

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 naturalidade 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 à naturalidade, os terrenos gerados alcançaram resultados próximos, mas não com as mesmas características dos terrenos reais analisados, sendo a erosão hidráulica o que mais se aproximou do esperado. As métricas utilizadas para a análise de naturalidade se mostraram viáveis, porém estudos mais aprofundados se fazem necessários.

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

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 nas paisagens, 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 real, 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 gerados (MEI; DECAUDIN; HU, 2007, p. 48).

Diante do apresentado, este trabalho propõe disponibilizar 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: analisar a performance dos algoritmos implementados comparando sua execução em CPU e GPU; e avaliar a naturalidade dos terrenos gerados com base em terrenos reais. Tratando-se da análise de naturalidade 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 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação dos assuntos abordados no trabalho, estando dividido 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 naturalidade 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).

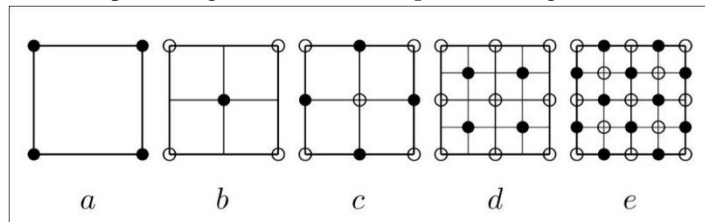
Estes resultados podem ser alcançados através do uso de técnicas procedurais que, segundo Ebert *et al.* (2003, p. 1, tradução nossa), "[...] são segmentos de código ou algoritmos que especificam alguma característica de um modelo ou efeito gerado por computador." Isto permite que características como 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 Van 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, o que ocorre quando sua largura atinge um valor menor que 3, tornando impossível de ser dividida igualmente por não possuir um ponto central (OLSEN, 2004, p. 2-3). A Figura 1 apresenta a execução de duas iterações do algoritmo, na qual o passo (a) representa a definição dos valores aleatórios para os 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 , e sua função é controlar 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; VAN 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.

2.2 MODELOS FÍSICOS E ALGORITMOS DE EROSÃO

À primeira vista terrenos gerados através de técnicas pro-ais 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 seu comportamento 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 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). Destaca-se que ambos trabalham com iterações sequenciais das operações que representam a passagem de tempo.

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* (ou *strength*) 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. Destaca-se 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. Ao utilizar d_{max} no cálculo da diferença a equação da preferência a movimentar os pontos mais íngremes primeiro, o que acredita-se ser condizente com o observado em eventos reais.

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:

- a) aparecimento de água da chuva;
- b) água da chuva dissolve a superfície do terreno transformando-a em sedimento;
- c) água e sedimento são transportados pelo fluxo de água;
- d) 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), Jáko 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*, representando a chuva em proporções iguais sob todo o terreno. Em seguida, a Equação 3 descreve

a etapa b), na qual uma quantidade de solo proporcional à quantidade de água multiplicado pela constante de solubilidade s é retirada da matriz h , representando o desprendimento de sedimentos ao longo do terreno baseado na quantidade de água presente na região.

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

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. Δa 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 Δw deve ser somado a todos os vizinhos em w e depois o total de todos os Δw deve ser subtraído da posição atual em w . Este processo simula a movimentação do excesso de água na posição atual (Δa) para todos os seus vizinhos com base na diferença entre eles (d_i). Por fim, a Equação 5 apresenta a última etapa, em que Δs corresponde tanto a quantidade de água evaporada de w quanto a quantidade de sedimentos adicionados em h (nesse caso multiplicado por s), e a variável e representa uma constante que controla o fator de evaporação.

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

$$\begin{aligned} \Delta s &= (w_{i,j} - h_{i,j}) - ((w_{i,j} - h_{i,j})(1 - e)) \\ w_{i,j} &= w_{i,j} - \Delta s \\ h_{i,j} &= h_{i,j} + s(\Delta s) \end{aligned} \quad (5)$$

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 (também utilizado no desenvolvimento). 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*, com cada *thread* sendo 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 1.025×1.025 . 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 tem-se $41 \times 25 = 1.025$ 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 1.024 e o limite de grupos é de 65.535 (KHRONOS, 2019).

Quadro 1 - Exemplo de *compute shader* para processar uma matriz de 1.025×1.025

```
// 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. Esta diferença de performance tende a aumentar conforme o tamanho dos dados a serem processados aumenta (HANGÜN; EYECIOĞLU, 2017, p. 39-40). Che *et al.* (2007) propõem o mesmo estilo de análise, porém com algoritmos de propósito geral 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 DA NATURALIDADE 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 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 quão 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 σ_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* ε é definido como a diferença entre o desvio padrão σ_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.

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

2.4.2 Lei de Benford

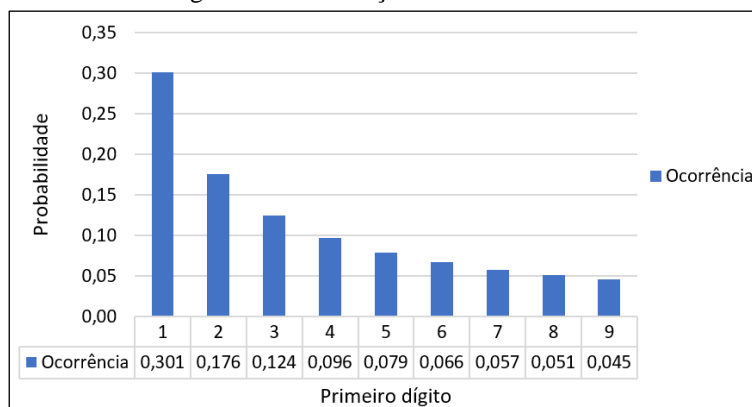
Também conhecida como lei do primeiro dígito, a lei de Benford foi observada e documentada pela primeira vez por Simon Newcomb em 1881, e novamente por Frank Benford em 1938, em homenagem a quem o fenômeno foi batizado (HILL, 1998, p. 1). A lei aponta que, em conjuntos de dados estatísticos com uma distribuição natural, 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 e 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). Por compreender diversos contextos de dados naturais, acredita-se que ela possa ser usada também para identificar a naturalidade de terrenos gerados de forma procedural.

Figura 2 - Distribuição da lei de Benford



Fonte: elaborado pelo autor.

2.5 TRABALHOS CORRELATOS

A seguir são apresentados três trabalhos que possuem características semelhantes aos principais objetivos do trabalho proposto e que utilizam algumas das técnicas apresentadas anteriormente. O Quadro 2 apresenta o trabalho de Olsen (2004) que demonstra métodos para sintetizar terrenos de aparência natural utilizando algoritmos de geração procedural e erosão, o Quadro 3 apresenta um simulador de dinâmica de relevos que utiliza algoritmos de erosão térmica e hidráulica desenvolvido por Diegoli Neto (2017) e o Quadro 4 apresenta um *framework* para modelagem de terrenos complexos desenvolvido por Peytavie *et al.* (2009).

Quadro 2 – Trabalho correlato 1

Referência	Olsen (2004).
Objetivos	Apresentar e analisar diversos métodos de geração procedural de terrenos erodidos em tempo real adequados para uso em jogos e ambientes de realidade virtual. Propõe otimizações de alguns dos métodos apresentados e uma forma de avaliar a naturalidade dos terrenos gerados.
Principais funcionalidades	Possibilita a geração procedural de terrenos com dimensões de 512x512. A geração do terreno base é feita através da combinação dos algoritmos <i>diamond-square</i> e diagrama de Voronoi. Em seguida o terreno é submetido a versões otimizadas dos algoritmos de erosão térmica e erosão hidráulica. A naturalidade do terreno final gerado é avaliada através de uma métrica proposta pelo autor chamada <i>erosion score</i> , que avalia o quão erodido o terreno está.
Ferramentas de desenvolvimento	Desenvolvido em Java (não especifica bibliotecas ou <i>frameworks</i>). Não menciona uso de processamento em GPU.
Resultados e conclusões	Os terrenos gerados foram adequados para a finalidade do autor de serem utilizados em jogos. As versões otimizadas dos algoritmos alcançaram o requisito de processamento em tempo real e realismo esperados. O algoritmo de erosão térmica mostrou-se rápido o suficiente, mas não atingiu o nível de naturalidade do terreno esperado. Já o algoritmo de erosão hidráulica atingiu um nível de naturalidade adequado, mas se mostrou mais lento que o desejado.

Fonte: elaborado pelo autor.

Quadro 3 - Trabalho correlato 2

Referência	Diegoli Neto (2017).
Objetivos	Desenvolver uma ferramenta para simulação de deslizamento de terra em tempo real e de forma simplificada o suficiente para permitir que usuários sem conhecimento técnico compreendam os processos.
Principais funcionalidades	Realiza a transformação do relevo para simular processos de deslizamento de terra e chuva através de versões modificadas dos algoritmos de erosão térmica e hidráulica apresentados por Olsen (2004). Os algoritmos trabalham com quatro camadas de solo diferentes (solo exposto, grama, floresta e concreto), permitindo que os processos de erosão afetem o terreno de forma diferente dependendo da camada existente naquela posição.
Ferramentas de desenvolvimento	Desenvolvido em C# utilizando o motor gráfico Unity. Não realiza processamento em GPU.
Resultados e conclusões	A ferramenta apresenta uma visualização condizente com a realidade, mesmo não sendo capaz de oferecer uma representação precisa dos processos simulados. O autor destaca que a ferramenta sofreu alguns problemas de baixa performance, possivelmente devido a limitações da Unity em trabalhar com atualizações constantes do relevo. Não foram apresentadas análises referentes à naturalidade dos terrenos gerados.

Fonte: elaborado pelo autor.

Quadro 4 - Trabalho correlato 3

Referência	Peytavie <i>et al.</i> (2009).
Objetivos	Apresentar um <i>framework</i> para criação de terrenos com características de relevo complexas como saliências, arcos e cavernas.
Principais funcionalidades	Permite que o usuário trabalhe com materiais como pedra, rocha, areia, água e ar distribuídos em diferentes camadas de solo. Essa organização torna possível representar estruturas como cavernas e arcos. O terreno é primeiro modelado pelo usuário e depois submetido a algoritmos de simulação de estabilização de materiais, criando assim cenários mais próximos do real. Também possibilita criar rachaduras, redes de túneis, pilhas de rochas e erodir superfícies.
Ferramentas de desenvolvimento	Não especificado. Não menciona uso de processamento em GPU.
Resultados e conclusões	Os autores afirmam terem apresentado uma abordagem original e eficaz para a representação de estruturas complexas em terrenos, fornecendo ferramentas que possibilitam a modelagem de estruturas impossíveis de serem representadas em mapas de altura convencionais. Não foram apresentadas análises referentes à naturalidade dos terrenos gerados.

Fonte: elaborado pelo autor.

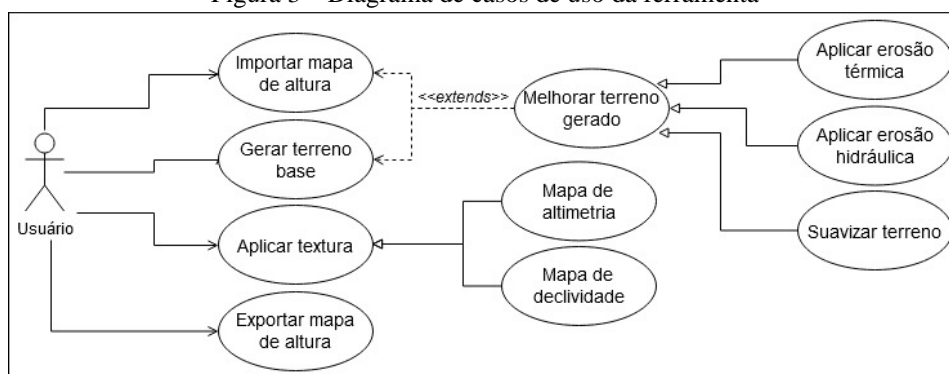
3 DESCRIÇÃO DA FERRAMENTA

Este capítulo apresenta a especificação da ferramenta com suas principais funcionalidades. A seção 3.1 apresenta os principais aspectos de usabilidade da ferramenta, as funcionalidades disponíveis para o usuário, o fluxo de execução de algumas etapas e a interface gráfica a partir da qual o usuário irá interagir com a ferramenta. A seção 3.2 descreve a implementação de três dos principais algoritmos utilizados pela ferramenta, sendo eles o *diamond-square*, erosão térmica e erosão hidráulica. A ferramenta, código fonte e documentação complementar sobre seu uso estão disponíveis no GitHub no repositório de Gonçalves (2020).

3.1 ESPECIFICAÇÃO

Para o desenvolvimento da ferramenta foi utilizado o motor gráfico Unity, sendo que a estruturação geral e os algoritmos executados em CPU foram desenvolvidos na linguagem de programação C#, enquanto os algoritmos executados em GPU foram desenvolvidos na linguagem HLSL. A ferramenta desenvolvida pode ser utilizada de duas formas diferentes: através de um executável para desktop ou através de um *plugin* para a Unity (conhecido como *asset*). O executável desktop é voltado para o usuário final que busca gerar um terreno e exportá-lo a fim de utilizá-lo em outra aplicação, enquanto o *plugin* destina-se a desenvolvedores que estejam trabalhando com projetos Unity e que necessitem de terrenos para cenários ou simulações de ambiente. Em ambos os casos o fluxo de execução da ferramenta e suas funcionalidades se mantêm iguais, com a única diferença sendo a interface a partir da qual o usuário irá interagir. A Figura 3 demonstra o diagrama de casos de uso da ferramenta com suas principais funcionalidades.

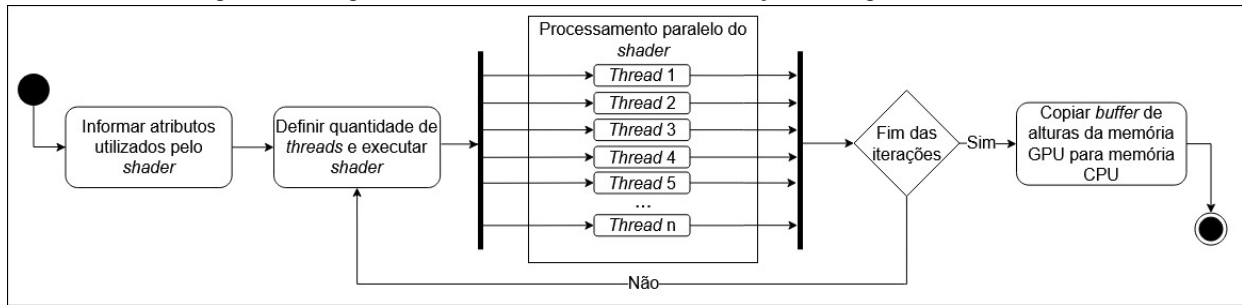
Figura 3 – Diagrama de casos de uso da ferramenta



Fonte: elaborado pelo autor.

De todas as funcionalidades apresentadas, as únicas não possíveis de serem executadas em GPU são a importação e exportação do terreno gerado. A geração do terreno base e a aplicação dos algoritmos de erosão podem ser executados tanto em CPU com algoritmos em C# como em GPU com *compute shaders* em HLSL. A texturização do terreno ocorre também em GPU, porém através de um *surface shader* que modifica a cor da posição atual com base em sua altura no caso do mapa de altimetria ou inclinação no caso do mapa de declividade. A aplicação de erosão no terreno pode ser feita através dos processos de erosão térmica ou hidráulica. Por fim, a exportação e importação de terrenos gerados são feitas através de uma imagem do mapa de altura do terreno em escala de cinza, comumente conhecida como *heightmap* e amplamente utilizada por outras ferramentas, o que possibilita a reutilização do terreno gerado em outros ambientes. Os algoritmos executados em GPU com *compute shaders* possuem um fluxo de execução semelhante entre si, que pode ser observado na Figura 4.

Figura 4 – Diagrama de atividades do fluxo de execução dos algoritmos em GPU

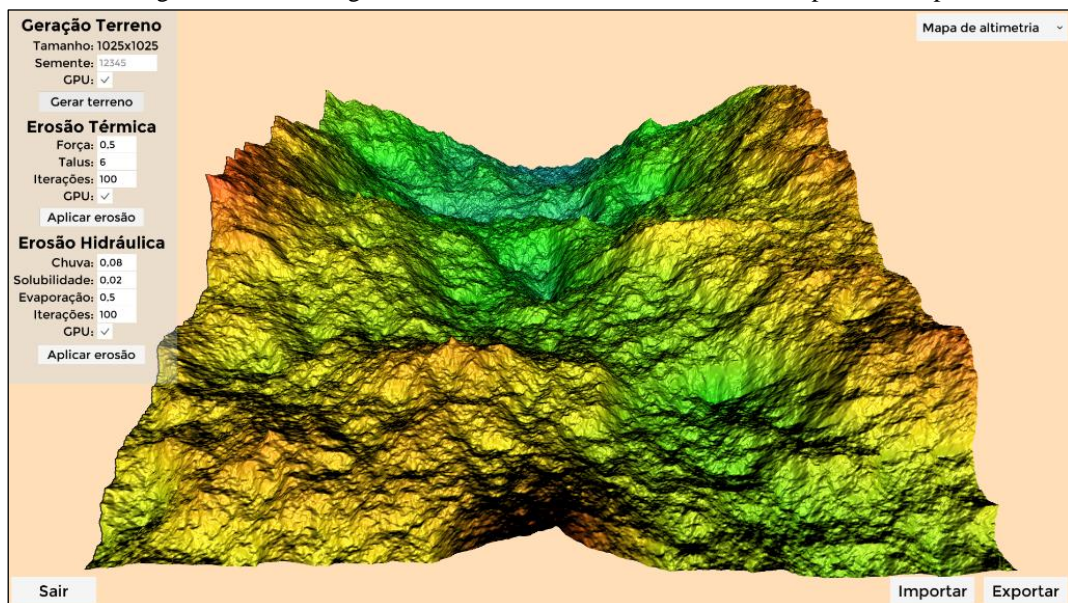


Fonte: elaborado pelo autor.

Em todos os casos é necessário primeiro informar para o *shader* quais valores serão utilizados durante o processamento em GPU, que serão copiados da memória em CPU para a memória compartilhada da GPU. Em seguida deve-se definir a quantidade de *threads* a serem utilizadas e realizar o *dispatch* do *shader*. O processo de *dispatch* trata de copiar os valores informados para a memória em GPU e iniciar a execução paralela do *shader*. As duas primeiras etapas do fluxo são implementadas em C# e executadas pela CPU, sendo apenas a etapa de processamento do *shader* executada em GPU. Finalizado o processamento de todas as *threads*, o fluxo de execução retorna para o código C# que fica encarregado de repetir o processo ou, caso as iterações tenham finalizado, recuperar da memória em GPU as informações processadas pelo *shader*. Tanto o algoritmo de geração do terreno base como os de erosão trabalham com iterações sequenciais das mesmas operações, em que para cada iteração é realizado um novo *dispatch*. As informações processadas pelo *shader* são recuperadas apenas após o término de todas as iterações.

Como mencionado anteriormente, a ferramenta pode ser utilizada como um executável independente ou como um *plugin* na Unity. Cada versão possui uma interface gráfica diferente para permitir a interação do usuário, porém ambas possuem as mesmas opções e configurações, apenas apresentadas de forma diferente. A Figura 5 apresenta a interface gráfica da ferramenta na versão executável. À esquerda é possível observar o menu com todas as opções configuráveis para geração dos terrenos e ao fundo é exibido o terreno gerado texturizado com o mapa de altimetria. O mapa de altimetria pode ser alterado para o de declividade a partir do menu *dropdown* na direita superior. Além disso, observa-se os botões *Importar* e *Exportar* na direita inferior para importar e exportar mapas de altura de terrenos como imagens, além do botão *Sair* na esquerda inferior para fechar a ferramenta. As propriedades do menu esquerdo correspondem a variáveis dos algoritmos utilizados, cujas utilidades foram explicadas na seção 2. Ao posicionar o mouse acima de qualquer uma delas, o usuário verá uma breve explicação sobre sua utilidade. Por fim, cada operação presente no painel lateral pode ser executada em GPU ou CPU, sendo controlável pelo campo GPU, ativado por padrão.

Figura 5 - Interface gráfica da ferramenta na versão executável para desktop

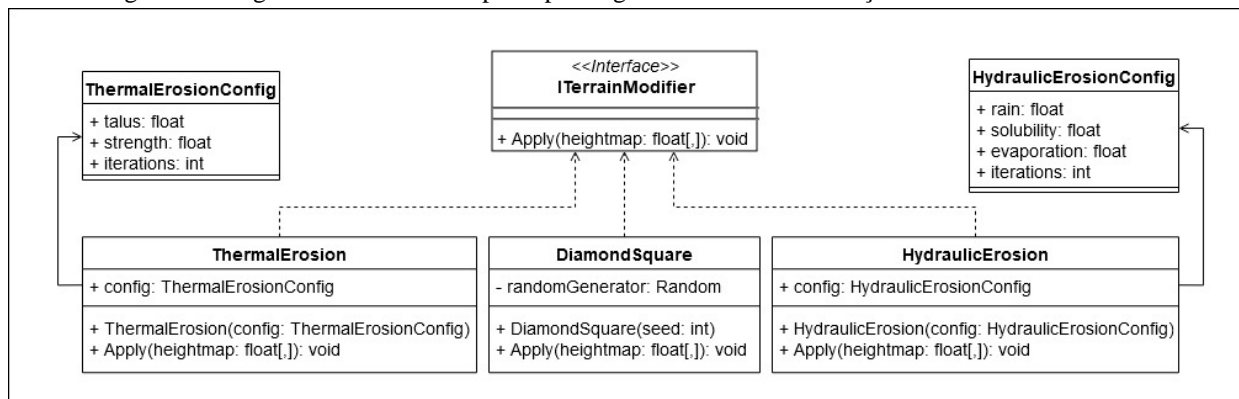


Fonte: elaborado pelo autor.

Todos os algoritmos que modificam o relevo implementam a interface *ITerrainModifier*, que disponibiliza um único método *Apply* que recebe como parâmetro o mapa de altura do relevo (matriz de números decimais) e o modifica como quiser. A padronização da chamada dos algoritmos a partir de um mesmo método possibilita uma melhor 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 responsáveis pela 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 1.025x1.025. Ao ser transferida para os algoritmos em *compute shader* esta matriz passa a ser representada por um vetor unidimensional com tamanho 1025², 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 *InitComputeShaderProperties* chamado na primeira linha do Quadro 5, que retorna como resultado o id do *kernel (shader)* a ser executado.

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;
}
  
```

Fonte: elaborado pelo autor.

A variável H inicia com o valor 1 e a variável `squareSize` com a largura total do terreno, sendo que no final de cada iteração ambas terão seus valores divididos pela metade. Como a largura e altura do terreno são sempre iguais, `squareSize` representa ambas, ou seja, dividi-la pela metade faz com que o terreno original seja subdividido em quatro. A variável `numThreads` representa a quantidade de *threads* em x e y a serem utilizadas pelo *shader* a cada iteração, sendo que cada quadrado deve ser processado por uma *thread* diferente. Como a quantidade de quadrados a serem processados quadruplica a cada iteração (causado pela divisão de `squareSize`), a quantidade de *threads* necessárias deve refletir este número. Como as *threads* no *compute shader* são separadas em x e y e a variável `numThreads` representa ambos os eixos, basta multiplicá-la por dois a cada iteração para englobar todos os quadrados. Por fim, a função `RunComputeShader` trata de iniciar o *compute shader* contendo a lógica que será paralelizada em GPU para cada iteração, informando o número de *threads* a serem utilizadas e os valores atualizados das variáveis `squareSize` e H .

No *compute shader*, o funcionamento correto do algoritmo depende fortemente da navegação entre os pontos do terreno. Cada *thread* possui acesso a todo o terreno (representado aqui por um vetor unidimensional) mas deve trabalhar apenas dentro dos limites do seu quadrado (definidos por `squareSize`). Como cada *thread* possui um `id` incremental, é possível identificar o início de cada quadrado através das expressões `id.x * squareSize` para a coluna e `id.y * squareSize` para a linha. Além disso, para navegar entre as linhas e colunas do *heightmap* é necessário sempre multiplicar a posição atual por `width + 1`. Por exemplo, `id.x * squareSize` retorna a coluna em relação ao quadrado atual, enquanto `id.x * squareSize * (width + 1)` retorna a coluna em relação ao terreno inteiro. A partir destas informações é possível recuperar todos os pontos necessários para executar as etapas *diamond step* e *square step* do algoritmo explicadas na seção 2.1. O Quadro 6 apresenta as expressões necessárias para conseguir cada uma das coordenadas e a Figura 7 demonstra estas mesmas coordenadas dentro de um dos quadrados de um terreno subdividido em quatro (o que representaria a segunda iteração do algoritmo).

Quadro 6 – Expressões para recuperar coordenadas necessárias para as etapas do *diamond-square*

```
uint col = id.x * squareSize;
uint row = id.y * squareSize;

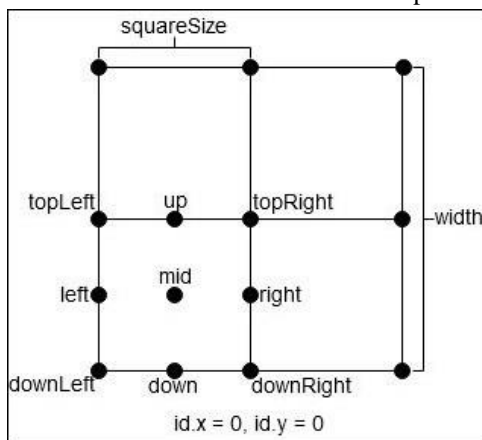
uint rowSize = width + 1;
uint halfSize = squareSize / 2;

uint mid = (row + halfSize) * rowSize + (col + halfSize);
uint topLeft = row * rowSize + col;
uint topRight = row * rowSize + (col + squareSize);
uint bottomLeft = (row + squareSize) * rowSize + col;
uint bottomRight = (row + squareSize) * rowSize + (col + squareSize);

uint up = topLeft + halfSize;
uint down = bottomLeft + halfSize;
uint left = mid - halfSize;
uint right = mid + halfSize;
```

Fonte: elaborado pelo autor.

Figura 7 – Exemplo de subdivisão com coordenadas necessárias para as etapas do *diamond-square*



Fonte: elaborado pelo autor.

Tendo todas as coordenadas, resta apenas realizar as etapas *diamond step*, na qual o ponto `mid` recebe a média dos pontos `topLeft`, `topRight`, `bottomLeft` e `bottomRight` somada a um valor aleatório entre 0 e 1 multiplicado por H ; e *square step*, na qual os pontos `up`, `down`, `left` e `right` recebem a média de seus vizinhos somada novamente a um valor aleatório entre 0 e 1 multiplicado por H . Estas etapas são apresentadas no Quadro 7 (por questões de visibilidade do

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

Os algoritmos de erosão possuem uma lógica de paralelismo relativamente mais simples que o *diamond-square* apresentado na seção anterior. Todos trabalham com iterações, porém, diferente do *diamond-square*, a cada iteração os algoritmos de erosão são aplicados igualmente a todas as posições do terreno. Ou seja, o número de *threads* utilizadas em cada iteração é sempre o mesmo, o que facilita sua implementação neste quesito. Como o terreno utilizado possui dimensões de 1.025x1.025, optou-se por utilizar a configuração de 41 grupos de *threads* em *x* e *y* com 25 *threads* em *x* e *y* cada ($41 \times 25 = 1.025$). 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.

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* necessários 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.

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 (d > talus) {
        dTotal += di;
        if (di > dMax)
            dMax = d;
    }
}
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* para 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 compreendendo as duas primeiras etapas e dois compreendendo 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. As alterações no terreno são realizadas através da manipulação dos vetores `heightmap` que representa o terreno e `watermap` que representa a água da chuva sob o terreno. O Quadro 9 apresenta o *shader* responsável pelas duas primeiras etapas (Equação 2 e 3), em que primeiro soma-se o valor da constante `rainFactor` a todas as posições de `watermap`, representando a chuva sob o terreno. Depois, subtrai-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 trata apenas de percorrer todos os vizinhos da posição atual coletando a diferença total entre alturas (`dTotal`) e a média de altura dos vizinhos (`avgNeighborsHeight`). O segundo loop 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 avgNeighborsHeight = 0;
int countHeights = 0;
float dTotal = 0;

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

avgNeighborsHeight /= countHeights;
float deltaSurfaceHeight = watermap[currentPosition] - avgNeighborsHeight;
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 (seção 2.2). 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).

Quadro 11 - Etapa d) do algoritmo de erosão hidráulica em *compute shader*

```
uint currentPosition = id.x + (id.y * width);
float evaporationPercent = 1 - evaporationFactor;

float waterVolume = watermap[currentPosition] - heightmap[currentPosition];
float deltaSediment = waterVolume - (waterVolume * evaporationPercent);

watermap[currentPosition] = watermap[currentPosition] - deltaSediment;
heightmap[currentPosition] = heightmap[currentPosition] + (solubility * deltaSediment);
```

Fonte: elaborado pelo autor.

4 RESULTADOS

Este capítulo apresenta os resultados alcançados a partir do trabalho desenvolvido, focando nos algoritmos *diamond-square*, erosão térmica e erosão hidráulica por serem os mais relevantes na geração do terreno. Para a análise foram considerados 50 terrenos gerados pelo *diamond-square* e melhorados pelos algoritmos de erosão térmica e hidráulica. A fim de facilitar a leitura dos resultados, as tabelas e gráficos apresentados neste capítulo considerarão apenas a média geral dos terrenos analisados. Porém, os dados completos com a lista de todos os 50 terrenos analisados, suas sementes e resultados alcançados estão disponíveis no Apêndice B, assim como os parâmetros utilizados em cada algoritmo. A seção 4.1 analisa os algoritmos do ponto de vista de performance, levando em consideração o tempo de execução de cada um. Em seguida, a seção 4.2 analisa os algoritmos considerando a naturalidade dos terrenos gerados, utilizando como métricas o *erosion score* (OLSEN, 2004) e a lei de Benford (BENFORD, 1938), ambas apresentadas nas seções 2.4.1 e 2.4.2.

4.1 ANÁLISE DE PERFORMANCE

Para a análise de performance dos algoritmos implementados considerou-se unicamente o tempo de execução de cada um. Cada teste foi realizado tanto com os algoritmos em GPU como com suas respectivas versões em CPU a fim de permitir uma comparação não apenas entre os algoritmos, mas também entre as plataformas utilizadas. Para possibilitar uma análise com diferentes quantidades de dados, cada algoritmo foi aplicado a terrenos com dimensões de 65x65, 129x129, 257x257, 513x513 e 1.025x1.025. Todos os testes foram realizados em um computador Windows 10 Pro 64 bits com processador AMD A10-785K APU, 8 GB de RAM e placa de vídeo AMD Radeon R9 200 Series. A distribuição de *threads* utilizada pelos algoritmos em GPU por cada dimensão está disponível no Quadro 14 do Apêndice B.

A Tabela 1 apresenta a média de tempo em milissegundos (ms) das 50 execuções do algoritmo *diamond-square* em CPU e GPU em cada uma das dimensões mencionadas. Na Tabela 1 também é exibida a quantidade de *threads* utilizadas para cada uma das dimensões de terrenos testados, sendo que como o algoritmo trabalha com a subdivisão recursiva do terreno, a quantidade de *threads* utilizadas quadriplica a cada iteração (duplica em *x* e *y*). A partir dos resultados é possível observar que em ambos os casos o tempo de execução aumenta gradativamente conforme o tamanho do terreno aumenta, variando de 0,02 a 131 ms no caso da execução em CPU e de 0,62 a 26 ms em GPU, o que indica que a versão em GPU possui melhor performance em todos os casos com exceção do primeiro (provavelmente devido a baixa quantidade de *threads* utilizadas nele).

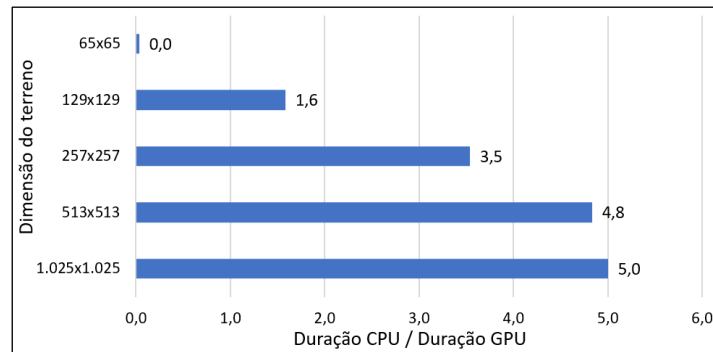
Tabela 1 - Média em milissegundos de 50 execuções do *diamond-square* em CPU e GPU

Dimensões	CPU (ms)	GPU (ms)	Nº total <i>threads</i>
65x65	0,02	0,62	127
129x129	1,36	0,86	255
257x257	7,36	2,08	511
513x513	31,6	6,54	1.023
1.025x1.025	131,24	26,24	2.047

Fonte: elaborado pelo autor.

Além da maior velocidade do algoritmo em GPU, também é possível observar que a diferença de execução entre CPU e GPU (tempo de execução em CPU dividido pelo tempo de execução em GPU) aumenta conforme o tamanho do terreno aumenta. No gráfico da Figura 8 é possível observar que a versão em GPU torna-se 1,6 vezes mais rápida nos terrenos de 129x129 e aumenta gradativamente chegando a ser 5 vezes mais rápida para os terrenos de 1.025x1.025. Estes resultados refletem o comportamento esperado de algoritmos paralelizados em que seu uso se torna mais viável conforme a quantidade de dados a serem processados (e a quantidade de *threads* utilizadas) aumenta. Por mais que o ganho de performance seja visível, ainda assim é uma melhoria pequena se comparada aos algoritmos de erosão que serão apresentados a seguir. Acredita-se que este baixo ganho de performance deve-se ao fato de o *diamond-square* não possuir um número constante de *threads* a serem executadas, sendo que o número aumenta a cada iteração e, mesmo quando considerado a quantidade total de *threads* utilizadas pelo algoritmo (como exibido na última coluna da Tabela 1), ainda assim tem-se um número relativamente pequeno para justificar o uso de GPU se comparado com os algoritmos de erosão.

Figura 8 - Diferença entre tempo de execução em CPU e GPU do *diamond-square*



Fonte: elaborado pelo autor.

Os testes de performance para os algoritmos de erosão térmica e hidráulica foram executados nos mesmos terrenos gerados pelo *diamond-square* com as cinco dimensões diferentes. Porém, como estes algoritmos trabalham com iterações repetitivas das operações, também foram executados com cinco quantidades diferentes de iterações, sendo: 100, 200, 300, 400 e 500 iterações. A Tabela 2 apresenta os resultados destas execuções para o algoritmo de erosão térmica. Nela é possível observar que em todos os casos o tempo da versão em CPU aumenta entre quatro e cinco vezes conforme as dimensões do terreno aumentam, além de aumentar gradativamente a cada 100 iterações, porém a diferença diminui cada vez mais conforme as iterações aumentam. Já no caso da GPU o tempo de execução tende a menos que dobrar conforme as iterações e dimensões aumentam, com exceção da terceira e última dimensões que possuem um intervalo acima das outras.

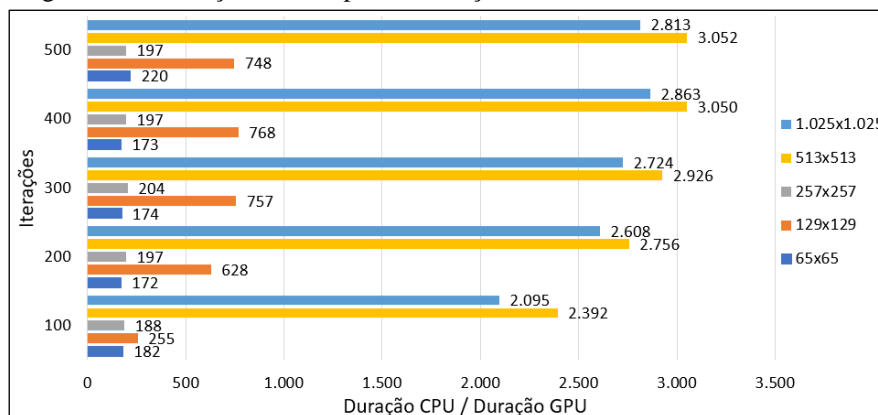
Tabela 2 - Média em milissegundos da execução da erosão térmica em CPU e GPU

Dimensões	100 iterações		200 iterações		300 iterações		400 iterações		500 iterações	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
65x65	182	1	344	2	523	3	692	4	880	4
129x129	765	3	1.508	2	2.271	3	3.070	4	3.739	5
257x257	2.636	14	5.312	27	8.171	40	10.430	53	12.984	66
513x513	9.567	4	19.290	7	29.261	10	39.656	13	48.828	16
1.025x1.025	37.706	18	75.620	29	111.701	41	151.762	53	180.006	64

Fonte: elaborado pelo autor.

O gráfico da Figura 9 apresenta a diferença de tempo entre as versões em CPU e GPU. Nele é possível observar que a diferença se mantém pequena ao longo das diferentes iterações, porém aumenta drasticamente conforme a dimensão dos terrenos também aumenta, chegando ao ponto de a versão em GPU ser até 3.052 vezes mais rápida que a versão em CPU no caso dos terrenos com dimensão de 513x513, nos quais a versão em GPU levou 16 milissegundos enquanto a versão em CPU levou 48.828 (aproximadamente 49 segundos).

Figura 9 - Diferença entre tempo de execução em CPU e GPU da erosão térmica



Fonte: elaborado pelo autor.

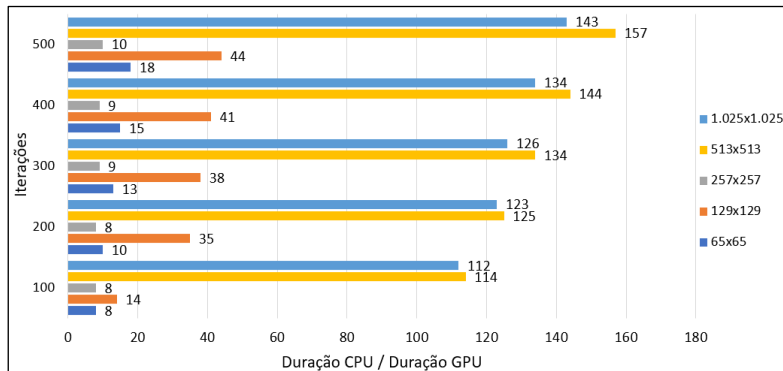
No caso do algoritmo de erosão hidráulica é possível observar um comportamento muito semelhante ao erosão térmica, com a diferença de que a versão em CPU do erosão hidráulica demonstra ser mais rápida que a versão em CPU do erosão térmica, enquanto na versão em GPU o cenário se inverte, com o erosão 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 1.025x1.025.

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

	100 iterações		200 iterações		300 iterações		400 iterações		500 iterações	
Dimensões	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	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	1.362	164	2.054	241	2.711	312	3.601	376
513x513	2.848	25	5.620	45	8.645	64	11.796	82	15.407	98
1.025x1.025	10.955	98	21.830	178	32.360	256	43.879	328	55.879	391

Fonte: elaborado pelo autor.

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



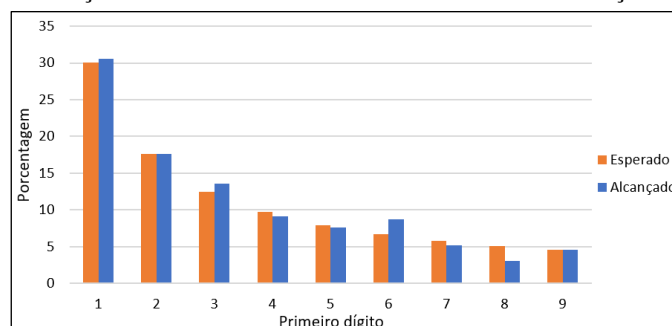
Fonte: elaborado pelo autor.

A partir destes resultados observa-se que as versões em GPU dos três algoritmos foram mais eficientes que as 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 à sua versão em CPU, o que indica que mesmo sendo o mais rápido ele obteve o menor ganho de performance. O erosão térmica obteve a maior diferença entre 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, acredita-se que esta diferença se deve ao fato de o erosão hidráulica executar três *shaders* sequencialmente, enquanto o erosão térmica utiliza apenas um. Destaca-se também o fato de os terrenos com dimensões de 513x513 terem performado melhor que os de 1.025x1.025, o que neste caso provavelmente deve-se ao fato de que os terrenos de 1.025x1.025 atingem o limite máximo do hardware utilizado, gerando um atraso no processamento.

4.2 ANÁLISE DE NATURALIDADE

Para avaliar a naturalidade 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 aplicadas 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.

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

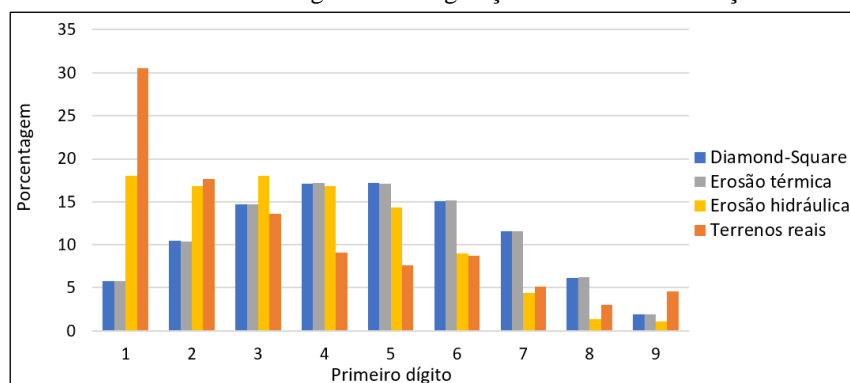


Fonte: elaborado pelo autor.

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. 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.

Em relação aos 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 1.025x1.025. O gráfico da Figura 12 apresenta os resultados alcançados ao aplicar a lei de Benford nos terrenos gerados pelo *diamond-square* e melhorados pela erosão térmica e 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.

Tabela 4 - *Erosion score* dos três algoritmos de geração e dos terrenos reais coletados

Modelos	Erosion score
Diamond-square	0,431
Erosão térmica	0,403
Erosão hidráulica	0,487
Terrenos reais	0,697

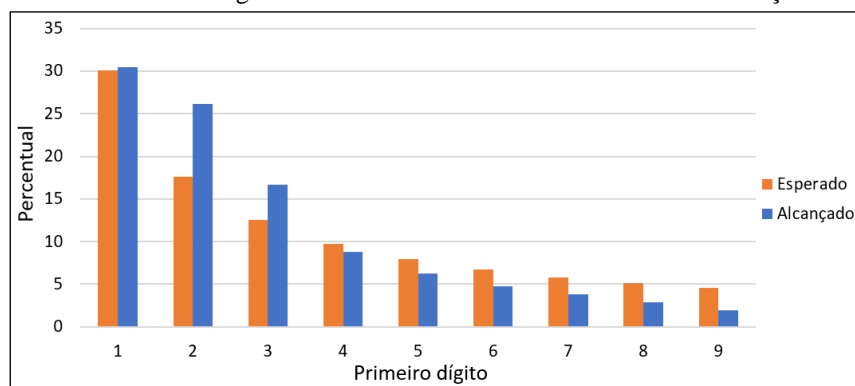
Fonte: elaborado pelo autor.

Outro ponto observável a partir dos resultados da Figura 12 e Tabela 4 é que o algoritmo de erosão térmica causou mudanças insignificantes na distribuição dos números, além de ter decrementado o *erosion score* ao invés de aumentá-lo como esperado. Este comportamento é curioso pois mostra que o algoritmo de erosão térmica utilizado, no melhor dos casos, não causa qualquer mudança significativa no terreno e no pior dos casos pode até prejudicar sua naturalidade, visto que o *erosion score* diminuiu após a execução. Assumindo que os dados coletados estejam corretos, isto pode indicar que, por mais que algumas técnicas apresentem um resultado visual convincente de melhora na naturalidade do terreno, a análise visual por si só não é suficiente para garantir que as características do terreno gerado se assemelhem às de terrenos reais, ou, em outras palavras, um terreno gerado que pareça natural não necessariamente possui características naturais.

A partir dos resultados apresentados é possível concluir que os melhores resultados, tanto da lei de Benford como do *erosion score*, foram alcançados através da combinação do *diamond-square* para gerar o terreno base e o erosão

hidráulica para melhorar o terreno gerado. Porém, como a diferença entre o resultado alcançado e esperado ainda é considerável, tentou-se uma nova abordagem utilizando ainda os dois algoritmos, porém com a diferença de que apenas as melhores gerações do *diamond-square* são consideradas e passadas adiante para a etapa de erosão hidráulica. Esta seleção das melhores gerações é realizada através de um filtro baseado na lei de Benford que verifica o percentual de primeiros dígitos 1. Como a ocorrência esperada deste dígito é de aproximadamente 30%, o algoritmo trata de selecionar apenas os terrenos que possuem um percentual igual ou superior a 28%. O gráfico da Figura 13 apresenta os resultados da lei de Benford alcançados a partir desta nova versão do algoritmo. Nele é possível observar que a distribuição dos dígitos se assemelha mais ao esperado. Além disso, observa-se que a distribuição respeita o padrão de crescimento previsto pela lei, algo que não acontece na distribuição do algoritmo original da Figura 12. Por outro lado, o *erosion score* não apresentou mudanças significativas, alcançando um valor médio de 0,484.

Figura 13 - Lei de Benford do algoritmo com filtro de melhores resultados em relação a terrenos reais



Fonte: elaborado pelo autor.

Por mais que os resultados de naturalidade alcançados pela combinação dos algoritmos *diamond-square* com filtro de 28% e erosão hidráulica pareçam satisfatórios, esta abordagem possui dois problemas a serem considerados. O primeiro problema é referente à performance do algoritmo. Como a abordagem não faz uso de qualquer heurística, baseando-se puramente na repetição do algoritmo até encontrar um resultado que satisfaça o critério de 28%, o tempo de execução do algoritmo aumenta consideravelmente, podendo variar entre 2 segundos nos melhores casos a até 6 minutos nos piores. O alto tempo de execução combinado com o intervalo de variação imprevisível inviabilizam esta abordagem para casos em que se deseje uma geração em tempo de execução (tempo real).

O segundo ponto negativo desta abordagem refere-se à distribuição das alturas em relação a lei de Benford. Por mais que a distribuição média dos 50 terrenos analisados apresentada na Figura 13 respeite consideravelmente a lei de Benford, a análise individual da distribuição de cada terreno apresenta uma grande diferença dos resultados alcançados pelos terrenos reais. Enquanto os terrenos reais, analisados individualmente, apresentam grande variação nos resultados entre si, os terrenos gerados através do algoritmo com filtro possuem uma variação mais reduzida. Por exemplo, ao analisar o percentual de primeiro dígito 1 dos terrenos reais apresentados na Tabela 5, é possível observar que, por mais que a média final deste dígito seja 30,5%, os valores individuais de cada terreno variam entre 4% e 88%, sendo que a mesma variação também é observada nos outros dígitos. Nos dados coletados da combinação original do *diamond-square* 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 naturalidade 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 respeitam 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 naturalidade 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 naturalidade dos terrenos gerados em comparação com terrenos reais, com as métricas utilizadas sendo o *erosion score* (OLSEN, 2004) e a 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. O objetivo geral de disponibilizar uma ferramenta para geração de terrenos com aparência natural utilizando GPU foi alcançado, porém com limitações referentes à naturalidade dos terrenos gerados, com o executável da ferramenta para desktop e o *plugin* para a Unity 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. De forma complementar, 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).

Em relação aos aspectos de performance da ferramenta, as versões em GPU dos três algoritmos alcançaram os objetivos desejados, sendo mais eficientes que suas versões em CPU. Considerando as versões em GPU, o *diamond-square* teve o menor tempo de execução, com duração máxima de 26 milissegundos. Seguido do erosão térmica com 64 milissegundos e o erosão hidráulica com 391 milissegundos. 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 (180.006 milissegundos), o ganho alcançado com os algoritmos em GPU se mostrou significativo.

As métricas de naturalidade 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 diminuísse, 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 para mostrar que o *erosion score* talvez 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.

Referente à naturalidade dos terrenos gerados, pode-se concluir que o objetivo de gerar terrenos com aparência natural foi desenvolvido, mas não totalmente alcançado. Ao submeter os terrenos sob avaliação das métricas de naturalidade 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 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”.

Por mais que existam diversas pesquisas, propostas de ferramentas ou melhorias relacionadas a processos de geração procedural de terrenos, a análise de naturalidade 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 naturalidade 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 entre si 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 deva-se à falta de pontos de referência no terreno);
- 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. Mesmo a execução destes algoritmos tendo atingido os resultados esperados, é possível dedicar mais esforços a fim de otimizar sua execução através de técnicas de *branchless programming*, gerencia de memória compartilhada e melhor organização de *threads*.

REFERÊNCIAS

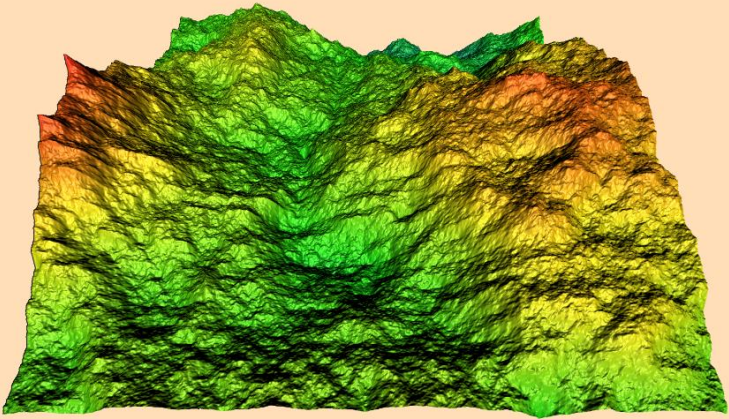
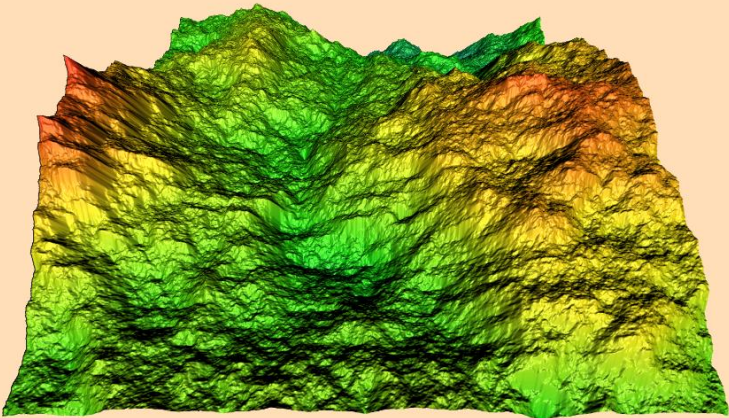
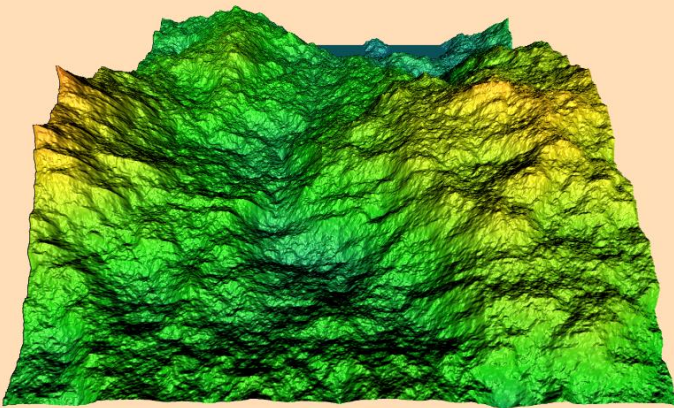
- BENFORD, Frank. The law of anomalous numbers. **Proceedings of The American Philosophical Society**, Philadelphia, v. 78, n. 4, p. 551-572, mar. 1938.
- BENEŠ, Bedřich; FORSBACH, Rafael. Visual simulation of hydraulic erosion. **Journal of WSCG**, West Bohemia. v. 10, p. 79–86, fev. 2002.
- BEYER, 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.l.], 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.l.], 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.
- GÉNEVAUX, Jean-David *et al.* Terrain generation using procedural models based on hydrology. **ACM Transactions on Graphics**, New York, v. 32, n. 4, p. 1-13, jul. 2013.
- GONÇALVES, Alex S. **Geração procedural de terrenos com aparência natural**. Blumenau, 2020. Disponível em: <<https://github.com/AlexSerodio/procedural-terrain-generation>>. Acesso em: 29 de nov. de 2020.
- GOSWAMI, Prashant; MARKOWICZ, Christian; HASSAN, Ali. Real-time particle-based snow simulation on the GPU. In: EUROGRAPHICS SYMPOSIUM ON PARALLEL GRAPHICS AND VISUALIZATION, 2019, Porto. **Proceedings...** Goslar: Eurographics Association, 2019. p. 49-57.
- HANGÜN, Batuhan; EYECIOĞLU, Önder. Performance Comparison Between OpenCV Built in CPU and GPU Functions on Image Processing Operations. **International Journal of Engineering Science and Application**, Istanbul, v. 1, n. 2, p. 35-41, jul. 2017.
- HARRIS, Mark J. *et al.* Simulation of cloud dynamics on graphics hardware. In: ACM SIGGRAPH/EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE, 3., 2003, San Diego. **Proceedings...** Goslar: Eurographics Association, 2003. p. 92-101.
- HEERMANN, Dieter W. **Computer Simulation Methods in Theoretical Physics**. 2. ed. Heidelberg: Springer, 1990.
- HILL, Theodore P. The first digit phenomenon: A century-old observation about an unexpected pattern in many numerical tables applies to the stock market, census statistics and accounting data. **American Scientist**, [S.l.], v. 86, n. 4, p. 358-363, 1998.
- ISHIDA, Daichi; ANDO, Ryoichi; MORISHIMA, Shigeo. GPU Smoke Simulation on Compressed DCT Space. In: EUROGRAPHICS, 40., 2019, Genoa. **Proceedings...** Goslar: Eurographics Association, 2019. p. 5-8.

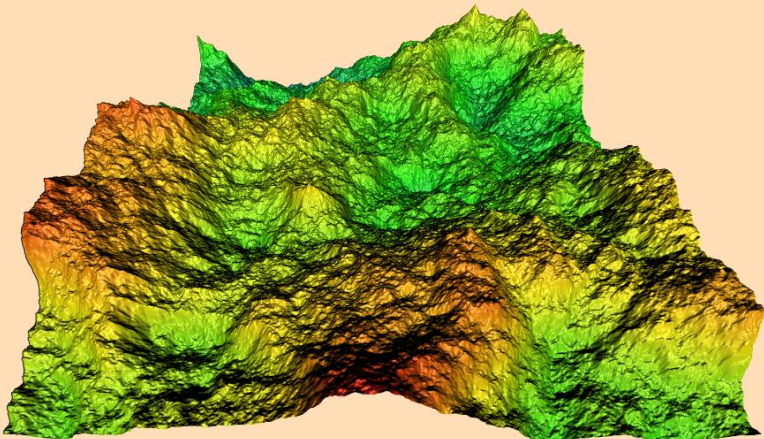
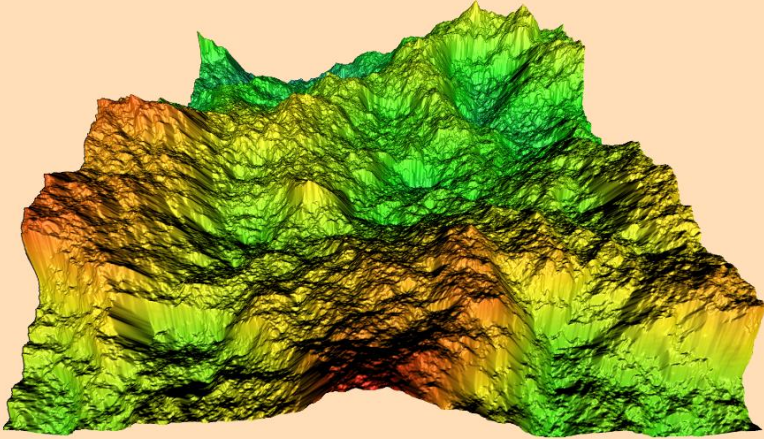
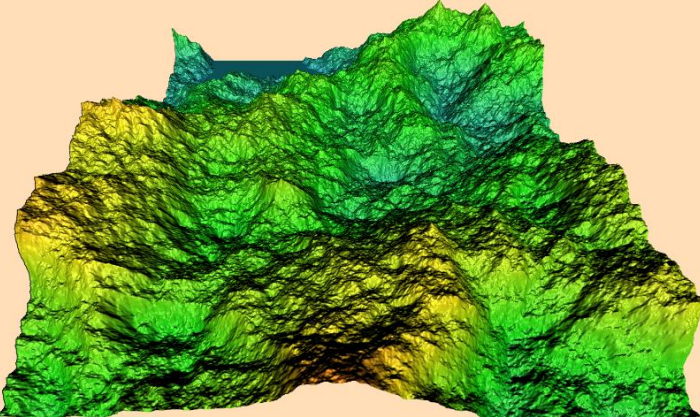
- JÁKÓ, Balázs; TÓTH, Balázs. Fast Hydraulic and Thermal Erosion on the GPU. In: 15TH CENTRAL EUROPEAN SEMINAR ON COMPUTER GRAPHICS, 15., 2011, Vinicné. **Proceedings...** Viena: TU Wien, 2011. p. 139-146.
- KELLY, George; MCCABE, Hugh. Citygen: An Interactive System for Procedural City Generation. In: INTERNATIONAL CONFERENCE ON GAME DESIGN AND TECHNOLOGY, 5., 2007, Liverpool. **Proceedings...** Liverpool: Liverpool John Moores University, 2007. p. 8-16.
- KHRONOS. **Compute Shader**. Beaverton, 2019. Disponível em: <https://www.khronos.org/opengl/wiki/Compute_Shader>. Acesso em: 3 de nov. de 2020.
- LONG, Mansheng; HE, Dongjian. Hydraulic erosion simulation using finite volume method on graphics processing unit. In: INTERNATIONAL CONFERENCE ON INFORMATION ENGINEERING AND COMPUTER SCIENCE, 2009, [Wuhan]. **Proceedings...** Piscataway: IEEE, 2009. p. 1-4.
- LONGAY, Steven *et al.* Treesketch: interactive procedural modeling of trees on a tablet. In: SYMPOSIUM ON SKETCH-BASED INTERFACES AND MODELING, 5., 2012, Annecy. **Proceedings...** Goslar: Eurographics Association, 2012. p. 107-120.
- MANDELBROT, Benoit B.; VAN NESS, John W. Fractional Brownian motions, fractional noises and applications. **SIAM review**, Philadelphia. v. 10, n. 4, p. 422-437, out. 1968.
- MEI, Xing; DECAUDIN, Philippe; HU, Bao-Gang. Fast hydraulic erosion simulation and visualization on GPU. In: PACIFIC CONFERENCE ON COMPUTER GRAPHICS AND APPLICATIONS, 15., 2007, Maui. **Proceedings...** Piscataway: IEEE, 2007. p. 47-56.
- MICROSOFT. **Graphics Pipeline**. [S.l.], 2018a. Disponível em: <<https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline>>. Acesso em: 1 de jun. de 2020.
- MICROSOFT. **Compute Shader Overview**. [S.l.], 2018b. Disponível em: <<https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader>>. Acesso em: 3 de nov. de 2020.
- MÜLLER, Matthias; CHARYPAR, David; GROSS, Markus. Particle-based fluid simulation for interactive applications. In: ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON COMPUTER ANIMATION, 3., 2003, San Diego. **Proceedings...** Goslar: Eurographics Association, 2003. p. 154-159.
- MUSGRAVE, F. Kenton; KOLB, Craig E.; MACE, Robert S. The synthesis and rendering of eroded fractal terrains. **ACM Siggraph Computer Graphics**, [S.l.], v. 23, n. 3, p. 41-50, jul. 1989.
- OLSEN, Jacob. **Realtime Procedural Terrain Generation**: Realtime Synthesis of Eroded Fractal Terrain for Use in Computer Games, 2004. Odense: University of Southern Denmark. 20 p.
- PARK, In K. *et al.* Design and Performance Evaluation of Image Processing Algorithms on GPUs. **IEEE Transactions on Parallel and Distributed Systems**, Piscataway, v. 22, n. 1, p. 91-104, jan. 2011.
- PEYTAVIE, Adrien *et al.* Arches: a framework for modeling complex terrains. **Eurographics**, Blackwell, v. 28, n. 2, p. 457-467, abr. 2009.
- PLIENINGER, Tobias *et al.* Exploring ecosystem-change and society through a landscape lens: recent progress in European landscape research. **Ecology and Society**, [S.l.], v. 20, n. 2, art. 5, jun. 2015.
- REED, Nhatan. **Quick And Easy GPU Random Numbers In D3D11**. [S.l.], 2013. Disponível em: <<http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/>>. Acesso em: 29 de out. de 2020.
- RODEN, Timothy; PARBERRY, Ian. Clouds and stars: efficient real-time procedural sky rendering using 3d hardware. In: ACM SIGCHI INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTER ENTERTAINMENT TECHNOLOGY, 5., 2005, Valencia. **Proceedings...** New York: Association for Computing Machinery, 2005. p. 434-437.
- SAHA, Diptarup *et al.* Implementation of Image Enhancement Algorithms and Recursive Ray Tracing using CUDA. **Procedia Computer Science**, Netherlands, v. 79, p. 516-524, 2016.
- VALDETARO, Alexandre *et al.* Understanding Shader Model 5.0 with DirectX 11. In: SIMPÓSIO BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL, 9., 2010, Florianópolis. **Anais...** Florianópolis: [s.n.], 2010. p. 1-18.
- UNITY. **Compute shaders**. [S.l.], 2020. Disponível em: <<https://docs.unity3d.com/Manual/class-ComputeShader.html>>. Acesso em: 3 de nov. de 2020.

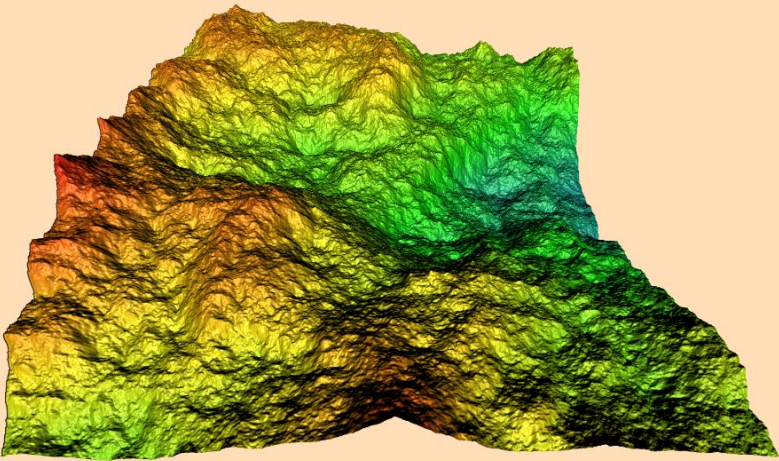
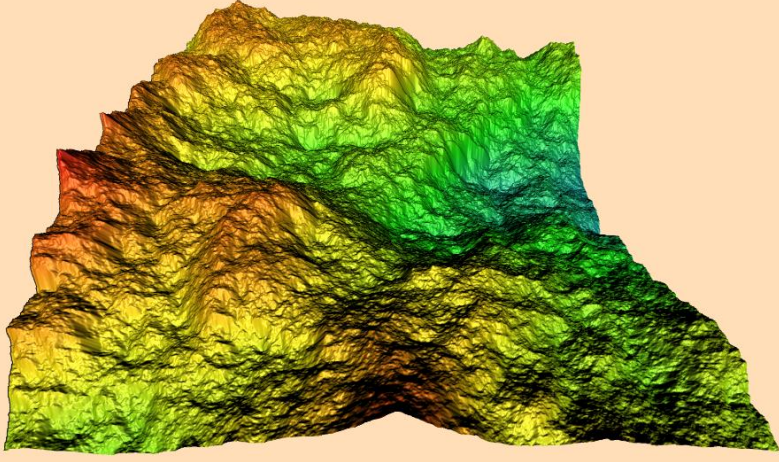
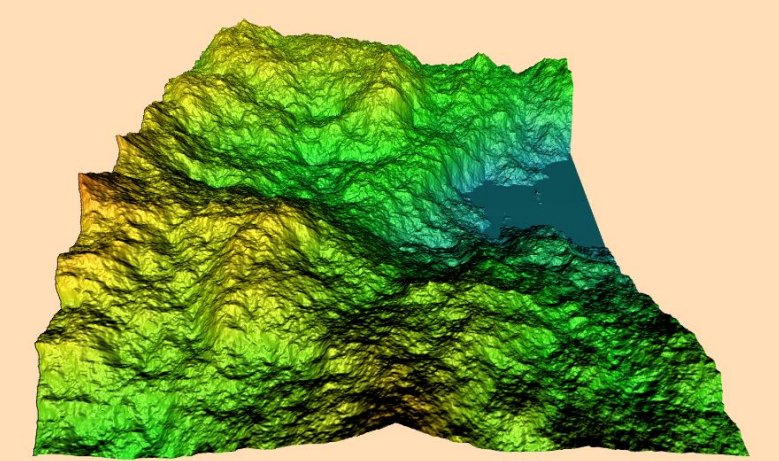
APÊNDICE A – TERRENOS GERADOS PELA FERRAMENTA

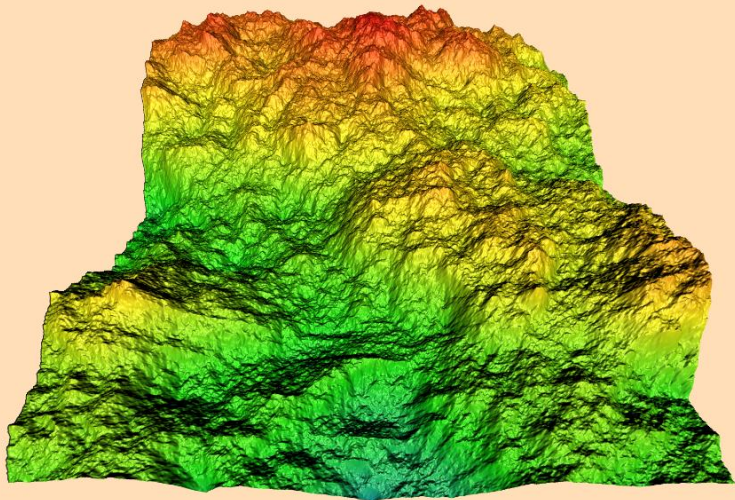
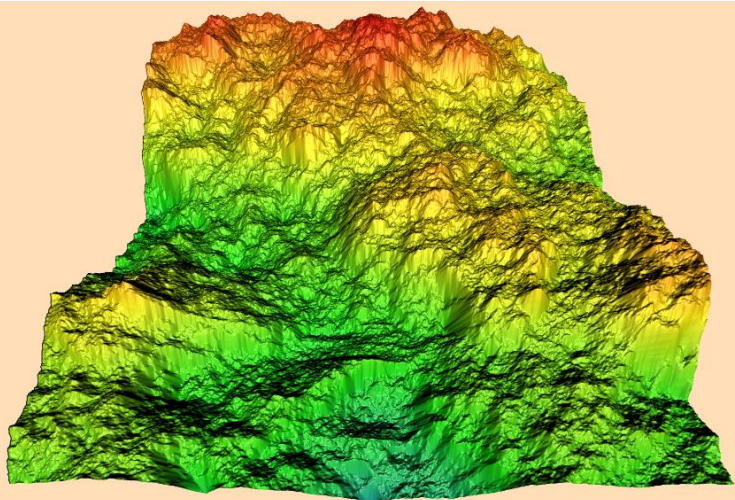
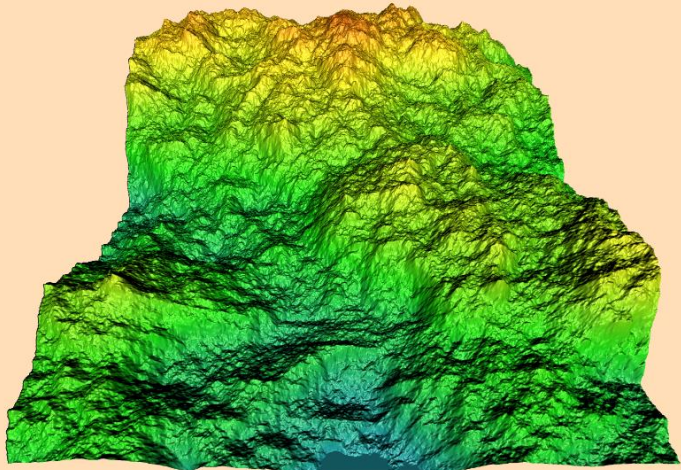
Este apêndice apresenta alguns dos terrenos gerados pela ferramenta. O Quadro 12 apresenta 5 terrenos gerados, sendo que para cada um é exibido a semente utilizada na geração, o terreno base gerado pelo *diamond-square* e as versões do mesmo terreno submetido ao erosão térmica e erosão hidráulica. Assim é possível observar as diferenças causadas em diferentes tipos de terrenos por cada um dos algoritmos.

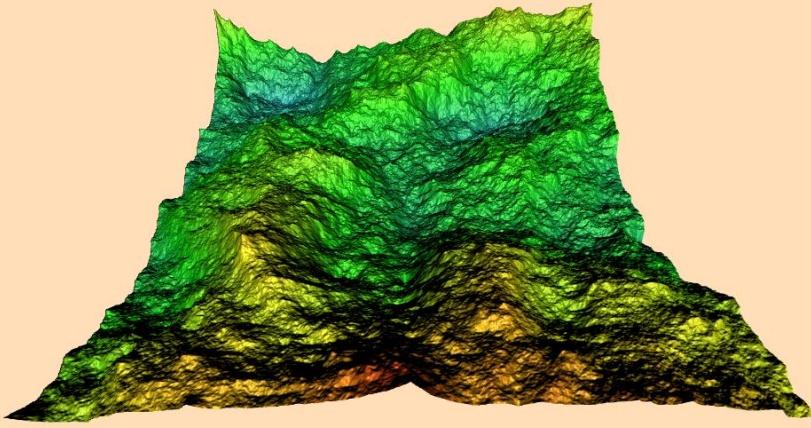
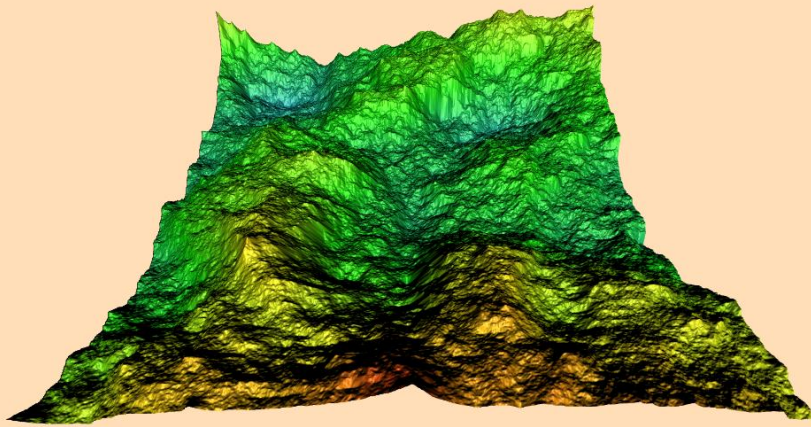
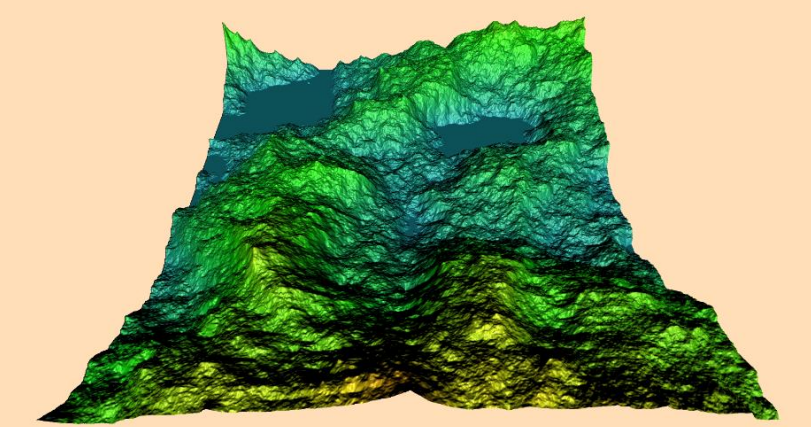
Quadro 12 - Terrenos gerados pela ferramenta através do *diamond-square*, erosão térmica e erosão hidráulica

Sementes	Terrenos gerados
1428799138	 <p><i>Diamond-square</i></p>
	 <p>Erosão térmica</p>
	 <p>Erosão hidráulica</p>

	 <p><i>Diamond-square</i></p>
705741660	 <p>Erosão térmica</p>
	 <p>Erosão hidráulica</p>

	 <p><i>Diamond-square</i></p>
928586617	 <p>Erosão térmica</p>
	 <p>Erosão hidráulica</p>

877219381	 <p><i>Diamond-square</i></p>
	 <p>Erosão térmica</p>
	 <p>Erosão hidráulica</p>

	 <p><i>Diamond-square</i></p>
2100398819	 <p>Erosão térmica</p>
	 <p>Erosão hidráulica</p>

Fonte: elaborado pelo autor.

APÊNDICE B – DADOS DE NATURALIDADE DOS TERRENOS COLETADOS PARA ANÁLISE

Este apêndice apresenta todos os dados de naturalidade coletados, a partir dos quais extraiu-se as médias utilizadas nas discussões do capítulo 4. O Quadro 13 apresenta os parâmetros utilizados por cada um dos algoritmos para a realização dos testes, seguido do Quadro 14 que apresenta a relação de grupos de *threads* e *threads* por grupo utilizada para cada dimensão dos terrenos testados.

Quadro 13 - Parâmetros utilizados nos testes dos algoritmos de erosão térmica e hidráulica

Algoritmos	Parâmetros
Erosão térmica	strength = 0,5; talus = 6; iterations = 100
Erosão hidráulica	rain = 0,08; solubility = 0,02; evaporation = 0,5; iterations = 100

Fonte: elaborado pelo autor.

Destaca-se que o fato dos terrenos com dimensão de 257x257 apresentado no Quadro 14 possuírem 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*.

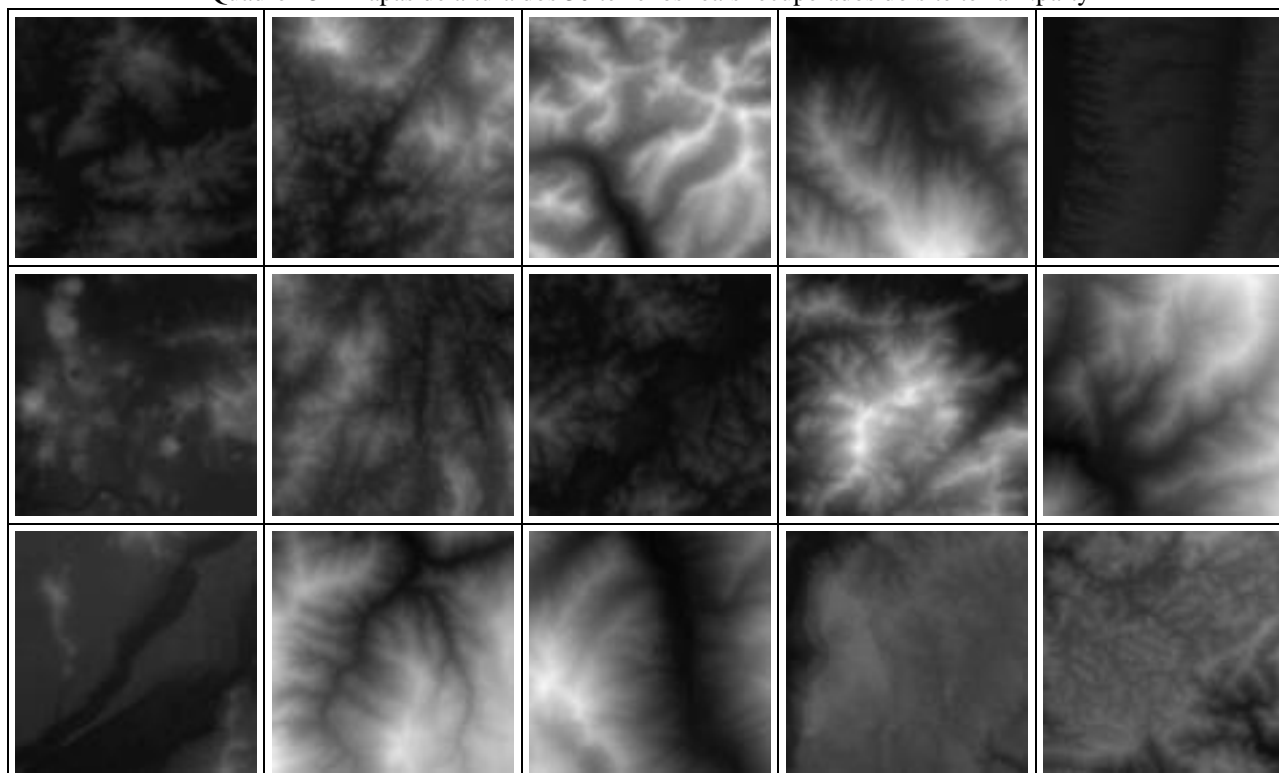
Quadro 14 - Relação entre *threads* e grupos de *threads* utilizada para cada dimensão de terreno

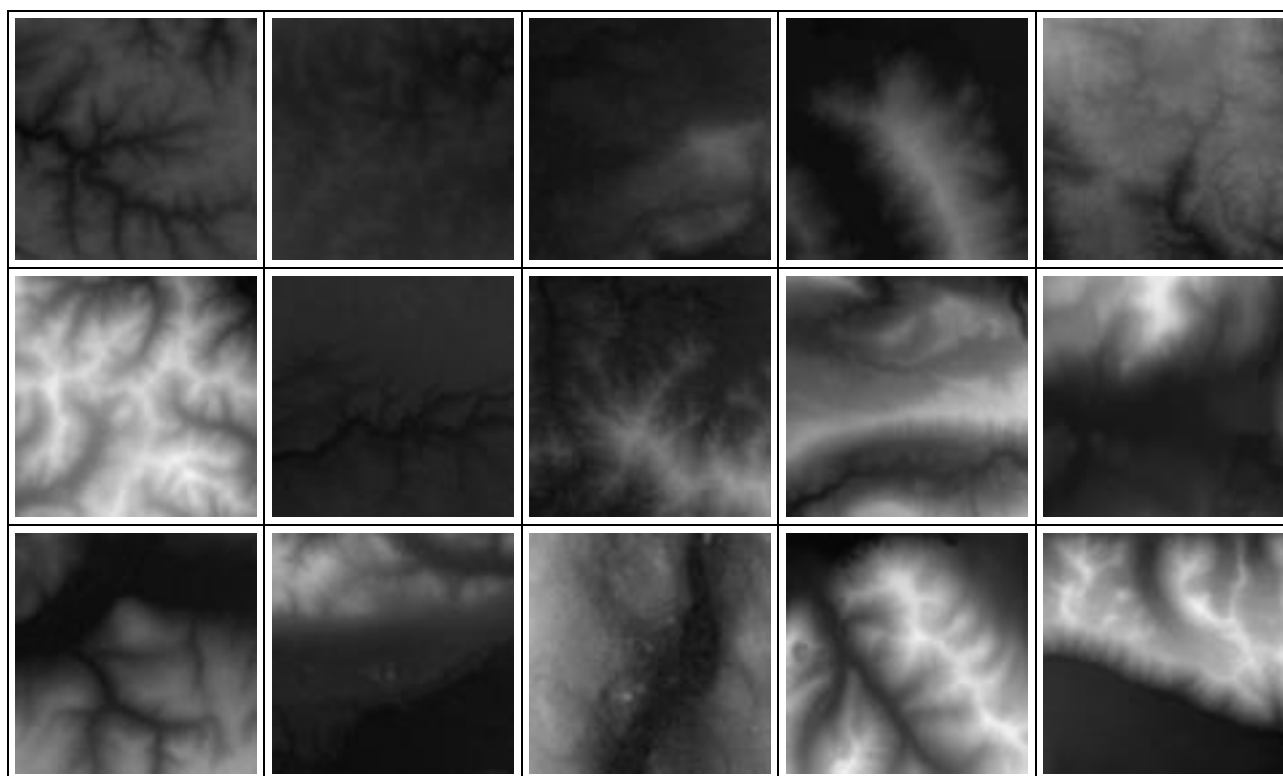
Dimensões	Grupos de <i>threads</i> em <i>x</i> e <i>y</i>	<i>Threads</i> por grupo em <i>x</i> e <i>y</i>	Total de <i>threads</i>
65 x 65	5 x 5	13 x 13	4.225
129 x 129	43 x 43	3 x 3	16.641
257 x 257	257 x 257	1 x 1	66.049
513 x 513	19 x 19	27 x 27	263.169
1.025 x 1.025	41 x 41	25 x 25	1.050.625

Fonte: elaborado pelo autor.

O Quadro 15 apresenta os mapas de altura dos 30 terrenos reais recuperados a partir do site terrain.party e analisados pela ferramenta para realizar as comparações apresentadas no capítulo 5. Em seguida a Tabela 5 apresenta os resultados da aplicação da lei de Benford e *erosion score* nestes terrenos.

Quadro 15 - Mapas de altura dos 30 terrenos reais recuperados do site terrain.party





Fonte: elaborado pelo autor.

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.

Com a análise dos terrenos reais concluída, restam os resultados da aplicação da lei de Benford e *erosion score* nos terrenos gerados pela ferramenta, sendo neste caso 50 terrenos gerados pelos algoritmos desenvolvidos. Os resultados do *diamond-square* são apresentados na Tabela 6, seguido dos de erosão térmica na Tabela 7, erosão hidráulica na Tabela 8 e, por fim, do *diamond-square* com filtro de melhores resultados na Tabela 9.

Tabela 6 – Resultados da geração de terrenos com *diamond-square*

Número	Semente terreno	Distribuição da lei de Benford									Erosion score
		1	2	3	4	5	6	7	8	9	
1	1428799138	3	5	12	31	20	13	11	5	0	0,434
2	2070800132	3	12	17	22	20	12	9	4	1	0,435
3	464531400	5	7	9	13	16	18	18	12	2	0,421
4	705741660	3	9	14	20	23	16	10	4	1	0,430
5	303002533	9	13	15	14	15	13	12	7	2	0,424
6	438907051	4	9	14	16	18	19	13	5	2	0,439
7	2003470750	6	19	20	18	10	12	8	5	2	0,440
8	949691303	5	12	20	17	15	11	9	8	3	0,424
9	928586617	6	6	8	15	19	19	21	5	1	0,436
10	877219381	1	3	13	17	16	23	13	11	3	0,429
11	1237917249	0	1	4	12	20	25	26	11	1	0,438
12	1060139585	4	7	12	12	23	24	11	6	1	0,437
13	1720505882	18	10	13	11	9	10	15	9	5	0,450
14	1130495001	6	12	17	15	11	12	14	10	3	0,427
15	76715554	1	2	4	11	19	16	26	16	5	0,429
16	870751325	4	11	27	30	13	6	6	2	1	0,425
17	273649511	17	15	21	16	16	9	3	2	1	0,432
18	2130790706	5	14	12	13	25	11	10	7	3	0,432
19	1464243004	5	7	13	18	14	15	18	9	1	0,425
20	1061503877	1	4	9	12	32	24	13	5	0	0,420
21	546368075	8	13	11	12	14	15	17	7	3	0,440
22	263116556	19	10	11	11	15	11	9	9	5	0,449
23	1281945677	11	18	19	17	12	10	5	5	3	0,422
24	1417850195	1	4	14	19	26	20	7	7	2	0,429
25	1127507743	8	15	23	29	14	5	3	2	1	0,422
26	1727180827	7	10	12	23	23	9	8	5	3	0,444
27	1017682338	5	6	13	21	20	15	8	7	5	0,427
28	1071620844	1	2	10	9	13	18	27	17	3	0,429
29	1739078074	4	10	12	18	22	20	11	2	1	0,423
30	947613573	5	12	18	16	10	11	14	12	2	0,437
31	1203005699	8	26	17	10	13	14	7	4	1	0,434
32	1451306892	4	7	9	17	27	23	10	2	1	0,426
33	2100398819	15	26	20	13	11	8	5	1	1	0,438
34	575927986	11	11	9	10	16	20	14	7	2	0,436
35	864153932	5	6	9	16	20	24	13	6	1	0,439
36	2025978504	4	11	13	17	18	15	11	9	2	0,428
37	1607277231	2	7	12	20	25	15	11	7	1	0,421
38	1588743815	4	7	15	20	24	19	10	1	0	0,428
39	1792600592	5	6	16	24	24	17	5	2	1	0,432
40	1307895453	6	12	15	16	16	16	10	7	2	0,421
41	1675684254	5	6	16	18	17	20	14	3	1	0,435
42	1811588772	4	8	11	9	13	22	21	9	3	0,428
43	941184293	7	14	14	15	17	13	9	8	3	0,427
44	1833602986	7	22	27	21	8	8	5	1	1	0,420
45	1505738938	5	13	14	18	14	16	14	4	2	0,434
46	803331382	3	11	20	30	15	11	5	4	1	0,428
47	1477879545	2	9	23	25	12	9	8	8	4	0,432
48	72728207	6	11	17	23	18	12	8	4	1	0,431
49	672401291	10	22	24	10	9	10	10	3	2	0,435
50	1363366364	3	11	16	13	19	20	14	3	1	0,427
Média		5,8	10,5	14,7	17,1	17,2	15,1	11,6	6,2	1,9	0,431

Fonte: elaborado pelo autor.

Tabela 7 - Resultados da aplicação da erosão térmica sob terreno gerado pelo *diamond-square*

Número	Semente terreno	Distribuição da lei de Benford									Erosion score
		1	2	3	4	5	6	7	8	9	
1	1428799138	3	5	13	30	20	13	11	5	0	0,420
2	2070800132	3	11	17	22	21	12	9	4	1	0,380
3	464531400	5	7	9	13	16	18	18	12	2	0,369
4	705741660	3	9	14	21	22	16	10	4	1	0,387
5	303002533	9	13	15	14	15	13	12	7	2	0,354
6	438907051	4	9	14	16	18	19	13	5	2	0,395
7	2003470750	6	18	21	18	10	12	8	5	2	0,428
8	949691303	5	12	20	17	15	11	9	8	3	0,383
9	928586617	6	6	8	16	18	19	21	5	1	0,421
10	877219381	1	3	13	17	16	24	13	10	3	0,404
11	1237917249	0	1	4	12	20	25	26	11	1	0,422
12	1060139585	4	7	12	12	23	24	11	6	1	0,428
13	1720505882	18	11	13	11	9	11	13	9	5	0,432
14	1130495001	8	12	17	15	10	11	14	10	3	0,410
15	76715554	0	2	4	11	19	16	26	16	6	0,406
16	870751325	3	11	28	30	13	6	6	2	1	0,408
17	273649511	16	15	20	17	16	10	3	2	1	0,418
18	2130790706	5	14	12	14	25	12	9	7	2	0,405
19	1464243004	5	7	13	18	14	15	18	9	1	0,361
20	1061503877	1	4	9	12	31	25	13	5	0	0,397
21	546368075	8	13	11	12	14	15	16	8	3	0,426
22	263116556	19	10	11	11	15	11	9	9	5	0,435
23	1281945677	11	18	19	17	12	10	5	5	3	0,363
24	1417850195	1	4	15	19	26	19	7	7	2	0,398
25	1127507743	8	15	23	28	15	5	3	2	1	0,407
26	1727180827	7	10	12	23	23	9	8	5	3	0,440
27	1017682338	5	6	13	21	20	15	8	7	5	0,409
28	1071620844	1	2	10	9	13	18	27	17	3	0,421
29	1739078074	4	10	12	18	22	20	11	2	1	0,320
30	947613573	6	11	18	16	10	11	14	12	2	0,425
31	1203005699	8	26	17	9	13	15	7	4	1	0,422
32	1451306892	4	7	9	18	27	23	9	2	1	0,377
33	2100398819	15	26	20	13	11	8	5	1	1	0,424
34	575927986	10	12	9	10	16	20	14	7	2	0,418
35	864153932	5	6	9	16	20	25	12	6	1	0,407
36	2025978504	3	11	13	17	18	15	12	9	2	0,415
37	1607277231	3	7	12	20	24	15	11	7	1	0,356
38	1588743815	4	7	15	20	24	19	10	1	0	0,408
39	1792600592	4	6	16	25	24	17	5	2	1	0,407
40	1307895453	7	11	14	17	16	16	10	7	2	0,371
41	1675684254	5	6	16	18	17	20	14	3	1	0,427
42	1811588772	4	7	11	9	12	21	23	10	3	0,386
43	941184293	7	14	14	15	18	13	9	8	2	0,406
44	1833602986	7	22	27	21	8	9	5	1	1	0,387
45	1505738938	4	13	15	18	14	16	14	4	2	0,386
46	803331382	3	10	20	31	14	11	6	4	1	0,392
47	1477879545	3	9	23	25	12	9	8	8	3	0,418
48	72728207	6	12	16	23	18	12	8	4	1	0,415
49	672401291	10	22	24	10	9	10	10	3	2	0,426
50	1363366364	3	11	16	13	19	20	14	3	1	0,416
Média		5,8	10,4	14,7	17,2	17,1	15,2	11,5	6,2	1,9	0,403

Fonte: elaborado pelo autor.

Tabela 8 - Resultados da aplicação da erosão hidráulica sob terreno gerado pelo *diamond-square*

Número	Semente terreno	Distribuição da lei de Benford									Erosion score
		1	2	3	4	5	6	7	8	9	
1	1428799138	10	24	27	15	13	7	2	1	1	0,465
2	2070800132	19	18	24	15	12	6	4	1	1	0,456
3	464531400	12	12	14	17	20	17	6	1	1	0,464
4	705741660	14	18	25	19	12	8	2	1	1	0,471
5	303002533	23	16	16	15	13	10	5	1	1	0,507
6	438907051	17	14	19	18	18	7	5	1	1	0,490
7	2003470750	20	17	16	9	12	6	6	2	2	0,475
8	949691303	21	19	16	15	10	8	8	2	1	0,470
9	928586617	13	10	20	19	22	12	2	1	1	0,503
10	877219381	8	17	16	24	17	11	6	1	0	0,435
11	1237917249	2	9	17	22	28	17	5	0	0	0,437
12	1060139585	13	12	17	27	19	7	3	1	1	0,475
13	1720505882	30	14	10	9	16	10	8	2	1	0,662
14	1130495001	21	19	12	11	13	15	6	2	1	0,498
15	76715554	4	6	19	13	25	20	12	1	0	0,431
16	870751325	25	30	23	8	6	4	2	1	1	0,459
17	273649511	29	23	18	15	7	3	2	1	2	0,534
18	2130790706	18	12	22	20	12	8	5	2	1	0,455
19	1464243004	14	17	15	16	18	15	3	1	1	0,474
20	1061503877	7	11	21	32	18	9	2	0	0	0,427
21	546368075	20	13	14	15	16	13	6	2	1	0,524
22	263116556	29	13	15	14	10	9	7	2	1	0,688
23	1281945677	30	21	14	12	7	6	6	2	2	0,545
24	1417850195	10	18	22	26	11	7	5	1	0	0,431
25	1127507743	24	28	24	8	5	4	3	2	2	0,479
26	1727180827	15	17	29	14	8	8	6	2	1	0,516
27	1017682338	14	18	23	15	13	7	7	2	1	0,489
28	1071620844	5	12	9	17	24	24	9	0	0	0,433
29	1739078074	13	15	21	24	18	5	2	1	1	0,449
30	947613573	17	20	13	11	13	17	6	2	1	0,470
31	1203005699	32	11	12	18	10	7	4	3	3	0,503
32	1451306892	12	13	26	27	14	5	1	1	1	0,469
33	2100398819	33	19	13	13	8	5	3	3	3	0,545
34	575927986	18	11	13	20	20	11	5	1	1	0,526
35	864153932	11	13	19	23	21	8	3	1	1	0,489
36	2025978504	18	15	17	19	13	12	4	1	1	0,454
37	1607277231	12	17	23	20	13	9	4	1	1	0,448
38	1588743815	15	18	23	21	16	4	1	1	1	0,480
39	1792600592	15	23	24	23	9	3	1	1	1	0,497
40	1307895453	16	17	17	17	13	11	5	2	1	0,456
41	1675684254	15	19	17	19	19	7	2	1	1	0,487
42	1811588772	14	10	11	16	27	13	7	1	1	0,467
43	941184293	23	15	17	18	9	10	4	2	2	0,510
44	1833602986	38	24	13	8	8	3	2	2	2	0,516
45	1505738938	19	17	17	13	19	8	4	2	1	0,505
46	803331382	16	31	22	13	9	4	3	1	1	0,449
47	1477879545	19	28	17	10	9	8	7	1	1	0,450
48	72728207	20	19	22	15	10	8	3	1	2	0,487
49	672401291	38	16	10	10	12	5	4	3	2	0,568
50	1363366364	18	14	15	21	20	8	2	1	1	0,439
Média		17,98	16,86	17,98	16,78	14,3	8,98	4,4	1,38	1,12	0,487

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
49	555618055	29	24	15	9	10	5	5	2	1	0,512
50	860038552	30	31	22	11	2	1	1	1	1	0,478
Média		30,5	26,1	16,6	8,8	6,3	4,8	3,8	2,9	1,9	0,484

Fonte: elaborado pelo autor.