Geração procedimental de ambientes para jogos eletrônicos.

Gustavo Teixeira da Cunha Coelho

Trabalho de conclusão de curso

Orientador: Prof. Dr. Marco Dimas Gubitoso

São Paulo, Fevereiro de 2014

Agradecimentos

Gostaria de simplesmente agradecer todas as pessoas que me ajudaram durante o curso. Também gostaria de pedir desculpas por todos os possíveis transtornos e perguntas estúpidas que eu fiz durante esses anos.

Introdução

É comum jogos eletrônicos possuírem um "mapa de jogo", uma descrição do mundo na qual um ou mais jogadores existem e jogam. Um mapa de jogo pode descrever, por exemplo, uma casa por onde um jogador se move, um terreno natural como uma montanha ou mesmo um lugar que não existe em nosso mundo.

Tais mapas precisam ser feitos manualmente ou podem ser gerados proceduralmente usando um ou mais algoritmos. Ao se fazer um mapa manualmente, o desenvolvedor precisa se importar com vários detalhes do terreno, como sua verossimilidade, tamanho, se atentar a detalhes e popular o mapa com objetos, o que acaba tomando uma grande parte de seu tempo. Logo ele pode optar por implementar um algoritmo que gere mapas para ele, usando seu resultado final no jogo ou usando estes mapas gerados como base e então polindo eles manualmente para incluir em seu jogo. Isto, porém, também irá tomar seu tempo. Este tempo gasto com a geração de mapas poderia ser usado para o desenvolvimento de outras partes do jogo.

O objetivo deste trabalho, então, é desenvolver uma ferramenta para a criação desses mapas de jogo de acordo com as necessidades dos desenvolvedores, assim economizando tempo que seria gasto na implementação de código ou na criação de mapas manualmente. Os mapas criados aceitam vários parâmetros para se adequar às necessidades da maior quantidade de desenvolvedores possível.

Sendo assim, foram estudadas e implementadas algumas técnicas de geração de terreno aleatório, sua visualização e alteração para uso por um usuário final, que poderá usar tal ambiente gerado em seu jogo como quiser. Também implementei um visualizador para os mapas gerados para efeitos de depuração dos algoritmos implementados.

Sumário

Parte I

Parte Objetiva

Capítulo 1

Conceitos e tecnologias estudadas

1.1 Relevo

Uma parte importante do mundo no qual vivemos ou de um mundo virtual é o seu relevo. Se pensarmos no mundo no qual vivemos, é a forma das superfícies que compõem o planeta. O mesmo vale para o mundo virtual que se deseja ter num jogo. Alguns exemplos de relevos são planícies, planaltos e montanhas. Como o relevo é uma parte bastante importante do mundo, boa parte do trabalho consiste da geração procedimental de um relevo se utilizando de várias técnicas.

1.2 Mapa de altura

Como já dito, queremos gerar um relevo. Queremos também que ele se assemelhe a um terreno natural, como uma planície ou uma montanha.

Ou seja, o que queremos gerar é na verdade um *mapa de altura*, uma estrutura de dados, normalmente uma imagem, que armazena valores que descrevem a elevação, terreno ou "altura" de uma superfície. Cada pixel da imagem representa um valor de altura associado a um plano. É normalmente usada em computação gráfica para a exibição de gráficos tridimensionais ou na técnica de *bump mapping*[1]. Mapas de altura podem ser gerados manualmente, usando um editor de imagens qualquer ou uma ferramenta específica, ou através de um algoritmo de geração de terrenos.

1.3 Paisagens Fractais

Queremos então gerar mapas de altura que descrevam um terreno plausível na vida real. Alguns tipos de terrenos naturais apresentam uma característica chamada de *autossimilaridade*, ou seja, uma parte do todo pode ser considerada uma imagem em escala reduzida do todo.[1] Tal característica é encontrada em certos tipos de terrenos naturais e em vários outros sistemas naturais. Como um exemplo, pensemos num pequeno pedaço de um planalto ou de uma planície. Tais pedaços possuem propriedades que são encontradas ao longo de todo o planalto ou planície.

Como terrenos naturais possuem esse comportamento, nós iremos querer replicá-lo para gerar um terreno que pareça natural. Um terreno gerado de forma a ter tais características auto-similares é chamado de *paisagem fractal*.

É bom notar, porém, que *alguns* terrenos naturais possuem essa característica, ou seja, apenas um subconjunto de todos os terrenos naturais possíveis são gerados pelo algoritmo. O algoritmo implementado não simula, por exemplo, mudanças "bruscas" de terreno, como as causadas, por exemplo, por falhas geológicas. Ainda assim, o resultado do algoritmo é uma aproximação razoável do terreno natural e assim pode ser usada satisfatoriamente num jogo eletrônico e até mesmo outras aplicações.

[1] O bump mapping é uma técnica usada para adicionar "rugosidade" a objetos, normalmente se utilizando dos pixels de um mapa de altura para perturbar as normais da superfície do objeto.

1.4 Densidade espectral

O espectro de uma série temporal, de um sinal ou de um ruído qualquer é uma função real positiva de uma frequência associada a um processo estocástico, ou uma função determinística do tempo, que possua dimensão de energia ou força por Hertz.

A *densidade espectral* então, nos diz quanto de "energia" uma série temporal ou sinal carrega em relação à sua frequência. Assim podemos decompor um processo estocástico em diferentes frequências para melhor análise desta. Tal conceito será necessário para conectar os conceitos de ruído rosa e o processo estocástico de movimento browniano fracionário, que serão logo introduzidos.

1.5 Ruído

Ao falarmos de *ruído*, nos referimos normalmente a padrões aleatórios que interferem na qualidade de sinais de áudio, vídeo ou imagem. Usaremos na geração de mapas de altura algumas funções geradoras de ruído, ou seja, funções cuja saída é um valor aleatório tais como a função

$$Noise(x, y) = z$$

que recebe um ponto $(x, y) \in \mathbb{R}^2$ de entrada e devolve um número real como resposta. Poderíamos implementar uma função que recebe um ponto em $\mathbb{R}^n$ com n qualquer, mas iremos nos ater a funções com domínio em $\mathbb{R}^2$ neste trabalho.

As funções de ruído aqui implementadas, porém, são funções geradoras de ruído coerente. Por ruído coerente, dizemos uma função de ruído que possui três características:

- É pseudo-aleatória, isto é, para um mesmo valor de entrada, temos sempre o mesmo valor de saída.
- Com uma pequena variação no valor de entrada, temos pequenas variações no valor de saída.
- Com uma grande variação no valor de entrada, a variação no valor de saída é desconhecida.

A grande diferença de funções geradores de ruído coerente para as outras funções geradores de ruído é o segundo item. Como essas funções serão usadas na geração do terreno, é possível ver uma analogia com o terreno de saída e essa funções: assim como pontos em $\mathbb{R}^2$ próximos dados como entrada para as funções de ruído coerente possuem uma pequena diferença no seu valor de saída, dois pontos próximos do terreno gerado terão uma diferença pequena de altura.

Particularmente, estamos interessados em um tipo de ruído chamado de *ruído rosa*, um ruído no qual a densidade espectral é inversamente proporcional à frequência. Sendo assim, cada oitava carrega consigo uma quantidade igual de energia. O ruído rosa pode ser definido mais rigorosamente como qualquer ruído com a densidade espectral de forma

$$S(f) = \frac{1}{f^\alpha}$$

com $0 < \alpha < 2$ e normalmente próximo de 1. Iremos gerar um ruído dessa forma através da somatória de várias instâncias das funções de ruído coerente que foram implementadas. O ruído rosa também pode ser chamado de *ruído fractal*. Este tipo de ruído está ligado ao processo estocástico de movimento browniano fracionário que será explicado na próxima seção. Tal relação foi estudada por Mandelbrot e está documentada em um de seus artigos[2]. É por isso que queremos gerar tal ruído rosa, para gerar algo que se aproxime deste processo estocástico.

1.6 Movimento browniano fracionário

É um processo estocástico que generaliza o processo estocástico de movimento browniano. É um processo gaussiano, autossimilar e com tempo contínuo $B_H(t)$ em $[0, T]$, começando do valor 0, com expectativa 0 e a seguinte função de covariância:

$$Cov(B_H(t), B_H(s)) = \frac{1}{2}(|t|^{2H} + |s|^{2H} - |t - s|^{2H})$$

O valor H é chamado de expoente de Hurst e dita quão "liso" ou "rugoso" é o terreno resultante. Apesar do algoritmo implementado ter o mesmo nome deste processo, ele é apenas uma aproximação da saída deste processo. É bom notar também a existência da propriedade de auto-similaridade, que já foi dita existir nos terrenos que queremos gerar.

O algoritmo descrito neste trabalho é dito uma "aproximação" do processo estocástico de movimento browniano fracionário no sentido de que a densidade espectral da saída obtida é condizente com a densidade espectral deste processo estocástico.

1.7 OpenGL

O *OpenGL* é uma API (Application Programming Interface ou Interface de Programação de Aplicativos) para a renderização de gráficos 2D ou 3D. Ela foi usada para mostrar um modelo 3D do mapa de altura gerado.

Capítulo 2

Geração de mapas de altura

Como o objetivo do trabalho é o de gerar conteúdo que será usado por um usuário final, praticamente todas as partes do trabalho envolveram a escrita de código. O código resultante pode ser encontrado num repositório em meu Github:

https://github.com/gustavoteixeira/tcc

O trabalho se divide em duas partes:

- A geração de mapas de altura que servirão como o terreno do mapa
- A análise do mapa de altura gerado para a criação do mapa de jogo final

A geração dos mapas de altura, por sua vez, é dividida em dois passos:

- A implementação de funções geradoras de ruído coerente
- A implementação do algoritmo chamado de Fractional Brownian Motion

Este capítulo dedica-se a primeira parte, ou seja, à geração dos mapas de altura em si. No próximo capítulo eu lidarei com a criação do mapa final.

Foram implementados dois algoritmos de geração de ruído coerente, o *Value Noise* e o *Perlin Noise*. Esses algoritmos possuem generalizações N-dimensionais, mas as versões implementadas aqui são as versões 2D.

2.1 Value Noise

A função é definida numa grade 2D com valores $(x, y) \in \mathbb{N}^2$ pertencendo à grade e valores $(x, y) \in \mathbb{R}^2$ estando entre valores na grade. Para o ponto de entrada nós inicialmente encontramos os quatro pontos da da grade mais próximos do valor de entrada. Chamaremos esses pontos de (x_0, y_0), (x_0, y_1), (x_1, y_0) e (x_1, y_1).

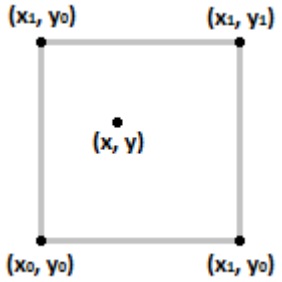

Figura 2.1: *Ponto de entrada e seus quatro pontos mais próximos pertencentes à grade*

Cada um dos pontos da grade possui um valor associado gerado aleatoriamente. Nós temos então a função

$random(x, y) = valor$

que recebe um ponto na grade e retorna um número inteiro aleatório numa distribuição quase uniforme com valor entre 0 e 255. Para encontrar o valor de saída, calculamos a contribuição de cada um desses valores aleatórios para o valor de saída. Fazemos isso calculando a interpolação desses valores. A interpolação é uma função

$interpolar(v1, v2, t) = v_{random}$

que recebe três valores: os dois valores que serão interpolados e um valor entre 0 e 1 que diz qual é a contribuição de cada um dos dois valores para o valor final. Sendo o valor de entrada um ponto $(x, y) \in \mathbb{R}^2$, este terceiro valor é a parte fracionária de x ou de y deste ponto, dependendo dos valores a serem interpolados.

Dado o ponto de entrada $(x, y) \in \mathbb{R}^2$, definimos x_{grid} como a parte fracionária de x e y_{grid} como a parte fracionária de y. Fazemos as seguintes interpolações:

$interpolar(random(x_0, y_0), random(x_1, y_0), x_{grid}) = v1$

$interpolar(random(x_0, y_0), random(x_1, y_0), x_{grid}) = v2$

$interpolar(v1, v2, y_{grid}) = v_{final}$

ou seja, interpolamos (x_0, y_0) e (x_1, y_0) usando x_{grid} , depois (x_0, y_1) e (x_1, y_1) também usando x_{grid}. Temos agora dois valores que serão também interpolados usando y_{grid}. Este é o valor de retorno.

A implementação do algoritmo e da função de interpolação usada se encontram abaixo:

```
1  double ValueNoise(double x, double y) {
2      int x_int = (int) floor(x);
3      int y_int = (int) floor(y);
4      double grid_x = x - x_int;
5      double grid_y = y - y_int;
6
7      double g1 = random_values[ x_int     ][ y_int];
8      double g2 = random_values[(x_int + 1)][ y_int];
9      double g3 = random_values[ x_int     ][(y_int + 1)];
10     double g4 = random_values[(x_int + 1)][(y_int + 1)];
11
12     double interpolation1 = cosineInterpolation(g1, g2, grid_x);
13     double interpolation2 = cosineInterpolation(g3, g4, grid_x);
14     return cosineInterpolation(interpolation1, interpolation2, grid_y);
15 }
16
17 // Cosine interpolation function
18 double cosineInterpolation(double v1, double v2, double grid_value) {
19     double value = (1.0f - cos( grid_value * PI ))/2;
20     return(v1 * (1.0f - value) + v2 * value);
21 }
```

2.2 Perlin Noise

O algoritmo *Perlin Noise* é um algoritmo de geração de ruído desenvolvido por Ken Perlin. Ele possui algumas similaridades com o algoritmo Value Noise.

A função é definida numa grade 2D com valores $(x, y) \in \mathbb{N}^2$ pertencendo à grade e valores $(x, y) \in \mathbb{R}^2$ estando entre valores na grade. Para o ponto de entrada nós inicialmente encontramos os quatro pontos da grade mais próximos do valor de entrada. Chamaremos esses pontos de (x_0, y_0), (x_0, y_1), (x_1, y_0) e (x_1, y_1).

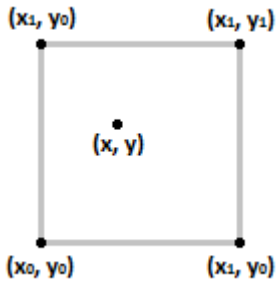

Figura 2.2: *Ponto de entrada e seus quatro pontos mais próximos pertencentes à grade*

Agora definimos a função

$gradiente(x, y) = (g_x, g_y)$

que recebe um valor na grade e retorna um vetor 2D unitário. Os valores desses vetores são gerados de forma pseudo-aleatória para cada instância do gerador de ruído.

Também geramos um vetor a partir da distância de cada um desses pontos na grade até o ponto de entrada usando uma subtração simples.

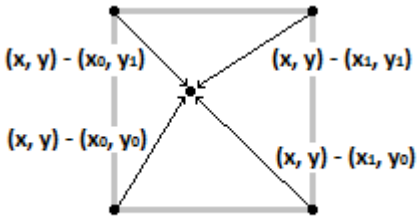

Figura 2.3: *Vetores calculados a partir das subtrações.*

Para calcular o valor de saída precisamos da contribuição de cada um dos gradientes dos valores da grade para o valor final. Para cada ponto na grade calculamos o produto escalar entre o gradiente neste ponto e a distância entre este ponto e o ponto de entrada:

$v1 = gradiente(x_0, y_0) \cdot ((x, y) - (x_0, y_0))$
$v2 = gradiente(x_1, y_0) \cdot ((x, y) - (x_1, y_0))$
$v3 = gradiente(x_0, y_1) \cdot ((x, y) - (x_0, y_1))$
$v4 = gradiente(x_1, y_1) \cdot ((x, y) - (x_1, y_1))$

Para encontrar o valor de saída, calculamos a contribuição de cada um desses valores aleatórios para o valor de saída. Fazemos isso calculando a interpolação desses valores. A interpolação é uma função

$interpolar(v1, v2, t) = v_{random}$

que recebe três valores: os dois valores que serão interpolados e um valor entre 0 e 1 que diz qual é a contribuição de cada um dos dois valores para o valor final. Sendo o valor de entrada um ponto $(x, y) \in \mathbb{R}^2$, este terceiro valor é a parte fracionária de x ou de y deste ponto, dependendo dos valores a serem interpolados.

Dado o ponto de entrada $(x, y) \in \mathbb{R}^2$, definimos x_{grid} como a parte fracionária de x e y_{grid} como a parte fracionária de y. Fazemos as seguintes interpolações:

$interpolar(v1, v2, x_{grid}) = x1$

$interpolar(v3, v4, x_{grid}) = x2$

$interpolar(x1, x2, y_{grid}) = v_{final}$

ou seja, interpolamos $v1$ e $v2$ usando x_{grid}, depois $v3$ e $v4$ também usando x_{grid}. Temos agora dois valores que serão também interpolados usando y_{grid}. Este é o valor de retorno.

A implementação do algoritmo e da função de interpolação usada descrita por Ken Perlin[3]:

```
1  double PerlinNoise(double x, double y) {
2      int x_int = (int) floor(x);
3      int y_int = (int) floor(y);
4      double grid_x = x - x_int;
5      double grid_y = y - y_int;
6
7      vector2D g1 = gradient[(permutation[( x_int   +permutation[ y_int   %256])%256])%4];
8      vector2D g2 = gradient[(permutation[((x_int+1)+permutation[ y_int   %256])%256])%4];
9      vector2D g3 = gradient[(permutation[( x_int   +permutation[(y_int+1)%256])%256])%4];
10     vector2D g4 = gradient[(permutation[((x_int+1)+permutation[(y_int+1)%256])%256])%4];
11
12     double v1 = dotproduct( g1 , grid_x    , grid_y     );
13     double v2 = dotproduct( g2 , grid_x - 1, grid_y     );
14     double v3 = dotproduct( g3 , grid_x    , grid_y - 1 );
15     double v4 = dotproduct( g4 , grid_x - 1, grid_y - 1 );
16
17     double x1 = interpolate(v1, v2, grid_x);
18     double x2 = interpolate(v3, v4, grid_x);
19     return interpolate(x1, x2, grid_y);
20 }
21
22 // Linear Interpolation using Ken Perlins fade function
23 // using the 6t^5 - 15t^4 + 10t^3 polynomial
24 double interpolate(double v1, double v2, double t) {
25    t =  t*t*t*(t*(t*6-15)+10);
26    return (1.0f-t)*v1 + t * v2;
27 }
```

Um detalhe da implementação é que ao invés de gerarmos um gradiente aleatório para cada ponto na grade, são usados apenas quatro vetores 2D como gradientes: são eles $(0,1)$, $(1,0)$, $(-1,0)$ e $(0,-1)$.

Para encontrar o valor aleatório de um ponto na grade usamos um vetor de permutações de tamanho 256. Ou seja, para um ponto (x,y) pertencente à grade, o valor de seu gradiente dependerá unicamente dos valores x_{int}, a parte inteira de x e de y_{int}, a parte inteira de y.

2.3 Fractional Brownian Motion

Fractional Brownian Motion refere-se aqui ao algoritmo que utiliza a função de ruído para a geração do mapa de altura.

O algoritmo é simples: para uma função de ruído chamada de `noise` qualquer, tal como *Value Noise* ou *Perlin Noise*, fazemos a seguinte soma:

$$fbm(x,y) = \sum_{i=1}^{n} a.g^i \,.\, Noise(f \,.\, l^i \,.\, x, f \,.\, l^i \,.\, y)$$

ou seja, somamos múltiplas instâncias da função geradora de ruído com diferentes escalas. Temos aqui cinco variáveis:

- n é o número de oitavas a serem somadas. Este valor especifica quantas iterações da função de ruído escolhida são usadas para calcular o resultado final.
- f é a frequência. Este valor que determina quantos valores aleatórios são usados para o cálculo do mapa de altura resultante. Quanto maior a frequência, mais pontos são usados para calcular o valor de saída.
- a é a amplitude. Esta variável está diretamente relacionada com a altura do mapa de altura resultante.
- g é o ganho. É o valor que especifica o crescimento ou decrescimento da frequência a medida que o valor de n aumenta.
- l é a lacunaridade. é o valor que especifica o crescimento ou decrescimento da amplitude a medida que o valor de n aumenta.

Como o algoritmo se utiliza de ruído, usa-se aqui uma terminologia comum em processamento de sinais, com nomes tais como frequência e oitava.

É bom notar que os valores g e l devem ser bem escolhidos de modo que a saída se assemelhe de fato a terreno natural. Como já explicado, esse algoritmo é uma "aproximação" do processo estocástico de movimento browniano fracionário, cujo valor H depende dos valores de g e l. Como já mencionado, $0 < H < 1$, e a aparência do terreno gerado é altamente dependente deste valor. Como calculamos H a partir de g e l não será explicado aqui, mas para melhores resultados, recomenda-se valores de g e l tal que $g * l$ seja próximo de 1, com g e l positivos tal que $g < 1$ e $l > 1$.

A implementação do algoritmo é a seguinte:

```
1  double fbm(int x, int y, double amplitude) {
2      double total = 0.0f;
3      double height_frequency = 1.0f/ height, width_frequency = 1.0f/ width;
4      for (int i = 0; i < octaves; ++i) {
5          total += Noise((double)x * width_frequency, (double)y * height_frequency) * amplitude
               ;
6          width_frequency *= lacunarity; height_frequency *= lacunarity;
7          amplitude *= gain;
8      }
9      return total;
10 }
```

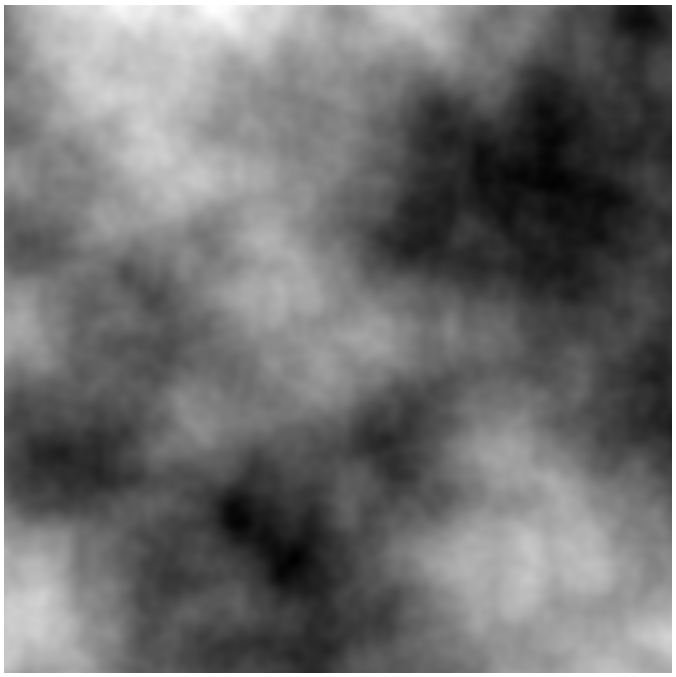

Figura 2.4: *Mapa de altura gerado usando Fractional Brownian Motion com função de ruído Perlin Noise*

Capítulo 3

Processador de mapas de altura

No capítulo anterior eu detalhei como gerar um mapa de altura, e este lidará com o uso de tais mapas de altura para criar um mapa de jogo. Logo teremos como entrada para o processador descrito neste capítulo um mapa de altura e teremos como saída um mapa de jogo.

3.1 Parâmetros de entrada

O mapa de jogo gerado depende de alguns valores de entrada que são guardados num arquivo de especificações cujo nome é passado via linha de comando. Os valores são os seguintes:

- `scale` é um valor que multiplica todos os pontos do mapa de altura de entrada, resultando no seu achatamento ou ampliação.
- `cutoff` é o nível do mar. Valores abaixo desta altura são considerados mar. Se esse valor não for definido, assumimos que o mapa de jogo gerado não possui mar.
- `rivers` determina a quantidade de rios gerados no mapa de jogo. Se não for definido, assumimos zero
- `biomes` é uma lista dos biomas que podem existir no mapa de jogo gerado.

Exemplo de um arquivo de especificações de exemplo:

```
scale 0.5
cutoff 0.4
rivers 2
biomes 3 Forest grassland beach
```

Por razões técnicas precisamos colocar a quantidade de biomas após a definição da variável `biomes`.

3.2 Geração do mapa de jogo

A geração do mapa de jogo é dividida nos seguintes passos:

1. **Multiplicar o mapa o parâmetro de escala**

   Neste passo, simplesmente multiplicamos cada ponto do mapa pela variável `scale` que é recebida como parâmetro de entrada. Este valor irá "achatar" ou "esticar" o mapa final. Se esse parâmetros não for especificado, o mapa não será alterado.

2. **Aplicar no mapa a linha de nível do mar**

   Neste passo, usaremos o parâmetro `cutoff` de entrada para calcular os pontos do mapa que são mar. Nós calculamos o valor mais alto do mapa e o multiplicamos pelo valor de `cutoff`, e qualquer ponto menor que este será considerado como parte do mar.

3. **Cálculo de valores importantes dos pontos do mapa**

   Neste passo, para cada ponto do mapa calculamos alguns valores que serão usados na divisão de biomas e para a visualização 3D, como a distância até o mar, vetor normal e ângulo de inclinação.

4. **Divisão do mapa em biomas**

   A divisão do mapa em biomas segue uma regra simples: para cada um dos biomas, chamamos uma função associada que retorna um valor entre 0 e 1. O ponto do mapa ficará associado ao bioma cuja função associada retornar o maior valor. A função usa parâmetros como a distância até o mar, altura e mais para o cálculo do valor. Um ponto perto do mar, por exemplo, tem um valor mais próximo de 1 do que um ponto longe do mar, pontos mais altos possuem valores mais próximos de 1 para biomas gelados e assim sucessivamente. Como critério de desempate usamos a regra de que o bioma declarado antes toma precedência sobre os outros.

5. **Geração de conteúdo**

   O único conteúdo gerado nesta primeira versão da ferramenta são rios. A especificação do algoritmo que gera tais rios é a seguinte:

   (a) Escolhemos um ponto (x, y) aleatório a partir do qual será gerado o rio.

   (b) Escolhemos um novo ponto que pertença ao rio (x_1, y_1) tal que $altura(x_1, y_1) < altura(x, y)$ e esse ponto seja o menor dentre os vizinhos de (x, y). Repetimos esse processo até que não haja nenhum ponto (x_n, y_n) vizinho de (x, y) tal que $altura(x_n, y_n) < altura(x, y)$.

   (c) A partir do ponto inicial escolhemos um novo ponto que pertença ao rio (x_1, y_1) tal que $altura(x_1, y_1) > altura(x, y)$ e esse ponto seja o maior dentre os vizinhos de (x, y). Repetimos esse processo até que não haja nenhum ponto (x_n, y_n) vizinho de (x, y) tal que $altura(x_n, y_n) > altura(x, y)$.

   A ideia aqui é criar um rio a partir de um ponto qualquer com sua origem no máximo local encontrado a partir do ponto de entrada e que desagua no mínimo local também encontrado a partir do ponto de entrada.

6. **Geração de assentamentos humanos**

   Infelizmente, eu não tive tempo de implementar a geração de assentamentos humanos. Este foi o único item não implementado.

3.3 Especificação do arquivo de saída

Aqui está um exemplo de arquivo de saída:

```
1000 1000
forest
grassland
beach

60.4606 1 0.939747 713 -0.807406 0.00205789 0.589992 ... 31.4191 2 0.630648 24 -0.399546
    0.433673 0.807645
    .
    .
    .
91.195 1 0.581825 362 -0.0682601 0.545294 0.835461 ... 84.7794 1 0.562717 351 -0.25395
    0.469167 0.845808
```

Na primeira linha temos a largura e a altura do mapa gerado.

Após isso temos uma quantidade indeterminada de linhas cada uma com o nome de um bioma. Consideramos que cada bioma possui um valor inteiro implícito associado de acordo com a linha em que está. Ou seja, na linha dois temos o bioma 1, na linha três temos o bioma 2 e assim sucessivamente. Quando a lista de biomas acabar teremos uma linha vazia.

Após isso temos tantas linhas quanto for o tamanho da altura do mapa. Em cada linha teremos largura * 7 valores. Os 7 primeiros valores indicam os dados do primeiro ponto do mapa nesta linha, os próximos 7 valores indicam os dados do segundo ponto desta linha e assim sucessivamente. Cada um dos 7 valores especificam as seguintes informações do mapa:

1. A altura deste ponto.
2. O bioma associado à este ponto.
3. O ângulo de inclinação neste ponto entre -1 e 1.
4. A distância deste ponto até o mar.
5. A componente x do vetor normal neste ponto do mapa.
6. A componente y do vetor normal neste ponto do mapa.
7. A componente z do vetor normal neste ponto do mapa.

Capítulo 4

Visualizando o mapa gerado

Para ajudar a visualizar o mapa gerado foi criado um programa escrito em C++ e utilizando OpenGL para mostrar o mapa de jogo resultante e os mapas de altura em 3D. Este código também está disponível no repositório junto com os outros algoritmos. Por razões de eficiência, o visualizador usa diretamente os valores do processador ao invés de lê-los do arquivo de saída.

Cada pixel do mapa de altura determina a posição nos eixos `x`, `y` e `z` do vértice. A cor depende do tipo de bioma, ou seja, uma praia possui cor bege, o mar é azul e as planícies são verdes. Como cada ponto possui uma normal, estas são usadas para iluminar apropriadamente o modelo 3D.

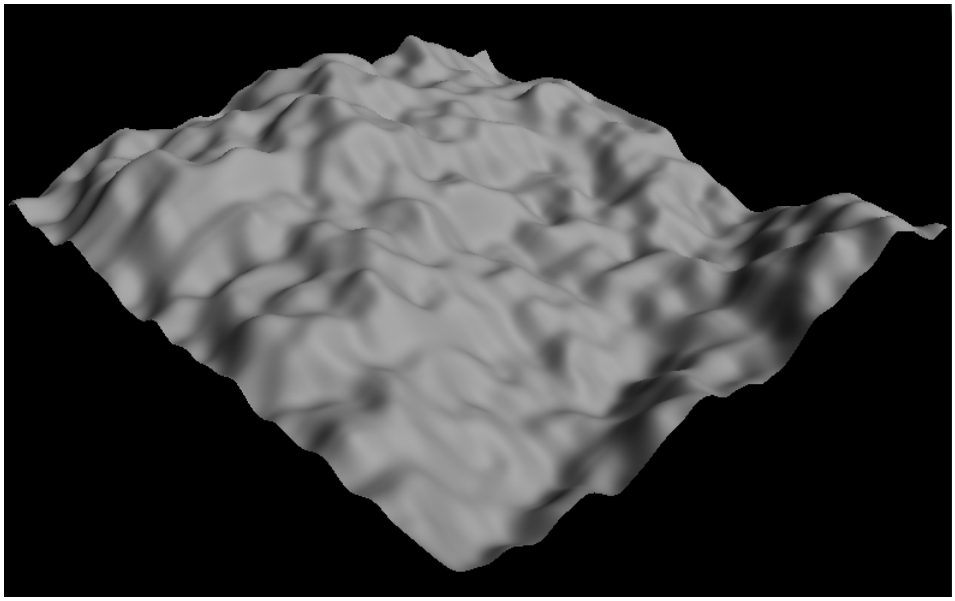

Figura 4.1: *Mapa de altura mostrado pelo visualizador 3D*

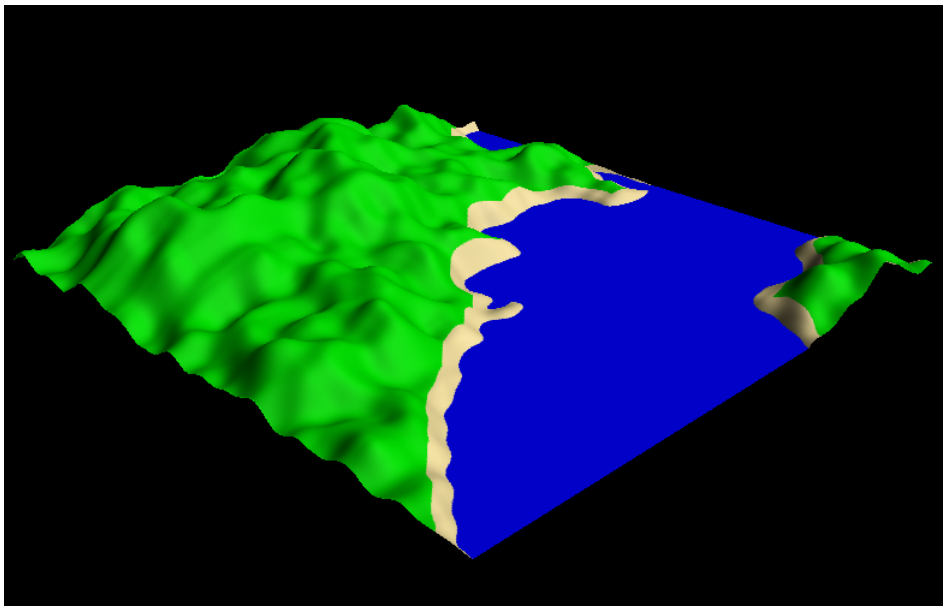

Figura 4.2: *Mapa de jogo mostrado pelo visualizador 3D*

Capítulo 5

Resultados obtidos

O conjunto de algoritmos resultante é bom e pode já ser usado num jogo caso algum usuário quisesse. Mesmo assim, seria interessante que eu melhorasse esses algoritmos para expandir a diversidade de mapas gerados e facilitar o seu uso por um usuário final.

Na parte de geração de terreno há áreas importantes para melhoria, tais como terrenos mais variados usando mais técnicas de geração de relevo. A geração atual não inclui variações bruscas de terreno como falhas geológicas, que poderiam ser desejadas por um usuário final.

Na parte de análise do mapa, a geração de biomas precisa ser melhorada com mais descrições de biomas, já que apenas 3 biomas possíveis são gerados. A geração de conteúdo também precisa ser populada com algo além de rios, possibilitando gerar, por exemplo, lagos ou as árvores de uma floresta. A saída da análise poderia ser mais eficiente para gerar arquivos menores de saída para diminuir tempos de escrita e leitura.

Pretendo também futuramente fazer ao menos um protótipo de jogo que use esses algoritmos para a geração do mapa de jogo, utilizando esta experiência para facilitar o uso por um usuário final e para tentar perceber o que seria mais imediatamente necessário na geração de mapas.

Parte II

Parte Subjetiva

Capítulo 6

Disciplinas Relevantes

- MAC0122 - Princípios de Desenvolvimento de Algoritmos e MAC0323 - Estruturas de Dados
  Essas disciplinas me ensinaram a entender melhor como funciona a linguagem C, o que foi fundamental para a implementação dos algoritmos.
- MAE0228 - Noções de Probabilidade e Processos Estocásticos
  Uma parte importante do trabalho foi entender o uso do processo estocástico de movimento browniano fracionário para a geração de terrenos e a sua aproximação usando ruído. Mesmo que eu não tenha entendido completamente o processo, esta matéria me ajudou a entender melhor as ideias, e mesmo não tendo estudado esta matéria por um bom tempo, ter feito ela me ajudou a lembrar bastante mais rapidamente os conceitos por trás do processo estocástico estudado.
- MAC0420 - Introdução a Computação Gráfica
  Essa disciplina me ensinou a usar bem o OpenGL, o que foi necessário para implementar o visualizador de mapas de altura.

Capítulo 7

Desafios e Frustrações

Um problema inesperado ao tentar usar os de geração de mapas de altura em sistemas Windows é que o gerador de número aleatórios do Windows, mesmo quando usando o MinGW, era muito ruim, o que tornou necessário utilizar um gerador de números aleatórios não padrão. Como solução inicial usei um código simples tirado da Wikipédia implementando um *Mersenne Twister* para a geração de números aleatórios, mas por sorte o C++11 disponibiliza uma boa biblioteca de geração de números aleatórios, o que solucionou o problema. Vale notar que tal problema não existe no Linux, sendo que aparentemente o GCC não possui o problema com a geração de números aleatórios que existe no Windows.

Além disso, boa parte dos fundamentos matemáticos que explicam como a geração dos mapas de altura funciona são bastante complexos. Apesar de ter procurado bastante, em nenhum lugar encontrei uma explicação satisfatória do algoritmo de Fractional Brownian Motion, já que a maioria dos recursos que encontrei simplesmente se destinavam apenas a explicar os passos do algoritmo e não a explicar como ele funcionava detalhadamente. Por causa disso, isso não consegui entender satisfatoriamente o porquê do algoritmo de Fractional Brownian Motion funcionar.

No fim, decidi me ater simplesmente a implementar o algoritmo descrito e experimentar utilizando a implementação dos algoritmos para estuda a relação entre ruído e o processo estocástico de movimento browniano fracionário, o que é melhor explicado, mas ainda assim bastante complexo e não tão bem detalhado.

Quanto ao processador dos mapas de altura, não existem muitos recursos que expliquem algo como isso. A maioria dos recursos detalhando a geração procedimental de mundos se atem a explicar a geração de mapas de altura sem falar muito do clima e objetos que compõem o mundo. Sendo assim, tive que fazer algo que fizesse sentido no mundo real, usando parâmetros como altura, distância até o mar e outros para a divisão do mundo em biomas. Ainda assim, julgo que o resultado obtido é bom o suficiente para ser usado por mim e até por outros desenvolvedores.

Capítulo 8

Conclusão

Este trabalho foi bastante interessante por estar diretamente ligado a uma área de interesse minha, que é a de jogos eletrônicos. Poder fazer algo que outras pessoas possam usar é muito interessante, apesar de achar que os algoritmos finais necessitem de mais melhorias antes que um possível usuário pense em usá-los no seu jogo.

A parte de implementação dos algoritmos foi bastante produtiva, fazendo com que eu procurasse sobre coisas como a geração de números aleatórios e buscar entender melhor como funciona o OpenGL. Também aprendi sobre algoritmos como o Perlin Noise, que também são úteis em outras partes de computação gráfica, como a geração procedimental de texturas.

Uma parte extremamente frustrante, porém, foi não conseguir entender completamente os fundamentos matemáticos por trás da computação, me relegando apenas a entender parte deles, enquanto outra parte foi meramente implementada usando a descrição dos algoritmos mas sem compreendê-los de fato.

Referências Bibliográficas

[1] B. Mandelbrot, "How long is the coast of britain? statistical self-similarity and fractional dimension," *Science*, vol. 156, no. 3775, pp. 636–638, 1967. 3

[2] B. B. Mandelbrot and J. W. V. Ness, "Fractional brownian motions, fractional noises and applications," *SIAM Review*, vol. 10, no. 4, pp. 422 – 437, 1968. 4

[3] K. Perlin, "Improving noise," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2002*, vol. 21, no. 3, pp. 681–682, 2002. 10