Universidade Federal Fluminense

Bruno José Dembogurski

Geração Procedural de Terrenos através de Extração de Isosuperfícies na GPU

NITERÓI

2009

Universidade Federal Fluminense

Bruno José Dembogurski

Geração Procedural de Terrenos através de Extração de Isosuperfícies na GPU

Dissertação de **Mestrado** *submetida* ao "Programa de Pós-Graduação em Computação"da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces.

Orientador:
Prof. Esteban Walter Clua, D.Sc.

NITERÓI

2009

Geração Procedural de Terrenos através de Extração de Isosuperfícies na GPU

Bruno José Dembogurski

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces.

Aprovada por:

Prof. Esteban Walter Gonzalez Clua, D.Sc. / IC-UFF
(Orientador)

Prof. Marcelo B. Vieira, D.Sc. / DCC-UFJF
(Co-Orientador)

Prof. Marcelo Dreux, Ph.D. / PUC-RIO

Prof. Anselmo Antunes Montenegro, D.Sc. / IC-UFF

Prof. Fabiana Rodrigues Leta, D.Sc. / UFF

"Aqueles a quem muito é dado, muito é exigido"

Lucas 12:48

Dedico este trabalho a minha família, pois sem eles nada disso seria possível.

Agradecimentos

Primeiramente agradeço a Deus por tudo que me proporcionou e as possibilidades que me apresentou.

Agradeço a minha família. Minha mãe e meu irmão, pois sempre me apoiaram e me ajudaram em todas as minhas escolhas e nas horas de grande dificuldade.

Agradeço também pelos meus amigos Carlos, Edelberto, Felipe, Gustavo, Lucas e Tadeu que passaram todas as dificuldades (alguns mais dificuldades outros menos) deste mestrado junto comigo e todos os acontecimentos em nossa república.

Agradeço aos meus orientadores Esteban e Marcelo pela paciência e por terem acreditado em mim.

Um grande agradecimento ao pessoal do MediaLab, mesmo que eu não apareça tanto lá.

Agradeço aos outros professores que me passaram um grande conhecimento durante o curso.

Agradeço a ajuda de Christopher Dyken e Gernot Ziegler por toda a ajuda com minhas dúvidas.

Agradeço a CAPES pela ajuda financeira, pois sem ela seria muito difícil terminar este curso.

Resumo

Esta dissertação apresenta uma forma de geração procedural de terrenos através de extração de isosuperfícies na GPU, desenvolvendo todos os conceitos básicos relativos a funções procedurais, extração de isosuperfícies e geração de terrenos. Também discute as técnicas e funções mais conhecidas, e justificando os algoritmos escolhidos para a implementação desenvolvida durante esse trabalho. A técnica utilizada é o conhecido algorítmo de *Marching Cubes*, mas implementado utilizando o método de *Histogram Pyramids* que acelera o processo de extração.

Palavras-chave: Geração Procedural, Geração de Terrenos, Extração de Isosuperfícies, GPU.

Abstract

This dissertation presents a procedural terrain generation using isosurface extraction at GPU level, by showing all basic concepts related with procedural functions, isosurface extraction and terrain generation. It also discusses traditional and well-known techniques and functions, justifies the algorithms chosen for the implementation developed during this work. The technique used is the well known algorithm of Marching Cubes, but implemented with the Histogram Pyramids approach which speeds up the extraction process.

Keywords: Procedural Generation, Terrain Generation, Isosurface Extraction, GPU.

Abreviações

CPU	:	Central Processing Unit
CUDA	:	Compute Unified Device Architecture
fBm	:	Fractal Brownian Motion
GPU	:	Graphic Processing Unit
GLSL	:	OpenGL Shader Language
VBO	:	Vertex Buffer Object
FBO	:	Frame Buffer Object

Sumário

Lista de Figuras **xi**

Lista de Tabelas **xiv**

Lista de Algoritmos **15**

1 Introdução **1**

1.1 Visão geral do problema 2

1.2 Contribuições 4

1.3 Trabalhos Relevantes 5

1.4 Organização da Dissertação 7

2 Ruído Coerente e Geração Procedural de Terrenos **9**

2.1 Funções Procedurais 9

2.1.1 Funções Básicas 9

2.1.1.1 Função de Ruído 10

2.1.1.2 *Wavelet Noise* 12

2.1.2 Fractais 13

2.1.2.1 Turbulência 14

2.1.2.2 *Fractal Brownian Motion - FBM* 16

2.1.2.3 Multifractais 17

2.2 Geração Procedural de Terrenos 17

3 Extração de Isosuperfícies **20**

3.1 *Marching Cubes* 20

3.2 *Histogram Pyramids* 25

3.3 *Marching Cubes* utilizando *Histogram Pyramids* 28

4 Geração Procedural de Terrenos por Extração de Isosuperfícies **32**

4.1 Introdução 32

4.2 Funções de Ruído na GPU 33

4.3 Extração do Terreno 36

5 Resultados **40**

6 Conclusão e Trabalhos Futuros **51**

Referências Bibliográficas **53**

Referências **53**

Apêndice A – Códigos **56**

A.1 Shader das Funções de Ruído e Turbulência 56

A.2 Construção da Pirâmide 60

Lista de Figuras

2.1 Grade no espaço R^3 10

2.2 Interpolações com a função de suavização 11

2.3 Comparação entre ruído de Perlin e *Wavelet noise* 13

2.4 Soma fractal da função de ruído 14

2.5 Turbulência 15

2.6 Perturbação utilizando a função de turbulência 16

2.7 Fractal Brownian Motion com coeficiente de Hurst = 0.2 e 0.6, respectivamente 17

2.8 Terreno gerado com a função de ruído de Perlin 18

2.9 Terreno gerado com a função de turbulência 19

2.10 Relação entre mapa de altura e terreno 19

3.1 Nomeclatura dos vértices da célula de *Marching Cubes* 21

3.2 Padrões existentes no algoritmo de *Marching Cubes* 21

3.3 Subdivisões do algoritmo de *Marching Tetrahedra* 22

3.4 Grade do *Marching Tetrahedra* 23

3.5 Compactação dos dados. 26

3.6 Extração da sequência de saída 27

3.7 Fluxograma da abordagem de *Marching Cubes* com *Histogram Pyramids* . 28

4.1 Tabela de vetores gradientes na forma de textura 34

5.1 Terreno gerado utilizando apenas a função de ruído 40

5.2 Terreno gerado com a função de turbulência 41

5.3 Terreno gerado com *warping* de coordenada 42

5.4 Terreno alienigena também gerado com *warping* de coordenada 43

5.5 Terreno gerado com 4 iterações fractais 45

5.6 Terreno gerado com 7 iterações fractais 45

5.7 Terreno com densidade 32^3 46

5.8 Terreno com densidade 64^3 46

5.9 Terreno com densidade 128^3 46

5.10 Terreno com densidade 256^3 46

5.11 Ganho igual a 0.0 47

5.12 Ganho igual a 0.3 47

5.13 Ganho igual a 0.5 47

5.14 Ganho igual a 0.6 47

5.15 Ganho igual a 0.0 48

5.16 Ganho igual a -0.3 48

5.17 Ganho igual a -0.5 48

5.18 Ganho igual a -0.6 48

5.19 lacunaridade igual a 0.0 49

5.20 lacunaridade igual a 1.5 49

5.21 lacunaridade igual a 2.0 49

5.22 lacunaridade igual a 2.4 49

5.23 Isovalor de -1.5 50

5.24 Isovalor de -1.65 50

5.25 Isovalor de -1.95 50

5.26 Isovalor de -2.1 50

Lista de Tabelas

5.1 Comparação entre as diferentes densidades utilizando a função de turbulência (7 iterações) 45

5.2 Comparação entre as diferentes densidades utilizando a função de ruído . . 45

Lista de Algoritmos

1 Tabela de permutação 34

2 Tabela de vetores gradientes 34

3 Função de ruído de Perlin em GLSL 35

4 Função de turbulência 36

5 Inicialização do terreno, variáveis e texturas do *shader* 38

Capítulo 1

Introdução

A geração procedural é importante na área de computação gráfica devido às várias possibilidades que apresenta. Vários modelos podem ser criados facilmente através deste tipo de geração de forma eficiente. Esta se baseia em realizar a modelagem dos objetos através de funções matemáticas e algoritmos, que podem ou não ser baseados em leis físicas reais com relação aos fenômenos ou elementos que se deseja recriar.

A modelagem de elementos presentes na natureza normalmente apresenta um grande desafio a computação gráfica, dada a extrema complexidade que estes elementos podem possuir. A modelagem tradicional procura representar tais elementos através de um conjunto de polígonos e superfícies bem definidas e comportadas. Existem, porém, objetos em que esta abordagem não é suficiente, devido ao alto grau de complexidade ou ao grande número de sub-elementos que estes podem vir a ter. Um simples exemplo que demonstra esta complexidade é apresentado por [1] sendo relativo à modelagem de uma paisagem com uma floresta incluída, onde se supõe que existam 500 árvores. Considerando que em cada árvore existam cerca de 10.000 folhas onde cada folha necessita de aproximadamente 20 polígonos para sua representação, então, para cada árvore seriam necessários 200.000 polígonos apenas para folhas. Já para a floresta inteira seriam necessários 100.000.000 polígonos, apenas para as folhas das árvores que compõem a floresta.

Existem ainda casos em que uma modelagem realista através de superfícies não é possível, pois a complexidade do objeto não permite. Exemplos de tais fenômenos são

fogo, vapor d'água ou uma coluna de fumaça.

Dentre os vários elementos da natureza o terreno é um dos mais interessantes, pois sua representação pode ser obtida através de funções procedurais simples e apresentando resultados bem realistas. A computação gráfica tem uma longa história de tentativas de modelar terrenos do mundo real. Estas tentativas buscavam capturar os praticamente infinitos detalhes presentes em um terreno através de métodos de modelagem e/ou renderização.

O principal problema tratado nessa dissertação é a geração procedural e em tempo real de terrenos arbitrariamente detalhados através da extração de isosuperfícies.

1.1 Visão geral do problema

Uma das características mais importantes das técnicas procedurais é a abstração. Em uma abordagem procedural, ao invés de se explicitamente especificar e armazenar todos os detalhes de uma cena, estes são abstraídos em uma função ou um algoritmo e são avaliados apenas quando necessário. Desta forma, se tem a vantagem de não necessitar de espaço de armazenamento. Os detalhes não são mais especificados explicitamente, mas implícitos no procedimento e o requerimento de tempo para a especificação de detalhes é passado do programador para o computador.

Outro benefício deste tipo de abordagem é o controle paramétrico, onde se pode atribuir a um parâmetro um valor significativo (e.g., um parâmetro que define se uma montanha terá picos agudos ou se esta será mais suave). O controle paramétrico permite ao usuário ter um maior controle sobre as especificações de detalhes. Outra característica interesssante são os resultados inesperados que se podem obter de funções procedurais, resultados estes, geralmente bons, que ocorrem principalmente em procedimentos estocásticos.

A modelagem procedural permite criar de modelos e texturas de várias resoluções

e avaliar em tempo real qual resolução é necessária. No caso de um terreno, pode-se modificar como sua topografia se comporta em deteminadas áreas.

A vantagem de se criar um terreno proceduralmente é conseguir reproduzir detalhes afiados em qualquer nível desejado, ou seja, produzir um terreno arbitrariamente detalhado. Isto é bastante trabalhoso e custoso utilizando-se métodos tradicionais de modelagem, onde se tenta recriar tais detalhes através de polígonos e superfícies.

Modelos procedurais também oferecem flexibilidade. O desenvolvedor dos algoritmos pode capturar a essência do objeto, fenômeno ou movimento sem se preocupar com as restrições de leis físicas complexas, mas, caso necessário, as técnicas procedurais permitem incluir na função tais restrições com qualquer nível de exatidão.

O terreno, em um ambiente virtual, é um dos principais fatores que definirá o nível de imersão, ou seja, influenciará diretamente na experiência do usuário durante a interação com a aplicação. Isto é muito comum em jogos e simuladores, onde o ambiente pode influenciar nas decisões tomadas pelos jogadores (usuários). Muitos jogos são rotulados "bons"ou "ruins"dependendo da capacidade de criar um ambiente imersivo e interessante para o jogador.

Em um simulador o terreno tem um papel importante de representar o ambiente em determinada situação. Um exemplo simples é a modelagem de um poço de petróleo, onde o terreno representa todas as variações possíveis do processo de escavação. Este tipo de modelagem é extremamente útil, para profissionais na área de geologia, na detecção de problemas e variações durante o processo de escavação, que poderiam causar algum prejuizo ou apresentar algum perigo caso não fossem precebidos.

Para uma modelagem ainda mais realista é interessante se utilizar uma abordagem volumétrica, ou seja, utilizando-se extração de isosuperfícies. Este tipo de representação possibilita criar terrenos com diferentes características como túneis, cavernas, arcos. Um exemplo para este tipo de aplicação seria modelar o processo de perfuração de um terreno através de dados de sondas.

Uma representação volumétrica fornece uma informação importante, que é referente ao volume do modelo, possibilitando diferentes abordagens e manipulações para o terreno. Porém, este método é custoso e para se realizar manipulações em tempo-real é necessário utilizar um *hardware* poderoso, no caso a GPU (*Graphics Processing Unit*). A arquitetura paralela da GPU se encaixa muito bem neste problema, pois a geração de um terreno é um processo paralelizável, assim como os algoritmos de extração de superfícies.

Hoje a GPU está em nível de desenvolvimento maior que o das CPUs (*Central Processin Unit*) e com uma capacidade de processamento maior. Para se gerar terrenos procedurais com um alto nível de complexidade a taxas interativas de quadros por segundo é necessário recorrer a GPU, justificando a sua escolha neste trabalho. Outro ponto importante é a possibilidade de utilizar o *framework* CUDA (*Compute Unified Device Architecture*) criado pela NVIDIA, que possibilita a utilização da GPU com maior flexibilidade.

1.2 Contribuições

A principal contribuição deste trabalho é a utilização de dois métodos conhecidos na área de computação gráfica, mas pouco explorados. A geração procedural de terrenos e extração de isosuperfícies podem proporcionar resultados interessantes, possibilitando a criação de várias aplicações, as quais podem ser utilizadas em diferentes áreas.

Nesta dissertação esta abordagem é implementada utilizando-se uma nova abordagem para o algoritmo de *Marching Cubes* apresentada em [2]. Este foi estendido para a geração procedural de terrenos e para possibilitar a portabilidade entre os vários sistemas operacionais e placas gráficas. A parte de geração de terrenos utiliza as funções de ruído, primeiramente descritas por Ken Perlin [3], mas com uma implementação totalmente baseada em GPU utilizando OpenGL, visando a manipulação do terreno em tempo real.

1.3 Trabalhos Relevantes

No fim dos anos 60, o Dr. Benoit Mandelbrot relacionou formas naturais que mantêm um certo nível de auto-similaridade (formas que são exatamente ou aproximadamente similares a si mesmo ou parte de si) [4]. Em 1975, Mandelbrot cunhou o termo "'fractal"' para denotar um objeto que possua a sua dimensão Hausdorff-Besicovitch maior que a sua dimensão topológica. Uma definição formal de uma forma fractal, como definida por Mandelbrot, é: "'São aqueles que a dimensão Hausdorff não é inteira"' [5].

Mas foi em 1982 que a idéia de se modelar terrenos de forma procedural foi introduzida por Mandelbrot [6], que observou uma semelhança entre a curva gerada pela função fractal conhecida como *Fractal Brownian Motion - FBM*, com a silhueta de uma montanha.

Desde os primeiros trabalhos publicados por Mandelbrot várias técnicas de geração procedural foram criadas, mas a maior referência sobre o assunto é o livro chamado *Texturing and Modeling: A procedural approach* que foi escrito pelos maiores autores da área, sendo alguns deles os criadores de funções utilizadas até hoje como base da geração procedural. Estes autores são: David S. Ebert, Kenton F. Musgrave, Darwyn Peachey, Ken Perlin e Steven Worley. [7].

Neste livro há um tópico bastante relevante para este trabalho [7]. É abordada a criação de terrenos procedurais por funções fractais, tratando várias partes importantes como mapas de altura, a geração de terrenos homogêneos utilizando-se a função fBm, dimensão fractal e os efeitos visuais possíveis. Este capítulo também cobre a geração de modelos de terrenos heterogêneos, a partir de estatísticas das altitudes do terreno, considerando funções fractais híbridas e a criação de terrenos multifractais. Um outro detalhe interessante deste livro são as implementações apresentadas, como uma versão multifractal da função fBm.

O trabalho [1] também é uma boa referência, sendo apresentados vários conceitos básicos da área de geração procedural. Clua discute vários aspectos do "'por quê"' se uti-

lizar geração procedural quando se modela elementos da natureza. Este mostra as formas de como as funções fractais podem ser criadas a partir de funções básicas como o ruído de Perlin [3], além de apresentar toda teoria necessária ao entendimento da implementação de tais funções. Clua também apresenta formas de se criar terrenos proceduralmente a partir destas funções, produzindo resultados interessantes que foram de grande utilidade nesta dissertação.

No trabalho [8], terrenos são gerados utilizando diferentes métodos. O principal fenômeno que dá forma ao terreno é a erosão. Olsen considera dois tipos de erosão, a hidráulica e a térmica, sendo estas as mais comuns presentes na natureza. Para gerar o terreno, se utiliza o método conhecido como *deslocamento do ponto médio* usando o algoritmo *diamante-quadrado* [9] que simula a função de ruído. Porém, o problema de se utilizar apenas a função ruído para simular um terreno do mundo real é que esta gera um terreno estatisticamente homogêneo e isotrópico que são propriedades que um terreno no mundo real não possui. Assim Olsen utiliza diagramas de Voronoi, descritos em [7] por Steven Worley, combinados com a função de ruído para criar os mapas de alturas. Para eliminar as linhas presentes nos diagramas de Voronoi uma perturbação também descrita em [7] é realizada. Esta perturbação consiste em aplicar a função ruído um deslocamento nas células do diagrama com distâncias e direções aleatórias.

Em seu recente trabalho, Ryan Geiss [10] cria terrenos proceduralmente utilizando uma abordagem volumétrica com o algoritmo de *Marching Cubes* [11]. Neste trabalho, a função que gera os terrenos é a interação fractal da função ruído produzindo resultados bem interessantes. Vale ressaltar que neste trabalho toda a implementação é feita na GPU, utilizando-se vários recursos da API DirectX 10, como *shaders* de geometria, fluxos de saída (*stream output*) e renderização para texturas 3D.

Além da geração do terreno, Geiss realiza todos os métodos de texturização na GPU e ainda apresenta formas de se utilizar um sistema de partículas, totalmente implementado na GPU, para produzir quedas d'água e tornar o terreno mais realista. Este trabalho

ainda trata problemas de iluminação, mapeamento de rugosidade (*bump mapping*) e de deslocamento (*displacement mapping*).

O método *Marching Cubes* [11], criado em 1987 por Loransen e Cline, é eficiente para se extrair isosuperfícies a partir de campos escalares 3D. Este método é usado em todos os campos de visualização volumétrica e em diversas aplicações matemáticas que necessitam vários níveis de extração, como por exemplo métodos de dinâmica de fluidos. Provavelmente esse é o algoritmo mais usado para se produzir isosuperfícies, representado por triângulos, de campos escalares discretos. Este método será de grande importância nesta dissertação.

Depois de sua criação diversas variações do algoritmo de *Marching Cubes* surgiram, como o *Marching Tetrahedra*, o *Dual Marching Cubes* apresentado em [12]. Existem ainda variações que utilizam estruturas, como as *octrees*, para acelerar o processo de extração como em [13], [14] e [15]. Temos ainda variações da implementação original de Loransen em GPU como [16], [17] e [18].

Mais recentemente, ainda em abordagens relacionadas a GPU, temos os trabalhos de [19], [20] e [2], sendo este último de grande valor para esta dissertação.

Estas variações do algoritmo de *Marching Cubes* serão apresentados com maiores detalhes no capítulo 3.

1.4 Organização da Dissertação

Este trabalho está organizado como segue:

Capítulo 2 - Ruído Coerente e Geração Procedural de Terrenos

O capítulo 2 apresenta os fundamentos necessários para o entendimento da geração procedural de terrenos, discutindo as funções mais utilizadas e sua utilização neste tipo de geração.

Capítulo 3 - Extração de Isosuperfícies

O capítulo 3 descreve de forma geral a extração de isosuperfícies, onde os algoritmos mais utilizados são apresentados e discutidos.

Capítulo 4 - Geração Procedural de Terrenos por Extração de Isosuperfícies

O capítulo 4 apresenta como estes dois métodos podem ser combinados a fim de se obter uma geração de terrenos mais complexa e interessante.

Capítulo 5 - Resultados

O capítulo 5 apresenta os resultados obtidos com esta abordagem a nível de desempenho computacional e de qualidade visual.

Capítulo 6 - Conclusão e Trabalhos Futuros

Finalmente no Capítulo 6 são apresentadas as considerações finais e as propostas para trabalhos futuros.

Capítulo 2

Ruído Coerente e Geração Procedural de Terrenos

Neste capítulo são apresentadas diversas funções e suas aplicações na geração de terrenos.

2.1 Funções Procedurais

2.1.1 Funções Básicas

Existem várias funções procedurais capazes de produzir modelos e fenômenos que vemos na natureza. Porém, estas funções, em sua maioria, são compostas de um grupo de funções normalmente chamadas de funções básicas [1], as quais funcionam como base para funções mais complexas.

Ao se criar uma função procedimental devemos respeitar algumas propriedades, sendo estas inerentes às funções básicas:

- Pseudo-aleatoriedade: Os resultados desta função são aleatórios, mas para uma mesma entrada uma mesma saída deve ser apresentada;
- Não pode haver periodicidade de padrões;
- Estas funções devem ser estacionárias, invariantes a rotação, translação e serem isotrópicas.

Dentre as várias abordagens criadas na tentativa de se implementar funções básicas, a

mais famosa é a função de ruído criada por Ken Perlin [3]. Esta função é muito relevante para este trabalho, pois será utilizada na construção de funções mais complexas.

2.1.1.1 Função de Ruído

Esta função como o próprio nome sugere produz um ruído estocástico, ou seja, gera padrões aleatórios, obedecendo as propriedades inerentes às funções básicas.

A função de ruído é um mapeamento do R^n para o R, onde a entrada é um ponto com coordenadas reais de dimensão n e a função retorna um valor real. A função de ruído, quando observada como sinal, possui uma banda limitada onde quase toda sua energia é concentrada em uma pequena parte do espectro. As altas frequências (detalhes visualmente pequenos) e baixas frequências (formas mais alongadas) contribuem muito pouco para a energia total.

A definição do algoritmo da função de ruído não é complicada. Define-se uma grade como uma discretização do espaço R^n. Dado um ponto de entrada P, verificar os pontos vizinhos na grade. Em duas dimensões teremos 4 vizinhos, em três dimensões teremos 8 vizinhos e na n-ésima dimensão teremos 2^n vizinhos. Esta grade pode ser vista na Figura 2.1.

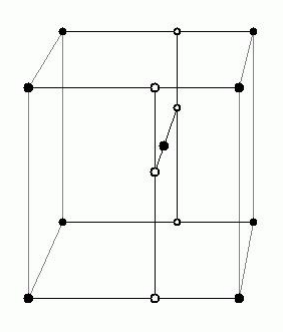

Figura 2.1: Grade no espaço R^3

Para cada ponto Q na grade, escolher um vetor gradiente pseudo-aleatório G. É importante ressaltar que para cada ponto particular na grade deve-se escolher o mesmo vetor gradiente. Assim computa-se o produto interno $G * P - Q$, isto resultará no valor do ponto P na função linear com o gradiente G, o qual é zero no ponto Q.

Assim tem-se 2^n valores. Interpolando estes pontos até o ponto escolhido, utilizando uma curva de transição suave (*cross-fade*) em forma de S, a curva utilizada por Perlin em seu primeiro trabalho [3] era da forma $3t^2-2t^3$, e é utilizada como peso para a interpolação em cada dimensão. Neste passo é necessário o cálculo de n curvas S e em seguida 2^n-1 interpolações. Uma representação gráfica destas interpolações, as quais foram retiradas do site de Ken Perlin [21], podem ser vistas na Figura 2.2.

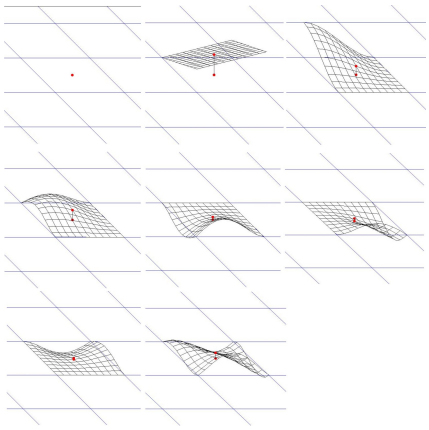

Figura 2.2: Interpolações com a função de suavização

Em três dimensões, temos oito pontos na grade, para se calcular suas respectivas influências é necessário uma interpolação trilinear (interpolação em cada uma das três dimensões). Na prática isto significa que temos que calcular a função de suavização $3t^2-2t^3$ em cada z, y e z e então realizar sete interpolações lineares para obter o resultado final. Na Figura 2.1 os pontos brancos representam os resultados das primeiras seis interpolações lineares, sendo elas quatro em x, duas em y e a sétima em z.

O próximo problema a ser resolvido é o cálculo dos gradientes citados anteriormente. Este cálculo deve ser realizado rápidamente, visto que isto será realizado um número de vezes considerável a cada iteração da função de ruído. Também é importante que não haja relação visível entre os valores sucessivos de gradientes nas vizinhanças da grade, para que não ocorram padrões visíveis. Estes gradientes são armazenados em uma tabela pré-computada. Deve-se então indexar cada [i,j,k] da grade a um valor n, onde $0 < n < N$ sendo N o tamanho da tabela de gradientes. Perlin em sua página [21] descreve esta indexação de *fold* e é da seguinte forma:

```
fold (i, j, k)
    n = Gradientes [i mod 256]
    n = Gradientes [(n + j) mod 256]
    n = Gradientes [(n + k) mod 256]
retorne n
```

Para criar a tabela de gradientes Perlin realiza uma simulação de Monte Carlo. O problema era calcular 256 vetores gradientes uniformente ao redor da superfície de uma esfera. Perlin sabia que podia computar pontos uniformemente dentro de um volume cúbico, selecionando uniformemente valores x, y e z aleatórios através de uma função do tipo *rand()*.

Para se realizar uma distribuição aleatória uniforme na superfície de uma esfera, é necessário apenas distribuir uniformemente pontos aleatórios dentro de um volume esférico, e então normalizar estes valores (redimensionar todos para um tamanho unitário). A simulação de Monte Carlo utilizada por Perlin era da seguinte forma: pontos aleatórios são gerados uniformemente dentro de um cubo que envolve a esfera unitária. Todos os pontos que se encontram fora da esfera são descartados e se armazenam todos os pontos que se encontram dentro desta esfera. Estes pontos são da forma: $x^2 + y^2 + z^2 < 1$.

O processo se encerra quando as 256 entradas da tabela de gradientes forem preechidas. Após isso todas as entradas da tabela são normalizadas e o processo está completo.

2.1.1.2 *Wavelet Noise*

A função original de ruído introduzida por Perlin ainda é a mais popular, pois é rápida, simples e muitos efeitos interessantes podem ser criados a partir dela. Porém, esta função possui alguns problemas com *aliasing* e perda de detalhes. Cook e Rose [22] apresentam uma nova função chamada *wavelet noise* que consegue evitar os problemas apresentados acima. Esta abordagem é baseada na idéia de que a função de Perlin a qual utiliza versões escaladas e atenuadas de uma função de banda limitada, o que também é conhecido como multiresolução. Após o trabalho de Perlin, a análise de *wavelets* ou análise de multiresolução, surgiu como uma poderosa forma de analisar e construir tais funções.

Na parte esquerda da Figura 2.3 podemos ter uma idéia dos problemas apresentados pela função de ruído criada por Perlin e na parte direita como estes problemas são resolvidos na *wavelet noise*. Ao fundo da figura na parte destacada temos um problema de *aliasing* bastante visível [22].

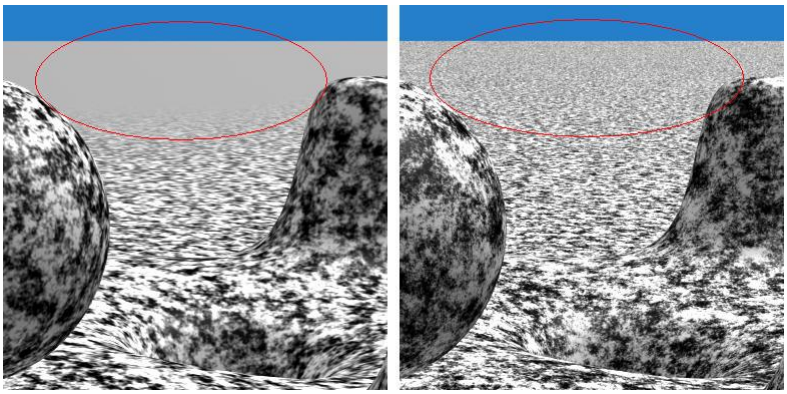

Figura 2.3: Comparação entre ruído de Perlin e *Wavelet noise*

2.1.2 Fractais

Além da função de ruído existem outras funções básicas (e.g., como a função baseada na distância celular descrita por Steve Worley [23]), mas a explicação destas tomaria algum tempo e fugiria ao escopo deste trabalho.

Utilizando apenas as funções básicas já seria possível criar terrenos proceduralmente. Porém, os terrenos na natureza possuem uma riqueza tão grande de detalhes que apenas estas funções não são suficientes para representar tantas caracteristicas. Isto é possível através de fractais [6].

Em teoria, para se criar um objeto realmente fractal, é necessária um repetição infinita de suas características com uma variação de escala. Na prática esta repetição não é necessária, pois a partir de certo momento as diferenças geradas pelas recursões serão imperceptíveis na mídia em que se está amostrando o resultado. Está repetição é importante na modelagem de objetos (terrrenos) ou na criação de texturas para este objeto. Desta forma podemos dar *zoom* infinitamente sem a preocupaçao da perda de detalhes.

As funções matemáticas a serem repetidas nas funções fractais são as funções básicas

apresentadas anteriormente, mais especificamente, neste trabalho, a função de ruído.

Uma primeira função apresentada por Perlin utiliza apenas uma soma fractal da função de ruído e é da forma:

$$noise(p) + (1/2)noise(2p) + (1/4)noise(4p) + ... \tag{2.1}$$

Esta soma fractal continua até que a soma do próximo termo seja muito pequena para se perceber. Isto resulta em um custo logarítimico maior do que a função de ruído [21], mas apresenta resultado melhores. Perlin [21] exemplifica esta função como na Figura 2.4 onde 8 iterações foram necessárias para criar tal padrão. Logo qualquer detalhe menor que $(0.5)^8$ apenas contribuiria com ruídos na imagem.

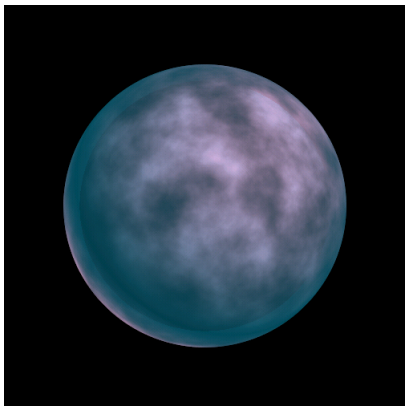

Figura 2.4: Soma fractal da função de ruído

Esta função é interessante para a geração de nuvens se utilizados os padrões de cores corretos, como pode ser visto na Figura 2.4 onde temos a impressão de um planeta basicamente azul coberto de nuvens.

2.1.2.1 Turbulência

Utilizando a mesma idéia, Perlin introduz a função conhecida como turbulência. Nesta função ao invés de utilizar a soma fractal da função de ruído, é realizada a soma fractal do valor absoluto da função de ruído. Ao se utilizar apenas o valor absoluto, Perlin adicionou uma descontinuidade à função. Em todos os lugares em que a função cruza o zero temos

uma variação alta no resultado. Ao se somar estas funções em várias escalas temos o resultado visual de cúspides (extremidades agudos), que são nos gradientes em todas as escalas e direções.

$$|noise(p)| + (1/2)|noise(2p)| + (1/4)|noise(4p)|... \tag{2.2}$$

Esta função possui este nome devido a sua aparência de fluxo turbulento e é boa para representar elementos como fogo como na Figura 2.5, quando colorida corretamente. Na geração de terrenos, esta função apresenta bons resultados visuais [21].

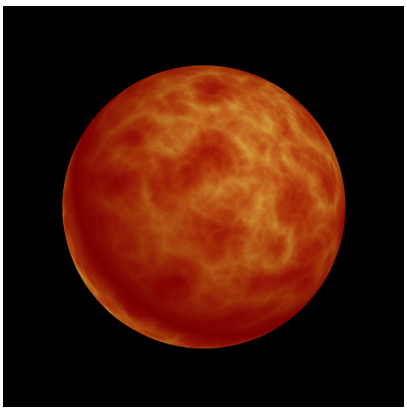

Figura 2.5: Turbulência

Uma outra variação das iterações fractais da função de ruído é criar um padrão de mámore. A função para este tipo de modificação é a seguinte:

$$sin(x + |noise(p)| + (1/2)|noise(2p)| + ...) \tag{2.3}$$

A Figura 2.6 foi criada utilizando-se a função de turbulência e utilizando-a para realizar uma mudança de fase em uma sequência de linhas. Esta sequência de linhas é criada pela função seno da coordenada x na superfície do modelo e a perturbação de sua fase cria uma distorção em forma de ondulação.

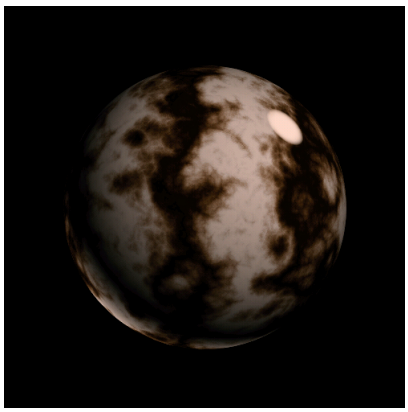

Figura 2.6: Perturbação utilizando a função de turbulência

2.1.2.2 *Fractal Brownian Motion - FBM*

A função *fractal Brownian Motion* é uma generalização da definição de fractais. A fractal Brownian motion normalizada $B^H(t)$ em $[0, T]$, $T \in R$ é um processo Gaussiano contínuo começando do valor zero e possui a seguinte função de correlação:

$$E[B^H(t)B^H(s)] = 1/2(|t|^{2H} + |s|^{2H} - |t-s|^{2H}) \tag{2.4}$$

Onde H, chamado de indice de *Hurst* ou parâmetro de *Hurst* associado com a fractal Brownian motion, é um número real em $[0, 1]$. O valor H determina o tipo de processo da fBm:

- Se $H = 1/2$, o processo é uma fBm regular
- Se $H > 1/2$, os incrementos do processo são positivamente correlacionados
- Se $H < 1/2$, os incrementos do processo são negativamente correlacionados

Podemos ver na Figura 2.7 um exemplo com diferentes valores de H [1].

Porém, para alguns tipos de modelos, a função fBm possui um inconveniente, que é o fato desta ser homogênea (possui a mesma distribuição em todo o espaço) e isotrópica (mesma distribuição em todas as direções). Para resolver tal problema Kenton Musgrave introduziu o conceito de multifractais. [24]

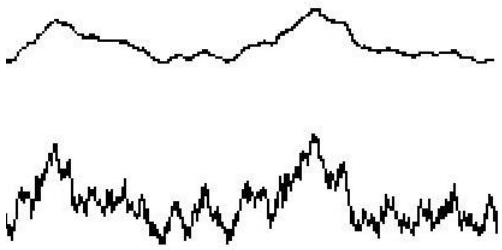

Figura 2.7: Fractal Brownian Motion com coeficiente de Hurst = 0.2 e 0.6, respectivamente

2.1.2.3 Multifractais

As funções fractais apresentadas anteriormente possuem o problema de serem estatisticamente homogêneas em todo o espaço. Isto ocorre, pois a cada iteração fractal o resultado encontrado é simplesmente somado ao valor anterior, [25] define este tipo de iteração fractal como sendo cascatas aditivas. As funções multifractais são construídas em cascata multiplicativa, ou seja, ao invés de se somar o resultado de cada iteração, realiza-se uma multiplicação.

Isto faz com que a distribuição estocástica deixe de ser homogênea, uma vez que o resultado final de uma iteração dependerá de todas as chamadas de uma função básica. Basta que em uma iteração, uma das chamadas retorne um valor pequeno (próximo a zero) para que todo o resultado da função seja pequeno. Os picos desta função acontecerão apenas nas situações em que todas as chamadas à função básica retornem valores altos. Por isso, costuma-se chamar este tipo de função de heterogênea.

2.2 Geração Procedural de Terrenos

A geração procedural de terrenos é uma área com grande potencial de desenvolvimento. Aplicações de computação gráfica hoje estão em constante desenvolvimento e o terreno é parte essencial em alguns destes, como jogos e simuladores. O nível de exigência dos usuários cresce cada vez mais em relação ao realismo e fidelidade ao que se encontra no mundo real. Porém, modelar terrenos utilizando métodos de modelagem tradicional pode

ser extremamente complicado e demandar uma quantidade de tempo razoável, dado o nível de detalhes que possam existir no terreno. Utilizando-se uma abordagem procedural este processo passa a ser mais simples e rápido, com resultados realistas e visualmente ricos.

Utilizando-se as funções apresentadas na seção anterior podemos criar vários tipos de terrenos. As funções básicas já seriam suficientes para a criação de terrenos, mas os resultados não representam o que temos na realidade, pois os resultados que estas funções apresentam geralmente são homogêneos e isotrópicos.

Na Figura 2.9 podemos ver como apenas a função de ruído de Perlin consegue representar um terreno, mas com poucas características interessantes e bastante homogeneidade.

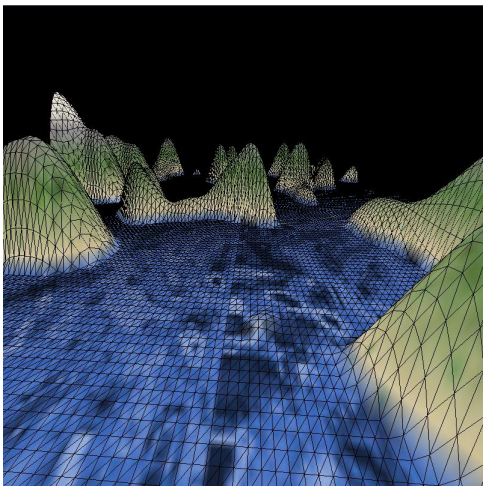

Figura 2.8: Terreno gerado com a função de ruído de Perlin

Porém, sua iteração fractal consegue representar muito melhor um terreno existente na natureza, apresentando zonas planas, montanhosas, vales e etc.

As funções apresentadas anteriormente podem ser utilizadas para se criar mapas de alturas em forma de texturas, onde a topografia do terreno será relativa à intensidade de cor de cada *pixel*. Por exemplo se utilizarmos a função de ruído para criarmos um mapa de alturas teremos o resultado que pode ser visto na Figura 2.10. Neste mesmo exemplo podemos ver que as zonas mais claras do mapa de alturas representam os valores de maior

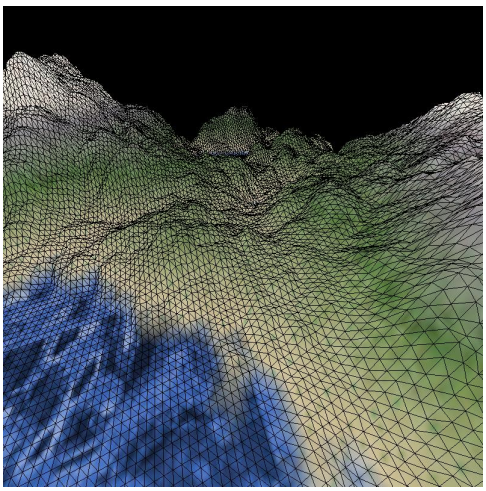

Figura 2.9: Terreno gerado com a função de turbulência

altura no terreno e as zonas mais escuras representam as depressões.

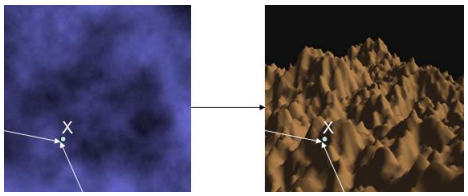

Figura 2.10: Relação entre mapa de altura e terreno

Os mapas de alturas podem ser pré-processados gerando um terreno fixo ou podem ser criados em tempo de execução na GPU. Esta segunda abordagem dá a liberdade de manipular o terreno livremente e pode ser implementada através de *shaders*. Esta é a abordagem utilizada nesta dissertação e será explicada em detalhes no capítulo 4.

Capítulo 3

Extração de Isosuperfícies

Define-se como isosuperfície a superfície que representa os pontos de valor constante dentro de um volume no espaço. Em outras palavras, é o nível de uma função contínua no espaço 3D.

3.1 *Marching Cubes*

Entre os métodos de extração de isosuperfícies, o algoritmo de *Marching Cubes* [11]é o mais conhecido. Este método é de grande importância para esta dissertação, e sua descrição é apresentada a seguir.

Seja uma grade discreta 3D de M_i x M_j x M_k valores escalares representando um campo escalar. Assim as células (*voxels*) são formados pelos 8 pontos pertencentes aos vértices $M_i - 1$ x $M_j - 1$ x $M_k - 1$. A idéia básica do algorítmo é "'marchar"' através de todas as células, uma por uma, e para cada célula produzir uma sequência de triângulos que aproxima a isosuperfície que está naquela célula particular.

Pode-se assumir que a geometria da isosuperfície do voxel pode ser completamente determinada pela classificação dos 8 vértices da célula em "'dentro"' ou "'fora"' da isosuperfície, como apresentado na Figura 3.1. Sendo $S_n^{ijk} = 1$ se o vértice n do voxel (i, j, k) estiver dentro da isosuperfície e 0 caso contrário, podemos determinar a classe do *voxel*, através da fórmula:

$$c^{ijk} = s_0^{ijk} + s_1^{ijk} + s_2^{ijk} + s_3^{ijk} + s_4^{ijk} + s_5^{ijk} + s_6^{ijk} + s_7^{ijk} \tag{3.1}$$

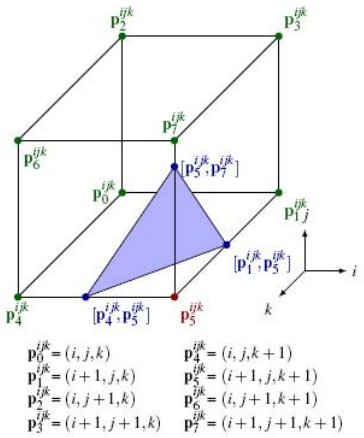

Figura 3.1: Nomeclatura dos vértices da célula de *Marching Cubes*

Por exemplo se o voxel da Figura 3.1 possuir o vértice P^{ijk} dentro da isosuperfície e todos os outros vértices fora, como o exemplo a abaixo,

$$(s_0, ..., s_7) = (0, 0, 0, 1, 0, 0, 0) \tag{3.2}$$

correspondendo a classe $c^{ijk} = 16$. Existem no total 256 classes, que podem ser reduzidos a 14 padrões [**?**] devido à simetria. Estes podem ser vistos na Figura 3.2.

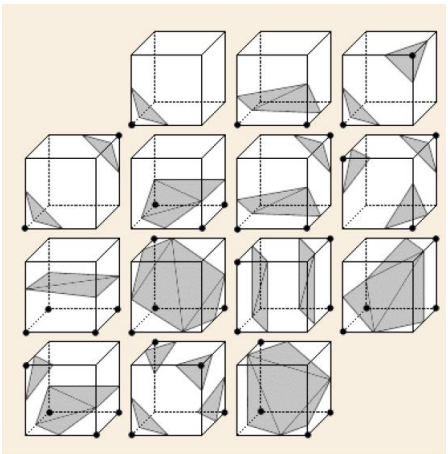

Figura 3.2: Padrões existentes no algoritmo de *Marching Cubes*

A classe do *voxel* também determina quais das doze arestas "perfuram"a isosuperfície. Uma interseção ocorre quando um dos vértices está dentro da isosuperfície enquanto

os outros estão fora. Por exemplo, no voxel da Figura 3.1, o vértice p_5^{ijk} está dentro da isosuperfície enquanto os outros vértices estão fora, logo as arestas [p_1^{ijk}, p_5^{ijk}], [p_4^{ijk}, p_5^{ijk}] e [p_5^{ijk}, p_7^{ijk}] perfuram a isosuperfície. Para cada uma das 256 classes existe uma triangulação correspondente para as interseções com as arestas. Assim quando a classe do *voxel* é determinada pode-se simplesmente acessar uma tabela de triangulação e designar as posições dos vértices para as posições onde as arestas perfuram a isosuperfície.

O próximo passo é determinar as interseções das arestas. Utilizando campos escalares binários, o que é comum em imagens médicas, pode-se assumir que a interseções estão no meio da aresta. Porém, esta abordagem resulta em uma superfície de baixa qualidade. Utilizando um campo escalar contínuo é possível aproximar este campo ao longo da aresta com uma equação linear polinomial e assim encontrar o ponto de interseção, o que resulta em uma superfície muito mais suave.

Como dito anteriormente, o algoritmo de *Marching Cubes* é extremamente conhecido e utilizado. Desde sua criação em 1987, várias modificações e variações foram implementadas. Uma destas é conhecida como *Marching Tetrahedra* onde cada célula (cubo) é dividido em 6 tetraedros 3.3 cortando-se diagonalmente através dos três pares de faces opostas. Assim eles compartilham umas das diagonais principais do cubo. Ao invés de termos apenas doze arestas do cubo, temos dezenove, sendo as doze originais mais as seis diagonais de cada face 3.4 e a diagonal principal. Assim como no *Marching Cubes* as interseções destas arestas com a isosuperfície são calculadas pela interpolação linear dos pontos da grade. Cubos adjacentes dividem todas as arestas nas faces em que se conectam, o que previne defeitos na superfície que será extraída.

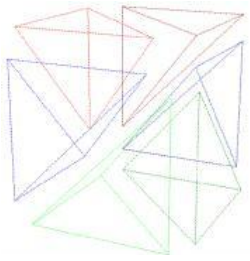

Figura 3.3: Subdivisões do algoritmo de *Marching Tetrahedra*

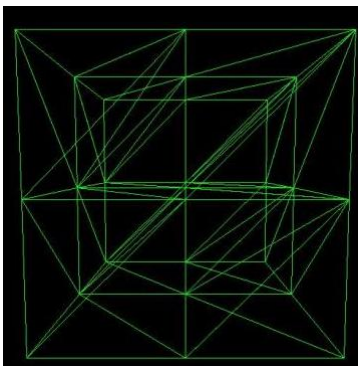

Figura 3.4: Grade do *Marching Tetrahedra*

Outra variação é o *Dual Marching Cubes* [12] que apresenta um método para se contornar uma função implícita que utiliza uma grade topológicamente dual para grades estruturadas como *octrees*. Atinge-se um bom nível de detalhamento sem a necessidade de um grande número de sub-divisões, produzindo uma superfície sem rachaduras e com uma poligonização adaptativa que reproduz detalhes apurados.

Além destes existem outros métodos que utilizam *octrees* para se otimizar o processo de "'marchar"' pelo volume como em [13], [14] e [15]. Outra exelente referência sobre o assunto é o *survey* escrito por Aaron Knoll [26] onde são apresentadas e comparadas várias abordagens que utilizam *octrees* em métodos de renderização de volumes com objetivos de compressão, simplificação, extração e renderização.

Com o surgimento das GPUs a maioria destes métodos foi adaptada ao paradigma de programação em GPUs. A principio, o algoritimo de *Marching Tetrahedra* se comportava melhor, pois não era possível se realizar operações por triângulo na GPU e a versão deste algoritmo, a qual utiliza método de se ignorar faces ocultas, produzia apenas 0, 1 ou 2 triângulos para cada tetraedro examinado. Porém, para uma grade cúbica, cada cubo deve ser divido em pelo menos 5 tetraedros, o que gera um total de triângulos geralmente maior do que uma malha criada pelo algoritmo de *Marching Cubes* tradicional. Um primeiro exemplo de trabalhos utilizando *Marching Tetrahedra* é de Pascucci et al. [27] que representa cada tetraedo com um quadrado e utiliza o *vertex shader* para realizar os cálculos de interseção. A geometria de entrada é expressa na forma de *strips* de triângulos

organizadas em uma curva que preenche o espaço, o que diminui a carga de trabalho do *vertex shader*.

A abordagem de *Marching Tetrahedra* de [16] renderiza a geometria em vetores de vértices e passa toda a computação para os *fragment shaders* atingindo picos de 7,7 milhões de tetraedros por segundo, utilizando uma ATI Radeon 9800 Pro.

Uma melhora deste método foi sugerido em [17] que observou que as arestas eram compartilhadas pelos cubos adjacentes e isto deveria ser utilizado como estrutura de dados básica na avaliação das interseções.

Buatois [18] aplicou múltiplos passos de *shaders* e *lookups* em texturas para reduzir cálculos redundantes.

Algumas abordagens optaram por utilizar a CPU para classificar as células de *Marching Cubes* e apenas enviar à GPU as células relevantes. Esta abordagem é utilizada em [28] e abordada também em [29] que utilizam uma estrutura de *kd-tree* para eliminar as regiões vazias. Vale ressaltar que estes autores também notaram que este pré-processamento pela CPU limita a velocidade do algoritmo.

Uma solução é a utilização da etapa de *Geometry Shaders*, presentes em *hardwares Shader Model 4.0*, ou seja placas gráficas que dêem suporte a uma série de funcionalidades, entre elas a utilização de *shaders* de geometria. Isto possibilita criar e descartar geometria em tempo real. Uralsky [30] apresenta uma abordagem do algoritmo de *Marching Tetrahedra* utilizando *geometry shaders* para *grids* cúbicos, dividindo cada célula em 6 tetraedros. Esta implementação é disponibilizada no Nvidia OpenGL SDK-10.

A maioria deste métodos consegue prover uma cópia da isosuperfície para a memória da GPU, utilizando *buffers* de vértices ou o novo mecanismo chamado *transform feedback* presente em *hardwares Shader Model 4.0*. Porém, excluindo as abordagem que utilizam *geometry shaders*, a cópia para a memória de vídeo pode degenerar a geometria. Assim, técnicas de pós-processamento são necessárias (e.g.*stream compaction*) para produzir uma sequência "compacta"de triângulos.

Harris [19] apresenta uma implementação eficiente do algoritmo de *Prefix Sum (Scan)*, o qual utiliza um esquema na forma de pirâmide, onde é criado, em paralelo, uma tabela que associa cada elemento de entrada com os deslocamentos dos elementos de saída. Assim, usando *scattering*, a GPU consegue processar os elementos de entrada e diretamente alocar a saída utilizando esta tabela de deslocamentos. A NVIDIA disponibiliza uma implementação do algoritmo de *Marching Cubes* utilizando *Scan*, no NVIDIA CUDA SDK 2.0.

Ziegler [20] apresentou uma outra abordagem para a compactação de dados. Este introduziu o algoritmo de *Histogram Pyramids* para a compactação de dados na GPU em *hardwares Shader Model 3.0*.

Christopher Dyken e Gernot Ziegler [2] estenderam este método, possibilitando a expansão de fluxo de dados (*stream expansion*) na GPU. Este trabalho é baseado na interpretação do algoritmo de *Marching Cubes* como um processo de compactação e expansão de dados e foi implementado utilizando o método introduzido por Ziegler (*Histogram Pyramids*). Este trabalho provê uma forma eficiente de se gerar geometria diretamente na GPU e hoje é o método mais rápido entre os métodos baseados em GPU.

3.2 *Histogram Pyramids*

Histogram Pyramids é um estrutura de dados hierárquica que foi recentemente introduzida na programação em GPU [20]. A natureza local dos algoritmos associados a esta estrutura possibilita a expansão e compactação de dados, uma tarefa tradicionalmente dificil em processadores de fluxo [2]. A seguir será apresentado como esta estrutura de dados funciona e como é construída.

A construção da pirâmide ocorre da base para o topo. Assim, todos os elementos de entrada estarão na base da pirâmide. Na Figura 3.5 vemos os elementos de entrada no *base level* (na base da pirâmide). Estes elementos definem o número de elementos alocados na saída de dados [2].

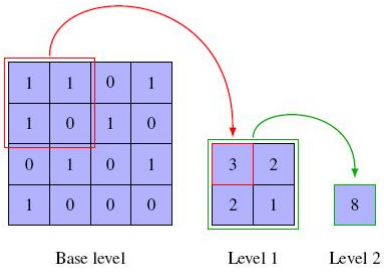

Figura 3.5: Compactação dos dados.

Na entrada, os dados dos *voxels* são mapeados do domínio 3D para o 2D através de texturas, assim pode-se indexar a sequência utilizando-se coordenadas de textura. Na base, cada dado de entrada corresponde a um *texel* (um *texel* ou elemento de textura, é a unidade fundamental do espaço de textura), onde este *texel* tem o número de elementos a serem alocados na saída. Por exemplo, na Figura 3.5 temos os elementos **ordenados da esquerda para direita e de cima para baixo**. Temos que os elementos 0, 1, 3, 4, 6, 11 e 12 alocaram cada um apenas um elemento na saída (*stream pass-through* - fluxo direto). O elemento número 9 alocou dois elementos na saída (*stream expansion* - expansão de dados) e os outros elementos não alocaram nada (*stream compaction* - compactação de dados) e serão descartados.

Na construção da pirâmide cada nível é construído sendo um *texel* deste nível possuir a soma dos quatro *texels* do nível abaixo, da mesma forma que se constroi um *Mipmap*, mas neste caso os elementos são somados ao invés de se obter a média deles. Assim o nível de cima da pirâmide contém o número de elementos a serem alocados na sequência de saída.

Para se extrair os elementos é necessário percorrer a pirâmide. Isto está representado na Figura 3.6, [2].

Sendo a chave o índice do elemento na sequência de saída, na Figura 3.6 queremos o quinto elemento da sequência de saída, ou seja, o elemento com índice chave igual a 4. Comparamos a chave com o elemento do topo da pirâmide, como este é menos que 8 sabemos que este elemento é parte da sequência de saída. Assim descemos um nível

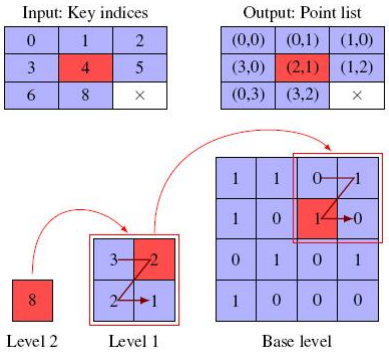

Figura 3.6: Extração da sequência de saída

na pirâmide, onde temos quatro intervalos de elementos. O primeiro é de zero até três exclusivo, o segundo de três até cinco, o terceiro de cinco até sete e o último de sete a oito.

$$A = [0, 3), B = [3, 5), C = [5, 7), D = [7, 8) \tag{3.3}$$

Assim, o sub-intervalo em que a chave se encontra define em qual das quatro sub-pirâmides devemos percorrer. Neste caso, a chave se encontra na segunda sub-pirâmide de intervalo três a cinco. Devemos, também, ajustar a chave a cada descida de acordo com o começo do intervalo em que esta se encontra. Nesta sub-pirâmide o intervalo começa em três assim a chave deve ser subtraída deste índice neste caso $chave = 4 - 3$. Descendo novamente até a base da pirâmide temos os quatro *texels* que formam os seguintes intervalos:

$$A = [0, 0), B = [0, 1), C = [1, 2), D = [2, 2) \tag{3.4}$$

Temos que esta nova chave se encontra no intervalo de um até dois (C), assim ajustamos a chave, sendo desta vez $chave = 1 - 1$, e atualizamos a lista de pontos (2, 1). O resto da chave (neste caso igual a zero) contém o deslocamento na sequência de elementos de saída alocados por este elemento particular do nível base da pirâmide.

O interessante deste processo é que não há dependência entre cada consulta à estrutura. Assim, os elementos de saída podem ser extraídos em qualquer ordem e até mesmo em paralelo.

O esquema de texturas 2D do *Histogram Pyramids* se encaixa bem no *hardware* gráfico. Pode-se perceber que em um domínio de calculo de coordenadas de texturas normalizadas, as buscas nas texturas se sobrepõem às buscas no nível abaixo. Isso possibilita o uso de caches de texturas 2D, para auxiliar a percorrer a estrutura, melhorando sua performance.

A cada descida durante a pesquisa na estrutura, deve-se pesquisar os valores de quatro *texels*, o que resulta em quatro buscas em texturas. Porém, como sempre se busca em blocos 2 x 2, pode-se utilizar uma textura com quatro canais de cores e codificar estes valores como valores RGBA. Ao se fazer isto, as dimensões de todas as texturas em ambas as direções são divididas por dois, possibilitando criar texturas para a estrutura de *Histogram Pyramids* quatro vezes maior dentro dos mesmos limites de tamanho de texturas. Como o *hardware* gráfico é eficiente em realizar buscas em valores RGBA, esta abordagem geralmente gera algum ganho de desempenho.

3.3 *Marching Cubes* utilizando *Histogram Pyramids*

Como apresentado em [2] a implementação de *Marching Cubes* é vista como uma sequência de operações de fluxo de dados, sendo o fluxo de dados de entrada o campo escalar 3D e o fluxo de dados de saída um conjunto de vértices, que formam os triângulos da iso-superfície. As operações de fluxo de dados são executadas no algorítmo de *Histogram Pyramids*, ou em uma variação, no *shader* de geometria, o qual compacta e expande o fluxo de dados de acordo com a necessidade.

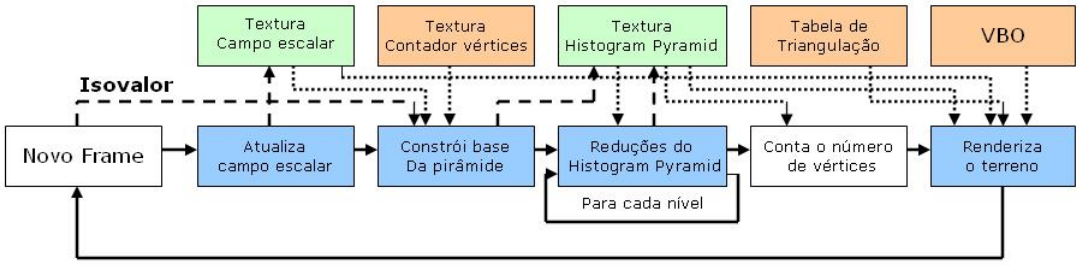

Figura 3.7: Fluxograma da abordagem de *Marching Cubes* com *Histogram Pyramids*

Na Figura 3.7 as setas contínuas significam o fluxo de controle, sendo as etapas azuis realizadas na GPU e as brancas na CPU. As setas pontilhadas e tracejadas representam o fluxo de dados, sendo as setas pontilhadas operações de leitura e as tracejadas de escrita. As caixas verdes representam os dados dinâmicos. As vermelhas para dados estáticos.

Este algoritmo utiliza quatro texturas. A primeira representa o número de vértices, a segunda a tabela de triangulação, que são texturas pequenas pré-computadas. A terceira textura representa a função que representará o campo escalar e a quarta é referente ao *Histogram Pyramids*. A textura que representa o número de vértices é uma tabela 1D contendo o número de vértices necessário para se triangular cada uma das 256 classes de *voxels*. A textura que representa a tabela de triangulação contém uma tabela 16 por 256, onde uma entrada ij diz qual classe de *voxels* j o vértice i da aresta que intercepta a isosuperfície pertence. A textura que representa a função, obviamente, contém o campo escalar do qual se extrairá a isosuperfície. O campo escalar é armazenado na forma de uma textura 2D grande subdividida em *tiles*, onde cada subdivisão, (*tile*), corresponde a uma fatia do espaço 3D. Esta abordagem é conhecida como *Flat3D layout* [31]. *Histogram Pyramids* é representado em uma textura RGBA com quatro canais, contendo o número de vértices no fluxo de entrada necessários para triangular cada uma dos *voxels*.

O algoritmo contém quatro etapas:

- O primeiro estágio é popular a textura da função. Onde os dados podem vir por exemplo dos cálculos de uma passada GPGPU.

- Segundo estágio é construir a base da pirâmide do *Histogram Pyramids*. Uma textura RGBA de quatro canais, assim cada *texel* corresponde um pequeno grupo 2 x 2 x 1 de *voxel* que são vizinhos e dividem os vértices. Acessam-se os valores da textura da função que correspondem aos vértices dos *voxels*, e os comparam ao *isovalue* para se determinar se os vértices estão dentro ou fora da isosuperfície. Com os resultados pode-se determinar a classe destes *voxels* e, com a classe, pode-se consultar a tabela que contém o número de vértices de cada classe e, assim, determinar o número de

vértices que os *voxels* precisam no fluxo de saída. Para *texels* correspondendo a *voxels* fora do volume, os valores de número de triângulos e classe são definidos como zero. O número de vértices é necessário para se construir o *Histogram Pyramids* e a classe de *voxels* é necessária no estágio de extração.

- O terceiro estágio é construir o resto do *Histogram Pyramids* utilizando reduções consecutivas. Cada redução é realizada em um passo GPGPU. A saída é a soma dos quatro *texels* diretamente inferiores ao *texel* atual na pirâmide de *MipMap*. Cada passo é realizado em um *render-to-texture loop* com geração de *MipMap*, e cada nível da pirâmide de *MipMap* é armazenado separadamente em um *framebuffer object* (FBO).

- O quarto estágio é a extração. Utilizando o *Histogram Pyramids* é possível extrair todos os vértices que formam os triângulos da isosuperfície.

O primeiro passo é ler o único *texel* do topo da estrutura do *Histogram Pyramids* para a memória. A soma dos quatro destes *texels* (RGBA) é o número de vértices (N_v) que formam a isosuperfície [2], e este número é três vezes o número de triângulos.

Depois começam-se a renderizar os triângulos e o processo dos (N_v) vértices. Os vértices precisam ser numerados com indices, e o único atributo de entrada para o *vertex shader* é este índice. Os vértices são acionados renderizando-se um VBO (*Vertex Buffer Object*), onde a coordenada x de cada vértice é o índice. Mesmo utilizando-se placas que possuam *Shader Model 4.0* onde se pode utilizar o atributo $gl_VertexID$, OpenGL não pode iniciar o processamento dos vértices sem algum dado de atributo, assim é necessário utilizar um VBO de qualquer maneira. Para cada vértice o *vertex shader* utiliza o índice (este índice será utilizado como chave, como visto na seção 3.0.2) para percorrer a estrutura do *Histogram Pyramids* e determinar a qual célula e a qual aresta este vértice pertence. Com a classe do *voxel* e o resto da chave pode-se consultar a tabela de triângulação para se determinar qual aresta do *voxel* utilizar.

Utilizando a posição do vértice pode-se determinar onde a aresta intercepta a isosuperfície. Isto é feito aproximando o campo escalar com uma função polinomial linear ao longo da aresta. O valor zero desta função aponta o lugar onde ocorreu a interseção, o qual é utilizado como posição para o vértice. O vetor normal pode ser calculado através do gradiente do campo escalar.

Capítulo 4

Geração Procedural de Terrenos por Extração de Isosuperfícies

A geração procedural possibilita a criação de terrenos interessantes e com um bom grau de customização. Porém, podem-se criar terrenos mais complexos utilizando-se uma abordagem volumétrica. A principal contribuição deste trabalho é a união destes dois métodos, a geração procedural de terrenos e uma visualização volumétrica utilizando *Marching Cubes*, ambos na GPU.

Neste capítulo será apresentado como criar a função procedural na GPU, neste caso a função de ruído de Perlin, e como esta função pode ser utilizada na criação de terrenos. Em seguida será apresentada a forma como isto pode ser implementado em uma abordagem volumétrica.

4.1 Introdução

A geração de terrenos se encaixa bem no paradigma de programação em GPU, devido de sua natureza paralela. Os vértices em um terreno podem ser processados independentemente, o que acelera bastante o processo e facilita sua implementação utilizando *shaders*. Este tipo de abordagem também possibilita a geração e manipulação do terreno em tempo real, sendo esta é uma das principais vantagens da geração procedural. O controle paramétrico do modelo o que possibilita a manipulação do terreno, modificando sua topografia inserindo algum parâmetro diretamente na função escolhida.

Utilizando-se uma abordagem volumétrica ampliamos o leque de opções para a geração de terrenos. Podem-se conseguir diferentes formas com esta abordagem, como túneis, pontes naturais (ligam duas partes do terreno) e cavernas. Temos ainda a chance de sermos surpreendidos pelas funções utilizadas e obter resultados inesperados, pois estas funções são pseudo-aleatórias.

Neste tipo de abordagem a função procedural que define o terreno representa um campo escalar de onde será extraído o terreno. Utilizando algoritmos como o *Marching Cubes* pode-se extrair toda a superfície do terreno dado um isovalor. Nesta dissertação o método de extração escolhido foi a implementação do algoritmo de *Marching Cubes* presente em [2], que realiza todo o processo na GPU. Maiores explicações sobre este método são apresentadas no capítulo 3, mais especificamente na seção 3.1.

4.2 Funções de Ruído na GPU

A função escolhida para gerar os terrenos foi a iteração fractal da função de ruído de Perlin conhecida como turbulência . Sua implementação é realizada totalmente na GPU [32]. Nesta abordagem a função de ruído é implementada totalmente em *Pixel Shaders*. Atualmente a função de ruído procedural é implementada na GPU a partir de texturas 3D pré-computadas, porém a abordagem utilizando somente *shaders* possui algumas vantagens:

- O método utilizando *Pixel Shaders* necessita de menos memória para armazenar texturas.
- O período é maior, ou seja, não se repetem os padrões facilmente.
- Os resultados são os mesmos apresentados pela versão implementada na CPU.
- Possibilita a criação de ruído 4D, o que é util para efeitos 3D com animação.
- A interpolação possui uma qualidade maior.

A função de ruído utiliza duas tabelas em sua implementação, uma é a tabela de permutação e a outra uma tabela de vetores gradientes 3D. A tabela de permutação contém 256 valores variando de 0 a 255 em uma ordem aleatória e pode ser escrita da seguinte forma:

```
static int permutation[ ] = { 151,160,137,91,90,15,131,13,201,95,96,53,
194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,190,6,148,247,120,
234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,88,237,149,56,87,
174,20,125,136,171,168,68,175,74,165,71,134,139,48,27,166,77,146,158,231,
83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,102,143,54,
65,25,63,161,1,216,80,73,209,76,132,187,208,89,18,169,200,196,152,135,
130,116,188,159,86,164,100,109,198,173,186,3,64,52,217,226,250,124,123,
5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,
28,42,223,183,170,213,119,248,2,44,154,163,70,221,153,101,155,167,43,
172,9,129,22,39,253,19,98,108,110,79,113,224,232,178,185,112,104,218,246,
97,228,251,34,242,193,238,210,144,12,191,179,162,241,81,51,145,235,249,
14,239,107,49,192,214,31,181,199,106,157,184,84,204,176,115,121,50,45,127,
4,150,254,138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,
156,180};
```

Algoritmo 1: Tabela de permutação

O vetor de gradientes é da seguinte forma:

```
static float3 g[ ] = {
1,1,0, -1,1,0, 1,-1,0, -1,-1,0,
1,0,1, -1,0,1, 1,0,-1, -1,0,-1,
0,1,1, 0,-1,1, 0,1,-1, 0,-1,-1,
1,1,0, 0,-1,1, -1,1,0, 0,-1,-1,
};
```

Algoritmo 2: Tabela de vetores gradientes

As tabelas consultadas durante o algoritmo são criadas a priori. A tabela representada no 2, por exemplo, pode ser armazenada em uma textura 1D onde os valores são armazenados nos canais RGB, um exemplo desta textura é apresentada na Figura 4.1:

Figura 4.1: Tabela de vetores gradientes na forma de textura

A implementação da função de ruído na GPU em *Pixel Shaders* é realizada na linguagem *GLSL*(*OpenGL Shader Language*) e seu código pode ser visto abaixo: A entrada

```
float noise(vec3 p)
{
   //Cubo unitário que contém o ponto
   vec3 P = mod(floor(p), 256.0);

   //x,y,z relativo do ponto no cubo
   p -= floor(p);

   //cálculo da função para cada x,y,z
   vec3 f = fade(p);

   P = P / 256.0;
   const float one = 1.0 / 256.0;

   // coordenadas hash dos 8 cantos do cubo
   vec4 AA = perm2d(P.xy) + P.z;

   //adicionar interpolando os 8 valores
   return mix( mix(
               mix( gradperm(AA.x, p ),
                    gradperm(AA.z, p + vec3(-1, 0, 0) ),
                    f.x),
                mix( gradperm(AA.y, p + vec3(0, -1, 0) ),
                gradperm(AA.w, p + vec3(-1, -1, 0) ), f.x),
                               f.y),

                mix(mix(gradperm(AA.x+one, p + vec3(0, 0, -1)),
                        gradperm(AA.z+one, p + vec3(-1, 0, -1)),
                                           f.x),
                    mix(gradperm(AA.y+one, p + vec3(0, -1, -1)),
                        gradperm(AA.w+one, p + vec3(-1, -1, -1)), f.x),f.y),f.z);
}
```

Algoritmo 3: Função de ruído de Perlin em GLSL

da função é o ponto p.Podemos ver o cálculo da contribuição dos 8 cantos do cubo unitário, utilizando as texturas pré-computadas. A função *hash*, utilizada para se acessar os valores da tabela de permutação, primeiro indexa a tabela utilizando a coordenada x do ponto p. Em seguida, o valor da coordenada y é adicionado ao valor retornado pela tabela e este valor é utilizado para indexar novamente a tabela e o processo é repetido para a coordenada z. Assim temos um valor pseudo-aleatório para cada valor no espaço.

Em um segundo passo, este valor é utilizado para indexar a tabela de vetores gradientes 3D. Sendo esta indexação realizada através da função *fold* descrita no capítulo 2.

A função *fade* é a função de interpolação e nesta implementação foi utilizada a nova função apresentada por Perlin [33]. Esta função pode ser vista a seguir:

$$t * t * t * (t * (t * 6.0 - 15.0) + 10.0) \tag{4.1}$$

Esta curva é uma polinomial Hermite de quinto grau, que possui derivadas contínuas até segunda ordem, produzindo um ruído de melhor qualidade. Pode-se utilizar a função apresentada no capítulo 2, a qual é computacionalmente mais leve de se avaliar, porém aquela resulta em descontinuidades em suas derivadas segundas.

Podemos então definir a função de turbulência no mesmo *shader* com o seguinte algoritmo em *GLSL*:

```
float turbulence(vec3 p, int octaves, float lacunarity, float gain)
{
   float sum = 0.0;
   float freq = 1.0, amp = 1.0;
   for(int i=0; i<octaves; i++) {
      sum += abs(onoise(p*freq))*amp;
      freq *= lacunarity+lacunarityN;
      amp *= gain+gainN;
   }
   return sum;
}
```

Algoritmo 4: Função de turbulência

Neste trabalho a variável lacunaridade e ganho (*gain*) são modificadas dinamicamente, ou seja, é possível alterar esses valores durante a execução, facilitando a customização e a realização de testes no terreno. Estes valores serão alterados por eventos de teclado. Os valores definidos inicialmente para estas duas variáveis é de 0,5 no caso do ganho e 2,0 para lacunaridade onde a amplitude é dividida pela metade a cada oitava e a frequência dobra.

4.3 Extração do Terreno

Concluída a função que irá definir o terreno deve-se agora extrair a superfície. Neste caso iremos utilizar o algoritmo de *Marching Cubes*, mais especificamente a abordagem utilizada em [2]. Este trabalho foi escolhido por apresentar um desempenho melhor que as outras implementações em GPU e por possuir maior flexibilidade. Esta abordagem ainda possui a vantagem de ter sido originalmente desenvolvida para o ambiente UNIX. Nesta dissertação esta abordagem foi portada para o ambiente Windows, mas mantendo a compatibilidade com os sistemas UNIX.

Apesar da área de geração de terrenos ser uma área bastante pesquisada, a sua implementação utilizando uma abordagem volumétrica não é muito explorada, contendo poucos trabalhos nesta área. O único trabalho relevante encontrado durante as pesquisas para este trabalho é o de Ryan Geiss [10], onde uma versão do algoritmo *Marching Cubes* é implementado em GPU. A abordagem de Geiss possui algumas desavantagens, como a falta de portabilidade, pois esta é implementada utilizando-se a API DirectX 10, e por isso necessita do sistema operacional Windows Vista como requerimento, além de ser dependente de *Geometry Shaders*.

Utilizando a abordagem de [2] podemos gerar o campo escalar em tempo de execução. Para isso, é necessário utilizar os *Shaders* desenvolvidos na seção anterior para preencher a textura que representa o campo escalar do terreno (como apresentado no Capítulo 3, mais especificamente na seção 3.1). Para isso, uma função para a geração do terreno é gerada, a qual irá atualizar a função de ruído durante a renderização. Foi utilizado um FBO para armazenar a saída do *pixel shader*, é possível então realizar os cálculos do tamanho da base do *Histogram Pyramids* [34]. O código para esta operação é apresentado no algoritmo 5.

Este algoritmo começa testando se o que está sendo gerado é o terreno. Primeiro compara-se o tipo de *update* do campo escalar, pois nesta aplicação podem-se ter vários tipos de entrada, como por exemplo ter um modelo escaneado ou uma outra função qualquer. Assim, escolhido o tipo de atualização do campo escalar, é necessário definir alguns valores utilizados pelo *shader*. No caso da *escalaN* tem-se uma escala no terreno onde é possível variar a influência do ponto na função de turbulência ou ruído. As variáves de lacunaridade e ganho, *lacunarityN* e *gainN*, respectivamente, permitem a manipulação, em tempo de execução, da amplitude e da frequência da função de turbulência. Este tipo de manipulação permite testar várias topografias diferentes para o terreno e também possibilita observar melhor os resultados apresentados pelas funções escolhidas.

As duas texturas seguintes nomeadas *permTexture2d* e *permGradTexture* são as tabelas

```
location_escalaN = glGetUniformLocation(m_field_update_p, "escalaN");
glUniform1f(location_escalaN, m_config.m_escalaN);

location_lacunarity = glGetUniformLocation(m_field_update_p, "lacunarityN");
glUniform1f(location_lacunarity, m_config.m_lacunarity);

location_gain = glGetUniformLocation(m_field_update_p, "gainN");
glUniform1f(location_gain, m_config.m_gain);


location_permTexture2d = glGetUniformLocation(m_field_update_p,
                                              "permSampler2d");

location_permGradTexture = glGetUniformLocation(m_field_update_p,
                                                "permGradSampler");


glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, m_field_update_fb );

int cols_log2 = (int)ceilf(log2(sqrt((double)m_config.m_function_slices)));
int cols = 1<<cols_log2;
int rows = (m_config.m_function_slices+cols+1)/cols;
int tsize = 1<<m_config.m_function_tsize_log2;
int func_w = tsize*cols;
int func_h = tsize*rows;


glViewport( 0, 0, func_w, func_h );
glBegin( GL_QUADS );
glVertex2f( -1.0f, -1.0f );
glVertex2f(  1.0f, -1.0f );
glVertex2f(  1.0f,  1.0f );
glVertex2f( -1.0f,  1.0f );
glEnd();

glPopAttrib();



int N = buildHistopyramid();
```

Algoritmo 5: Inicialização do terreno, variáveis e texturas do *shader*

de permutação e de vetores gradientes necessárias à função de ruído. Neste caso é utilizada uma versão otimizada da função de ruído onde a tabela de permutação é armazenada em uma textura. Atualiza-se, então, o *framebuffer* e calculam-se as dimensões da base do *Histogram Pyramids* e então começa-se a construção da pirâmide. Detalhes sobre a construção da pirâmide podem ser encontrados em [2].

No processo de extração, percorremos a pirâmide e extraímos a isosuperfície (no caso o terreno). O algoritmo transforma o fluxo de dados 3D do campo escalar em um fluxo de vértices, gerados em tempo de execução enquanto se renderiza a isosuperfície, é possível armazenar a geometria em um buffer na GPU na forma de *transform feedback buffers* ou via *render-to-vertex-buffer*. A primeira opção está disponível apenas em *hardwares* com mod-

elo de *shader* 4.0. A segunda opção está presente em diferentes modelos de placas gráficas. As duas opções são utilizadas nesta implementação, sendo que a primeira opção apresenta melhores resultados [2]. A abordagem utilizando *transform feedback buffers* é interessante caso uma isosuperfície seja necessária para algum tipo de pós-processamento onde a geometria pode ser armazenada diretamente como uma lista compacta de triângulos na memória da GPU. Nos apêndices, na seção A.2, pode-se encontrar o algoritmo relativo à implementação completa da construção da pirâmide.

Capítulo 5

Resultados

Os testes relativos aos resultados apresentados neste capítulo foram realizados em um computador com processador Atlhon64 3500+ a 2.2 Ghz, 2GB de RAM e uma GeForce 8800GTS com 512Mb de RAM.

Os resultados foram variados e promissores, onde os terrenos podem ser customizados de várias maneiras, seja manipulando a topografia ou produzindo alguma colorização ou texturização diferente. Várias dimensões do volume foram testadas, como $32^3, 64^3, 128^3, 256^3$, para se verificar qual a densidade que melhor representa o terreno sem esgotar os recursos da placa gráfica.

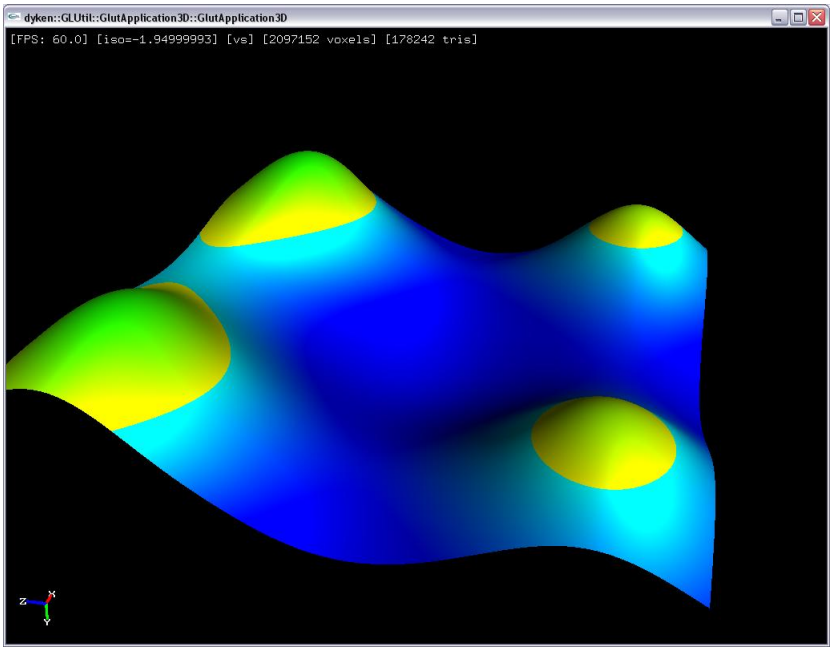

Figura 5.1: Terreno gerado utilizando apenas a função de ruído

Como apresentado no capítulo 2 a função de ruído, apenas, não consegue criar terrenos interessantes, por isso é utilizada a sua iteração fractal. Na Figura 5.2 temos uma visão geral de como a implementação se comporta. Este é um terreno gerado com uma densidade de 128^3 e com a função de turbulência.

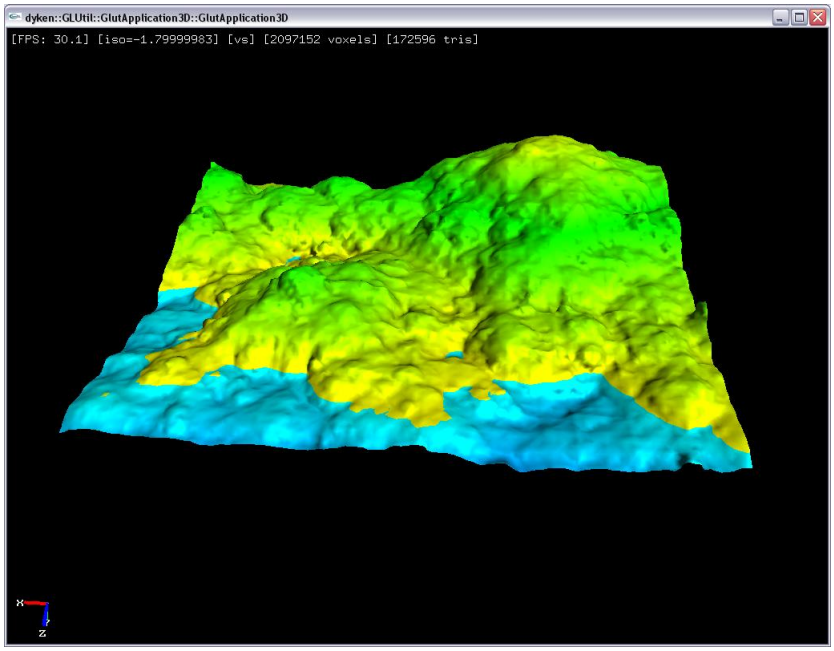

Figura 5.2: Terreno gerado com a função de turbulência

Uma forma interessante de se customizar o terreno é aplicar uma função de *warp* na coordenada antes de utilizá-la na função que dará forma ao campo escalar. Isto pode apresentar resultados diferentes dependendo da função de *warp* utilizada. Neste trabalho a própria função de ruído foi utilizada para distorcer a coordenada antes desta ser passada à função de turbulência. O resultado pode ser visto na Figura 5.3.

O terreno, utilizando este tipo de abordagem, fica com uma aparência mais orgânica e até mesmo alienígena, dependendo das manipulações realizadas, como podemos ver na Figura 5.4

A variação de densidade do modelo afeta diretamente o desempenho do algoritmo e também a capacidade de representação do terreno. Podemos ver uma comparação entre cada modelo gerado na Tabela 5.1.

Os valores marcados com asterisco significam estão limitados a 60 quadros por se-

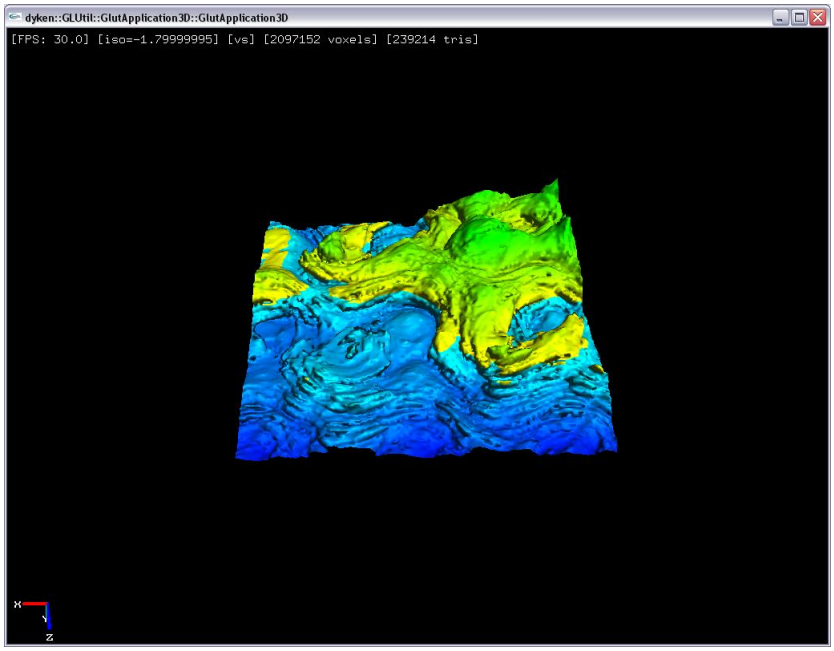

Figura 5.3: Terreno gerado com *warping* de coordenada

gundo, ou seja, na realidade apresentam uma taxa maior de frames por segundo. Com uma densidade igual 32^3 o terreno apresenta com poucos detalhes e sua representação não é muito interessante do ponto de vista visual, justamente pelo baixo número de triângulos. Porém, possui um peso computacional bem baixo e pode ser utilizado para a geração de terrenos, caso texturizado corretamente. A representação com densidade igual a 256^3 é extremamente pesada apresentando um número muito alto de triângulos. Nesta representação o terreno fica muito detalhado e as curvas bem suaves, mas não se pode utilizar este nível de densidade na prática sem algum modo de se reduzir o número de triângulos, como *culling* ou nível de detalhe. Temos assim que as densidades que melhor representam o terreno são de 64^3 e 128^3 onde temos uma boa relação de triângulos/*voxels*/*fps*.

Podemos ver nas Figuras 5.7, 5.8, 5.9 e 5.10 as diferentes representações dos terrenos em cada dimensão. Nota-se que a diferença entre as dimensões 32 e 64 já é grande. Em relação à densidade de 128 temos uma diferença ainda maior. Porém, esta idéia não acompanha a densidade de 256 onde a diferença visual para a densidade 128 é muito pequena para se notar alguma melhora, mas o custo computacional desta ultima representação é substancialmente maior.

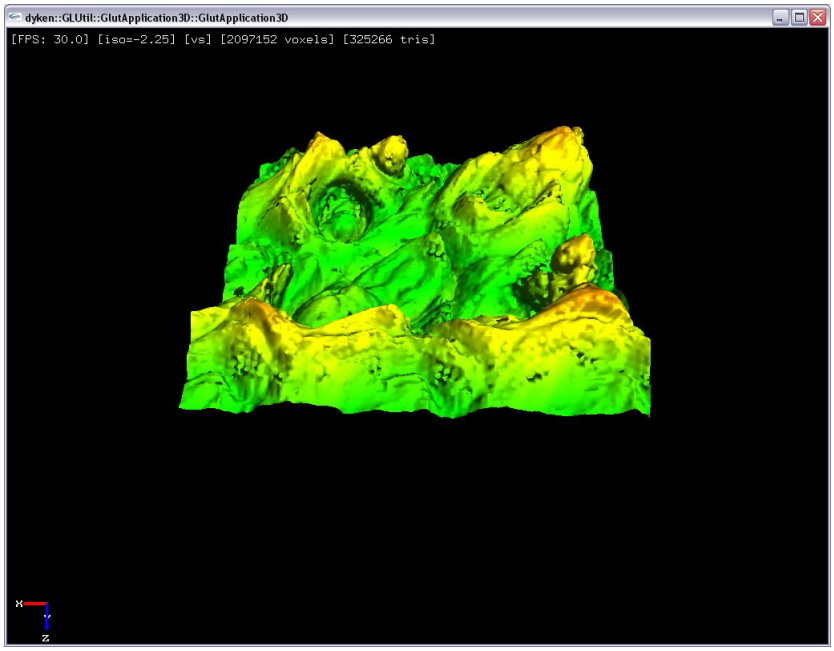

Figura 5.4: Terreno alienigena também gerado com *warping* de coordenada

O custo computacional da iteração fractal da função de ruído, tem efeito direto sobre o desempenho do método. Temos que a função de turbulência gera uma carga maior para a GPU refletindo na taxa de quadros por segundo. Isto pode ser observado na Tabela 5.2.

A diferença de desempenho está diretamente relacionada ao número de iterações fractais que serão utilizadas. Na Tabela 5.1 os valores foram obtidos utilizando-se 7 iterarções fractais (oitavas) da função de turbulência, diminuindo este número de iterações existe um ganho considerável de desempenho, porém a qualidade do visual do terreno em si dimuniu consideravelmente.

Nas Figuras 5.5 e 5.6 podemos ver a comparação do mesmo terreno utilizando 4 iterações e 7 iterações fractais da função de turbulência.

A variação dos valores nas funções utilizadas podem ser vistos nas Figuras 5.11 à 5.14. Primeiro, serão apresentadas imagens com as variações na variável de amplitude da função. A função de amplitude é baseada em um valor de ganho (*gain*) e pode ser vista na fórmula abaixo:

$$amplitude = amplitude * ganho; \tag{5.1}$$

Nestas Figuras podemos ver que o terreno varia bastante baseado apenas na função de amplitude. Porém, nem todos os valores são adequados para um terreno realista, como é o caso do terreno 5.14, onde temos um valor de ganho, no caso qualquer valor acima de 0.5, que não resulta em uma aparência normal de um terreno.

Podemos também utilizar valores negativos de ganho. Estes apresentam resultados bastante convincentes da mesma forma que os valores positivos. Nas figuras 5.15 à 5.18 podemos ver os resultados destas variações. Da mesma forma a partir de -0.5 a função de amplitude começa a apresentar resultados que deformam a aparência do terreno.

Outra variação possível é o valor de frequência da função. Este valor depende da variação da lacunaridade de acordo com a função abaixo:

$$frequencia = frequencia * lacunaridade; \tag{5.2}$$

Novamente temos um intervalo interessante de valores que geram possíveis terrenos. Nas Figuras 5.19 à 5.22 temos as variações da lacunaridade entre os valores 0.0 à 2.4. Nos testes realizados durante este trabalho valores superiores a 2.4 não são interessantes à geração de terrenos, pois o resultado obtido não representa bem um terreno na natureza. O mesmo ocorre para valores negativos desta função.

O último resultado a ser apresentado é relativo ao isovalor escolhido para representar o terreno. A variação deste valor modifica drasticamente a topografia do terreno, assim é possível criar diferentes terrenos apenas modificando este valor. Alguns exemplos podem ser vistos nas Figuras 5.23 à 5.26.

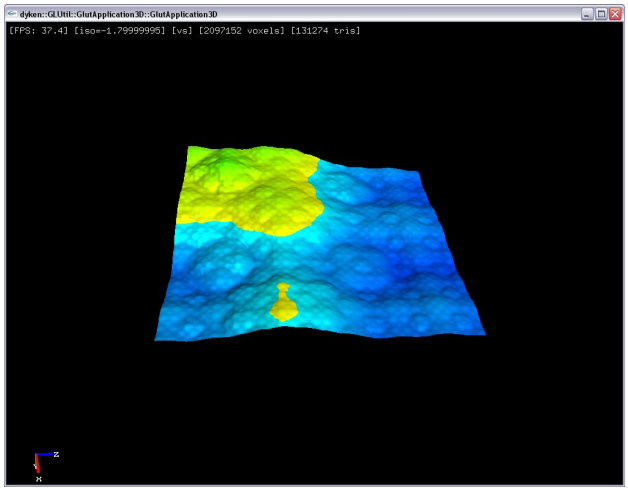

Figura 5.5: Terreno gerado com 4 iterações fractais

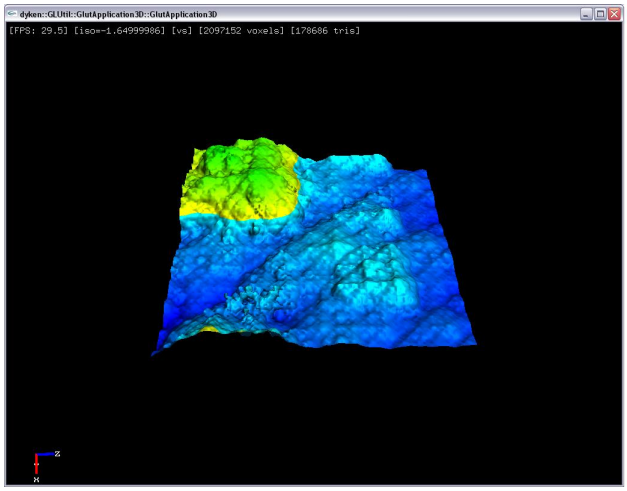

Figura 5.6: Terreno gerado com 7 iterações fractais

Densidade	32^3	64^3	128^3	256^3
FPS	60*	60*	30	6
Voxels	32.000+	250.000+	2.000.000+	16.000.000+
Triângulos	≈ 7.500	≈ 43.000	≈ 165.000	≈ 680.000

Tabela 5.1: Comparação entre as diferentes densidades utilizando a função de turbulência (7 iterações)

Densidade	32^3	64^3	128^3	256^3
FPS	60*	60*	60*	12
Voxels	32.000+	250.000+	2.000.000+	16.000.000+
Triângulos	≈ 7.500	≈ 43.000	≈ 165.000	≈ 680.000

Tabela 5.2: Comparação entre as diferentes densidades utilizando a função de ruído

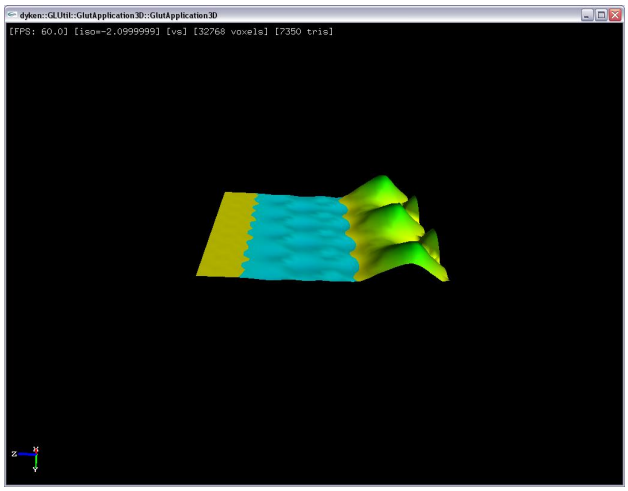

Figura 5.7: Terreno com densidade 32^3

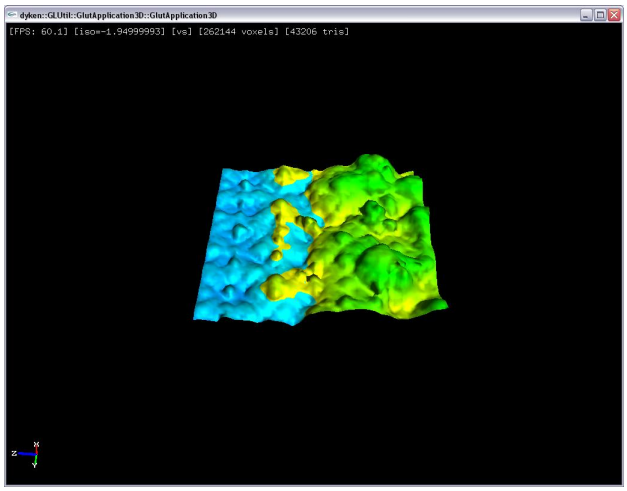

Figura 5.8: Terreno com densidade 64^3

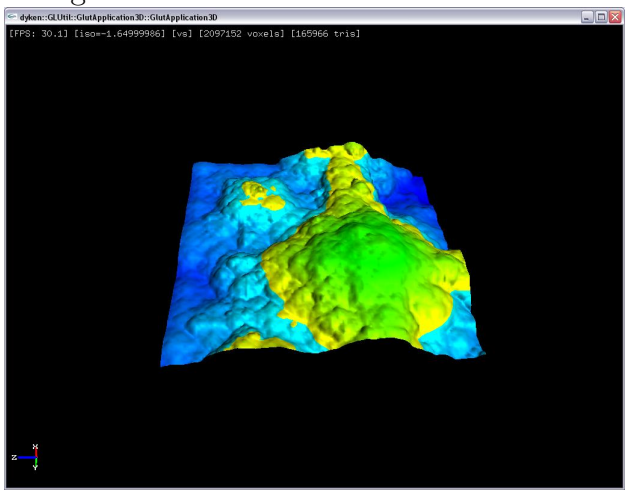

Figura 5.9: Terreno com densidade 128^3

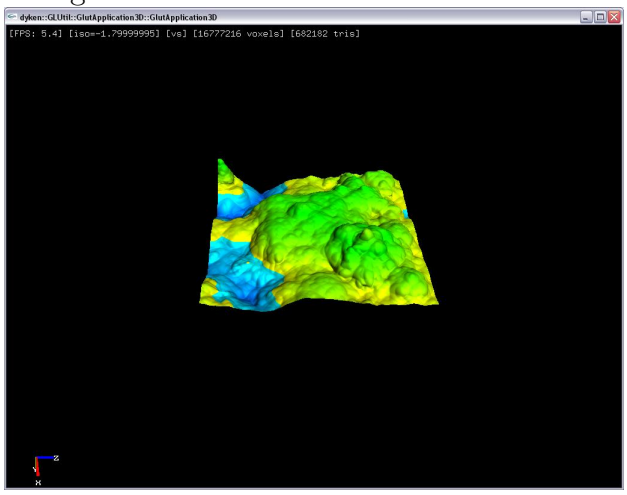

Figura 5.10: Terreno com densidade 256^3

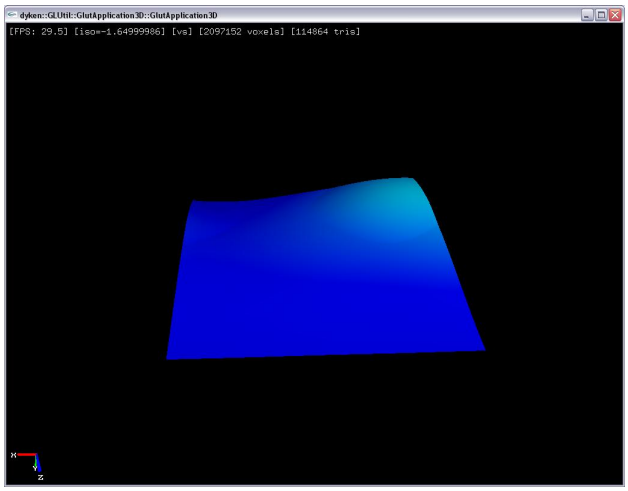

Figura 5.11: Ganho igual a 0.0

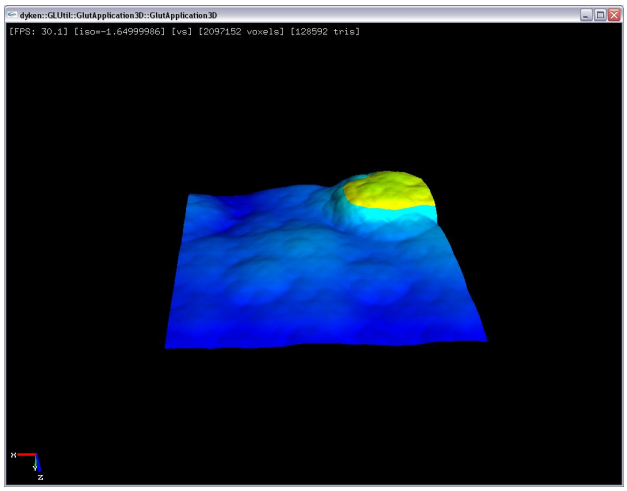

Figura 5.12: Ganho igual a 0.3

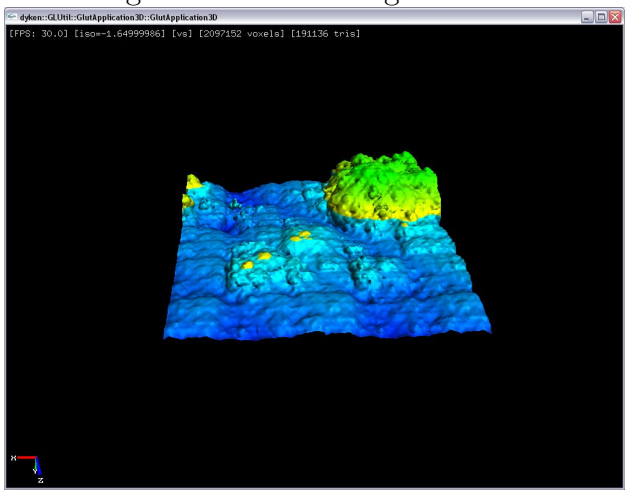

Figura 5.13: Ganho igual a 0.5

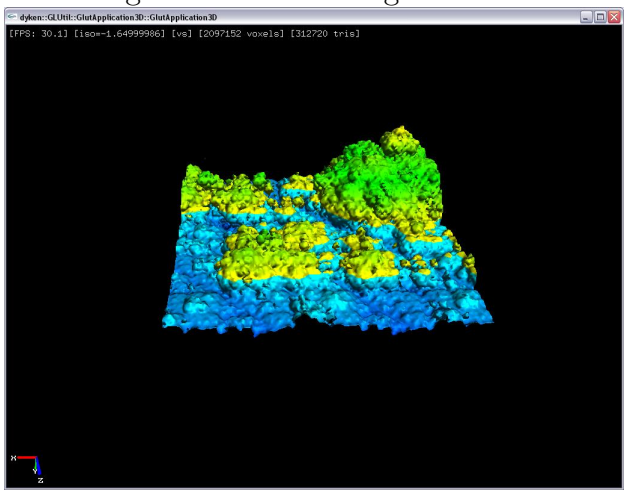

Figura 5.14: Ganho igual a 0.6

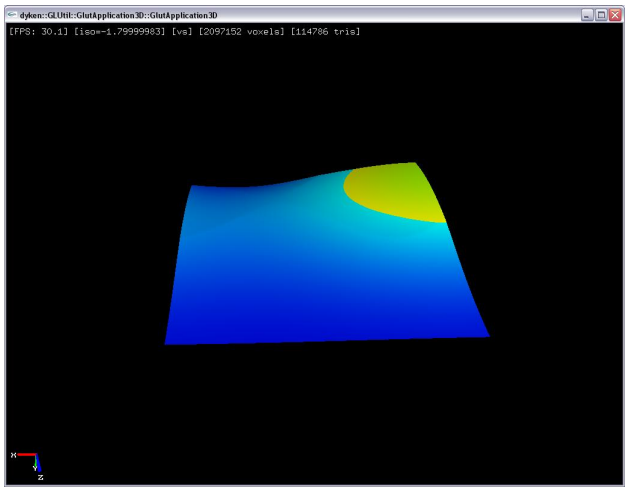

Figura 5.15: Ganho igual a 0.0

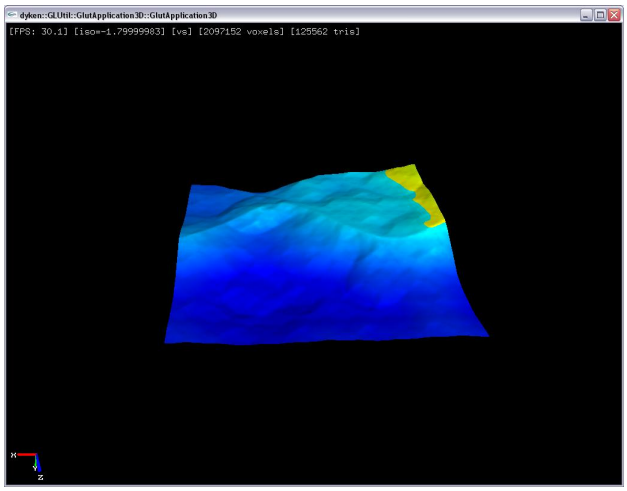

Figura 5.16: Ganho igual a -0.3

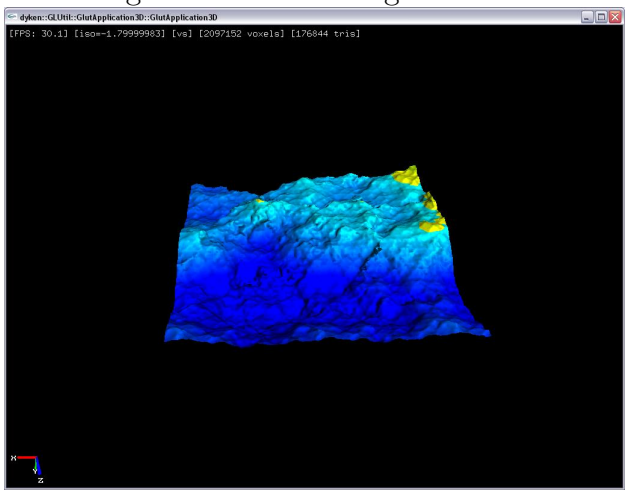

Figura 5.17: Ganho igual a -0.5

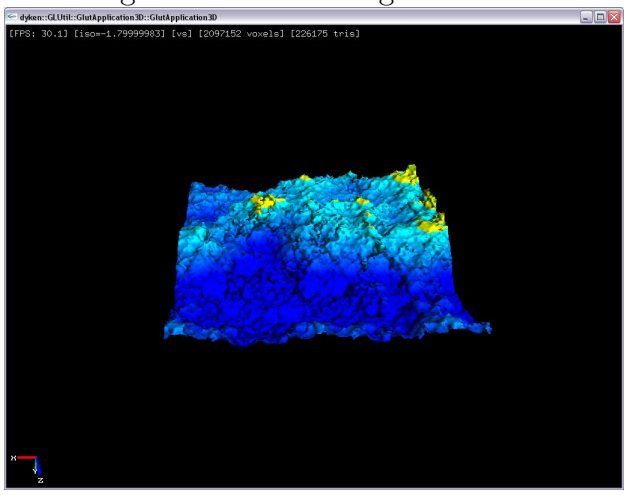

Figura 5.18: Ganho igual a -0.6

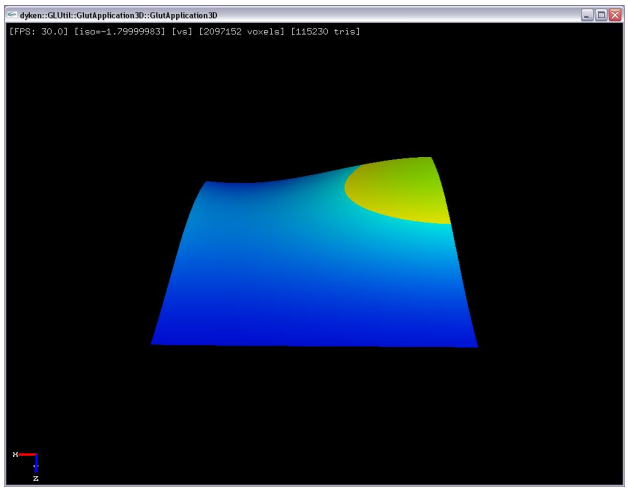

Figura 5.19: lacunaridade igual a 0.0

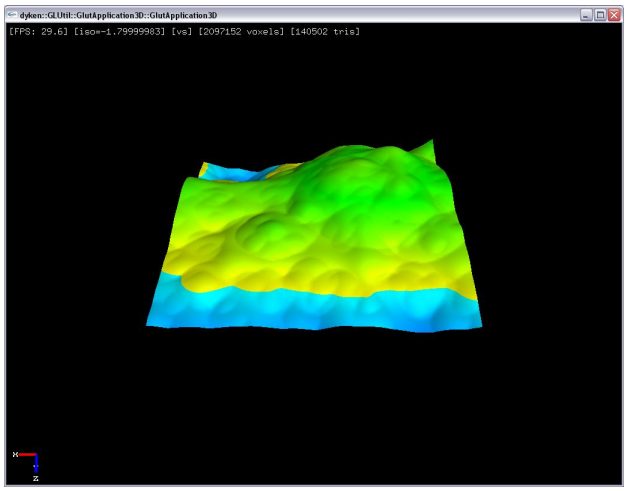

Figura 5.20: lacunaridade igual a 1.5

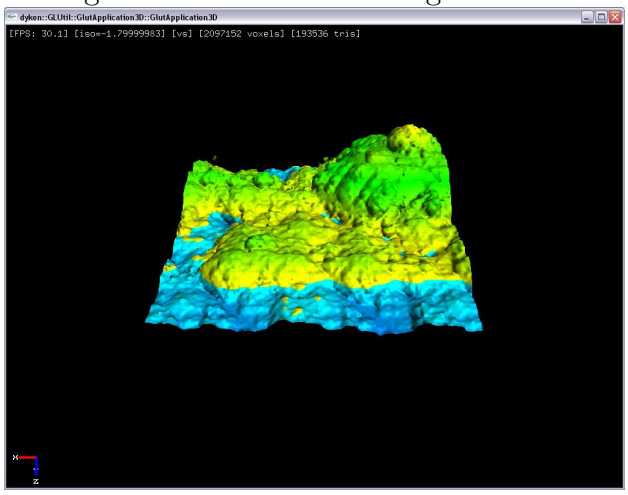

Figura 5.21: lacunaridade igual a 2.0

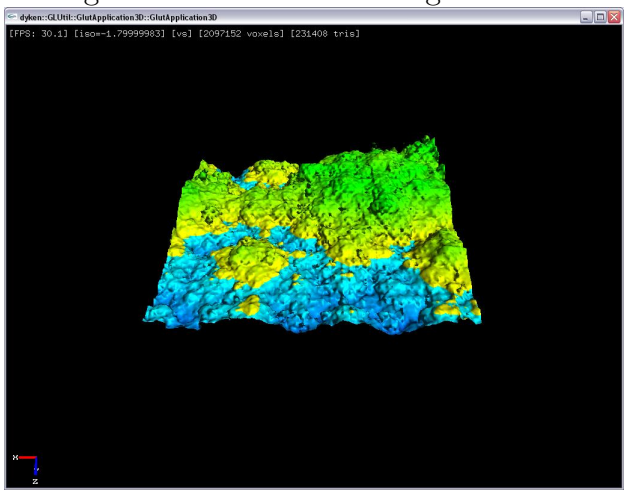

Figura 5.22: lacunaridade igual a 2.4

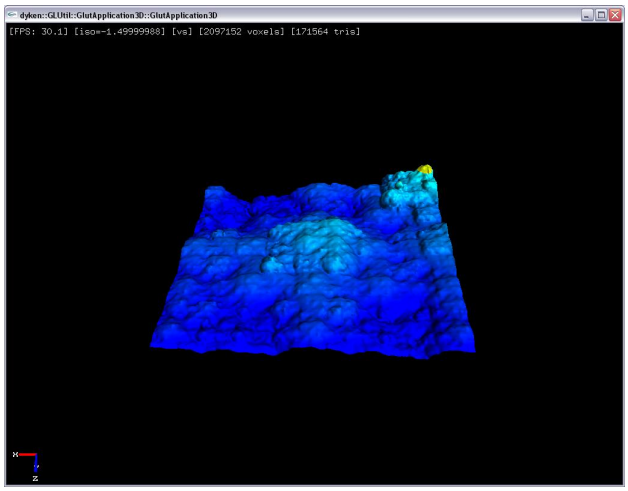

Figura 5.23: Isovalor de -1.5

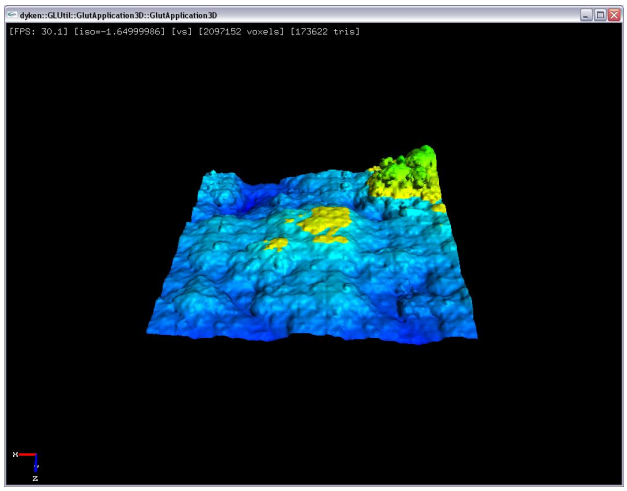

Figura 5.24: Isovalor de -1.65

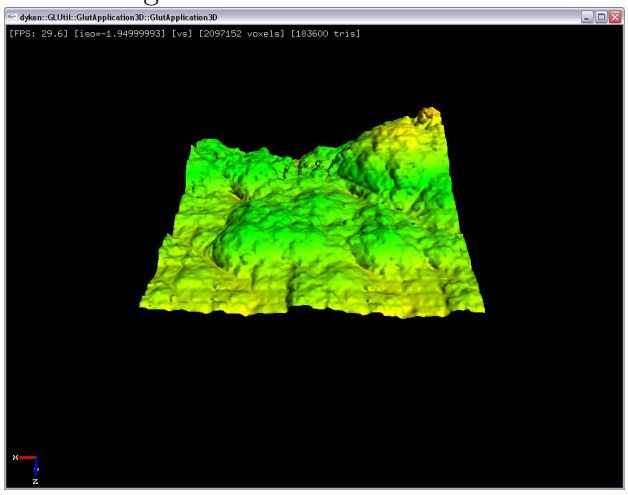

Figura 5.25: Isovalor de -1.95

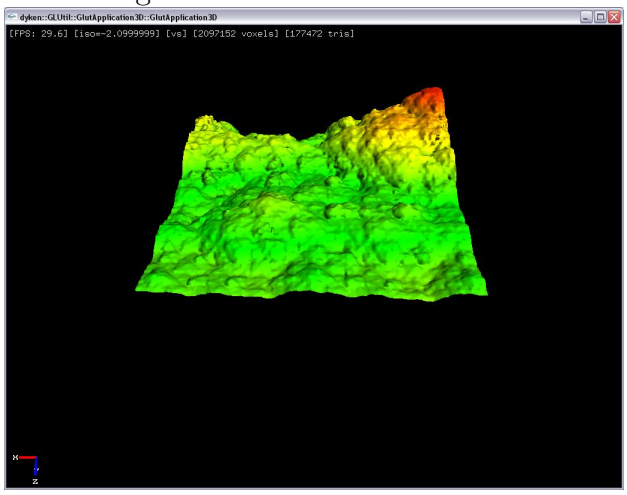

Figura 5.26: Isovalor de -2.1

Capítulo 6

Conclusão e Trabalhos Futuros

A geração procedural de terrenos por extração de isosuperfícies apresenta um grande desafio em virtude da natureza deste tipo de algoritmo. São algoritmos pesados computacionalmente, mas que apresentam resultados bons e possibilitam o desenvolvimento de uma variedade de aplicações que não são possiveis com outras abordagens.

Como contribuições iniciais, este trabalho faz uma revisão sobre as funções procedurais e como aplicar estas funções para a geração procedural de terrenos, além de apresentar uma extensa revisão dos métodos de extração de isosuperfícies.

Este trabalho apresenta um modelo de geração procedural de terreno utilizando uma abordagem volumétrica, sendo ambos implementados na placa gráfica. A utilização de uma abordagem volumétrica implica no uso de métodos de extração de isosuperfícies, que é um algoritmo custoso, mas que implementado em GPU apresenta um bom desempenho.

Testes preliminares mostram que é viável este tipo de abordagem. Os terrenos gerados apresentam uma boa qualidade e apresentam várias formas de customização. Pode-se utilizar este tipo de geração para representar dados de sondas de escavação, perfuração de poços e vários outras aplicações que não são possíveis em outros tipos de geração.

Como trabalhos futuros é proposta a implementação do terreno em uma superfície esférica, o que representa um desafio devido a mudança de topologia, para a geração de planetas por exemplo. Além disso, a implementação de texturização é importante para o realismo do terreno e deve ser o próximo passo de desenvolvimento. Outro ponto

importante é o desenvolvimento do método utilizando o *framework* CUDA, que deve prover um ganho de desempenho, pois agora a saída de dados para texturas 2D são suportadas.

Referências

[1] Clua, E. W. G., *Modelagem Procedimental para Visualização de Elementos da Natureza*, Master's thesis, Departamento de Informática, Pontífica Universidade Católica do Rio de Janeiro (Dezembro 1999).

[2] Dyken, C. and Ziegler, G., "High-speed marching cubes using histogram pyramids," *Computer Graphics Forum* **27**(8), 2028 – 2039 (2008).

[3] Perlin, K., "An image synthesizer," *SIGGRAPH 85 Proceedings* **19**, 287–296 (1985).

[4] Mandelbrot, B., "How long is the coast of britain? statistical self-similarity and fractional dimension," *Science. New Series* **156**(3775), 636–638 (1967).

[5] Peitgen, R., [*The Beauty of Fractals*], Springer-Verlag (1986).

[6] Mandelbrot, B., "The fractal geometry of nature," in [*The Fractal Geometry of Nature*], W. H. Freeman and Co (1982).

[7] Ebert, D., Musgrave, K., Peachey, D., Perlin, K., and Worley, S., [*Texturing and Modeling: A procedural approach*], Morgan Kaufmann, 3rd ed. (2003).

[8] Olsen, J., "Realtime procedural terrain generation: Realtime synthesis of eroded fractal terrain for use in computer game," (October 2004).

[9] Gavin, S. and Miller, P., "The definition and rendering of terrain maps," *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* , 39–48 (1986).

[10] Geiss, R., "Generating complex procedural terrains using the gpu," in [*GPU Gems 3*], Addison-Wesley. (2007).

[11] Lorensen, W. and Cline, H. E., "Marching cubes: A high resolution 3d surface construction algorithm," *SIGGRAPH 87 Proceedings* **21**(4), 163:170 (1987).

[12] Schaefer, S. and Warren, J., "Dual marching cubes: Primal contouring of dual grids," in [*Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*], (2004).

[13] Wilhelms, J. and Gelder, A. V., "Octrees for faster isosurface generation," *ACM SIGGRAPH Computer Graphics* **24**(5), 57 – 62 (1990).

[14] Shekhar, R., Fayyad, E., Yagel, R., and Cornhill, F., "Octree-based decimation of marching cubes surfaces," *Proceedings of the 7th conference on Visualization '96* , 335 – ff. (1996).

[15] Boada, I., Navazo, I., and Scopigno, R., "Multiresolution volume visualization with a texture-based octree," *The Visual Computer (2001)* **17**(8), 185 – 197 (2001).

[16] Klein, T., Stegmaier, S., and Ertl, T., "Hardware accelerated reconstruction of polygonal isosurface representations on unstructured grids," *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference* , 186 – 195 (2004).

[17] Kipfer, P. and Westermann, R., "Gpu construction and transparent rendering of isosurface," *Proceedings of Vision, Modeling, and Visualization (VMV) 2005* (2005).

[18] Buatois, L., Caumon, G., and Levy, B., "Gpu accelerated isosurface extraction on tetrahedral grids," *International Symposium on Visual Computing* (2006).

[19] Harris, M., "Parallel prefix sum (scan) with cuda," (2007).

[20] Ziegler, G., Tevs, A., Theobalt, C., and Seidel, H. P., "Gpu point list generation through histogram pyramids," tech. rep., Max-Planck-Institut für Informatik (2006). Technical Report MPI-I-2006-4-002.

[21] Perlin, K., "Making noise," (December 2008). `http://www.noisemachine.com`.

[22] Cook, R. L. and DeRose, T., "Wavelet noise," *ACM Transactions on Graphics (TOG)* **24**(3), 803 – 811 (2005).

[23] Worley, S., "A cellular texture basis function," *Computer Graphics - SIGGRAPH 96 Proceedings* **30**, 291–294 (1996).

[24] Musgrave, F. K., Kolb, C. E., and Mace, R. S., "The synthesis and rendering of eroded fractal terrains," *Computer Graphics* **23**, 41–50 (1989).

[25] Evertsz, C. J. G. and Mandelbrot, B. B., [*Multifractal Measures*], ASpringer-Verlag (1992).

[26] Knoll, A., "A survey of implicit surface rendering methods, and a proposal for a common sampling framework," *GI Lecture Notes in Informatics, Proceedings of the 2nd IRTG Workshop* (2007).

[27] Pascucci, V., "Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping," *Joint Eurographics - IEEE TVCG Symposium on Visualization* , 292 – 300 (2004).

[28] Goetz, F., Junklewitz, T., and Domiks, G., "Real-time marching cubes on the vertex shader," *Eurographics 2005 Short Presentations* (2005).

[29] Johansson, G. and Carr, H., "Accelerating marching cubes with graphics hardware," *CASCON 06 - Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research* (2006).

[30] Uralsky, Y., "Dx10: Practical metaballs and implicit surfaces," (2006).

[31] Harris, M. J., Scheuermannt, I. W. V. B., and Lastra, A., "Simulation of cloud dynamics on graphics hardware," *Proceedings of Graphics Hardware 2003* (2003).

[32] Green, S., "Implementing improved perlin noise," in [*GPU Gems 2*], Addison-Wesley. (2005).

[33] Perlin, K., "Implementing improved perlin noise," in [*GPU Gems*], Addison-Wesley (2005).

[34] Dembogurski, B., Clua, E. W. G., Vieira, M. B., and Leta, F., "Procedural terrain generation with marching cubes at gpu level," *Simpósio Brasileiro de Jogos - SBGames* (2008).

APÊNDICE A – Códigos

A.1 Shader das Funções de Ruído e Turbulência

```
uniform float escalaN;
uniform float lacunarityN;
uniform float gainN;

uniform sampler2D permSampler2d;
uniform sampler1D permGradSampler;

uniform sampler1D perm2Sampler;
uniform sampler1D grad2Sampler;

varying vec2 vTexCoord;
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
vec3 fade(vec3 t)
{
   return t * t * t * (t * (t * 6.0 - 15.0) + 10.0); // nova curva
   //return t * t * (3 - 2 * t); // curva antiga
}
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
float gradperm(float x, vec3 p)
{
   return dot(texture1D(permGradSampler, x).rgb, p);
}

vec4 perm2d(vec2 p)
{
   return texture2D(permSampler2d, p);
}
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
```

```
float onoise(vec3 p)
{
   //Cubo unitário que contém o ponto
   vec3 P = mod(floor(p), 256.0);

   //x,y,z relativo do ponto no cubo
   p -= floor(p);

   //cálculo da função para cada x,y,z
   vec3 f = fade(p);

   P = P / 256.0;
   const float one = 1.0 / 256.0;

   // coordenadas hash dos 8 cantos do cubo
   vec4 AA = perm2d(P.xy) + P.z;

   //adicionar interpolando os 8 valores
   return mix( mix(
               mix( gradperm(AA.x, p ),
                    gradperm(AA.z, p + vec3(-1, 0, 0) ),
                    f.x),
                mix( gradperm(AA.y, p + vec3(0, -1, 0) ),
                gradperm(AA.w, p + vec3(-1, -1, 0) ), f.x),
                                f.y),

                mix(mix(gradperm(AA.x+one, p + vec3(0, 0, -1)),
                        gradperm(AA.z+one, p + vec3(-1, 0, -1)),
                                                f.x),
                    mix(gradperm(AA.y+one, p + vec3(0, -1, -1)),
                        gradperm(AA.w+one, p + vec3(-1, -1, -1)),
                                                f.x),
                                                f.y),
                                                f.z);
}
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
float grad(float x, vec3 p)
{
   //return dot(texture1D(gradSampler, x * 16.0).rgb, p);
   //return dot(g[int(mod(x * 16.0, 16.0))], p);
   return dot(texture1D(grad2Sampler, x * 16.0).rgb, p);
}
```

```
float perm(float x)
{
   //return texture1D(permSampler, x).r;
   //return int(permutation[int(mod(x, 256.0))]);
   return texture1D(perm2Sampler, x).r;
}
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
float inoise(vec3 p)
{
   //Cubo unitário que contém o ponto
   vec3 P = mod(floor(p), 256.0);

   //x,y,z relativo do ponto no cubo
   p -= floor(p);

   //cálculo da função para cada x,y,z
   vec3 f = fade(p);

   P = P / 256.0;
   const float one = 1.0 / 256.0;

   //adicionar interpolando os 8 valores
   float A = perm(P.x) + P.y;
   vec4 AA;
   AA.x = perm(A) + P.z;
   AA.y = perm(A + one) + P.z;
   float B =  perm(P.x + one) + P.y;
   AA.z = perm(B) + P.z;
   AA.w = perm(B + one) + P.z;

   //adicionar interpolando os 8 valores
   return mix(
            mix(
                mix(grad(perm(AA.x), p), grad(perm(AA.z),
                    p + vec3(-1, 0, 0)), f.x),
                mix(grad(perm(AA.y), p + vec3(0, -1, 0)),
                    grad(perm(AA.w), p + vec3(-1, -1, 0)), f.x),
                f.y),
            mix(
                mix(grad(perm(AA.x + one), p + vec3(0, 0, -1)),
                    grad(perm(AA.z + one), p + vec3(-1, 0, -1)),
```

```
                                f.x),
                mix(grad(perm(AA.y + one), p + vec3(0, -1, -1)),
                    grad(perm(AA.w + one), p + vec3(-1, -1, -1)),
                    f.x),
                f.y),
            f.z);
}
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
float turbulence(vec3 p, int octaves, float lacunarity, float gain)
{
   float sum = 0.0;
   float freq = 1.0, amp = 1.0;
   for(int i=0; i<octaves; i++) {
      sum += abs(onoise(p*freq))*amp;
      freq *= lacunarity+lacunarityN;
      amp *= gain+gainN;
   }
   return sum;
}
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
void main(void)
{

   vec2 foo = vec2(FUNC_COLS,FUNC_ROWS)*gl_TexCoord[0].xy;
   float slice = dot(vec2(1.0,float(FUNC_COLS)), floor(foo));
   vec3 p = 1.2*(vec3( 2.0*fract(foo),
               (2.0/FUNC_SLICES)*(slice+0.5)) - vec3(2.5));

   // Alien Terrain
   //float warp = onoise(p);
   //float warp = turbulence(p, 3, 2.0, 0.50);
   //float Anoise = turbulence((p+warp)*0.25*escalaN,
   //                          5, 2.0, 0.50);

   //float Anoise = onoise(p*escalaN);
   float Anoise = turbulence(p*escalaN, 7, 2.0, 0.56);

   gl_FragColor = vec4(p.y+Anoise);

   return;
}
```

A.2 Construção da Pirâmide

```
GLMarcher::buildHistopyramid()
{
        if(m_profile) {
                glBeginQuery( GL_TIME_ELAPSED_EXT, m_timers[0] );
        }

        glPushAttrib( GL_VIEWPORT_BIT );

        if(m_config.m_gl_vs_tile_table ||
           m_config.m_gl_fs_tile_table ||
           m_config.m_gl_gs_tile_table)
        {
                glActiveTextureARB( GL_TEXTURE3_ARB );
                glBindTexture( GL_TEXTURE_1D, m_tex_tile_table );
        }

        glActiveTextureARB( GL_TEXTURE2_ARB );
        glBindTexture( GL_TEXTURE_1D, m_tex_tricount );

        glActiveTextureARB( GL_TEXTURE1_ARB );
        if(m_config.m_gl_use_tex3d)
                glBindTexture( GL_TEXTURE_3D, m_tex_function );
        else
                glBindTexture( GL_TEXTURE_2D, m_tex_function );

        glActiveTextureARB( GL_TEXTURE0_ARB );
        glBindTexture( GL_TEXTURE_2D, m_hp_tex );
        glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0 );
        glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);

        // --- nível base da pirâmide
        glUseProgram( m_baselevel_p );
        glUniform1f( m_uniform_baselevel_threshold,
                     m_isovalue_scale*m_config.m_function_isovalue );

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, m_hp_framebuffers[0]);
        glViewport( 0, 0, 1<<m_hp_size_log2, 1<<m_hp_size_log2 );
        glBegin( GL_QUADS );
        glVertex2f( -1.0f, -1.0f );
        glVertex2f(  1.0f, -1.0f );
        glVertex2f(  1.0f,  1.0f );
```

```
glVertex2f( -1.0f,  1.0f );
glEnd();

if(m_profile) {
        glEndQuery( GL_TIME_ELAPSED_EXT );
        glBeginQuery( GL_TIME_ELAPSED_EXT, m_timers[1] );
}

glBindTexture( GL_TEXTURE_2D, m_hp_tex );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0 );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);

// --- first reduction
glUseProgram( m_reduce_base_p );
glUniform2f( m_reduce_base_uniform_delta,
                -0.5/(1<<m_hp_size_log2),
           0.5/(1<<m_hp_size_log2) );

glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, m_hp_framebuffers[1] );
glViewport( 0, 0, 1<<(m_hp_size_log2-1), 1<<(m_hp_size_log2-1) );
glBegin( GL_QUADS );
glVertex2f( -1.0f, -1.0f );
glVertex2f(  1.0f, -1.0f );
glVertex2f(  1.0f,  1.0f );
glVertex2f( -1.0f,  1.0f );
glEnd();
CHECK_GL;

// --- resto das reduções da pirâmide
glUseProgram( m_reduce_p );
for(int m=2; m<=m_hp_size_log2; m++) {
    glBindTexture( GL_TEXTURE_2D, m_hp_tex );
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, m-1);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, m-1);

        glUniform2f(m_reduce_uniform_delta,
                        -0.5/(1<<(m_hp_size_log2+1-m)),
             0.5/(1<<(m_hp_size_log2+1-m)) );

        glBindFramebufferEXT( GL_FRAMEBUFFER_EXT,
                                    m_hp_framebuffers[m] );

        glViewport( 0, 0, 1<<(m_hp_size_log2-m),
```

```
                                        1<<(m_hp_size_log2-m) );

                glBegin( GL_QUADS );
                glVertex2f( -1.0f, -1.0f );
                glVertex2f(  1.0f, -1.0f );
                glVertex2f(  1.0f,  1.0f );
                glVertex2f( -1.0f,  1.0f );
                glEnd();
        }

        GLfloat mem[4];
        glTexParameteri( GL_TEXTURE_2D,
                         GL_TEXTURE_BASE_LEVEL,
                         m_hp_size_log2 );

        glTexParameteri( GL_TEXTURE_2D,
                         GL_TEXTURE_MAX_LEVEL,
                         m_hp_size_log2 );

        glGetTexImage( GL_TEXTURE_2D,
                       m_hp_size_log2,
                       GL_RGBA, GL_FLOAT, &mem[0] );
        CHECK_GL;

        int N = (int)(mem[0]+mem[1]+mem[2]+mem[3]);
        m_no_triangles += N;
        m_no_frames++;



        glPopAttrib();

        if(m_profile) {
                glEndQuery( GL_TIME_ELAPSED_EXT );
        }
        return N;
}
```