

# Um Sistema para Geração Procedural de Terrenos Pseudo-Infinitos em Tempo-Real Utilizando GPU e CPU

60334

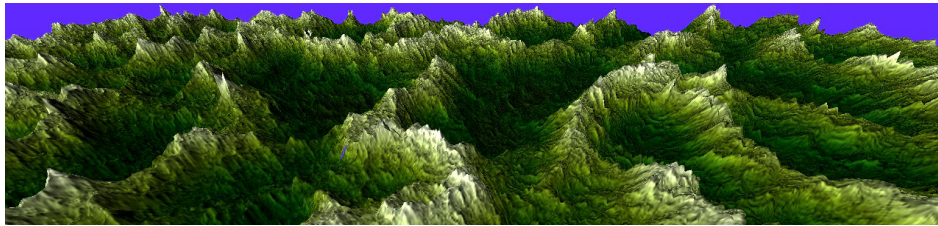


Figura 1: Terreno gerado proceduralmente.

## Resumo

O rápido crescimento do poder de processamento das placas gráficas fez com que diversas tarefas migrassem da CPU para a GPU. Porém, as unidades de processamento gráfico podem ser vistas como aliadas da CPU, e não rivais. Este trabalho propõe um sistema *multithread* que utiliza tanto a CPU quanto a GPU para minimizar o tempo gasto com a geração procedural de terrenos e permitir uma navegação fluida através de um mundo pseudo-infinito gerado proceduralmente.

Ao final, uma comparação é feita com base em experimentos utilizando diferentes modelos de geração, com o objetivo de expor suas vantagens e desvantagens.

**Keywords::** modelagem procedural; gpu; gpgpu; programação paralela

**Author's Contact:**

## 1 Introdução

A técnica de geração procedural é uma área da Ciência da Computação que propõe o uso de rotinas e algoritmos para a geração de algum tipo de conteúdo, seja ele um modelo tridimensional, música, animações, etc. Tal técnica vem se tornando bastante popular nos últimos tempos, considerando 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. Assim, técnica de geração procedural vem como uma alternativa à utilização do trabalho de artistas e modeladores na criação de modelos tridimensionais.

A área de geração procedural envolve uma pesquisa constante por algoritmos que sintetizem certos aspectos da realidade ao nosso redor, sejam terrenos [Ebert et al. 2002], cidades [Chen et al. 2008], ou vegetação [Prusinkiewicz and Lindenmayer 1996]. Paralelo a isso, há uma outra vertente de pesquisa que busca executar tais algoritmos de modo que eles possam ser utilizados em aplicações interativas de tempo-real.

Este trabalho propõe um sistema de geração procedural de terrenos que utilize tanto a CPU quanto a GPU, de maneira isolada ou combinada, com cada arquitetura ficando responsável por uma determinada porcentagem da carga de trabalho referente à geração. Através do sistema, será possível navegar em um terreno pseudo-infinito de forma interativa e sem quedas bruscas do *frame rate*.

O trabalho está organizado da seguinte maneira: na Seção 2 são apresentados alguns trabalhos relacionados e as principais contribuições deste trabalho. A Seção 3 revisa alguns conceitos pertinentes. A Seção 4 apresenta o sistema proposto para a geração procedural utilizando tanto a CPU quanto a GPU e a Seção 5 detalha sua implementação. A Seção 6 faz uma discussão sobre os

experimentos realizados. Finalmente, a Seção 7 apresenta a conclusão e a perspectiva para futuros trabalhos.

## 2 Trabalhos relacionados

A modelagem procedural de terrenos é uma área vastamente pesquisada, com diversos trabalhos que buscam sempre criar os terrenos mais realistas possíveis.

Uma das técnicas de modelagem procedural de terrenos é o ruído Perlin [Perlin 1985], uma função pseudo-aleatória que, dada uma entrada (posição), retorna um valor que possui uma suave transição com os seus vizinhos. Em [Perlin 2004] foi apresentado um ruído Perlin otimizado, que buscou adaptar o ruído às novas arquiteturas (GPUs), melhorar as propriedades visuais e introduzir uma única versão do ruído que retornaria os mesmos valores independentemente da plataforma de *hardware* ou *software*.

Em [Ebert et al. 2002] são apresentados alguns algoritmos que fazem uso do ruído Perlin e que são capazes de gerar terrenos de uma forma significativamente realista. Podemos citar os algoritmos *fBm*, *heterogenous terrain*, *hybrid multifractal* e *ridged multifractal*, sendo que este último foi o algoritmo utilizado neste trabalho.

Em [Cordeiro and Chaimowicz ], os autores apresentam um paradigma para a modelagem procedural (terrenos, vegetação, etc.) utilizando múltiplas *threads*. Uma implementação é proposta utilizando apenas as unidades de processamento disponíveis na CPU.

A geração procedural utilizando a GPU foi explorada em [Geiss 2007] e [Schneider et al. 2006]. O primeiro trabalho, faz uso de *geometry shaders* e está limitado às placas de vídeo com suporte a DirectX 10. O segundo trabalho, mais abrangente quanto às placas de vídeo suportadas, gera os terrenos na GPU com o uso de algoritmos multifractais (semelhante ao que é proposto aqui). Nenhum dos dois trabalhos, porém, faz uma comparação entre implementações de geração de terrenos utilizando a CPU e a GPU, e também não buscam uma plataforma que utilize as duas arquiteturas.

Portanto, o trabalho apresentado aqui possui as seguintes contribuições:

- Propor um sistema de geração procedural de terrenos que faça uso tanto da CPU quanto da GPU.
- Fazer uma comparação entre a eficiência da geração na CPU e na GPU.

## 3 Conceitos básicos

A modelagem procedural de terrenos pode ser feita através da construção de mapas de altura dos terrenos. Para a geração destes mapas, as técnicas mais utilizadas são as baseadas em ruídos, como será apresentado a seguir.

### 3.1 Ruído Perlin

O ruído Perlin foi criado pelo Professor Ken Perlin [Perlin 1985], 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; 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:

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

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 branco (puramente aleatório).

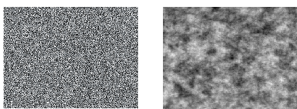


Figura 2: Esquerda: Ruído aleatório. Direita: Ruído Perlin.

As características fundamentais do ruído Perlin são 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.

### 3.2 Fractais

Fractais podem ser descritos, segundo [Ebert et al. 2002], 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:

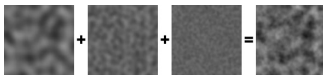


Figura 3: Exemplo de um fractal a partir de ruído Perlin.

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.

### 3.3 Ridged Multifractal

O *Ridged multifractal* é um modelo para geração de terrenos apresentado em [Ebert et al. 2002]. O seu principal ponto é a captura da *heterogeneidade de terrenos em grande escala*, apresentando montanhas, planaltos e crateras. Os seguintes parâmetros são levados em consideração na execução do algoritmo:

- **Octaves:** Número de iterações (e, conseqüentemente somas) feitas sobre a função de ruído.
- **Amplitude:** Máximo valor adicionado ao valor total do ruído.
- **Frequency:** Número de valores de ruídos definidos entre dois pontos (quanto maior a frequência, maior o distúrbio 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.
- **Offset:** Fator multifractal.
- **Tamanho:** Tamanho do mapa de altura que será salvo o resultado da geração procedural.

O tempo de execução é dependente apenas do tamanho do mapa e do número de octaves.

## 4 Metodologia Proposta

O sistema proposto tem como objetivo gerar proceduralmente os terrenos tanto na GPU quanto na CPU a medida em que o usuário percorra o terreno. Portanto, foi preciso estabelecer uma base comum às duas arquiteturas.

O terreno geral é dividido em terrenos retangulares menores (chamados *patches*), como mostra o *grid* da Figura 4. Dessa forma, apenas *patches* de interesse do usuário (que estão mais próximos, por exemplo) precisarão ser gerados.

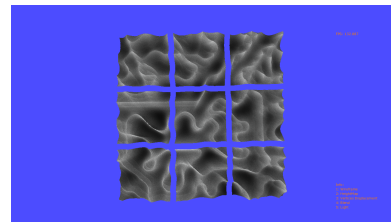


Figura 4: Patches exibidos em um grid.

Considerando o usuário inicialmente localizado no *patch* central, ao mover-se para um *patch* vizinho, o sistema irá requisitar a geração de novos *patches*, 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.

Para garantir uma visualização fluida do terreno, minimizando as interrupções com a geração, o sistema proposto decidirá qual arquitetura (GPU ou CPU) será utilizada na geração dos *patches* a partir de uma variável  $\alpha$ , que representa a porcentagem de gerações que ocorrerão na GPU.  $1 - \alpha$  representará, portanto a porcentagem de gerações na CPU.

A Figura 5 mostra como se dá o fluxo de geração.

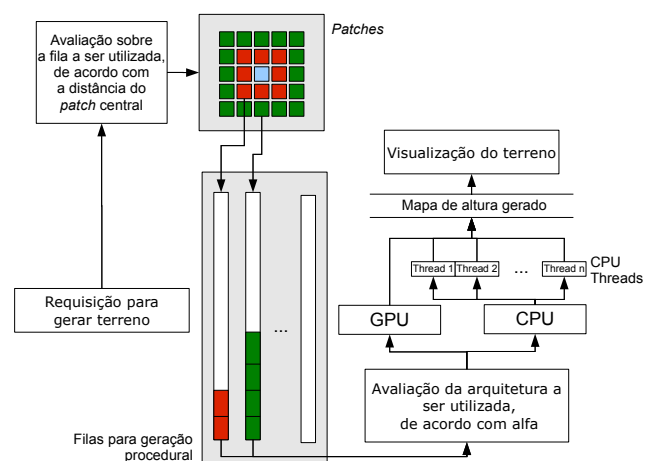


Figura 5: Fluxo da geração procedural.

As requisições por novos terrenos serão adicionadas a uma fila e uma política *First In, First Out* (FIFO) será utilizada para decidir qual terreno será gerado. Como é possível ver na Figura 5, o número de filas existentes no sistema será igual ao número de vizinhos do *patch* central. Dessa forma, é possível decidir quais terrenos serão gerados a partir de sua distância da câmera.

A *thread 0* (principal) ficará encarregada da requisição para gerar novos eventos, avaliação da fila, avaliação da arquitetura a ser utilizada (GPU ou CPU), e também será responsável pelas chamadas às funções OpenGL, incluindo aquelas responsáveis por iniciar a execução das instruções que serão executadas na GPU. A geração na CPU ocorrerá em outras *threads*, não a principal.

## 4.1 O Cálculo de $\alpha$

O valor da variável  $\alpha$  é, atualmente, controlado manualmente pelo usuário. A sua variação de acordo com a utilização de cada arquitetura será um tema a ser abordado em trabalhos futuros.

Atualmente, a maior dificuldade para medir o tempo de geração tanto na GPU é a falta de um padrão nas extensões disponíveis em OpenGL. A extensão `GL_EXT_timer_query`, por exemplo, só está disponível em placas NVidia, algo que anularia a possibilidade da execução deste trabalho em placas ATI.

A utilização de chamadas como `glFinish()` para sincronizar a CPU e a GPU e assim medir o tempo de geração dos terrenos poderia prejudicar a performance do sistema, já que pára a execução da CPU enquanto todos os comandos OpenGL não forem executados.

Uma outra opção para a sincronização seria a extensão `GL_NV_fence`, que oferece funções para sincronização semelhantes ao `glFinish()` e `glFlush()`, porém com um grau maior de controle sobre quais comandos OpenGL deverão ser executados na chamada. Mais uma vez, porém, a extensão não está disponível para placas ATI.

## 4.2 Geração

Toda a geração dos terrenos na GPU é feita através de um *fragment shader* (versão 3.0), utilizando o ruído Perlin como foi proposto em [Perlin 2004]. 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 extensão 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.

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.

A geração utilizando a CPU é feita utilizando o mesmo algoritmo implementado na GPU. Como o mapa de altura gerado reside na memória principal, sua renderização dependerá da transferência para a memória da placa de vídeo.

## 4.3 Visualização

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 6. 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 algoritmo de nível de detalhe (quanto maior a distância do centro, menor será a necessidade de se renderizar o terreno em alta fidelidade).

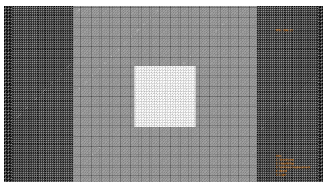


Figura 6: Malha inicial para visualização dos terrenos.

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 7

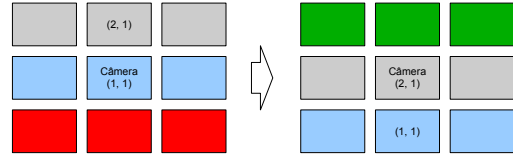


Figura 7: Movimentação da câmera para um outro patches.

Na Figura 7 é possível notar o deslocamento dos mapas de altura quando a câmera move para a *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 responsável por renderizar todas as primitivas, resultando na translação de todos os *patches*.

Este algoritmo de nível de detalhe se mostrou bastante eficaz para este trabalho, não sendo necessária a implementação de um algoritmo LOD mais robusto. Além disso, como sabemos o número de vértices antecipadamente, a performance do aplicativo tem um menor chance de sofrer quedas bruscas de rendimento.

## 5 Implementação

A Figura 10 apresenta as camadas do sistema, bem como as bibliotecas utilizadas.

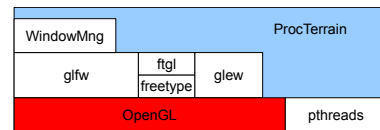


Figura 10: Camadas do sistema.

A camada *WindowMng* tem como propósito simular a 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 (ou acoplado a uma *engine*).

A Figura 11 apresenta em detalhes os módulos presentes nas camadas *WindowMng* e *ProcTerrain*. A seguir, uma explicação sobre cada um dos módulos.

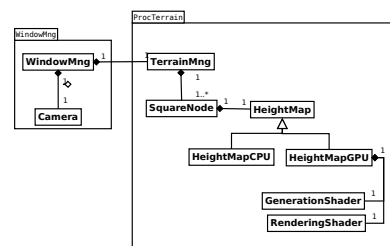
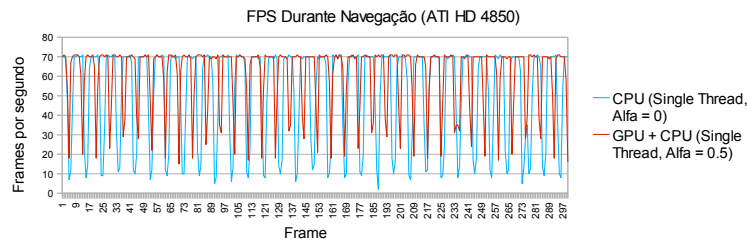
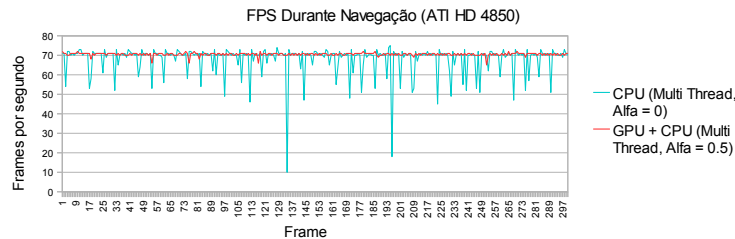


Figura 11: Diagrama com as principais classes do sistema implementado.

- **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.



**Figura 8:** Gráfico com o FPS na navegação pelo mundo durante 300 segundos, utilizando uma única thread na CPU.



**Figura 9:** Gráfico com o FPS na navegação pelo mundo durante 300 segundos, utilizando múltiplas threads na CPU.

- **HeightMap:** Módulo que implementa os mapas de altura dos terrenos gerados na CPU ou na GPU. Possui os métodos **GenerateGPU** e **GenerateCPU**, que são executados em paralelo à *thread* principal.
- **GenerationShader:** *Shader* responsável pela geração dos terrenos.
- **RenderingShader:** *Shader* responsável pela renderização dos terrenos.

## 6 Resultados Experimentais

Para verificar a eficiência das gerações na CPU e GPU, foi feito um experimento que consiste na navegação por um trajeto constante durante 300 segundos. O computador utilizado foi 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 seguintes configurações foram utilizadas para a geração procedural: 8 *octaves*, *lacunarity* igual a 2.5, *gain* igual a 0.5, *offset* como 0.9, número de vizinhos referente ao *patch* central igual a 2, tamanho de textura 256. A resolução do programa foi de 1280 x 720.

Quatro experimentos foram executados, cada um com uma configuração diferente de utilização da CPU e GPU. Os dois primeiros experimentos (Figura 8) foram executados utilizando a GPU e uma única *thread* na CPU. Os experimentos três e quatro (Figura 9) utilizaram a GPU e múltiplas *threads* na CPU. A carga de geração dos terrenos foi dividida igualmente entre as duas arquiteturas em todos os experimentos ( $\alpha = 0.5$ ).

Como mostra os experimentos exibidos na Figura 8, a geração utilizando uma *thread* na CPU possui quedas bruscas no *framerate*, com momentos em que este chega até a 2 FPS; O *framerate* médio fica em 51.79. Com a geração conjunta na GPU, as quedas são sensivelmente menores, nunca caindo abaixo dos 15 FPS, e o *framerate* médio fica em 60.94.

Utilizando uma configuração do sistema com múltiplas *threads* (Figura 9), é possível notar algumas quedas do *framerate*, embora a navegação seja consideravelmente mais fluida do que a obtida utilizando apenas uma única *thread*. O *framerate* mínimo nessa situação foi de 10 FPS, e a média durante a navegação foi de 68.08 FPS. Utilizando a geração em conjunto na GPU, o *framerate* nunca ficou abaixo de 65 FPS, e a média foi de 70.45 FPS.

## 7 Conclusão 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. A opção de se gerar nas duas arquiteturas mostra-se promissor, principalmente considerando a utilização do sistema acoplado a um *game* ou simulador. Como haverá inevitavelmente outras tarefas sendo executadas (*path-finding*, sombras, HDR), a possibilidade de se migrar a carga de trabalho envolvida na geração procedural pode ser bastante vantajosa.

Como trabalho futuro, espera-se encontrar uma estratégia eficiente para o cálculo de  $\alpha$ . Dessa forma, a geração procedural poderá ser balanceada automaticamente entre as diferentes arquiteturas utilizadas (CPU e GPU).

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

## Referências

- CHEN, G., ESCH, G., WONKA, P., MULLER, P., AND ZHANG, E. 2008. Interactive procedural street modeling. *ACM Trans. Graph.* 27, 3.
- CORDEIRO, C. S., AND CHAIMOWICZ, L. Parallel lazy amplification: Real-time procedural modeling and rendering of multi-terabyte scenes on a single pc. *VII Brazilian Symposium on Computer Games and Digital Entertainment*, 2008, Belo Horizonte. *SBGames 2008*.
- EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 2002. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- GEISS, R. 2007. *GPU Gems 3*. Addison-Wesley Professional, ch. 1 Generating Complex Procedural Terrains Using the GPU, 7–37.
- PERLIN, K. 1985. An image synthesizer. *SIGGRAPH Comput. Graph.* 19, 3, 287–296.
- PERLIN, K. 2004. Implementing improved perlin noise. In *GPU Gems*, R. Fernando, Ed. Addison Wesley Professional, March, ch. 13.
- PRUSINKIEWICZ, P., AND LINDENMAYER, A. 1996. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA.
- SCHNEIDER, J., BOLDTE, T., AND WESTERMANN, R. 2006. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Vision, Modeling and Visualization 2006*.