UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANDRÉ DE OLIVEIRA MELO

GERAÇÃO PROCEDURAL EM TEMPO REAL DE ÁRVORES 3D EM GPU USANDO GEOMETRY SHADERS

NITERÓI
2009

ANDRÉ DE OLIVEIRA MELO

GERAÇÃO PROCEDURAL EM TEMPO REAL DE ÁRVORES 3D EM GPU USANDO GEOMETRY SHADERS

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação

_____

Prof. DSc. Esteban Walter Gonzalez Clua
(Orientador)

_____

Prof. DSc. Anselmo Antunes Montenegro

_____

Profª. DSc. Aura Conci

Niterói

2009

Agradecimentos

À Universidade Federal Fluminense,

Aos professores,

À família,

À Deus, a essência do nosso viver.

Resumo

A computação gráfica de tempo real tem evoluído muito rapidamente nos últimos anos em grande parte devido ao acelerado aumento de performance das placas gráficas. Tal aumento acabou tornando as GPUs muito mais rápidas e eficientes que as CPUs para o desempenho de determinados tipos de aplicações. Isso vem tornando possível o uso em computação gráfica de tempo real de diversas técnicas antes consideradas inviáveis, como a geração procedural de modelos 3D. Este trabalho tem como objetivo aproveitar esse grande potencial das novas placas gráficas propondo um modelo de geração procedural em tempo real em GPU de árvores 3D através de L-*systems* (Lindemayer Systems). É também apresentada neste projeto uma implementação em *geometry shaders* utilizando OpenGL Shading Language do modelo proposto, além de testes, resultados e ideias de trabalhos futuros.

Palavras-chave: L-*systems*, *geometry shaders*, árvores 3D, PRNG, geração procedural, GLSL, computação gráfica de tempo real, pipeline gráfico.

Abstract

Real-time computer graphics has experienced great improvements in the last years, most of all because of the fast paced growth of graphics cards performance. This accelerated growth led the GPUs to become much faster and efficient than CPUs for certain types of applications. It has made possible the use in real time computer graphics of techniques which were considered unfeasible before, such as the procedural generation of 3D models. The purpose of this work is to take advantage of this great potential of modern graphics cards proposing a model of real-time procedural generation of 3D trees in GPU using L-systems. It is also presented in this work one implementation of the proposed model in OpenGL, using geometry shaders and programmed in OpenGL Shading Language, tests and results for evaluating its performance and ideas for future works.

Keywords: L-systems, geometry shaders, 3D trees, PRNG, procedural generation, GLSL, real-time computer graphics, graphics pipeline.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

O desenvolvimento de GPUs (Graphics Processing Units) cada vez mais poderosas e com maior capacidade de armazenamento e processamento e um grau maior de paralelismo tem viabilizado a implementação de aplicações de tempo real com tarefas cada vez mais pesadas e complexas. Ultimamente o ritmo de desenvolvimento dessas unidades de processamento gráfico tem superado o das CPUs (Central Processing Units).

Uma das aplicações que vêm se tornando viáveis consiste na geração procedural de modelos e texturas. As árvores são um caso de estrutura que pode ser gerada proceduralmente com alto grau de realismo por ter sua morfologia parecida com a de um fractal estocástico[1], uma vez que tem como principal característica a repetição de um determinado padrão de ramificação.

A geração procedural não só de árvores como plantas em geral além de outras estruturas com propriedades semelhantes são muito bem modelados por Lindemayer *systems*[2] (ou L-*systems*). A geração automática de árvores 3D através desse tipo de modelagem permite grande economia de trabalho, já que a modelagem manual ou o scanneamento desses tipos de estrutura em três dimensões é extremamente custoso e demorado.

[1]Fractal gerado por processo aleatório ao invés de determinístico.
[2]Sistema paralelo de re-escrita muito usado para a modelagem do desenvolvimento de plantas e da morfologia de organismos.

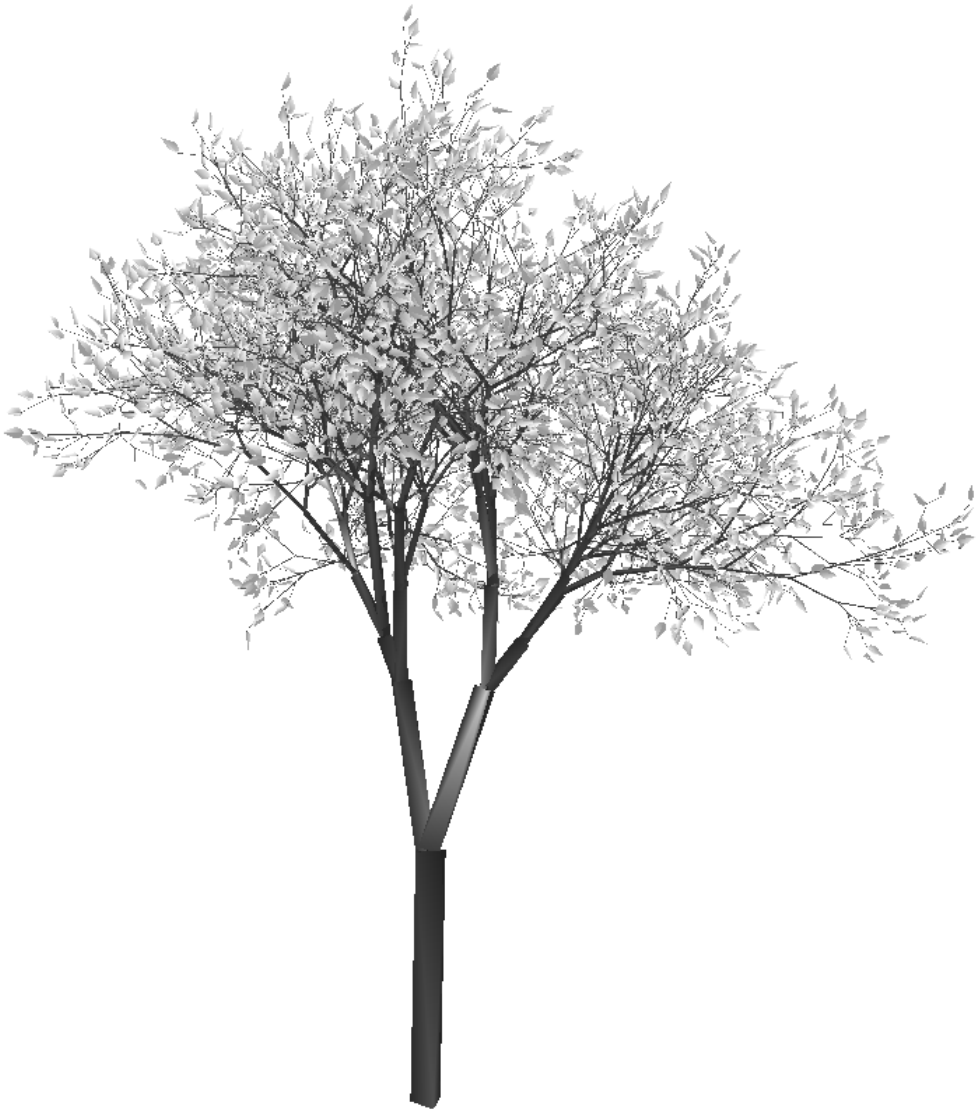

Figura 1: Exemplo de árvore 3D gerada proceduralmente

Na Figura 1 pode-se observar o alto grau de realismo de uma árvore 3D gerada proceduralmente, o que contrasta com a simplicidade dos modelos que a geram, mas apesar disso, esses modelos não são muitos utilizados em computação gráfica de tempo real, pois consomem bastantes recursos de processamento e memória, podendo assim comprometer o desempenho da aplicação.

O objetivo deste projeto é propor uma implementação da geração de árvores 3D baseados em L-*systems* em GPU utilizando *geometry shaders*[3]. A intensão é que com isso possam ser aproveitados os recursos e o alto grau de paralelismo das novas placas gráficas buscando tornar mais viável a utilização desse modelo de geração de árvores em computação gráfica de tempo real.

Procura-se com esta abordagem poder-se gerar e visualizar as árvores totalmente em GPU, diminuindo a carga de processamento necessário em CPU e na transferência de dados da CPU para GPU.

A implementação de tal abordagem traz muitos desafios, que começam pela utilização de uma tecnologia extremamente nova como *geometry shaders*, que todavia possui limitações técnicas e ainda não existem muitos trabalhos parecidos com o desenvolvido neste projeto. Além disso, as linguagens de *shader* impõem restrições que dificultam muito a implementação da abordagem proposta, uma vez que a mesma requer fundamentalmente o uso de recursão, que não é suportada na arquitetura de GPUs.

Esta monografia começa com uma breve apresentação sobre o pipeline gráfico, o que é a GPU e sua arquitetura, Lindenmayer *systems*, *geometry shaders* e geradores de números pseudo-aleatórios que estão no Capítulo 2, onde são apresentados trabalhos relacionados que foram utilizados como base para o desenvolvimento deste projeto. No Capítulo 3 é descrito em mais detalhes a ideia e teoria do projeto além de como o mesmo

[3]Programa existente no pipeline gráfico de GPUs modernas que tem como objetivo alterar modelos geométricos enviados pela aplicação principal.

foi implementado. Em seguida, no Capítulo 4, são apresentados os testes e resultados obtidos e finalmente no Capítulo 5, são apresentados as considerações finais sobre o projeto como conclusão desta monografia.

2 Trabalhos relacionados

2.1 Pipeline gráfico

A computação na GPU segue um *pipeline* gráfico. Este *pipeline* foi desenvolvido para manter altas frequências de computação através de execuções paralelas. O *pipeline* gráfico convencional é composto por vários estágios parametrizáveis via API[4]. No *pipeline* convencional (Figura 2), a aplicação envia à GPU um conjunto de vértices. Estes vértices são transformados segundo matrizes de modelagem e visualização, depois são iluminados, projetados e mapeados para a tela. Após este conjunto de operações, a GPU combina os vértices para gerar algum tipo de primitiva (ponto, linha, triangulo, etc). O rasterizador gera um fragmento para cada *pixel* que compõe a primitiva. Para cada fragmento, operações de mapeamento de textura, combinações de cores e testes de descarte podem ser feitos (CIRNE, 2008).

O resultado dessa operação é então encaminhado ao *framebuffer*, também conhecido como memória de vídeo, que se encarrega de armazenar e transferir para a tela os dados processados nos estágios anteriores do *pipeline*.

[4] Aplication Programming Interface (Interface de Programação de Aplicativos) um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por programas aplicativos que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços

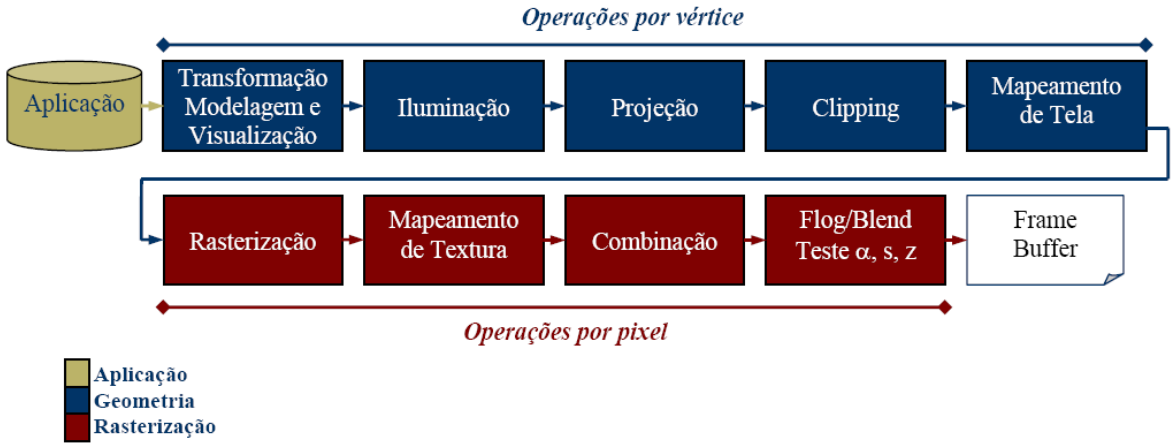

Figura 2: Estágios funcionais do pipeline (CELES, 2006)

Com o surgimento das placas gráficas programáveis, houve também uma mudança na concepção geral do *pipeline* gráfico. Alguns estágios do *pipeline* gráfico podem ser substituídos por programas que manipulam vértices e fragmentos (Figura 3). No entanto, quando um programa desse tipo é implementado, ele deve também implementar todos os estágios do *pipeline* gráfico que ele substitui.

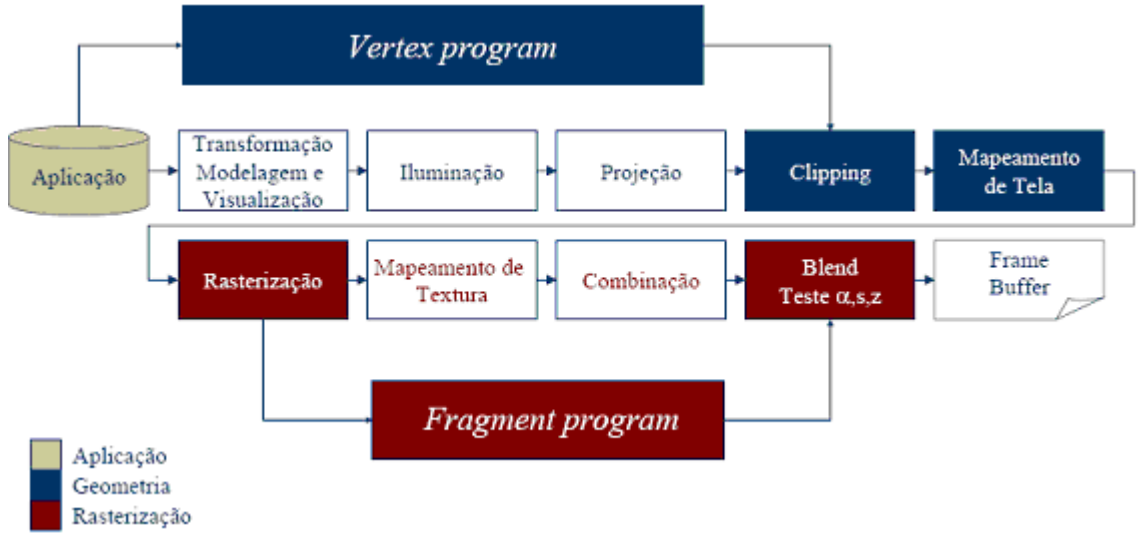

Figura 3: Estágios funcionais do pipeline programável (CELES, 2006)

Além disso, um novo estágio foi adicionado ao *pipeline* gráfico programável: o estágio de geometria, responsável por todo o processo de renderização de uma cena. Este estágio quebra o pipeline em duas partes. Uma é chamada de *Stream-Output*, responsável pela alocação de vértices oriundos do *geometry shader* em um ou mais *buffers* na memória, podendo também gerar novos vértices para o pipeline. A outra parte é a de rasterização que responsável pela geração da entrada para o *pixel shader* (CIRNE, 2008).

Dessa forma, o novo pipeline adquire os estágios de *vertex shader*, *geometry shader* e *pixel shader*, como mostra a Figura 4.

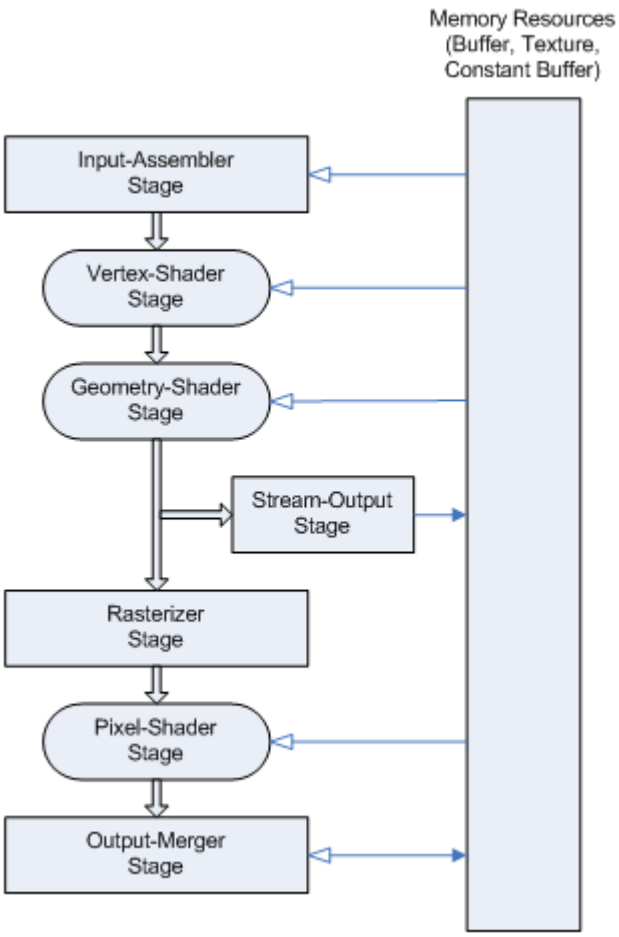

Figura 4: Módulos do pipeline gráfico programável (incluindo geometry shader)

2.2 GPU

As GPUs (do inglês Graphics Processing Unit) são processadores especializados que basicamente efetuam operações ligadas a aplicativos gráficos 3D. São usadas com frequência em sistemas embarcados, consoles de videogames, estações de trabalho, etc. Elas também são bastante eficientes na manipulação de tarefas de Computação Gráfica em geral, conseguindo obter um desempenho superior ao de CPUs de propósito geral, devido à sua estrutura altamente paralelizada e especializada no processamento de aplicações gráficas (CIRNE, 2008).

Basicamente, as GPUs são dedicadas para o cálculo de operações de ponto flutuante, destinadas a funções gráficas em geral. Os algoritmos de renderização normalmente trabalham com uma quantidade significativa dessas operações, as quais são executadas de uma maneira eficiente, graças à existência de operações matemáticas especiais.

A GPU é uma máquina paralela de processamento de *stream*. *Stream* é definido como uma sequência de dados de um mesmo tipo. Dessa maneiram, a GPU é eficiente para o processamento de aplicações que precisem fazer a computação de *streams* com grandes quantidades de elementos que sofram o mesmo tipo de operação (REIS, CONTI e VENETILLO, 2007).

Um *shader* opera sobre todos os elementos de um *stream*. Dentro desses programas, a computação de um elemento não depende dos outros elementos do *stream* e a saída produzida é função apenas dos parâmetros de entrada do programa providos por cada um dos elementos. Essa característica torna este tipo de aplicação extremamente paralelizável o que faz com que as GPUs adotem o modelo de processamento chamado de SIMD[5] (REIS, CONTI e VENETILLO, 2007).

Outra característica importante a ser ressaltada é o fato de o *hardware* das GPUs ser especializado, obtendo uma eficiência muito maior do que os *hardwares* de propósito geral (como as CPUs). Ao longo dos anos, a disparidade entre GPUs e CPUs, em relação à performance, foi se tornando cada vez maior, como mostra o gráfico da Figura 5. Uma segunda vantagem que as GPUs levam sobre as CPUs é que elas utilizam a maioria de seus transistores para a computação e muito pouco para a parte de controle, obtendo-se um poder de computação maior, mas tornando os fluxos de programas um pouco mais limitados.

[5] Single Instruction, Multiple Data, ou seja, fluxo único de instruções e múltiplo de dados.

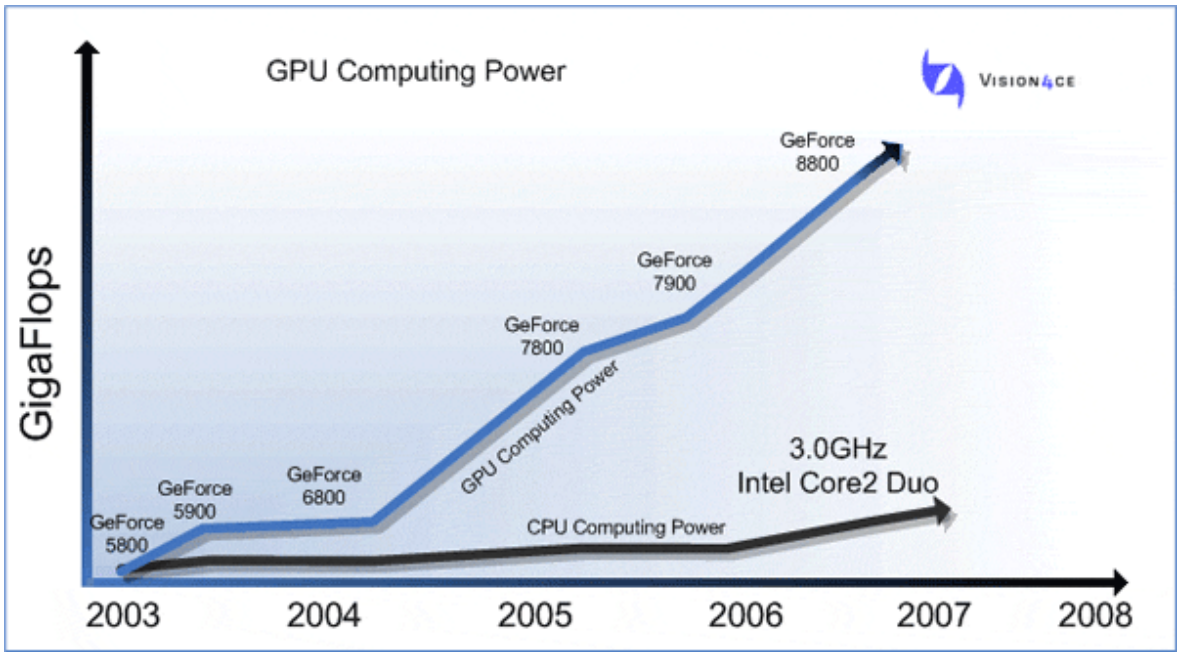

Figura 5: Gráfico comparativo entre evolução de desempenho de CPU e GPU

Outro aspecto importante a se considerar é com relação ao acesso à memória. As GPUs procuram maximizar o *throughput*, ao passo que as CPUs priorizam a latência no acesso. Isso ocorre porque a computação das GPUs tira mais proveito do chamado princípio da localidade do que a computação genérica. Nesse caso, valoriza-se mais a eficiência na transferência de um determinado conjunto de elementos, em detrimento do tempo gasto para acessá-los.

2.3 Geometry shaders

Geometry Shader é um programa que roda em GPU e que foi recentemente introduzido como uma etapa do *pipeline* gráfico das placas de vídeo atuais. Esse *shader* é executado depois do *Vertex Shader* e antes da fase de projeção e transformação.

O objetivo do *shader* em questão é gerar zero ou mais primitivas geométricas de um tipo específico como saída a partir de uma única primitiva tomada como entrada, podendo assim aumentar ou reduzir o número de vértices. Uma característica importante das linguagens de *shaders* a falta de suporte à recursão

Uma das principais aplicações atuais dos *geometry shaders* é a modificação automática de complexidade de malhas geométricas, geração de *point sprite*, *geometry tesselation* e *shadow volume extrusion*.

Atualmente, as linguagens de programação de *geometry shader* mais utilizadas são Cg[6] (NVidia), GLSL[7] (OpenGL) e HLSL[8] (Direct3D). No caso de GLSL, que é a linguagem utilizada neste projeto, existem três tipos de variáveis especiais para os *shaders*, sendo dois para comunicação entre aplicação e *shader* e um para comunicação entre *shaders*.

A comunicação aplicação-*shader* pode ser efetuada através das variáveis *uniform* e *attribute*. As primeiras são variáveis que são associadas a uma primitiva geométrica e podem ser acessadas em *vertex*, *geometry* e *fragment* shader como dados do tipo somente leitura. Já as variáveis *attribute* são associadas a um vértice e só podem ser acessadas em *vertex* e *geometry shader* também como somente leitura. Para a comunicação entre *shaders*

[6]C for graphics
[7]OpenGL Shading Language
[8]High Level Shading Language

existem as variáveis *varying*. Estas variáveis são definidas em *vertex shader* e podem ser lidas em *geometry* ou *fragment shader*.

2.4 Geradores de Números Pseudo-aleatórios

Um Gerador de Números Pseudo-aleatórios, também conhecido como PRNG (Pseudorandom Number Generator), é um algorítimo de geração de uma sequência de números que tem propriedades que aproximam às dos números aleatórios. A sequência não é verdadeiramente aleatória, ela é completamente determinada por um pequeno conjunto de valores iniciais, chamados de estado do PRNG.

Os números pseudo-aleatórios são extremamente importantes na prática de simulações e são parte crítica na criptografia e geração procedural. Em muitos casos é requerida uma cuidadosa análise dos mesmos com o objetivo de determinar o nível de segurança na geração de números suficientemente "aleatórios", ou seja, o nível de dificuldade de prever o próximo número da sequência.

Uma importante propriedade de um PRNG é a periodicidade. Cada sequência pode ser iniciada por um estado inicial arbitrário, com a definição das chamadas sementes ou *seeds*. Um mesmo PRNG sempre gerará uma mesma sequência sempre que utilizado o mesmo estado inicial. A periodicidade consiste na quantidade de números gerados até que seja repetida a sequência.

Na prática, uma sequência gerada por um PRNG pode ser distinguida de uma sequência verdadeiramente aleatória por algum dos problemas listados abaixo, que podem ser apresentados por muitos dos algorítimos existentes.

- Períodos mais curtos que o esperado para determinados estados iniciais (que são chamados de estados fracos).

- Falta de uniformidade na distribuição em grandes sequências geradas.

- Correlação entre valores sucessivos

- Distâncias entre as ocorrências de um determinado valor são distribuídas de maneira diferente das de uma sequência aleatória

2.4.1 Gerador Linear Congruencial

Um Gerador Linear Congruencial, ou LCG (Linear Congruential Generator) é um dos mais antigos e conhecidos algoritmos PRNG. A teoria por trás deste tipo de gerador é muito simples, fácil e rápida de implementar.

O gerador é definido pela seguinte relação de recorrência:

$$X_{n+1} = (aX_n + c) \bmod m$$

Onde X_n é a sequência de números pseudo-aleatórios, e os seguintes termos são constantes inteiras que especificam o gerador:

$m \mid 0 < m$ é o módulo

$a \mid 0 < a < m$ é o multiplicador

$c \mid 0 \leq c < m$ é o incremento

$X_0 \mid 0 \leq X_0 < m$ é a semente ou valor inicial

LCGs são extremamente rápidos e requerem uma quantidade muito pequena de memória, isso os tornam bastante apropriados para o processamento de múltiplos streams independentes e para sistemas com recursos limitados de processamento e memória.

Devido à sua simplicidade, este tipo de gerador não é apropriado para aplicações que requerem uma alta qualidade de "aleatoriedade" , como por exemplo simulações de Monte Carlo[9] e aplicações de criptográficas.

[9] Método estatístico utilizado em simulações estocásticas utilizado como forma de obter aproximações numéricas de funções complexas

2.5 Lindenmayer systems

L-*systems* ou Lindenmayer *systems* é um formalismo matemático criado como um fundamento para uma teoria de desenvolvimento de estruturas biológicas, que foi introduzido e desenvolvido em 1968 por biologista teórico e botânico húngaro Aristid Lindenmayer (1925-1989) da Universidade de Utrecht. Este sistema tem sido utilizado em aplicações na computação gráfica, 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 re-escrita, onde a ideia básica é definir objetos complexos pela sucessiva recolocação de geometrias utilizando um conjunto de regras de produções. A re-escrita pode ser realizada recursivamente (EBERT, 2003).

Esse processo de re-escrever pode ser observado no clássico exemplo do Floco de Neve de Koch[10] na Figura 6. A construção dessa estrutura (proposta por Koch em 1905) começa com duas formas: um objeto inicial e uma regra de produção. A regra de produção modifica o objeto inicial substituindo o predecessor da regra pelo sucessor. Essa substituição pode acontecer iterativamente por qualquer número de repetições.

[10] Niels Fabian Helge von Koch (1879,1924), matemático sueco

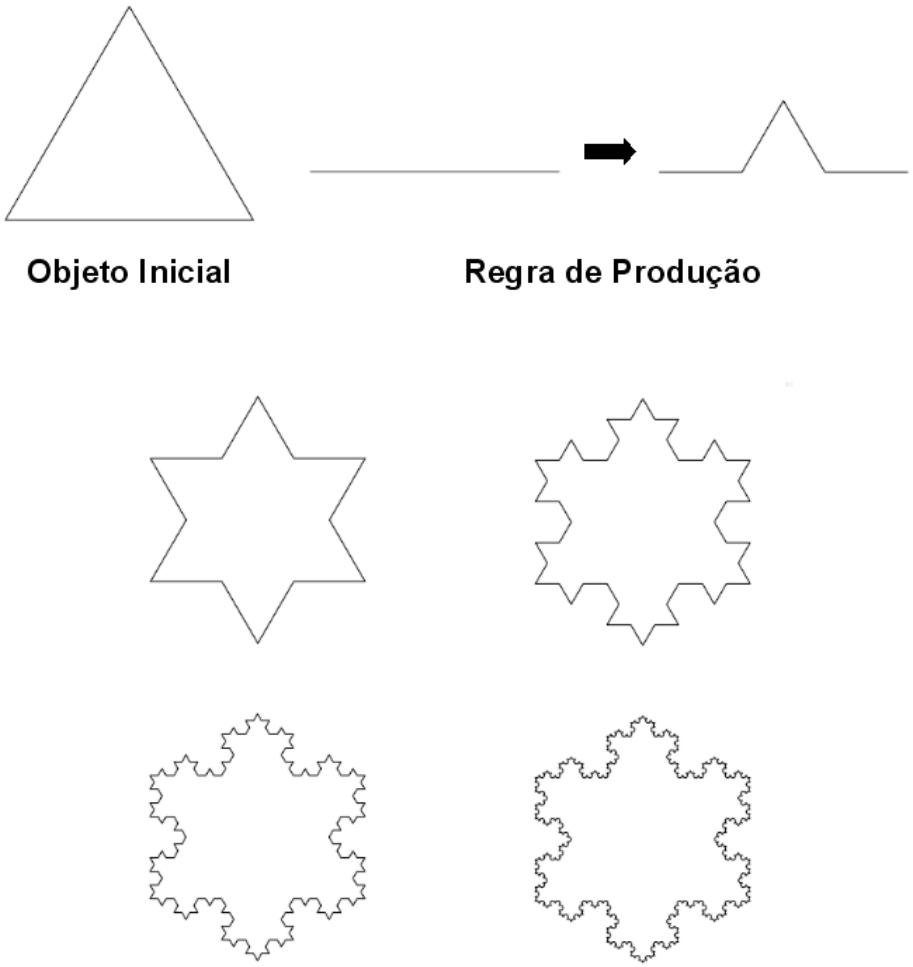

Figura 6: Floco de Neve de Koch

Aristid Lindenmayer introduziu em seu trabalho um novo tipo de re-escrita de *strings*, o L-*system*. A diferença essencial entre as gramáticas de Chomsky[11] e L-*systems* é no método pelo qual é aplicado as produções. Na gramática de Chomsky produções são aplicadas sequencialmente, 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* (EBERT, 2003).

[11] Avram Noam Chomsky (nascido em 1928), matemático norte-americano professor de linguística do MIT

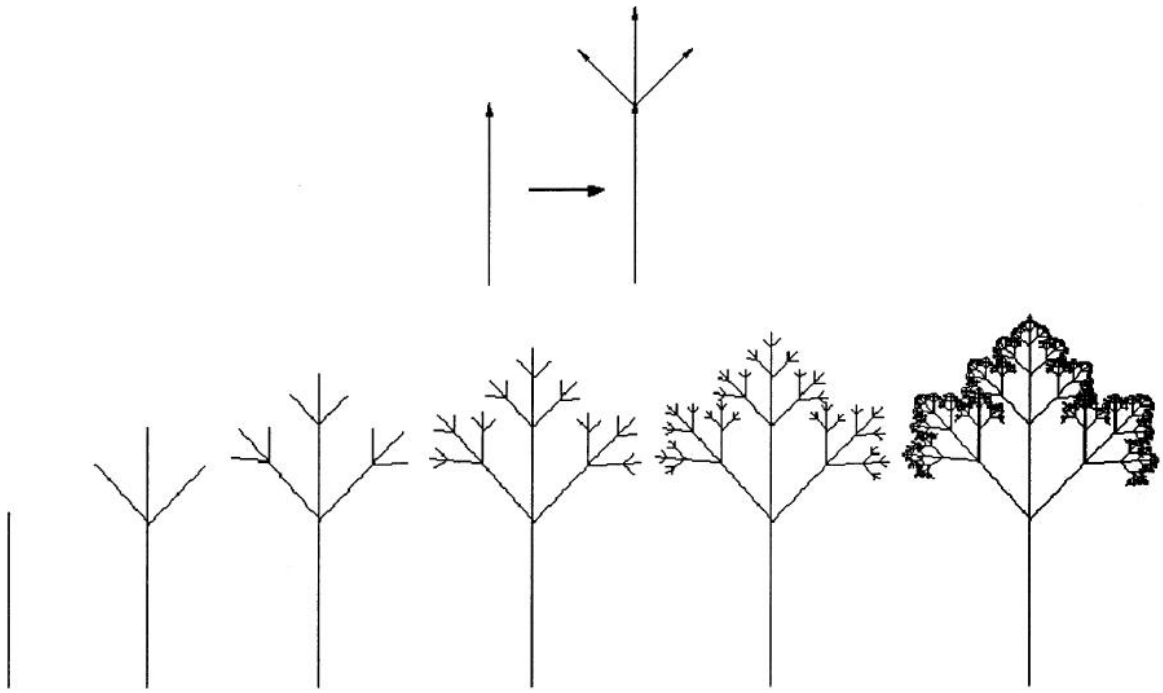

Figura 7: Exemplo simples de L-system de árvore 2D com regra e etapas de produção (MUHAR, 2001)

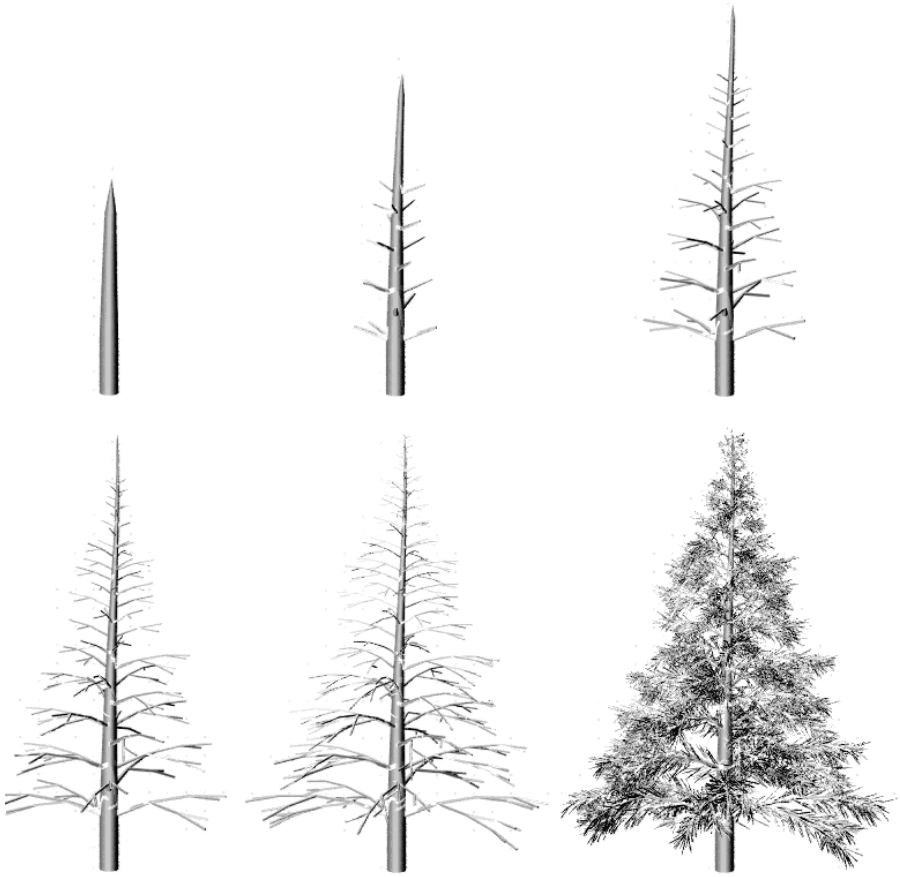

Figura 8: Exemplos de árvore 3D gerada por L-system e suas etapas de produção

A aplicação dessas regras de produção em paralelo se adequa perfeitamente na modelagem de árvores. A Figura 7 mostra um exemplo simples de uma árvore 2D com sua regra e as etapas de produção. Já a Figura 8, ilustra as etapas de produção em um exemplo mais complexo e com resultado visual bastante realista de uma árvore 3D também gerada por L-*system*. Apesar de inicialmente ter sido proposto para modelagem de plantas, os L-*systems* hoje são utilizados de maneira muito mais ampla para descrever quaisquer modelos de fractais.

A natureza recursiva dos L-*systems* os tornam ideais para descrever quaisquer objetos com características fractais, em especial diversas estruturas criadas pela natureza. Esse sistema é usado para modelar matematicamente árvores, flores, folhas, camadas celulares.

2.5.1 Parametric 0L-system

Existem várias classes de L-*systems*, porém neste artigo será utilizado apenas o *Parametric* 0L-*systems*, que é livre de contexto e como o nome já diz, suporta regras de produção com passagem de parâmetros, que pertencem ao alfabeto V.

Um *Parametric* 0L-*system* é descrito por uma gramática definida pela seguinte tupla:

$$G = \{V, \Sigma, \omega, P\}$$

onde:

- V (o alfabeto) é um conjunto não vazio de símbolos chamado de alfabeto do sistema.

- Σ é o conjunto de parâmetros formais.

- ω (início, axioma ou iniciador) é um *string* de símbolos de V que definem o estado inicial do sistema.

- P é um conjunto de normas de produção ou produções que definem a maneira que as variáveis podem ser substituídas por combinações de constantes e de outras variáveis. A produção é constituída por dois strings: o predecessor e o sucessor.

Uma produção, (a, C, x) é normalmente denotada por:

$$a : C \rightarrow x$$

Onde a é o predecessor, C é uma expressão lógica chamada de condição e x é o sucessor. Caso a condição seja vazia, a produção pode ser denotada por $a \rightarrow x$. Para uma determinada produção é assumido que um parâmetro formal não pode aparecer mais de

uma vez no predecessor e todos os parâmetros dentro da condição ou sucessor têm que aparecer no predecessor.

A evolução do L-*system* é definida pela sequência $\{g_n\}$, $n = 0, 1, 2, 3, ...$, 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.

2.5.2 A interpretação da tartaruga

Para a interpretação geométrica dos L-systems é utilizado neste artigo o baseado na tartaruga do LOGO[12], sendo constituído de operações básicas como rotação e estendido para o ambiente 3D. As operações do modelo são as seguintes:

- $+(\Theta)$: rotaciona Θ no eixo Z.
- $-(\Theta)$: rotaciona $-\Theta$ no eixo Z.
- $\&(\varphi)$: rotaciona φ no eixo Y.
- $\wedge(\varphi)$: rotaciona $-\varphi$ no eixo Y.

[12] Logic Oriented Graphic Oriented é uma linguagem de programação interpretada que envolve uma tartaruga gráfica, um robô que responde aos comandos do usuário

- \ (δ) : rotaciona δ no eixo X.

- / (δ) : rotaciona -δ no eixo X.

- | : dá "meia-volta", inverte o sentido.

- [: empilha o estado corrente (incluindo posição, direção)

-] : desempilha o estado corrente

- $F(l)$: move a tartaruga uma distancia L, considerando distancia e sentido correntes

Um estado da tartaruga é definido como uma 6-upla (x, y, z, φ, Θ, δ), onde as coordenadas cartesianas (x, y, z) representam a posição da tartaruga e os ângulos φ, Θ e δ são interpretado como a direção para qual a tartaruga está apontando.

As informações dos estados da tartaruga 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 Figura 9 o L-*system* representado pela produção:

$$F(L) \rightarrow F\,(L)\,[+(30^{o})\,F(L)]\,F(L)\,[-(30^{o})\,F(L)]\,F(L)$$

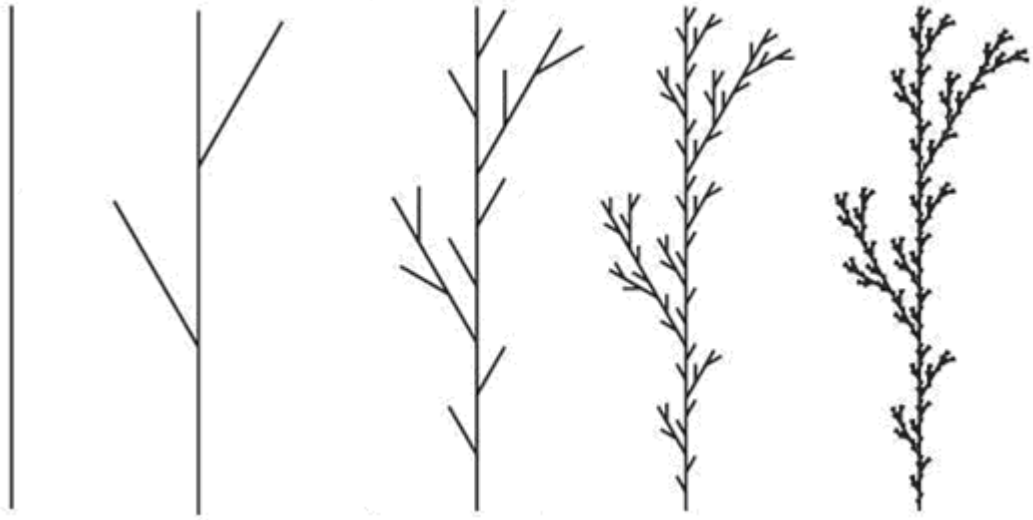

Figura 9: Exemplo de estrutura criada através do empilhamento de estados

Um exemplo de descrição de um *Parametric* 0L-*system* usando a interpretação da tartaruga é definido abaixo:

V : $\{ \varphi, \Theta, L, R, W, +, -, \wedge, l, r\}$

Σ : $\{l, r\}$

ω : $A(L,R)$

P : $A(l,r)$: $r>0 \rightarrow F(l)[+(\Theta)\wedge(\varphi)A(l*W,r\text{-}1)]\,[-(\Theta)\wedge(\varphi)A(l*W,r\text{-}1)]$

$$A(l,r): r \leq 0 \rightarrow F(l)$$

onde:

- l é o comprimento do galho a ser desenhado
- r é o contador de recursões
- φ é o ângulo de rotação no eixo Y (*pitch*)
- Θ é o ângulo de rotação no eixo Z (*yaw*)
- L é o comprimento do primeiro galho
- R é o numero de recursões da árvore
- W é o coeficiente de redução de L

A Figura 10 ilustra uma árvore gerada em OpenGL e modelada pelo *Parametric* 0L-*system* definido acima e com os seguintes atribuições de constantes:

- $\varphi = 45$
- $\Theta = 90$
- $L = 1$
- $R = 10$
- $W = 0{,}75$

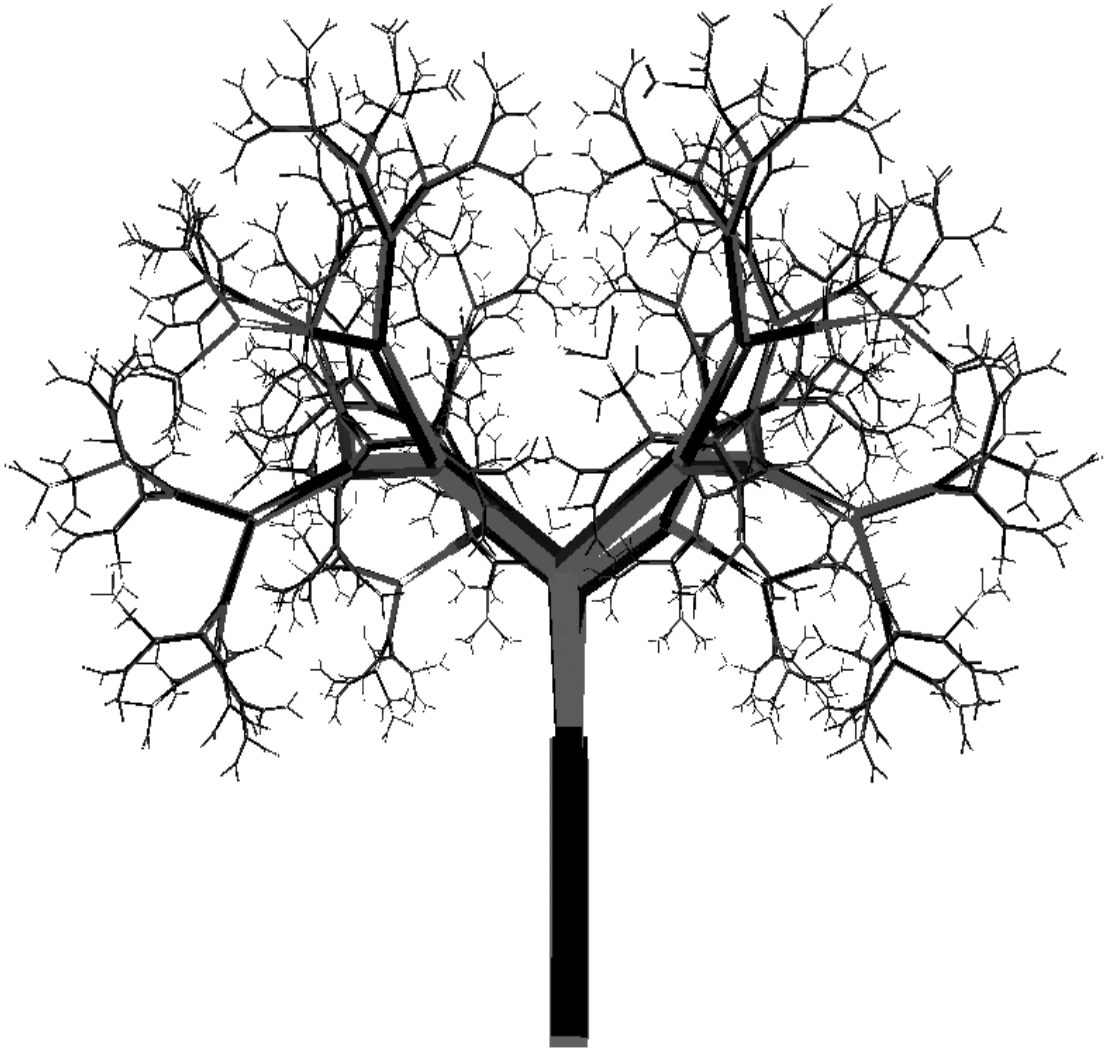

Figura 10: Árvore modelada por Parametric 0L-systems

2.5.3 Árvores aleatórias

Com o objetivo de gerar arvores com morfologias aleatórias e proporcionar um maior realismo aos modelos gerados, buscando simular os padrões irregulares encontrados na natureza, é interessante atribuir valores aleatórios dentro de um domínio especificado às variáveis.

Este tipo de modelo tem sido utilizado para simular a formação de árvores, inclusive com a utilização de desvios de simetria simulando o crescimento em um ambiente natural (KRUSZEWSKI e WHITESIDES, 1998)

Para obter-se esse efeito de aleatoriedade, basta inserir na descrição do L-*system* uma função aleatória como por exemplo:

$$rand(x,y)$$

onde $rand(x,y)$ retorna um número real aleatório maior que x e menor que y, e caso x seja maior que y retorna x.

Esta função *rand* então pode substituir variáveis ou constantes na parte direita das regras de produção da gramática de um L-*system*. Dessa maneira, pode-se por exemplo tornar aleatório o comprimento dos galhos de uma árvore substituindo $F(l)$ por $F(rand(0,l))$, fazendo-se assim que o valor atribuído ao comprimento seja um número real aleatório entre 0 e l.

Isso também pode ser utilizado para todos os atributos de uma árvore e até mesmo na passagem de parâmetros de um *Parametric* L-*system*. A Figura 11 mostra um exemplo de L-system que inclui a aleatoriedade na atribuição de tamanho e da direção de crescimento de cada ramo do arbusto 2D.

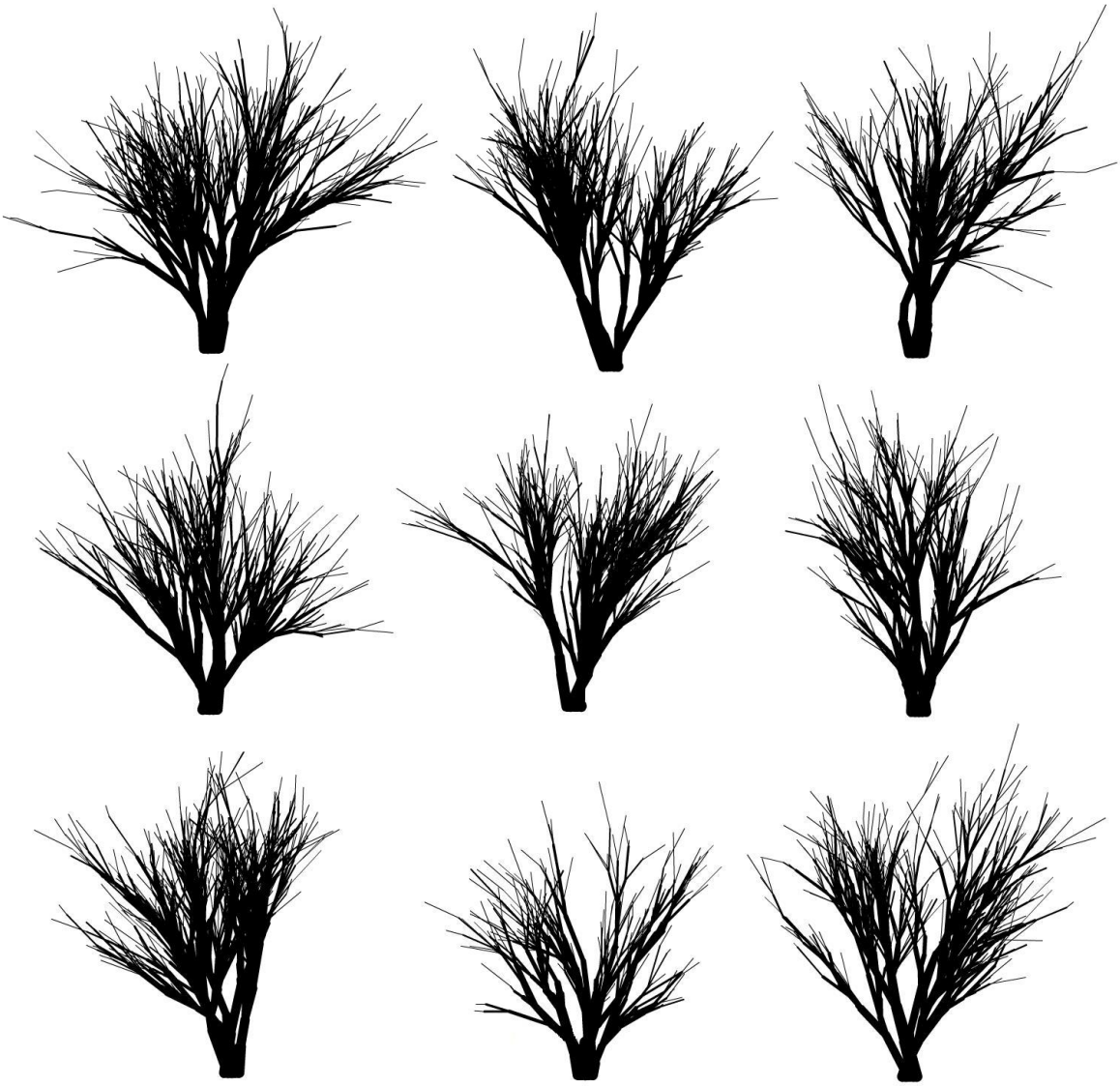

Figura 11: Arbustos gerados aleatoriamente por um mesmo modelo[13]

[13] Cada arbusto da figura obtida na aplicação Flash da página http://bendapkiewicz.com/flash/tree.html

3 Geração procedural em tempo real de árvores 3D em GPU

3.1 Visão geral

A ideia do projeto é implementar a geração de árvores 3D em *geometry shader*, transferindo para a GPU essa tarefa que normalmente deveria ser efetuada pela CPU. Dessa maneira, as árvores passam a ser geradas e visualizadas totalmente em GPU, diminuindo a carga de processamento necessário em CPU e na transferência de dados da CPU para GPU.

Mais especificamente, o programa principal deixa gerar proceduralmente a árvore a cada *frame* e emitir todos os seus vértices para então passar-los para a GPU que executaria todo o pipeline gráfico para apenas gerar uma primitiva simples acompanhada de parâmetros que possuam todas as informações necessárias para que a GPU efetue todo o trabalho de geração da árvore.

O modelo proposto suporta árvores aleatórias, implementando em *geometry shader* um PRNG que gerará a sequência pseudo-aleatória de atributos de cada galho da árvore. Dessa maneira, a aplicação principal envia as sementes de cada árvore com parâmetro para a GPU que se encarrega do cálculo dos demais termos da sequência. Com isso, é possível que toda uma floresta formada por árvores diferentes entre si, mas que sigam um mesmo modelo de L-*system*, seja armazenada no programa principal mantendo apenas as sementes de PRNG de cada árvore em memória.

Um ponto importante a ser levado na implementação é o fato das linguagens de *shader* não suportarem recursão. Como a geração procedural de árvores é de natureza essencialmente recursiva, a recursão tem que ser simulada iterativamente no próprio programa do *shader*.

No decorrer deste capítulo, o projeto é explicado mais detalhadamente além de também apontar todos os os problemas encontrados e soluções apresentadas durante o desenvolvimento do projeto.

3.2 O modelo de árvore utilizado

O modelo de árvore 3D utilizado na implementação deste projeto é a versão suportando aleatoriedade de um *Parametric* 0L-*system*, e é representado da seguinte maneira:

V: $\{\varphi, \Theta, L, R, W, +, -, \wedge, l, r\}$

Σ: $\{l, r\}$

ω: $A(L,R)$

P: $A(l,r)$: $r>0 \rightarrow F(rand(0,l))[+(rand(\Theta_{min},\Theta_{max}))\hat{}(rand(\varphi_{min}, \varphi_{max}))A(l*W,r\text{-}1)]$

$[- (rand(\Theta_{min},\Theta_{max}))\hat{}(rand(\varphi_{min}, \varphi_{max}))A(l*W,r\text{-}1)]$

$A(l,r)$: $r \leq 0 \rightarrow F(rand(0,l))$

A Figura 12 mostra quatro exemplos de árvores geradas a partir do modelo descrito acima implementado em OpenGL e obtido com os seguintes valores de constantes:

- $L = 1$
- $R = 10$
- $W = 0{,}85$
- $\Theta_{min} = 0°$
- $\Theta_{max} = 90°$
- $\varphi_{min} = 5°$
- $\varphi_{max} = 45°$

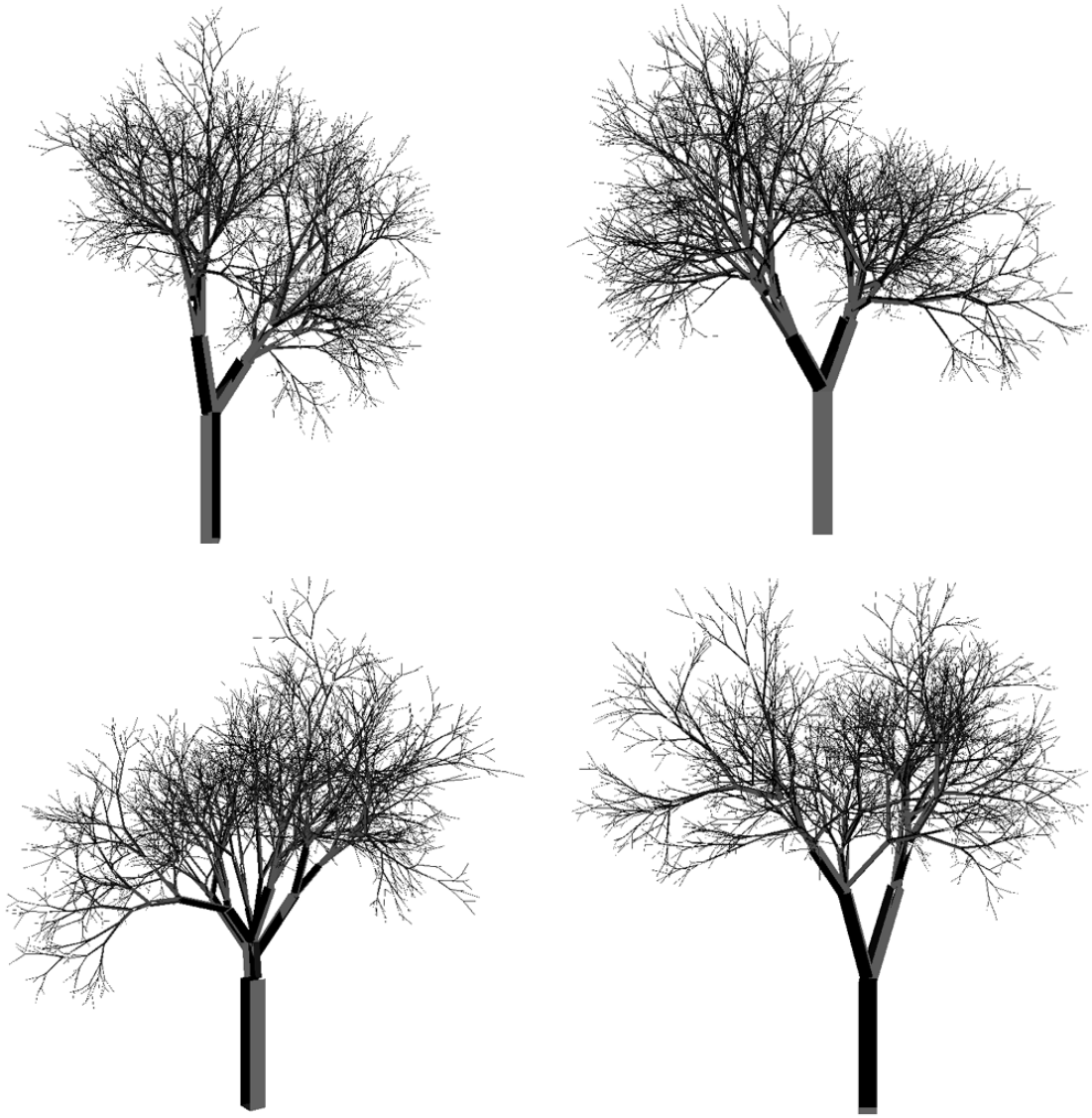

Figura 12: Quatro árvores aleatórias geradas a partir do mesmo algoritmo

3.2.1 Armazenamento da árvore gerada aleatoriamente

Como visto anteriormente, a aleatoriedade na geração das árvores proporciona uma maior naturalidade e realismo às árvores geradas, podendo-se produzir várias árvores a partir do mesmo algoritmo.

Porém essa abordagem requer a persistência dos atributos de cada galho da árvore a cada *frame* quando ela for desenhada, afim de que a forma da árvore não tenha a sua forma alterada ao longo da aplicação.

Os atributos a em questão dependem do modelo de árvore proposto, porém tomando como exemplo a árvore descrita por *Parametric* 0L-*systems* anteriormente, os atributos são:

- l é o comprimento do galho a ser desenhado.

- φ é o ângulo de rotação no eixo Y.

- Θ é o ângulo de rotação no eixo Z.

Duas abordagens de manter a persistência dos atributos dos galhos serão discutidas nos próximos sub-tópicos.

3.2.1.1 Armazenamento em memória

Uma abordagem possível é o armazenamento em memória de GPU dos atributos de todos os galhos da árvore, com posteriores acessos à memória para consultar os atributos para desenhar a árvore a cada *frame* da aplicação.

Com grande obviedade, uma sugestão de estrutura de dados para representar os atributos de cada galho no caso do exemplo utilizado é a de uma árvore binária.

Esta é uma boa sugestão, porém considerando que de uma maneira geral, em uma aplicação padrão, a estrutura da árvore não é modificada e que os atributos são acessados sempre sequencialmente, essa estrutura se mostra desnecessariamente complexa, e com um considerável *overhead* decorrente da necessidade de ponteiros.

Analisando por esse lado, um simples vetor atende perfeitamente as necessidades, sendo necessário um vetor de tamanho L^2-1.

Fazendo uma analogia com uma árvore binária, a ordem de percorrimento dos nós em *depth-first* é a mesma ordem pela qual estão organizados os nós de atributos dos galhos no vetor.

Essa abordagem pode não ser muito adequada em casos onde o espaço disponível de memória é restrito ou onde o acesso à memória é lento, pois ela requer um espaço em memória de ordem $O(2^n)$, onde n é o número de recursões , e além de número de acessos à memória da mesma ordem.

3.2.1.2 Geradores pseudo-aleatórios

Com a utilização de PRNGs, ao invés de armazenar os atributos de todos os nós da árvore, pode-se armazenar apenas uma semente para cada atributo de nó. Uma vez obtidas as sementes, a própria GPU pode gerar os atributos de todos os galhos subsequentes

Assim, o espaço armazenado e o número de acessos à memória deixa de ser de ordem $O(2^n)$ para ser um valor constante.

O custo disso é que para cada nó a GPU precisará calcular os atributos de todos os nós a cada *frame*. Porém, para as necessidades deste projeto, um Gerador Linear Congruencial é bastante interessante, pois consegue gerar uma sequência aleatória com custo de processamento e de memória bastante baixos e são extremamente simples.

Os problemas de segurança e de periodicidade desse tipo de PRNG não são relevantes uma vez que não existem requisitos de seguranças e as sequências geradas são bastante curtas.

Neste projeto foi utilizado um LCG com os seguintes valores para as constantes:

- $a = 16598013$

- $c = 9820163$

- $m = 16777216$

Com esta abordagem apresentada, o trabalho de obtenção dos atributos dos 2^L-1 galhos da árvore a cada *frame* é transferido da leitura de memória da placa gráfica para o processamento do correspondente PRNG, efetuado na própria GPU. Além de economizar espaço em memória, assim como em CPU, em GPU o acesso a memória é significativamente mais lento que o de processamento, e como os custos de processamento do PRNG apresentado é extremamente baixo, fica claro que esta abordagem é melhor opção.

3.3 Simulação da recursão

Como as linguagens de programação de GPUs não suportam recursão, e o algoritmo de geração das árvores é essencialmente recursivo, a mesma precisa ser simulada de maneira iterativa.

Para isso foi utilizado um algoritmo bastante simples que conta o número de vezes que se passa por cada nível de recursão da árvore.

Em um percorrimento *depth-first* de uma árvore binária, cada nó de nível *N* (que não seja um nó folha) é acessado três vezes:

1. A primeira vez vindo do nó pai (*N*-1) e indo para o primeiro nó filho (*N*+1).

2. A segunda vez vindo do primeiro nó filho (*N*+1) e indo para o segundo nó filho (*N*+1).

3. A terceira vez vindo do segundo nó filho (*N*+1) e retornando do nó pai (*N*-1).

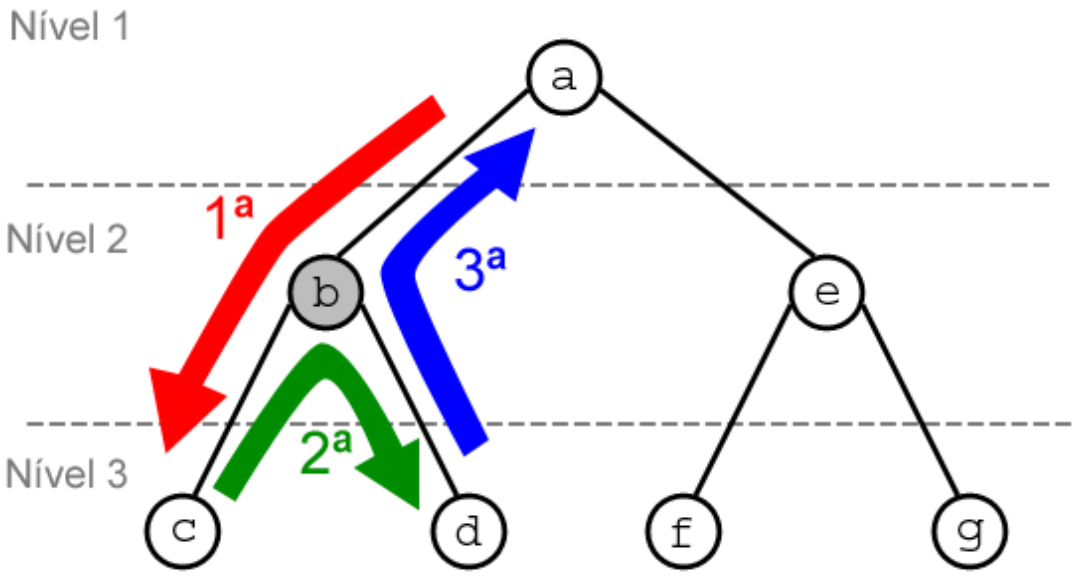

Figura 13: Passos de simulação de recursão

Essa propriedade pode ser facilmente entendida ao observar-se a Figura 13, onde o nível 2 é acessado três vezes para o nó *b*.

Dessa maneira, é possível simular o percorrimento em *depth-first* de uma árvore binária completa armazenando-se um vetor V de tamanho L e uma variável de controle n, onde L é o número total de níveis (ou de recursões) da árvore, n indica o nível corrente do percorrimento da árvore e o conteúdo da posição n do vetor V é o número de vezes que se passou pelo nível n. Para o caso da árvore aleatória, é necessário também uma variável i para acesso ao vetor de atributos dos galhos.

O pseudocódigo abaixo ilustra o funcionamento do algoritmo:

```
n = raiz;  // n recebe o nível do nó raiz
V = zeros;
while (V[raiz] < 2) {
      if (n<numMaxRecursoes) //Critério de parada
      {
            mod = modulo(V[n],3);
            V[n]++;
            if (mod==0) {
                  //Código antes das chamadas recursivas
                  n++;
            }
            if (mod==1) {
                  //Código entre as chamadas recursivas
                  n++;
            }
            if (mod==2) {
                  //Código depois das chamadas recursivas
            n--;
      }
```

```
        }
        else {
                //Desenhar folha, por exemplo
                n--;
        }
}
```

No caso da Figura 13, por exemplo, o valor da variável *n* para cada iteração do algoritmo seria:

1, 2, 3, 2, 3, 2, 1, 2, 3, 2, 3, 2, 1

As três primeiras vezes em que se passa pelo nível 2 são para o nó *b*, enquanto que as três posteriores são para o nó *e*.

No caso do exemplo utilizado, é preciso saber se o nó atual é primeiro ou segundo filho para saber qual operação sobre o eixo *Z* será utilizado (+(Θ) ou -(Θ)). Para isso, basta avaliar o resultado da seguinte expressão:

$$módulo((V[n] / 3) , 2)$$

Se o resultado for 0 é o primeiro filho e se for 1 é o segundo filho.

Esse modelo de simulação de recursão pode ser facilmente estendido para árvores N-árias.

3.4 Implementação

Como o conceito dos *geometry shaders* de receber uma primitiva geométrica como entrada e gerar a partir dela múltiplas primitivas de um determinado tipo como saída se encaixa muito bem na ideia de geração de árvores proposta neste projeto, essa tecnologia foi escolhida para ser utilizada na implementação. A linguagem de programação de *shaders* utilizada foi GLSL (OpenGL Shading Language) e foi escolhida por questão de familiaridade com OpenGL.

Esta implementação usa o modelo de geração de árvores proposto com o L-*system* descrito nesta monografia. Foi utilizado também o método de simulação iterativa da recursão, já que a mesma não é suportada pelas linguagens de *shaders*, e o Gerador Linear Congruencial de Números Pseudo-aleatórios para a produção dos atributos aleatórios das árvores em GPU.

O *geometry shader* implementado neste projeto recebe como entrada uma primitiva geométrica do tipo *GL_LINE* e gera como saída primitivas do tipo *GL_QUAD_STRIP*. A ideia é receber o primeiro vértice da entrada como a posição onde a árvore deverá ser gerada e o segundo vértice menos o primeiro como um vetor e seu módulo, que indicarão

respectivamente o eixo de crescimento e o tamanho da árvore. Qualquer outro vértice recebido como entrada é ignorado.

Os demais atributos, como as sementes das sequências pseudo-aleatórias para os atributos de tamanho, *yaw* e *pitch* são passados para o *geometry shader* através das chamadas *uniform variables* (variáveis uniformes). Os valores iniciais do L-*system* (comprimento de galho inicial, e número de recurções) e suas constantes (fator de redução, *yaw* máximo, *yaw* mínimo, *pitch* máximo e *pitch* mínimo) também são transferidos da mesma maneira.

Como as variáveis uniformes podem variar para cada primitiva de entrada, a aplicação principal em CPU pode definir diferentes valores dessas variáveis para cada primitiva de entrada diferente, dando o poder de coordenação do aspecto das árvores à aplicação principal que pode tanto gerar árvores iguais ou diferentes, de acordo com o requerido.

O principal problema encontrado nesta implementação é o limite no número de vértices de saída que podem ser gerados pelos *geometry shaders*. Atualmente este valor máximo é de apenas 1024 vértices.

Considerando que em uma abordagem muito simples onde as folhas da árvore são ignoradas e cada galho é formado por uma primitiva do tipo *GL_LINE* consumindo 2 vértices, apenas 512 galhos podem ser desenhados. Assim, é possível gerar uma árvore

com apenas 9 recursões. Apesar disso, é crível que esta limitação será melhorada muito com o desenvolvimento das placas gráficas em um futuro próximo.

Uma possível solução para superar esta limitação no número de vértices de saída é dividir uma árvore mais complexa em sub-árvores de 1024 vértices cada que seriam geradas pelo *geometry shader* e agregadas duas (em caso de árvore de ramificação binária) à cada uma das extremidades de uma sub-árvore mãe gerada em CPU.

Como exemplo de tal abordagem para o problema, pode-se tomar uma árvore binária com 11 recursões consequentemente com 2047 (2^{11}-1) galhos e 8 vértices por galho. Essa árvore seria dividida em um sub-árvore principal gerada em CPU com 4 recursões e 16 sub-árvores filhas, cada uma com 7 recursões e um total de 1016 vértices, geradas através de *geometry shaders*, sendo anexadas um par de sub-árvores filhas a cada uma das extremidades da sub-árvore mãe.

Para dar uma ideia, as árvores da Figura 14 foram desenhadas com 7 recursões, tendo assim 127 (2^7-1) galhos. Como nessa figura cada galho é composto por uma primitiva *GL_TRIANGLE_STRIP* de 6 triângulos, isso resulta em 8 vértices por galho. Assim a árvore toda requer 1016 (8*127) vértices, o que está dentro das atuais limitações dos *geometry shaders*.

Figura 14: Floresta composta por árvores com 1016 vértices cada

Outro problema importante é que para que uma árvore 3D gerada por L-*system* tenha um resultado visual interessante, são necessários em geral um número de vértices muito grande, o que pode tornar inviável para uma aplicação de tempo real.

Uma solução para isso é adicionar ao L-*system* o uso de *sprites*, que é atualmente empregado em modelos de árvores 3D na maioria das aplicações gráficas de tempo-real. Isso poderia substituir o grande número de vértices necessários para os galhos menores e folhas por texturas que teriam um efeito aproximado ao de um L-*system* mas com um número aceitável de vértices.

Contudo, as soluções dos dois últimos problemas apresentados todavia não são implementados neste projeto por questões de restrição de tempo.

5 Testes e resultados

Para avaliar o desempenho da implementação em *geometry shader*, serão comparados os resultados de tempo de processamento de *frame* e taxa de *frames* por segundo da implementação proposta neste projeto com os resultados da implementação da abordagem tradicional do problema, gerando e armazenando árvores em CPU. Foi desenvolvido um teste simples envolvendo exclusivamente a geração e exibição de árvores.

O tempo médio de processamento de *frame* foi calculado a partir do tempo de processamento de 1000 *frames* em ambas as implementações para 5 cenários com 1, 10, 100, 1000 e 10000 árvores, cada uma com 128 vértices. Esse experimento foi executado em uma máquina com processador Core 2 Quad com 4GB de memória RAM e duas placas de vídeo GeForce 8800GTS (com *Shader Model* 4.0) ligadas em paralelo através da tecnologia SLI (Scalable Link Interface). Os dados foram coletados e são ilustrados no gráfico e tabela exibidos abaixo.

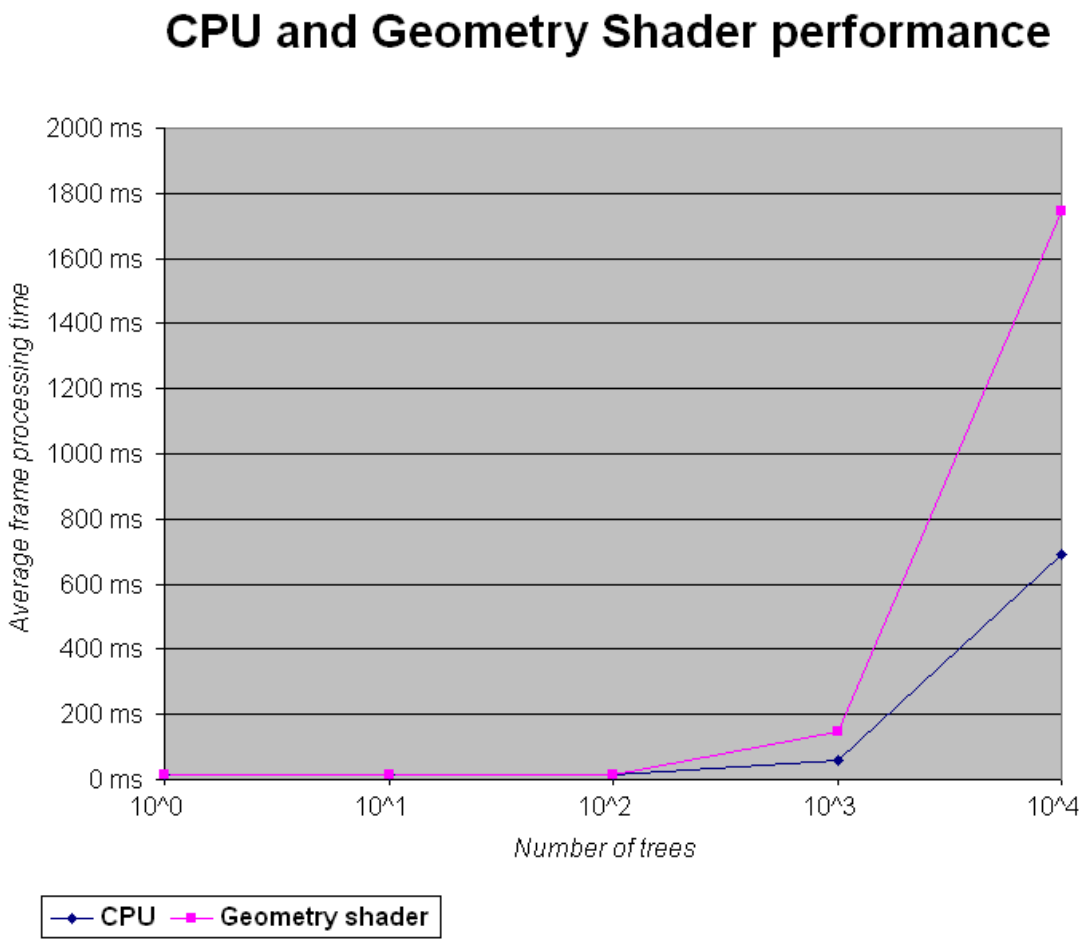

Figura 15: Gráfico comparativo do tempo médio de processamento de frame entre as implementações em GPU e em geometry shader

Number of trees	Average frame rate (FPS)	
	CPU	Geometry Shader
1	60.07975387	59.88987051
10	59.88270973	59.88867773
100	59.88911730	59.88270973
1000	17.57156758	6.87307478
10000	1.44413981	0.57341839

Tabela 1: Comparação de frame rate médio entre as implementações em GPU e em geometry shader

Como pode ser notado no Figura 15 e na Tabela 1, os resultados obtidos no experimento não atingiram as expectativas. Apesar de que até a quantidade de 100 árvores o geometry shader teve resultado satisfatório, apresentando desempenho bastante semelhante ao da implementação tradicional em CPU.

Porém, quando se chega às quantidades de 1000 e 10000 árvores a implementação em GPU apresenta um desempenho significativamente inferior ao da implementação em CPU, o que é exatamente o oposto do inicialmente esperado.

Este resultado tem que como uma explicação o fato de que o teste executado envolve apenas a geração e exibição de árvores. Isso significa que ao transferir a tarefa de geração das árvores 3D de CPU para GPU, a aplicação se torna extremamente intensiva em GPU. Desta maneira, para grandes quantidades de árvores como 1000 e 10000, a implementação em CPU leva vantagem já que toda a carga é dividida entre CPU (geração) e GPU (exibição), enquanto que na implementação proposta a GPU é saturada pelas duas tarefas.

Isso deixaria de ser problema no caso de aplicações mais complexas que sejam mais intensivas em CPU ou que possuam gargalo na transferência de dados entre CPU e GPU.

Outra fator que pode ter contribuído como causa de tal resultado é o grande número de desvios de fluxo de execução existentes no *geometry shader* implementado. A

simulação de recursão necessária para a geração a árvore requer uma grande quantidade de *ifs* e *whiles*, e como já visto anteriormente no Capítulo 2.2, elas utilizam a maioria de seus transistores para a computação e muito pouco para a parte de controle, tornando os fluxos de programas um pouco mais limitados.

Outro fator importante é que geometry shaders, talvez por ser uma tecnologia extremamente nova, ainda possui diversas limitações, como o número máximo de vértives de saida, o que possivelmente deve ser melhorado com o desenvolvimento de novas versões.

6 Conclusão

Neste trabalho foi apresentado um modelo de geração de árvores 3D em tempo real em GPU utilizando geometry shaders, e tentou-se com isso utilizar o potencial das placas gráficas atuais para melhorar o desempenho dessa tarefa de geração de árvores em tempo real. Alguns dos problemas presentes neste modelo, assim como sugestões de possíveis soluções foram apresentados no Capítulo 5 desta monografia.

Como visto, a implementação em geometry shader não obteve resultados satisfatórios, demostrando que a utilização desse tipo de shader não foi adequada para o objetivo proposto. Apesar de que os resultados obtidos nos testes deste trabalho não terem sido satisfatórios, todavia é interessante realizar testes em condições semelhantes a de uma aplicação 3D de tempo-real, com grande utilização de CPU, o que devido a restrições de tempo, ainda não foi possível de ser realizado. Além da execução de testes mais detalhados, fica como proposta de trabalhos futuros a avaliação da possibilidade de uso de outras tecnologias para programação de GPU, como por exemplo CUDA e a adição ao modelo proposto do uso de sprites, com o objetivo de reduzir a quantidade de vértices necessários e melhorar o resultado visual.

Referências Bibliográficas

P. PRUSINKIEWICZ, A. LINDENMAYER. 1990. The Algorithmic Beauty of Plants. Springer-Verlag (New York, 1996).

J. MCCORMACK. Interactive Evolution of L-System Grammars for Computer Graphics Modelling. ISO Press (Amsterdam, 1993)

T. DOKKEN, T. HAGEN, J. HJELMERVICK. The GPU as a high performance computational resource, submitted for publication in "Spring Conference on Computer Graphics 2005".

R. J. ROST. OpenGL(R) Shading Language, Addison-Wesley, 2004.

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

L'ECUYER, P.: Random number generation. In Jerry Banks, editor(s), The Handbook of Simulation. Wiley, 1998.

Brian P. SORGE, Jeremy T. BARRON, Timothy A. DAVIS. Real-Time Procedural. Animation of Trees. Eurographics 2001.

Przemyslaw PRUSINKIEWICZ, Mark HAMMEL, Jim HANAN, and Radomir MECH. L-systems: from the theory to visual models of plants. In: Proceedings of the second CSIRO symposium on computational challenges in life sciences; 1996.

ROST, Randi J. OpenGL Shading Language. Boston: Addison-Wesley. 2004.

FOSNER, Ron. Real-Time Shader Programming. San Francisco: Morgan Kaufmann Publishers. 2003.

Marcos Vinícius CIRNE. Introdução à Arquitetura de GPUs. Instituto de Computação - UNICAMP. 2008.

David LUEBKE, Greg HUMPHREYS. How GPUs work. In Computer Magazine, IEEE Computer Society, 2007.

John KESSENICH, Dave BALDWIN, Randi ROST. The OpenGL® Shading Language (Language Version: 1.50, Document Revision: 09), 2009.

Diogo LIMA, Henry BRAUN. Exibição de terrenos em tempo real, uma abordagem a terrenos com larga escala geométrica. Faculdade de Informática - PUC-RS, 2008.

W. CELES Notas de Aula. PUC-Rio, 2006.

David REIS, Ivan CONTI, Jeronimo VENETILLO. GPU - Graphics Processor Units. DCC-UFMG, 2007.

Andreas MUHAR. Three-dimensional modeling and visualization of vegetation for landscape simulation. Landscape and Urban Planning, 2001.

S.K. PARK, K.W. MILLER. Random Number Generators: Good Ones Are Hard To Find. Communications of the ACM, 1988.

Tree Generator [2009], disponível em <http://bendapkiewicz.com/flash/tree.html>, Acessado 05/11/2005 as 11:11.

P. KRUSZEWSKI, S. WHITESIDES, A general combinatorial model of botanical trees. J. Theoret. Biol., 1998.

C. L. PEREIRA. Modelagem de Plantas Utilizando L-System (Easy Tree). 2004. Trabalho de Conclusão de Curso. (Graduação em Ciência da Computação) - Universidade Federal Fluminense. Orientador: Aura Conci.

L. L. MARTINEZ. Modelagem de Plantas usando Algoritmo Tartaruga. 2004. Trabalho de Conclusão de Curso. (Graduação em Ciência da Computação) - Universidade Federal Fluminense. Orientador: Aura Conci.