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 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 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 infinitos. Um comparativo com geração de terrenos na *CPU* também é feito, com o objetivo de 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 plans to utilizy the power of the modern video cards (GPU) to generate infinit terrains. A comparative between GPU generation and CPU is made, in order to avaliate the performance of the two different architectures.

Keywords: Procedural content generation, terrain, gpu.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

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

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* (GPU)). Estes microprocessadores, incorporados às placas de vídeo atuais, são especializados em processar gráficos. O avanço da indústria de *games* fez com que as *GPUs* se tornassem cada vez mais rápidas, tornando-as atraente 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 proceduralmente.

O primeiro aspecto (a visualização) é um problema muito estudado no campo da computação. O modelo de um terreno é algo que pode demandar um número extremamente alto de triângulos. 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 comparar a eficiência e velocidade das duas abordagens.

1.3 Organização

Este trabalho está organizado da seguinte maneira. O Capítulo 1 apresenta uma breve introdução e objetivos. O Capítulo 2 apresenta o referencial teórico pertinente a este trabalho. O Capítulo 3 mostra a metodologia envolvida. O Capítulo 4 apresenta os resultados obtidos e, finalmente, o Capítulo 5 mostra a conclusão do trabalho e propostas para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Para ume melhor compreensão deste trabalho, é necessário explicar o contexto em que ele se situa e também pesquisas relacionadas. O referencial teórico pertinente a este trabalho 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 Seçao 2.2 será apresentada 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 ([1] e [2]), outros abordam a geração de terrenos realistas (em tempo-real, como os trabalhos [3] e [4], ou não, como o *MojoWorld* [5]), ou então a geração de árvores [6]. A principal referência na área é o livro *Texturing and Modeling: A Procedural Approach* [7], 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*)[8], que, através de uma gramática, modela o crescimento de plantas; geometrias fractais [9]; e também ruído de Perlin (*Perlin noise* [10]). Algumas técnicas pertinentes a este trabalho serão abordadas na Seção 2.1.1.

2.1.1 Terrenos procedurais

Existem uma série de técnicas para criação de terrenos proceduralmente, como ruído de Perlin, *fractal plasma*, *fault formation*, *circles*. Para este trabalho, porém, foi escolhido o *Ridged multifractal noise*, uma variação do ruído de Perlin, por este ser o que melhor representa terrenos [7].

2.1.1.1 Ruído de Perlin

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

Para criar um ruído de Perlin, precisamos de uma função que retorne, para um dado domínio, números entre 0 e 1. Para gerarmos esses números, é preciso uma semente (*seed*); dessa forma, em uma segunda execução, com uma mesma semente, teremos os mesmos números entre 0 e 1.

A Figura 1 [11] mostra três funções de ruído, criadas com diferentes valores de amplitude e frequência. A primeira função poderia ser uma representação de montanhas, a segunda de morros, a terceira de blocos de pedras. A quarta função é a soma das três funções anteriores. Cada função de ruído é chamada de um *octave*.

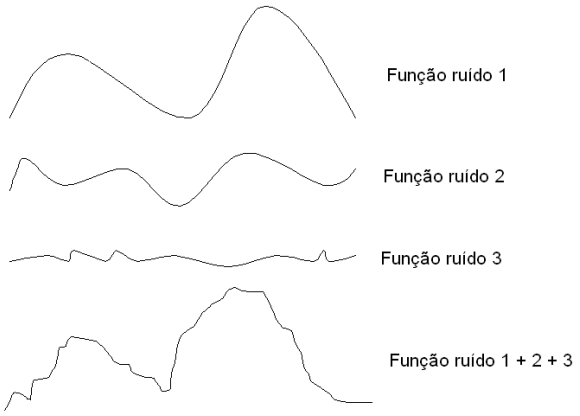

Figura 1: Exemplo de um ruído de Perlin.

A maior vantagem do ruído de Perlin é ele ser coerente, ou seja, com os mesmos valores de entrada, obteremos os mesmos valores de saída.

2.1.1.2 *Ridged multifractal noise*

O *Ridged multifractal noise* é uma variação do ruído de Perlin, e foi apresentado em [7]. O maior 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 o espaço entre sucessivas frequências e, finalmente, **offset** é o valor adicionado ao resultado do *ridged multifractal noise* (elevando o terreno acima do plano $z = 0$).

2.1.2 Mapa de altura

Um mapa de altura (ou *heightmap*) é uma imagem bidimensional que armazena dados referentes à altura de um terreno. Geralmente, tons mais claros representam pontos mais altos, enquanto tons mais escuros são pontos mais baixos do mapa. A Figura 2 [12] é um exemplo de mapa de altura.

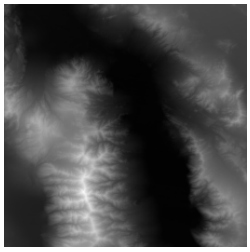

Figura 2: Exemplo de um mapa de altura.

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 3 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.

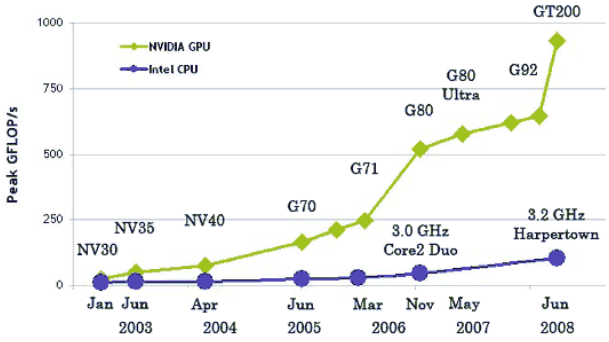

Figura 3: Gráfico com a evolução das *GPUs*, em comparação com as *CPUs*.

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. Assim, certos aspectos de um processador, como *cache*, 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 das do *vertex shader* e fragment shader, sendo estes estágios programáveis.

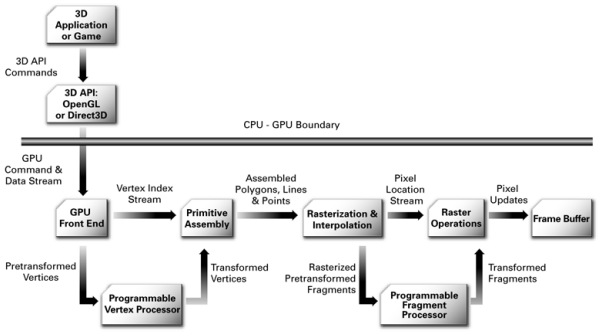

Figura 4: *Pipeline* gráfico.

Ao longo da evolução das *GPUs*, uma série de linguagens para programação de *shaders* foram criadas. Entre elas, podemos ressaltar a HLSL, da *Microsoft*, Cg, da *NVidia*, e GLSL, da 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*). O resultado do processamento também ficará limitado a tais primitivas. O destino da renderização de todas as *APIs* é o chamado *Frame Buffer*, que nada mais é do que um vetor com informações sobre cor, profundidade, etc. Como é possível criar novos *Frame buffers*, também é possível criar novos destinos de renderização para os *shaders*.

2.3 Geração Procedural utilizando *GPUs*

A geração procedural de terrenos na *GPU* já foi abordada nos trabalhos [13] e [14]. O primeiro faz uso da plataforma CUDA, da empresa *NVidia*, para a geração procedural, e, 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 ([14]) 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 VBO, que armazena os vértices do modelo 3D diretamente na memória da *Graphics Processing Unit* (GPU), diminuindo o número de chamadas para a renderização, e o 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 grande número de computadores.

O segundo passo foi pesquisar diversas técnicas procedurais envolvidas na geração de terrenos, bem como algumas sistemas que oferecem soluções ligadas à geração procedural, como o *CityEngine* [15] e o *MojoWorld* [5]. 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 RESULTADOS E DISCUSSÃO

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 tal sistema, bem como decições e detalhes de implementação. Finalmente, na Seção 4.3 será apresentado alguns exemplos de terrenos gerados proceduralmente.

4.1 O Sistema Implementado

O sistema implementado neste trabalho teve como principal objetivo permitir a geração procedural de terrenos tanto na *GPU* quanto na *CPU*. A Figura 5 apresenta as camadas do sistema, destacando as bibliotecas utilizadas (como é explicado a seguir).

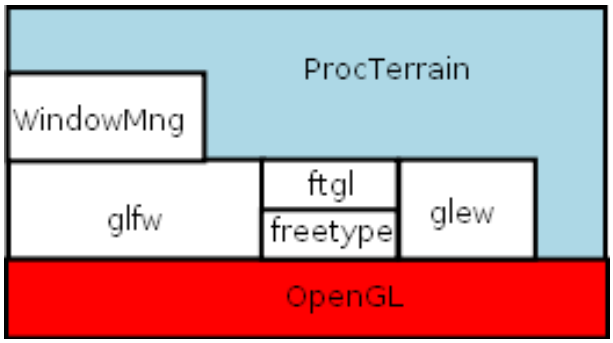

Figura 5: Camadas do sistema.

- **OpenGL**: *API* gráfica utilizada para a renderização gráfica.
- **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 proposito 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 vir a ser desenvolvidos.

A Figura 6 apresenta em detalhes os módulos presentes nas camadas *WindowMng* e *ProcTerrain*. A seguir, uma explicação sobre cada um dos módulos.

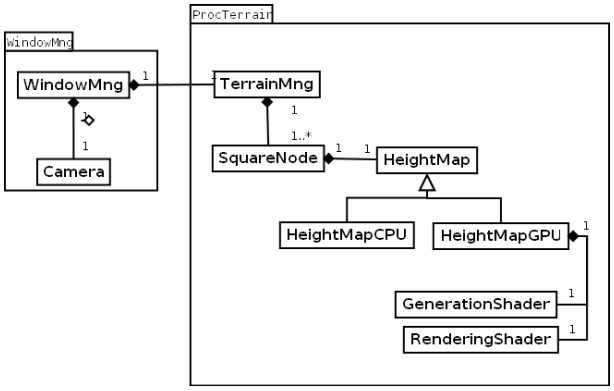

Figura 6: Diagrama com as principais classes do sistema implementado.

- **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 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.

4.1.1 Geração do Terreno

Nesta seção, será abordado 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*).

4.1.1.1 Geração do Terreno na *GPU*

Toda a geração dos terrenos na *GPU* é feita através de um *fragment shader*. Como toda computação de *shaders* só pode ser aplicada com base em geometrias ou texturas, foi preciso renderizar um quadrado. Entra aí então o FBO, extensão do *OpenGL* que permite criar *Frame Buffers* e realizar renderizações *off-screen* (que não são exibidas na tela). O quadrado é renderizado em um novo *frame buffer*, e o *shader* de geração procedural é aplicado sobre ele. O resultado pode ser visto na Figura 7.

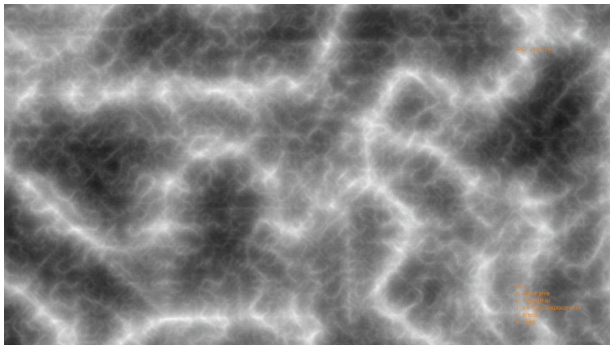

Figura 7: Mapa de altura gerado pelo *shader*.

Um aspecto importante é que, durante a geração do mapa de altura, os valores das normais de cada vértice também são caulados.

4.2 Geração do Terreno na *CPU*

A geração na *CPU* é feita de maneira tradicional. Uma lista de vértices é criada e, para cada vértice, é calculado a sua altura de acordo com o algoritmo *Ridged multifractal noise*.

4.2.0.2 Visualização do Terreno

Com o mapa de altura gerado, o próximo passo é exibir o terreno para o usuário. Da mesma forma, aqui também será exposto alguns detalhes específicos da visualização

do terreno gerado na *GPU*, na Seção 4.2.0.3. Antes, porém, é necessário detalhar funções comuns tanto à visualização de terrenos gerados na *CPU* quanto na *GPU*.

O passo inicial é a geração de uma malha (conjunto de vértices) de tamanho pré-determinado, como mostra a Figura 8.

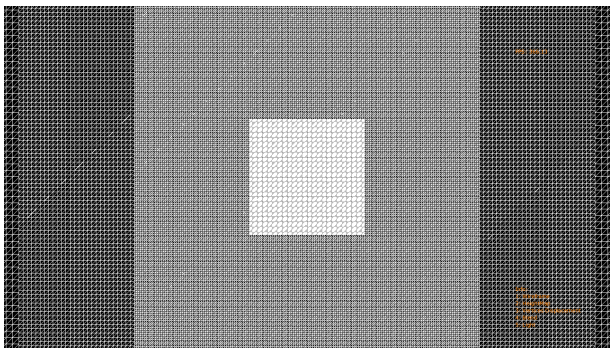

Figura 8: Malha inicial para visualização dos terrenos.

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 são trocados, como mostra a Figura 9

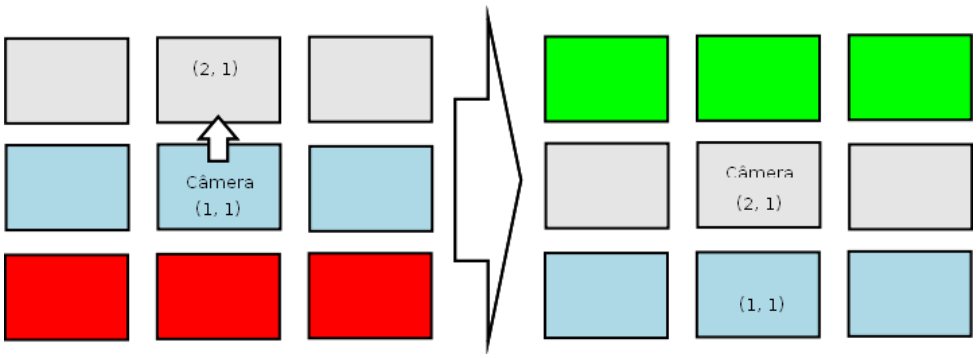

Figura 9: Movimentação da câmera do nodo (1,1), na esquerda, para o nodo (2, 1), na direita.

Na Figura 9 é possível notar o deslocamento dos mapas de textura quando a câmera se move do nodo (1,1) para o nodo (2,1). Este método, possível somente para terrenos gerados na *GPU*, 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.

4.2.0.3 Visualização do Terreno Gerado na *GPU*

Para a visualização do terreno gerado na *GPU*, um *vertex shader* lê a altura do mapa de altura gerado e desloca a posição de z do vértice correspondente na malha. A iluminação é feita na própria

4.3 Terrenos gerados

Nesta Seção, será apresentado uma série de imagens com o resultado das gerações procedurais na *GPU*.

A Figura 10 mostra a renderização de uma cena apenas com a textura do mapa de altura aplicado em um quadrado.

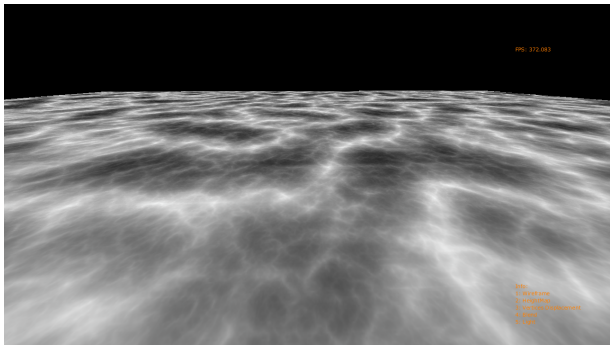

Figura 10: Mapa de altura aplicado a um quadrado.

A Figura 11 mostra o mesmo mapa de altura, mas agora com o deslocamento dos vértices em no eixo z.

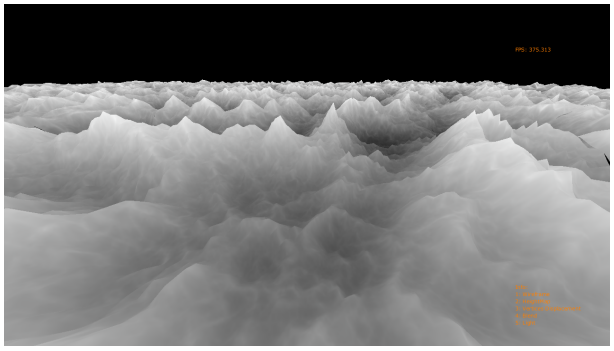

Figura 11: Mapa de altura aplicado a um quadrado, deslocando a altura.

A Figura 12 mostra agora a renderização da malha com texturas para simular o grama, pedras, neve, etc.

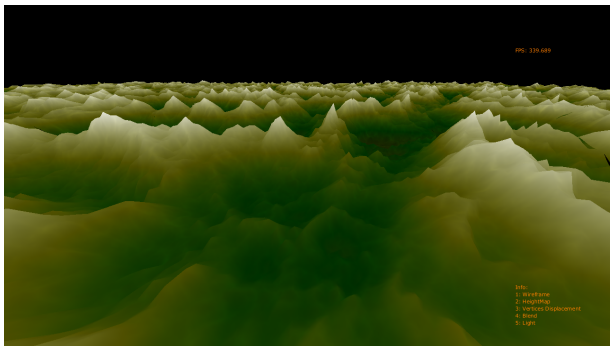

Figura 12: Mapa de altura aplicado a um quadrado, deslocando a altura, e com texturas.

A Figura 13 mostra o resultado final, agora com a aplicação de iluminação.

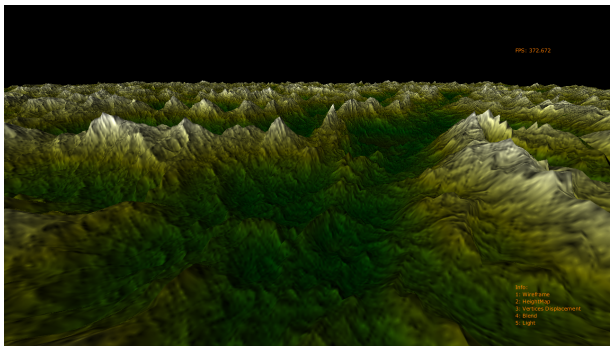

Figura 13: Mapa de altura aplicado a um quadrado, deslocando a altura, com texturas, e iluminação.

4.4 Testes

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.

A Seção 4.4.1 mostra um teste com a geração de 100 terrenos proceduralmente.

4.4.1 Tempos de Geração do Terreno

Neste teste foi medido o tempo gasto com a geração de terrenos tanto na *GPU* quanto na *CPU*. Para cada número de *octaves* (4, 8, 12, 16), foi gerado 100 terrenos, e medido o tempo gasto. Quaisquer outros parâmetros (número de vértices, tamanho de texturas, números de vizinhos) foram mantidos iguais para a geração nas duas arquiteturas. Os resultados obtidos estão na tabela 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 média (em ms) de geração dos terrenos, com número variável de *octaves*

A Figura 14 apresenta os tempos anteriores.

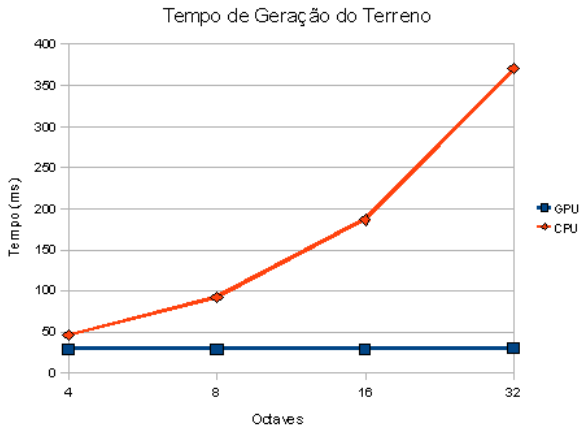

Figura 14: Gráfico do tempo médio (em ms) de geração dos terrenos, com número variável de *octaves*.

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 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 certamente diminuiria 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 terrenos, diminuindo assim a ilusão de estar andando em um terreno realmente infinito.

O sistema foi implementado sempre tendo em mente a sua utilização acoplado a outras aplicativos. Dessa forma, adotá-lo em um simulador ou *game* demandaria pouco esforço.

Referências

1 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.

2 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.

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

4 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.

5 MOJOWORLD Generator. Disponível em: <http://www.mojoworld.org/>. Acesso em: 23 nov. 2008.

6 SPEEDTREE — IDV, Inc. Disponível em: <http://www.speedtree.com/>. Acesso em: 23 nov. 2008.

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

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

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

10 KEN Perlin's homepage. Disponível em: <http://mrl.nyu.edu/~perlin/>. Acesso em: 23 nov. 2008.

11 ACMC Projects, CG Rendering of Coral at The University of Queensland. Disponível em: <http://www.acmc.uq.edu.au/Projects/CG_Rendering.html>. Acesso em: 23 nov. 2008.

12 STUDENT Lesson Guide. Disponível em: <http://southport.jpl.nasa.gov/cdrom-/sirced03/cdrom/DOCUMENT/HTML/LESSONS/COVERPAG.HTM>. Acesso em: 23 nov. 2008.

13 NGUYEN, H. *GPU Gems 3*. [S.l.]: Addison-Wesley Professional, 2007. Hardcover. ISBN 0321515269.

14 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.

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