

**ATENÇÃO:** aqui constam somente as páginas que tinham alguma anotação na revisão.

# GERAÇÃO PROCEDURAL DE TERRENOS VIRTUAIS COM APARÊNCIA NATURAL UTILIZANDO GPU

**Alex Seródio Gonçalves, Dalton Solano dos Reis – Orientador**

Curso de Bacharel em Ciência da Computação

Departamento de Sistemas e Computação

Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brasil

alexserodio@furb.br, dalton@furb.br

**Resumo:** Este artigo apresenta o desenvolvimento de uma ferramenta para geração de terrenos virtuais com aparência natural utilizando o algoritmo Diamond-Square para gerar o terreno base e algoritmos de erosão térmica e hidráulica para melhorar o terreno, sendo todos executados em GPU com shaders. Os resultados alcançados foram avaliados através do tempo de execução dos algoritmos e qualidade dos terrenos gerados, analisada através das métricas *erosion score* e *lei de Benford* em comparação com terrenos reais. Tratando-se de performance a ferramenta se mostrou eficiente, com a execução em GPU durando um máximo de 391 milissegundos no caso da erosão hidráulica. Em relação a qualidade, os terrenos gerados conseguiram alcançar resultados próximos, mas não iguais aos terrenos reais analisados, sendo o algoritmo de erosão hidráulica o que mais se aproximou do esperado. As métricas utilizadas para a análise de qualidade se mostraram viáveis, porém estudos mais aprofundados se fazem necessários.

**Palavras-chave:** Terreno. Geração procedural. Diamond-Square. Erosão. Lei de Benford.

1  
não com as mesmas características dos terrenos

## 1 INTRODUÇÃO

A forma como cidades e sociedades são construídas ao longo da história está diretamente relacionada com o relevo das paisagens da região. Essas paisagens estão em constante mudança, sendo influenciadas frequentemente não apenas por forças naturais, mas também por forças econômicas e culturais (PLIENINGER *et al.*, 2015). Algumas dessas mudanças podem se mostrar complexas de serem analisadas e estudadas no mundo real. Nestes casos, simulações computacionais possibilitam a criação e validação de modelos que representam tais cenários, permitindo assim a execução de experimentos, análise dos resultados e validação de teorias em um ambiente controlado (HEERMANN, 1990, p. 8). Estas simulações podem ser utilizadas para analisar os processos naturais e sociais causadores de mudanças na paisagem, sendo necessário, porém, a utilização de cenários adequados à simulação.

Para garantir a utilização de terrenos naturais e possibilitar uma variedade de cenários diferentes para as simulações, é possível utilizar terrenos gerados de forma procedural. Este processo pode ser realizado através da utilização de modelos estocásticos e/ou modelos físicos. O primeiro permite a geração controlada de terrenos com características de relevo pseudoaleatórias (EBERT *et al.*, 2003). Já o segundo realiza a criação de terrenos através da simulação de eventos físicos presentes no mundo, como processos de erosão que modificam o relevo (MUSGRAVE; KOLB; MACE, 1989). No caso dos modelos físicos, os algoritmos exigem um poder de processamento considerável para atingir resultados realistas, tornando sua execução em tempo real impraticável em muitos casos (MEI; DECAUDIN; HU, 2007, p. 47). Este problema pode ser contornado com a utilização da Graphic Processing Unit (GPU) para processamento em paralelo, cujo uso se mostra viável principalmente conforme a quantidade de dados a serem processados aumenta (HANGÜN; EYECIOĞLU, 2017, p. 34). Desta forma é possível realizar o processamento dos algoritmos de erosão de forma eficiente e ainda preservar o nível de realismo dos terrenos (MEI; DECAUDIN; HU, 2007, p. 48).

2 Diante do apresentado, este trabalho propõe a disponibilização de uma ferramenta para geração procedural de terrenos virtuais com aparência natural utilizando modelos estocásticos e físicos executados em GPU. Destaca-se que o termo “aparência natural” refere-se à representação gráfica de características geologicamente fidedignas de terrenos reais. Os objetivos específicos do trabalho são: disponibilizar a ferramenta através de um executável desktop; disponibilizar a ferramenta através de um *plugin* para o motor gráfico Unity; analisar a performance dos algoritmos implementados comparando sua execução em CPU e GPU; e avaliar a qualidade dos terrenos gerados com base em terrenos reais. Tratando-se da análise de qualidade dos terrenos, serão utilizadas como métricas o *erosion score* proposto por Olsen (2004) para avaliar o nível de erosão de terrenos virtuais e a lei de Benford (BENFORD, 1938) que demonstra um padrão de distribuição de dígitos encontrado em conjuntos de dados estatísticos gerados naturalmente.

2  
este trabalho disponibiliza uma

## 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação dos assuntos abordados no trabalho e se divide em cinco seções. A seção 2.1 descreve a relação entre modelos estocásticos e geração procedural, apresentando o algoritmo Diamond-Square para geração de terrenos. A seção 2.2 descreve o conceito de modelos físicos e como podem ser aplicados na geração de terrenos naturais através de algoritmos de erosão térmica e hidráulica, cujo funcionamento também é explicado. A seção

2.3 aborda o processamento paralelo em GPU, algumas vantagens, cuidados a serem tomados e um exemplo de programação em GPU através de um *compute shader* na linguagem HLSL. A seção 2.4 apresenta duas métricas que podem ser utilizadas para avaliar a qualidade de terrenos gerados de forma procedural. Por fim, a seção 2.5 apresenta três trabalhos correlatos que utilizam algumas das técnicas mencionadas para gerar, modelar ou transformar terrenos virtuais.

## 2.1 MODELOS ESTOCÁSTICOS E GERAÇÃO PROCEDURAL

Tradicionalmente, objetos gráficos virtuais são modelados a partir de polígonos. Estes polígonos podem ser descritos matematicamente através de equações determinísticas, o que faz com que suas características sejam previsíveis e facilmente reproduzíveis. Porém, ao tentar modelar objetos naturais como rochas, nuvens e árvores utilizando tais equações, o resultado tende a ser objetos que não se assemelham totalmente com a realidade, justamente por possuírem características irregulares difíceis de serem reproduzidas por este método (FOURNIER; FUSSELL; CARPENTER, 1982, p. 371). Uma alternativa para modelar estes objetos é a utilização de modelos estocásticos. Enquanto em um modelo determinístico um mesmo conjunto de pontos submetido a uma mesma função gerará sempre o mesmo resultado, em um modelo estocástico um mesmo conjunto de pontos submetido a uma mesma função pode gerar resultados diferentes, variando de forma irregular ao longo do trajeto, o que possibilita resultados mais próximos à realidade ao modelar objetos naturais (FOURNIER; FUSSELL; CARPENTER, 1982, p. 372).

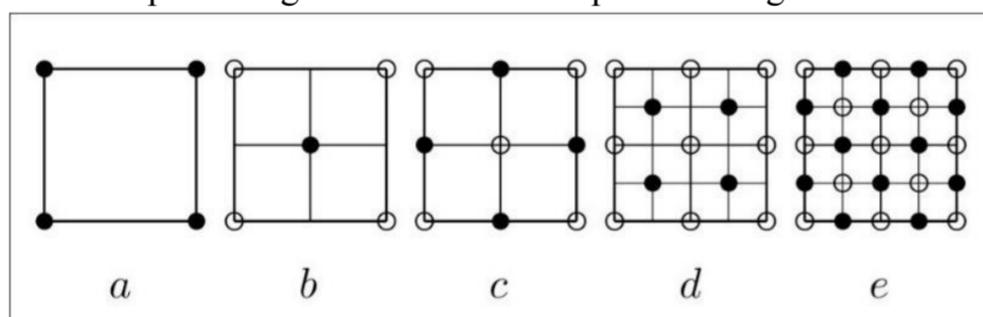
Estes resultados podem ser alcançados através do uso de técnicas procedurais que, segundo Ebert *et al.* (2003, p. 1, tradução nossa), "[...] são segmentos de código ou algoritmos que especificam alguma característica de um modelo ou efeito gerado por computador.". Isto permite que características como forma, altura e textura do objeto sejam determinadas automaticamente através da utilização de algoritmos e funções matemáticas, eliminando a necessidade de escanear um objeto real para reproduzi-lo virtualmente ou modelar o objeto virtual por completo (EBERT *et al.*, 2003, p. 1). Estas técnicas podem ser utilizadas para criação de vários tipos diferentes de objetos naturais, como rochas (PEYTAVIE *et al.*, 2009, p. 7-8), nuvens e estrelas (RODEN; PARBERRY, 2005), árvores (LONGAY *et al.*, 2012), rios (GÉNEVAUX *et al.*, 2013) e até mesmo cenários não naturais como cidades com estradas e prédios (KELLY; MCCABE, 2007). Tratando-se de terrenos, uma técnica de geração procedural que pode ser utilizada é o algoritmo Diamond-Square.

Diamond-Square é um algoritmo de *midpoint displacement* proposto por Fournier, Fussell e Carpenter (1982) como uma forma de gerar superfícies estocásticas parametrizáveis, ou seja, que possuem uma natureza estocástica ao invés de determinística, mas que mantenham as propriedades necessárias para representar objetos gráficos (FOURNIER; FUSSELL; CARPENTER, 1982, p. 380). Seu funcionamento é baseado na técnica *fractional Brownian motion* (fBm) proposta por Mandelbrot e Ness (1968) e consiste em calcular novos valores entre valores já conhecidos através da subdivisão recursiva da superfície. Pode-se dividir o algoritmo em duas etapas que se repetem a cada iteração, sendo elas:

- diamond step*: partindo do vértice central do quadrado, calcular o valor deste vértice como sendo a média dos quatro vizinhos na diagonal somada a um valor aleatório;
- square step*: partindo do vértice central do quadrado, calcular o valor dos vizinhos na horizontal e vertical como sendo a média de seus vizinhos nos mesmos eixos.

O algoritmo começa com um único quadrado que engloba toda a superfície. Antes da primeira iteração, deve-se definir valores aleatórios para cada canto da superfície. A cada iteração do algoritmo cada quadrado será subdividido em quatro, sendo que as etapas *diamond step* e *square step* devem ser aplicadas a cada um deles. O processo se repete recursivamente até que não seja mais possível subdividir a superfície (OLSEN, 2004, p. 2-3). A Figura 1 apresenta a execução do algoritmo ao longo de duas iterações, na qual o passo (a) representa a definição inicial dos valores aleatórios dos cantos, os passos (b) e (d) representam a etapa *diamond step* e os passos (c) e (e) a etapa *square step*.

Figura 1 - Etapas do algoritmo Diamond-Square ao longo de duas iterações



Fonte: Olsen (2004, p. 3).

O valor aleatório adicionado a cada novo vértice é conhecido no algoritmo fBm como expoente de Hölder ou simplesmente  $H$ , que controla a magnitude de variação de altura dos vértices da superfície, sendo  $0 < H < 1$ . Também é possível combiná-lo com um coeficiente utilizado para reduzir o valor de  $H$  a cada iteração, reduzindo assim a variação de altura a cada iteração (MANDELBROT; NESS, 1968). Por fim, para que o algoritmo funcione corretamente é importante que a superfície em questão tenha dimensões de  $N \times N$ , onde  $N$  equivale a  $2^n + 1$ , o que garante que a superfície tenha sempre um vértice central, permitindo que a subdivisão seja sempre simétrica.

Dúvida: porque não é mais possível dividir?

Limitação do número?

## 2.2 MODELOS FÍSICOS E ALGORITMOS DE EROSÃO

À primeira vista terrenos gerados através de técnicas procedurais como as descritas na seção 2.1 parecem convincentemente naturais. Porém, uma análise mais meticulosa revelaria a ausência de certas características presentes em terrenos reais. Um exemplo ocorre ao comparar áreas altas e baixas de uma mesma montanha. Em um terreno real, as partes inferiores da montanha estariam cobertas por resíduos provenientes do topo que se desprendem de áreas íngremes e deslizam até áreas planas ao longo do tempo. Isto não ocorre em terrenos gerados apenas através de técnicas procedurais, visto que tais técnicas não levam processos físicos em consideração (MUSGRAVE; KOLB; MACE, 1989, p. 41).

Modelos físicos são utilizados para reproduzir fenômenos físicos computacionalmente, a fim de produzir distribuições espaciais e temporais detalhadas do fenômeno em questão (LONG; HE, 2009, p. 1). Isto é feito através da implementação de algoritmos que simulam processos físicos reais ao longo de um determinado tempo. Além de geração de terrenos, modelos físicos podem ser utilizados em diversas outras áreas, como simulação de fluídos (MÜLLER; CHARYPAR; GROSS, 2003), nuvens (HARRIS *et al.*, 2003), neve (GOSWAMI; MARKOWICZ; HASSAN, 2019) e fumaça (ISHIDA; ANDO; MORISHIMA, 2019). Estes modelos geralmente não trabalham apenas com a criação de algum objeto, cenário ou fenômeno, mas sim com a simulação de seus comportamentos ao serem submetidos aos processos ao longo de um determinado tempo. No caso de terrenos, modelos físicos são normalmente utilizados para simular processos de erosão, como vazão de água, chuva, choques térmicos e ventos, sendo que os mais utilizados tendem a ser os de erosão térmica e hidráulica por causarem mudanças mais impactantes no terreno (JÁKÓ; TÓTH, 2011, p. 57).

O uso dos algoritmos de erosão térmica e hidráulica na criação de terrenos virtuais ganhou notoriedade com o trabalho de Musgrave, Kolb e Mace (1989), sendo amplamente aceitos desde então. Seu objetivo consiste em simular alguns dos processos de erosão aos quais terrenos reais são submetidos constantemente a fim de reproduzir estas características em terrenos virtuais. Ambos focam na transformação do terreno através da movimentação de sedimentos que se desprendem do relevo e são carregados para outras regiões. O algoritmo de erosão térmica trata de remover sedimentos de áreas íngremes de montanhas e depositá-los em áreas planas como os pés da montanha. Já o algoritmo de erosão hidráulica simula os processos de chuva e escoamento de água pelo terreno, no qual sedimentos são carregados e depositados em diferentes localizações do terreno dependendo do fluxo e quantidade de água (MUSGRAVE; KOLB; MACE, 1986, p. 3).

Como a erosão térmica atua apenas em áreas íngremes, é necessário definir a partir de qual inclinação o terreno sofrerá os efeitos de erosão. Este limite é conhecido como *talus*, sendo que áreas com inclinação maior que *talus* terão seu relevo transformado. A inclinação de um ponto neste caso é calculada simplesmente pela diferença de altura entre a posição atual e seus vizinhos mais próximos. A lógica para o transporte de sedimentos resume-se basicamente à equação

$$s = c(d_{max} - talus) \frac{d_i}{d_{total}} \quad (1)$$

onde  $c$  é uma constante que controla a quantidade de sedimento a ser transferido,  $d_i$  corresponde à diferença de altura entre a posição atual e o vizinho  $i$ ,  $d_{max}$  é a maior diferença de altura entre a posição atual e todos os vizinhos e  $d_{total}$  é a diferença total de altura entre a posição atual e todos os vizinhos, sendo que para o cálculo de  $d_{total}$  e  $d_{max}$  deve-se considerar apenas as inclinações que ultrapassem o limite *talus* (OLSEN, 2004, p. 6). Esta equação deve ser aplicada a todos os vizinhos cuja inclinação ultrapasse o limite *talus*, sendo que o valor  $s$  calculado deve ser subtraído da posição atual e somado ao vizinho em questão.

O processo de erosão hidráulica proposto por Beneš e Forsbach (2002) utiliza como base para suas equações três matrizes, uma  $h$  que representa o terreno, outra  $w$  para representar a água e outra  $m$  que representa os sedimentos. O algoritmo é dividido em quatro etapas, sendo elas:

- aparecimento de água da chuva;
- água da chuva dissolve a superfície do terreno transformando-a em sedimento;
- água e sedimento são transportados pelo fluxo de água;
- sedimento é depositado em outro local após evaporação da água.

Por mais que as quatro etapas apresentadas por Beneš e Forsbach (2002) descrevam o processo corretamente, algumas implementações variam de sua proposta em certos pontos, como é o caso das versões de Olsen (2004), Mei, Decaudin e Hu (2007), Long e He (2009), Jákó e Tóth (2011) e Diegoli Neto (2017). Neste trabalho optou-se por utilizar como base a implementação feita por Diegoli Neto (2017), que realiza as mesmas etapas sem utilizar uma matriz auxiliar para representar os sedimentos, mas ainda utilizando uma matriz  $w$  para representar a água, a qual inicia com os mesmos valores de altura que a matriz de terreno  $h$ . A seguir serão apresentadas equações elaboradas pelo autor que descrevem os algoritmos implementados por Diegoli Neto (2017) para atender as quatro etapas descritas acima.

O funcionamento da etapa a) é descrito pela Equação 2, na qual um valor constante de água  $r$  é adicionado a cada posição da matriz  $w$ . Em seguida, a Equação 3 descreve a etapa b), na qual uma quantidade de solo proporcional à

1  
Citação não referenciada.

2  
Citação não referenciada.

3  
Citação não referenciada.

4  
Citação não referenciada.

5  
vizinhos. Sendo

quantidade de água multiplicado pela constante de solubilidade  $s$  é retirada da matriz  $h$ , representando o desprendimento de sedimentos.

$$w_{i,j} = w_{i,j} + r \quad (2) \quad h_{i,j} = h_{i,j} - s(w_{i,j} - h_{i,j}) \quad (3)$$

A etapa c) se assemelha com o algoritmo de erosão térmica, sendo que neste caso a água será movimentada com base na diferença de altura entre os vizinhos, como demonstrado na Equação 4 que deve ser aplicada a todos os vizinhos da posição atual.  $\Delta a$  representa o valor de  $w$  atual menos a média de todos os vizinhos em  $w$ ,  $d_i$  é a diferença em  $w$  entre a posição atual e o vizinho atual e, por fim,  $d_{total}$  é a soma de todas as diferenças em  $w$  entre a posição atual e os vizinhos. O valor de  $\Delta w$  deve ser somado a todos os vizinhos em  $w$  e depois o total de todos os  $\Delta w$  deve ser subtraído da posição atual em  $w$ . Por fim, a Equação 5 apresenta a última etapa, em que  $\Delta s$  corresponde tanto a quantidade de água evaporada de  $w$  quanto a quantidade de sedimentos adicionados a  $h$  e a variável  $e$  representa uma constante que controla o fator de evaporação.

$$\Delta w_{i,j} = w_{i,j} + \min(w - \Delta a) \frac{d_i}{d_{total}} \quad (4)$$

$$\Delta s = (w_{i,j} - h_{i,j}) - ((w_{i,j} - h_{i,j})(1 - e)) \quad (5)$$

$$w_{i,j} = w_{i,j} - \Delta s$$

$$h_{i,j} = h_{i,j} + \Delta s$$

### 2.3 PROGRAMAÇÃO EM GPU COM COMPUTE SHADERS

Park *et al.* (2011, p. 2) destacam a existência de estratégias diferentes de programação em GPU dependendo da plataforma adotada. Isto significa que um algoritmo implementado em uma plataforma pode não ser facilmente adaptável para outra, o que torna a escolha de uma plataforma algo importante para o desenvolvimento em GPU. Exemplos de plataformas seriam: Compute Unified Device Architecture (CUDA) da Nvidia, DirectX da Microsoft e Metal da Apple. Neste trabalho optou-se por utilizar o DirectX 11 com a linguagem High Level Shading Language (HLSL) por razões de maior compatibilidade com o motor gráfico Unity. Como o próprio nome sugere, HLSL é uma linguagem para criação de *shaders*, que podem ser definidos como *scripts* responsáveis por comunicar à GPU quais cálculos realizar para produzir uma transformação ou efeito desejado (VALDETARO *et al.*, 2010, p. 1). O DirectX 11 apresenta diferentes tipos de *shaders* responsáveis por diferentes etapas da *pipeline* gráfica, sendo que das oito etapas existentes na *pipeline* do DirectX 11, cinco são programáveis através de *shaders* (MICROSOFT, 2018a).

Além dos *shaders* da *pipeline* gráfica, existe ainda outro chamado *compute shader* utilizado para computação de propósitos gerais, não se limitando apenas a tarefas diretamente relacionadas a renderização de objetos (KHRONOS, 2019). Este tipo de *shader* aproveita da quantidade de núcleos da GPU para otimizar a velocidade de processamento através de métodos de paralelismo, fazendo uso de compartilhamento de memória e sincronização de *threads*, sendo cada *thread* é uma unidade independente de processamento executada em um núcleo da GPU (MICROSOFT, 2018b). A organização de unidades nestes *shaders* é dividida em *threads* e grupos de *threads*, sendo que ambos possuem as dimensões  $x$ ,  $y$  e  $z$ . Na plataforma Unity, a execução de um *compute shader* é realizada através da chamada ao método *Dispatch*, sendo necessário informar o *id* do *shader* e a quantidade de grupos desejados em  $x$ ,  $y$  e  $z$ , com o número de *threads* por grupo sendo definido dentro do próprio *compute shader* (UNITY, 2020).

Ao utilizar estes *shaders* para o processamento de imagens, malhas, superfícies ou qualquer outra estrutura representada por vetores, é comum seguir um padrão de *threads* e grupos que juntos equivalham ao número de posições da estrutura utilizada. O Quadro 1 apresenta um exemplo de *compute shader* para processar uma matriz de 1025x1025. Nele é possível observar que a quantidade de grupos definidos no método *Dispatch* corresponde a 41 em  $x$  e  $y$ , enquanto a quantidade de *threads* por grupo definido no *shader* é de 25 em  $x$  e  $y$ . Multiplicando a quantidade de grupos pela quantidade de *threads* em cada grupo temos  $41 \times 25 = 1025$  nos eixos  $x$  e  $y$ , ou seja, o *shader* em questão será executado uma vez para cada posição da matriz. Destaca-se que, no DirectX 11, o limite de *threads* por grupo é de 1024 e o limite de grupos é de 65535 (KHRONOS, 2019).

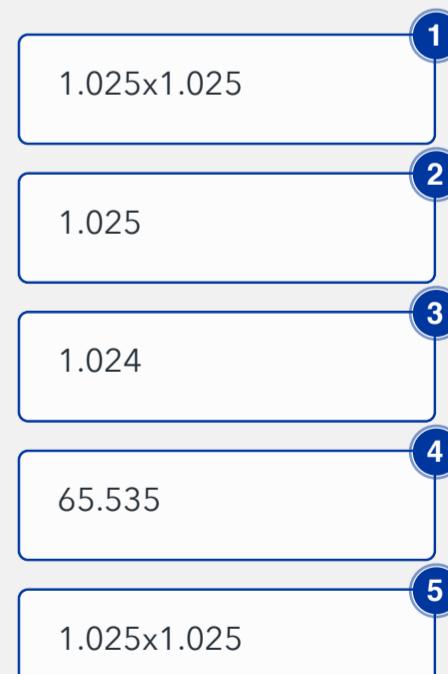
Quadro 1 - Exemplo de *compute shader* para processar uma matriz de 1025x1025

```
// execução do compute shader em C# pela Unity
shader.Dispatch(shaderId, 41, 41, 1);

// compute shader em HLSL
RWTexture2D<float4> texture;

[numthreads(25,25,1)]
void FillWithRed (uint3 id : SV_DispatchThreadID) {
    texture[id.xy] = float4(1,0,0,1);
}
```

Fonte: adaptado de Unity (2020).



O ganho de performance ao realizar processamento paralelo em GPU é visível em trabalhos como os de Hangün e Eyecioğlu (2017) e Saha *et al.* (2016), que implementam diversos algoritmos de filtros de imagens tanto em CPU como em GPU, observando melhorias de até 120% na performance de alguns dos algoritmos em GPU, sendo que esta diferença de performance aumenta conforme o tamanho do dado a ser processado aumenta (HANGÜN; EYECIOĞLU, 2017, p. 39-40). Che *et al.* (2007) propõem o mesmo estilo de análise, porém com algoritmos de propósitos gerais como simulação térmica e *K-Means*. Também se destacam os casos em que a versão em GPU do algoritmo não apresenta ganhos consideráveis se comparada com a versão em CPU. Isto pode ocorrer quando existem muitas operações sendo executadas em *threads* diferentes porém dependentes entre si, o que diminui a eficácia do paralelismo (CHE *et al.*, 2007, p. 8); ou em casos em que são necessários transferências de dados constantes entre a memória da GPU e CPU e o processo não foi corretamente configurado, fazendo mau uso da memória compartilhada da GPU (PARK *et al.*, 2011, p. 3).

## 2.4 MÉTRICAS PARA AVALIAÇÃO DE QUALIDADE DE TERRENOS

A fim de garantir que a naturalidade dos terrenos gerados não seja avaliada apenas a partir do ponto de vista visual, propõe-se o uso de duas métricas para avaliar a qualidade e naturalidade dos terrenos gerados, sendo elas o *erosion score* (OLSEN, 2004) e a lei de Benford (BENFORD, 1938). Esta seção apresenta brevemente estes métodos, suas características e aplicabilidades.

### 2.4.1 Erosion score

O *erosion score* trata-se de uma pontuação proposta por Olsen (2004) com o objetivo de medir o quanto erodido um terreno está. Este método baseia-se na suposição de que terrenos que possuem um maior índice de erosão, ou seja, um *erosion score* maior, consequentemente possuem uma aparência mais natural. Porém, Olsen (2004) destaca a dificuldade de se descrever matematicamente efeitos de erosão e que, portanto, a pontuação proposta visa avaliar a erosão de terrenos apenas a partir de características relevantes para o desenvolvimento de jogos (OLSEN, 2004, p. 1-2), que é o foco de seu trabalho e geralmente não requerem terrenos extremamente realistas.

A técnica proposta utiliza como base um mapa de declive  $s$  com as mesmas dimensões que o mapa de altura  $h$ , no qual cada posição corresponde à maior diferença de altura entre a posição atual e seus quatro vizinhos, como descrito na Equação 9. O cálculo do *erosion score* considera dois fatores do mapa de declive: a média  $\bar{s}$  (Equação 7) e o desvio padrão  $\sigma_s$  (Equação 8). Para Olsen (2004), um terreno natural para suas finalidades deve possuir uma área consideravelmente plana para possibilitar a construção de edificações e outras estruturas. Essa característica é representada por um valor de média baixo. Ao mesmo tempo, um terreno natural também deve possuir montanhas e declives, que são representados por um desvio padrão alto. Assim, o *erosion score*  $\varepsilon$  é definido como a diferença entre o desvio padrão  $\sigma_s$  e a média  $\bar{s}$  do mapa de declive  $s$ , como descrito na Equação 6. Para Olsen (2004, p. 18), terrenos adequados para suas finalidades devem possuir um *erosion score* próximo a 2.

$$\varepsilon = \frac{\sigma_s}{\bar{s}} \quad (6) \quad \bar{s} = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} s_{i,j} \quad (7) \quad \sigma_s = \sqrt{\frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (s_{i,j} - \bar{s})^2} \quad (8) \quad s_{i,j} = \max \left( \begin{array}{l} |h_{i,j} - h_{i+1,j}| \\ |h_{i,j} - h_{i-1,j}| \\ |h_{i,j} - h_{i,j+1}| \\ |h_{i,j} - h_{i,j-1}| \end{array} \right) \quad (9)$$

### 2.4.2 Lei de Benford

Também conhecida como lei do primeiro dígito, a lei de Benford foi observada e documentada pela primeira vez por Simon Newcomb em 1881, e novamente por Frank Benford em 1938, em homenagem a quem o fenômeno foi batizado (HILL, 1998, p. 1). A lei aponta que, em conjuntos de dados estatísticos, números com o primeiro dígito sendo 1 tendem a aparecer com uma frequência de aproximadamente 30%, diferente dos 11% esperados (com primeiros dígitos podendo ser entre 1 a 9, o esperado seria 1/9 para cada dígito), sendo que a probabilidade diminui gradativamente até chegar próximo a 4,6% para o primeiro dígito 9 (BENFORD, 1938, p. 554). A formalização da lei é dada pela equação

$$F_a = \log \left( \frac{a+1}{a} \right) \quad (10)$$

na qual  $F$  representa a frequência de ocorrência de  $a$ , sendo  $a$  um número inteiro de 1 a 9 (BENFORD, 1938, p. 554). Ao submeter os números inteiros de 1 a 9 à equação, tem-se a distribuição apresentada na Figura 2.

Benford (1938) observou e documentou este padrão em mais de 20 fontes de dados diferentes e totalmente distintas, como endereços de casas, área de rios, pontuações de ligas de baseball, pesos atômicos de elementos, entre outros (BENFORD, 1938, p. 553). Por se tratar de um comportamento observado em dados estatísticos naturais, esta lei tem sido utilizada em diversos contextos, alguns deles sendo identificação de fraudes em dados contábeis (DURTSCHI; HILLISON; PACINI, 2004), identificação de alterações em imagens digitais (FU; SHI; SU, 2007) e análise de dados de genoma (FRIAR; GOLDMAN; PÉREZ-MERCADER, 2012). Justamente por compreender diversos contextos, acredita-se que ela possa ser usada também para identificar terrenos naturais e avaliar terrenos gerados de forma procedural.

Citação não referenciada.

GPU. Sendo

CPU, e o

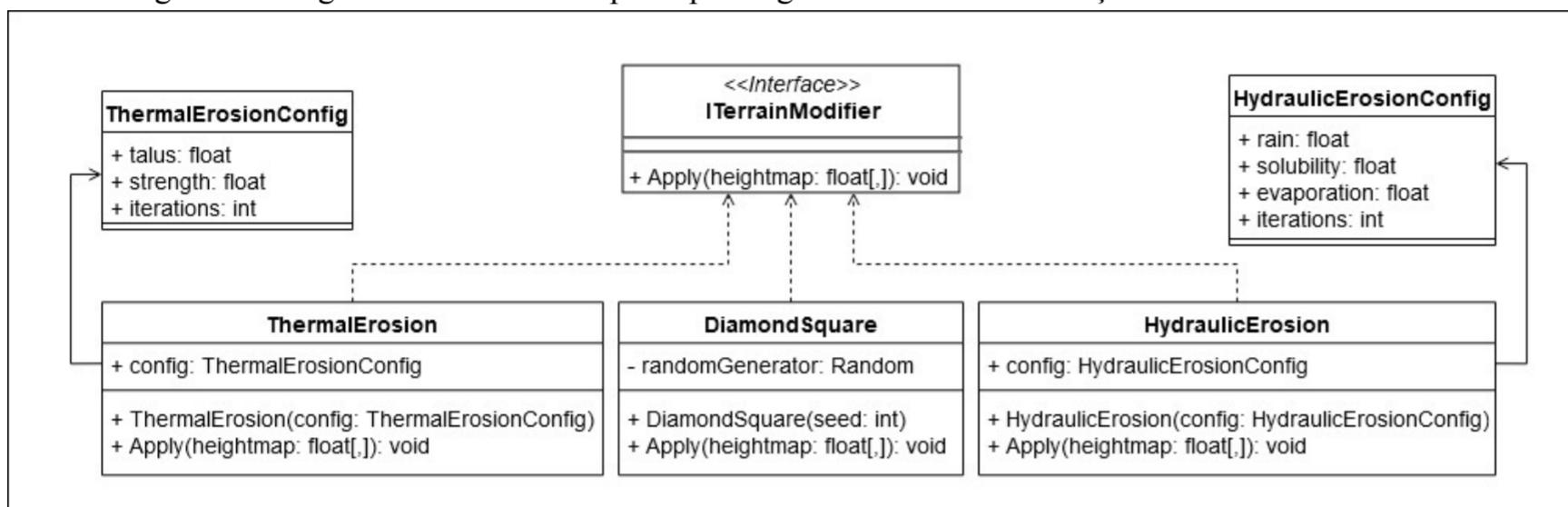
1

2

3

generalização das operações. Porém, como cada algoritmo necessita de parâmetros específicos para sua execução, estes necessitam ser passados através do construtor de cada classe, geralmente através de um objeto de configuração. A Figura 6 apresenta o diagrama de classes que demonstra estas relações, no qual é possível ver as classes DiamondSquare, ThermalErosion e HydraulicErosion implementando a interface ITerrainModifier e recebendo seus parâmetros de configuração através dos construtores.

Figura 6 - Diagrama de classes dos principais algoritmos de transformação de relevo da ferramenta



Fonte: elaborado pelo autor.

### 3.2 IMPLEMENTAÇÃO

Esta seção apresenta a implementação dos principais algoritmos por trás da construção do terreno, sendo eles o Diamond-Square, erosão térmica e erosão hidráulica. A fundamentação por trás do funcionamento destes algoritmos foi apresentada nas seções 2.1 e 2.2, assim como explicações sobre *compute shaders* na seção 2.3 que são utilizados pelos algoritmos. Por mais que o trabalho realize implementação dos algoritmos tanto na linguagem C# para execução em CPU como em HLSL para execução em GPU, esta seção focará apenas nas versões dos algoritmos para GPU por terem maior relevância para a ferramenta.

Todos os algoritmos C# responsáveis pela execução dos *compute shaders* e definição das variáveis utilizadas por eles representam o terreno através de uma matriz bidimensional de pontos flutuantes com dimensões de 1025x1025. Ao ser transferida para os algoritmos em *compute shader* esta matriz passa a ser representada por um vetor unidimensional com tamanho 1025<sup>2</sup>, visto que os *shaders* trabalham com *buffers* de memória sequenciais. Em ambos os casos cada posição da estrutura possui valores entre 0 e 1 que representam a altura do relevo naquela posição. Além disso, todos os algoritmos possuem validações para evitar erros como não ultrapassar os limites das bordas do terreno, não multiplicar valores por 0, entre outros. Porém, a fim de focar apenas nas partes mais relevantes, estas validações foram ocultadas dos códigos apresentados nesta seção. Exemplos de terrenos gerados por cada algoritmo estão disponíveis no Apêndice A.

#### 3.2.1 Algoritmo Diamond-Square

O algoritmo Diamond-Square começa na linguagem de programação C#, onde primeiro são definidos valores aleatórios entre -1 e 1 para os quatro cantos do terreno. Em seguida são configuradas as propriedades necessárias para executar o *compute shader* que contém a lógica principal do algoritmo e então iniciam-se as iterações, como mostrado no Quadro 5. As propriedades necessárias para o *shader* são a matriz que representa o terreno (chamada de *heightmap*), sua largura, um número aleatório (cuja utilidade será explicada adiante), o tamanho do quadrado atual e o fator *H* que controla a variação de altura. Todas estas propriedades são definidas pelo método chamado na primeira linha do Quadro 5.

Quadro 5 - Inicialização das propriedades e execução do *compute shader* do Diamond-Square

```

int kernelId = InitComputeShaderProperties();

float H = 1.0f;
int squareSize = terrainSize;
int numThreads = 0;

while (squareSize > 0) {
    numThreads = numThreads > 0 ? (numThreads * 2) : 1;

    RunComputeShader(kernelId, squareSize, H, numThreads);

    squareSize /= 2;
    H /= 2f;
}
  
```

Porque está letras maiúsculo e os outros dois nomes não?

código o vetor que representa o terreno foi renomeado para `map`). Por fim, em relação aos números aleatórios utilizados nas etapas, não há nenhum gerador disponibilizado pelo DirectX 11 para isso. Portanto optou-se por utilizar o algoritmo Wang Hash para geração destes números por ser considerada uma técnica eficiente para geração de números aleatórios em *shaders* (REED, 2013). Foi utilizado como semente para o *hash* uma combinação do id da *thread*, a coordenada atual e um valor aleatório gerado fora da GPU a cada iteração. Sem este valor auxiliar cada *thread* em uma iteração teria uma semente diferente para o *hash* (devido ao id de cada *thread*) mas todas as iterações teriam as mesmas sementes.

Quadro 7 - Etapas *diamond step* e *square step* do Diamond-Square

```
// diamond step
map[mid] = (map[topLeft]+map[topRight]+map[bottomLeft]+map[bottomRight])/4 + rand()*H;

// square step
map[up]= (map[topLeft]+map[topRight]+map[mid]+map[up+halfSize])/4 + rand()*H;
map[down]= (map[bottomLeft]+map[bottomRight]+map[mid]+map[down-halfSize])/4 + rand()*H;
map[left]= (map[topLeft]+map[bottomLeft]+map[mid]+map[left-halfSize])/4 + rand()*h;
map[right]= (map[topRight]+map[bottomRight]+map[mid]+map[right+halfSize])/4 + rand()*H;
```

Fonte: elaborado pelo autor.

### 3.2.2 Algoritmos de erosão

Tratando-se da lógica de paralelismo, os algoritmos de erosão são relativamente mais simples que o Diamond-Square. Todos trabalham com iterações, porém, diferente do Diamond-Square, os algoritmos de erosão são aplicados igualmente em todas as posições do terreno a cada iteração. Ou seja, o número de *threads* utilizadas a cada iteração é sempre o mesmo, o que facilita sua implementação neste quesito. Como o terreno possui dimensões de 1025x1025, optou-se por utilizar a configuração de 41 grupos de *threads* em *x* e *y*, com 25 *threads* em *x* e *y* cada ( $41 * 25 = 1025$ ). Além disso, optou-se por utilizar a vizinhança de Von Neumann com quatro vizinhos. Estas configurações foram mantidas para ambos os algoritmos de erosão térmica e hidráulica.

1 Aconselhasse a não iniciar uma frase com verbo.

#### 3.2.2.1 Algoritmo de erosão térmica

A cada iteração do algoritmo, o *compute shader* apresentado no Quadro 8 é aplicado a cada posição do vetor `heightmap`, que representa o terreno. Os nomes das variáveis utilizadas foram mantidos o mais semelhante possível aos da Equação 1 apresentada na seção 2.2. Lembrando que por utilizar a vizinhança de Von Neumann a constante `NEIGHBORHOOD_SIZE` possui o valor 4. O primeiro loop do algoritmo não realiza qualquer modificação no terreno, tratando apenas de calcular os valores de `dTotal` e `dMax` necessárias para a segunda parte, onde ocorre o cálculo de movimentação de sedimento. As únicas variáveis parametrizáveis no algoritmo são `talus`, cujo valor pode variar entre  $2/N$  e  $12/N$  (valores menores tornam o terreno excessivamente plano e maiores não causam alteração no terreno); e `c`, cujo valor é mantido normalmente em 0,5 ou próximo disso.

2 Itálico.

Quadro 8 - Algoritmo de erosão térmica em *compute shader*

```
uint currentPosition = id.x + (id.y * width);
getNeighbors(currentPosition);

float dMax = 0;
float dTotal = 0;
float di;

for(uint i = 0; i < NEIGHBORHOOD_SIZE; i++) {
    di = heightmap[currentPosition] - heightmap[neighbors[i]];
    if (di > talus) {
        dTotal += di;
        if (di > dMax)
            dMax = di;
    }
}
for(uint j = 0; j < NEIGHBORHOOD_SIZE; j++) {
    di = heightmap[currentPosition] - heightmap[neighbors[j]];

    if (di > talus && isValidPosition(id)) {
        float sediment = c * (dMax - talus) * (di / dTotal);

        heightmap[currentPosition] -= sediment;
        heightmap[neighbors[j]] += sediment;
    }
}
```

Fonte: elaborado pelo autor.

### 3.2.2.2 Algoritmo de erosão hidráulica

Como apresentado na seção 2.2, o algoritmo de erosão hidráulica é dividido em quatro etapas. As etapas a) e b) em especial podem ser executadas simultaneamente e portanto foram implementadas em um único *compute shader* a fim de otimizar o tempo de execução. Já as etapas c) e d) requerem que todas as posições do terreno tenham sido processadas pelas etapas anteriores e portanto foram implementadas em *compute shaders* específicos. Sendo assim, este algoritmo é composto por três *compute shaders*, um comprendendo as duas primeiras etapas e dois comprendendo as duas últimas. Estes *shaders* são executados sequencialmente através de chamadas separadas ao método *Dispatch* utilizando as mesmas configurações de grupos e *threads* mencionadas anteriormente, sendo que as alterações no terreno são realizadas através da manipulação dos vetores *watermap* que representa a água da chuva sob o terreno e *heightmap* que representa o terreno. O Quadro 9 apresenta o *shader* responsável pelas duas primeiras etapas (Equação 2 e 3), em que primeiro some-se o valor da constante *rainFactor* a todas as posições de *watermap*, representando a chuva sob o terreno. Depois, subtraí-se de todas as posições de *heightmap* um percentual com base na quantidade de água no local multiplicado pela constante *solubility*, representando a transformação de terreno em sedimento causado pelo excesso de água da chuva.

Quadro 9 - Etapas a) e b) do algoritmo de erosão hidráulica em *compute shader*

```
uint currentPosition = id.x + (id.y * width);

// Etapa a) - aparecimento de água da chuva
watermap[currentPosition] += rainFactor;

// Etapa b) - dissolve superfície em sedimento;
float waterVolume = watermap[currentPosition] - heightmap[currentPosition];
heightmap[currentPosition] -= solubility * waterVolume;
```

Fonte: elaborado pelo autor.

Em seguida a etapa c) trata de simular o fluxo de movimentação de água através de um algoritmo similar ao de erosão térmica, como apresentado no Quadro 10. O primeiro loop 2 aplica apenas de percorrer todos os vizinhos da posição atual coletando a diferença total entre alturas (*dTotal*) e a média de altura dos vizinhos (*avgSurfaceHeight*). O segundo loop 3 aplica a lógica da Equação 4, adicionando uma quantidade *deltaWater* diferente para cada vizinho e depois subtraindo o *totalDeltaWater* da posição atual, simulando a movimentação de água da posição atual para cada um dos vizinhos, sendo que a quantidade transferida varia dependendo da diferença de altura do vizinho (*di*).

Quadro 10 - Etapa c) do algoritmo de erosão hidráulica em *compute shader*

```
uint currentPosition = id.x + (id.y * width);

float localWaterVolume = watermap[currentPosition] - heightmap[currentPosition];
float avgSurfaceHeight = 0;
int countHeights = 0;
float dTotal = 0;

for(uint i = 0; i < NEIGHBORHOOD_SIZE; i++) {
    dTotal += watermap[currentPosition] - watermap[neighbors[i]];
    avgSurfaceHeight += watermap[neighbors[i]];
    countHeights++;
}

avgSurfaceHeight /= countHeights;
float deltaSurfaceHeight = watermap[currentPosition] - avgSurfaceHeight;
float totalDeltaWater = 0;

for(i = 0; i < NEIGHBORHOOD_SIZE; i++) {
    float di = watermap[currentPosition] - watermap[neighbors[i]];
    float deltaWater = min(localWaterVolume, deltaSurfaceHeight) * (di / dTotal);

    watermap[neighbors[i]] += deltaWater;
    totalDeltaWater += deltaWater;
}
watermap[currentPosition] -= totalDeltaWater;
```

Fonte: elaborado pelo autor.

Por fim, o Quadro 11 apresenta a última etapa do algoritmo representada pela Equação 5. Nesta etapa são calculadas a quantidade de água evaporada de cada posição e o total de sedimento adicionado a cada posição após a evaporação da água. A evaporação é realizada através da subtração de uma quantidade de água de *watermap* com base na constante *evaporationPercent*, enquanto a adição de sedimentos adiciona o mesmo valor ao *heightmap*, porém multiplicado pela constante *solubility*, similar ao processo inverso realizado na etapa b).

anteriormente. Sendo

Itálico.

Itálico.

Equação 5 (seção 2.2).  
Nesta

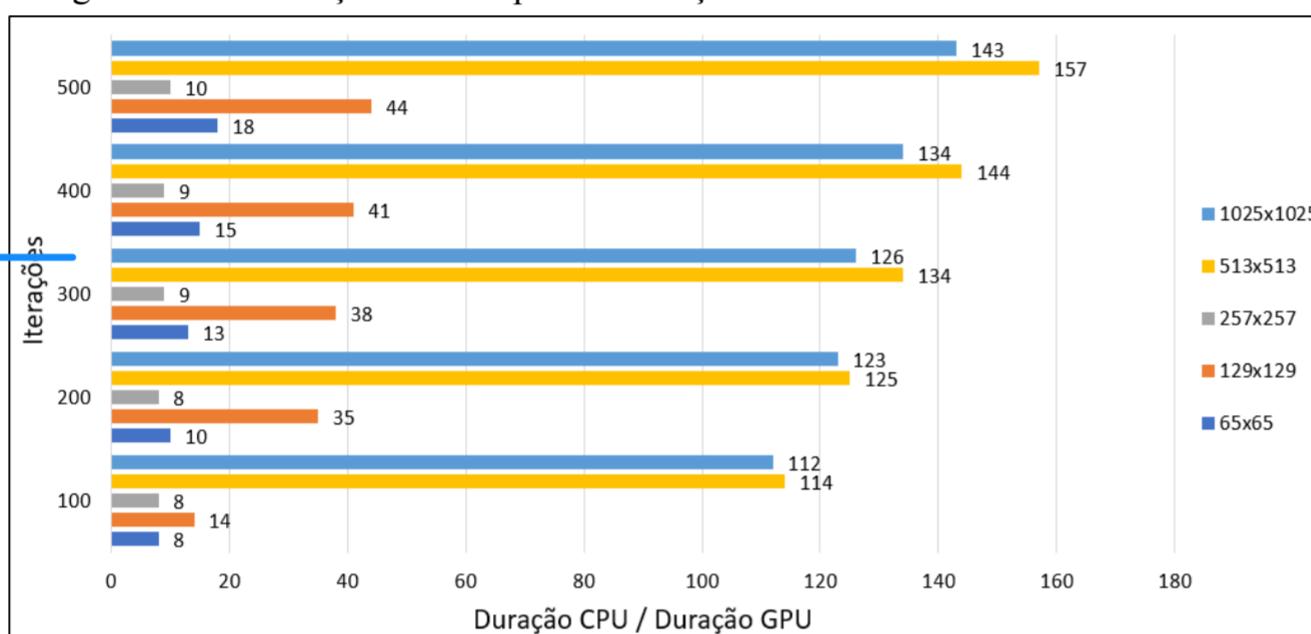
No caso do algoritmo de erosão hidráulica é possível observar um comportamento muito semelhante ao erosão térmica, com a diferença de que a versão em CPU do erosão hidráulica demonstra ser mais rápida que a versão em GPU do erosão térmica, enquanto que na versão em GPU o cenário se inverte, com o erosão hidráulica sendo mais lento que o erosão térmica. A Tabela 3 apresenta os dados de execução do erosão hidráulica, seguido da Figura 10 que demonstra as diferenças entre CPU e GPU para cada iteração e dimensão de terreno, no qual observa-se que a versão em GPU do algoritmo chega a ser 142 vezes mais rápida que a versão em CPU com os terrenos com dimensão de 1025x1025.

Tabela 3 - Média em milissegundos da execução da erosão hidráulica em CPU e GPU

Dimensões	100 iterações		200 iterações		300 iterações		400 iterações		500 iterações	
	CPU	GPU								
65x65	44	5	87	9	132	10	177	12	235	13
129x129	175	13	344	10	518	14	703	17	920	21
257x257	683	84	1362	164	2054	241	2711	312	3601	376
513x513	2848	25	5620	45	8645	64	11796	82	15407	98
1025x1025	10955	98	21830	178	32360	256	43879	328	55879	391

Fonte: elaborado pelo autor.

Figura 10 - Diferença entre tempo de execução em CPU e GPU da erosão hidráulica



Fonte: elaborado pelo autor.

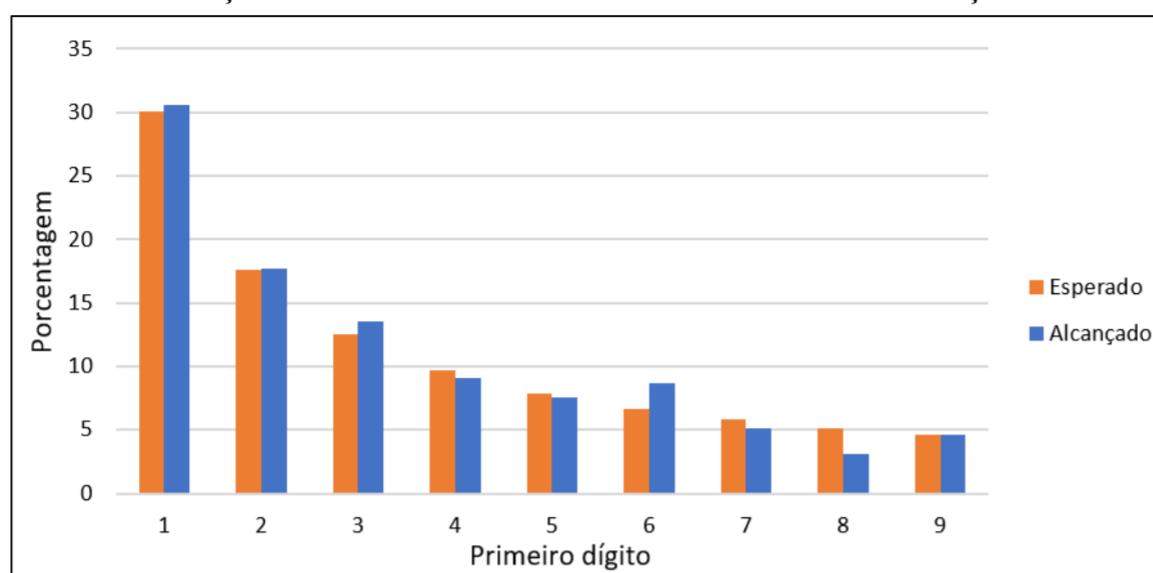
A partir destes resultados observa-se que as versões em GPU dos três algoritmos foram mais eficientes que os em CPU, com o algoritmo de erosão térmica tendo a maior diferença. O Diamond-Square obteve o melhor tempo de execução entre os três e a menor diferença em relação a sua versão em CPU, o que indica que sua versão em GPU gerou o menor ganho de performance comparado aos outros algoritmos. Já o erosão térmica obteve a maior diferença entre a versão em CPU e GPU, sendo o algoritmo cuja versão em CPU teve o maior tempo de execução. Por fim, a versão em GPU do erosão hidráulica obteve o maior tempo de execução entre os três, levando cerca de 5 vezes mais que o erosão térmica. Como os dois algoritmos de erosão realizam operações semelhantes e mesmo que o erosão hidráulica realize mais operações nenhuma apresenta grande complexidade computacional, acredita-se que esta diferença se deve ao fato de o erosão hidráulica executar três *shaders* sequencialmente, enquanto o erosão térmica utiliza apenas um.

#### 4.2 ANÁLISE DE QUALIDADE

Para avaliar a qualidade dos terrenos gerados foram utilizadas as métricas *erosion score* e lei de Benford, apresentadas nas seções 2.4.1 e 2.4.2. Porém, antes de submeter os terrenos gerados à esta análise, as métricas foram primeiro aplicas em terrenos reais, possibilitando assim comparar se os resultados alcançados pelos terrenos gerados correspondem aos alcançados por terrenos reais. Para isso, foram escolhidos de forma aleatória mapas de altura de 30 regiões diferentes através da ferramenta *online terrain.party*, que disponibiliza mapas de altura dos datasets ASTER, USGS NED e SRTM3. Os mapas foram coletados manualmente de diversas regiões do globo sem qualquer critério de escolha, a fim de representar uma amostragem aleatória. O gráfico da Figura 11 apresenta a média de distribuição de alturas dos relevos coletados em relação à lei de Benford, no qual a primeira coluna corresponde ao percentual esperado pela lei e a segunda corresponde ao resultado obtido pelos terrenos testados. Destaca-se que a lei de Benford leva em consideração a porcentagem de ocorrência dos primeiros números entre 1 e 9 em distribuições de dados naturais, sendo que neste caso os dados considerados na validação são as alturas do mapa de altura dos terrenos. Todos os dados coletados a partir desta análise estão disponíveis na Tabela 5 do Apêndice B e todos os mapas de altura dos terrenos reais utilizados estão disponíveis no Quadro 15 do mesmo apêndice.

Porque as amostras com 513x513 é maior que as de 1025x1025?

Figura 11 - Distribuição de alturas de terrenos reais analisados em relação à lei de Benford

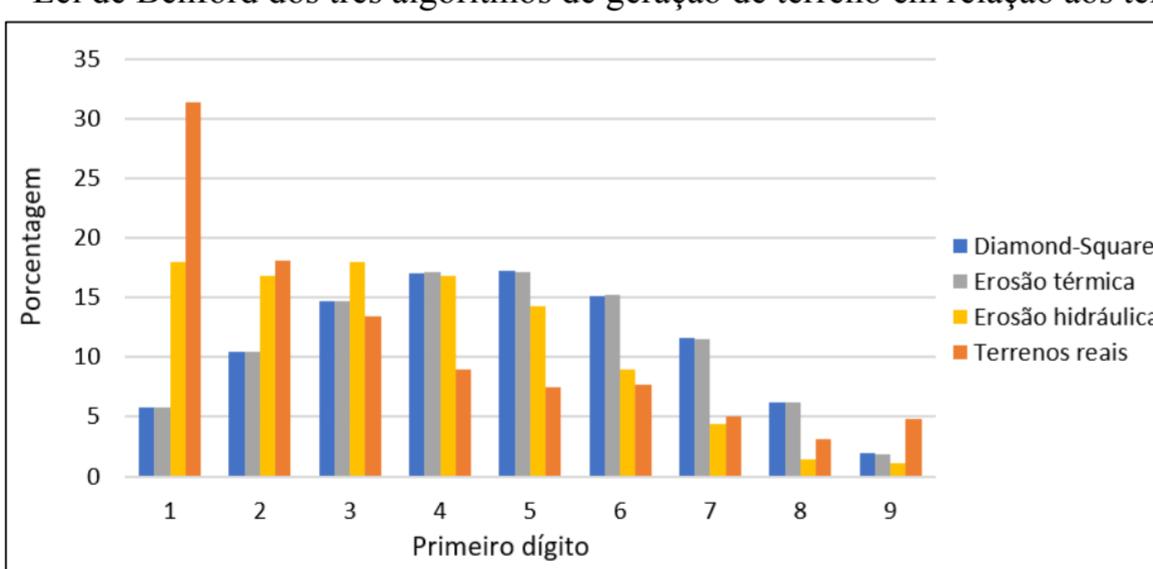


Fonte: elaborado pelo autor.

A partir do gráfico é possível observar que as alturas dos terrenos reais analisados correspondem quase que perfeitamente à distribuição esperada pela lei de Benford. Todos os números tiveram divergências de alguns pontos percentuais, porém, como os próprios dados coletados por Benford (1938) possuíam pequenas divergências, acredita-se que tais diferenças observadas neste contexto podem ser ignoradas. Parte das divergências também pode ser justificada por perda de precisão visto que os terrenos foram exportados pela ferramenta terrain.party em formato PNG e importados na ferramenta desenvolvida para que pudessem ser analisados. Apesar disto, este experimento serve para comprovar que terrenos reais respeitam a distribuição da lei de Benford e, sendo assim, tal técnica pode ser utilizada para avaliar a naturalidade dos terrenos gerados pela ferramenta desenvolvida. Além da lei de Benford, analisou-se também o *erosion score* dos terrenos reais seguindo as equações apresentadas na seção 2.4.1, no qual os terrenos avaliados atingiram valor médio de 0,697. Desta forma, espera-se que os terrenos gerados pela ferramenta desenvolvida alcancem distribuições e pontuações próximas a estas.

Tratando-se dos terrenos gerados, foram analisados os mesmos 50 terrenos utilizados na análise de performance, porém neste caso não houve necessidade de considerar diferentes dimensões de um mesmo terreno e, portanto, as análises foram feitas com base apenas nos terrenos com dimensões de 1025x1025. O gráfico da Figura 12 apresenta os resultados alcançados ao aplicar a lei de Benford nos terrenos gerados pelo Diamond-Square, melhorados pela erosão térmica e melhorados pela erosão hidráulica em comparação aos terrenos reais. Em seguida a Tabela 4 apresenta o *erosion score* alcançado por cada algoritmo e pelos terrenos reais analisados.

Figura 12 - Lei de Benford dos três algoritmos de geração de terreno em relação aos terrenos reais



Fonte: elaborado pelo autor.

A partir dos resultados alcançados pela lei de Benford é possível observar que nenhum dos algoritmos utilizados teve um resultado próximo ao dos terrenos reais. Entre os três, o algoritmo de erosão hidráulica foi o que mais se aproximou dos terrenos reais, porém ainda tendo uma diferença de 13 pontos percentuais no caso do primeiro dígito 1. O Diamond-Square, como previsto, teve uma distribuição totalmente diferente da esperada pela lei de Benford, com os números começando baixo, atingindo seu pico no dígito 5 e voltando a decair. Esta distribuição anormal expressa o fato de terrenos gerados de forma unicamente procedural através de modelos estocásticos (como é o caso do Diamond-Square) não possuírem características naturais o suficiente para se assemelharem a terrenos reais, o que exige a melhoria desses terrenos através de modelos físicos como algoritmos de erosão.

1 Aconselhasse não iniciar frases com verbo.

2 melhorados pela erosão térmica e erosão hidráulica

e erosão hidráulica apresentados na Tabela 8 o primeiro dígito 1 varia entre 2% e 38%. Já nos dados coletados do algoritmo com filtro apresentados na Tabela 9, como o valor mínimo aceito para o primeiro dígito 1 é 28%, os valores variam apenas entre 28% e 44%, ou seja, possui a menor variação. Todas as tabelas mencionadas encontram-se no Apêndice B.

Acredita-se que uma maior variação individual dos resultados represente uma maior diversificação de formatos de terrenos gerados, o que é positivo para a ferramenta visto que o objetivo é a geração de terrenos que se assemelhem aos reais. Por estes fatores e pelo alto tempo de execução, acredita-se que mesmo o algoritmo com filtro tendo atingido resultados gerais de qualidade mais positivos, a combinação original de Diamond-Square e erosão hidráulica ainda é preferível por ter um tempo de execução consideravelmente menor e maior diversificação na distribuição de alturas. Porém o algoritmo com filtro ainda é uma opção para geração de terrenos que individualmente respeite a lei de Benford.

## 5 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de uma ferramenta para geração de terrenos com aparência natural utilizando programação em GPU. Os principais algoritmos utilizados durante o processo foram o Diamond-Square para geração do terreno base e erosão térmica e erosão hidráulica para melhorar a qualidade do terreno. Os algoritmos de erosão foram adaptados para GPU a partir das versões desenvolvidas por Diegoli Neto (2017), enquanto o Diamond-Square foi baseado na versão original de Fournier, Fussell e Carpenter (1982). Além da geração de terrenos, este trabalho também analisou a aplicabilidade de métricas para avaliar a qualidade dos terrenos gerados em comparação com terrenos reais, com as métricas utilizadas sendo o *erosion score* (OLSEN, 2004) e lei de Benford (BENFORD, 1938).

O desenvolvimento da ferramenta foi realizado com o motor gráfico Unity nas linguagens de programação C# para estruturação geral da ferramenta e implementação dos algoritmos em CPU e HLSL para implementação dos *compute shaders* executados em GPU. Os objetivos de disponibilizar um executável da ferramenta para desktop e um *plugin* para a Unity foram ambos alcançados, estando disponíveis para download no repositório de Gonçalves (2020). O uso da Unity proporcionou uma maior facilidade no desenvolvimento de aspectos gerais da ferramenta, como interface gráfica e interação entre algoritmos e componentes gráficos, além de proporcionar o ~~desenvolvimento multiplataforma~~. O uso de *compute shaders* para execução dos algoritmos com maior custo computacional possibilitou uma alta performance da ferramenta, algo que a Unity sozinha (sem uso de GPU) não conseguiu proporcionar em trabalhos anteriores, como concluído por Diegoli Neto (2017, p. 57).

2 Tratando-se dos aspectos de performance da ferramenta, as versões em GPU dos três algoritmos alcançaram os objetivos desejados, sendo mais eficientes que suas versões em CPU em todos os cenários testados. Considerando as versões em GPU, o erosão hidráulica teve o maior tempo de execução, com duração de 391 milissegundos e, portanto, mesmo ao considerar uma execução sequencial dos três algoritmos, o tempo total ainda seria menor que um segundo (desconsiderando o tempo de renderização do terreno). Considerando que uma única execução do erosão térmica sozinho em CPU pode levar até aproximadamente 3 minutos (180006 milissegundos), o ganho alcançado com os algoritmos em GPU se mostrou bastante significativo.

As métricas de qualidade utilizadas mostraram-se adequadas para as análises realizadas dos terrenos gerados, porém como ainda não são métricas consolidadas, serão necessários mais estudos acerca de sua viabilidade neste cenário. Acredita-se que ambas o *erosion score* e lei de Benford foram adequadas principalmente devido ao fato de terrenos reais terem sido usados como base para determinar quais seriam os valores desejados para os terrenos gerados. Ao aplicar a lei de Benford em terrenos reais, seu comportamento correspondeu quase que perfeitamente ao esperado, o que indica que a lei se aplica corretamente a terrenos. O *erosion score*, por outro lado, não havia sido aplicado a terrenos reais até então e, portanto, não possuía um “valor ideal” que pudesse ser utilizado como base, o que torna sua viabilidade mais duvidosa. Porém ao analisar a correlação entre o *erosion score* e lei de Benford observa-se que em todos os casos em que a lei de Benford aumenta, o *erosion score* também aumenta. Seguindo este padrão, em um caso como o do erosão térmica, que provocou diminuição no *erosion score* (demonstrado na Tabela 4), esperava-se que a lei de Benford também diminuisse, porém nesse caso ela permaneceu sem nenhuma mudança significativa. Isto pode ser um indicativo de inconsistência entre as duas métricas, porém como não houve nenhum caso em que o comportamento das duas foram contrários (uma aumentar e a outra diminuir), ainda acredita-se que ambas sejam confiáveis e que este caso do erosão térmica sirva talvez para mostrar que o *erosion score* seja mais sensível a alterações no terreno do que a lei de Benford, visto que as mudanças causadas pelo erosão térmica são mínimas se comparado aos outros algoritmos.

4 Referente a qualidade dos terrenos gerados, pode-se concluir que o objetivo de gerar terrenos com aparência natural foi trabalhado, mas não totalmente alcançado. Ao submeter os terrenos sob avaliação das métricas de qualidade utilizadas o algoritmo de erosão hidráulica apresentou melhorias consideráveis no terreno, porém não foram suficientes para que os terrenos gerados se igualassem aos reais analisados. A inclusão de um filtro de qualidade no algoritmo Diamond-Square que selecionasse apenas os melhores terrenos gerados trouxe melhorias significativas, resultando em um valor da lei de Benford bastante próximo ao dos terrenos reais. Porém, como discutido durante a análise dos resultados, este ganho veio acompanhado de uma grande perda de performance e do que acredita-se ser uma diminuição na diversidade de relevo dos terrenos gerados se comparado aos terrenos reais, o que invalidou seu uso como “solução ideal”.

Dúvida .. por usar HLSL DirectX vai executar em MacOs,Linux,iOS e Android?

Aconselhasse não iniciar frase com verbo.

180.006

foi desenvolvido, mas

Por mais que existam diversas pesquisas, propostas de ferramentas ou melhorias relacionadas a processos de geração procedural de terrenos, a análise de qualidade dos terrenos gerados tende a não ser uma prioridade em grande parte delas, sendo que de todos os autores estudados durante o desenvolvimento deste trabalho, nenhum deles apresentou análises comparativas entre os terrenos gerados e terrenos reais. Nestes casos, a análise da semelhança entre os terrenos gerados e reais se dá apenas com base em conclusões subjetivas tiradas a partir de sua representação visual, o que levanta a questão de se os terrenos gerados nestes trabalhos realmente possuem características naturais semelhantes a terrenos reais, ou se estas supostas semelhanças são apenas conclusões incorretas de observações visuais. Neste aspecto, mesmo o trabalho desenvolvido não tendo atingido os resultados esperados na qualidade dos terrenos gerados, acredita-se que a análise comparativa realizada entre os terrenos gerados e reais possa servir como incentivo para novas pesquisas na área, como por exemplo propostas de novas métricas para análise, melhoria na aplicação das já utilizadas ou comprovação ou não de sua viabilidade nestes contextos.

A partir destas discussões, levantam-se algumas extensões para expandir o trabalho realizado, sendo elas:

- a) incluir diferentes camadas de solo que influenciem nas dinâmicas de relevo causadas pelos algoritmos de erosão, como feito por Diegoli Neto (2017) e Peytavie *et al.* (2009);
- b) utilizar outros algoritmos de erosão como o de Beyer (2015) que apresenta um algoritmo de erosão hidráulica que simula o comportamento de diversas gotas de água independentes erodindo o terreno;
- c) incluir elementos naturais como rios ao processo de geração de terrenos, como proposto por Génevaux *et al.* (2013) que geram redes de rios conectados através de uma estrutura de grafo;
- d) identificar a escala dos terrenos gerados em unidades reais. Tanto o trabalho desenvolvido como os estudados não apresentam qualquer relação entre a escala dos terrenos gerados e terrenos reais, tornando difícil relacioná-los (acredita-se que a falta de escala deve-se à falta de pontos de referência);
- e) analisar mais terrenos reais a fim de coletar outras informações que possam ser utilizadas para validar a naturalidade de terrenos gerados. Isto pode ser feito através da análise de *datasets* inteiros de dados geológicos como os ASTER, USGS NED e SRTM3 utilizados pela ferramenta terrain.party;
- f) expandir os estudos sobre a lei de Benford, *erosion score* e outras métricas considerando sua aplicabilidade na análise de terrenos naturais. Algumas métricas como a lei de Benford podem funcionar corretamente ao analisar aspectos macros do terreno, mas talvez não funcionem para aspectos micros, sendo necessário o uso de outras técnicas não estudadas neste trabalho;
- g) implementar melhorias de performance nos algoritmos em GPU. Por mais que a execução destes algoritmos tenha atingido os resultados esperados, ainda é possível dedicar mais esforços a fim de otimizar sua execução através de técnicas de *branchless programming*, gerencia de memória compartilhada e melhor gerencia de *threads* nas etapas de algoritmos que precisam ser sequenciais, como é o caso da erosão hidráulica.

## REFERÊNCIAS

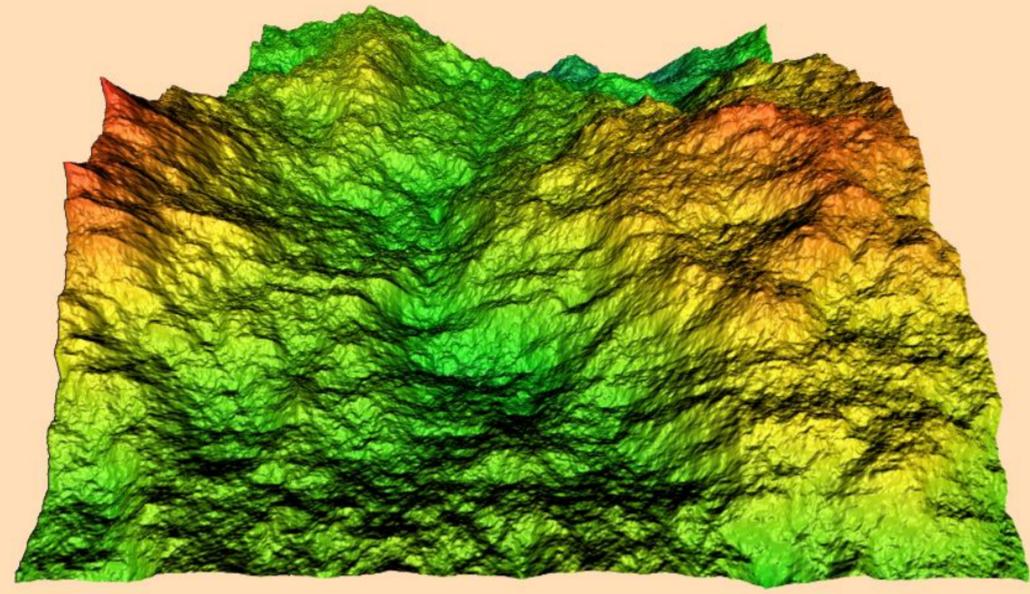
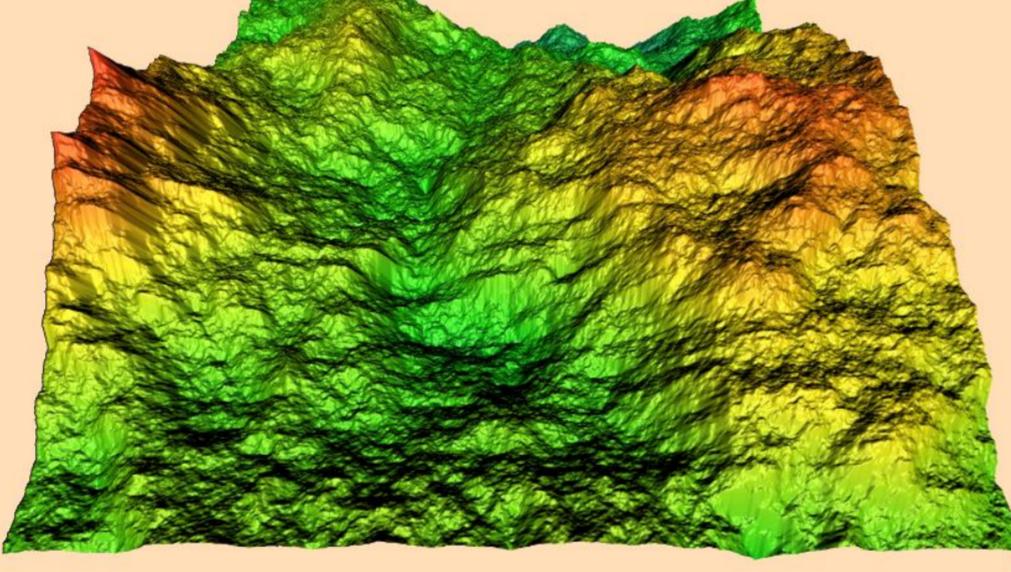
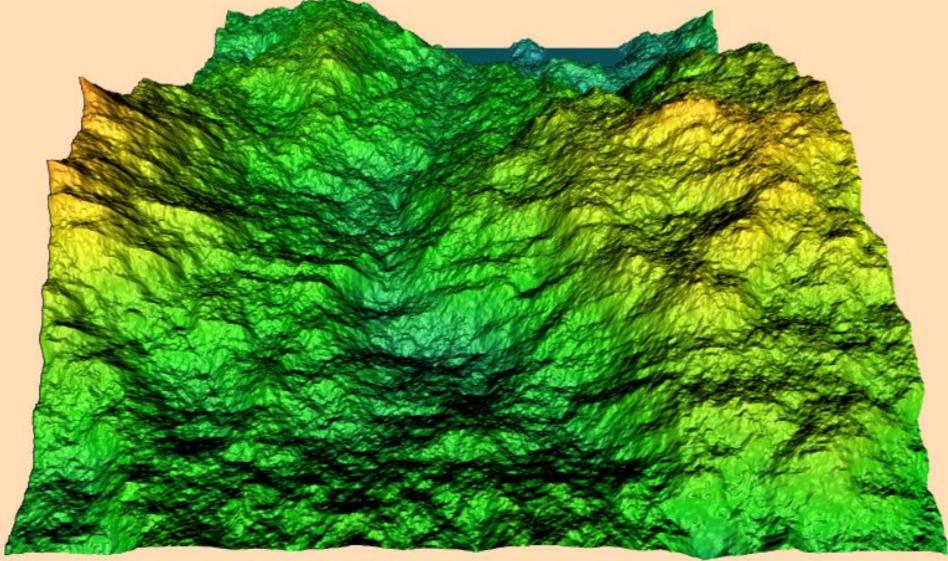
- BENFORD, Frank. The law of anomalous numbers. **Proceedings of The American Philosophical Society**, Philadelphia, v. 78, n. 4, p. 551-572, mar. 1938.
- 1** BENES, Bedřich; FORSBACH, Rafael. Visual simulation of hydraulic erosion. **Journal of WSCG**, West Bohemia. v. 10, p. 79–86, fev. 2020.
- BEYER, Hans T. **Implementation of a method for hydraulic erosion**. 2015. 29 f. Thesis (Bachelor's in Informatics: Games Engineering) - Department of Informatics, Technische Universität München, München.
- CHE, Shuai *et al.* A performance study of general purpose applications on graphics processors. In: FIRST WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS, 1., 2007, Boston. **Proceedings...** Boston: [s.n.], 2007. p. 1-10.
- DIEGOLI NETO, Guilherme. **Simulação de dinâmica do relevo através da transformação de mapas de altura**. 2017. 64 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.
- DURTSCHI, Cindy; HILLISON, William; PACINI, Carl. The effective use of Benford's law to assist in detecting fraud in accounting data. **Journal of Forensic Accounting**, [S.I.], v. 5, n. 1, p. 17-34, 2004.
- EBERT, David. S. *et al.* **Texturing and Modeling**: A Procedural Approach. 3. ed. São Francisco: Morgan Kaufmann, 2003.
- FOURNIER, Alain; FUSSELL, Don; CARPENTER, Loren. Computer rendering of stochastic models. **Communications of the ACM**, New York, v. 25, n. 6, p. 371-384, jun. 1982.
- FRIAR, James L.; GOLDMAN, Terrance; PÉREZ-MERCADER, Juan. Genome sizes and the Benford distribution. **PLoS One**, [S.I.], v. 7, n. 5, p. e36624, maio 2012.
- FU, Dongdong; SHI, Yun Q.; SU, Wei. A generalized Benford's law for JPEG coefficients and its applications in image forensics. In: SECURITY, STEGANOGRAPHY, AND WATERMARKING OF MULTIMEDIA CONTENTS, 9., 2007, San José, CA. **Proceedings...** New York: Curran Associates, Inc, 2007. p. 1-11.

Ano 2002?

## APÊNDICE A – TERRENOS GERADOS PELA FERRAMENTA

Este apêndice apresenta alguns dos terrenos gerados pela ferramenta. O Quadro 13 apresenta 5 terrenos gerados, sendo que para cada um é exibido a semente utilizada na geração, o terreno base gerado pelo Diamond-Square e as versões do mesmo terreno submetido ao erosão térmica e erosão hidráulica, sendo possível assim observar as diferenças causadas no terreno por cada um dos algoritmos.

Quadro 13 - Terrenos gerados pela ferramenta através do Diamond-Square, erosão térmica e erosão hidráulica

Sementes	Terrenos gerados
	 Diamond-Square
1428799138	 Erosão térmica
	 Erosão hidráulica

hidráulica. Assim permitido observar

## APÊNDICE B – DADOS DE QUALIDADE COLETADOS PARA ANÁLISE

Este apêndice apresenta todos os dados de qualidade coletados, a partir dos quais extraiu-se as médias utilizadas nas discussões do capítulo 4. O Quadro 14 apresenta a relação de grupos de *threads* e *threads* por grupo utilizada para cada dimensão dos terrenos testados. A Tabela 5 apresenta os resultados da lei de Benford e *erosion score* de 30 terrenos reais recuperados a partir do site [terrain.party](#), seguido do Quadro 15 listando os mapas de altura destes terrenos respeitando a ordem da Tabela 5. Em seguida são apresentados os resultados de lei de Benford e *erosion score* gerados por 50 execuções dos algoritmos desenvolvidos, sendo a Tabela 6 referente ao Diamond-Square, Tabela 7 ao erosão térmica, Tabela 8 ao erosão hidráulica e, por fim, a Tabela 9 com os dados da versão do Diamond-Square com filtro de melhores resultados.

Destaca-se que o fato de os terrenos com dimensão de 257x257 apresentado no Quadro 14 possuir a organização de 257 grupos de *threads* em x e y e uma única *thread* por grupo deve-se ao fato de 257 ser um número primo e, portanto, divisível apenas por ele mesmo e um, o que invalida qualquer outra distribuição possível de *threads*.

Quadro 14 - Relação entre *threads* e grupos de *threads* utilizada para cada dimensão de terreno

Dimensões	Grupos de <i>threads</i> em x e y	<i>Threads</i> por grupo em x e y	Total de <i>threads</i>
65 x 65	5 x 5	13 x 13	4225
129 x 129	43 x 43	3 x 3	16641
257 x 257	257 x 257	1 x 1	66049
513 x 513	19 x 19	27 x 27	263169
1025 x 1025	41 x 41	25 x 25	1050625

Fonte: elaborado pelo autor.

dos

Tabela 5 - Resultados dos terrenos reais recuperados do site [terrain.party](#)

Número	Distribuição da lei de Benford									Erosion score
	1	2	3	4	5	6	7	8	9	
1	37	6	1	8	17	14	5	4	8	0,785
2	22	28	22	12	6	4	2	1	3	0,578
3	4	7	15	23	24	15	8	3	1	0,582
4	17	17	18	16	14	8	4	3	3	0,587
5	62	0	0	1	3	9	6	6	13	0,872
6	56	20	6	1	1	1	4	4	7	0,835
7	42	31	18	4	1	0	0	1	3	0,597
8	35	5	1	12	11	15	6	5	10	0,842
9	14	16	18	14	15	12	5	3	3	0,591
10	14	17	14	13	13	13	9	4	3	0,547
11	60	14	2	0	1	8	5	4	6	0,795
12	9	12	16	17	14	17	11	3	1	0,509
13	17	19	18	12	8	9	7	6	4	0,593
14	7	60	28	0	1	2	1	0	1	0,709
15	7	33	53	5	1	0	1	0	0	0,559
16	31	63	1	0	0	1	1	1	2	0,687
17	88	2	0	0	0	2	1	2	5	0,806
18	49	14	3	1	1	6	5	7	14	0,750
19	21	14	10	7	6	22	12	4	4	0,920
20	10	22	39	24	4	0	0	0	1	0,644
21	2	3	8	12	18	24	23	9	1	0,480
22	85	0	0	0	0	2	2	2	9	0,825
23	50	15	9	8	3	2	2	2	9	0,651
24	13	25	24	19	10	5	3	0	1	0,656
25	53	11	7	8	6	3	2	2	8	0,980
26	18	20	17	11	8	15	5	2	4	0,690
27	33	13	9	7	7	20	5	2	4	0,792
28	21	22	26	15	9	3	0	1	3	0,634
29	11	14	16	14	14	12	10	7	2	0,627
30	28	7	8	10	12	17	9	4	5	0,778
Média	30,5	17,7	13,6	9,1	7,6	8,7	5,1	3,1	4,6	0,697

Fonte: elaborado pelo autor.

Tabela 9 - Resultados da geração de terrenos com Diamond-Square com filtro de melhores resultados

Número	Semente terreno	Distribuição da lei de Benford									Erosion score
		1	2	3	4	5	6	7	8	9	
1	782241903	29	22	10	10	9	5	7	5	3	0,496
2	1570264382	28	19	10	8	9	9	8	6	3	0,462
3	802178387	30	17	25	11	5	3	4	2	3	0,494
4	1249439482	29	26	17	13	4	3	3	3	2	0,475
5	1252959056	30	18	14	11	11	10	3	2	1	0,516
6	357137117	29	19	11	7	6	10	10	5	3	0,473
7	883286405	28	27	20	8	4	4	3	4	2	0,502
8	1783199010	30	22	14	10	8	7	5	2	1	0,494
9	458636383	29	21	12	14	10	4	5	3	2	0,467
10	934470309	35	27	18	8	3	2	2	2	3	0,472
11	1157147153	34	41	13	3	2	2	2	2	1	0,473
12	1121819615	29	30	17	10	6	2	2	2	2	0,472
13	452816971	30	33	15	6	7	4	2	2	1	0,494
14	143692977	28	24	15	12	8	7	3	2	1	0,493
15	1339338449	28	33	15	8	5	3	3	3	2	0,480
16	1097089324	38	34	16	3	2	2	2	2	1	0,516
17	1589794924	30	28	19	10	4	3	2	2	2	0,478
18	897777977	29	33	10	10	8	4	3	2	1	0,491
19	2008144752	28	22	23	12	5	4	2	2	2	0,482
20	1929075437	28	30	14	11	7	4	2	2	2	0,482
21	1052825779	28	38	18	5	3	2	2	2	2	0,501
22	37516232	29	29	19	9	4	4	3	2	1	0,473
23	1491957076	29	33	14	4	3	3	6	6	2	0,493
24	2011105992	37	19	100	9	7	5	5	5	3	0,488
25	1572649106	28	25	18	11	7	4	2	3	2	0,493
26	1568206995	36	31	10	4	6	4	4	3	2	0,486
27	1654979321	32	28	11	12	5	4	3	3	2	0,484
28	1028069721	28	23	9	9	9	11	6	3	2	0,476
29	1566168041	28	27	24	7	4	3	3	2	2	0,486
30	524947494	31	30	20	5	3	2	3	3	3	0,477
31	970130859	29	25	10	14	8	5	4	3	2	0,470
32	1445290123	29	21	13	10	8	7	5	4	2	0,472
33	636486175	33	24	18	9	6	4	3	1	2	0,481
34	1025666313	29	31	22	9	4	2	1	1	1	0,472
35	615004277	36	21	9	9	7	8	5	3	2	0,512
36	783494740	28	19	9	7	8	11	10	6	2	0,477
37	1864249819	30	31	17	8	4	2	2	3	3	0,471
38	1717812346	32	28	11	8	9	4	4	2	2	0,481
39	1338165074	28	17	15	11	9	6	7	4	3	0,480
40	1309631019	29	23	13	9	8	6	5	4	3	0,486
41	471851261	33	21	19	10	6	3	3	3	2	0,470
42	842497983	30	20	14	14	7	6	4	3	2	0,516
43	846601658	31	34	15	9	3	2	2	2	2	0,471
44	1769944554	29	21	13	8	8	8	6	5	2	0,460
45	592196612	31	17	9	8	15	13	4	2	1	0,470
46	751231761	30	21	12	7	8	6	7	6	3	0,501
47	523905917	44	32	10	3	3	2	2	2	2	0,472
48	897149676	28	37	15	7	6	4	2	1	0	0,481
				15	9	10	5	5	2	1	0,512
				22	11	2	1	1	1	1	0,478
				16,6	8,8	6,3	4,8	3,8	2,9	1,9	0,484

Fonte: elaborado pelo au-

