Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Fábio Markus Nunes Miranda

MONOGRAFIA DE PROJETO ORIENTADO EM COMPUTAÇÃO II

Geração Procedural e Visualização de Terrenos Pseudo-Infinitos na GPU

Belo Horizonte – MG
2009 / 1º semestre

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Geração Procedural e Visualização de Terrenos Pseudo-Infinitos na GPU

por

Fábio Markus Nunes Miranda

Monografia de Projeto Orientado em Computação I

Apresentado como requisito da disciplina de Projeto Orientado em Computação II do Curso de Bacharelado em Ciência da Computação da UFMG

Prof. Dr. Luiz Chaimowicz
Orientador

Carlúcio Cordeiro
Co-Orientador

Assinatura do aluno:

Assinatura do orientador:

Assinatura do co-orientador:

Belo Horizonte – MG
2009 / 1º semestre

Aos meus pais,
aos professores e
aos colegas de curso,
dedico este trabalho.

Agradecimentos

Agradeço aos meus pais, pela minha formação e pelo apoio durante o curso.
Aos meus professores e orientadores, pelos conhecimentos adquiridos.
E finalmente aos colegas de curso pela convivência e trocas de experiências.

"'Natural complexity seems to defeat all human attempts to oversimplify and tame it.
And maybe that's a good thing. "

Ken Perlin

Resumo

Com o aumento da demanda por modelos 3D em áreas como jogos eletrônicos, a geração procedural se faz cada vez mais necessária, para reduzir custos e tempo de desenvolvimento. Através de técnicas procedurais, é possível criar um grande número de modelos com a variação de alguns poucos parâmetros dos algoritmos responsáveis pela geração. Este trabalho tem como objetivo utilizar o poder das placas gráficas atuais (GPUs) para a geração de terrenos pseudo-infinitos, criados a medida em que o usuário navega pelo cenário. Um comparativo com geração de terrenos na CPU também é feito, para avaliar a performance das duas diferentes arquiteturas.

Palavras-chave: Geração procedural de conteúdo, terreno, gpu.

Abstract

With the increasing demand for 3D models in areas such as games, the procedural content generation is becoming increasingly popular, in order to reduce costs and development time. With procedural techniques, it is possible to create a large number of models modifying a few parameters on the algorithms responsable for the generation. This work intends to use the power of the modern video cards (GPU) to generate pseudo infinite terrains, created as the user navigates through the scene. A comparative study between GPU generation and CPU is made, in order to evaluate the performance of the two different architectures.

Keywords: Procedural content generation, terrain, gpu.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

GPUs	Graphics Processing Unit
HLSL	High Level Shader Language
Cg	C for Graphics
GLSL	OpenGL Shading Language
OpenGL ARB	OpenGL Architecture Review Board
API	Application Programming Interface
CUDA	Compute Unified Device Architecture
VBO	Vertex Buffer Object
FBO	Frame Buffer Object
FPS	Frames por segundo

Sumário

1 INTRODUÇÃO

1.1 Visão geral

A geração procedural de modelos é uma área da Ciência da Computação que propõe que modelos gráficos tridimensionais (representação em polígonos de algum objeto) possam ser gerados através de rotinas e algoritmos. Tal técnica vem se tornando bastante popular nos últimos tempos, tendo em vista que, com o crescimento da indústria do entretenimento, há uma necessidade de se construir modelos cada vez maiores e com um grande nível de detalhe. A técnica de geração procedural vem então como uma alternativa à utilização do trabalho de artistas e modeladores na criação de modelos tridimensionais.

Outro fato também muito relevante atualmente são as *Graphics Processing Unit* (GPUs), microprocessadores incorporados às placas de vídeo e especializados em processamento gráfico. O avanço da indústria de *games* fez com que as GPUs se tornassem cada vez mais rápidas, tornando-as atraentes para outras áreas da computação.

1.2 Objetivo, justificativa e motivação

O objetivo deste trabalho é permitir a criação de terrenos proceduralmente em tempo real, aproveitando o poder de processamento da GPU, e comparar as velocidades de geração em relação à CPU. O trabalho pode ser dividido em duas vertentes: visualização de terrenos e sua geração procedural.

O primeiro aspecto, a visualização, é um problema muito estudado em computação, pois o modelo de um terreno é algo que pode demandar um número extremamente alto de triângulos e o processo de gerar a imagem a partir desse modelo, ou renderização, em tempo real fica inviabilizado nos computadores atuais. Faz-se então necessária a utilização de técnicas que limitam e minimizam o número de triângulos a serem desenhados na tela. Entra aí o uso de *culling* e níveis de detalhe dos modelos.

A segunda parte, geração de terrenos proceduralmente, busca, através de algoritmos, criar terrenos realistas e que possam ser utilizados em jogos eletrônicos, simuladores, ou qualquer tipo de aplicação que necessita de um ambiente virtual tridimensional. Os principais benefícios dessa geração são:

- Compressão de dados: todos os modelos são criados por algoritmos, e não há a necessidade de se armazenar os dados dos modelos gerados, diminuindo assim o tempo necessário para a transmissão do conteúdo por uma rede, por exemplo.
- Conteúdo gerado pelo usuário de forma fácil: com a geração procedural, o usuário não precisa de um grande conhecimento ou então a contratação de artistas para poder construir modelos 3D. Uma interface amigável e alguns ajustes de parâmetros são o bastante para gerar modelos interessantes.
- Produtividade: quanto menor o número de entradas um sistema procedural possuir, menor o trabalho necessário para criar modelos.

Este trabalho irá implementar soluções para a geração procedural de terrenos na GPU e também na CPU, com o objetivo de permitir ao usuário a navegação por um terreno pseudo-infinito, limitado apenas pela precisão dos tipos primitivos utilizados (*double* e *integer*). Além disso, será comparado a eficiência e velocidade das duas arquiteturas.

1.3 Organização

O restante do trabalho está organizado da seguinte maneira: O Capítulo 2 apresenta o referencial teórico pertinente a este trabalho. O Capítulo 3 mostra a metodologia envolvida. O Capítulo 4 mostra como o trabalho foi desenvolvido e implementado. O Capítulo 5 apresenta os resultados obtidos e, finalmente, o Capítulo 6 mostra a conclusão do trabalho e propostas para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Para uma melhor compreensão deste trabalho, é necessário explicar o contexto em que ele se situa e também alguns trabalhos relacionadas. O referencial teórico pode ser dividido em duas áreas: geração procedural e GPU. A parte referente à geração procedural será apresentada na Seção 2.1, enquanto que na Seção 2.2 será apresentado conceitos e trabalhos relativos à programação utilizando a GPU.

2.1 Geração Procedural

Vários trabalhos publicados abordam a geração de modelos tridimensionais com o uso de algoritmos. Alguns destes trabalhos abordam a geração de cidades ([3] e [4]), outros abordam a geração de terrenos realistas (em tempo-real, como os trabalhos [5] e [6], ou não, como o *MojoWorld* [7]), ou então a geração de árvores [8]. A principal referência na área é o livro *Texturing and Modeling: A Procedural Approach* [9], em que é explicada a geração procedural de diversos tipos de modelos.

Algumas técnicas largamente utilizadas na geração procedural são: Sistemas de Lindenmayer (*l-System*)[10], que, através de uma gramática, pode modelar o crescimento de plantas; geometrias fractais [11]; e também ruído Perlin (*Perlin noise* [12]).

2.1.1 Terrenos procedurais

Algumas técnicas populares para criação de terrenos proceduralmente são ruído Perlin, *fractal plasma*, *fault formation*, *circles*. Para este trabalho, porém, foi escolhido o *Ridged multifractal noise*, uma variação do ruído Perlin, por este ser o que melhor representa terrenos [9].

Na Seção 2.1.1.1 será explicado como os dados gerados são armazenados, através de mapas de altura. Nas Seções 2.1.1.2 e 2.1.1.3 serão apresentados os conceitos de ruído Perlin e fractais, necessários para o entendimento do algoritmo *Ridged multifractal noise*,

mostrado na Seção 2.1.1.4.

2.1.1.1 Mapa de altura

Um mapa de altura (ou *heightmap*) é uma imagem bidimensional que armazena dados referentes ao relevo de um terreno. Geralmente, tons mais claros representam pontos mais altos, enquanto tons mais escuros são pontos mais baixos do mapa. A Figura 1 [13] é um exemplo de mapa de altura.

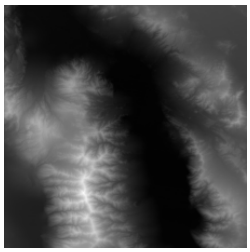

Figura 1: Exemplo de um mapa de altura.

Os algoritmos de geração procedural irão retornar valores que representam a altura de um determinado ponto do terreno, e estes serão armazenados em um mapa de altura.

2.1.1.2 Ruído Perlin

O ruído Perlin foi criado pelo Professor Ken Perlin [14], da *New York University* e é usado para simular estruturas naturais, como nuvens, texturas de árvores, e terrenos.

A função ruído retorna, para um dado domínio e as mesmas sementes (*seeds*), números entre 0 e 1; dessa forma, em uma segunda execução, com as mesmas entradas, teremos os mesmos números entre 0 e 1. Cada valor retornado é o resultado do seguinte produto interno:

$$G \cdot (P-Q)$$

Onde P é a posição do ponto que está sendo calculado o valor do ruído, Q é a posição de um de seus vizinhos, e G é o valor de um vetor gradiente pseudo-aleatório. Os resultados do produto interno dos vizinhos é então interpolado, garantindo assim que haverá uma suave transição entre todos os valores retornados.

O resultado, como pode ser visto na Figura 2, apresenta transições suaves, diferentemente do ruído aleatório.

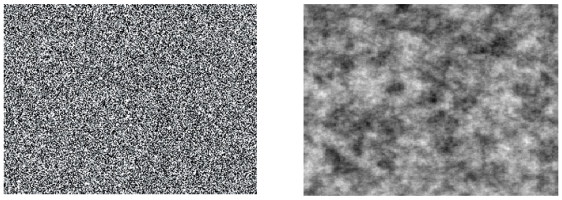

Figura 2: *Esquerda:* Ruído aleatório. *Direita:* Ruído Perlin.

As características fundamentais do ruído Perlin são então a sua aparente aleatoriedade (ao menos para o olho humano); sua capacidade de ser reproduzido, dado os mesmos valores dos gradientes; e sua transição suave entre valores.

2.1.1.3 Fractais

Fractais podem ser descritos, segundo [9], como objetos geométricos complexos, na qual a complexidade surge da repetição de uma forma em uma extensão de escalas. Um exemplo simples pode ser visto na Figura 3:

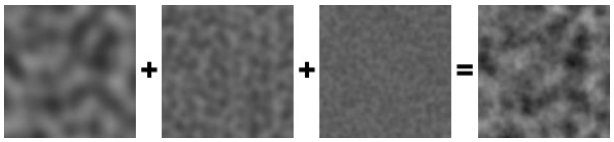

Figura 3: Exemplo de um fractal a pertir de ruído Perlin.

Os três ruídos Perlin estão em escalas diferentes e, uma vez somados, formam um fractal, segundo a definição citada. Multifractais já são um subgrupo caracterizado pela variação de sua dimensão fractal ao longo de sua localização.

2.1.1.4 *Ridged multifractal noise*

O *Ridged multifractal noise* é uma variação do ruído Perlin, e foi apresentado em [9]. O principal ponto do algoritmo é que ele captura a *heterogeneidade de terrenos em grande escala*, apresentando montanhas, planaltos e crateras. Um pseudo-algoritmo pode ser visto na Tabela 1.

```
for(int i=0; i<octaves; i++) {
        float n = ridge(noise(position*freq), offset);
        height += n*amplitude*previous;
        previous = n;
        frequency *= lacunarity;
        amplitude *= gain;
}
```

Tabela 1: Pseudo-algoritmo do *Ridged multifractal noise*.

No algoritmo, o número de **octaves** representa o número de iterações (e, consequentemente somas) feitas sobre a função de ruído. A **amplitude** é o máximo valor adicionado ao valor total do ruído. **Frequency** é o número de valores de ruídos definidos entre dois pontos (quanto maior a frequência, maior o disturbio da textura resultante). **Lacunarity** é um termo usado no cálculo de fractais, e dita o espaço entre sucessivas frequências, aumentando ou diminuindo a densidade do resultado final. Finalmente, **offset** é o fator multifractal; a sua variação torna o resultado mais heterogêneo (menor *offset*), ou mais uniforme (maior *offset*).

A função *noise* retorna um valor de acordo com o ruído Perlin, alterado pela função *ridge*, que leva em consideração o fator multifractal (*offset*).

2.2 GPUs

As GPUs são as unidades de processamento inseridas na maioria das placas de vídeo atuais. A sua evolução foi incentivada pela alta demanda do mercado de renderização 3D em tempo-real, como *games* e simuladores virtuais, sempre em busca de representar a realidade da maneira mais fidedígna possível. A imagem 4 mostra a rápida evolução das arquiteturas da *NVidia* (uma das principais produtores de placas de vídeo), considerando o número de operações de pontos flutuantes executados por segundo, em contraste com a evolução das CPUs.

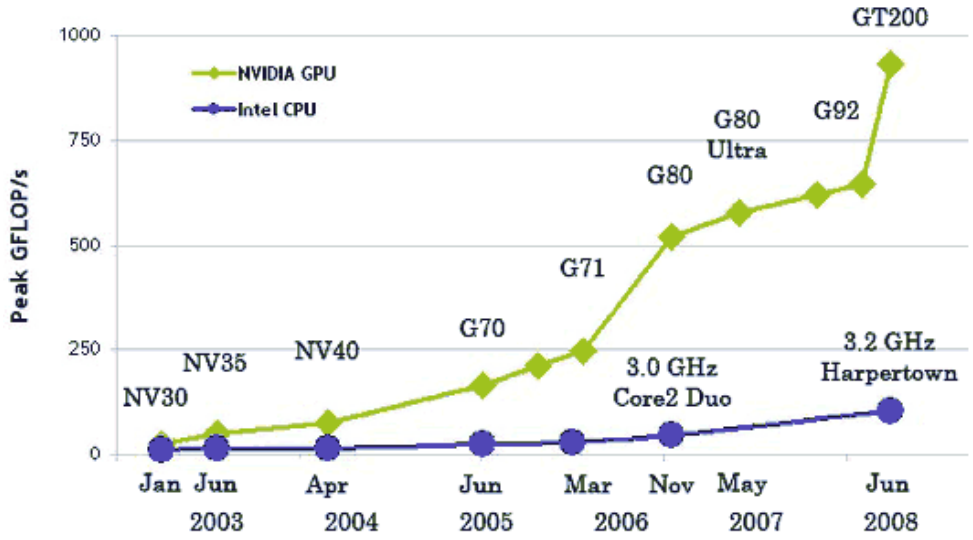

Figura 4: Gráfico com a evolução das GPUs, em comparação com as CPUs [1].

Tamanha diferença evolutiva deve-se, principalmente, ao fato de que as GPUs foram construídas de forma a processar da melhor maneira possível um grande número de dados homogêneos e independentes entre si, como é o caso de vértices e *pixels*, base de todo o processamento gráfico. Assim, certos aspectos de um processador, como *cache*, sincronização de *threads* e a comunicação entre elas, são relegados.

A programação utilizando GPUs iniciou de forma mais concreta no ano de 2001, com a placa *Geforce3*, da *NVidia*. Essa geração permitiu que programas (conhecidos como *shaders*) fossem escritos para serem executados diretamente na GPU. Inicialmente, tais programas modificavam os valores e propriedades dos vértices de uma cena. Na geração seguinte (representada pelas placas *GeForce FX*), foi possível também executar *shaders* que modificavam os valores dos *pixels* (ou fragmentos) da cena. A Figura abaixo mostra o *pipeline* gráfico das GPUs, destacando a posição dos *programmable vertex processor* e *programmable fragment processor*, estágios responsáveis por executar o *vertex shader* e *fragment shader*, respectivamente.

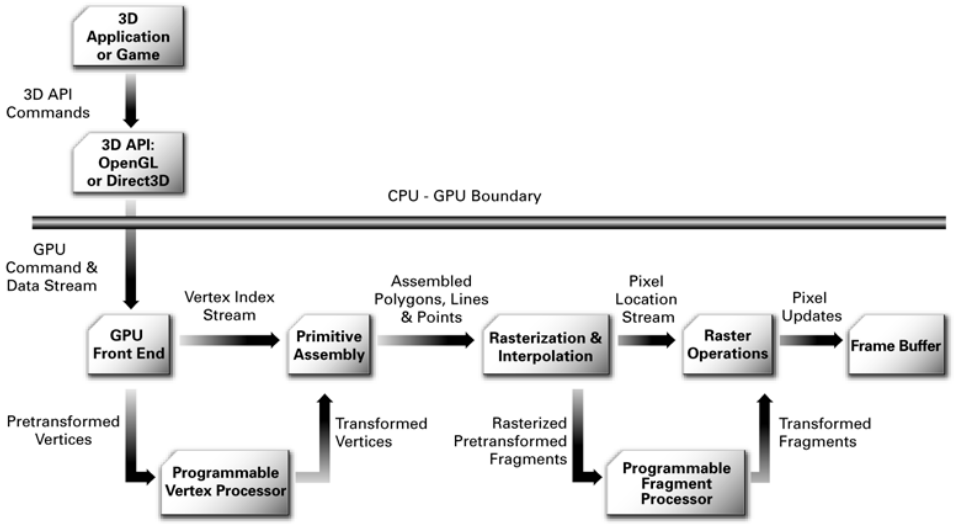

Figura 5: *Pipeline* gráfico [2].

O *vertex shader* é o responsável por manipular as propriedades (posição, cor, coordenada de textura) dos vértices enviados para a GPU, e o *fragment shader* calcula a cor de todos os fragmentos da cena. Placas de vídeo modernas apresentam um outro estágio, denominado *geometry shader*, capaz de criar novas primitivas.

Ao longo da evolução das GPUs, uma série de linguagens para programação de *shaders* foram criadas. Entre elas, podemos ressaltar a *High Level Shader Language* (HLSL), da *Microsoft*, *C for Graphics* (Cg), da *NVidia*, e *OpenGL Shading Language* (GLSL), da *OpenGL Architecture Review Board* (OpenGL ARB). Ao se utilizar uma dessas linguagens, todo o processamento será feito com base nos vértices (em um *vertex shader*) ou fragmentos (*fragment shader*), e o resultado do processamento também ficará limitado a tais primitivas.

O destino da renderização de todas as *Application Programming Interfaces* (API) é o chamado *framebuffer*, que nada mais é do que um espaço de memória que armazena informações sobre cores dos pixels. Como é possível criar novos *framebuffers*, também é possível criar novos destinos de renderização para os *shaders*, não ficando restrito apenas ao *framebuffer* que é exibido na tela.

A arquitetura *CUDA* CUDA, da *NVidia*, também permite que programas rodem na GPU, porém de uma maneira diferente. Não há a utilização de *shaders* e programas não ficam limitados à operações em cima de vértices e fragmentos. A arquitetura só está disponível para placas *NVidia* a partir da geração *GeForce 8*.

2.3 Geração Procedural utilizando GPUs

A geração procedural de terrenos na GPU já foi abordada nos trabalhos [15] e [16]. O primeiro faz uso dos *geometry shaders*, disponíveis nas placas mais recentes, para a geração procedural. Apesar de mostrar resultados excelentes, está limitado às placas com suporte à API *DirectX 10*. O trabalho descrito aqui pode ser executado em qualquer placa com suporte a *DirectX 9*.

O segundo trabalho ([16]) é o que mais se assemelha ao que é proposto aqui. Ele porém não busca comparar a geração utilizando a GPU e a CPU.

3 METODOLOGIA

3.1 Tipo de Pesquisa

O trabalho proposto é uma pesquisa de natureza aplicada, pois busca aplicar um conhecimento teórico (técnicas de geração procedural na GPU) e obter um resultado prático na forma de um sistema, e tem um objetivo exploratório. A pesquisa se dá em laboratório, pois se trata de um ambiente controlado.

3.2 Procedimentos metodológicos

O primeiro passo do trabalho foi a escolha de uma API. Por se tratar de uma plataforma aberta e já estudada em matérias durante o curso, o *OpenGL* foi escolhido, juntamente com a linguagem C++. Além disso, essa API oferece algumas extensões que permitem maximizar a performance do sistema, como o a estrutura *Vertex Buffer Object* (VBO), que armazena os vértices do modelo 3D diretamente na memória da GPU, diminuindo o número de chamadas para a renderização, e o *Frame Buffer Object* (FBO), que permite renderizar o resultado de uma certa computação, realizada em um *shader*, diretamente em uma textura.

Durante este tempo de pesquisa, foi considerado a utilização da plataforma CUDA, da *NVidia*, destinada especificamente para o uso da GPU em computação de propósito geral. Porém, ele foi descartado, uma vez que seu uso é limitado às placas *NVidia* e, de uma maneira geral, tudo que é proposto aqui pode ser feito com o uso de plataformas abertas e que podem ser executadas em um maior número de computadores.

O segundo passo foi pesquisar diversas técnicas procedurais envolvidas na geração de terrenos, bem como alguns sistemas que oferecem soluções ligadas à geração procedural, como o *CityEngine* [17] e o *MojoWorld* [7]. Foi pesquisado também trabalhos relacionados à geração procedural de terrenos na GPU.

O terceiro passo envolveu a construção de um sistema que permite a geração procedural de terrenos infinitos tanto na GPU quanto na CPU, sendo possível navegar pelo cenário de maneira intuitiva.

Finalmente, foram executados testes comparativos entre a geração procedural na GPU e na CPU, a fim de observar os benefícios obtidos em cada arquitetura.

4 IMPLEMENTAÇÃO

Os conhecimentos adquiridos ao longo desse trabalho permitiram a criação de um sistema capaz de gerar terrenos procedurais tanto na GPU quanto na CPU, e também permite a navegação do usuário por tal terreno. Na Seção 4.1 será apresentado uma visão geral do sistema. As Seções 4.2 e 4.3 mostrarão como os terrenos são gerados e visualizados.

4.1 Visão Geral do Sistema

O sistema implementado neste trabalho teve como principal objetivo permitir a geração procedural de terrenos tanto na GPU quanto na CPU. A Figura 6 apresenta as camadas do sistema, destacando as bibliotecas utilizadas (como é explicado a seguir).

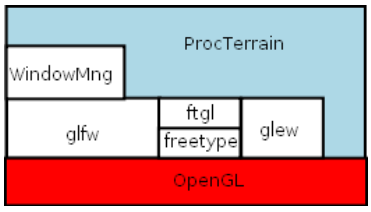

Figura 6: Camadas do sistema.

- **OpenGL**: *API* gráfica utilizada para a renderização.
- **glew**: Biblioteca para carregamento de extensão do *OpenGL*.
- **glfw**: Biblioteca que facilita o tratamento de entradas e também criação de janelas.
- **ftgl**: Biblioteca para a renderização de textos.
- **FreeType**: Biblioteca para renderização de textos (dependência do *ftgl*).
- **WindowMng**: Camada responsável por criar a tela e tratar os eventos de entrada.
- **ProcTerrain**: Camada responsável por gerar e exibir os terrenos.

As camadas *WindowMng* e *ProcTerrain* foram implementadas neste trabalho. O *WindowMng* tem como propósito simular a camada de um aplicativo gráfico genérico (*game*, simulador, etc.); desta forma, o sistema poderá ser posteriormente adaptado para funcionar em conjunto com outros aplicativos que possam ser desenvolvidos.

A Figura 7 apresenta em detalhes os módulos presentes nas camadas *WindowMng* e *ProcTerrain*. A seguir, uma explicação sobre cada um dos módulos.

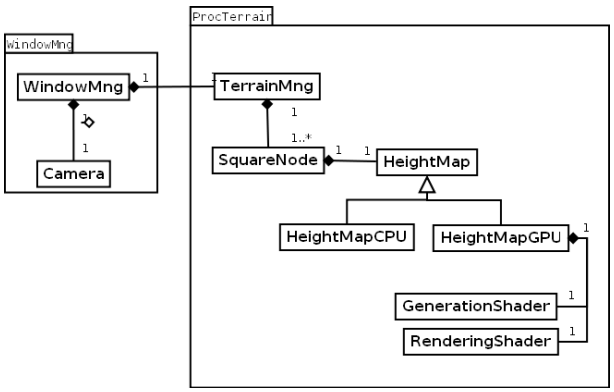

Figura 7: Diagrama com as principais classes do sistema implementado.

- **WindowMng**: Responsável por simular um aplicativo gráfico genérico, e chamar os devidos *callbacks* do pacote *ProcTerrain*.
- **Camera**: Módulo que implementa uma câmera controlada pelo jogador e navegando pelo mundo.
- **TerrainMng**: Módulo responsável por gerar e controlar os terrenos.
- **SquareNode**: Nodo que representa uma fatia (*patch*) do terreno.
- **HeightMap**, **HeightMapCPU** e **HeightMapGPU**: Módulos que implementam os mapas de altura dos terrenos gerados na CPU ou na GPU.
- **GenerationShader**: *Shader* responsável pela geração dos terrenos.
- **RenderingShader**: *Shader* responsável pela renderização dos terrenos.

4.2 Geração do Terreno

Nesta seção, será abordada a implementação da geração de terrenos, tanto na GPU, quanto na CPU. Os dois têm, em comum, o algoritmo usado para a geração (*Ridged multifractal noise*, descrito na Seção 2.1.1.4).

O terreno geral é dividido em terrenos menores (chamados *patchs*), como mostra o *grid* da Figura 8:

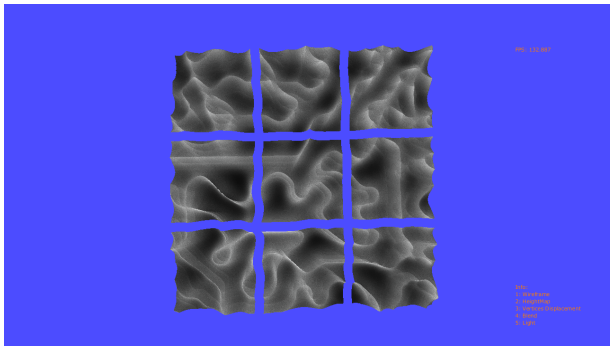

Figura 8: *Patchs* exibidos em um *grid*.

Considerando o usuário inicialmente localizado no *patch* central, ao mover-se para um *patch* vizinho, o sistema irá requisitar a geraçaõ de novos *patchs*, vizinhos a aqueles que estão na borda do grid. O número de vizinhos gerados, bem como a quantidade de vizinhos do *patch* central são variáveis do sistema, podendo ser adaptadas, pelo usuário, de acordo com o poder de processamento de sua máquina.

4.2.1 Geração do Terreno na GPU

Toda a geração dos terrenos na GPU é feita através de um *fragment shader*. Como toda computação de *shaders* fica limitada a geometrias ou texturas, foi preciso renderizar um quadrado utilizando as funções OpenGL, para que, dessa forma, fosse possível aplicar os *shaders* às suas primitivas e iniciar os cálculos necessários. O resultado da geração é renderizado em um *framebuffer off-screen*, que não é exibido na tela, através da extenção FBO, que permite criar novos *buffers*.

O cálculo dos vetores gradientes, necessário no ruído Perlin, é feito na CPU, apenas no início do sistema, e depois é acessado no *fragment shader* como uma textura 2D.

A Figura 9 apresenta o resultado da geração, visto como um mapa de altura.

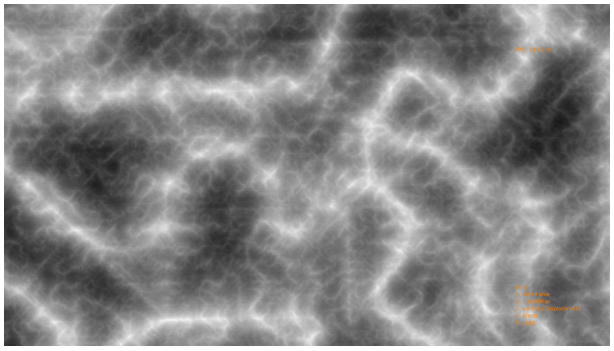

Figura 9: Mapa de altura gerado pelo *shader*.

Como o mapa de altura é gerado na GPU, não há qualquer tipo de perda de desempenho com a transferência entre a memória RAM e a memória da placa de vídeo. Um aspecto importante é que, durante a geração do mapa de altura, os valores das normais de cada vértice também são calculados.

4.2.2 Geração do Terreno na CPU

A geração na CPU é feita de maneira tradicional. Uma matrix com o tamanho da textura do mapa de altura é preenchida de acordo com o algoritmo *Ridged multifractal noise*, e posteriormente enviada para a memória da GPU.

4.3 Visualização do Terreno

Com o mapa de altura gerado, o próximo passo é exibir o terreno para o usuário, que é feito de forma idêntica tanto para os terrenos gerados na GPU quanto para os gerados na CPU.

O passo inicial é a geração de uma malha (conjunto de vértices) de tamanho pré-determinado, como mostra a Figura 10.

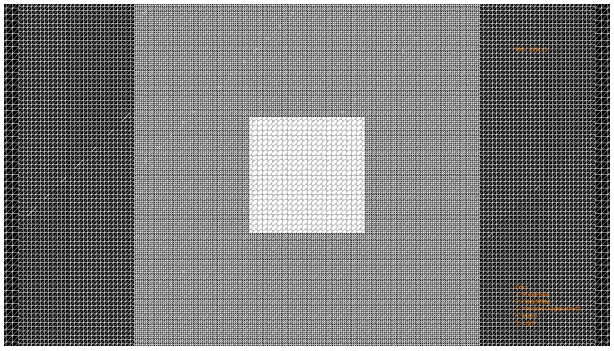

Figura 10: Malha inicial para visualização dos terrenos.

A malha é gerada de tal forma que um número maior de vértices está concentrado no centro. Quanto maior a distância, menor o número de vértices presentes. Isto propicia uma maneira rápida e fácil de implementar um nível de detalhamento (quanto maior a distância do centro, menor será a necessidade de se renderizar o terreno em alta fidelidade).

Como a malha é gerada apenas uma única vez (no início da execução), não é preciso criar repetidas malhas a medida que o jogador percorre o terreno. Apenas os mapas de altura de cada *patch* são trocados, como mostra a Figura 11

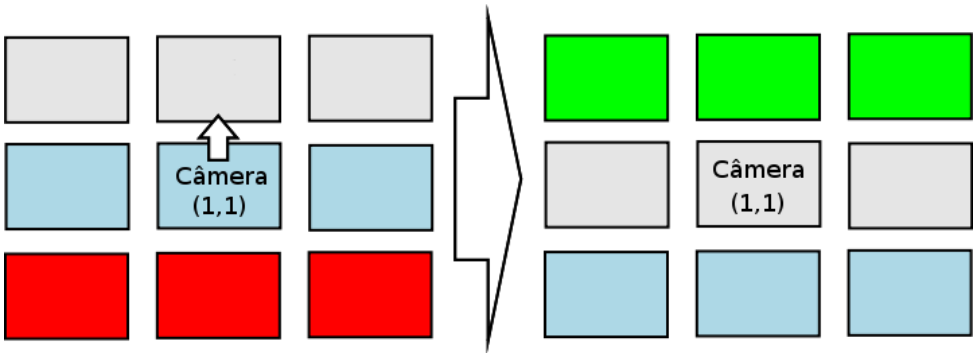

Figura 11: Movimentação da câmera para um outro *patchs*.

Na Figura 11 é possível notar o deslocamento dos mapas de textura quando a câmera move para o *patch* superior ao (1,1). Para que haja uma transição, uma matriz de translação, com valores iguais ao tamanho do *patch*, é feita e multiplicada à matriz *MODELVIEW*, responsável por renderizar todas as primitivas, resultando na translação de todos os *patchs*.

Este método diminuiu a necessidade de implementação de um algoritmo de nível de detalhe mais robusto. Além disso, como sabemos o número de vértices antecipadamente, a performance do aplicativo tem uma menor chance de sofrer quedas bruscas de rendimento.

Após a geração dos vértices, as texturas com os mapas de altura gerados proceduralmente são aplicados à malha. Um *vertex shader* lê então a altura presente no mapa e desloca a posição de z do vértice correspondente na malha.

A cor de cada fragmento é calculada a partir de quatro diferentes texturas, simulando areia, grama, pedras e neve. A participação de cada uma delas na cor final dependerá da altura do vértice correspondente do fragmento. Para pontos mais altos, a textura de neve será predominante e, pontos mais baixos, serão cobertos pela textura de grama. Entre esses pontos, haverá uma mistura das outras texturas.

5 RESULTADOS E DISCUSSÃO

Neste capítulo, será apresentado algumas imagens de terrenos gerados (Seção 5.1) e também algum testes executados (5.2).

5.1 Terrenos gerados

A Figura 12 mostra a renderização de uma cena apenas com a textura do mapa de altura aplicado em um quadrado.

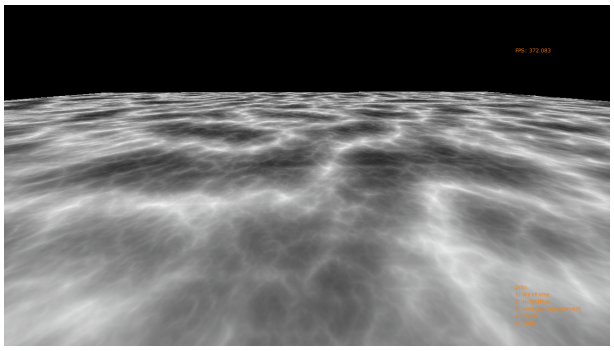

Figura 12: Mapa de altura aplicado a um quadrado.

A Figura 13 mostra o mesmo mapa de altura, mas agora com o deslocamento dos vértices no eixo z.

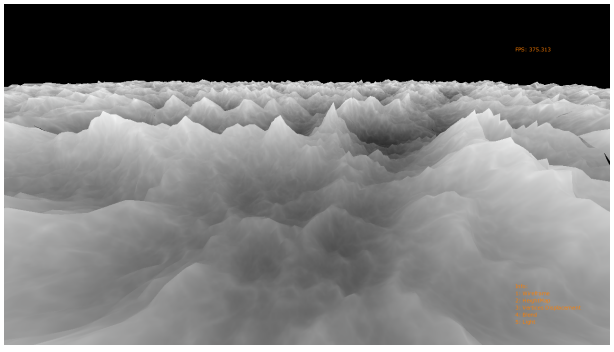

Figura 13: Mapa de altura aplicado a um quadrado, deslocando a altura.

A Figura 14 mostra agora a renderização da malha com cores para simular grama, pedras, neve, etc.

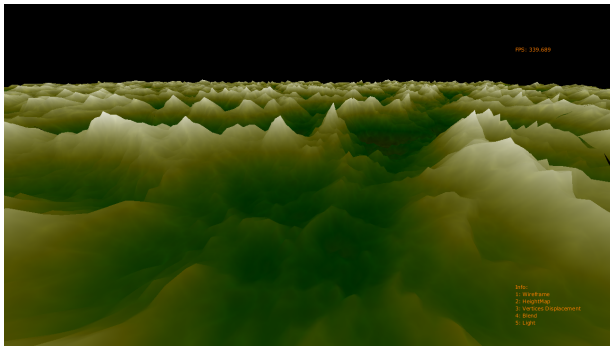

Figura 14: Mapa de altura aplicado a um quadrado, deslocando a altura, e com texturas.

A Figura 15 mostra o resultado final, agora com a aplicação de iluminação.

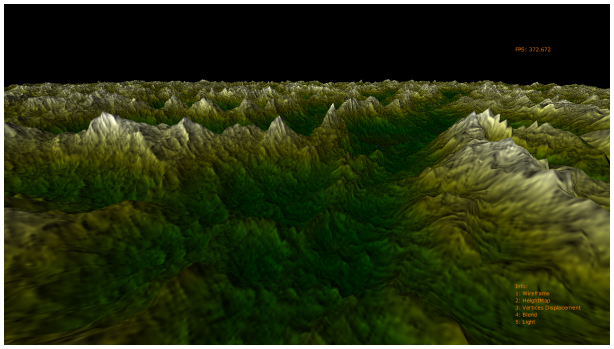

Figura 15: Mapa de altura aplicado a um quadrado, deslocando a altura, com texturas, e iluminação.

Um vídeo demonstrando a navegação pelo terreno pode ser visto em [18].

5.2 Testes de Desempenho

Alguns testes foram feitos para avaliar a velocidade de geração dos terrenos com a alteração de alguns parâmetros, utilizando tanto a GPU quanto a CPU. Eles foram executados em um *Core 2 Duo E7400*, com 2GB de memória *RAM* e placa de vídeo *ATI Radeon HD 4850* com 512MB de memória *RAM*, e *driver* versão 8.612. As tabelas com os tempos e as conclusões dos testes são apresentadas a seguir.

5.2.1 Tempos de Geração do Terreno

Neste teste foi medido o tempo médio gasto com a geração de 100 terrenos proceduralmente tanto na GPU quanto na CPU. Os seguintes parâmetros foram utilizados:

- *Octaves*: Variável (4, 8, 12, 16)
- *Lacunarity*: 2.5
- Ganho: 0.5
- *Offset*: 1.0
- Tamanho da textura: 512
- Número de divisões dos quadrados: 150
- Tamanho dos quadrados: 5.0
- Fator *LOD*: 2

Octaves	GPU	CPU
4	28,7236	46,2948
8	28,7432	92,4299
16	28,7887	187,103
32	30,1095	370,841

Tabela 2: Tempo médio (em ms) de geração dos terrenos, com número variável de *octaves*

A Figura 16 apresenta os tempos anteriores.

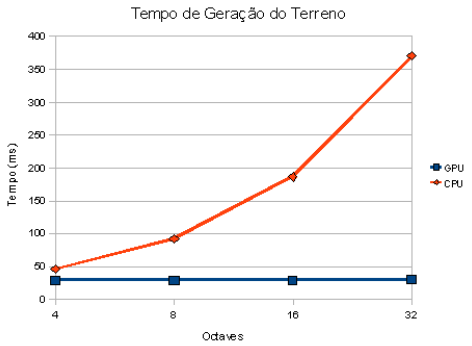

Figura 16: Gráfico do tempo médio (em ms) de geração dos terrenos, com número variável de *octaves*.

Quanto maior o número de *octaves*, maior o número de iterações feitas sobre a função de ruído, como explicado na Seção 2.1.1.4, necessitando assim de um maior processamento. Como pode ser visto, o tempo de geração do terreno na GPU permanece quase constante, enquanto a geração na CPU tem um comportamento praticamente linear.

5.2.2 *Frames* por Segundo Durante Navegação

Neste teste foi medido o *Frames por segundo* (FPS) médio durante a navegação pelo terreno gerado proceduralmente, por 30 segundos. Os seguintes parâmetros foram utilizados:

- *Octaves*: 16
- *Lacunarity*: 2.5
- Ganho: 0.5
- *Offset*: 1.0
- Tamanho da textura: 512
- Número de divisões dos quadrados: 150
- Tamanho dos quadrados: 5.0
- Fator *LOD*: 2

A Tabela 3 mostra os tempos médios, mínimos e a média de *FPS*. Além disso, há o número de *frames* renderizados durante o percurso:

	Total de *Frames*	*FPS* Mínimo	*FPS* Máximo	*FPS* Médio
CPU	9664	248	397	322.133
GPU	11287	361	394	376.233

Tabela 3: Dados sobre a navegação pelo mundo durante 30 segundos

A Figura 17 apresenta os tempos anteriores:

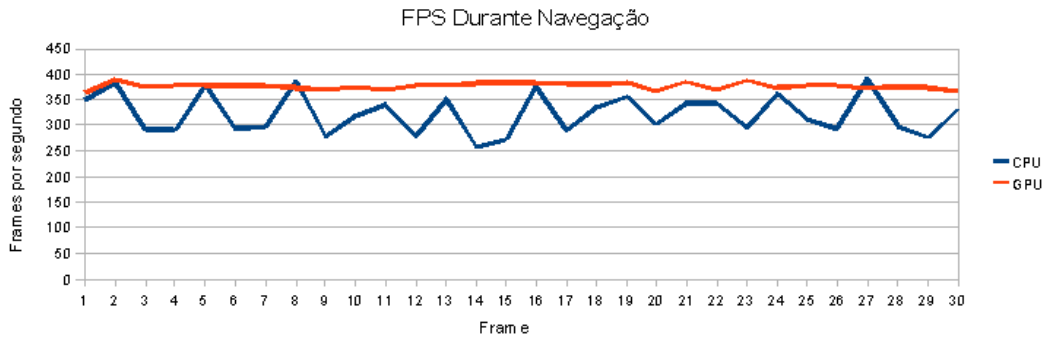

Figura 17: Gráfico com o FPS na navegação pelo mundo durante 30 segundos.

Como pode ser visto através do gráfico, a navegação pelo mundo, utilizando a geração dos terrenos na GPU, é muito mais fluida, sem quedas bruscas de *FPS*, como nos *frames* 3, 6, 9, 12, 14, 17, 20, 23, 26 e 29, graças às centenas de unidades de processamento presentes na GPU.

Na geração pela CPU, tais quedas correspondem justamente aos momentos em que o sistema gera novos terrenos e podem significar um menor senso de imersão do usuário no mundo virtual.

6 CONCLUSÕES E TRABALHOS FUTUROS

Os resultados obtidos na geração procedural de terrenos na GPU mostram o poder de processamento das placas gráficas, em relação às CPU. Este trabalho, porém, não implementou uma alternativa *multi-core* para a geração procedural na CPU. Isto poderia diminuir o tempo gasto na geração. Outro aspecto que pode ser abordado no futuro é o escalonamento entre GPU e CPU, dependendo do nível de ociosidade de cada processador.

Um ponto a ser melhorado na geração procedural é quanto à heterogeneidade do terreno. Apesar de apresentar um resultado satisfatório para um local limitado, quando se olha o terreno como um todo, percebe-se que há uma semelhança muito grande entre os *patchs*, diminuindo assim a ilusão de estar andando em algo realmente infinito.

O sistema foi implementado sempre tendo em mente a sua utilização acoplada a outras aplicativos. Dessa forma, adotá-lo em um simulador ou *game* demandaria pouco esforço, desde que o aplicativo utilize *OpenGL*.

Referências

1 PROGRAMMING with CUDA. Disponível em: <http://theinf2.informatik-.uni-jena.de/For% underline Students-p-9/Lectures/Programming_with_CUDA-p-41-/WS_2008_2009.html>. Acesso em: 25 jun. 2009.

2 FERNANDO, R.; KILGARD, M. J. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321194969.

3 PARISH, Y. I. H.; MüLLER, P. Procedural modelling of cities. In: *in Proc. ACM SIGGRAPH, (Los Angeles, 2001) ACM Press*. [S.l.: s.n.], 2001. p. 301–308.

4 GREUTER, S. et al. Real-time procedural generation of 'pseudo infinite' cities. In: *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. New York, NY, USA: ACM, 2003. p. 87–ff. ISBN 1-58113-578-5.

5 OLSEN, J. Realtime procedural terrain generation. In: *Department of Mathematics And Computer Science (IMADA)*. [S.l.: s.n.], 2004.

6 ZIMMERLI, L.; VERSCHURE, P. Delivering environmental presence through procedural virtual environments. In: *PRESENCE 2007, The 10th Annual International Workshop on Presence*. [S.l.: s.n.], 2007.

7 MOJOWORLD Generator. Disponível em: <http://www.mojoworld.org/>. Acesso em: 25 jun. 2009.

8 SPEEDTREE — IDV, Inc. Disponível em: <http://www.speedtree.com/>. Acesso em: 25 jun. 2009.

9 EBERT, D. S. et al. *Texturing and Modeling: A Procedural Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1558608486.

10 PRUSINKIEWICZ, P.; LINDENMAYER, A. *The algorithmic beauty of plants*. New York, NY, USA: Springer-Verlag New York, Inc., 1996. ISBN 0-387-94676-4.

11 MANDELBROT, B. B. *The Fractal Geometry of Nature*. [S.l.]: W. H. Freeman, 1982. Hardcover. ISBN 0716711869.

12 KEN Perlin's homepage. Disponível em: <http://mrl.nyu.edu/~perlin/>. Acesso em: 25 jun. 2009.

13 STUDENT Lesson Guide. Disponível em: <http://southport.jpl.nasa.gov/cdrom-/sirced03/cdrom/DOCUMENT/HTML/LESSONS/COVERPAG.HTM>. Acesso em: 25 jun. 2009.

14 PERLIN, K. An image synthesizer. *SIGGRAPH Comput. Graph.*, ACM, New York, NY, USA, v. 19, n. 3, p. 287–296, 1985. ISSN 0097-8930.

15 GEISS, R. Gpu gems 3. In: _____. [S.l.]: Addison-Wesley Professional, 2007. cap. 1 Generating Complex Procedural Terrains Using the GPU, p. 7–37.

16 SCHNEIDER, J.; BOLDTE, T.; WESTERMANN, R. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In: *Vision, Modeling and Visualization 2006*. [S.l.: s.n.], 2006.

17 KELLY, G.; MCCABE, H. Citygen: An interactive system for procedural city generation. In: *Game Design & Technology Workshop*. [S.l.: s.n.], 2006.

18 YOUTUBE - Infinite Procedural Terrain Generation using the GPU (POC / DCC / UFMG). Disponível em: <http://www.youtube.com/watch?v=IdYP0PUPQpA>. Acesso em: 25 jun. 2009.