UNIVERSIDADE FEDERAL DE PELOTAS

INSTITUTO DE FÍSICA E MATEMÁTICA

DEPARTAMENTO DE INFORMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GERAÇÃO PROCEDURAL E VISUALIZAÇÃO DE CIDADES PSEUDO-INFINITAS

DIEGO DE OLIVEIRA DUARTE

Pelotas, 2006

Diego de Oliveira Duarte

GERAÇÃO PROCEDURAL E VISUALIZAÇÃO DE CIDADES PSEUDO-INFINITAS

Trabalho acadêmico apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação

Orientador: Prof. Dr. João Luiz Dihl Comba
UFRGS

Co-Orientadora: Prof.ª Eliane da Silva Alcoforado Diniz
UFPel

Pelotas, 2006

BANCA EXAMINADORA:

___

Prof. Dr. Lucas Ferrari de Oliveira

___

Prof. Dr. Paulo Roberto Gomes Luzzardi

RESUMO

A Computação Gráfica tem sido aplicada em várias áreas, tanto na pesquisa científica como na indústria do entretenimento. As necessidades de modelos de cidades em três dimensões - 3D estão expandindo-se rapidamente em áreas relacionadas ao entretenimento, como em filmes e jogos eletrônicos, ou também em planejamento urbano. Simulações de realidade virtual, vídeo games e animações computadorizadas tornaram-se muito populares. A necessidade de gerar cada vez mais conteúdo como modelos tridimensionais cenários mais detalhados e vastos levou a adotar abordagens algorítmicas procedurais ao invés de modelagem manual. A pesquisa em modelagem procedural tem produzido soluções para geração de geometrias, modelos de objetos naturais e fenômenos, como terrenos, fogo, explosões, água, chuva, nuvens, plantas, etc. Este trabalho tem por objetivo mostrar um estudo das técnicas procedurais de geração de conteúdo. Mostrar os fundamentos utilizados como base para o *design* de um *engine* de geração procedural de uma cidade virtual pseudo-infinita. Faz parte deste trabalho também o design e implementação de um modelo primitivo para geração de uma cidade virtual pseudo-infinita.

Palavras chave: Computação gráfica. Realidade virtual. Geração procedural. Simulação. L-*system*. Modelagem. Jogos.

ABSTRACT

Computer Graphics has been applied in several areas, like scientific research and entertainment industry. The needs of 3D city models are expanding in areas related with entertainment, such movies and games, also urban planning. Simulations of virtual reality, video games and computer animations become much popular, the need of generate more content, tridimensional models, detailed and extensive landscapes take to follow procedural algorithmically approaches instead of manual modeling. The research in procedural modeling has produce solutions for geometries generation, natural objects and phenomena models. Like terrains, fire, explosions, water, rain, clouds, plants, etc. This work focus in the study of procedural techniques in content generation, considering all aspects. Showing the bases that are useful to design an engine of a procedurally generated virtual pseudo-infinite city. Also is part of the work the design and implementation of a primitive model of pseudo infinite city procedurally generated.

Keywords: Computer graphics. Virtual reality. Procedural generation, L-system, modeling, games.

LISTA DE FIGURAS

Figura 2.1: Volume de visualização geral do *OpenGL*.. 18
Figura 2.2: Volume de visualização *Frustum*... 18
Figura 2.3: Parâmetros da *gluLookAt*.. 19
Figura 2.4: Modelo de cidade virtual com *frustum view*...................................... 20
Figura 2.5: Estrutura geral da *OpenGL*... 22
Figura 2.6: *Pipeline* de visualização 2D.. 24
Figura 2.7: *Pipeline* de visualização 3D.. 24
Figura 2.8: *Pipeline OpenGL*.. 25
Figura 3.1: Jogo .kkrieger………………………………………………………….... 32
Figura 3.2: SpeedTreeCAD Interface……………………………………………… 35
Figura 3.3: *Unreal Engine 3*………………………………………………………… 35
Figura 3.4: Exemplo de D0L-*system*.. 39
Figura 3.5: A produção para a curva de Koch.. 41
Figura 3.6: Produções da curva de Koch... 42
Figura 3.7: Ilha quadrática de Koch derivações de tamanho n =0, 1, 2 e 3......... 43
respectivamente, com ângulo b igual a 90º
Figura 3.8: Exemplo de utilização dos símbolos [e].. 44
Figura 3.9: Exemplo de utilização dos símbolos \ e /.. 44
Figura 3.10: L-*system* 3D parametrizado.. 45
Figura 3.11: Funções aplicadas a um sucessor... 46
Figura 4.1: Escopo de uma forma. O ponto P, juntamente com os eixos X, Y e.. 55
Z e o tamanho S definem uma caixa quadrangular no espaço contendo a forma.
Figura 4.2: Um modelo primitivo formado de três primitivas de formas............... 56
Figura 4.3: Divisão de uma fachada sobre eixo Y.. 57
Figura 4.4 Subdivisão da fachada sobre eixo X... 58
Figura 4.5: Subdivisões em múltiplos eixos e aplicação de texturas................... 58
Figura 4.6: Disposição final da fachada... 59
Figura 4.7: Modelagem de telhado.. 60
Figura 5.1: Plano de chão.. 62
Figura 5.2: Disposição inicial do cenário, 1ª iteração.. 63

Figura 5.3: Cenário após 2ª iteração.. 64
Figura 5.4: Cenário após 7ª iteração.. 64
Figura 5.5: Formato de desenho da primitiva *GL_QUADS*.................................. 65
Figura 5.6: Exclusão de nodo da lista de quadras... 67
Figura 5.7: Cidade virtual.. 68
Figura 5.8: Cidade virtual.. 68
Figura 5.9: Texturas.. 69

LISTA DE TABELAS

Tabela 2.1: Bibliotecas da *OpenGL*.. 22
Tabela 5.1: Matriz de coordenadas de uma quadra.. 66

LISTA DE ABREVIATURAS E SIGLAS

API - *Application programming interface* - Interface de Programação de Aplicativos

ARB - *Architecture Review Board*

CAD - *Computer Aided Desing* - Desenho Auxiliado por Computador

CD - *Compact Disc*

CPU - *Central Processing Unit* - Unidade Central de Processamento

DVD - *Digital Video Disc*

GFSR - *Generalised Feedback Shift Register*

Ghz - *Gigahertz*

GL - *Graphics Library*

GLU - *Graphics Utility Library*

GLUT - GL *Utility Toolkit*

GLX - *OpenGL Extension to the X Window System*

GPU - *Graphics Processing Unit* - Unidade de Processamento Gráfico

HD-DVD - *High Density Digital Versatile Disc* - Disco Digital Versátil de Alta Densidade

IA - Inteligência Artificial

KB - *Kilobyte*

L-System - Lindenmayer *system*

MB - *Megabyte*

MIDI - *Musical Instrument Digital Interface* - Interface Digital para Instrumentos Musicais

OpenGL - *Open Graphics Library*

RAM - *Random Access Memory* - Memória de acesso aleatório

SDK - *Software Development Kit - K*it de Desenvolvimento de Software

TGFSR - *Two-tap Generalised Feedback Shift Register*

WGL - W*indows Graphics Library*

2D - Duas Dimensões

3D - Três Dimensões

SUMÁRIO

1 INTRODUÇÃO 14

2 FUNDAMENTOS PARA GERAÇÃO PROCEDURAL DE CIDADES VIRTUAIS............... 17

2.1 Volume de Visualização.. 17

2.2 *View Frustum Filling*……………………………………………………….. 19

2.3 *Rendering*………………………………………………………………… 20

2.4 *OpenGL*………………………………………………………………… 21

2.4.1 Extensões.. 23

2.5 *Pipeline* Gráfico... 24

3 GERAÇÃO PROCEDURAL 26

3.1 Paradigmas de construção de cidades virtuais.................................... 28

3.2 Trabalhos relacionados.. 28

3.2.1 Geração procedural em jogos... 29

3.2.2 Exemplos de utilização da Geração Procedural.................................... 30

3.2.2.1 Jogo .kkrieger... 30

3.2.2.1.1 Tecnologia... 31

3.2.2.2 Suíte para geração de árvores e plantas *Speedtree*....................... 32

3.2.2.2.1 Componentes…..…………………………………………………... 32

3.2.2.2.1.1 *SpeedTreeRT*………………………………………………… 33

3.2.2.2.1.2 *SpeedTreeCAD*…...……………….…………….…………… 33

3.2.2.2.2 *SpeedTree* em jogos……………………………………………... 33

3.2.2.2.3 *SpeedTree* em outras aplicações... 34

3.3 L-*Systems*.. 36

3.3.1 Origem... 36

3.3.2 Estrutura.. 37

3.3.3 D0L-*system*.. 38

3.3.4 Fractais e a interpretação gráfica das *strings*..................................... 39

3.3.5 *Turtle graphics* e L-*Systems*………………………………………….. 40

3.3.6 L-*systems* estendidos - criando um mapa de ruas.............................. 45

3.4 Geração de números pseudo aleatórios... 49

3.4.1 Geradores congruentes lineares... 49

3.4.2 Mersenne twister... 50

3.5 *Hashing*.. 50

4 GERAÇÃO DE PRÉDIOS.. 52

4.1 Geração dos planos de chão.. 52

4.2 Gramáticas de formas.. 52

4.2.1 Formas... 53

4.2.2 Produções... 54

4.2.3 Notação.. 55

4.2.4 Regras de escopo.. 55

4.3 Construção de fachadas... 56

4.3.1 Regras para subdivisão das formas.. 56

4.4 Modelos de telhado.. 59

5 ESTUDO DE CASO - FERRAMENTA PARA GERAÇÃO PROCEDURAL DE CIDADE PSEUDO-INFINITA 61

5.1 Plano de chão.. 62

5.2 Estruturas de dados.. 64

5.2.1 Estruturas.. 65

5.2.2 Matriz de coordenadas de uma quadra.. 65

5.3 Controle do tamanho do cenário... 66

5.4 Geração das construções.. 67

6 CONCLUSÃO.. 70

7 REFERÊNCIAS BIBLIOGRÁFICAS... 72

APÊNDICES... 77

1 INTRODUÇÃO

A geração de modelos computacionais de entidades geométricas é estudada no campo da Computação Gráfica. Cidades virtuais aparecem em diversas aplicações gráficas. Neste trabalho iremos nos concentrar numa classe específica de cidades virtuais chamada de cidades pseudo-infinitas, com prédios de geometria variadas que são gerados de acordo com a demanda de visualização.

Cidades pseudo-infinitas são modelos de cidades que dão a impressão de um ambiente com um horizonte praticamente infinito o qual não se consegue visualizar seu fim.

Neste trabalho será realizado um estudo abrangendo alguns dos conceitos necessários para a criação de um sistema de geração procedural e visualização de cidades virtuais pseudo-infinitas. Uma ferramenta para geração precedural de cidades pseudo-infinitas foi criada aplicando parte dos conceitos estudados no trabalho. Uma abordagem procedural configurada através de diversos parâmetros é usada em conjunto com diferentes classes de algoritmos para gerar uma variedade de prédios e ruas que simulam a aparência de uma cidade. Para incrementar a diversidade, prédios individuais são gerados no ambiente conforme são encontrados pelo usuário, resultando numa cidade que expande sobre demanda. Uma cidade virtual gerada pode ser explorada livremente através de uma ferramenta de visualização com interação 3D, gerando imagens sobre a perspectiva do observador.A Computação Gráfica tem sido aplicada em várias áreas, tanto na pesquisa científica como na indústria do entretenimento. As necessidades de modelos de cidades 3D estão expandindo-se rapidamente em áreas relacionadas ao entretenimento, como em filmes e jogos eletrônicos, ou também em planejamento urbano.

Enquanto os cenários dos jogos de computador ganham complexidade com cada geração de *hardware* e *software*, fica cada vez mais difícil para os artistas e

desenvolvedores criar estes mundos ricos em detalhes. A geração procedural é uma solução que pode simplificar estas tarefas.

No planejamento urbano várias aplicações podem fazer uso de cidades virtuais, tais como: desenvolvimento regional, turismo virtual, prevenção de inundações e outros. Várias aplicações discutem o uso de cidades virtuais para melhor tratar estes problemas de planejamento urbano.

Este trabalho irá colocar em prática vários conceitos aprendidos no curso, como linguagens formais para criar gramáticas de geração de padrões de ruas, criação de formas para as construções e padrões para fachadas. Também, estrutura de dados e algoritmos para geração e representação dos grandes conjuntos de prédios a serem gerados. Visando principalmente os conhecimentos de computação gráfica para o controle de uma câmera 3D utilizada para navegar pela cidade virtual, incluindo também o processo de renderização o qual conta com demais efeitos a serem utilizados na visualização final do modelo.

Simulações de realidade virtual, jogos e animações computadorizadas são muito populares. Tais aplicações demandam cada vez mais geração de conteúdo e modelos tridimensionais assim como cenários mais detalhados e vastos. Desta forma um fator de motivação para adotar abordagens algorítmicas procedurais ao invés de modelagem manual.

A pesquisa em modelagem procedural é utilizada também para produzir soluções para geração de modelos de objetos naturais e fenômenos, como terrenos, fogo, explosões, água, chuva, nuvens, plantas, etc.

Geração procedural de objetos criados pelo homem é de particular relevância. Com as possibilidades oferecidas pelos novos hardwares gráficos que continuamente estão em desenvolvimento, a indústria do entretenimento está sempre sob pressão para produzir mais conteúdo em seus produtos, porém deve manter o tempo e custo de produção. Companhias de cinema usam sets virtuais com muita freqüência. Profissionais de planejamento urbano e arquitetos necessitam também de ferramentas para visualização de suas idéias, e ferramentas para mostrar suas idéias aos seus alunos ou clientes.

Este trabalho será organizado da seguinte forma:

No capítulo 2, são apresentados conceitos fundamentais de computação gráfica. Também uma breve descrição do funcionamento da Interface de Programação de Aplicativos - API gráfica *Open Graphics Library* - *OpenGL*. Que servirão de base para entendimento do restante do trabalho.

No capítulo 3, é abordado o tema principal deste trabalho, a geração procedural. São mostrados seus paradigmas, e também técnicas utilizadas conjuntamente com os algoritmos procedurais.

O capítulo 4 mostra técnicas para síntese de prédios, desde a geração do seu plano de chão até aplicação de textura.

No capítulo 5, é mostrado um estudo de caso para um modelo primitivo de cidade virtual pseudo-infinita. Contém a descrição da implementação do projeto.

2 FUNDAMENTOS PARA GERAÇÃO PROCEDURAL DE CIDADES VIRTUAIS

A geração procedural abrange diversas áreas de conhecimento. Porém o desenvolvimento de projeto desta natureza passa obrigatoriamente por alguns conceitos fundamentais da computação gráfica e de especificidades da API gráfica, neste caso a *OpenGL*.

Cidades virtuais pseudo-infinitas são ambientes com aspecto de cidade que possuem visual de aparência infinita, ou seja, ligada diretamente aos conceitos do volume de visualização. Que delimitam a área visível do observador, a qual pode ser utilizada como estrutura de controle no preenchimento da tela com as geometrias.

Este capítulo apresenta conceitos os quais servem de base para o desenvolvimento de qualquer sistema de geração procedural de cidades, baseados em *OpenGL*.Tais como manuseio da câmera, volume de visualização, bem como utilização das suas bibliotecas e extensões para enriquecer a cena em detalhes. E também noções do funcionamento do *pipeline* gráfico no desenha de uma cena.

2.1 Volume de Visualização

O volume de visualização delimita o espaço tridimensional e identifica quais objetos irão ser visualizados na janela. A partir dos limites da janela de visualização é definido o volume de visualização, a forma e volume dependem do tipo de projeção utilizado na visualização. Na *OpenGL* utiliza-se uma analogia comparando visualização 3D com o ato de tirar fotografias com uma câmera (Fig. 2.1).

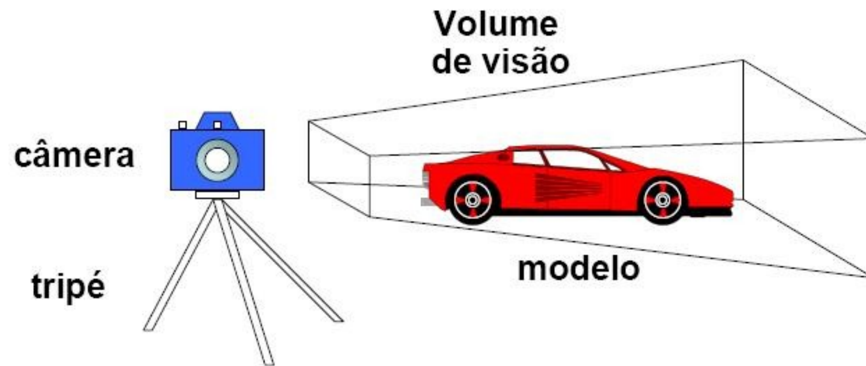


Figura 2.1 - Volume de visualização geral da *OpenGL*.
Fonte: ESPERANÇA, 2005, p.11.

No trabalho em questão para definir o volume de visualização será utilizada uma projeção em perspectiva, que define seu volume na forma de tronco de pirâmide (Fig. 2.2). Na OpenGL é especificado pela função:

glFrustum(left, rigth, bottom, top, near, far);

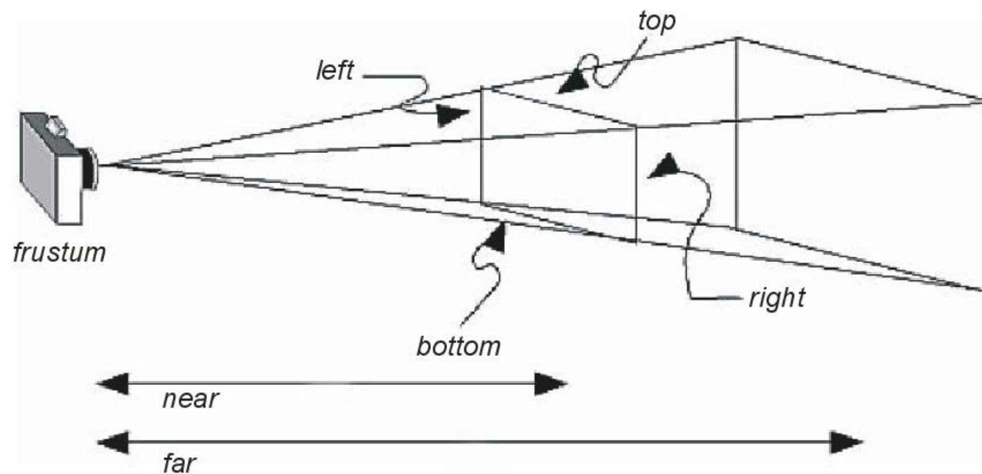


Figura 2.2 - Volume de visualização *Frustum*.
Fonte: ESPERANÇA, 2005, p.20.

Importante também mostrar a função que controla a câmera, *gluLookAt*, que possui a seguinte definição: *gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble aimx, GLdouble aimy,GLdouble aimz, GLdouble upx, GLdouble upy, GLdouble upz);*

Sendo

- *eye*: ponto onde a câmera será localizada nos eixos x, y e z

- *aim*: ponto para onde a câmera está apontada
- *up*: vetor que dá a direção para cima da câmera (inclinação)

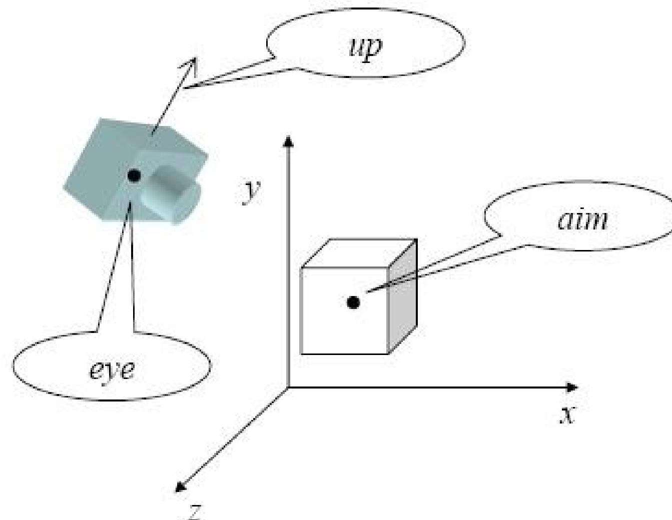


Figura 2.3 - Parâmetros da gluLookAt

Fonte: ESPERANÇA, 2005, p.18.

2.2 *View Frustum Filling*

View frustum é a região do espaço no cenário modelado, que deve aparecer na tela, é o campo de visão da câmera. O formato desta região varia de acordo com o tipo de câmera simulada, mas normalmente é em formato de uma pirâmide retangular. Os planos que cortam a perpendicular do volume de visualização podem ser chamados de plano *near* e plano *far*. Os objetos que encontram-se antes do plano *near* e após o plano *far* não são desenhados.

View frustum filling tem por objetivo preencher a *view frustum* com geometria procedural. É utilizado o termo *frustum filling* para descrever a restrição da geração procedural às partes do cenário localizadas dentro do campo de visão da câmera. A área do *frustum filling* é definida antes da geração do cenário.

A Fig. 2.4 mostra um terreno divido em células quadriculares em uma grade de duas dimensões - 2D. Cada célula representando uma quadra ou rua, e possuem ligação as suas células vizinhas. A partir de sua de localização no espaço fornecem

entrada para os parâmetros dos algoritmos de geração procedural. O potencial de visualização de uma célula é determinado pelo ângulo entre a célula e a direção da câmera, assim como também à distância (GREUTER, 2003).

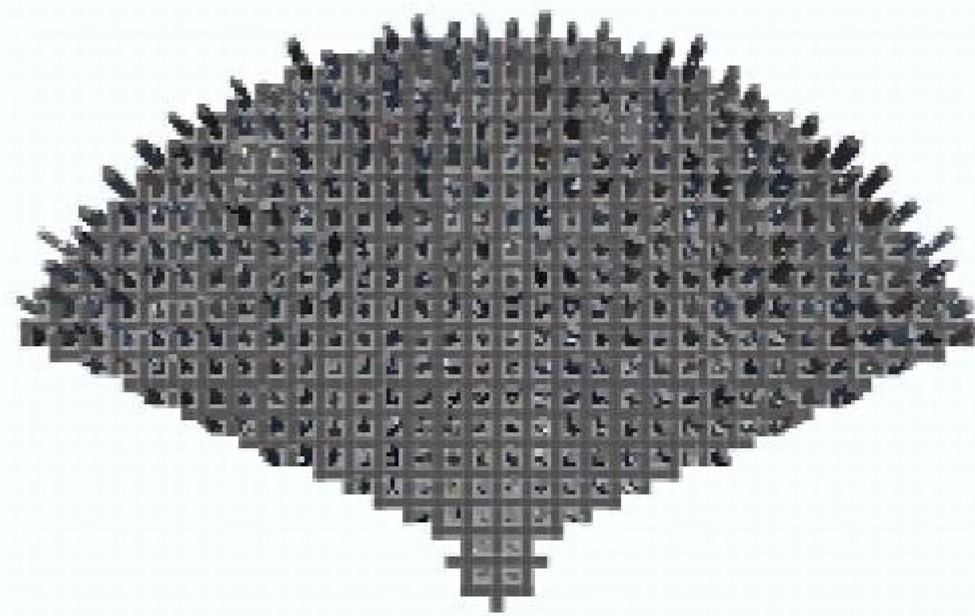


Figura 2.4 - Modelo de cidade virtual com *frustum view*

Fonte: GREUTER, 2003 p.88.

2.3 *Rendering*

É o processo de geração de uma imagem a partir de um modelo digital, através de um aplicativo de *software*. O modelo é uma descrição de objetos bi ou tridimensionais em uma linguagem definida ou estrutura de dados. Pode conter informações sobre sua geometria, *viewpoint*, textura e luz. A cena é uma imagem digital ou de gráficos rasterizados (RENDERING (COMPUTER..., 2006).

Rendering tem aplicações em jogos, simuladores, efeitos especiais de filmes e TV e visualização de *design*. Como produto, há uma grande variedade de *renderers*. Alguns integrados em pacotes de modelagem maiores, alguns em modo autônomo, outros em projetos *open-source* (RENDERING (COMPUTER..., 2006).

2.4 *OpenGL*

OpenGL é uma API gráfica[1]. É portável e rápida, portanto utilizada no desenvolvimento de jogos eletrônicos, como *Doom 3*, *Quake*, *Counter Strike*, entre outros. Porém *OpenGL* não é uma linguagem de programação, e sim uma API gráfica. Portanto quando um programa é feito em *OpenGL* significa que são feitas uma ou mais chamadas às suas funções (OPENGL - ..., 2006).

OpenGL é uma especificação que define uma API multiplataforma para escrever aplicações capazes de produzir gráficos computacionais 2D e 3D. A interface consiste em mais de 250 funções diferentes que podem ser utilizadas para desenhar cenas complexas a partir de primitivas simples. *OpenGL* é largamente usada em *Computer Aided Design* - CAD, realidade virtual, visualização científica, visualização de informações, simuladores e desenvolvimento de jogos (OPENGL - ..., 2006).

OpenGL tem por objetivo esconder a complexidade da interface de diferentes placas aceleradoras 3D, apresentando ao programador uma API uniforme. Também para esconder as capacidades das plataformas de *hardware*, exigindo que todas as implementações suportem inteiramente o conjunto de instruções do *OpenGL*, se necessário utilizando emulação por *software*.

As operações básicas são aceitar primitivas básicas como pontos, linhas e polígonos, e convertê-las em *pixels*, além de sistemas de iluminação, coloração, textura diversos outros recursos. Isto é feito pelo *pipeline* gráfico conhecido com a máquina de estados *OpenGL*. *OpenGL* é uma API procedural de baixo nível, exigindo que o programador descreva os passos exatos requeridos para renderizar a cena. Isto contrasta com API's descritivas, onde o programador somente necessita descrever a cena e deixa a biblioteca controlar os detalhes da renderização. O *design* em baixo nível do *OpenGL* exige que os programadores tenham um entendimento do *pipeline* gráfico, mas também fornece liberdade para implementar algoritmos novos.

---

[1] API gráfica desenvolvida em 1992 pela *Silicon Graphics*, uma das maiores empresas de computação gráfica e animação do mundo

Está disponível através de componentes ou toolkits prontos como as bibliotecas para C: GL, GLU e GLUT, que fornecem funções auxiliares (tab. 2.1). Funções de entrada e interface com o sistema são separadas (Fig. 2.5).

Tabela 2.1 - Bibliotecas da *OpenGL*.

Biblioteca	Funcionalidades
GL (*Graphics Library*)	• primitivas geométricas e gráficas • controle de atributos, etc
GLU (*Graphics Utility Library*)	• código *OpenGL* para objetos mais complexos
GLUT (*GL Utility Toolkit*)	• funções para comunicação com sistema de janelas

As funções *OpenGL* seguem um padrão em relação às suas notações. De forma geral são esquematizadas da seguinte maneira:

gl{nome da função}{número de variáveis}{tipo de variáveis}{forma vetorial}(arg 1, arg 2, ..., arg n};

Exemplo:

glVertex3f (0.0, 0.0, 0.0);

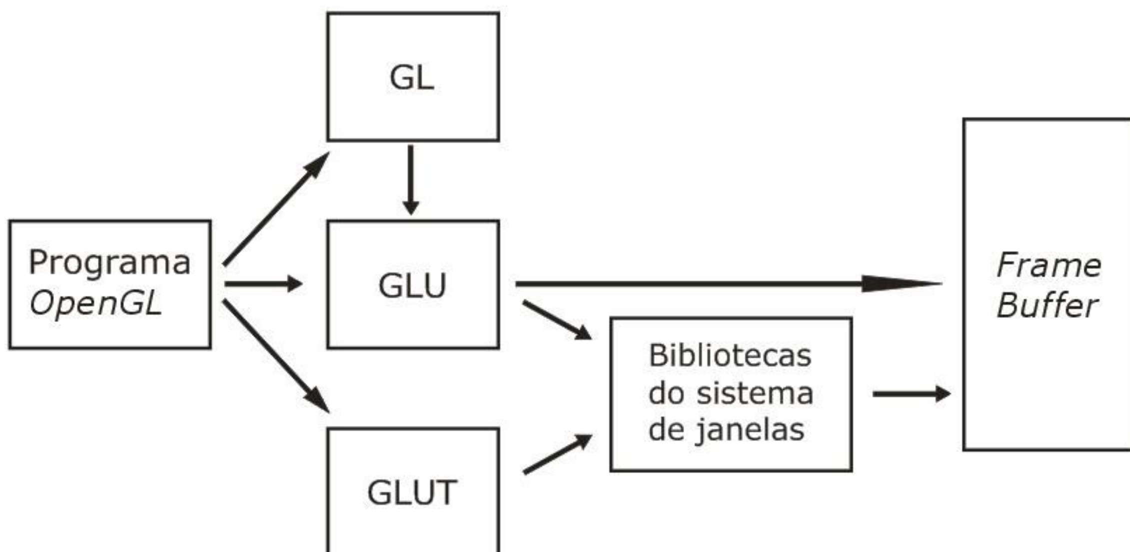


Figura 2.5 - Estrutura geral da *OpenGL*.

2.4.1 Extensões

O padrão *OpenGL* permite adicionar funcionalidades através de extensões como uma nova tecnologia. Extensões podem introduzir novas funções e novas constantes, e pode modificar ou remover restrições nas funções padrões. Muitas extensões, assim como extensões relacionadas com API's como GLU, GLX e WGL, tem sido definidas pelas empresas. O registro das extensões é mantido pela SGI e contém especificação de todas as extensões conhecidas. O registro também define convenções de nome, guias para criar novas extensões e escrever especificações de extensões adequadas, e demais documentações relacionadas (OPENGL - ..., 2006).

A *OpenGL Architecture Review Board* - ARB[2] define os testes necessários, aprova as especificações e define o padrão. Empresas que produzem esse tipo de extensões normalmente utilizam abreviações no nome das novas funções e constantes. A nVidia por exemplo utiliza *NV* na definição de suas funções proprietárias, como exemplo: *glCombinerParameterfvNV()* e constantes: *GL_NORMAL_MAP_NV*. A ATI suporta extensões em *Windows* e MacOS, como por exemplo *GL_ATI_draw_buffers* (OPENGL - ..., 2006).

Extensões as quais mais de um fabricante utiliza para uma mesma funcionalidade passam a utilizar a notação *EXT*. Pode ainda ocorrer que a ARB aprove esta extensão, e torne ela uma extensão padrão, então a abreviação ARB é utilizada. A primeira extensão ARB foi *GL_ARB_multitexture*, introduzida na versão 1.2.1. Seguindo o caminho oficial, *multitexturing* deixou de ser uma extensão opcional implementada pela ARB, mas tornou-se parte do *core* do *OpenGL* desde sua versão 1.3 (OPENGL - ..., 2006).

---

[2] *OpenGL Architecture Review Board* - ARB é uma consórcio de industrias que controla a especificação do *OpenGL*. Ela foi criada em 1992. Em julho de 2006 o controle das especificações do *OpenGL* passou para o *Khronos Group*. Os membros votantes são 3Dlabs, *Apple Computer*, ATI, Dell, IBM, Intel, nVidia, SGI e *Sun Microsystems* entre outros contribuintes. *Microsoft* era um dos membros votantes originais, mas desde 2003 abandonou o grupo.

2.5 *Pipeline* Gráfico

O *pipeline* gráfico é a descrição do processo ao qual uma estrutura que representa um modelo gráfico sofre até a sua apresentação num dispositivo. Neste processo são descritos todos os passos e todos os algoritmos que são aplicados aos modelos até estes serem apresentados no dispositivo (SANTOS, 2005), como pode ser visto nas Fig. 2.6 e Fig. 2.7.

As implementações em *OpenGL*, seguem a ordem de operações de seu *pipeline* gráfico. A compreensão deste fluxo de processos é importante para o programador saber como e quando a *OpenGL* realiza suas operações (Fig. 2.8).

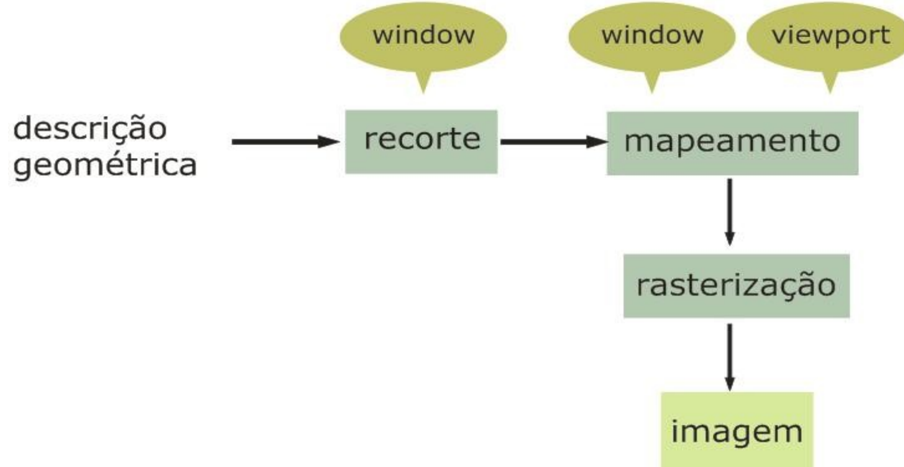


Figura 2.6 - *Pipeline* de visualização 2D.

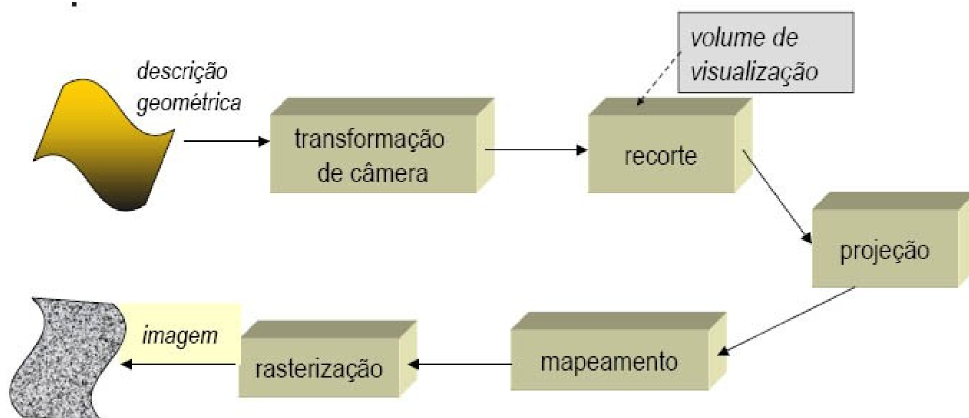


Figura 2.7 - *Pipeline* de visualização 3D

Fonte: FREITAS, 2003.

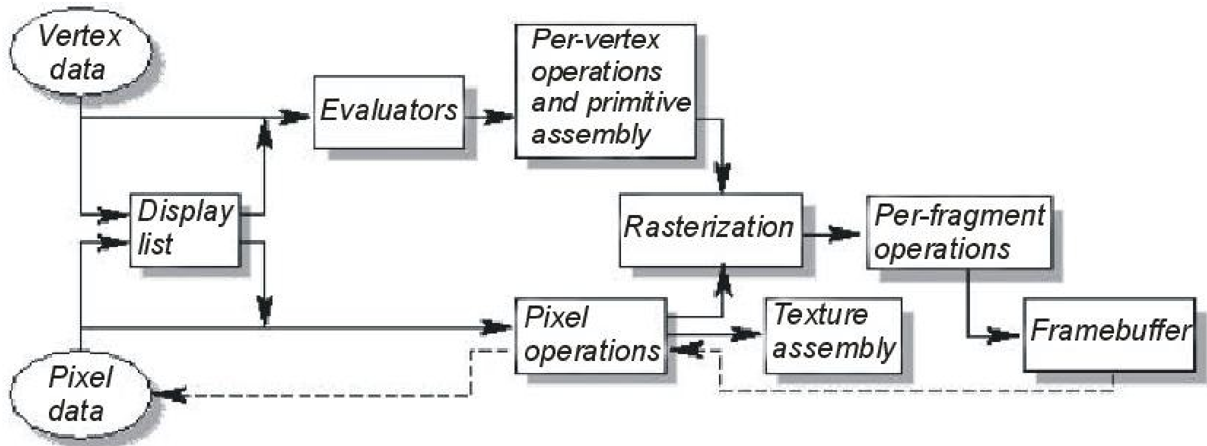


Figura 2.8 - *Pipeline OpenGL*.

Fonte: NEIDER, 1997, p.19.

3 GERAÇÃO PROCEDURAL

Geração Procedural é o uso de algoritmos para gerar conteúdo para uma cena gráfica. Também conhecido como modelagem procedural, síntese procedural, criação de conteúdo procedural, modelagem generativa, ou modelagem automática. Ainda não é um campo bem definido, mas possui uma variedade de métodos explorados em vários níveis nas últimas três décadas (EBERT, 2003).

Este tipo de geração é o termo utilizado para indicar a criação de conteúdo dinamicamente, ao contrário do conteúdo já modelado previamente. Geralmente relacionado às aplicações da computação gráfica. Também conhecido como Síntese Procedural. Este conceito é utilizado, por exemplo, nos fractais. Outros usos comuns incluem texturas e malhas de polígonos. O Som também pode ser gerado, mas raramente é feito desta forma em aplicações para PC. Técnicas de Geração Procedural são utilizadas a muito tempo em jogos, alguns poucos usam esta abordagem extensivamente, como: *Will Wright's Spore*, jogo de videogame que tem seu cenário povoado inteiramente com geração de conteúdo procedural; *Soldier of Fortune* utiliza rotinas para adicionar detalhes aos soldados inimigos; pode ser dito também que a iluminação em *Doom 3* foi gerada proceduralmente porque não conta com *lightmaps* pré-computados utilizando um processo de radiosidade (MARTIN, 2006).

A modelagem e visualização de sistemas feitos manualmente como grandes cidades é um grande desafio da computação gráfica. Cidades são sistemas com grande complexidade funcional e visual. Elas refletem as mudanças históricas, culturais, econômicas e sociais através do tempo em cada aspecto que é vista. A modelagem e visualização de grande áreas de cidades utilizando computadores tornou-se possível com a grande quantidade de memória, processamento e poder gráfico do *hardware* atual.

Modelagem visual de grandes e complexos sistemas tem longa tradição na computação gráfica. A maioria dessas abordagens enfatiza a aparência do fenômeno. O grande apelo dessas renderizações situa-se na possibilidade de descrever a complexidade de sistemas de larga escala, compostos de elementos mais simples.

Segundo Martin (2006), pode ser elaborada uma topologia de sua organização quanto ao tipo de geração de conteúdo:

- Síntese – texturas, animação e modelos.
- Fractais – terrenos e paisagens.
- Baseados em gramáticas – fachadas arquiteturais, construções, casas.
- Baseados em física – nuvens, outros fenômenos naturais diversos.

Geração de modelos baseado em gramáticas (especialmente L-*systems*) são empregados em computação gráfica principalmente para criar geometrias. Possibilitam também modelar os detalhes encontrados em formas naturais como plantas. Ao contrário da sombra, que é tratada como um operador que pode deformar a geometria de um objeto base como referência, a geração procedural de geometrias cria um objeto novo inteiramente pela geração de sua geometria a partir do nada (EBERT, 2003).

As idéias da síntese de geometrias procedurais estendem-se naturalmente à síntese de cenas inteiras procedurais. Estas técnicas não se limitam à descrição de crescimento de plantas a partir de um broto, mas possibilitam também preencher todo o cenário com florestas inteiras com instâncias únicas da árvore. Também podem ser aplicadas a aplicações não-biológicas, como geração automática de cidades inteiras .

O método mais popular para descrever modelos procedurais de plantas e formas naturais tem sido o L-*system*. O L-*system* é um sistema de regras de substituições que capturam vários aspectos próprios do formato biológico e desenvolvimento. A gramática opera em um conjunto de gráficos e símbolos que traduzem as palavras geradas pela gramática em objetos gráficos.

Técnicas procedurais são usadas para modelar detalhes, mas o processamento eficiente e renderização de cenas contendo grande quantidade de detalhes geométricos requerem organização hierárquica, com estrutura de dados coerente.

Porém, outras alternativas podem ser mais adequadas para modelar arquiteturas. A abordagem mais utilizada é a construção e análise do *design* arquitetônico utilizando gramáticas de formas, introduzida por Stiny (1975). Estas gramáticas operam diretamente nas formas e tem se mostrado muito úteis na análise e construção de muitos estilos. Entretanto, a derivação em tais gramáticas é normalmente feita manualmente, ou de forma assistida por computador, com alguém decidindo quais regras a serem aplicadas.

3.1 Paradigmas de construção de cidades virtuais

Algoritmos de geração procedural de geometrias classificam-se em duas classes. Algoritmos *data amplification* avaliam procedimentos para sintetizar toda sua geometria; algoritmos *lazy evaluation* avaliam os procedimentos de geração da geometria sob demanda.

Os sistemas de modelagem tendem a seguir um fluxo de dados padrão. O usuário elabora um modelo conceitual para o modelador. o modelador interpreta o modelo e converte em uma representação intermediária apropriada para manipulação, processamento, e renderização. O renderizador aceita a representação intermediária e sintetiza uma imagem do objeto. O usuário então observa o modelo renderizado, compara ao modelo conceitual, e faz mudanças de acordo com as diferenças apresentadas. A síntese procedural de geometria segue o mesmo modelo de fluxo de dados da modelagem (EBERT, 2003).

3.2 Trabalhos relacionados

Trabalhados baseados na geração por gramáticas, aplicados à geração de plantas (MËCH; PRUSINKIEWICZ, 1996; PRUSINKIEWICZ; LINDENMAYER, 1991; PRUSINKIEWICZ et al., 1994; SHLYAKHTER et al., 2001).

Técnicas de *noise* (PERLIN, 1985), fractais (MANDELBROT, 1977), L-*systems* (PRUSINKIEWICZ et al., 1990) e gramáticas de formas (STINY, 1975).

Estas técnicas compõem sistema de geração de entidades tais como nuvens (PALLISTER, 2000), árvores (OPPENHEIMER, 1986) e outros fenômenos naturais. MACRI & PALLISTER (2000) descrevem geração de paisagem tridimensional composta de árvores e nuvens, baseados no trabalho de PERLIN (1985), sobre *noise*.

Os princípios da geração de números aleatórios relacionados com geração de conteúdo procedural para cenários 'infinitos' em ambientes de jogos é discutido por Lecky-Thompson (2001).

O sistema chamado *CityEngine* (PARISH; MULLER, 2001) que utiliza L-*systems* estendidos para gerar modelos de cidades inteiras. Utiliza um conjunto de regras hierárquicas para gerar padrões de ruas e construções. Outra ferramenta para gerar cidades é o *The Other Manhattan Project* (YAP et al., 2001) que descreve ferramentas para geração automática de cidades, que baseadas em parâmetros estatísticos, lembre a aparência de Manhattan. Modelagem procedural de prédios, criando uma gramática de formas (MÜLLER et al., 2006).

3.2.1 Geração Procedural em jogos

Os primeiros jogos de computador eram fortemente limitados pela pouca capacidade de memória dos computadores antigos. Isto motivou a criação de conteúdo algoritmicamente sob demanda, pois simplesmente não havia capacidade de memória para armazenar quantidade suficiente de dados. Geradores de números pseudo-aleatórios foram muito utilizados em conjunto de valores de entrada pré-definidos a fim de criar cenários que aparentavam ser previamente modelados. O jogo *The Sentinel* tinha supostamente 10.000 níveis diferentes armazenados em somente 48 ou 64 kilobytes. Um caso extremo é do jogo Elite, que foi planejado para suportar até 2^{48} galáxias com 256 planetas cada, porém na versão final do jogo continha apenas 8 galáxias (PROCEDURAL GENERATION..., 2006).

Hoje em dia, a maioria dos jogos exige muito mais memória para armazenar dados como texturas, do que seus algoritmos, seu *engine*. Um exemplo é o jogo *Grand Theft Auto - San Andreas*, que mesmo possuindo um cenário muito vasto, seus prédios são todos modelados individualmente. Computadores pessoais de hoje possuem uma razoável capacidade de processamento e armazenamento

possibilitando gráficos mais detalhados, necessitando também mais trabalho por parte dos artistas. Isto implica que os artistas que modelam jogos cada vez mais percam mais tempo para modelar um personagem, veículo, prédio ou textura. Esperam-se jogos cada vez mais detalhados e realistas, aumentando assim os custos de produção.

Abordagens procedurais vêm para resolver alguns dos problemas relacionados com a geração de conteúdo. Artistas e programadores podem criar algoritmos que automaticamente geram geometrias, texturas entre outros. Em alguns casos a geração procedural provou-se um método mais barato e efetivo.

Supondo que um algoritmo possa gerar um prédio, o mesmo algoritmo pode ser modificado para gerar prédios randômicos, e desse modo preencher uma cidade inteira em tempo de execução, ao invés de modelar todas as construções e ter elas armazenadas para posterior renderização. O mesmo exemplo serve para gerar uma árvore e assim poder gerar uma floresta inteira. Estes métodos exigem mais poder de processamento, mas com as Unidades Centrais de Processamento - CPU's cada vez mais rápidos isto não se torna uma questão tão séria (PROCEDURAL GENERATION..., 2006).

A indústria de jogos, cinema e animação têm na Geração Procedural uma ferramenta muito útil na geração de seus cenários. Essa geração possibilita uma redução de custos de produção e aumento no detalhamento dos modelos, visto que o mercado demanda cenários cada vez mais detalhados. A seguir serão apresentados um jogo e uma ferramenta que gera ambientes preenchidos com vegetação, ambos utilizam os conceitos de Geração Procedural para produzir suas cenas.

3.2.2 Exemplos de utilização da Geração Procedural

3.2.2.1 Jogo .kkrieger

É um jogo de computador do estilo primeira pessoa, criado em 2004 pelo grupo alemão .theprodukkt. Em abril de 2004 ganhou a *96K competition* no *Breakpoint*[3].

---

[3] Reunião anual realizada na Alemanha para exibição de demos.

O jogo inteiro utiliza 97.280 bytes de espaço em disco. Seu tamanho reduzido deve-se a Geração Procedural de seu conteúdo. Ao contrário da maioria dos jogos de primeira pessoa hoje em dia que ocupam um *Digital Video Disc* - DVD inteiro, por exemplo, *Doom 3* ocupa o equivalente a 3 *Compact Disc's* - CDs já o *Far Cry* ocupa quase 3 gigabytes. Vale lembrar que o .kkrieger possui somente um nível (um cenário) enquanto *Doom 3* e *Far Cry* são jogos com diversos cenário diferentes (.THEPRODUKKT, 2004).

Entretanto integra recursos de jogos 'normais' tais como cenário tridimensional, texturas realistas, efeitos de iluminação e sombreamento, sons, música, Inteligência Artificial - IA e iteração. Permite importar dados de outras ferramentas, recurso que os outros sistemas procedurais normalmente não suportam, e também possui escalabilidade para tratar grandes cenários. De acordo com os desenvolvedores se ele fosse desenvolvido de maneira tradicional ocuparia entre 200 e 300 megabytes – MB de espaço em disco.

O grupo .theprodukkt desenvolve o .kkrieger desde 2002, utilizando sua ferramenta chamada .werkkzeug. Para o desenvolvimento do .kkrieger foi utilizada uma versão ainda não disponível, .werkkzeug3.

3.2.2.1.1 Tecnologia

O jogo .kkrieger (Fig. 3.1) utiliza a ferramenta .werrzeug3, responsável pela criação de conteúdo procedural e gerência. Ela cria todos os aspectos visuais do .kkrieger, materiais, texturas, níveis, personagens e animações. Seu princípio fundamental é similar aos sintetizadores modulares. Os geradores de textura e malhas de polígonos em partes simples e independentes (operadores), cada uma com seu conjunto de parâmetros e podendo ser conectadas a outros operadores. Esta combinação de operadores é capaz de produzir resultados complexos e detalhados (.THEPRODUKKT, 2004).

As texturas são armazenadas através do histórico de criação, ao contrário de uma base por *pixels*, deste modo requer apenas os dados de seu histórico e o código de seu gerador a ser compilado em tempo de execução, suas texturas ocupam aproximadamente 300 bytes em qualquer resolução utilizada. As malhas

são criadas a partir de sólidos básicos como cubos e cilindros, que são manipulados para atingir as formas desejadas (.THEPRODUKKT, 2004).

A forma de geração utilizada causa um tempo de carregamento (*loading*) grande, pois todos os itens relativos a jogabilidade são reproduzidos durante o período de carregamento do jogo. Os sons em musica do jogo são produzidos pelo sintetizador V2, que funciona a partir de um *stream* contínuo de dados da Interface Digital para Instrumentos Musicais - MIDI. Conseqüentemente a música também é gerada em tempo real (.THEPRODUKKT, 2004).

Figura 3.1 – Jogo .kkrieger.

Fonte: http://212.202.219.162/, 2004.

3.2.2.2 Suíte para geração de árvores e plantas *Speedtree*

SpeedTree é uma suíte produzida pela *Interactive Data Visualization, Inc*. Foi projetada para criar árvores e plantas em tempo real para jogos e simulações

3.2.2.2.1 Componentes

Em seu pacote padrão o *SpeedTree* contém o *SpeedTreeRT* e o *SpeedTreeCAD*. Porém também há *plug-ins* para os programas de modelagem mais populares como Studio Max e Maya,

3.2.2.2.1.1 *SpeedTreeRT*

É um kit de desenvolvimento de *software* - SDK escrito em C++. É geralmente integrado a um *engine* gráfico, com o *SpeedTree* administrando a renderização das árvores, folhagem e plantas

3.2.2.2.1.2 *SpeedTreeCAD*

É uma ferramenta de modelagem criada especificamente para criar e editar árvores, folhagem e plantas. O *SpeedTreeCAD* (Fig. 3.2) permite especificar variáveis como: comprimento de galho, ângulo do galho, textura e influência gravitacional.

3.2.2.2.2 *SpeedTree* em jogos

Alguns dos jogos que utilizam o *middleware* SpeedTree:

- *Age of Conan - Hyborian Adventures*
- *Auto Assault*
- *The Elder Scrolls IV: Oblivion*
- *Elveon*
- *Fallen Lords: Condemnation*
- *Fatal Inertia*
- *Gods and Heroes: Rome Rising*
- *Gothic III*
- *The Guild II*
- *Hero's Journey*
- *Huxley*
- *Immortals: The Heavenly Sage*
- *Neverwinter Nights 2*
- *The Outsider*
- *Pirates of the Burning Sea*
- *Project Gotham Racing 3*
- *Ragnesis Online*
- *Rise & Fall: Civilizations at War*
- *Rise of Nations: Rise of Legends*
- *Saints Row*

- *Second Life*
- *Soul of the Ultimate Nation*
- *SpellForce 2: Shadow Wars*
- *Tiger Woods PGA Tour 2006 (Xbox 360)*
- *Too Human*
- *Trials of Atlantis*
- *Vanguard: Saga of Heroes*
- *The Witcher*
- *World in Conflict*
- *World War II Online*
- *ZerA*
- *Unreal Engine 3* (Fig. 3.3)

3.2.2.2.3 *SpeedTree* em outras aplicações

SpeedTree também é utilizado para aplicações que não sejam jogos, como as seguintes:

- Simulação do exército americano de um helicóptero Apache.
- Simulação de combate desenvolvida pela *Emergent Game Tecnologies* para o departamento de defesa americano.

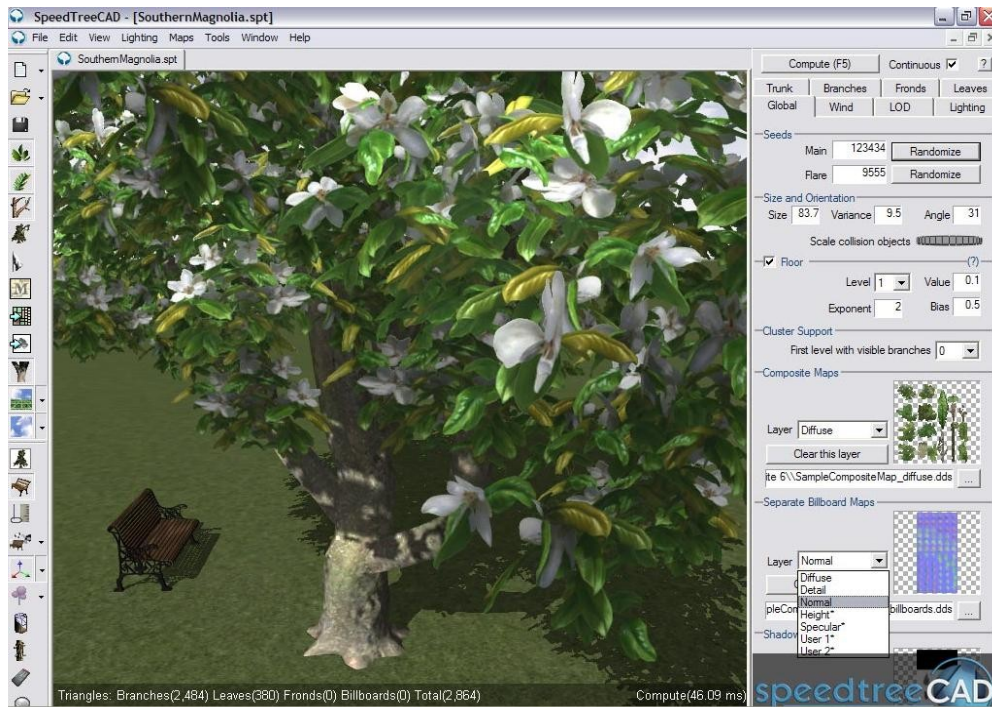


Figura 3.2 - *SpeedTreeCAD* Interface.

Fonte: http://www.speedtree.com/, 2006.

Figura 3.3 - *Unreal Engine* 3

Fonte: http://www.speedtree.com/, 2006.

3.3 *L-Systems*

L-*Systems* ou *Lindenmayer system*[4] é um formalismo matemático criado como um fundamento para uma teoria axiomática de desenvolvimento biológico. Mais recentemente, L-*systems* tem sido utilizado em aplicações na computação gráfica. L-*systems* são aplicados para geração de fractais, modelagem de plantas, criar padrões de ruas e estradas e vida artificial.

O fundamento principal ao L-*system* é a noção de reescrita, onde a idéia básica é definir objetos complexos pela sucessiva recolocação de geometrias utilizando um conjunto de regras de produções. A reescrita pode ser realizada recursivamente (EBERT, 2003).

O sistema de reescrita mais extensivamente estudado e melhor entendido opera em *strings* de caracteres. A partir do trabalho de Chomsky (1956) em gramáticas formais a ciência da computação teve início ao estudo mais aprofundado em gramáticas, tendo assim o surgimento das linguagens formais.

Aristid Lindenmayer introduziu em seu trabalho um novo tipo de reescrita de *strings*, o L-*system*. A diferença essencial entre as gramáticas de Chomsky e L-*Systems* é no método pelo qual é aplicado as produções. Na gramática de Chomsky produções são aplicadas seqüencialmente, enquanto nos L-*Systems* eles são aplicados em paralelo, substituindo simultaneamente todas as letras em uma dada palavra. Esta diferença reflete na motivação biológica do L-*system*. Produções são feitas com intuito de simular as divisões celulares em organismos multicelulares, onde muitas divisões ocorrem ao mesmo tempo (EBERT, 2003).

3.3.1 Origem

Como biólogo, Lindenmayer estudou os padrões de crescimento de vários tipos de algas, trabalhou com alguns tipos de fungos e leveduras. Originalmente os *L-systems* foram definidos para prover uma descrição formal do desenvolvimento destes organismos multicelulares simples e ilustrar os relacionamentos entre as

[4] L-*Systems* ou *Lindenmayer system* é um formalismo matemático proposto pelo biólogo e biologista teórico Húngaro Aristid Lindenmayer (1925-1989) em 1968 da Universidade de Utrecht.

células. Posteriormente, o sistema foi estendido para descrever plantas maiores e estruturas ramificadas, como raízes e galhos (L-SYSTEM..., 2006).

3.3.2 Estrutura

As regras do L-*system* tendem a recursão, e por meio disso tem similaridade às formas do tipo fractais, que são fáceis de descrever com L-*system*. Modelos de plantas e outras formas orgânicas são similarmente fáceis de definir. Aumentando o nível de recursão às formas lentamente crescem e tornam-se mais complexas.

L-*system* é também conhecido como um sistema paramétrico, definido pela tupla

$G = (V, S, \omega, P)$

onde

- **V** (alfabeto) é o conjunto finito de símbolos que podem ser substituídos (variáveis).
- **S** conjunto de símbolos contendo elementos que permanecem fixos (constantes).
- **ω** (início, axioma) é uma *string* de símbolos a partir de V definindo o estado inicial do sistema.
- **P** é o conjunto de regras de produções definindo como as variáveis podem ser substituídas por uma constante ou uma combinação de outras variáveis. A produção consiste de duas *strings*, o predecessor e o sucessor.

As regras da gramática são aplicadas iterativamente começando a partir do estado inicial ω**.** Um L-*system* é livre de contexto se cada regra de produção refere-se somente a um símbolo individual e não aos seus vizinhos. Se a regra não depende somente de um símbolo, mas também de seus seguintes é então definido como L-*system* sensível ao contexto (L-SYSTEM..., 2006).

3.3.3 D0L-*system*

Se há somente uma produção para cada símbolo, então o símbolo é classificado como determinístico. Um L-*system* determinístico livre de contexto é comumente chamado de D0L-*system*. D0L-*system* é a classe mais simples dos L-*systems*, D0L vem do determinismo e contexto-0 ou livre de contexto (WRIGHT, 1996).

Vejamos o exemplo criado por Prusinkiewicz & Lindenmayer (1991), o L-*System Fibonacci*:

Considera-se uma *string* formada de duas letras a e b, podendo ocorrer várias vezes na *string*. Para cada letra é especificada uma regra de produção. A regra a->b significa que a letra a será substituída pela letra b, e a regra b->ba significa que a letra b será substituída pela *string* ba. O processo de reescrita começa com uma *string* distinta chamada axioma. Assume-se que esse axioma consiste da letra a. Na primeira derivação (primeiro passo de reescrita) o axioma a é substituído por b usando a produção a->b. No segundo passo b é substituído por ba utilizando a produção b->ba. A palavra ba consiste de duas letras, ambas substituídas simultaneamente no próximo passo da derivação. Deste modo, b é substituído por ba, a por b, e a *string* bab gera babba que irá gerar babbabab, e assim por diante (Fig. 3.4).

A evolução do L-*system* é definida pela seqüência $\{g_n\}$, n = 0, 1, 2, 3, ... , onde cada geração g_n é uma *string* em V* que derivou a partir da geração anterior g_{n-1} pela aplicação das regras de produção para cada símbolo em g_{n-1} . a primeira geração é o axioma ω.

Temos então:

V = {a, b}

ω = a

P = {a->b; b->ba}

Figura 3.4 - Exemplo de D0L-*system*.

Podemos pensar que a's e b's são formas de vidas e as produções como estágios em suas vidas. Depois de uma geração, a forma de vida 'imatura' a amadurece em um adulto b. Depois o adulto b torna-se apto a produzir um bebê a em cada geração. Assim, este sistema simbólico modela um tipo de vida muito simples de população dinâmica.

A origem do nome Fibonacci vem da conexão com a seqüência de números 1, 1, 2, 3, 5, 8, 13, ... , conhecida como a série de Fibonacci. Estas são definidas por uma ordem de relação

$F_0 = F_1 = 1$

$F_{n+2} = F_{n+1} + F_n$ para $n \geq 0$

O número de organismos na n-ésima geração g_n é $|g_n| = F_n$. Isto significa que a geração n+2 consiste da geração *n*+1 seguida da geração *n*, por exemplo:

g_7 = (babbababbabba)(babbabab) = $g_6 g_5$

3.3.4 Fractais e a interpretação gráfica das *strings*

Lindenmayer *systems* foram criados como uma teoria matemática de desenvolvimento. Assim, aspectos geométricos foram inseridos no escopo da teoria. Surgiram algumas interpretações geométricas dos L-*systems* para transformá-lo em

uma ferramenta versátil para criação de fractais e modelagem de plantas, mais tarde utilizado para modelagem de prédios, padrões de ruas, etc.

Muitos fractais, ao menos suas aproximações finitas, podem ser interpretados como seqüências de elementos primitivos, como segmentos de linha. Para produzir fractais, as *strings* geradas por L-*systems* devem conter a informação necessária sobre a geometria da figura. Uma interpretação gráfica de *string*, baseada em geometria *turtle* é descrita por Prusinkiewicz et al. (1989, 1990). Esta interpretação pode ser usada para produzir imagens de fractais (OCHOA, 1998).

3.3.5 *Turtle graphics* e L-*Systems*

Seymour Papert inventou o *Turtle graphics* como um sistema para tradução de uma seqüência de símbolos em movimentos de um autômato (a tartaruga) no *display* gráfico. A idéia original era fornecer um objeto programável que fosse didático às crianças compreender geometria. Tornou-se um sistema ideal para dar uma interpretação geométrica aos L-*systems* dinâmicos. O sistema básico é o seguinte: é fixado um tamanho de passo *d* que será a distância coberta pela tartaruga a cada passo e um ângulo, normalmente 360º/*n,* para um inteiro n (EBERT, 2003).

Um estado da tartaruga é definido como uma 3-upla (x, y, a), onde as coordenadas cartesianas (x, y) representam a posição da tartaruga e o ângulo a é interpretado como a direção para qual a tartaruga está apontando. Dado o tamanho do passo d e o incremento de ângulo b, a tartaruga pode responder aos comandos representados pelos seguintes símbolos (WRIGHT, 1996):

- **F** move um passo à frente de tamanho d. O estado da tartaruga muda para (x’, y’, a), onde

  x’ = x + d cos(a) e

  y’ = y + d sen(a)

  Um segmento de linha entre os pontos (x, y) e (x’, y’) é desenhado.

- **f** move um passo de tamanho d sem desenhar uma linha.

- **+** vira à esquerda (sentido anti-horário) com ângulo b. O próximo estado da tartaruga é (x, y, a+b).
- **-** vira à direita (sentido horário) com ângulo b. O próximo estado da tartaruga é (x, y, a-b).

Como exemplo do uso destes símbolos mostraremos a curva de Koch:

V = {F, +, -}

ω = F

p_1 : F -> F + F - - F + F

Sem a especificação para alguns símbolos, assume-se que + e - são constantes. A geração 0 é somente uma linha reta; assume-se que o começo é sob o eixo horizontal positivo. Toma-se o ângulo β como 60º = 360º/6. Geometricamente, esta produção significa substituir o segmento de linha F pelo seguinte arranjo de 4 segmentos de linha (Fig. 3.5).

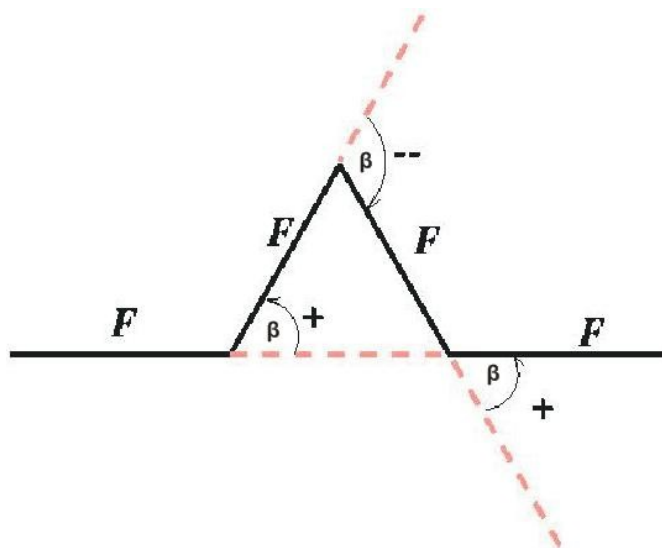


Figura 3.5 - A produção para a curva de Koch.

Fonte: WRIGHT, 1996.

Sendo assim, na Fig. 3.6 são apresentadas as próximas gerações:

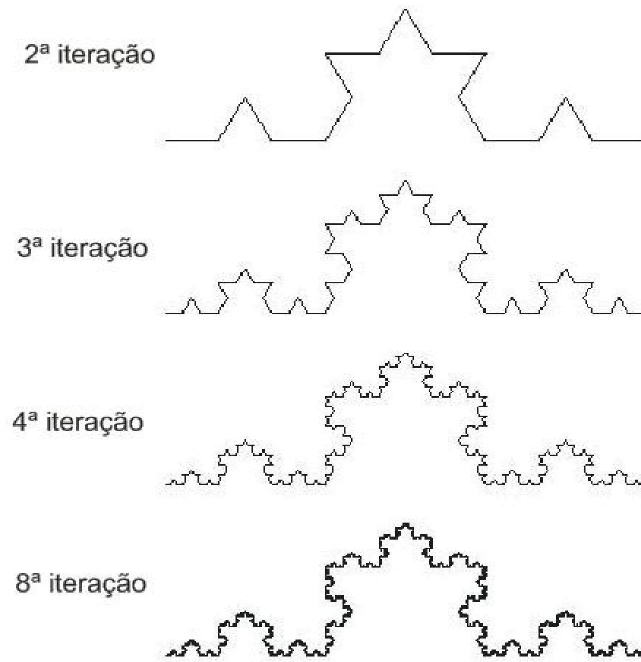


Figura 3.6 - Produções da curva de Koch.

Fonte: WRIGHT, 1996.

Outro exemplo, “Ilha quadrática de Koch” (Fig. 3.7)

V = {F, +, -}

ω = F+F+F+F

p_1 : F -> F + F - F - F F + F + F- F

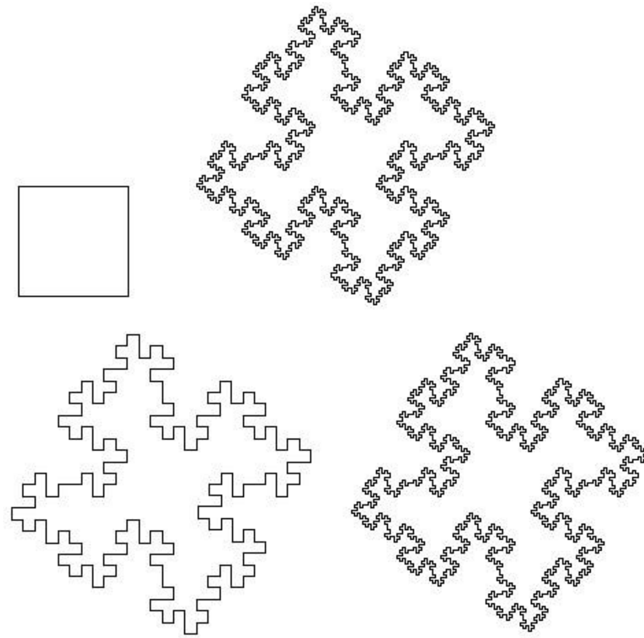


Figura 3.7 - Ilha quadrática de Koch derivações de tamanho n =0, 1, 2 e 3 respectivamente, com ângulo *b* igual a 90º.

Fonte: OCHOA, 1998.

Desta maneira é possível criar um grande variedade de linhas irregulares. Para utilizar os L-*systems*, na modelagem de estruturas como plantas ou padrões de ruas é necessário estender os símbolos utilizados no L-*system*.

- [salva a posição e orientação da tartaruga.
-] leva a tartaruga ao estado armazenado mais recente.

As informações dos estados podem ser armazenadas em uma pilha, permitindo que os colchetes possam ser aninhados. Estes estados permitem modelar estruturas do tipo galho ou ramo. Por exemplo, como pode ser visto na Fig. 3.8 o L-*system* representado pela produção:

F -> F [+ F] F [- F] F

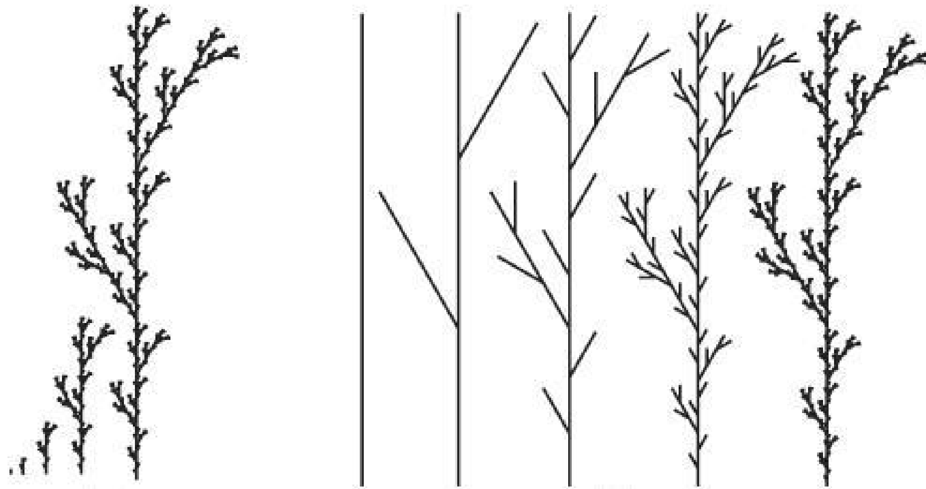


Figura 3.8 - Exemplo de utilização dos símbolos [e].

Fonte: EBERT, 2003, p.309.

Para estender suas funcionalidades às estruturas 3D, são adicionados os símbolos “^” e “&” que modificam a altura e também os símbolos “\” e “/” para controlar sua rotação (esquerda e direita) (EBERT, 2003). A Fig. 3.9 ilustra a seguinte produção:

F -> F [& F] [& / F] [& \ F]

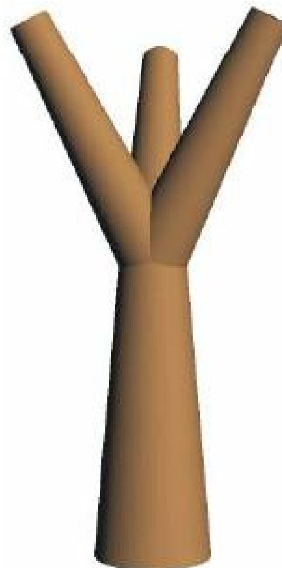


Figura 3.9 - Exemplo de utilização dos símbolos \ e /.

Fonte: EBERT, 2003, p.310.

Os elementos de geometria do L-*system* podem ser parametrizados para fornecer melhor controle. Por exemplo, F(50) desenha um segmento de 50 unidades, e +(30) rotaciona 30º sobre o eixo de direção. A Fig. 3.10 representa graficamente a produção:

F -> / (180) – (30) F + (60) F – (30) \ (180)

Figura 3.10 - L-system 3D parametrizado.

Fonte: EBERT, 2003, p.310.

3.3.6 L-*systems* estendidos - criando um mapa de ruas

Para criar um mapa de ruas necessita-se uma grande número de parâmetros e condições a serem implementados em um L-*system*. O número de produções e sua complexidade crescem muito rapidamente. Cada vez que uma restrição é implementada, algumas regras devem ser reescritas, isto torna muito difícil estender os L-*systems*. Deste modo, ao invés de tentar ajustar os parâmetros dos módulos dentro das produções o L-*system* cria somente um modelo de cada passo, uma espécie de sucessor ideal.

Por esta razão o ajuste e modificação dos parâmetros nos módulos dos L-*systems* é portado para funções externas. Estas funções obedecem a uma hierarquia para a distinção entre tarefas de alto nível e restrições locais. Um modelo

para esta implementação foi desenvolvido por PARISH & MUELLER (2001). *Procedural modelling of cities*; são criadas as funções *globalGoals* e *localConstraints*, para especificar os objetivos globais e as restrições globais respectivamente. Cada vez que uma regra é aplicada a uma *string* de módulos já existente, as seguintes ações são tomadas (Fig. 3.11):

- Retornar o *successor ideal* do *L-system*. Os parâmetros dos módulos são reinicializados.
- Chamar a função *globalGoals*. Esta função irá determinar os valores dos parâmetros dos objetivos globais dominantes.
- Chamar a função *localConstraints*. Os parâmetros são verificados dentro das restrições locais do ambiente. Os valores dos parâmetros são ajustados para se ajustar a estas restrições. Se esta função não pode encontrar parâmetros ideais, ela ajusta o estado do *flag* para *FAILED* e o módulo é subseqüentemente apagado.

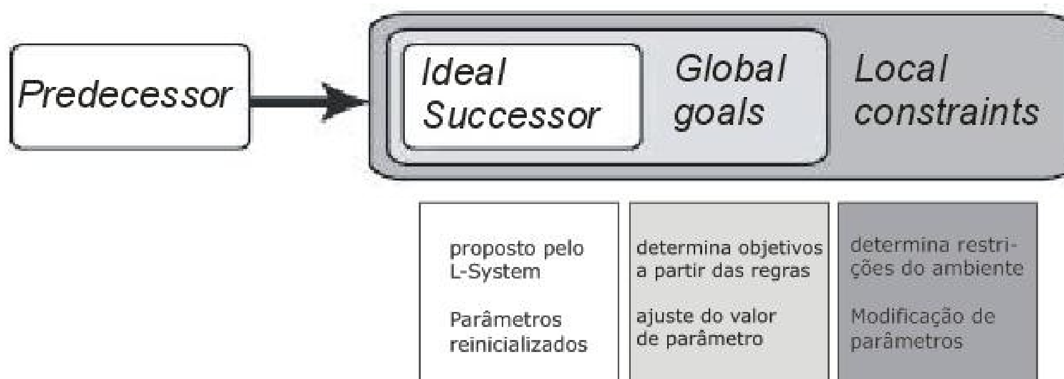


Figura 3.11 - Funções aplicadas a um sucessor.
Fonte: PARISH, 2001.

Os parâmetros das funções *globalGoals* e *localConstraints* são influenciados por características do ambiente ao qual deseja-se modelar. A criação do mapa de ruas deve distinguir as restrições globais e locais, elas podem ser classificadas da seguinte forma:

- Globais - padrão de rua desejado e densidade populacional.
- Locais - limites das construções, elevação do terreno e intersecção das ruas.

Com este *design* modularizado de L-*system* e os cálculos de parâmetros passados a outros procedimentos, as regras para este tipo de L-*system* são

simplificadas. Facilitando a introdução de novas regras e restrições, sem ter de modificar as regras de produções.

A seguir será apresentado um exemplo utilizado no sistema *CityEngine* (PARISH; MUELLER, 2001):

ω: R(0, initialRuleAttr) ?I(initRoadAttr, UNASSIGNED)

p1: R(del, ruleAttr) : del<0 → e

p2: R(del, ruleAttr) > ?I(roadAttr,state) : state==SUCCEED

{globalGoals(ruleAttr,roadAttr) cria os parâmetros

for: pDel[0-2], pRuleAttr[0-2], pRoadAttr[0-2]}

→ +(roadAttr.angle)F(roadAttr.length)

B(pDel[1],pRuleAttr[1],pRoadAttr[1]),

B(pDel[2],pRuleAttr[2],pRoadAttr[2]),

R(pDel[0],pRuleAttr[0]) ?I(pRoadAttr[0],UNASSIGNED)

p3: R(del, ruleAttr) > ?I(roadAttr, state) : state==FAILED → e

p4: B(del, ruleAttr, roadAttr) : del>0→B(del-1, ruleAttr, roadAttr)

p5: B(del, ruleAttr, roadAttr) : del==0

→[R(del, ruleAttr)?I(roadAttr, UNASSIGNED)]

p6: B(del,ruleAttr,roadAttr) : del<0→ e

p7: R(del,ruleAttr) < ?I(roadAttr,state) : del<0→ e

p8: ?I(roadAttr,state) : state==UNASSIGNED

{localConstraints(roadAttr) adjusts the parameters for:

state, roadAttr}

→ ?I(roadAttr, state)

p9: ?I(roadAttr,state) : state!=UNASSIGNED→ e

Na gramática acima, ω inicializa um módulo de ruas R e um módulo de inserção *?I*. Este segmento inicial necessita ser localizado dentro de uma área legal dos dados de entrada do usuário, como por exemplo: não localizado sobre uma construção, sobre terreno indesejável como água ou fora dos limites do cenário.

As primeiras três produções controlam o módulo R. A produção 2 controla a criação de uma estrada e suas ramificações. São criadas duas ramificações B e um módulo de estrada R após o módulo de inserção *?I* ser criado. Seus atributos são inicializados de acordo com os objetivos globais que retornam um *array* de atributos (*pDel[0-2]* para *delay* e deleção, *pRuleAttr[0-2]* para atributos específicos de regras e *pRoadAttr[0-2]* para os dados da estrada, como comprimento, ângulo, etc).

O módulo R usa o parâmetro *del* como um *flag* de parada. A função *globalGoals* pode ajustar isto a um valor negativo e o módulo R é apagado na próxima iteração pela produção *p1*. Em *p3* qualquer modulo R é apagado se seu estado é *FAILED* pela função *localConstraints*, por meio disto removendo o segmento de rua.

A produção 5 cria um novo segmento de rua nos pontos de ramificação depois do contador (*del*) atingir zero. A função *globalGoals* pode impedir ramificações, ajustando seu parâmetro de *del* para um valor negativo. A produção 7 apaga o módulo de inserção *?I* em casos onde *globalGoals* em *p2* ajusta o *delay* do R resultante a um valor negativo. As últimas duas produções determinam a criação da rua corrente, verificando se todos as restrições locais são satisfeitas chamando a função *localConstraints*, que ajusta os valores nos atributos da rua *roadAttr*. O estado variável é modificado pela função *localConstraints* para *FAILED* ou *SUCCED* e determina se o segmento de rua é criado.

3.4 Geração de números pseudo aleatórios

A geração de números pseudo aleatórios tem grande importância na geração procedural. Faz parte dos algoritmos de geração dos planos de chão, geração dos prédios, entre outros, e o que mais venha a ser gerado proceduralmente no cenário.

Um gerador de números pseudo randômicos produz uma seqüência de números randômicos a partir de um valor de entrada. Quando o gerador é inicializado com o mesmo valor de entrada, irá produzir uma seqüência idêntica de números. É importante garantir que a seqüência de números gerados seja a mesma, pois irá garantir para uma dada célula no espaço o mesmo prédio, sempre que o observador retornar a esta posição e assim mantém a coerência da cidade.

Um gerador de números adequado para esta tarefa são os geradores congruentes lineares, também é indicado o uso do gerador Mersenne twister, este último mais poderoso e mais adequado se o projeto exige uma melhor qualidade dos números, exigindo pouca memória a mais que o método anterior.

3.4.1 Geradores congruentes lineares

Os Geradores congruentes lineares são um dos mais antigos e conhecidos algoritmos de geração de números pseudo-aleatórios. Sua teoria é simples e de fácil implementação, porém não é dos geradores mais eficientes, mas dependendo das exigências do projeto envolvido ele supre as necessidades. Garantindo uma diversidade razoável e rapidez no processo.

Definido por:

$$Z_{i+1} = (a \cdot Z_i + c) \bmod m$$

Onde:

- Z é a semente (*seed*)
- a constante multiplicadora
- c incremento
- m módulo

O comportamento cíclico é inevitável nos geradores, visto que quando o número gerado já tiver sido gerado anteriormente, o ciclo reiniciará. Este ciclo é chamado de período do gerador. Na melhor das hipóteses o período poderá ser *m*.

3.4.2 Mersenne twister

Mersenne twister[5] é um algoritmo de geração de números pseudo-aleatórios do tipo *twisted generalised feedback shift register, twisted* - GFSR ou TGFSR. Possui período de $2^{19937} - 1$, para efeito de comparação para garantir que um conjunto de cartas de um baralho seja completamente aleatório é requerido 52! combinações, aproximadamente 2^{225}. Tem rápida geração, tão rápido quando a função *rand()* do C. Possui uso de memória eficiente, o código implementado em C consome 624 palavras na memória. Na página oficial dos pesquisadores encontra-se diversas implementações do algoritmo, em diversas linguagens e especificações de arquiteturas (MATSUMOTO; NISHIMURA, 2002).

3.5 *Hashing*

Outra forma de conseguir a geração de um número pseudo-aleatório é utilização de funções de *hashing*. Os prédios gerados terão forma e aparência a serem determinados pelos valores de entrada fornecidos pelos geradores de números e também pela sua posição. Essa seqüência de números irá determinar características como número de andares, altura, largura e textura.

Um bom exemplo que pode ser utilizado é a função de *hashing* de Thomas Wang a *Thomas Wang's 32 bit Mix Function*. É rápida e com boa distribuição de valores. Utiliza operações como soma, complemento de soma e deslocamento (*shift*), retornando um inteiro de 32 bits para qualquer *key*. Sua ultima versão:

```
int intHash(int key)
{
  key += ~(key << 15);
  key ^=  (key >>> 10);
  key +=  (key << 3);
```

[5] Mersenne twister é um algoritmo de geração de números pseudo-aleatórios desenvolvido pelos matemáticos japoneses Makoto Matsumoto e Takuji Nishimura em 1996/1997 e aperfeiçoado em 2002.

```
  key ^=  (key >>> 6);
  key += ~(key << 11);
  key ^=  (key >>> 16);
  return key;
}
```

4 GERAÇÃO DE PRÉDIOS

A modelagem de um prédio pode atingir diversos níveis de complexidade e detalhamento. Primeiramente são definidos os limites espaciais que ele vai ocupar no terreno. Um sólido simples pode ter aparência de uma construção apenas com aplicação de textura.

A partir daí podem ser feitas combinações de sólidos (*bounding volumes*) a fim de determinar sua forma. Complementarmente adicionar modelos de telhados. E então definir texturas para enriquecer seu visual.

4.1 Geração dos Planos de Chão

São basicamente polígonos de 2 dimensões, sobre o qual a cidade será gerada. Podendo ser utilizados mais de um plano dependendo das necessidades do projeto. Suas coordenadas podem ser utilizadas como parâmetros para geração das geometrias da cidade, como suas construções e características de relevo.

Estes planos são gerados de forma procedural (processos iterativos). A cada passo da câmera (observador) há iteração de suas coordenadas que irão servir de fonte para alguns dos parâmetros dos algoritmos de geração dos planos de chão e demais algoritmos de geração procedural. Cada célula do plano delimita os limites no espaço das construções ou da rua.

4.2 Gramáticas de formas

Como uma gramática para *design*, L-*systems* alcançaram resultados expressivos, especialmente na área da modelagem de plantas e também na modelagem de padrões de ruas. Porém, L-*systems* não se adapta facilmente a modelagem de prédios uma vez que eles simulam crescimento em espaços abertos.

Prédios ao contrário possuem restrições espaciais, e sua estrutura geralmente não reflete um processo de crescimento, mas sim uma seqüência de passos de particionamento (MÜLLER, 2006).

Por esta razão outras alternativas são mais apropriadas para a modelagem de arquiteturas. A abordagem mais comum é a geração e análise do design arquitetônico utilizando gramáticas de formas, introduzida por Stiny (1975). Estas gramáticas operam diretamente em formas e tem se mostrado muito úteis na análise e construção de estilos arquitetônicos. Entretanto, a derivação em tais gramáticas é feita manualmente ou de forma automatizada pelo computador, com o usuário decidindo quais regras aplicar. Isto torna gramáticas de formas uma ferramenta muito útil para algumas aplicações, possibilitando uma rápida modelagem de grandes áreas (MÜLLER, 2006).

A notação desta gramática e as regras como união, escala, translação e rotação são inspirados nos L-s*ystems*, mas estendidos para a modelagem de arquiteturas. Enquanto gramáticas paralelas como L-*systems* tem por objetivo capturar o crescimento através do tempo, uma aplicação seqüencial de regras permite a caracterização de uma estrutura (MÜLLER, 2006).

4.2.1 Formas

Uma forma tem a seguinte definição dada por Stiny (1980):

“Uma forma é um arranjo limitado de linhas retas em um espaço Euclidiano tridimensional.”

Uma linha l é determinada por dois pontos distintos p_1 e p_2 $(l = p_1, p_2)$. Duas linhas são idênticas se seus pontos finais são idênticos, e conseqüentemente, duas formas são idênticas se elas contém o mesmo conjunto de linhas. Formas podem ser atribuídas com um conjunto de pontos identificados por um rótulo identificador. Um ponto com o identificador p: A é um ponto p com um símbolo A atribuído a ele. Uma forma idêntica é denotada por (f, P), onde P é o conjunto de pontos identificados. Além disso, formas podem ser parametrizadas, permitindo que as coordenadas dos pontos que compõem as linhas possam ser variáveis (WONKA, 2006).

As formas trabalhadas pelas gramáticas consistem de um símbolo (*string*), geometria (atributos geométricos) e atributos numéricos. São identificadas pelos seus símbolos, que são representados por um terminal na gramática pertencente ao alfabeto, ou um símbolo não terminal. As formas correspondentes são chamadas formas terminais e formas não-terminais. Os atributos mais importantes são a posição *P*, e os vetores *X*, *Y*, e *Z*, que descrevem a posição espacial da forma, e o vetor de tamanho, *S*. estes atributos definem um *bounding volume* no espaço (WONKA, 2006).

4.2.2 Produções

De acordo com o modelo proposto por Pascal Müller em *Procedural Modeling of Buildings*: uma configuração é um conjunto finito de formas básicas. O processo de produção pode começar com uma configuração arbitrária de formas *A*, chamado axioma, o processo segue da seguinte forma:

1. Seleciona a forma ativa com o símbolo *B* no conjunto.
2. Escolhe a regra de produção para *B* do lado esquerdo para computar um sucessor para *B*, um novo conjunto de formas *BNEW*.
3. Marca *B* como inativo e adiciona as formas *BNEW* à configuração e continua com o passo 1.

Quando a configuração não possuir mais não-terminais, o processo de produção acaba.

Dependendo do algoritmo de seleção no passo 1, a árvore de derivação pode ser explorada tanto por *depth-first* ou *breadth-first*. Porém ambos os conceitos não permitem total controle das derivações. Portanto, é definida uma prioridade para todas as regras de acordo com os detalhes representados pela forma para obter uma derivação *breadth-first* - é selecionada a forma que possui a regra com maior prioridade no passo 1. Esta estratégia garante que a derivação seguirá uma ordem, de menor detalhamento para maior detalhamento das formas de forma controlada. Entretanto, as formas não são excluídas, mas são marcadas como inativas, depois de substituídas. Isto permite consultar a hierarquia da forma, ao invés de somente acessar a configuração ativa.

4.2.3 Notação

As regras de produção são definidas da seguinte forma:

id: predecessor: cond -> sucessor

Onde *id* é o identificador da regra, predecessor é um símbolo identificando a forma que será substituída pelo sucessor e cond uma função que deve ser *true* para que a regra seja aplicada. Ex.:

1: fac(h) : h > 9 -> andar(h/3) andar(h/3) andar(h/3)

Substitui a forma *fac* (fachada) pelas três formas *andar*, se o parâmetro *h* é maior que 9.

4.2.4 Regras de escopo

Similarmente aos L-*systems* são utilizadas regras gerais para modificar as formas: T(t_x, t_y, t_z) é um vetor de translação que é adicionado à posição do escopo *P*, R_x*(ângulo)*, R_y*(ângulo)*, e R_z*(ângulo)* rotaciona o respectivo eixo do sistema de coordenadas, e S(S_x, S_y, S_z) ajusta o tamanho do escopo (Fig. 4.1). Utiliza-se [e] para armazenar e consultar o escopo atual da pilha. Qualquer símbolo não terminal na regra será criado com o escopo atual. Da mesma forma, o comando *I*(*objId*) adiciona uma instância de uma primitiva geométrica com o identificador *objId*. Objetos normais incluem cubos, cilindros, também podem ser utilizadas outras formas tridimensionais (MÜLLER, 2006).

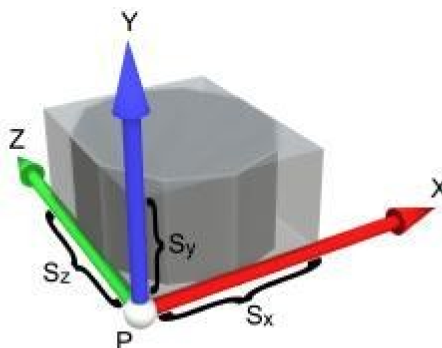


Figura 4.1 - Escopo de uma forma. O ponto P, juntamente com os eixos X, Y e Z e o tamanho S definem uma caixa quadrangular no espaço contendo a forma.

Fonte: MÜLLER, 2006.

A Fig. 4.2 ilustra a seguinte produção (o uso de colchetes indica a forma base do volume, as transformações dos outros volumes são realizados tendo por base os vetores desta forma base):

I: A -> [T(0,0,6) S(8,10,18) I("cubo")] T(6,0,0) S(7,13,18) I("cubo") T(0,0,16) S(8,15,8) I("cilindro")

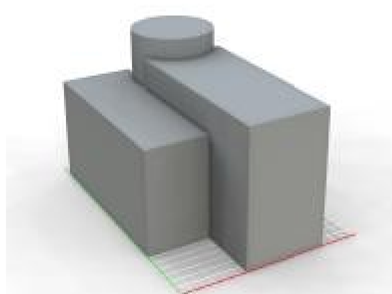


Figura 4.2 - Um modelo primitivo formado de três primitivas de formas.

Fonte: MÜLLER, 2006.

4.3 Construção de Fachadas

Depois de modeladas as formas obtêm-se um volume que representa um prédio. Resta então aplicação da textura, que pode vir a ser uma textura única para cada face da forma ou então para aumentar seu nível de detalhamento e realismo pode-se subdividir a fachada. Desta forma tratando cada parte da fachada de forma separada, como portas, janelas e paredes. São utilizadas então regras para uma subdivisão das formas existentes.

4.3.1 Regras para subdivisão das formas

Regras de divisão são regras que dividem o escopo corrente sobre um eixo. Pode-se dividir a fachada de um prédio para formar seus respectivos andares. Por exemplo, pode-se considerar uma regra para dividir uma fachada em quatro andares e uma borda entre o primeiro e segundo andar (MÜLLER, 2006):

I: fac -> Subdiv("Y",3.5,0.5,3,3,3){ andar1 | borda | andar2 | andar3 | andar4}

O primeiro parâmetro descreve o eixo de divisão (*“X”, “Y”,* ou *“Z”*) e os parâmetros restantes descrevem o tamanho das divisões. Entre os delimitadores { e } encontra-se lista de formas, separadas por |. Também são utilizadas regras para divisões em múltiplos eixos (*“XY”, “XZ”, “YZ”,* ou *“XYZ”*), divisões aninhadas ou combinações aninhadas de divisões e regras L-*system* (WONKA, 2006). Um processo de divisão de fachada é mostrado a partir da Fig. 4.3, neste passo a fachada é dividida sobre o eixo Y em 5 partes formando 4 andares, entre o primeiro e segundo andar é adicionado uma borda. Na Fig. 4.4 toma-se o segundo andar para a seqüência do processo de divisão da fachada, este andar é então dividido sobre o eixo X em quatro porções a fim de formar janelas separadas. Na Fig. 4.5 toma-se a janela1 que é subdividida sobre múltiplos eixos. E na Fig. 4.6 mostra a disposição final do segundo andar.

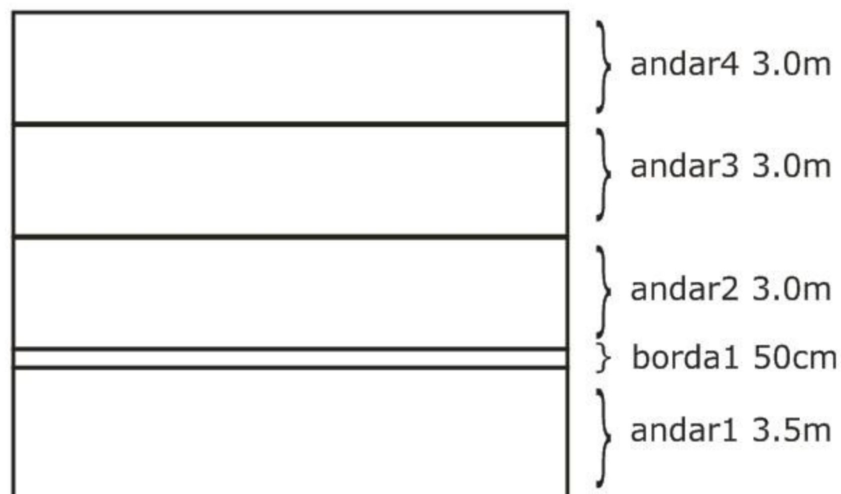


Figura 4.3 - Divisão de uma fachada sobre eixo *Y.*

Para dividir um dos andares em 4 porções iguais:

I:andar1 -> Subdiv(“X”,3.7,3.7,3.7,3.7){ janela1 | janela2 | janela3 | janela4}

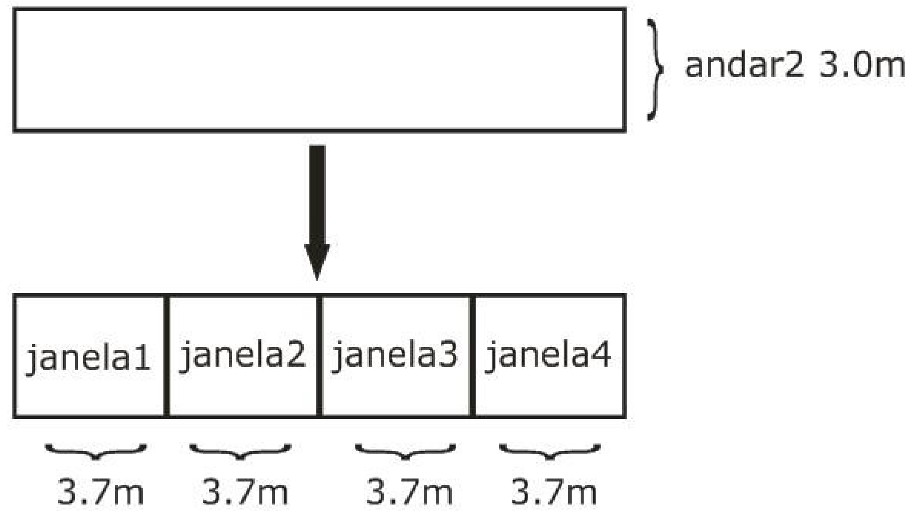


Figura 4.4 - Subdivisão da fachada sobre eixo *X.*

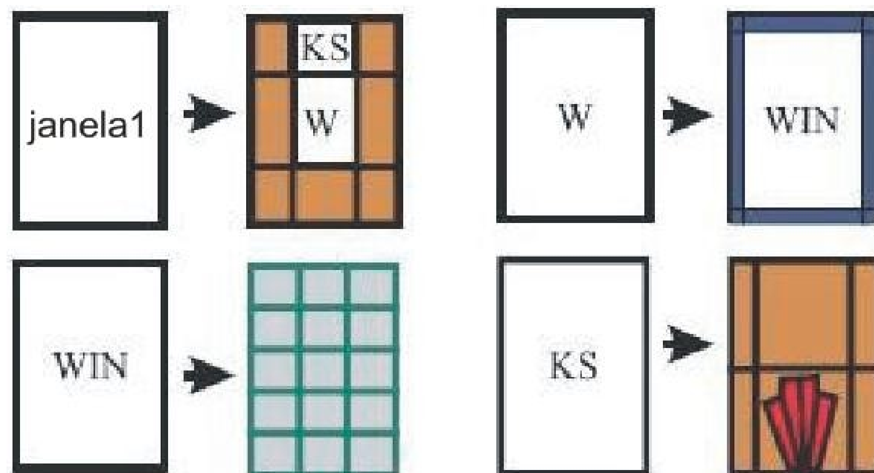


Figura 4.5 - Subdivisões em múltiplos eixos e aplicação de texturas.

Fonte: WONKA, 2003.

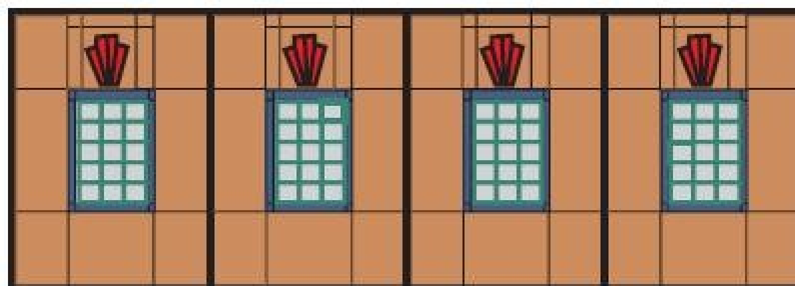


Figura 4.6 - Disposição final da fachada.

Fonte: WONKA, 2003.

4.4 Modelagem de telhado

Para mudar o perfil das construções, até então prédios comerciais apenas, pode-se adicionar mais detalhes à construção, como um telhado dando assim outro aspecto à construção. Após a geração do prédio e a obtenção da área de cobertura de sua parte superior, tem início o processo da geração de outra forma, a qual irá formar o teto (LAYCOCK, 2003). Uma abordagem simples pode ser vista na Fig. 4.7, onde é apresentada a modelagem de forma do telhado, a qual é descrita pelos seguintes passos:

1. Obter a área do teto de uma construção já pronta.
2. Determinar o esqueleto que irá formar o telhado.
3. Determinar a distância *d* que define a altura do telhado.
4. Percorrer os limites do telhado para determinar sua área.

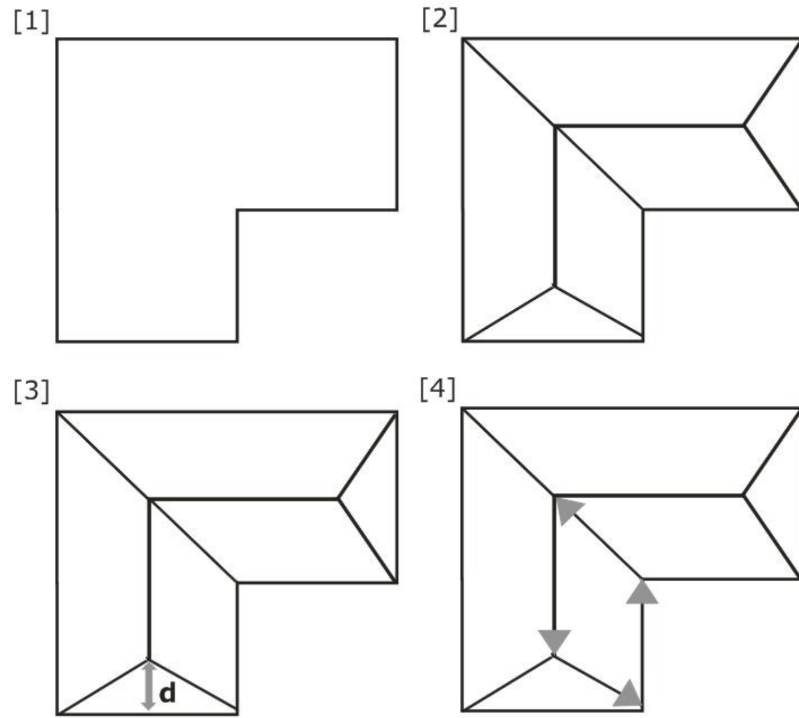


Figura 4.7 - Modelagem de telhado.

5 ESTUDO DE CASO – FERRAMENTA PARA GERAÇÃO PROCEDURAL DE CIDADE PSEUDO-INFINITA

Como parte do trabalho, o estudo das técnicas procedurais para geração de cidades pseudo-infinitas, foi criada uma ferrada para gerar uma cidade pseudo-infinita proceduralmente sob demanda do campo de visão do observador. Esta ferramenta é a aplicação de parte dos conceitos visto neste trabalho. Em sua primeira versão foram incluídos os seguintes módulos:

- Navegação pelo cenário - descrição de uma câmera virtual.
- Geração de plano de chão - baseado nas estruturas de dados. Serve como base para organização espacial do cenário.
- Geração de prédios - gerados sob demanda conforme sua localização espacial.
- Aplicação de texturas - para melhor caracterização dos prédios e cenário.
- Sistema de iluminação - compondo o visual da cidade.
- Efeitos de névoa
- Gerador de números pseudo-randômicos - utilizado para geração dos parâmetros dos procedimentos que geram os prédios.

Seu projeto foi baseado principalmente em dois fatores: as especificações da API gráfica OpenGL e o uso de algoritmos que visam otimizar às exigências de armazenamento em memória (principal motivação da Geração Procedural).

Foi implementada utilizando a linguagem C, e a OpenGL como sua extensão gráfica. Os ambientes de desenvolvimento foram as IDE’s Dev-C++ versão 4.9.9.2 e *Visual Studio* 2005.

Seu funcionamento consiste da geração de um ambiente virtual com aspecto de cidade, utilizando as estruturas de dados como controle da geração do seu plano de chão. A estrutura base do sistema é uma lista encadeada dinâmica de ponteiros do tipo FIFO contendo em cada um de seus nodos as informações relativas a um bloco de quadras. Cada bloco de quadras contém informações referentes a 11 quadrantes do plano de chão. Após determnada quantidade pré-definida de passos do observador, um nodo é adicionado à lista. Quando da adição de um nodo à lista são acionados os algoritmos de Geração Procedural para gerar as geometrias. Define-se então que cada iteração do sistema representa um nodo adicionado à lista e geração de suas respectivas construções.

5.1 Plano de chão

O plano de chão é a estrutura que organiza o cenário. Delimita os limites espaciais (sobre os eixos X e Z) das quadras e ruas da cidade. A Fig. 5.1 mostra o plano de chão, os quadrantes coloridos como sendo área destinada às construções, e as áreas claras para as ruas.

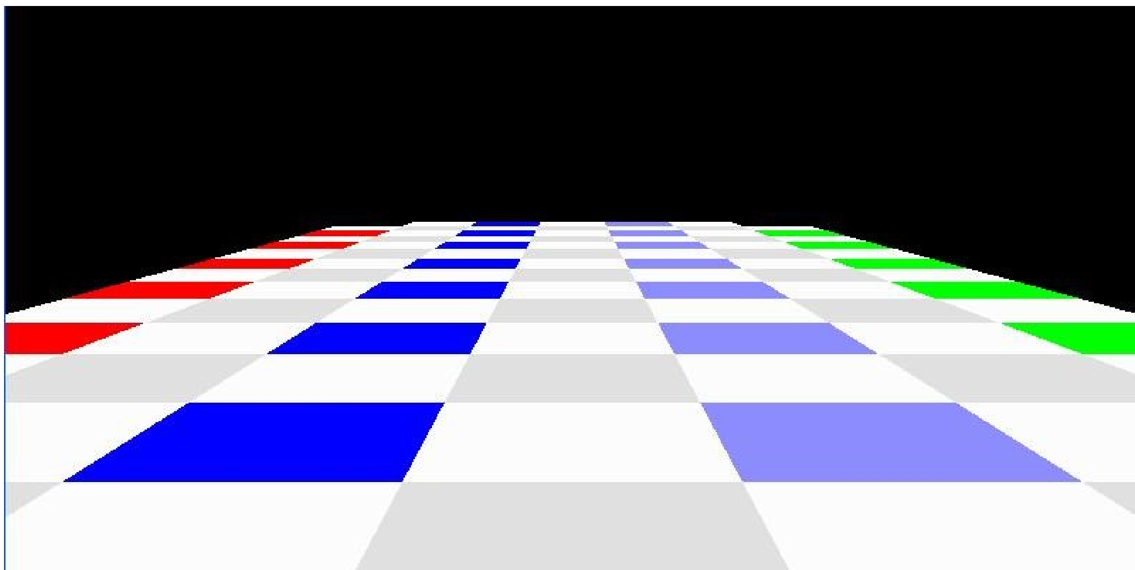
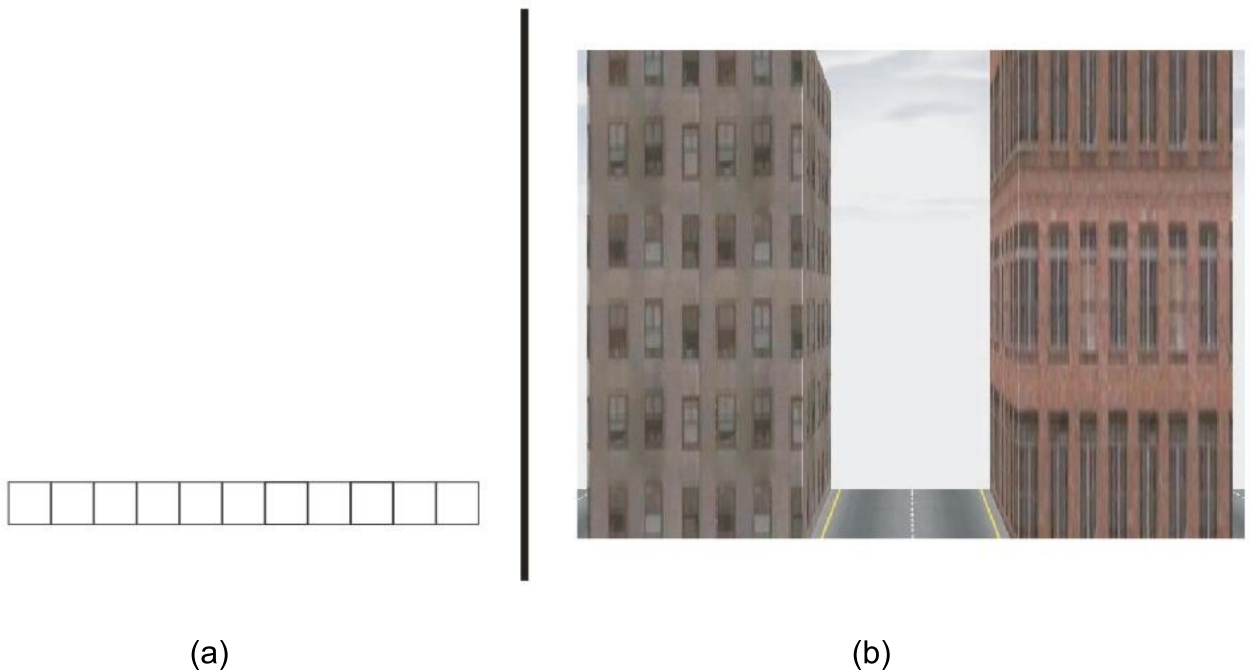


Figura 5.1 - Plano de chão.

Este plano de chão deve cobrir o campo de visão do observador. Seu formato foi idealizado para ser uma aproximação do preenchimento de cenário *view frustum filling*.

O sistema inicializa com a lista vazia, então após a 1ª iteração o cenário recebe seu prmeiro bloco de quadras, como mostra a Fig. 5.2. (b). Na Fig. 5.2. (a) é apresentado o plano de chão sobre o qual a quadra está sendo modelada. Após determinado número de passos há geração de uma nova quadra, a qual é ilustrada na Fig. 5.3. (b) com seu respectivo plano de chão (Fig. 5.3 (a)). Esta nova estrutura (quadra) adicionada, sendo então unida com o cenário já existente toda vez que for percorrido certa quantidade de passos. A Fig. 5.4 (b) mostra a cidade em sua 7ª iteração e seu respectivo plano de chão (Fig. 5.4. (a)).

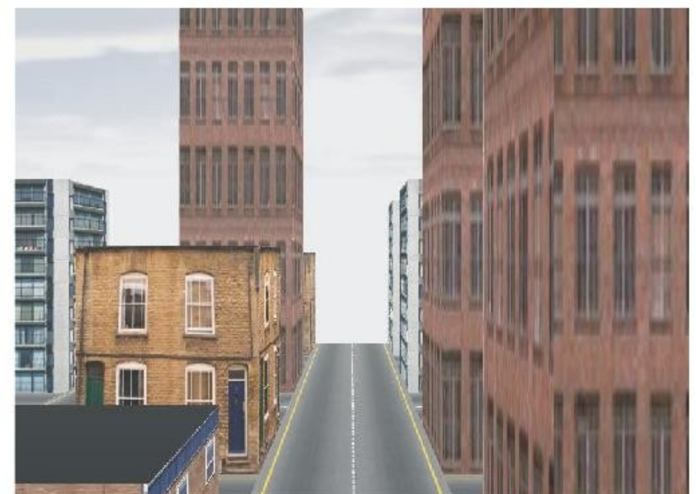


(a) (b)

Figura 5.2 - Disposição inicial do cenário, 1ª iteração.

(a) (b)

Figura 5.3 - Cenário após 2ª iteração.



(a) (b)

Figura 5.4 - Cenário após 7ª iteração.

5.2 Estruturas de dados

Possui três estruturas aninhadas que administram o tamanho do cenário, a altura, forma e posição das construções bem como suas texturas. As estruturas de dados estão disponíveis no apêndice juntamente com as rotinas que às manipulam.

5.2.1 Estruturas

Contém uma lista encadeada dinâmica de ponteiros, os quais apontam para as células que contém os dados referentes a cada estrutura do tipo quadra.

5.2.2 Matriz de coordenadas de uma quadra

O campo *float pos[11][4]*, é o que possui as informações de localização espacial das quadras. A cada 4 passos é gerada uma nova quadra, que é armazenada na forma de uma matriz 11x4. De acordo com as especificações do *OpenGL*, a primitiva *glBegin(GL_QUADS);* define para cada 4 pontos o desenho de um quadrilátero (Fig. 5.5); Sendo assim, cada linha da matriz é preenchida com essas coordenadas.

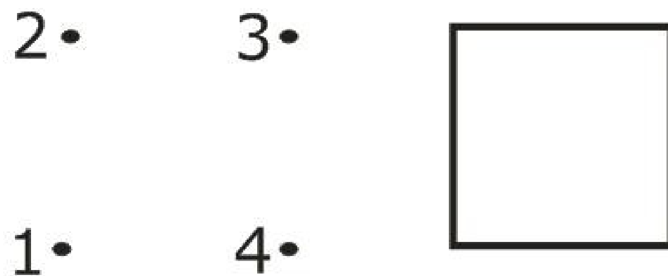


Figura 5.5 - Formato de desenho da primitiva GL_QUADS.

Cada célula tem então 4 coordenadas que definem sua posição no espaço. Como as coordenadas sobre o eixo Z não se alteram, são armazenadas somente as coordenadas do eixo X. Na matriz são representadas conforme ilustrado na Tabela 5.1 (exemplo para 4ª iteração do sistema):

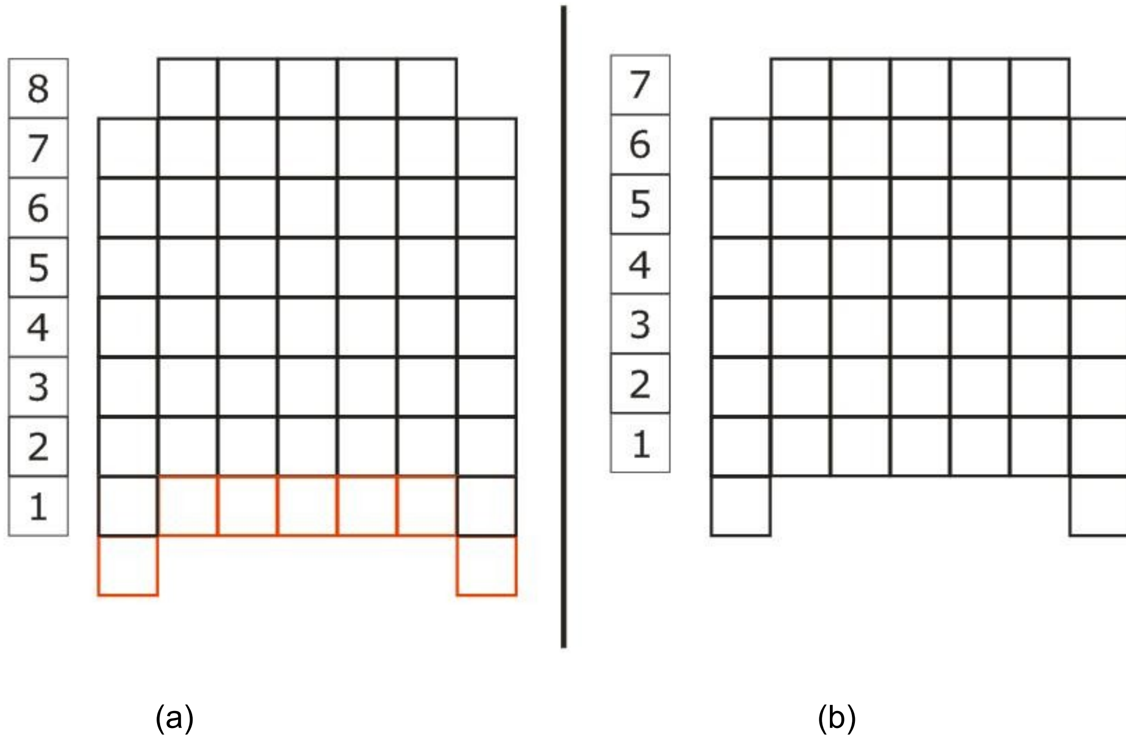
Tabela 5.1 - Matriz de coordenadas de uma quadra.

	Coordenadas sobre o eixo X			
Células	└	┌	┐	┘
1	6	7	7	6
2	6	7	7	6
3	7	8	8	7
4	7	8	8	7
5	7	8	8	7
6	7	8	8	7
7	7	8	8	7
8	7	8	8	7
9	7	8	8	7
10	7	8	8	7
11	7	8	8	7

Estas coordenadas além de delimitar os limites no espaço de cada célula, também fornecem os valores de entrada para os algoritmos de Geração Procedural, como da construção dos prédios.

5.3 Controle do tamanho do cenário

A fim de preservar memória, partes não visíveis do cenário não necessitam ficar alocados. São adotadas estratégias para gerência das estruturas de dados dos planos de chão. Em sua primeira implementação este controle é feito através da limitação do tamanho da lista que contém as estruturas do tipo *quadra* a serem desenhadas. A Fig. 5.6 (a) apresenta uma lista de tamanho 8. A Fig 5.6 (b) mostra um exemplo para uma lista restrita ao tamanho 7, quando da iteração que irá gerar a oitava quadra, a primeira estrutura é então apagada da lista e os endereços das demais estruturas ajustados.



(a) (b)

Figura 5.6 - Exclusão de nodo da lista de quadras.

5.4 Geração das construções

As construções da cidade são geradas dinamicamente sob demanda do campo de visão do observador. Geradas de forma pseudo aleatória, os procedimentos de geração dos prédios utilizam como parâmetros a localização espacial da célula que irá conter a construção, a qual irá servir como *seed* em uma função de geração de números aleatórios. Dessa forma garantindo uma diversidade visual no cenário e consistência na geração dos prédios para uma mesma localização. De forma que o observador retorne a algum ponto e encontre a mesma construção visualizada anteriormente. As Fig. 5.7 e Fig. 5.8 mostram a cidade virtual com o campo de visão do observador preenchido.

Figura 5.7 - Cidade virtual

Figura 5.8 - Cidade virtual

Neste primeiro protótipo não foram implementados os métodos de geração de números aleatórios estudados anterioremente. O sistema como todo deveria ser adaptado para apresentar resultados satisfatórios. Então foi utilizada a função *rand()* do C. A qual garantiu diversidade razoável ao cenário final.

A função de randomização foi utilizada para controlar a altura dos prédios e a distribuição das texturas para os mesmos. As construções todas possuem forma de cubo, variando de aparência quanto à sua altura e textura. Ao todo foram utilizadas 12 texturas, exibidas na Fig. 5.9.

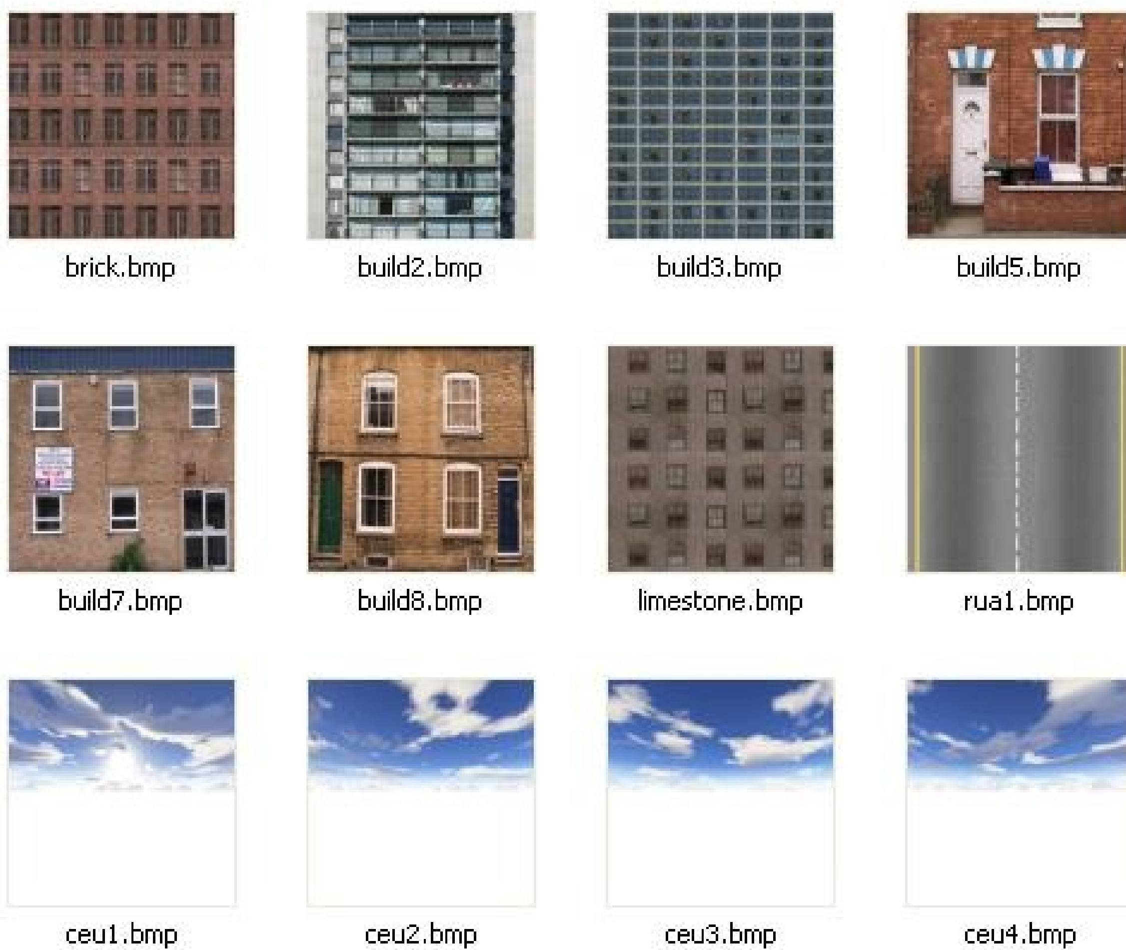


Fig. 5.9 - Texturas

6 CONCLUSÃO

As técnicas de Geração Procedural são uma alternativa muito apropriada para os paradigmas enfrentados atualmente no mundo da computação gráfica. O mercado demanda cada vez mais detalhamento gráfico, com isso os jogos e animações tendem a ocupar cada vez mais espaço. Jogos de computador primeiramente eram armazenados em disquetes, depois CDs e agora DVDs, em breve *High Density Digital Versatile Disc* - HD-DVD e *Bluray*.

Os jogos hoje em dia, em termos de *hardware* são limitados pelo poder da CPU, porém não são mais limitados pelas *Graphics Processing Unit* - GPU's. Os avanços de performance das CPU's não acompanhou o desenvolvimento das GPU's. Poder de processamento das GPU aumentou expressivamente nos últimos anos, arquiteturas paralelas permitem processamento de grandes quantidades de dados e processamento rápido. Jogos atuais tendem a ser limitados pelas exigências de memória e de processamento aritmético (cálculos de sombra e luz). GPUs podem consumir essa vasta quantidade dados, produzindo imagens de alta qualidade.

Ferramentas para Geração Procedural de geometrias são uma solução inteligente para este requisito. Gramáticas de produção fornecem um método para amplificação dos dados. Sendo assim aumentando a produtividade das ferramentas e reduzindo custos na etapa de desenvolvimento.

Nesse trabalho, foi realizado um estudo das técnicas de Geração Procedural, o qual serviu de base para a geração de ambientes que simulem o aspecto de uma cidade. Também teve como resultado, uma ferramenta para a Geração Procedural e Visualização de uma cidade virtual pseudo-infinita. Uma abordagem procedural configurada através de diversos parâmetros é usada em conjunto com diferentes classes de algoritmos para gerar uma variedade de prédios e ruas que simulam a aparência de uma cidade. Para incrementar a diversidade, prédios individuais são

gerados no ambiente a medida que são encontrados pelo usuário, resultando numa cidade que se expande sobre demanda. A ferramenta contém os seguintes módulos:

- Navegação pelo cenário - descrição de uma câmera virtual.
- Geração de plano de chão - baseado nas estruturas de dados. Serve como base para organização espacial do cenário.
- Geração de prédios - gerados sob demanda conforme sua localização espacial.
- Aplicação de texturas - para melhor caracterização dos prédios e cenário.
- Sistema de iluminação - compondo o visual da cidade.
- Efeitos de névoa
- Gerador de números pseudo-randômicos - utilizado para geração dos parâmetros dos procedimentos que geram os prédios.

Foi alcançado os objetivos do projeto, porém pode ainda ser otimizado em diversos pontos. Como a implementação de uma gramática de formas para uma maior diversidade na geração de prédios e a implementação de um L-*system* estendido para gerar padrões de ruas diferentes, ficando assim como proposta para trabalhos futuros.

7 REFERÊNCIAS BIBLIOGRÁFICAS

.theprodukkt. [2004] Disponível em: <http://212.202.219.162/> Acesso em: 02 jul. 2006, 00:00:00.

ANGEL, Edward. **Interactive Computer Graphics:** A top-down approach using OpenGL. 3.ed. San Francisco EUA: Addison Wesley, 2003. 721p.

An Introduction to Lindenmayer Systems. Site construído por Gabriela Ochoa, [1998] Disponível em: <http://www.biologie.uni-hamburg.de/b-online/e28_3/lsys.html> Acesso em: 10 jul. 2006, 02:00:00.

BERNDT, Rene; FELLNER, Dieter W; HAVEMANN, Sven ; Generative 3D models: a key to more information within less bandwidth at higher quality**.** 3D TECHNOLOGIES FOR THE WORLD WIDE WEB. ROCEEDINGS OF THE TENTH INTERNATIONAL CONFERENCE ON 3D WEB TECHNOLOGY Bangor, United Kingdom SESSION: Functional representation. 2005, p. 111-121.

BIRCH, P. BROWNE, S; JENNINGS, V; Rapid Procedural Modelling of Architectural Structures, *VAST2001 PROCEEDINGS, VIRTUAL REALITY, ARCHAEOLOGY AND CULTURAL HERITAGE*, Atenas, 2001, p. 187-196.

BUSS, Samuel R. **3-D Computer Graphics:** A Mathematical Introduction with OpenGL. Cambridge UK: Cambridge Press, 2003. 397p.

CHASE, Scott C. Design Modeling With Shape Algebras and Formal Logic**.** *ACADIA ’96 PROCEEDINGS*, Tucson, AZ, 31 out–3 nov. 1996.

CHOMSKY, Noam. **Tree models for the description of language.** Cambridge, EUA. 1956.

DANAHER, M. 2002. Dynamic landscape generation using page management. 10-TH INTERNATIONAL CONFERENCE IN CENTRAL EUROPE ON COMPUTER GRAPHICS, VISUALIZATION AND COMPUTER VISION ’2002 - WSCG 2002, p.135–138.

DELOURA, Mark A. **Game programming gems** 2.ed. Hingham, Mass: Charles River Media, 2001. 575p.

DILORENZO, P. C., ZORDAN, V. B., TRAN, D., Interactive animation of cities over time, 17TH INTERNATIONAL CONFERENCE ON COMPUTER ANIMATION AND SOCIAL AGENTS (CASA), Genebra, Suíça; 2004.

DÖLLNER, Jürgen; BUCHHOLZ, Henrik; Continuous Level-of-Detail Modeling of Buildings in 3D City Models**.** GEOGRAPHIC INFORMATION SYSTEMS, PROCEEDINGS OF THE 13TH ANNUAL ACM INTERNATIONAL WORKSHOP ON GEOGRAPHIC INFORMATION SYSTEMS, SESSION: Virtual reality and 3D, Bremen, Alemanha, 2005, p. 173 – 181.

Dynamical Systems and Fractals Lecture Notes. [1996] Disponível em: <http://www.math.okstate.edu/mathdept/dynamics/lecnotes/lecnotes.html>Acesso em: 02 jul. 2006, 01:10:00.

EBERT, David S. et al. **Texturing & Modeling:** A Procedural Approach. 3.ed. San Francisco EUA: Morgan Kaufmann Publishers, 2003. 722p.

FTÁČNIK M., BOROVSKÝ P; SAMUELČÍK M.: Low cost high quality 3D virtual city models, CORP CONFERENCE, Vienna 2004.

GREUTER, Stefan; PARKER, Jeremy; STEWART, Nigel. LEACH, Geoff; Undiscovered Worlds - Towards a Real Time Procedural World Generation Framework - MELBOURNEDAC - 5TH INTERNATIONAL DIGITAL ARTS AND CULTURE CONFERENCE, Melbourne Australia 19 mai 2003.

GREUTER, Stefan; STEWART, Nigel. LEACH, Geoff; Beyond the Horizon - Computer generated, three-dimensional, infinite virtual worlds without repetition in real-time - IMAGE TEXT AND SOUND CONFERENCE 2004

GREUTER, Stefan, PARKER, STEWART, Nigel. LEACH, Geoff, Real-time Procedural Generation of `Pseudo Infinite' Cities, GRAPHITE 2003 - INTERNATIONAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES IN AUSTRALASIA AND SOUTH EAST ASIA, p. 87-94, 2003.

INTRODUÇÃO A COMPUTAÇÃO GRÁFICA. ESPERANÇA, Cláudio, [200-], 23 slides.

INTRODUÇÃO À COMPUTAÇÃO GRÁFICA 3D. Porto Alegre: FREITAS, Carla M. D. S, [2003], 41 slides.

LAYCOCK, R.G; DAY, A.M., **Automatically generating large urban environments based on the footprint data of buildings**, ACM Solid Modelling 2003, Seattle, USA, 2003, p 346-351.

LAYCOCK, R.G; DAY, A.M., Automatically generating roof models from building footprints, WSCG, Pilsen, 2003.

LECHNER T.; WATSON B.; REN P.; WILENSKY U.; TISUE S.; FELSEN M.; **Procedural Modeling of Land Use in Cities.**

LECHNER, T; WATSON, B.A; WILENSKY, U; Proceduring city modeling. 1ST MIDWESTERN GRAPHICS CONFERENCE (St. Louis, MO), 2003.

LECHNER, T; WATSON, B.A; TISUE, S; WILENSKY, U; FELSEN, M; 2004. **Procedural modeling of land use in cities**. Technical report NWU-CS-04-38.

LECKY-THOMPSON, G. W. **Infinite game universe**: Mathematical Techniques. Charles River Media. 2001.

L-system. [200-] Disponível em: <http://en.wikipedia.org/wiki/L-systems> Acesso em: 09 ago. 2006, 00:00:00.

L-system. [2006] Disponível em: <http://en.wikipedia.org/wiki/L-systems> Acesso em: 10 ago. 2006, 16:33:00.

L-Systems. Site construído por David J. Wright, [1996]. Disponível em: <http://www.math.okstate.edu/mathdept/dynamics/lecnotes/node12.html#SECTION00040000000000000000> Acesso em: 10 ago 2006, 15:00:00.

MACRI, D., AND PALLISTER, K. **Procedural 3D content generation**. Tech. rep., Intel Developer Service. http://developer.intel.com. 2000.

MANDELBROT, B. **Fractale: Form**, Chance and Dimension. W.H. Freeman and Co. 1977.

MARTIN, Jess. **Algorithmic Beauty of Buildings Methods for Procedural Building Generation**, 2005. 64f. Tese (Graduação Ciência da Computação) Faculdade de Ciência da Computação de Trinity, San Antonio EUA.

MATSUMOTO, Makoto; NISHIMURA, Takuji; **Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator**, ACM Transactions on Modeling and Computer Simulation, v. 8, n. 1, Janeiro 1998, p. 3-30.

MÿECH, R., PRUSINKIEWICZ, P. **Visual models of plants interacting with their environment**. Proceedings of ACM SIGGRAPH 96, ACM Press, H. Rushmeier, 1996. p. 397.410.

MCBRYDE, Michael J.; **Generatinon of office buildings in large scale virtual worlds**, 2005. 84 p. Tese (Graduação Ciência da Computação) Faculdade de Ciência da Computação de Trinity, San Antonio EUA.

MÜLLER, Pascal. Procedural modeling of buildings. ACM TRANSACTIONS ON GRAPHICS (TOG). v. 25, Issue 3, New York ACM Press, jul. 2006.

MUELLER, Pascal, WONKA, Peter, HAEGLER, S.; ULMER, A; L. VAN GOOL.. ***Procedural Modeling of Buildings***. Proceedings of ACM SIGGRAPH 2006 / ACM Transactions on Graphics (TOG), ACM Press, 2006, v. 25, n. 3, p. 614-623.

NEIDER J; WOO M; DAVIS T. **OpenGL programming guide:** the official guide to learning OpenGL. Version 1.1. Upper Saddle River EUA, 1997.

OPPENHEIMER, P. E. **Real time design and animation of fractal plants and trees**. Computer Graphics (Proceedings of ACM SIGGRAPH 86), ACM, 55–64. 1986

PALLISTER, K. **Generating procedural clouds in real time on 3D hardware**. Tech. rep., Intel Developer Service. http://developer.intel.com. 2000.

PERLIN, K. 1985. **An image synthesizer**. B. A. Barsky, ed. Computer raphics (SIGGRAPH '85 Proceedings), 1985. p. 287–296.

OpenGL [200-] Disponível em: <http://en.wikipedia.org/wiki/Opengl> Acesso em: 01 set. 2006. 23:00:00.

PIPELINE GRÁFICO. Setúbal, Portugal: SANTOS, Carlos P. 2005, 12 slides.

Procedural Generation. [2006] Disponível em: <http://www.roadsafar.com/academic/procedural> Acesso em: 01 jul. 2006, 17:00:00.

Procedural Generation. [200-] Disponível em: <http://en.wikipedia.org/wiki/Procedural_generation> Acesso em: 10 ago. 2006, 22:10:00.

PRUSINKIEWICZ, P., AND LINDENMAYER, A. **The Algorithmic Beauty of Plants. Springer-Verlag.** 1991**.**

PRUSINKIEWICZ, P., JAMES, M., AND MÿECH, R. 1994. **Synthetic topiary**. Proceedings of ACM SIGGRAPH 94, ACM Press, A. Glassner, p. 351.358.

Rendering (computer graphics). [200-] Disponível em: <http://en.wikipedia.org/wiki/Renderer> Acesso em: 02 set. 2006. 00:00:00.

RUTTER, Jonathan. **Growing a 3D world.** 2004**.** 63f. Tese (Mestrado em software systems technology) Departamento de Ciência da Computação Universidade de Sheffield, Sheffield, UK.

Shape Grammars. [200-] Disponível em: <http://www.shapegrammar.org/> Acesso em: 05 ago. 2006, 18:00:00.

SHLYAKHTER, I., ROZENOER, M., DORSEY, J., AND TELLER, S. 2001. **Reconstructing 3D tree models from instrumented photographs**. IEEE Computer Graphics and Applications. 21 mai - 3 jun, p. 53.61.

STINY, G, ***Pictorial and Formal Aspects of Shapes and Shape Grammars*** (Birkhauser, Basel, Switzerland), 1975

SUN, Jing; BACIU, George; YU, Xiaobo; GREEN, Mark; Template-based generation of road networks for virtual city modeling VIRTUAL REALITY SOFTWARE AND TECHNOLOGY ARCHIVE PROCEEDINGS OF THE ACM SYMPOSIUM ON VIRTUAL REALITY SOFTWARE AND TECHNOLOGY TABLE OF CONTENTS HONG KONG, CHINA SESSION: Modeling/simulation table of contents, 2002, p. 33-40.

TAKASE, Y., SHO, N., SONE, A., SHIMIYA, K. Automatic Generation of 3D City Models And Related Applications, INTERNATIONAL ARCHIVES OF THE PHOTOGRAMMETRY, REMOTE SENSING AND SPATIAL INFORMATION SCIENCES, v. XXXIV-5/W10, 2003

ZACHMANN, G.; LANGETEPE E.; **Geometric Data Structures for Computer raphics**. Wellesley, Mass. EUA : A K Peters, 2006.

YAP, C. **The Other Manhattan Project.** Project description. http://www.cs.nyu.edu/visual/home/proj/manhattan, 1998.

WANG, T. **Integer hash function**. Tech. Rep., HP Enterprise java lab. [2000]

HTTP://WWW.CONCENTRIC.NET/_TTWANG/TECH/INTHASH.HTM.

WONKA, Peter, WIMMER, Michael, SILLION, Francois; RIBARSKY, William; **Instant Architecture** ACM Transactions on Graphics. v. 22. n. 3. p. 669 - 677. Jul 2003. apresentado em SIGGRAPH 2003.

APÊNDICES

APÊNDICE A – Estruturas de dados do estudo de caso

Lista

Define a lista de quadras do cenário. Consiste de uma estrutura do tipo lista encadeada FIFO (First In First Out).

- Celula *pPrimeiro - Ponteiro para primeiro nodo da lista.
- Celula *pUltimo - Ponteiro para ultimo nodo da lista.

```
typedef struct _lista
    {
    Celula *pPrimeiro, *pUltimo;
    } Lista;
```

Célula

São os nodos da lista.

- Quadra coord - Elemento a ser armazenado.
- Celula *pProxCelula - Ponteiro para o próximo elemento.

```
typedef struct _cel Celula;
struct _cel
    {
    Quadra coord;
    Celula *pProxCelula;
    };
```

Quadra

Estrutura que contém os dados referentes a um bloco de quadras.

- float[11][4] - Determina a posição de cada construção.
- int orX[3] - Define a largura da quadra, sobre o eixo x.
- float altura[6] - Determina a altura das construções.
- Int textura[6] - Determina a textura da construção.

```
typedef struct _quadra
       {
       float pos[11][4];
       int orX[3];
        float altura[6];
        int textura[6];
       } Quadra;
```

APÊNDICE B – Rotinas do estudo de caso

Rotação

Função para a rotação da câmera.

```
void Rotacao( void )
```

Inicializa Coordenadas

Inicializa as principais variáveis do sistema, Chamada da função:

```
void inicializaCoordenadas()

glMatrixMode(GL_PROJECTION);

glLoadIdentity();

glFrustum(camLeft,camRight,-1.0,1.0,camNear,camFar);

Rotacao();
```

- Especifica o sistema de coordenadas:

```
glMatrixMode(GL_MODELVIEW);
```

- Inicializa o sistema de coordenadas de projeção:

```
glLoadIdentity();
```

- Especifica a posição do observador e do alvo:

```
gluLookAt(posCameraX,posCameraY,posCameraZ,observacaoX+posCameraX,posCame
raY,observacaoZ+posCameraZ,0.0,1.0,0.0);
```

- Habilita o teste de profundidade:

```
glEnable(GL_DEPTH_TEST);
```

- Habilita o mapeamento de textura:

```
glEnable(GL_TEXTURE_2D);
```

- Especifica o modelo de shade das texturas:

```
glShadeModel(GL_SMOOTH);
```

Inicializa Lista

Inicializa a lista de quadras, apontando os ponteiros do primeiro e ultimo elemento para NULL.

```
void inicializaLista (_lista *pLista)
```

Lista Vazia

Testa se a lista está vazia

```
int listaVazia(Lista listaVazia)
```

Insere nodo

Insere novo nodo na lista, após o ultimo nodo.

```
int insereLista (Lista *pLista)
```

Remove nodo

Remove nodo do inicio da lista.

```
void removeInicioLista (Lista *pLista)
```

Define altura dos prédios

Determina a altura dos prédios, utiliza como seed para a função de geração de número aleatório a posição espacial da célula.

```
float defAlturaPredios(int seed)
```

Distribui Textura

Determina a textura de cada prédio. Sua escolha é feita baseada na altura da construção.

```
int distribuiTextura(int alturaPredios)
```

Textura

Função que faz a ligação dos bitmaps com os respectivos planos.

```
Textura();
```

Desenha cenário

Lê as estruturas armazenadas e em às desenha.

```
void desenhaCenario(void)
```

Teclado

Função de tratamento dos eventos do teclado.

```
 void Teclado( unsigned char tecla, int x, int y)
```