TERRAIN GENERATION

Mestrado Integrado em Engenharia Informática

Unidade Curricular de Visualização e Iluminação I

André Germano A71150 | Leandro Salgado A70949 | Luís Costa A73434 | Sofia Carvalho A76658

Índice

1 Introdução

Terrain Generation consiste na geração automática de terreno com base num mapa de alturas (uma textura de ruído, por exemplo) ou recorrendo a funções que produzam esse ruído em *runtime*. Esta técnica é usada em vários jogos que tentem disponibilizar ao jogador um mundo virtual de grandes dimensões que possa ser explorado.

O objetivo deste trabalho é a implementação de um programa que demonstre o conceito de *Terrain Generation*, a análise das decisões importantes de implementação e do correspondente desempenho do sistema criado.

A geração de terreno implementada pelo grupo é completamente automática, pelo que é considerada *procedural generation*. Esta implementação, ao contrário de outras que se baseiam em mapas de alturas, não requer qualquer *input* para funcionar.

Na secção 2 deste relatório são identificados os pontos principais da implementação do programa. Na secção 3, é feita uma análise do desempenho do programa com alterações entre execuções de forma a conseguir comparar os resultados diferentes. Na secção 4 iremos apontar quais as principais limitações do trabalho, bem como os pontos que o grupo considera que foram corretamente implementados. Finalmente, na secção 5, é feita uma conclusão sobre o trabalho.

2 Implementação

A demonstração do tema de *Terrain Generation* proposto é feita segundo uma técnica baseada em ruído, em que uma função calcula, para cada ponto de uma grelha, qual a altura que cada um dos vértices deve ter.

A função de ruído usada implementa um algoritmo criado por *Ken Perlin* designado de *Simplex Noise*, uma melhoria do seu algoritmo também por si criado em 1983 designado de *Perlin Noise*.

O grupo optou por utilizar o algoritmo de *Simplex Noise* nesta implementação, uma vez que é superior em termos de eficiência. Há, inclusive, casos em que o resultado produzido é melhor do que o produzido por *Perlin Noise*.

O algoritmo de *Simplex Noise* usado nesta implementação não é da autoria do grupo, tendo sido retirado de um repositório na *Internet*.

Com exceção da implementação do algoritmo de *noise*, todo o código foi feito pelo grupo. Utilizando a Nau3D para suporte, a técnica foi implementada em GLSL, fazendo uso de três *shaders*: *vertex shader*, *geometry shader*, e *fragment shader*.

Nas subsecções seguintes, é analisado todo o código de cada um dos *shaders* desenvolvidos, explicando as diferentes decisões tomadas. Na última subsecção são desenvolvidas algumas considerações acerca de vários aspetos relacionados com a implementação.

2.1 Vertex Shader

O *vertex shader* trata do cálculo da altura de cada vértice, fazendo uso de uma função de ruído *simplex 2D*[1], e devolve para o *pipeline* a posição de cada vértice da grelha com a sua devida altura. O cálculo dessa altura, para cada vértice, é dado pela função *turbulence*, que recebe como parâmetros a posição de um vértice (coordenadas *x* e *z*) e um valor que define quantos *octaves* serão usados para o cálculo do valor de *noise* em cada ponto. Esse cálculo não é feito recorrendo simplesmente recorrendo à função de *noise*, sendo antes uma combinação de vários valores retornados por essa função, para valores de frequência diferentes. Cada *octave* representa um cálculo de *noise* com uma certa frequência, e o valor final de *noise* é calculado através da soma de todos os valores de *noise* calculados em cada *octave*.

A frequência usada na função de *Simplex Noise (sperlin)* determina, na sua essência, o a variação dos valores que a função retorna. Dito de outra forma, uma baixa frequência resultaria num terreno com variações mais suaves, e uma alta frequência em variações muito mais repentinas. Somando os valores de *noise* retornados pela função *sperlin* calculados com frequências diferentes, o resultado final inclui vários tipos de variações, permitindo, no contexto de *terrain generation*, criar vários fenómenos geológicos de dimensões diferentes.

É no *vertex shader* que se calculam as cores que o terreno vai exibir. Devido à grande variedade de alturas que o terreno gerado exibe, faz sentido fazer a distinção entre zonas de baixa altitude, que são consideradas como zonas com água, zonas de elevada altitude, associadas aos picos das montanhas, que contêm neve, e zonas de altitude média, que são consideradas como terreno normal. As zonas de altitude média não são diferenciadas, portanto têm todas a mesma cor.

A função que determina a cor de cada vértice designa-se de *color*, e recebe como argumentos a altura do vértice em questão, calculada pela função *turbulence*, e a altura a partir da qual aparece neve.

O código da função *color* pode ser visto na Fig. 1.

```
vec4 color(float height, float snow){
    bool r = height < 0.0;
    int nr = int(!r);
    int s = int(height > snow);
    return vec4(s, max(nr, s), max(int(r), s), height * nr);
}
```

Figura 1: Código da função *color*.

O grupo teve em consideração os conteúdos lecionados nas aulas a respeito do uso de condições *if* e *else* não serem tão eficientes como na programação em CPU, pelo que houve um especial cuidado em evitar esse tipo de práticas. Essa estratégia pode ser executada porque as cores usadas pelo grupo são muito simples, pelo que é possível utilizar os valores numéricos das comparações efetuadas na função para codificarem cores *RGB*. Em aplicações de cores mais complexas poderia ser necessário aplicar expressões condicionais.

[1] *Simplex noise*, de Ken Perlin, modificado para *GLSL*, autoria de Ashima Arts

Como a função de *noise* pode retornar valores negativos, a função de *color* retorna 0 no quarto elemento do vetor de cor retornado, multiplicando o valor da altura recebido como parâmetro por 0 caso a altura seja negativa ou por 1 caso contrário.

Esse valor substitui o valor da posição final do ponto a ser processado, garantindo que a altura mínima do terreno é sempre 0.

O *vertex shader* retorna a posição do vértice depois de calculada a sua altura e a cor determinada para esse vértice.

O grupo optou por implementar uma versão alternativa do *vertex shader* que faz o cálculo das normais para cada vértice por aproximação, utilizando um método em que os vetores necessários para calcular a normal são obtidos por intermédio dos vértices vizinhos do vértice a ser processado. Essa decisão foi tomada porque o aspeto final do terreno fica significativamente melhor, perdendo um pouco na *performance* do programa em relação à versão em que as normais não são calculadas neste *shader*. Se o *vertex shader* calcular as normais, então o programa não precisa de ter *geometry shader*, e este é excluído do *pipeline* gráfico.

2.2 Geometry Shader

Esta subsecção corresponde ao *geometry shader*, pelo que tudo o que está escrito diz respeito à versão em que não foram calculadas as normais no *vertex shader*.

O *geometry shader* é o componente do programa responsável pelo cálculo dos vetores normais dos triângulos cujos vértices são calculados pelo *vertex shader*. Para as calcular, é feito um produto externo entre dois vetores retirados de cada triângulo. Esse cálculo é feito subtraindo a um determinado ponto do triângulo os outros dois pontos, obtendo assim dois vetores paralelos ao triângulo. Fazendo o produto externo desses dois vetores, o resultado é sempre um vetor perpendicular ao triângulo.

A imagem seguinte mostra um excerto de código do cálculo de uma normal de um triângulo.

```
vec3 v1 = vec3(DataIn[2].pos - DataIn[0].pos);
vec3 v2 = vec3(DataIn[1].pos - DataIn[0].pos);
vec3 tnormal = normalize(cross(v1, v2));
```

Figura 2: Cálculo do produto externo dos dois vetores extraídos do triângulo.

De forma a tentar introduzir um pouco de realismo, o grupo optou por alterar o valor da normal se esta apresentar uma inclinação demasiado elevada. Isso leva a que em locais muito inclinados como as encostas de uma montanha tenham locais com rochas em vez do terreno normal. Esse tipo de ocorrência dá-se tanto na encosta a verde (terreno normal) como na encosta com neve. Devido à baixa densidade de triângulos, o efeito criado não é ideal, ficando por vezes triângulos a cinzento no meio de triângulos verdes ou brancos. No entanto, com uma maior densidade, o efeito ganha melhor aspeto.

O cálculo das normais é feito neste *shader* porque é possível aceder aos vértices de cada triângulo para fazer o cálculo da normal diretamente. No caso em que esse cálculo é feito no *vertex shader*, as normais são calculadas por aproximação, assumindo que é conhecida a distância entre os pontos todos da grelha. Para além de serem uma aproximação,

essas normais implicam o cálculo da função de *turbulence* cinco vezes por vértice, em que quatro são para os vértices vizinhos do ponto a ser processado e uma para o próprio vértice.

O cálculo das normais no *geometry shader*, uma vez que estas são calculadas por triângulo, significa que não existe interpolação entre os diferentes pontos de cada triângulo. Portanto, o valor da iluminação será sempre o mesmo em cada triângulo. Visualmente, se houver variações acentuadas de altura, o terreno fica muito facetado, e os triângulos são bastante visíveis.

O cálculo das normais no *vertex shader* calculam uma normal por vértice, pelo que estas serão interpoladas para o triângulo e, consequentemente, o valor da iluminação será diferente para partes distintas do mesmo triângulo. O terreno final, aplicando esta estratégia, apresenta um aspeto muito menos facetado.

2.3 Fragment Shader

De todos os *shaders* usados para o trabalho, o *fragment shader* é o que apresenta menor complexidade, sendo a sua única função a de calcular a cor de cada píxel. Há, no entanto, um aspeto a referir no que diz respeito ao cálculo das intensidades. Quando a cor recebida pelo *shader* é branca (componentes *RGB* são todas 1.0), é usada uma *threshold* de forma a que a intensidade não baixe de um certo valor. O objetivo desta estratégia é fazer com que a neve visualizada não fique demasiado escura, dando assim um aspeto ligeiramente melhor aos cimos das montanhas do terreno gerado.

2.4 Câmera e terreno

Na implementação do *vertex shader*, há duas opções usadas que é relevante salientar.

Em primeiro lugar, às componentes *x* e *z* de cada vértice processado são adicionadas as mesmas componentes da câmara. Este passo é essencial para que a grelha que define o terreno acompanhe o movimento da câmara, criando um efeito de "terreno infinito". São estas as posições que servem de argumento para a função de *noise*.

Em segundo lugar, sempre que a posição da câmara é usada esta primeiro uma discretização, usando a função *floor* do *GLSL*. Esta estratégia foi implementada porque a câmara pode tomar posições com valores decimais. Caso isso aconteça, o terreno ganha um efeito "ondulatório" em que os triângulos mudam continuamente de posição de cada vez que existe movimento por parte da câmara. Fazer o *floor* das posições da câmara garante que os vértices da grelha só mudam de posição quando a câmara avança uma unidade no plano XZ.

A utilização dos *octaves* na função *turbulence,* como já foi referido, permite obter um ruído com melhores características. O número de *octaves* usados influencia imenso o aspeto do terreno até um limite de seis. A partir desse ponto o aspeto varia cada vez menos e o desempenho, no geral, piora.

As imagens seguintes ilustram as diferenças entre gerações com diferentes *octaves* (todas as imagens foram geradas com tamanho e divisões de 1024) e com *geometry shader*.

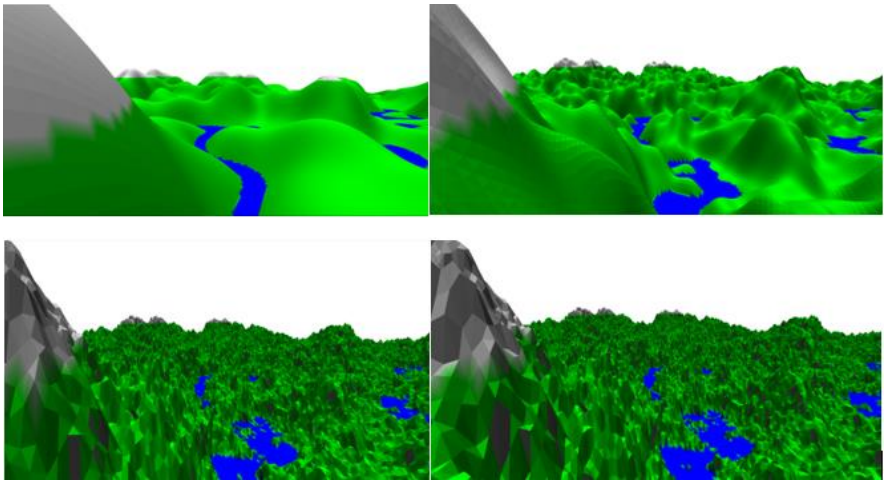

Figura 3: Exemplos do mesmo terreno gerado com, respetivamente, 3, 5, 8 e 16 *octaves.*

Como se pode ver, as diferenças entre as primeiras três imagens, para diferentes quantidades de *octaves* usados, é claramente visível. Já a diferença entre a terceira e quarta imagem é marginal, sendo a quantidade de *octaves* usada na quarta a imagem o dobro da terceira.

3 Análise de Desempenho

De modo a testar a escalabilidade dos *shaders*, foram testadas várias grelhas com diferentes tamanhos e divisões ao longo dos eixos e, com ajuda da *Nau3D,* geraram-se *logs* de cada teste. Os tamanhos e o número de divisões testados foram 512, 1024 e 2048 para ambos, num total de 9 grelhas diferentes testadas. O programa foi testado com uma placa gráfica *AMD Radeon R9 M200X Series* de 2 GB (DDR5) dedicados. Os tempos de execução analisados nesta secção são sempre considerados por *frame*.

Variando apenas o número de divisões em cada eixo, ao duplicar o número de divisões, verifica-se um aumento de aproximadamente 4 vezes no tempo de execução, consistente com o aumento de triângulos resultante da duplicação do número de divisões.

Variando apenas o tamanho da grelha, existem pequenas variações no tempo de execução, na ordem das décimas de milissegundo. No entanto, à medida que o tamanho da grelha se aproxima do número de divisões, verifica-se uma diminuição do tempo de execução. De qualquer das formas, a mudança do tamanho provoca alterações residuais, pelo que não é considerada nesta análise.

Em termos de eficiência da implementação desenvolvida, a função de *noise* é descrita pelo autor como sendo eficiente, o que pode ser comprovado aumentando os *octaves* na turbulência, por exemplo, de 1 para 8, em que um aumento de cálculos de *noise* em 8 vezes resulta num aumento do tempo de execução negligenciável (ao nível dos milissegundos) ou até inexistente. Da mesma forma, um aumento de 8 *octaves* para 16 resulta num pequeno

aumento do tempo de execução, com um mínimo aumento do detalhe. A partir deste ponto, o aumento dos *octaves* tem impacto cada vez maior na *performance* sem qualquer aumento significativo de detalhe.

Em termos de instruções condicionais, as funções contendo *ifs* foram reescritas de forma a atingir o mesmo resultado sem *ifs*, como foi referido na secção anterior. No entanto, não se observaram reduções significativas nos tempos de execução, uma vez que estas não englobam grandes blocos de código.

O tempo de execução foi de, aproximadamente, 3.60 milissegundos para 512 divisões, 14.01 milissegundos para 1024 e 54.95 milissegundos para 2048.

Ao contrário de outras técnicas existentes, a técnica implementada não requer nem permite a utilização de dados previamente gerados como texturas, sendo que os valores de altura são todos calculados em *runtime*.

Pela análise de desempenho, é possível inferir que esta implementação não escala bem para *meshes* muito densas. No entanto, seria possível melhorar o desempenho utilizando *tessellation* para reduzir o detalhe nos limites do *view frustrum* mais afastados da câmara.

Os valores indicados correspondem à implementação feita com as normais calculadas pelo *geometry shader*. Para efeitos de comparação, apresentamos também os mesmos valores para a implementação com o cálculo das normais no *vertex shader*.

Para 512 divisões, o tempo de execução foi aproximadamente de 9.46 milissegundos, para 1024 foi 35.87 e para 2048 foi 143.38. Note-se que neste caso também se verifica um aumento de quatro vezes no tempo de execução para o dobro das dimensões.

O gráfico seguinte mostra os tempos de execução aproximados para 512, 1024 e 2048 divisões.

Figura 4: Gráfico que ilustra os tempos de execução para diferentes números de divisões.

Os valores maiores para cada divisão correspondem aos valores da implementação das normais no *vertex shader*.

A imagem seguinte mostra a diferença entre as duas implementações do cálculo de normais.

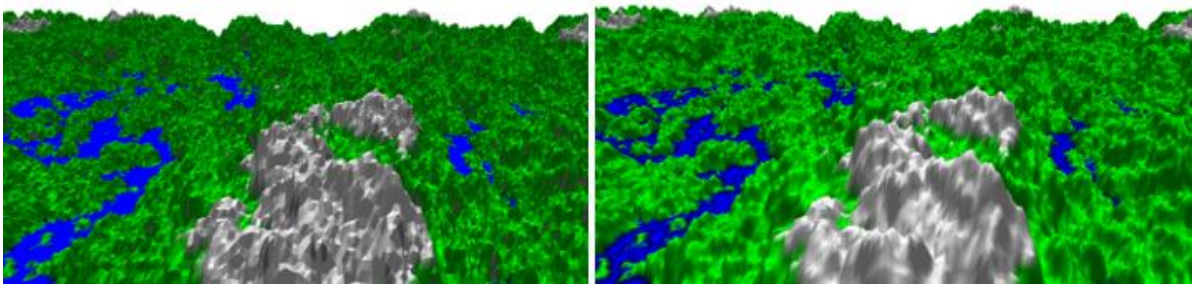

Figura 5: Exemplo de terreno gerado para as duas implementações. À esquerda está a implementação em *geometry shader* e à direita em *vertex shader*.

4 Considerações

Embora a implementação do trabalho não tenha sido único foco do trabalho, é importante salientar que há melhorias que podem certamente ser feitas ao programa que tornem o terreno mais realista e visualmente mais agradável.

O terreno gerado é bastante simples, tendo apenas locais com água, locais com terreno normal e locais com neve. Uma implementação mais completa poderia ter elementos adicionais, como por exemplo vegetação, como árvores e arbustos, diferentes cores para o terreno normal que não está em água nem em neve, ou ainda a aplicação de texturas em vez de cores para as diferentes partes do terreno.

Uma limitação da implementação passa por ser facilmente visível o limite físico do terreno, principalmente quando a câmara assume posições elevadas no terreno. Isso deve-se ao facto de o programa gerar terreno aparentemente infinito, e o modelo da grelha disponibilizado pela Nau3D não ser, obviamente, infinitamente grande. Tudo o que estiver para além dos limites de tamanho da grelha é "espaço vazio".

No caso da implementação em *geometry shader*, o aspeto facetado também é um aspeto negativo, que é parcialmente resolvido com a implementação em *vertex shader*.

Apesar de ter limitações, o programa desenvolvido implementa ao conceito de *Terrain Generation*. Baseando o funcionamento do programa numa função de *noise*, o terreno gerado possui um aspeto bastante heterogéneo, havendo várias formações geológicas que, de acordo com a opinião do grupo, representam fielmente o mundo natural, dentro do possível.

Dentro do possível, o grupo tentou otimizar o programa para que este conseguisse executar com um nível de *frames* por segundo aceitável. Na opinião do grupo, o programa tem bons tempos de execução.

Portanto, e para concluir, embora haja sem dúvida elementos que podem ser adicionados para criar mais realismo no terreno, o objetivo de criar um programa capaz de gerar terreno aparentemente infinito em tempo real foi alcançado.

5 Conclusão

Concluído o trabalho e tendo em conta todo o conhecimento adquirido no decorrer do semestre na unidade curricular de Visão e Iluminação I, ficou claro que, para gerar um terreno simples não é necessária a implementação de um sistema extenso e complexo.

De forma a tentar obter uma solução melhor, o grupo optou por experimentar duas implementações distintas do cálculo de normais. Depois da análise de desempenho efetuada, torna-se claro que a solução de cálculo no *vertex shader*, embora gere um terreno visualmente superior, tem um impacto significativo na *performance* do programa.

Analisando os resultados obtidos tanto a nível visual como de desempenho de GPU, concluímos que o trabalho foi realizado com sucesso e foram ultrapassados os principais obstáculos com que nos deparámos ao longo do desenvolvimento deste projeto.