PROJETO TCC - BCC	ANO/SEMESTRE:	2020/1

GERAÇÃO PROCEDURAL DE TERRENOS VIRTUAIS COM APARÊNCIA NATURAL UTILIZANDO GPU

Alex Seródio Gonçalves

Prof. Dalton Solano dos Reis – Orientador

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. Nesses casos, simulações computacionais possibilitam a criação e validação de modelos que representam estes cenários, permitindo assim a execução de experimentos, análise dos resultados e validação de teorias em um ambiente controlado (HEERMANN, 1990, p. 8). Estas simulações podem ser utilizadas para analisar os processos naturais e sociais causadores de mudanças na paisagem, sendo necessário, porém, a utilização de um cenário adequado 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 na formação de terrenos reais, como processos de erosão que modificam o relevo do terreno (OLSEN, 2004). No caso dos modelos físicos os algoritmos exigem um poder de processamento considerável para atingir resultados realistas, tornando em muitos casos sua execução em tempo real impraticável (MEI; DECAUDIN; HU, 2007, p. 47).

Com os constantes avanços no desenvolvimento de *hardware*, a utilização da GPU (*Graphic Processing Unit*) para resolução de problemas computacionalmente complexos vem se tornando cada vez mais viável. Esta diferença fica evidente principalmente conforme o tamanho do dado a ser processado aumenta (HANGÜN; EYECIOĞLU, 2017, p. 34). Este poder de paralelismo pode ser utilizado para realizar o processamento dos algoritmos de erosão de forma eficiente e ainda preservar o nível de realismo necessário para simulações (MEI; DECAUDIN; HU, 2007, p. 48).

1 da Graphic Processing Unit (GPU) para

O algoritmo de erosão hidráulica consiste em simular o processo de chuvas em terrenos, que faz com que a água corrente resultante da chuva dissolva materiais de certas partes do terreno, que são carregados pela correnteza e depositados em outro local (OLSEN, 2004, p. 8). Já a erosão térmica simula o processo de materiais se desprenderem de pontos elevados do terreno e deslizarem até pontos mais baixos (OLSEN, 2004, p. 5). Estes algoritmos foram utilizados ao longo dos anos por trabalhos como Musgrave, Kolb e Mace (1989), Olsen (2004), Mei, Decaudin e Hu (2007), Long e He (2009), Jákó e Tóth (2011) e Diegoli Neto (2017). Sendo que cada um apresenta sua própria implementação dos algoritmos, tendo como objetivo a geração de terrenos naturais ou a simulação dos processos em questão.

4.3 GPU

Com a crescente necessidade de processamento de grandes quantidades de dados, surge a demanda por plataformas capazes de processar massas de dados de forma eficiente. Uma forma de atingir isto é através do paralelismo. Isto fica evidente em áreas como Processamento de Imagem e Visão Computacional, que tendem a aplicar a mesma operação repetidamente em cada *pixel* de uma imagem. Este tipo de operação, por mais que realizada frequentemente de forma sequencial, é possível de ser paralelizada (PARK et al., 2011, p. 1).

Recentemente as GPUs tem se mostrado ferramentas adequadas para este tipo de processamento em paralelo (SAHA et al., 2016, p. 516). Isso se deve principalmente ao fato de sua arquitetura ser desenvolvida pensando exclusivamente em processamento paralelo, diferente das CPUs (*Central Process Unit*) que foram criadas para processamento serial e adaptadas para paralelo (HANGÜN; EYECIOĞLU, 2017, p. 34). A Figura 5 apresentada por Saha et al. (2016) ilustra a diferença arquitetural entre CPU e GPU, onde é possível observar que a GPU possui vários blocos de processamento compostos por ALUs (*Arithmetic Logic Unit*), o que permite o processamento de grandes quantidades de dados simultaneamente.

1 das Central Process Unit (CPU) que

2 por Arithmetic Logic Unit (ALU), o que

Figura 5 - Diagrama de blocos ilustrando a diferença arquitetural entre CPU e GPU

Fonte: Saha et al. (2016, p. 517).

Diferentes áreas da computação podem se beneficiar desta arquitetura para otimizar o processamento de seus algoritmos. O ganho de performance 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 a fim de comparar as diferenças de performance entre os dois, observando melhorias de até 120% na performance de alguns dos algoritmos em GPU. Che et al. (2007) propõe o mesmo estilo de análise, porém com algoritmos de propósitos gerais como simulação térmica e *K-Means* (utilizado em mineração de dados). Em todos os casos observa-se um ganho considerável de performance nas versões em GPU. Esta diferença de performance fica mais perceptível conforme o tamanho do dado a ser processado aumenta (HANGÜN; EYECIOĞLU, 2017, p. 39-40).

Também se destacam os casos em que a versão em GPU do algoritmo não apresenta ganhos consideráveis se comparada com a versão em CPU. Isso ocorre principalmente 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 constantes acessos à memória e o processo não foi corretamente configurado para isso, fazendo mau uso da memória compartilhada da GPU (PARK et al., 2011, p. 3).

4.4 DIRECTX E HLSL

Park et al. (2011, p. 2) destaca a existência de estratégias diferentes de programação em GPU dependendo da plataforma adotada. Isto significa que um algoritmo implementado em uma plataforma não é facilmente adaptável para outra, o que torna a escolha de uma plataforma algo importante para o desenvolvimento. Exemplos de plataformas seriam: CUDA (*Compute Unified Device Architecture*) da Nvidia, DirectX da Microsoft e Metal da Apple. Por razões de maior compatibilidade com o motor gráfico Unity, este trabalho será focado no uso do DirectX 11 com a linguagem HLSL (*High Level Shading Language*).

1 Compute Unified Device Architecture (CUDA)

Como o próprio nome sugere, HLSL é uma linguagem para criação de *shaders*. Um *shader* pode ser definido como um *script* responsável por comunicar à GPU quais cálculos realizar para produzir uma transformação ou efeito desejado (VALDETARO et al., 2010, p. 1). Algo importante para se ter em mente é a existência de diferentes tipos de *shaders*, que são responsáveis por diferentes etapas da *pipeline* gráfica. A Figura 6 apresenta um diagrama com todas as oito etapas existentes na *pipeline* gráfica do DirectX 11, sendo que das oito, cinco são compostas por *shaders* programáveis através da linguagem HLSL (MICROSOFT, 2018).

Por se tratar de um conteúdo extenso, este trabalho não entrará em detalhes sobre cada etapa da *pipeline*, porém este conhecimento se faz necessário para melhor uso da *pipeline*.