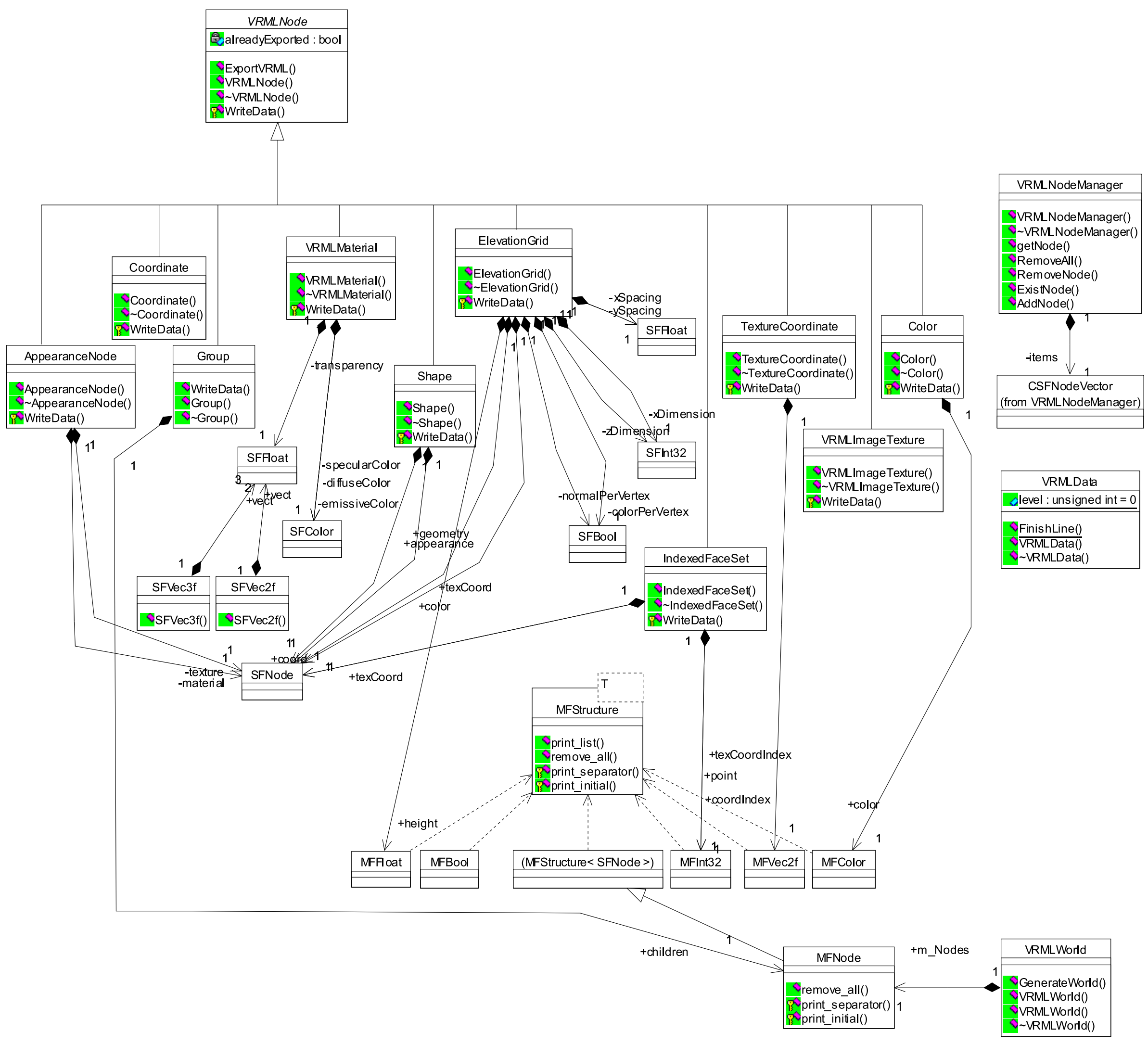
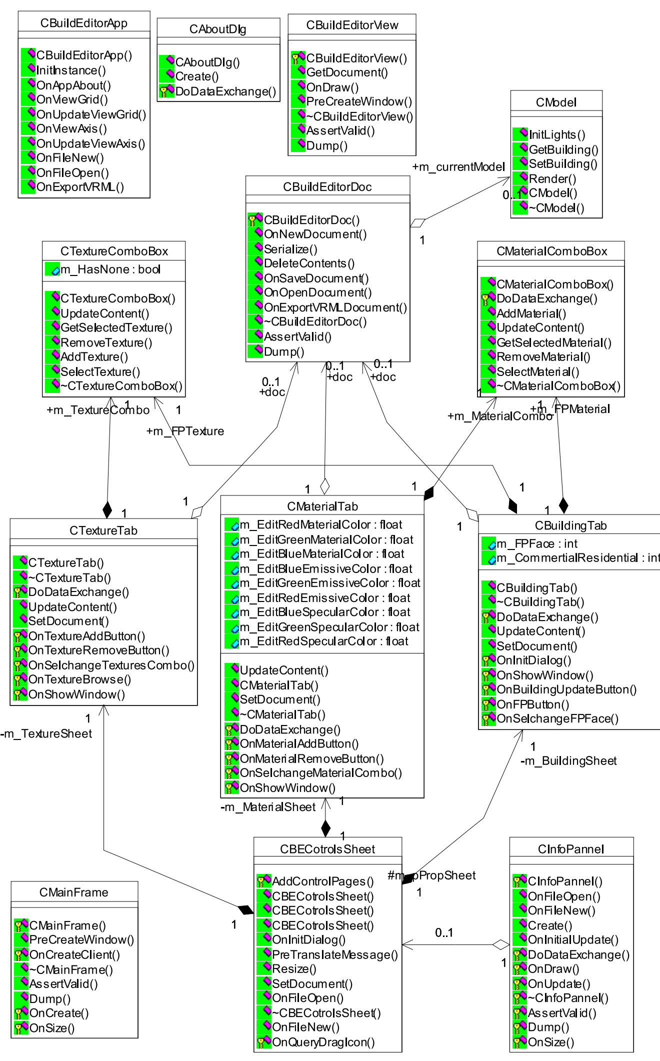


Figura 60: Classes do VRMLSaver

APÊNDICE H – Diagrama de Classe do BuildEditor

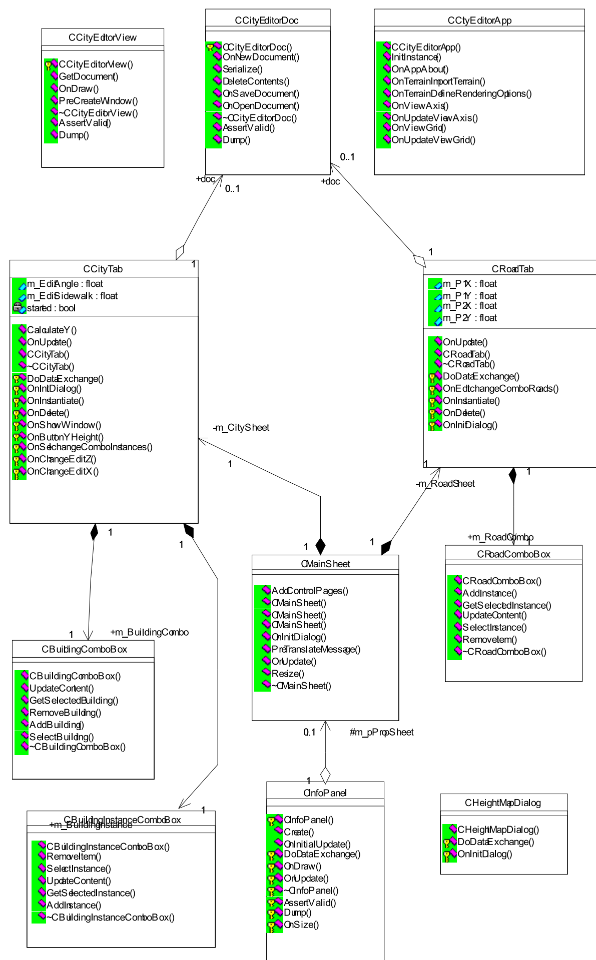
Classes de mais alto nível do editor de prédios.



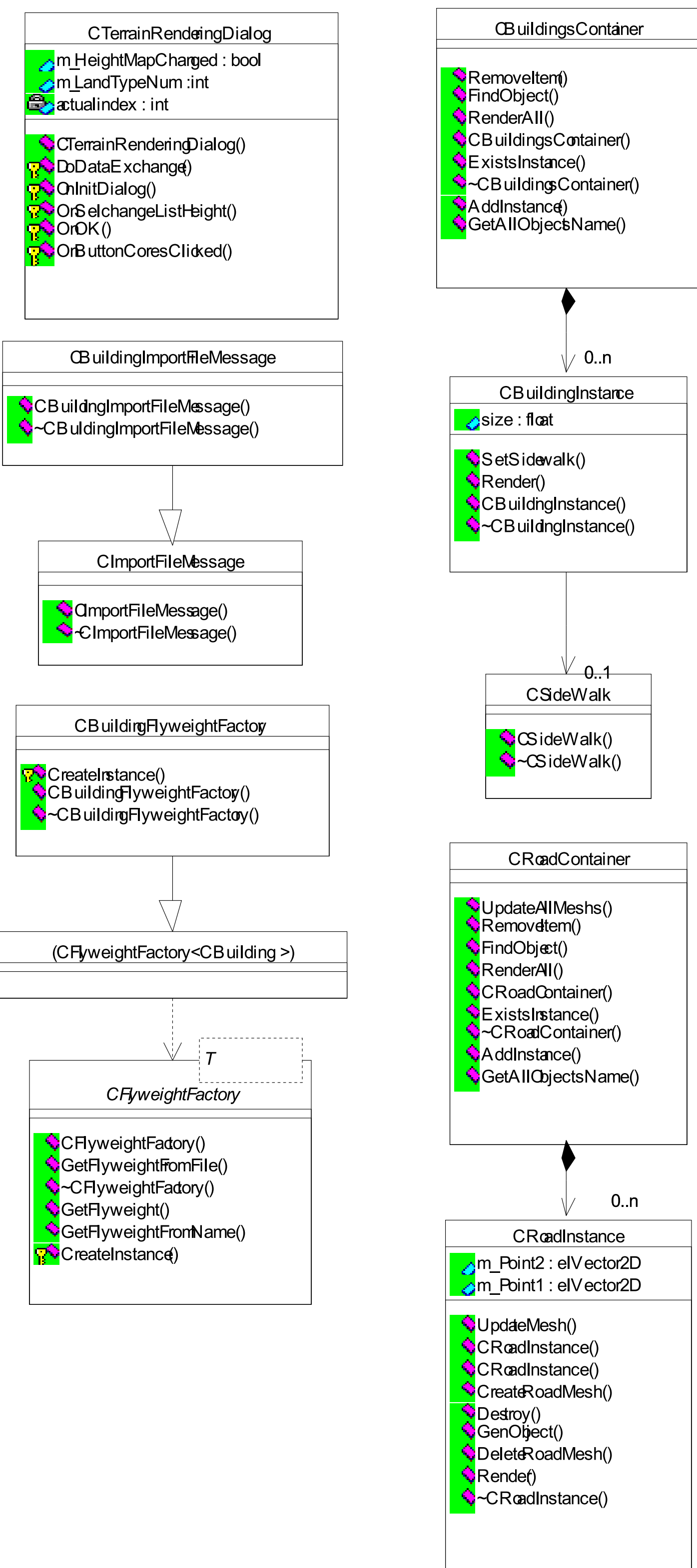
APÊNDICE I – Diagrama de Classe do CityEditor

Classes de mais alto nível do editor de cidades.

I.1 Diagrama 1

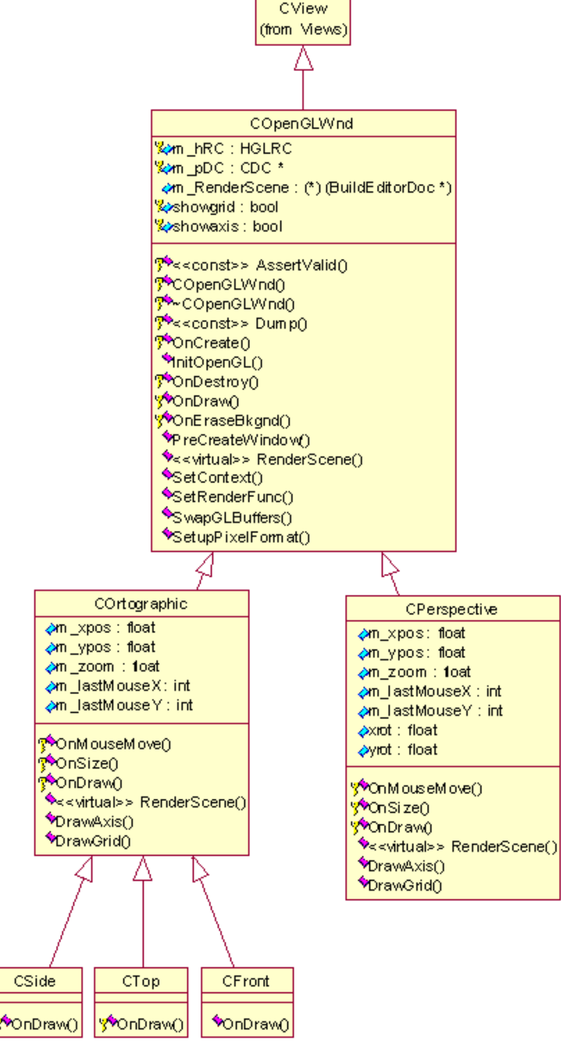


I.2 Diagrama 2



APÊNDICE J – Diagrama de Classes da Viewport

Classes para navegação e criação de Viewport.



APÊNDICE K – Código para renderizar uma sub-malha

```
void CSubMesh::Draw(CMesh & ParentMesh)
{
    GIGLVector *   pVertices  = &ParentMesh.m_VtxPositions[0];
    CTexCoord * pTexCoords;
    GIGLVector * pNormals;


    if(m_UsingTexCoords) {
        pTexCoords = &this->m_TexCoords[0];
    }
    else {
        pTexCoords    = &ParentMesh.m_TexCoords[0];
    }

    glBegin(GL_TRIANGLES);
    for (int i = 0; i < m_FaceCount; i++)
    {
        //isto esta errado
        int VtxIdx0 = m_VertexIndices[3*i];
        int VtxIdx1 = m_VertexIndices[3*i + 1];
        int VtxIdx2 = m_VertexIndices[3*i + 2];

        if(m_NormalMode==NORMALPERVERTEX)
        {
            pNormals   = &ParentMesh.m_VtxNormals[0];
            if (pNormals)
                glNormal3fv((float *)&pNormals[VtxIdx0]);
            if (pTexCoords) {
                if(m_UsingTexCoords)
                    glTexCoord2fv((float *)&pTexCoords[3*i]);
                else
                    glTexCoord2fv((float *)&pTexCoords[VtxIdx0]);
            }
            glVertex3fv((float *)&pVertices[VtxIdx0]);

            if (pNormals)
                glNormal3fv((float *)&pNormals[VtxIdx1]);
            if (pTexCoords){
                if(m_UsingTexCoords)
                    glTexCoord2fv((float *)&pTexCoords[3*i + 1]);
                else
                    glTexCoord2fv((float *)&pTexCoords[VtxIdx1]);
            }
```

```
        glVertex3fv((float *)&pVertices[VtxIdx1]);

        if (pNormals)
            glNormal3fv((float *)&pNormals[VtxIdx2]);
        if (pTexCoords){
            if(m_UsingTexCoords)
                glTexCoord2fv((float *)&pTexCoords[3*i + 2]);
            else
                glTexCoord2fv((float *)&pTexCoords[VtxIdx2]);
        }
        glVertex3fv((float *)&pVertices[VtxIdx2]);
    }

    if(m_NormalMode==NORMALPERFACE)
    {
        pNormals   = &this->m_VtxNormals[0];
        if (pNormals)
            glNormal3fv((float *)&pNormals[i]);


        if (pTexCoords) {
            if(m_UsingTexCoords)
                glTexCoord2fv((float *)&pTexCoords[3*i]);
            else
                glTexCoord2fv((float *)&pTexCoords[VtxIdx0]);
        }
        glVertex3fv((float *)&pVertices[VtxIdx0]);


        if (pTexCoords){
            if(m_UsingTexCoords)
                glTexCoord2fv((float *)&pTexCoords[3*i + 1]);
            else
                glTexCoord2fv((float *)&pTexCoords[VtxIdx1]);
        }
        glVertex3fv((float *)&pVertices[VtxIdx1]);


        if (pTexCoords){
            if(m_UsingTexCoords)
                glTexCoord2fv((float *)&pTexCoords[3*i + 2]);
            else
                glTexCoord2fv((float *)&pTexCoords[VtxIdx2]);
        }
        glVertex3fv((float *)&pVertices[VtxIdx2]);
    }

}
glEnd();
```

APÊNDICE L – Padrões de projetos utilizados

A seguir daremos um breve resumo sobre os padrões de projetos utilizados no desenvolvimento do editor. Esse resumo tem apenas como propósito guiar o leitor durante o entendimento deste projeto, mas nunca ser uma referência de padrões de projetos utilizados. Para uma maior referência procurar em livros específicos da área de engenharia de software (GAMMA, 1995).

A seguir temos os padrões de projetos utilizados:

L.1 Singleton

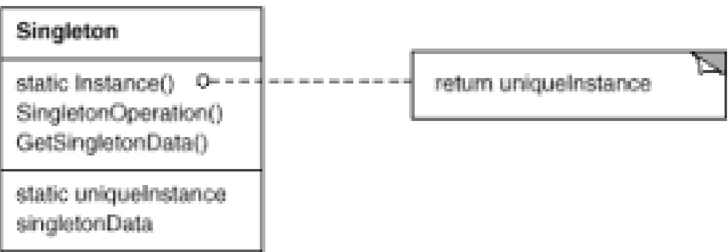
Este padrão tem como objetivo garantir que uma classe tenha apenas uma instância e prover um ponto de acesso global para ela.

L.1.1 Aplicabilidade

Usar o padrão Singleton quando:

- Tem que haver exatamente uma instância da classe, e ela tem que ser acessível pelo cliente de pontos bem conhecidos.
- Quando a instância é única e poderia ser extensível por herança, e o cliente deve ser capaz de usar uma instância estendida sem mudar seu código.

L.1.2 Estrutura



L.1.3 Participantes

- O Singleton
  - Define uma operação Instância que permite ao cliente acessar sua única instância. *Instance* é uma operação de classe (ou seja, uma função membro estática em C++).
  - Pode ser responsável por criar sua única instância.

L.2 Chain Of Responsibility

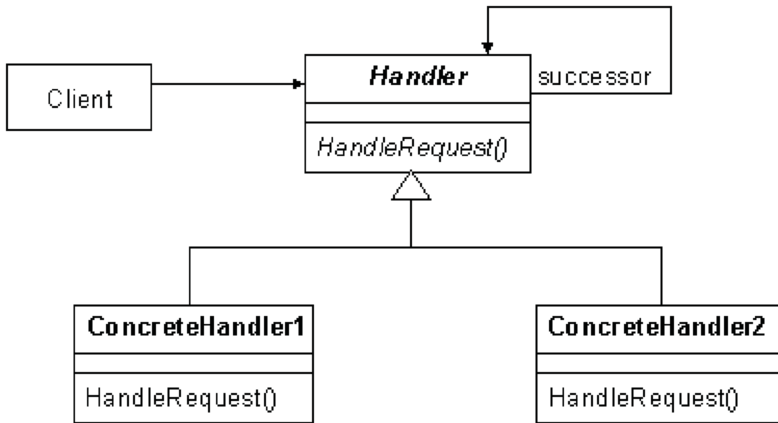
Evita o acoplamento do remetente de uma mensagem ao seu destinatário por dar a mais de um objeto a chance de tratar o pedido. Encadeia os objetos receptores e passa o pedido através da cadeia até um objeto tratá-lo.

L.2.1 Aplicabilidade

Usar Chain of Responsibility quando:

- Mais de um objeto pode tratar o pedido, e o responsável pelo tratamento não é sabido a priori. O responsável deveria ser atribuído automaticamente.
- Você quer enviar um pedido a um dos objetos sem especificar o receptor explicitamente.
- O conjunto de objetos que tratam a requisição poderiam ser especificados dinamicamente.

L.2.2 Estrutura



Uma estrutura típica de objetos deve se parecer com a figura abaixo:

L.2.3 Participantes

- Handler
  - define uma interface para tartar pedidos.
  - (opcional) implementa o link sucessor.
- ConcreteHandler
  - Trata o pedido pelo qual ele é responsável.
  - Pode acessar seu sucessor.
  - Se o ConcreteHandler pode tratar o pedido, ele o faz; caso contrário ele encaminha o pedido para o seu sucessor.
- Client
  - Inicializar um pedido em um objeto ConcreteHandler na cadeia de objetos.

L.3 Interpreter

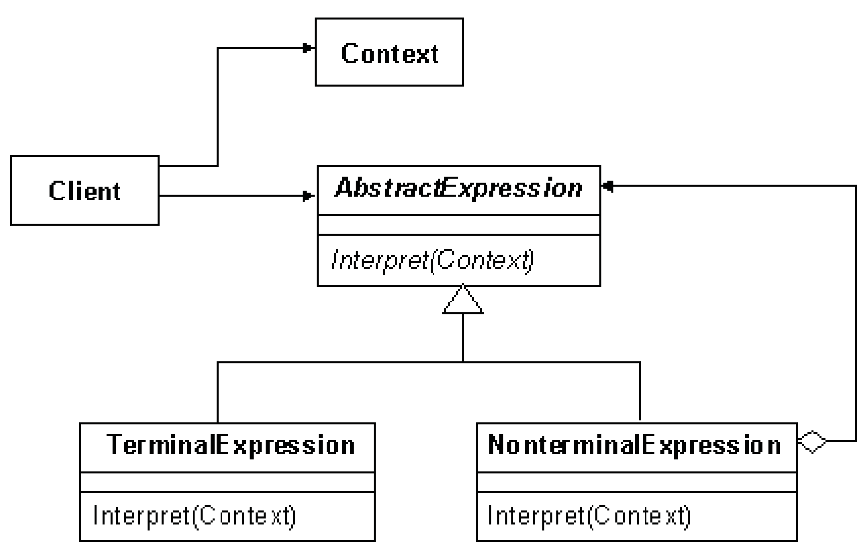
Dada uma linguagem, definir a representação da sua gramática em conjunto com um interpretador que usa a representação para interpretar sentenças na linguagem.

L.3.1 Aplicabilidade

Use o padrão *Interpreter* quando há uma linguagem a interpretar, e se puder representar uma instrução em uma linguagem como árvores de sintaxe abstratas. O padrão *Interpreter* trabalha melhor quando:

- A gramática é simples. Para gramáticas complexas, a hierarquia de classes para a gramática se torna grande e não gerenciável. Ferramentas, como geradores de parser, são a melhor alternativa nestes casos. Elas podem interpretar expressões sem construir árvores sintáticas abstratas, as quais podem salvar espaço e possivelmente tempo.
- Eficiência não é um elemento crítico. Os interpretadores mais eficientes tipicamente não foram implementados por meio de interpretação de árvores de parser diretamente. Por exemplo, expressões regulares são freqüentemente transformadas em maquinas de estados. Mas, mesmo assim, o tradutor pode ser implementado pelo padrão *Interpreter*, tal que o padrão ainda é aplicável.

L.3.2 Estrutura



L.3.3 Participantes

- AbstractExpression
  - Declara uma operação abstrata interpretada que é comum a todos os nós na árvore sintática abstrata.

- TerminalExpression
  - Implementa uma operação interpretada associada com símbolos terminais na gramática
  - Uma instância é requerida para todo símbolo terminal da seqüência.
- NonterminalExpression
  - Uma classe deste tipo é necessária para cada regra R ::= R_1 R_2 ... R_n na gramática.
  - Mantêm variáveis de instância do tipo *AbstractExpression* para cada um dos símbolos R_1 até R_n.
  - Implementa uma operação de interpretador para símbolos não terminais na gramática. *Interpreter*, tipicamente, chama a si mesmo recursivamente nas variáveis representando R_1 até R_n.
- Context
  - Contêm informação que é global ao interpretador.
- Client
  - Constrói (ou é dada) uma árvore sintática abstrata representando uma seqüência particular na linguagem que a gramática define. A árvore abstrata é montada das instâncias das classes *NonterminalExpression* e *TerminalExpression*.
  - Invoca a operação *Interpret*.

L.4 Flyweight

Usar compartilhamento para suportar um grande número de altamente granularizados eficientemente.

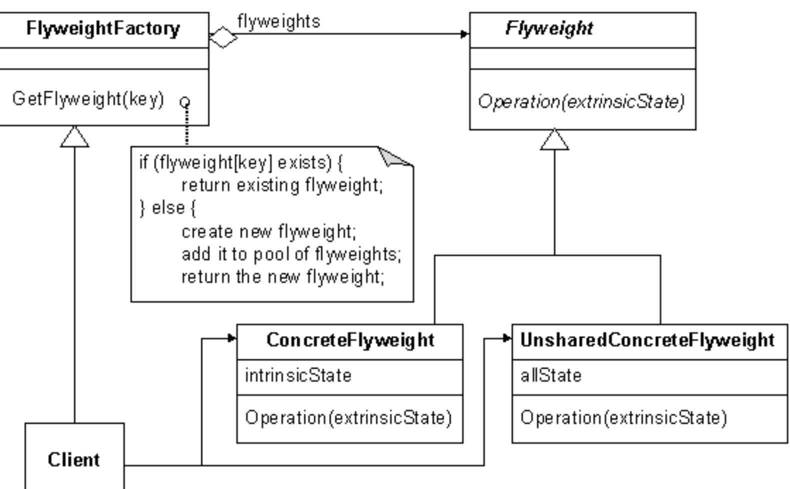
L.4.1 Aplicabilidade

Aplicar o padrão Flyweight todos os ítens a seguir forem verdade:

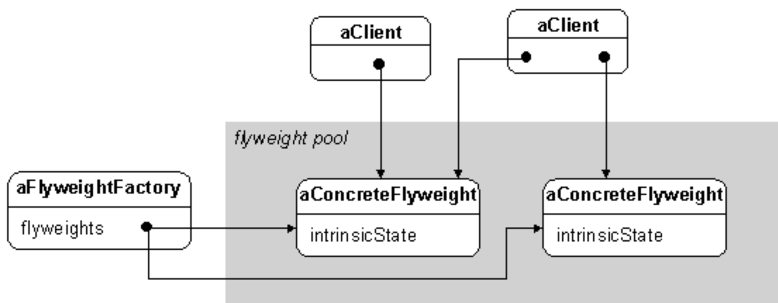
- Uma aplicação usar um número muito grande de objetos
- Custos de armazenamento são altos por causa da quantidade delicada de objetos.

- A maior parte do estado do objeto pode ser definida como extrínseca.
- Muitos grupos de objetos deveriam ser substituídos por relativamente poucos objetos desde que seu estado extrínseco seja removido.
- A aplicação não depende da identidade do objeto.

L.4.2 Estrutura



O diagrama a seguir mostra como os objetos flyweight são compartilhados.



L.4.3 Participantes

- Flyweight
  - Declara uma interface através da qual os flyweights podem receber e agir sobre um estado extrínseco.
- ConcreteFlyweight
  - Implementa a interface do Flyweight a adiciona local para o estado intrínseco, se houver algum.
  - Qualquer estado que um objeto *ConcreteFlyweight* guarda tem que ser intrínseco.
- UnsharedConcreteFlyweight
  - Nem todas as classes do Flyweight precisam ser compartilhadas. A interface Flyweight habilita compartilhamento, mas não o força.
  - É comum para objetos do tipo *UnsharedConcreteFlyweight* ter objetos do tipo *ConcreteFlyweight* como filhos em algum nível da estrutura de objetos flyweight.
- FlyweightFactory
  - Cria e gerencia objetos flyweight.
  - Assegura que os flyweights são compartilhados apropriadamente. Quando um cliente pede um flyweight, o objeto *FlyweightFactory* fornece uma instância existente ou cria uma, se nenhuma existir.
- Client
  - Mantém uma referência para o(s) flyweight(s).
  - Computa ou armazena o estado extrínseco do flyweight(s).

L.5 Composite

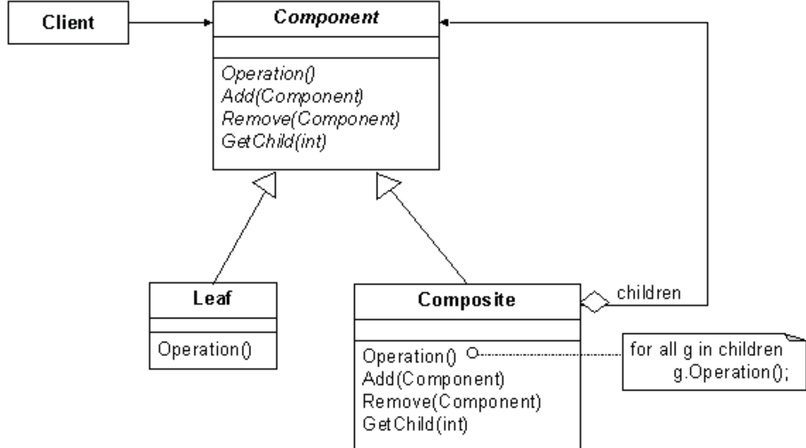
Compor objetos em estruturas de árvore para representar hierarquias do tipo parte-todo. *Composite* permite que clientes tratem objetos individuais e composições de objetos uniformemente.

L.5.1 Aplicabilidade

Aplicável quando:

- Deseja-se representar hierarquias parte-todo de objetos.
- Deseja-se que os clientes sejam capazes de ignorar as diferenças entre composições de objetos e objetos individuais.

L.5.2 Estrutura



L.5.3 Participantes

- *Component*
  - –Declara a interface para os objetos da composição.
  - –Implementa um comportamento comum à todas as classes.
  - –Declara uma interface para acessar e gerenciar seus nós filhos.
  - –Opcionalmente oferece uma interface para acessar o pai do componente na estrutura recursiva.
- *Leaf*
  - –Representa objetos folha na composição.
  - –Define o comportamento para objetos primitivos na composição.

- *Composite*
  - –Define comportamento para componentes agregados que têm filhos.
  - –Guarda componentes filhos.
  - –Implementa operações relacionadas ao filho na interface *Component*.
- *Client*
  - –Manipula os objetos na composição através da interface *Component*.

APÊNDICE M – *Exemplo de código VRML gerado pelo Building Editor*

```
#VRML V2.0 utf8
#############################################################
#      This VRML World is generated by Building Editor      #
#         Building Editor is authored by Mozart Filho       #
#     And a product of Federal University of Pernambuco     #
#############################################################
Group {
    children [
    Shape {
        appearance Appearance {
            material DEF _DEFAULT_MATERIAL_ Material {
                diffuseColor 0.8 0.8 0.8
                specularColor 0 0 0
                emissiveColor 0 0 0
                transparency 0

            }
            texture DEF CASA9 ImageTexture {
                url "casa-9.jpg"
            }
        }
        geometry IndexedFaceSet {
            coord Coordinate {
                point [0 0 0, 0 0 5, 5 0 5, 5 0 0, 0 6 0, 0 6 5, 5 6 5, 5 6 0]
            }
            coordIndex [4, 0, 1, -1, 4, 1, 5, -1]
            texCoord TextureCoordinate {
                point [0 1, 0 0, 1 0, 0 1, 1 0, 1 1]
            }
            texCoordIndex [0, 1, 2, -1, 3, 4, 5, -1]
        }
    }
    Shape {
        appearance Appearance {
            material USE _DEFAULT_MATERIAL_
            texture DEF CASA11 ImageTexture {
                url "casa-11.JPG"
            }
        }
        geometry IndexedFaceSet {
```

```
            coord Coordinate {
                point [0 0 0, 0 0 5, 5 0 5, 5 0 0, 0 6 0, 0 6 5, 5 6 5, 5 6 0]
            }
            coordIndex [5, 1, 2, -1, 5, 2, 6, -1]
            texCoord TextureCoordinate {
                point [0 1, 0 0, 1 0, 0 1, 1 0, 1 1]
            }
            texCoordIndex [0, 1, 2, -1, 3, 4, 5, -1]
        }
    }
    Shape {
        appearance Appearance {
            material USE _DEFAULT_MATERIAL_
        }
        geometry IndexedFaceSet {
            coord Coordinate {
                point [0 0 0, 0 0 5, 5 0 5, 5 0 0, 0 6 0, 0 6 5, 5 6 5, 5 6 0]
            }
            coordIndex [6, 2, 3, -1, 6, 3, 7, -1]

        }
    }
    Shape {
        appearance Appearance {
            material USE _DEFAULT_MATERIAL_
            texture USE CASA11
        }
        geometry IndexedFaceSet {
            coord Coordinate {
                point [0 0 0, 0 0 5, 5 0 5, 5 0 0, 0 6 0, 0 6 5, 5 6 5, 5 6 0]
            }
            coordIndex [7, 3, 0, -1, 7, 0, 4, -1]
            texCoord TextureCoordinate {
                point [0 1, 0 0, 1 0, 0 1, 1 0, 1 1]
            }
            texCoordIndex [0, 1, 2, -1, 3, 4, 5, -1]
        }
    }
    Shape {
        appearance Appearance {
            material USE _DEFAULT_MATERIAL_
        }
        geometry IndexedFaceSet {
            coord Coordinate {
                point [0 0 0, 0 0 5, 5 0 5, 5 0 0, 0 6 0, 0 6 5, 5 6 5, 5 6 0]
            }
            coordIndex [4, 5, 6, -1, 4, 6, 7, -1]

        }
    }
    ]
}
```

APÊNDICE N – Gerando a textura do terreno

```
void CTerrainTexture::Load() {
    pTexture = (unsigned char *)malloc(TEXTURE_SIZE*TEXTURE_SIZE*3);
    m_Width = TEXTURE_SIZE;
    m_Height = TEXTURE_SIZE;
    m_BPP = 3;
    m_AmbientLight = 0.3f;
    unsigned char *pTexWalk = pTexture;
    if ( !pTexture ) {
        return;
    }

    float len = sqrtf(SQR(LightX) + SQR(LightY) + SQR(LightZ));
    LightX /= len;
    LightY /= len;
    LightZ /= len;

    for ( UINT32 y = 0; y < TEXTURE_SIZE; y++ )
        for ( UINT32 x = 0; x < TEXTURE_SIZE; x++ )
        {
            #define TRI_SIZE (8)
            float shade=0.1f + (0.1f * rand())/RAND_MAX;

            UINT8 ApexZ = Landscape::m_pHeightMap->Height((x*MAP_SIZE)/TEXTURE_SIZE,
                                                          (y*MAP_SIZE)/TEXTURE_SIZE);
            float Ax, Ay, Az;
            float Bx, By, Bz;
            float Cx, Cy, Cz;

            if ( x < TEXTURE_SIZE-TRI_SIZE )
                Az = (float)Landscape::m_pHeightMap->Height((x+TRI_SIZE)*MAP_SIZE/TEXTURE_SIZE,
                                                            y*MAP_SIZE/TEXTURE_SIZE);
            else
                Az = 0;

            if ( y < TEXTURE_SIZE-TRI_SIZE )
                Bz = (float)Landscape::m_pHeightMap->Height(x*MAP_SIZE/TEXTURE_SIZE,
                                                            (y+TRI_SIZE)*MAP_SIZE/TEXTURE_SIZE);
            else
                Bz = 0;

            Ax = TRI_SIZE;
            Ay = 0;
            Az = Az - ApexZ;
```

```
            Bx = 0;
            By = TRI_SIZE;
            Bz = Bz - ApexZ;

            Cx = (Ay * Bz) - (Az * By);
            Cy = (Az * Bx) - (Ax * Bz);
            Cz = (Ax * By) - (Ay * Bx);

            len = sqrtf(SQR(Cx) + SQR(Cy) + SQR(Cz));
            Cx /= len;
            Cy /= len;
            Cz /= len;

            shade += (0.8f - m_AmbientLight) * (Cx * LightX + Cy * LightY + Cz * LightZ) +
                        m_AmbientLight ;

            float heightPct = (ApexZ & 0x0F) / 16.0f;

            ApexZ >>= 4;
            if ( shade > 1 )  shade = 1;
            if ( shade < 0 ) shade = 0;
            if ( ApexZ >= 0 )
            {
                *(pTexWalk++) = (UINT8)(shade * (UINT8)((heightPct * aLandColors[ApexZ][0]) +
                                ((1.0f-heightPct) * aLandColors[ApexZ-1][0])));
                *(pTexWalk++) = (UINT8)(shade * (UINT8)((heightPct * aLandColors[ApexZ][1]) +
                                ((1.0f-heightPct) * aLandColors[ApexZ-1][1])));
                *(pTexWalk++) = (UINT8)(shade * (UINT8)((heightPct * aLandColors[ApexZ][2]) +
                                ((1.0f-heightPct) * aLandColors[ApexZ-1][2])));
            }
        }
}
```

APÊNDICE O – Contribuições e Artigos publicados

O autor está desenvolvendo pesquisa na área de realidade virtual desde 1998. Quando começo a estudar VRML sobre a supervisão do professor Alejandro Frery, orientador desta tese. Desenvolveu primeiramente ambientes simples baseados em tutoriais, entre os quais podemos destacar o logotipo do CESAR (Centro de Estudos e Sistemas Avançados do Recife), e uma casa completamente mobiliada e detalhadamete modelada a mão, ambos disponíveis em Araújo Filho (2001).

Posteriormente, foi feito um estudo de caso sobre a utilização de avatares em ambientes 3D, no qual, o autor desenvolveu três avatares que funcionam como guias no mundo e mostram o potencial interativo deste tipo de aplicação junto ao usuário de ambientes 3D. Neste trabalho temos o *avatar formiga*, responsável por um tour terrestre no ambiente, o *avatar abelha* responsável por um tour aéreo ou vista superior, e o *avatar "pulador"* que nada mais é que um misto dos casos anteriores, um avatar que pode dar saltos dezenas de metros durante uma excursão pelo ambiente. Estes avatares foram usados posteriormente no projeto do *Campus Virtual*, vide (FRERY, 2003), projeto onde todo o campus da Universidade Federal de Pernambuco foi modelado, este trabalho de grande porte contou com muitos alunos, dos quais um era o autor.

Por volta do ano 2000, o autor participou do desenvolvimento de um simulador de alagamento. Programa feito em VRML e Java (Sun Microsystems, Inc., 2003) no qual o autor participou com um dos ítens do projeto, um editor de pontos de vista, item inseria ou excluía pontos de vista navegáveis em um ambiente VRML previamente existente.

Um outro projeto em que o autor estava a frente, no ano 2001, foi um estudo de caso de um ambiente multi-usuário VRML no qual o autor estudou questões arquiteturais deste tipo de aplicação, bem como fez extensões para o *DeepMatrix*, ambiente multi-usuário desenvolvido pela Blaxxun (2003) na época. Este ambiente colaborativo é totalmente desenvolvido em Java, juntamente com VRML, e se baseia em sockets para efetuar troca de mensagens remotamente. Entre os ítens que compreendem essa troca de mensagem podemos citar objetos que são animados distribuidamente, como os próprios avatares

ou uma luz que acende em uma sala, posição de avatares no mundo e interpolação de posição de avatares, o que permite a neutralização de eventuais efeitos quadro a quadro decorrentes da perda ou atraso de pacotes na rede.

No ano seguinte o Centro de Informática da Universidade Federal de Pernambuco adquiriu um *Head Mounted Display*, um *Tracker* e a ferramenta *World-Up* (Sense 8, 2003), o que permitiu ao aluno desenvolver ambientes e imersivos, bem como, estudar os mecanismos de interação entre esses ambientes e o *Tracker*. No desenrolar desta pesquisa foram feitos relatórios sobre o processo e a limitação de importação de certos ambientes e primitivas previamente construídas pelo centro para o World-Up. Este relatório foi feito baseado no processo de importação do Recife Virtual (FRERY, 2003), que retratava em VRML uma área da cidade do Recife denominada *Recife Antigo*.

Neste mesmo ano, o aluno fez um estudo comparativo de linguagens descritivas 3D, no qual analisou e comparou as linguagens Flatland, Java 3D e VRML. Neste estudo foram analisados os pontos fortes e fracos destas linguagens, bem como, as situações para qual elas são mais indicadas.

Chegamos, então, a esta tese de mestrado a qual traz consigo o estudo de um tipo de aplicação muito comum na área de Realidade Virtual, o editor de cenários. Neste projeto, analisamos todos os méritos de construção deste tipo de aplicação, bem como levantamos situações de interesse existentes em todas as fases do processo de construção deste tipo de aplicação. Mas antes de chegarmos ao resultado muito estudo e trabalho foi necessário, como a aprendizagem de MFC e o curso de OpenGL que foi ministrado na disciplina de Realidade Virtual e Multimídia, vide (FRERY; KELNER, 2002).

Para finalizarmos, durante o decorrer dos cursos de graduação e mestrado em Ciência da Computação foram publicados os seguintes artigos em congressos nacionais e internacionais:

- **(FRERY, 2000):**
  - Neste artigo é descrita uma ferramenta que permite visualizar o padrão de alagamento de áreas costeiras, incluindo um simulador de marés.
- **(KELNER, 2001):**
  - Este trabalho descreve uma aplicação de realidade virtual *desktop* que permite o estudo de quais áreas serão alagadas, dado o modelo numérico do terreno de interesse. A ferramenta oferece uma interface amigável e mecanismos de interação e de descoberta do conhecimento.
- **(KELNER, 2002):**

–Este artigo faz um apanhado geral dos principais problemas e soluções para a modelagem de mundos urbanos. O trabalho conclui com uma apresentação dos projetos desenvolvidos no Centro de Informática da UFPE.

- **(SANTOS, 2003):**

  –Este artigo propõe uma nova técnica de classificação de imagens, baseada em funções de pertinência nebulosas. Técnica esta que teve questões de desempenho e grau de acerto da classificação investigados no decorrer do trabalho.

- **(FRERY, 1999):**

  –O aluno participou do projeto ConVIRA, que tinha como objetivo principal fazer a cooperação entre Brasil e Alemanha no desenvolvimento de ambientes 3D com mecanismos de navegação conceituais. Neste projeto, o autor participou ativamente fazendo uma viagem de um mês de duração ao *GMD institute* na Alemanha em Fevereiro de 2001.

- **(ARAÚJO FILHO; FRERY, 2000, 1999; ARAÚJO FILHO; COSTA; PESSOA, 1998; PESSOA; ARAÚJO FILHO; COSTA, 1998; COSTA; ARAÚJO FILHO; PESSOA, 1998):**

  –O Estes artigos no congresso de Iniciação Científica da UFPE, contam um pouco da história da pesquisa enquanto o autor ainda se encontrava na graduação do curso de Ciência da Computação.

- **(ARAÚJO FILHO; ALMEIDA; TIMES, 2000; ALMEIDA; ARAÚJO FILHO; TIMES, 2000):**

  –Estes resumos contam dois resultados correspondentes ao desenvolvimento de uma aplicação de páginas amarelas para Web. Entre os resultados podemos citar o desenvolvimento de uma arquitetura para este tipo de aplicação, assim como uma amostragem comparativa dos vários algoritmos propostos para busca e armazenagem dos dados neste tipo de aplicação.