Andrey d'Almeida Rocha Rodrigues

Visualização de modelos digitais de elevação em multiresolucão utilizando programação em GPU

Dissertação de Mestrado

Departamento de Informática
Programa de Pós–graduação em Informática

Rio de Janeiro
Abril de 2016

Pontifícia Universidade Católica
do Rio de Janeiro

Andrey d'Almeida Rocha Rodrigues

Visualização de modelos digitais de elevação em multiresolucão utilizando programação em GPU

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC-Rio.

Orientador: Prof. Waldemar Celes Filho

Rio de Janeiro
Abril de 2016

Pontifícia Universidade Católica
do Rio de Janeiro

Andrey d'Almeida Rocha Rodrigues

Visualização de modelos digitais de elevação em multiresolucão utilizando programação em GPU

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Waldemar Celes Filho
Orientador
Departamento de Informática — PUC-Rio

Prof. Luiz Henrique de Figueiredo
IMPA

Prof. Hélio Côrtes Vieira Lopes
PUC-Rio

Prof. Marcelo Gattass
PUC-Rio

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico — PUC-Rio

Rio de Janeiro, 7 de abril de 2016

Andrey d'Almeida Rocha Rodrigues

Possui graduação em Ciência da computação pela Universidade Federal de Alagoas (2014). É aluno de Mestrado da Pontifícia Universidade Católica do Rio de Janeiro, membro do instituto TECGRAF, onde desenvolve pesquisas na área de visualização em tempo real de terrenos utilizando programação em placas gráficas.

Ficha Catalográfica

Rodrigues, Andrey d'Almeida Rocha

Visualização de modelos digitais de elevação em multiresolucão utilizando programação em GPU / Andrey d'Almeida Rocha Rodrigues; orientador: Waldemar Celes Filho. — Rio de Janeiro : PUC-Rio, Departamento de Informática, 2016.

37 f: il. (color.); 29,7 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2016.

Inclui bibliografia.

1. Informática – Teses. 2. Modelo dittital de elevação. 3. Multiresolução. 4. Programação em GPU. I. Filho, Waldemar Celes. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Ao meu orientador Prof. Dr. Waldemar Celes Filho pela oportunidade pessoal e profissional, por todas as sinceras críticas e ensinamentos compartilhados.

Ao CNPq e o instituto TECGRAF, pelos auxílios financeiros e pelo espaço disponibilizado para tornar possível esta pesquisa.

A minha fiel companheira, Jéssica Guedes, por todo o apoio e paciência.

A todos os amigos do grupo de visualização do TECGRAF, pelos momentos de companherismo e alegria.

Ao Prof. Dr. Marcelo Gattass, Prof. Dr. Hélio Côrtes Vieira Lopes e o Prof. Dr. Luiz Henrique de Figueiredo por aceitarem participar da comissão julgadora desta defesa de mestrado, contribuindo para a minha formação.

Resumo

Rodrigues, Andrey d'Almeida Rocha; Filho, Waldemar Celes. **Visualização de modelos digitais de elevação em multiresolução utilizando programação em GPU**. Rio de Janeiro, 2016. 37p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A visualização eficiente de grandes modelos digitais de elevação continua sendo um desafio para aplicações em tempo real. O uso direto de novas tecnologias de triangulação em placas gráficas tem uma aplicabilidade limitada no gerenciamento dos níveis de detalhe para grandes modelos. Embora o hardware gráfico seja capaz de controlar a resolução do modelo de um modo bastante eficiente, todos os dados devem estar em memória. Isso compromete a escalabilidade de soluções simples baseadas em GPU para controlar o nível de detalhe. Neste trabalho, é proposto um novo algoritmo eficiente e escalável para lidar com grandes modelos digitais de elevação. A proposta combina efetivamente a triangulação em GPU com a gerência de ladrilhos em CPU, tirando proveito da capacidade de processamento da GPU ao mesmo tempo que mantém o uso de memória gráfica dentro dos limites práticos. Também é proposta uma técnica para gerenciar o nível de detalhe da imagem aérea mapeada sobre o modelo de elevação como texturas. Ambas gerências de níveis de detalhe (geometria e textura) executam separadamente, e os ladrilhos são combinados sem a necessidade de carregar qualquer dado adicional. O gerenciamento de níveis de detalhe é então estendido para lidar com modelos com bordas irregulares e buracos.

Palavras–chave

Modelo dititat de elevação; Multiresolução; Programação em GPU;

Abstract

Rodrigues, Andrey d'Almeida Rocha; Filho, Waldemar Celes (Advisor). **Multi-resolution visualization of digital elevation models using GPU shaders**. Rio de Janeiro, 2016. 37p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Efficient rendering of large digital elevation models remains as a challenge for real-time applications. The direct use of hardware tessellation has limited applicability for managing level of detail of large models. Although the graphics hardware is capable of controlling the resolution of patches in a very efficient manner, the whole patch data must be loaded in memory. This compromises the scalability of GPU-based naive solutions for controlling level of detail. In this work, we propose an efficient and scalable new algorithm for large digital elevation models. Our proposal effectively combines GPU tessellation with CPU tile management, taking full advantage of GPU processing capabilities while maintaining graphics-memory use under practical limits. We also propose a technique to manage level of detail of aerial imagery mapped on top of elevation models as textures. Both geometry and texture level of detail management run independently, and tiles are combined with no need to load extra data. The proposed level of detail management is then extended to handle model with irregular border and holes.

Keywords

Digital Elevation Model; Multi-resolution; GPU programing;

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Devido aos avanços em tecnologias para aquisição de dados, modelos digitais de elevação (MDE) com alta resolução, contendo gigabytes de dados, podem ser facilmente encontrados. Como consequência, técnicas eficientes e escaláveis para visualizá-los são mandatórias, e possuem várias aplicações práticas. Sem nenhum tipo de tratamento especial, esta quantidade de dados facilmente excede os limites atuais de hardware, tanto em espaço de memória quanto em capacidade de geração de triângulos.

De modo geral, trabalhos anteriores abordaram este problema utilizando técnicas de compressão [1, 2], gerenciamento de memória secundária [3–5], e níveis de detalhe (LOD) adaptativo de triangulações [6–8]. Utilizando essas estratégias, é possível visualizar MDE's com tamanhos razoáveis. Porém, devido ao gradativo crescimento dos modelos de entrada (MDE's) e à crescente resolução dos monitores, a geração e visualização fidedigna das geometrias associadas a esses modelos tem se tornado uma grande limitação de desempenho [9].

Por outro lado, a programação em GPU utilizando, *tessellation* shaders, tem evoluído, possibilitando a construção de geometrias complexas em tempo de execução[10] e eliminando a necessidade de gerar toda a geometria em CPU. Consequentemente evita-se a transferência de grande volume de dados da CPU para o pipeline gráfico a cada quadro. Embora já existam propostas fazendo o uso dos *tessellation shaders* para visualizar MDE's, essas não são escaláveis o suficiente para lidar com modelos reais. Além disso, falta na literatura análises sobre a influência dos tamanhos dos ladrilhos na CPU e na GPU no uso de memória e no desempenho.

Neste trabalho, é apresentada uma estratégia híbrida CPU-GPU para lidar com grandes modelos de elevação de modo escalável, ou seja: independente do tamanho do dado de entrada, o mesmo algoritmo é capaz de entregar uma visualização final do dado sem extrapolar os limites do uso de memória do hardware utilizado. Em uma etapa de pré-processamento, o modelo é subdividido em grandes ladrilhos, utilizando uma estrutura convencional de *quadtree*. Cada ladrilho é subdividido em uma série de blocos que serão utilizados como *patches* pelos *tessellation shaders*. Para cada triangulação

possível dos *patches*, é computado o erro no espaço do objeto. Na etapa de visualização, um algoritmo *multithread* é responsável por gerenciar os ladrilhos na memória principal, e selecioná-los baseado no erro geométrico projetado na tela. Cada ladrilho selecionado é responsável por enviar os respectivos *patches* para GPU, onde também é utilizado o erro do objeto em espaço de tela para escolher a quantidade de triângulos do *patch*. A subdivisão dos ladrilhos em *patches*, combinada com a triangulação adaptativa em GPU, nos permitiu utilizar a estrutura de *quadtree* desbalanceada: ladrilhos vizinhos ativos podem diferir em até seis vezes em tamanho, sem introduzir *T-vértices* na interface entre eles. Isto também diminui drasticamente a quantidade de ladrilhos a serem processados por quadro, se comparado a uma implementação de *quadtree* restrita.

Nosso trabalho estendeu a abordagem tradicional da visualização de MDE's. Além do LOD de geometria, foi criada uma extensão para gerenciar os níveis de detalhe da textura (imagem aérea) que cobre o MDE. O algoritmo utiliza o tamanho do ladrilho projetado em tela para selecionar os nós ativos. Ambos os algoritmos de LOD (geometria e textura) executam separadamente. Como resultado final, podemos ter um mapeamento 1-para-n ou n-para-1 entre os ladrilhos de geometria e textura; ambos os casos são tratados de forma simples, sem a necessidade de carregar nenhum dado adicional.

Também foi criada uma extensão para visualizar MDE's contendo áreas com dados nulos, formando buracos ou bordas. Neste caso, o cálculo do erro no espaço do objeto mede também o erro horizontal gerado pelo descarte dos dados nulos em cada nível de detalhe. Em tempo de execução, a quantidade de triângulos de um *patch* é definida pela combinação do erro vertical e horizontal projetado na tela. Os triângulos que se encontram dentro dos buracos são descartados e, finalmente, triângulos especiais são costurados nas bordas, preservando suas feições. Todas estas etapas são realizadas em GPU, sem a necessidade de dados adicionais.

O resto deste trabalho é organizado como segue. Inicialmente os trabalhos relacionados são comentados no Capítulo 2. Depois é apresentada a proposta deste trabalho e os resultados obtidos nos Capítulos 3 e 5 respectivamente. Por último as conclusões são apresentadas.

2
Trabalhos relacionados

Pode ser encontrada uma gama de trabalhos na literatura que visualizam eficientemente modelos digitais de elevação. *Pajarola and Gobbetti*[11] apresentam um compreensivo estudo sobre as abordagens de multi-resolução para visualizar terrenos digitais, analisando diferentes escolhas de estruturas de dados e métricas de erro. A ideia geral consiste em estruturar os dados em um modelo hierárquico, gerenciando os seus níveis de detalhe.

Abordagens anteriores optaram por hierarquias de árvores binárias. *Lindstrom et al.*[7] e *Duchaineau et al.*[8] propuseram o uso de árvores binárias para controlar a resolução da triangulação baseada no erro do objeto em espaço de tela. Embora simples de implementar, essas técnicas são limitadas pela transferência de memória para GPU, já que a geometria precisa ser enviada para GPU a cada quadro. Para contornar esta limitação, técnicas como BDAM [12] e P-BDAM [2] optaram por uma hierarquia baseada em *patches*: cada nó representa uma superfície de triângulos pré-computada ao invés de um único triângulo. O uso de *patches* é escalável e eficiente, porém sofre com a falta de adaptatividade na triangulação, processando mais triângulos do que o necessário. A técnica *Geometry clipmaps* [1] utiliza uma pirâmide de *mipmap* para representar os dados de terreno, e aplica uma técnica de compressão para reduzir a necessidade de armazenamento; porém, a técnica limita o cálculo do erro geométrico em espaço de tela, afetando a qualidade da imagem final.

Técnicas recentes vem tentando explorar os *tessellation shaders* para acelerar a geração de malhas. *Fernandes e Oliveira*[13] apresentam uma visualização de terrenos com níveis de detalhe adaptativos e livres de falhas. O método estima o tamanho de um *patch* no espaço de tela e o triangula visando atingir um tamanho constante em pixels para os triângulos gerados. *Cervin*[14] faz o uso de mapas de densidade para escolher o nível de detalhe apropriado para um *patch* no estágio de triangulação, produzindo mais triângulos em áreas de alta densidade (terreno irregular) e poucos triângulos em áreas com pouca densidade (terreno plano). No entanto, essas técnicas não calculam o erro geométrico de forma precisa, podendo gerar erros no resultado final do algoritmo.

O método desenvolvido por *Yusov and Shevtsov*[15] aplica uma técnica

de compressão nos dados para minimizar o custo de transferência, além de permitir a descompressão em GPU. O método gera a superfície do terreno em dois estágios: no primeiro, ladrilhos são escolhidos e carregados em GPU; no segundo estágio, cada ladrilho é subdividido em blocos e os mesmos são mapeado para *patches* e triangulados utilizando *tessellation shaders*. Neste aspecto, nossa proposta é semelhante a deles, mas optamos por evitar os *T-vértices* na construção da malha final; nós também utilizamos uma estratégia diferente para controlar efetivamente os erros em ambos ladrilhos de geometria e de textura. *Kang et al.*[16] também utilizam um esquema de *quadtree* para representar os dados de terreno, executando a triangulação com os *tessellation shaders*. Porém, eles optaram por construir uma quadtree para cada ladrilho do terreno, e, deste modo, limitando a escalabilidade.

3
Algoritmo proposto

O algoritmo aqui proposto é uma estratégia híbrida CPU-GPU para visualizar modelos digitais de elevação em multi-resolução. A CPU é responsável por gerenciar uma *quadtree* de ladrilhos, sendo estes de geometria (MDE) ou textura (imagem aérea). A GPU recebe as informações dos ladrilhos e os respectivos erros de geometria, e determina a quantidade de triângulos necessária para respeitar a tolerância ao erro.

Como dados de entrada para o algoritmo, são necessários: (i) O modelo digital de elevação e opcionalmente a respectiva imagem aérea; (ii) Os parâmetros para o pré-processamento dos dados (tamanho do ladrilho e do patch); (iii) O erro máximo tolerado (em pixels) da geometria final gerada na etapa de visualização; (iv) A quantidade máxima de memória de GPU a ser utilizada pelo algoritmo em tempo de execução. O tamanho do ladrilho e do *patch* afetam o uso de memória e o desempenho do algoritmo, como será mostrado no capítulo 5. Uma vez escolhidos os tamanhos do ladrilho e *patch* e o pré-processamento acabar, os mesmos não poderão mais ser trocados, sendo necessário um novo pré-processamento. Já o erro máximo tolerado e a quantidade de memória são escolhidos na etapa de visualização dos dados, podendo ser modificados em tempo de execução.

Para facilitar o entendimento do leitor, este trabalho foi dividido em duas partes: *LOD* de Geometria, onde é explicada a organização dos dados de elevação e o algoritmo de LOD na etapa de visualização; e *LOD* de Textura, explicando a execução independente do LOD da imagem aérea e como é feita a combinação dos ladrilhos.

3.1
LOD de geometria

O *LOD* da geometria tem como objetivo reduzir a quantidade de dados carregados em memória de GPU, mas sem afetar a qualidade final do desenho. A métrica utilizada no algoritmo de LOD é o erro geométrico no espaço do objeto, que é calculado em pré-processamento e, uma vez computado, só mudará caso a superfície do terreno seja alterada. Por outro lado, o erro no espaço de tela é a projeção do erro do objeto na tela e é diferente a cada

quadro, dependendo da posição da câmera em relação ao terreno.

Em CPU, é feita a seleção dos ladrilhos com menor resolução que garantam o erro tolerado em espaço de tela. Cada ladrilho é subdividido em um conjunto de *patches* (Figura 3.1), onde cada um pode ser subdividido independentemente na GPU.

3.1.1 Pré-processamento

O pré-processamento da elevação tem como objetivo construir uma hierarquia multi-resolução, calculando os erros no espaço do objeto para cada ladrilho. Na fase de pré-processamento, a *quadtree* é construída aplicando um procedimento *bottom-up*. Primeiro é necessário escolher o tamanho do ladrilho, e diferentes escolhas afetam a memória utilizada e o desempenho. O dado de elevação é inteiramente subdividido em ladrilhos representando as folhas da *quadtree*. Então, para cada quatro vizinhos, um ladrilho pai é criado com metade da resolução dos filhos. Este processo é repetido recursivamente até preencher todos os níveis da *quadtree*.

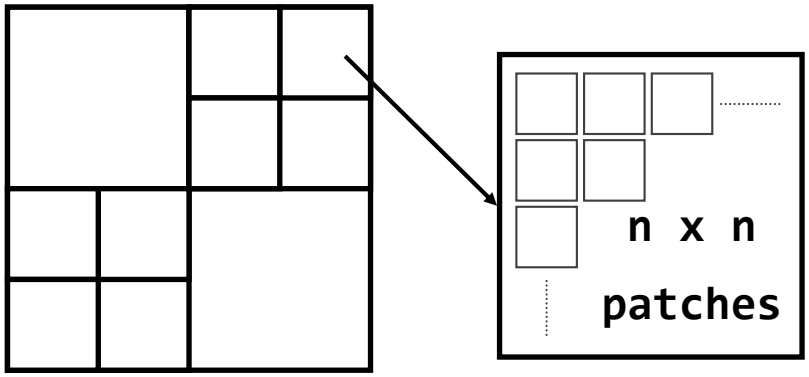


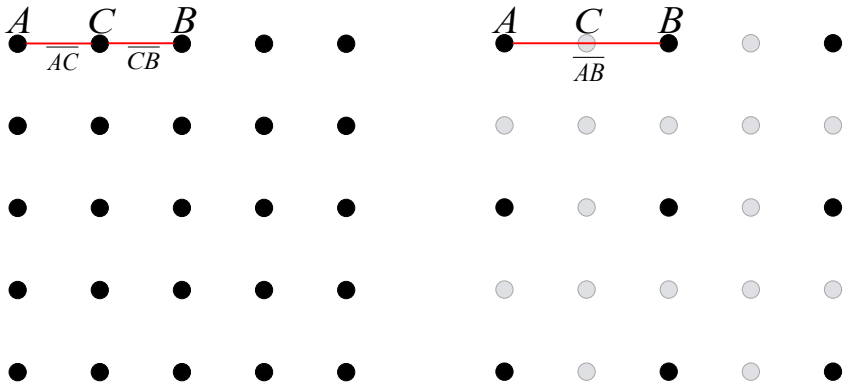
Figura 3.1 – Subdivisão dos ladrilhos. Cada ladrilho em CPU é subdividido em um número pré-definido de *patches*.

O tamanho de cada *patch*, e consequentemente o número de *patches* por ladrilho, também afeta o desempenho. O tamanho máximo é limitado pelo hardware gráfico. No Capítulo 5, são apresentados os testes computacionais feitos com diferentes tamanhos. Uma vez escolhido o tamanho do *patch*, são computados os erros no espaço do objeto associados a cada ladrilho e *patch*. O tamanho do ladrilho e do *patch* precisam ser de dimensão $2^n + 1$, pois ladrilhos (ou *patches*) vizinhos compartilham os mesmos pixels de borda. Por simplicidade, os valores de dimensão serão expressos em potência de 2 (2^n). Para ilustrar as discussões a seguir, vamos considerar que a dimensão do

ladrilho é 512 × 512 e a do *patch* é 64 × 64. Deste modo, cada ladrilho é subdividido em 8 × 8 *patches*.

Para cada ladrilho, é calculado o erro no espaço do objeto associado a cada *patch*, em diferentes triangulações. Para os ladrilhos folha, o erro de cada *patch* na triangulação máxima é zero. Nós simbolizamos esta propriedade por $\epsilon^l_{i_{64}} = 0$, onde i representa o *patch*, l o nível do ladrilho na *quadtree*, e 64 a triangulação do *patch* (64 × 64 triângulos). Em seguida, é computado o erro associado ao *patch* para cada simplificação geométrica: $\epsilon^l_{i_{32}}$, $\epsilon^l_{i_{16}}$, $\cdots$, $\epsilon^l_{i_1}$.

O erro de uma simplificação geométrica S ($\epsilon^l_{i_s}$) é o erro máximo de todas as suas arestas. Para cada aresta $\overline{AB}$ pertencente a S, o erro geométrico é o deslocamento gerado pela remoção das arestas $\overline{AC}$ e $\overline{CB}$, onde C é o vértice eliminado pela simplificação, como mostrado na Figura 3.2.



(a) Dado mais refinado (5x5) (b) Simplificação 3x3

Figura 3.2 – Erro Gerado pela simplificação do dado geométrico

O cálculo do erro percorre o dado simplificado testando todas as arestas possíveis: as verticais, horizontais e diagonais. Para cada uma é calculado o erro vertical, que mede o deslocamento da elevação introduzido pela remoção do vértice C, e pode ser calculado pela equação:

$$E_v = \left| \, elev(C) - \frac{elev(A) - elev(B)}{2} \, \right| \tag{3.1}$$

onde $elev(x)$ é o valor da elevação no vértice x.

O erro associado ao ladrilho é dado por: $E^l = \max_i \epsilon^l_{i_{64}}$, sendo naturalmente zero para os ladrilhos folha. O erros dos *patches* nos níveis superiores são herdados dos *patches* nos níveis abaixo com metade da resolução máxima:

$$\epsilon^{l-1}_{i_{64}} = \max_{i \in N} \epsilon^l_{i_{32}} \tag{3.2}$$

Onde N representa o conjunto dos quatro *patches* vizinhos associados ao *patch* pai. Este erro é propagado para as resoluções de triangulação mais baixas e, novamente, o erro associado ao ladrilho é $E^{l-1} = \max_i \epsilon^{l-1}_{i_{64}}$. Para cada ladrilho,

os erros de todos os *patches* são computados e guardados em uma textura para serem acessados no estágio de triangulação na GPU. A Figura 3.3 ilustra o processo de computação dos erros no espaço do objeto.

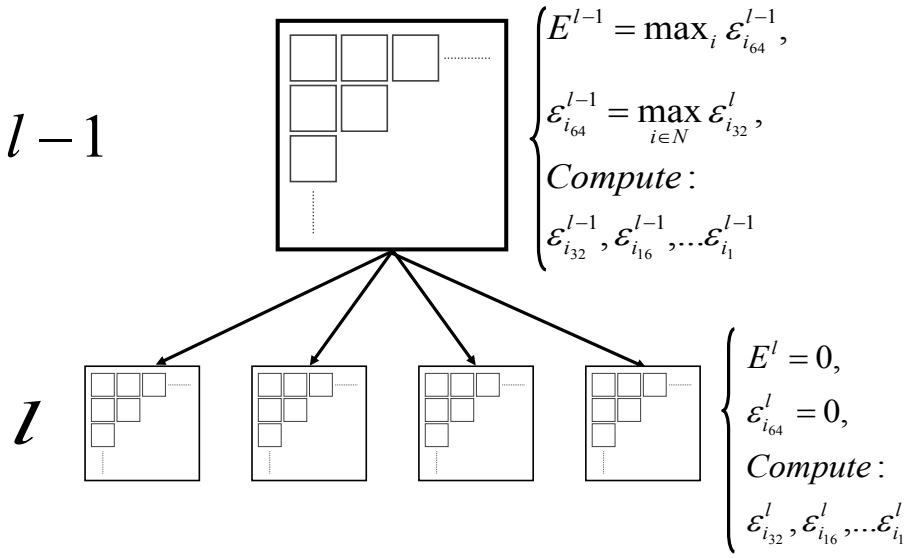


Figura 3.3 – Cálculo do erro no espaço do objeto: nos nós folha, o erro de cada *patch* na triangulação máxima é zero. Os erros dos *patches* nos níveis superiores são herdados dos níveis inferiores. O erro associado a cada ladrilho é o erro máximo dos seus *patches* na triangulação máxima.

3.1.2 Tempo de Execução

Em tempo de execução, o erro em espaço de tela ρ é computado do modo usual, tanto para selecionar os ladrilhos em CPU quanto para determinar o nível de triangulação de cada *patch* em GPU:

$$\rho = \lambda \frac{E}{d}, \quad \text{with} \quad \lambda = \frac{w}{2 \tan \frac{\theta}{2}} \tag{3.3}$$

sendo E o erro no espaço do objeto, w a resolução horizontal (em pixels) da tela, θ é o ângulo de visão e d a distância da câmera para o ponto mais próximo da caixa envolvente do ladrilho (ou *patch*).

Em CPU, o algoritmo de LOD é executado em duas threads: uma thread de carga e uma thread de visualização.

Thread de carga

A *thread de carga* é responsável por predizer o movimento da câmera e carregar os ladrilhos do disco para a GPU antecipadamente. O algoritmo de caminhamento na *quadtree* é *top-down*: iniciando do ladrilho raiz, o seu erro geométrico projetado em tela é avaliado; se o erro projetado for maior do que

o erro tolerado pelo usuário os filhos do ladrilho são processados, continuando o procedimento recursivo até obter o corte ativo, que possui os ladrilhos com o erro menor que o tolerado. Após o término do algoritmo de carga, todos os ladrilhos acima do corte ativo são carregados do disco para a GPU. Como o algoritmo possui duas threads acessando a mesma GPU, é necessária uma etapa de sincronização para evitar que a thread principal utilize o dado que está sendo carregado. Para esta sincronização, foi utilizado o objeto *GLSync* do *OpenGL* que informa quando o comando de *upload* para a GPU foi completado. Quando o ladrilho é finalmente carregado ele fica disponível para ser utilizado na *thread* de visualização.

Para garantir os limites aceitáveis de memória utilizada, a *thread* de carga também realiza um caminhamento para descartar os ladrilhos que não estão sendo utilizados. Neste caminhamento, o algoritmo descarrega da memória de GPU todos os ladrilhos que não foram utilizados nos últimos n quadros, até atingir a quantidade de memória aceitável. O valor de n utilizado neste trabalho foi de 1000 quadros.

Thread de Visualização

A *thread de visualização* é responsável por selecionar os ladrilhos, dentre os que estejam carregados, necessários para visualizar o terreno honrando a tolerância ao erro no quadro corrente [3]. A seleção dos ladrilhos é um procedimento *top-down* similar a *thread* de carga, com a restrição de caminhar somente nos carregados, como ilustrado no Algoritmo 1.

Algorithm 1 TRAVERSE(tile)

```
1: if LOADED(tile) and PROJECTEDERROR(camera, tile) < TOL then
2:    SelectTile(tile)
3: else
4:    TRAVERSE(tile.child_nw)
5:    TRAVERSE(tile.child_ne)
6:    TRAVERSE(tile.child_sw)
7:    TRAVERSE(tile.child_se)
8: end if
```

Quando os ladrilhos de um quadro são selecionados, a CPU é responsável por enviar os seus *patches* para a GPU. O nível de detalhe dos *patches* são calculados no *tessellation shader*. A triangulação interna de um *patch* é calculada utilizando o erro em espaço de tela do *patch*, escolhendo a menor resolução possível que assegure o erro tolerado. A escolha dos níveis externos de triangulação precisa levar em conta os *patches* adjacentes, para evitar os *T-vértices*. Existem dois casos, aqui chamados de *patch-patch*, para as interfaces

entre dois *patches* adjacentes de um mesmo ladrilho, e o caso *ladrilho-ladrilho*, para interfaces entre *patches* de ladrilhos diferentes. O caso *patch-patch* é simples de lidar, pois todos os dados necessários estão carregados com o ladrilho na memória. O nível de triangulação é dado pela subdivisão máxima requisitada pelos *patches* que compartilham a mesma interface. Deste modo o *patch* menos refinado ajusta sua triangulação externa:

$$n_{outer} = \max(n_i, n_j) \tag{3.4}$$

onde n_i e n_j representam o nível de subdivisão interno dos *patches* vizinhos.

O desafio está no caso ladrilho-ladrilho, já que os mesmos são computados separadamente e, deste modo, informações adicionais são necessárias. Para superar este problema, em CPU, após o algoritmo de LOD (seleção de ladrilhos), para cada ladrilho, as diferenças de níveis entre ladrilhos ativos vizinhos são anotadas, uma para cada borda. Se o nível do ladrilho adjacente for igual ou maior (o adjacente é mais refinado), o valor zero é anotado; se o nível do ladrilho adjacente for menor (menos refinado), é anotada a diferença de nível entre eles: $l_{ladrilho} - l_{adj}$. A Figura 3.4 ilustra os valores anotados para um exemplo de *quadtree* ativa. No *tessellation shader*, o nível de triangulação externo para as interfaces ladrilho-ladrilho é dado por:

$$n_{outer} = \frac{n_{max}}{2^{\delta}} \tag{3.5}$$

onde n_{max} é o número máximo de subdivisões do *patch* e δ é a diferença de nível anotada. Esta estratégia impõe um limite no desbalanceamento de ladrilhos adjacentes no corte ativo da árvore mensurado por $\log_2 n_{max}$, o que, na prática, não chega a ser um problema. Deste modo, todos os *patches* de um ladrilho possuem o mesmo número de subdivisões em cada borda. Nota-se que esta abordagem faz com que as interfaces entre os ladrilhos fiquem refinadas na melhor triangulação possível. Isto contribui para um aumento no número de triângulos gerados, porém torna simples o tratamento de *T-vértices*. Além disto, elimina-se a necessidade de utilizar procedimentos adicionais para preencher possíveis rachaduras geradas quando houver presença de *T-vértices*, como adição de bordas [15], costuras especiais [16] ou faixas verticais ao redor do ladrilho[17].

3.2 *LOD* de textura

Por cima da superfície em multi-resolução do terreno, é feito o mapeamento das imagens aéreas como textura. Ao mesmo tempo, o nível de detalhe da imagem aérea é gerenciado independente do LOD da geometria. O desa-

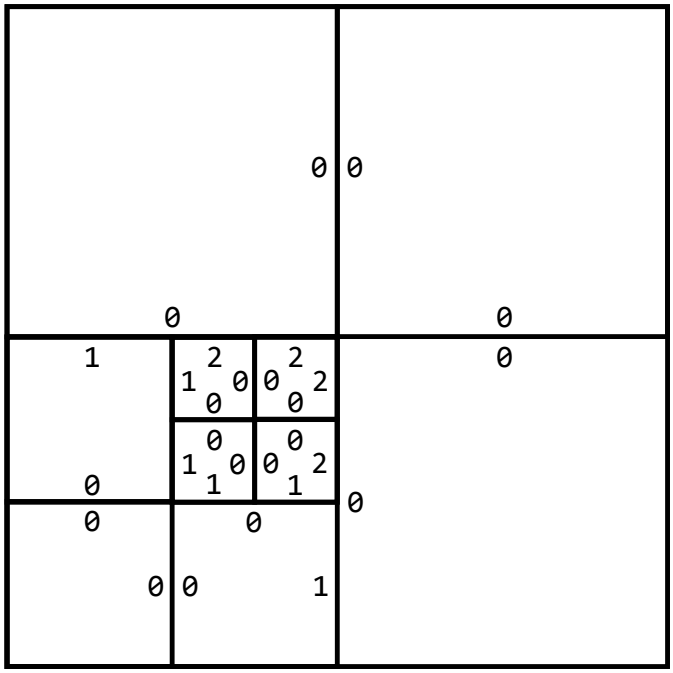


Figura 3.4 – Diferença de níveis ao longo da borda do ladrilho. Para cada ladrilho selecionado, os ladrilho vizinhos são checados; se o vizinho for do mesmo nível ou maior, a diferença é zero. Caso o ladrilho vizinho seja de um nível menor, a diferença é computada.

fio aqui reside em combinar os ladrilhos ativos de geometria e textura sem a necessidade de carregar nenhum dado adicional.

Na etapa de pré-processamento, uma estrutura convencional de *quadtree* é criada para armazenar a textura em diferentes resoluções. Em tempo de execução, novamente, uma segunda thread é utilizada para predizer, selecionar, e carregar os ladrilhos de textura necessários para os próximos quadros. A thread de desenho seleciona os ladrilhos necessários em cada quadro, considerando somente os que estão carregados.

A seleção dos ladrilhos de textura também utiliza o procedimento *top-down* mostrado no Algoritmo 1. Para cada ladrilho visitado, é computado o maior tamanho horizontal da sua caixa envolvente em espaço de tela, L_{proj}, e em seguida a taxa de ampliação:

$$r_{mag} = \frac{L_{proj}}{w_{ladrilho}} \tag{3.6}$$

onde $w_{ladrilho}$ representa a largura do ladrilho em pixels. Em nossos experimentos, o limite utilizado foi de 1.5: se a taxa de ampliação for menor que 1.5, o ladrilho é selecionado; senão, os seus quatro filhos são processados.

Como a seleção de ladrilhos executa de modo independente para a geometria e textura, nós acabamos com três casos possíveis de correspondência entre os ladrilhos: 1-para-1, n-para-1 e 1-para-n. A Figura 3.5 ilustra as três correspondências possíveis.

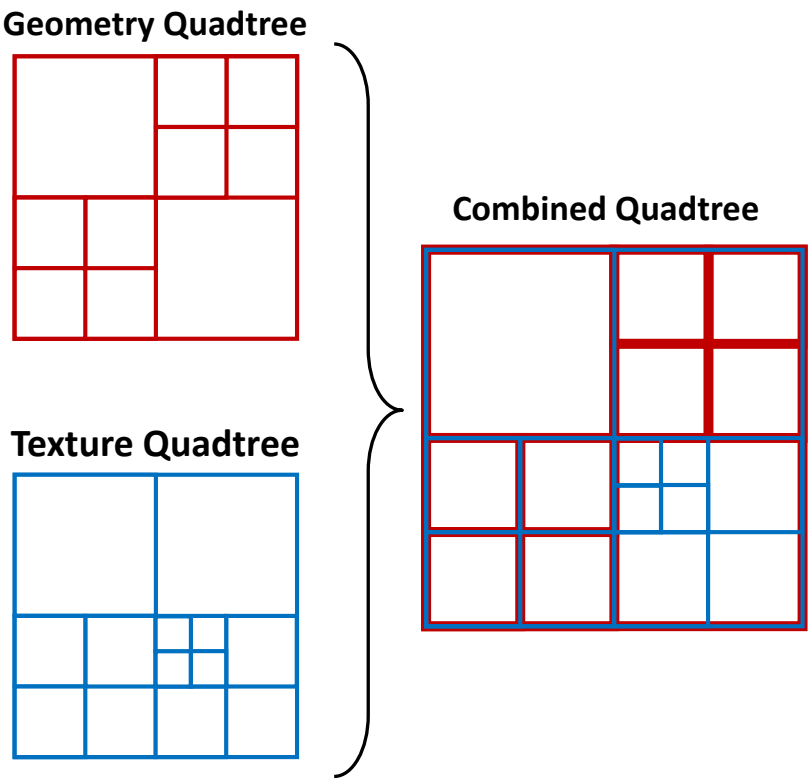


Figura 3.5 – Correspondência entre os ladrilhos de geometria(vermelhos) e de textura(azuis): nos quadrantes da esquerda a correspondência é de 1-para-1; o quadrante superior direito configura uma correspondência n-para-1 e, no quadrante inferior direito a correspondência 1-para-n, a qual requer o uso do *ladrilho virtual de geometria* para ser desenhado.

O caso 1-para-1 é o mais direto; nós temos um ladrilho de geometria para um de textura: os vértices dos *patches* recebem as coordenadas de textura de 0.0 a 1.0. O caso n-para-1 também é simples: existem n ladrilhos de geometria mapeados em um ladrilho de textura. As coordenadas de textura são transformadas de acordo, e todos os ladrilhos de geometria utilizam o mesmo objeto de textura.

O caso 1-para-n é o que requer uma solução mais elaborada: nós temos um ladrilho de geometria que precisa ser desenhado com vários objetos de textura. Deste modo, é necessário desenhar o ladrilho de geometria múltiplas vezes, um para cada ladrilho de textura. Para solucionar este problema nós vemos duas abordagens. Na primeira, para cada desenho, somente os *patches* que estivessem dentro do ladrilho de textura seriam desenhados. Embora pareça direta, esta solução apresenta duas grandes desvantagens. (i) o fator de correspondência n, seria limitado pelo número de *patches* em um ladrilho de geometria; (ii) um algoritmo especializado teria que ser desenvolvido para lidar com ladrilhos com menos *patches* do que o normal.

Nós optamos por um segunda solução: para cada ladrilho de textura

um *ladrilho virtual de geometria* é gerado. Cada ladrilho virtual é subdividido com o mesmo número de *patches* que um ladrilho regular, deste modo não é necessário modificar o algoritmo. O que diferencia o ladrilho virtual dos demais é que ele utiliza somente um subconjunto dos dados de elevação; as coordenadas de textura para acessá-los são computadas de acordo, assim como para acessar os dados de erro. A Figura 3.6 ilustra como esta proposta lida com o caso 1-para-n. Pode-se notar que o nível de detalhe requisitado para a textura (ladrilhos azuis) é maior que a geometria (ladrilhos vermelhos).

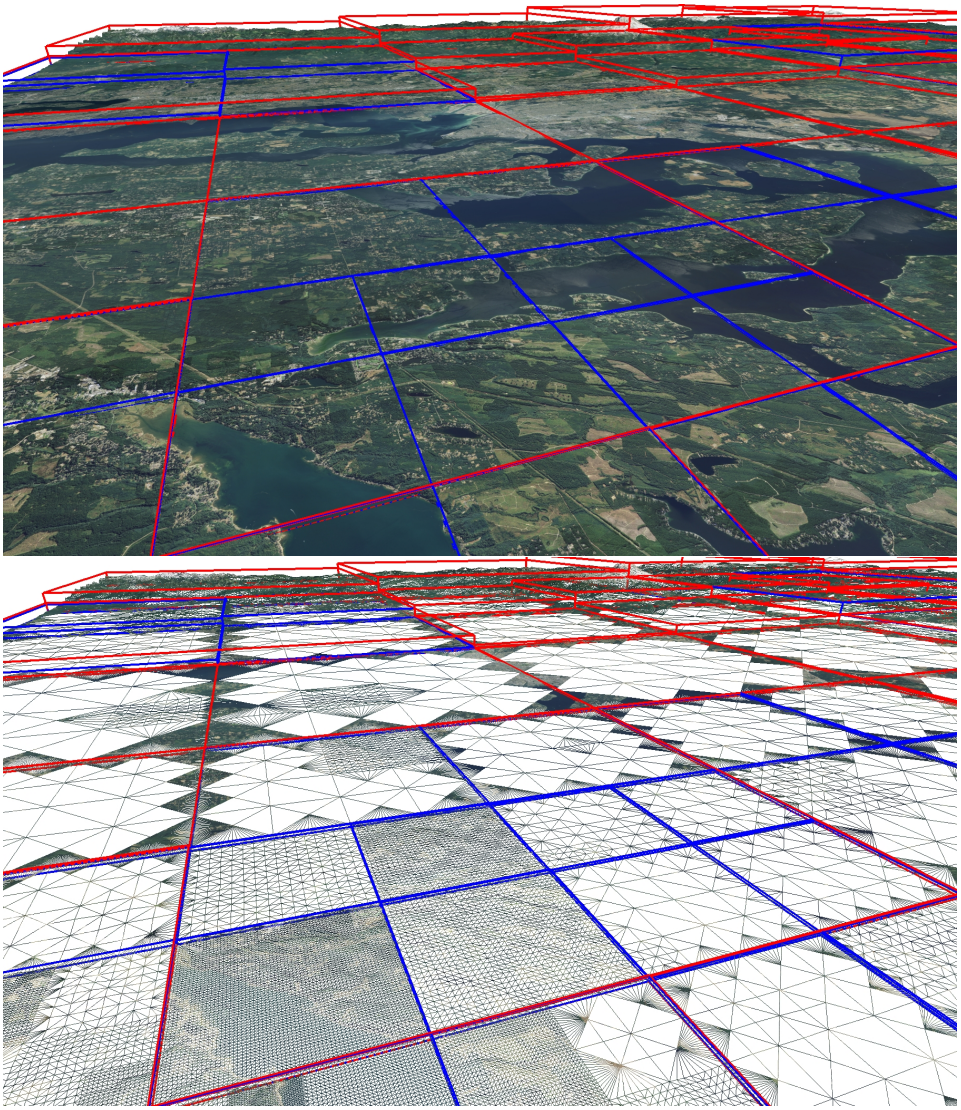
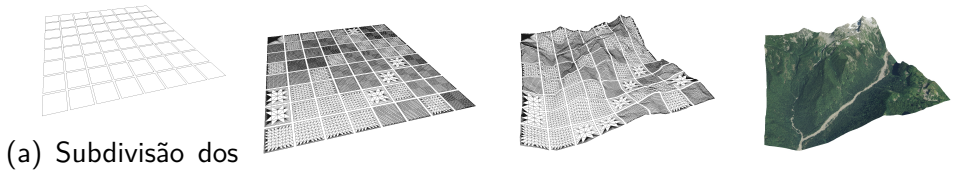


Figura 3.6 – Correspondência 1-para-n: um ladrilho de geometria para múltiplos de textura. Um *ladrilho virtual de geometria* é desenhado para cada ladrilho de textura; todos os ladrilho virtuais compartilham o mesmo dado de elevação.

3.3 Construção final do terreno

Com a estratégia apresentada, a CPU possui todas as informações necessárias para enviar os comandos de desenho para a GPU. Para cada ladrilho de geometria (inclusive os virtuais), é enviado um comando de desenho para a GPU contendo o ladrilho de textura correspondente e os erros de cada *patch* (armazenados em textura). A Figura 3.7 ilustra o pipeline da GPU após o envio de cada comando.



(a) Subdivisão dos ladrilhos em *patches* (b) Triangulação dos *patches* (c) *Displacement map* (d) Mapeamento de textura

Figura 3.7 – Pipeline de GPU da nossa proposta: A CPU seleciona e envia os ladrilhos para a GPU. Para cada *patch*, os níveis de triangulação são selecionados; cada vértice gerado recebe a sua altura respectiva, e a textura é mapeada sobre os triângulos gerados.

Inicialmente os *patches* enviado são tratados no *vertex shader*, colocando-os em suas posições devidas; em seguida cada *patch* é tratado pelo *tessellation control shader*, onde são calculadas as subdivisões internas e externas de cada *patch* seguindo os critérios para evitar *T-vértices*. No próximo estágio (*tessellation evaluation shader*), cada vértice gerado pela triangulação recebe sua elevação correspondente(*displacement map*). Finalmente, no *fragment shader* é feito o mapeamento da textura relativa a imagem aérea.

4
Modelos com bordas e buracos

Quando o modelo digital de elevação possui dados nulos, representando buracos e bordas irregulares, é necessário um tratamento especial para garantir a fidedignidade da visualização final gerada. Para isso, este trabalho propõe duas modificações no algoritmo: Um cálculo de erro no espaço do objeto específico para modelos com bordas; e uma triangulação especial para ressaltar as silhuetas das bordas e buracos, dando-lhes uma melhor aparência final.

4.1
Erro horizontal

Apesar do erro vertical produzir resultados precisos quando utilizado na visualização de terrenos, ele apresenta uma grande limitação quando é unicamente utilizado para visualizar terrenos com buracos. Para tratar esses modelos, introduzimos o cálculo de erro horizontal na etapa de pré-processamento.

A estratégia geral do erro horizontal é a mesma do vertical: para cada simplificação de um *patch* é calculado o erro horizontal e, este é propagado para os *patches* dos ladrilhos superiores até completar o cálculo em toda a *quadtree*, como ilustrado na Figura 3.3. Novamente, o cálculo do erro horizontal percorre todas as possíveis arestas de uma simplificação (horizontais, verticais e diagonais), avaliando o deslocamento da silhueta do buraco, quando presente, gerado pela retirada do vértice C e inclusão da aresta $\overline{AB}$, como ilustrado na Figura 4.1. Para o cálculo, é considerado que a silhueta é posicionada no ponto médio de dois vértices, um válido e outro inválido.

Quando todos os vértices (A, C e B) possuem dados de elevação ($\bullet$) ou são todos nulos ($\circ$) o erro horizontal é zero. Para todas as outras combinações possíveis, o erro horizontal é calculado segundo a Tabela 4.1.

4.2
Visualização

Na etapa de visualização, o tratamento de buracos e bordas adiciona a informação do erro horizontal calculada anteriormente. O erro no espaço do objeto, agora é dado pelo tamanho do vetor resultante da combinação do

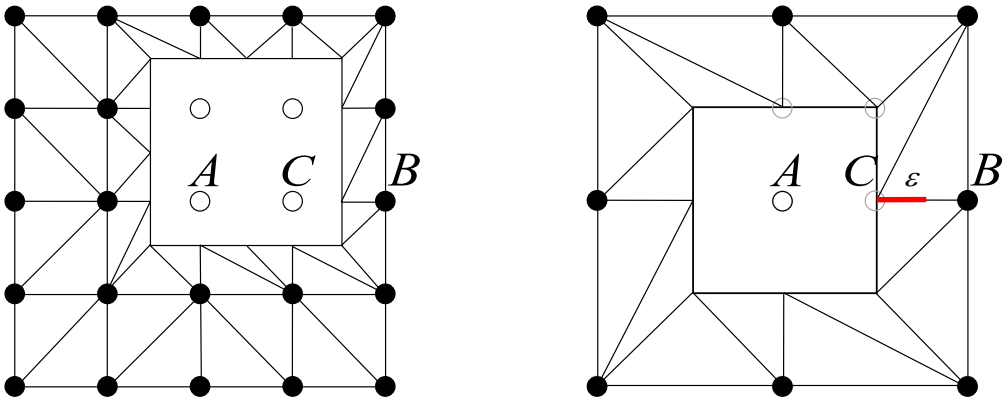


Figura 4.1 – Cálculo do erro horizontal. A esquerda o dado mais refinado, seguido de um simplificação. Com a eliminação do vértice C, a silhueta do buraco sofre um deslocamento em sua reconstrução, representado por ϵ (em vermelho).

Tabela 4.1 – Tabela para calcular os errors horizontais e verticais no espaço do objecto para MDE's contendo ausência de informação. A variável $hsize$ representa o tamanho do segmento de reta $\overline{AC}$ ou $\overline{CB}$

Caso $(A - C - B)$	Arestas horizontais/verticais	Arestas diagonais
$\bullet - \bullet - \circ$	$hsize/2$	$\sqrt{2} \cdot hsize/2$
$\bullet - \circ - \bullet$	$hsize$	$\sqrt{2} \cdot hsize$
$\bullet - \circ - \circ$	$hsize/2$	$\sqrt{2} \cdot hsize/2$
$\circ - \bullet - \bullet$	$hsize/2$	$\sqrt{2} \cdot hsize/2$
$\circ - \bullet - \circ$	$hsize$	$\sqrt{2} \cdot hsize$
$\circ - \circ - \bullet$	$hsize/2$	$\sqrt{2} \cdot hsize/2$

erro vertical e horizontal: $\epsilon_{final} = length(vec2(\epsilon_h, \epsilon_v))$. Consequentemente, a avaliação do erro na tela é modificado tanto no ladrilho (CPU) quanto no *patch* (GPU). Deste modo espera-se que a triangulação final do objeto gere triângulos suficientes para preservar as silhuetas dos buracos e bordas.

Após a geração dos triângulos, os que têm todos os vértices nulos podem ser descartados, enquanto os que possuem todos os vértices válidos são mantidos. Restam então os triângulos que tem tanto vértices nulos quanto válidos. Para estes casos, foi definido que a silhueta de um buraco ou borda está situada no ponto médio entre dois vértices: um vértice com informação de altura válida e o outro inválida. Esta escolha foi feita com base em testes experimentais onde se obteve silhuetas mais suaves ao se utilizar o ponto médio.

Foram implementados dois algoritmos para lidar com as silhuetas. O primeiro as cria geometricamente quando necessário, utilizando o *Geometry shader*. Existem quatro casos que precisam ser identificados e tratados apropriadamente. Os dois mais simples são os casos onde todos os vértices são nulos $A_\circ, B_\circ$ e $C_\circ$ ou todos são válidos $A_\bullet, B_\bullet$ e $C_\bullet$. Para estes, o triângulo é

enviado quando todos os vértices são válidos, e descartado caso todos sejam nulos; como ilustrado nas Figuras 4.2a e 4.2b, respectivamente. Os outros dois casos ocorrem quando um ou dois vértices do triângulo são nulos, significando que o mesmo pertence a uma área de borda.

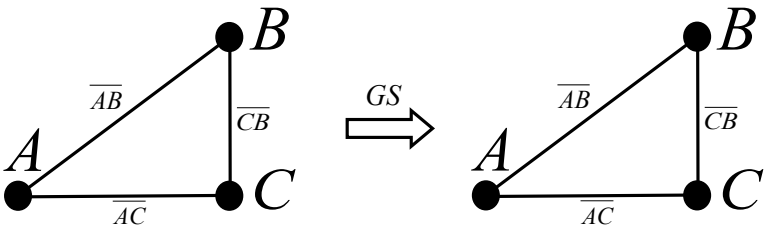
Nos casos onde um vértice é valido e dois são nulos ($A_\bullet, B_\circ$ e $C_\circ$), os vértices borda são dados pelos pontos médios das arestas $\overline{AC}$ e $\overline{AB}$, representados por C' e B', respectivamente. A altura de ambos é definida como igual a do vértice A. Por fim, o triângulo resultante é formado pelos vértices válidos A, B' e C' (Figura 4.2c).

Para os casos onde dois vértices são válidos e um é nulo ($A_\circ, B_\bullet$ e $C_\bullet$), a borda é formada pelos vértices C' e B' e as alturas dos novos vértices são $elev(C)$ e $elev(B)$, respectivamente. Como resultado, temos quatro vértices válidos e, portanto, precisamos gerar dois triângulos para representá-los corretamente: $\{C', B', B\}$ e $\{C', B, C\}$ (Figura 4.2d).

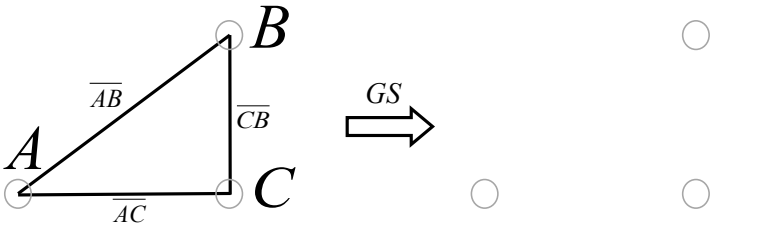
O *geometry shader* (GS) é executado após o *tessellation evaluation shader*, interceptando todos os triângulos gerados pela GPU e fazendo as alterações necessárias para gerar corretamente as bordas.

O segundo algoritmo para reconstruir as bordas dispensa o uso do *geometry shader* e utiliza o descarte de fragmentos para gerar as silhuetas. Inicialmente é feito o preenchimento de todos buracos em CPU, identificando os vértices que são nulos, para serem usadas pela GPU.

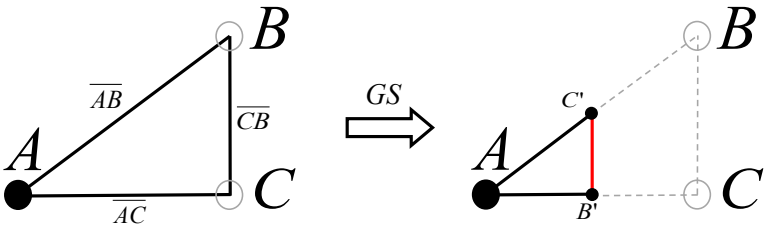
Em GPU, cada vértice gerado pela triangulação recebe um atributo tipo *float*, sendo 0 para vértices nulos e 1 pra vértices válidos. Estes atributos são interpolados ao longo do triângulo gerado e, no *fragment shader*, os fragmentos que tiverem valores para este atributo menores ou iguais a 0.5 são descartados, gerando a silhueta exatamente entre os vértices desejados.



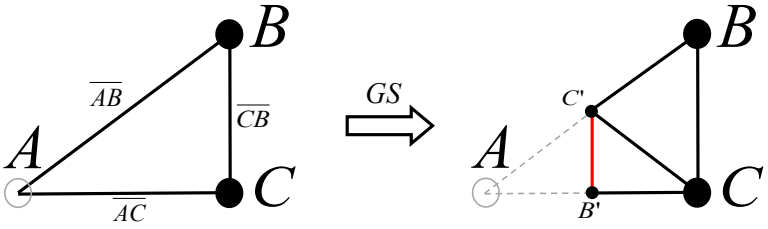
(a) Caso $A_\bullet$, $B_\bullet$ e $C_\bullet$



(b) Caso $A_\circ$, $B_\circ$ e $C_\circ$



(c) Caso $A_\bullet$, $B_\circ$ e $C_\circ$



(d) Caso $A_\circ$, $B_\circ$ e $C_\bullet$

Figura 4.2 – Possíveis configurações dos vértices no algoritmo de construção das silhuetas no *geometry shader*.

5
Resultados e discussões

Neste capítulo, será descrito uma série de experimentos computacionais que foram usados para testar e ajustar o método proposto. Todos os experimentos foram executados em um processador i7-3960X com 24GB de RAM, equipado com uma placa gráfica NVIDIA GeForce GTX 760 com 2GB de VRAM. Como dados de entrada, foi utilizada uma versão em alta resolução, com 64k $\times$ 64k, da base de dados *Puget Sound*, e um modelo digital de elevação extraído de um horizonte sísmico, com resolução de 2k $\times$ 2k, com uma quantidade grande de buracos ao longo da superfície. A resolução de tela utilizada nos testes foi de 1920 $\times$ 1080 e a tolerância máxima ao erro geométrico foi de 0.7 pixels para todas as análises de desempenho. Para executar os experimentos, foi definido um caminho de câmera sobre o terreno, cruzando áreas planas e montanhosas durante o caminho.

5.1
Puget Sound

O dado de elevação *Puget Sound*, disponível em [18], foi utilizada para validar o algoritmo descrito neste trabalho. Este dado possui uma vasta área plana em sua superfície, mas ao mesmo tempo regiões íngrimes e montanhosas, sendo um dado que explora vários aspectos do LOD de terrenos. Além disso, este é um dado muito utilizado por outros trabalhos relacionados. A versão do dado utilizada (65k $\times$ 65k) possui 2GB de dados de elevação e 8GB da imagem de satélite. O dado pré-processado com o tamanho de ladrilho 512 e *patch* 64 contém 17GB de elevação e 22GB de imagem de satélite, totalizando 39GB de dados.

O caminho de câmera definido sobrevoa o terreno em áreas planas e montanhosas, em diferentes altitudes. A Figura 5.1 mostra o caminho realizado visto de cima.

Inicialmente, foi analisada a influência da escolha do tamanho do ladrilho de geometria. Conforme o tamanho do ladrilho é aumentado, a altura da *quadtree* é reduzida, aliviando a carga de CPU. Por outro lado, aumentando o tamanho do ladrilho, a granularidade do nível de detalhe é perdida, sendo necessário carregar mais dados do que são realmente necessários, e consequen-

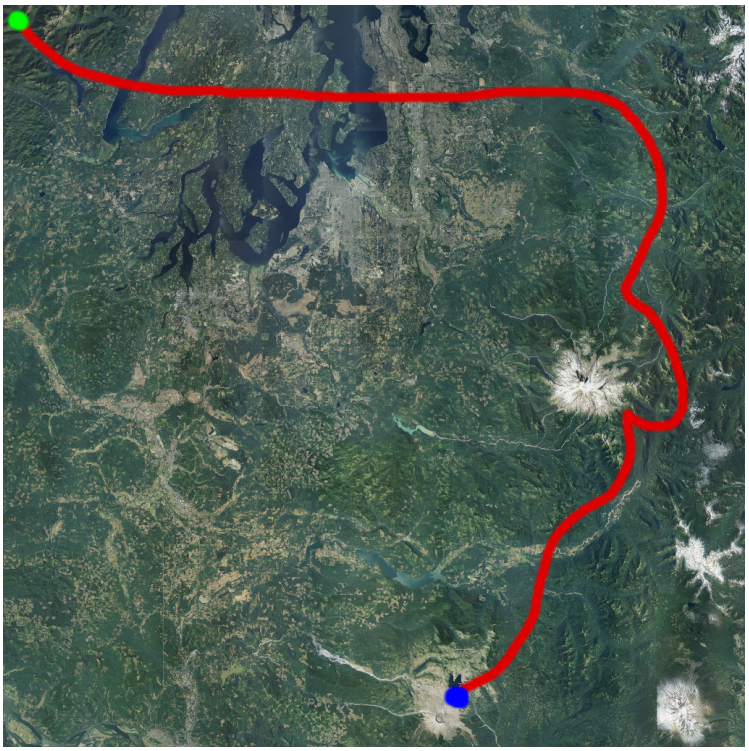


Figura 5.1 – Caminho de câmera dos testes. O círculo verde é o início do caminho e o azul representa o final

temente aumentando o uso de memória em GPU. O mesmo experimento foi repetido com três diferentes tamanhos de ladrilhos: 256×256, 512×512, e 1024×1024. Foram mensurados a memória de GPU e o número de ladrilhos ativos para cada quadro, ao longo do caminho da câmera. O número de ladrilhos ativos está diretamente relacionado com a quantidade de trabalho a ser realizado pela CPU. A quantidade de memória de GPU também afeta o desempenho, pois é necessário transferir mais memória para a placa gráfica. Pode ser notado nos gráficos da Figura 5.2, que ladrilhos 1024×1024 requerem uma grande quantidade de memória de GPU; em contrapartida, ladrilhos de 256×256 podem deixar a aplicação limitada por CPU. Os melhores resultados foram alcançados com ladrilhos de de 512×512, os quais foram usados nos demais testes.

Em um segundo teste, foi variado o tamanho do *patch*; foram testados *patches* de 16×16, 32×32, e 64×64 (o limite da placa gráfica utilizada). Conforme o tamanho do *patch* diminui, são necessários mais *patches* para cobrir um ladrilho. Por um lado, tamanhos menores de *patch* aumentam a granularidade, diminuindo o número de triângulos gerados. Por outro lado, *patches* maiores diminuem a quantidade de *patches* a serem processados. O experimento registrou o número de triângulos gerados ao longo do caminho da câmera. A Figura 5.3 mostra os gráficos. *patches* de 64×64 aumentam

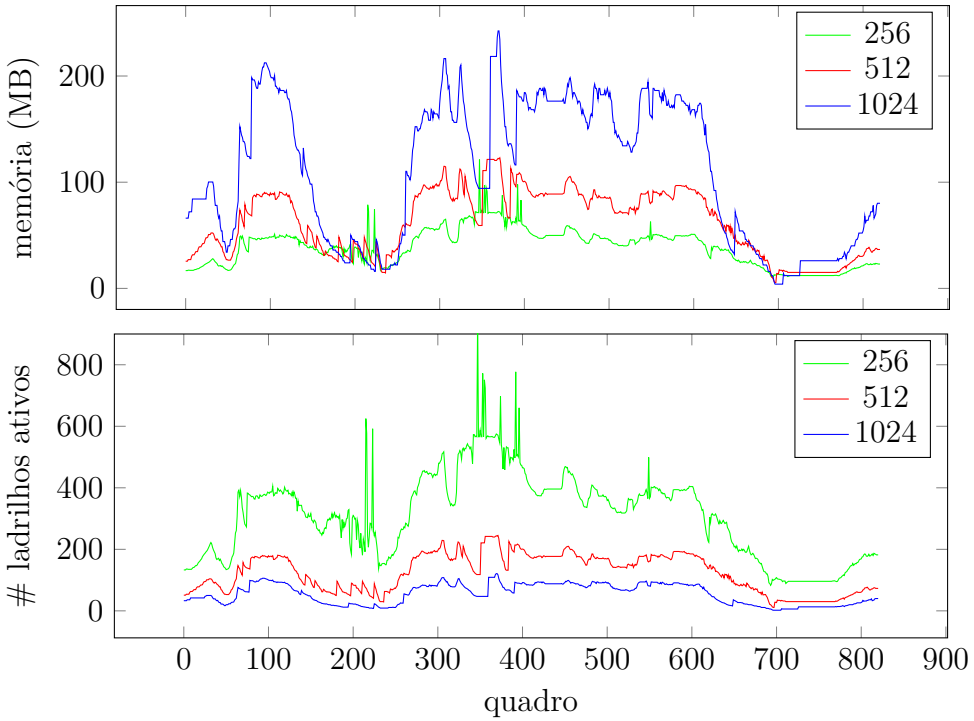


Figura 5.2 – A influência do tamanho do ladrilho na quantidade de memória de GPU necessária: ladrilhos pequenos demandam menos memória, porém aumentam a quantidade de trabalho em CPU.

o número de triângulos gerados, porém registram uma melhor performance. Como não existe penalidade de memória, 64×64 é a melhor escolha para este teste. Como consequência, o desbalanceamento máximo para a *quadtree* é $6(\log_2 64)$. Nos nossos testes este desbalanceamento nunca foi alcançado.

Finalmente, para ilustrar o algoritmo em ação, o experimento foi repetido utilizando ladrilhos de 512×512 e *patches* de 64×64, e algumas capturas de tela foram registradas ao longo do caminho, ilustradas na Figura 5.4. Pode-se observar que em regiões montanhosas, é preciso mais ladrilhos de geometria mas não necessariamente mais ladrilhos de textura; neste caso tendemos a ter o caso de correspondência n-para-1 de geometria-textura. Em regiões planas, como ilustrado na Figura 5.4b, nós temos menos ladrilhos de geometria, configurando o caso 1-para-n de correspondência. O método proposto funciona bem nos dois casos, entregando imagens de alta qualidade com o uso dos *tessellation shaders* da placa gráfica.

5.2 Horizonte sísmico

O horizonte sísmico foi utilizado para avaliar a qualidade final das silhuetas do buracos utilizando o erro horizontal. Nos testes do horizonte sísmico, foi utilizado um mesmo caminho de câmera, avaliando a diferença entre

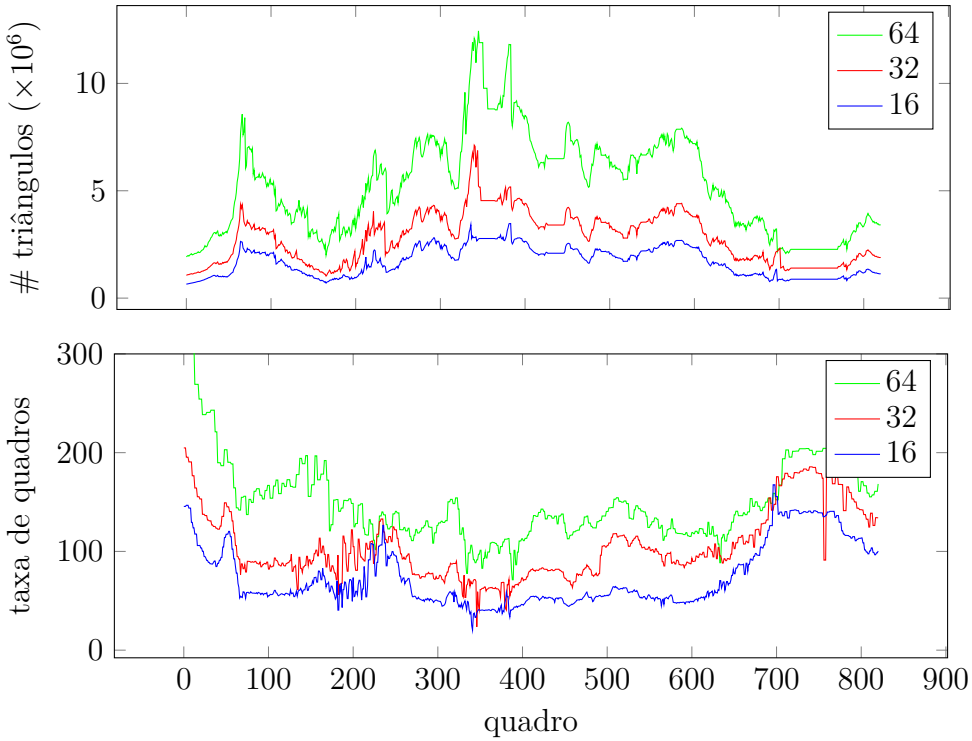
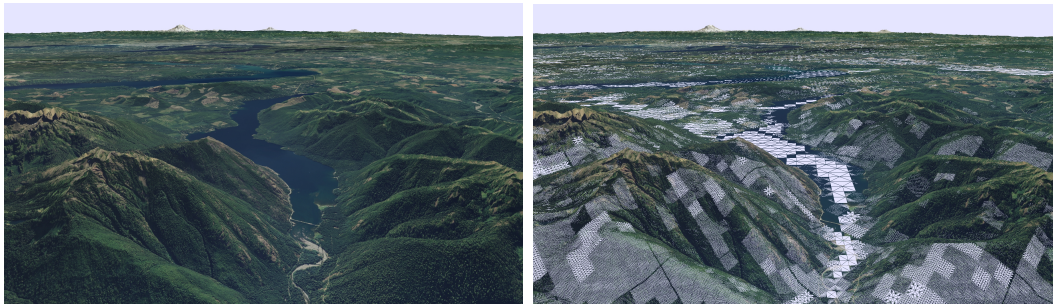


Figura 5.3 – Influência do tamanho do *patch* no número de triângulos gerados e na performance ao longo do caminho: *patches* maiores aumentam o número de triângulos gerados, porém entregam um melhor desempenho. A memória de GPU usada é a mesma.

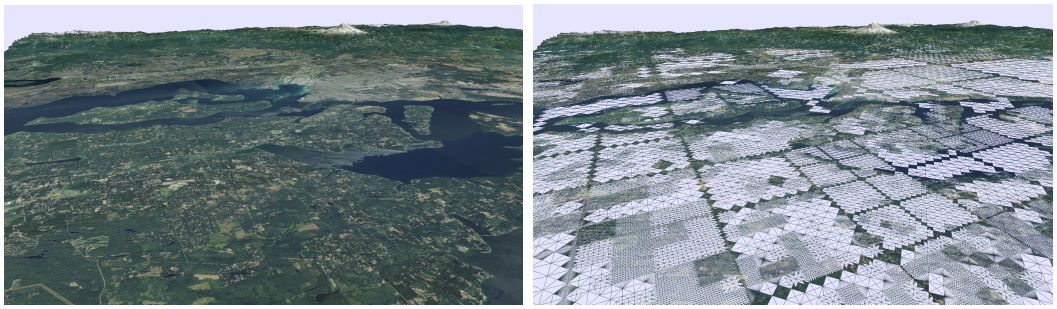
as duas técnicas de geração de silhuetas: no *geometry shader* e no *fragment shader*. Como o dado é relativamente pequeno (2049 × 2049), um tamanho de *patch* de 8 × 8 foi utilizado para permitir uma maior granularidade do *LOD*.

No quesito desempenho, o descarte em fragmento mostrou-se muito mais vantajoso em relação ao *GS*, como ilustrado na Figura 5.5. Como a quantidade de triângulos gerada por ambos os métodos é basicamente a mesma, a diferença de desempenho explica-se pelo *overhead* gerado com o uso do GS, pois o mesmo recebe todos os triângulos do estágio de *tessellation*, tendo que processá-los e gerá-los novamente para o estágio de *shading*.

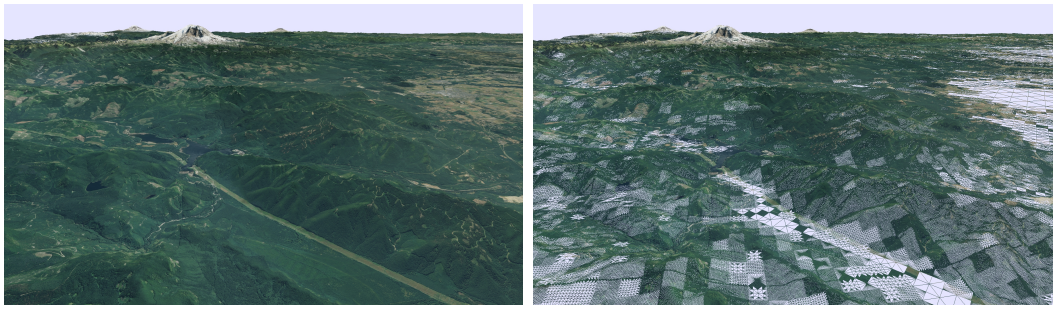
A fidedignidade das silhuetas foi avaliada pela diferença entre as imagens geradas utilizando LOD e a imagem da malha em resolução máxima. Quando a tolerância ao erro foi colocada em 0.7 pixels, a diferença entre as imagens foi nula, utilizando cerca de 1/3 dos triângulos da malha mais refinada para todo o caminho de câmera. As imagens na Figura 5.7 mostram capturas de tela em *wireframe* e com preenchimento, mostrando a diferença na quantidade gerada de triângulos, porém sem alterar o resultado final da imagem. A diferença na quantidade de triângulos gerada entre as malhas é mostrada no gráfico da Figura 5.6.



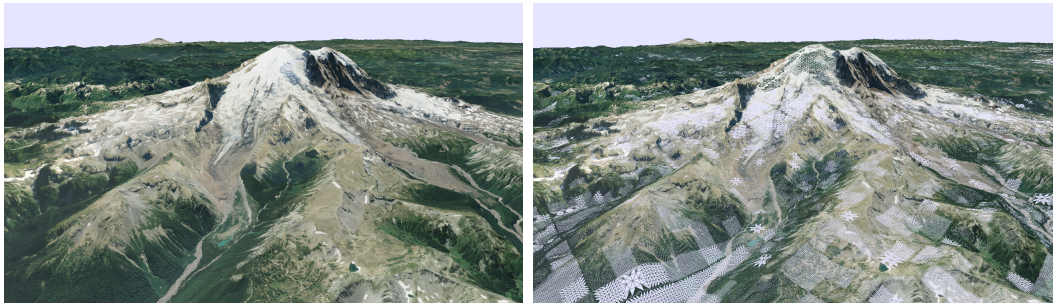
(a) Frame 96



(b) Frame 354



(c) Frame 620



(d) Frame 860

Figura 5.4 – *Puget Sound*: Capturas de tela registradas ao longo da câmera.

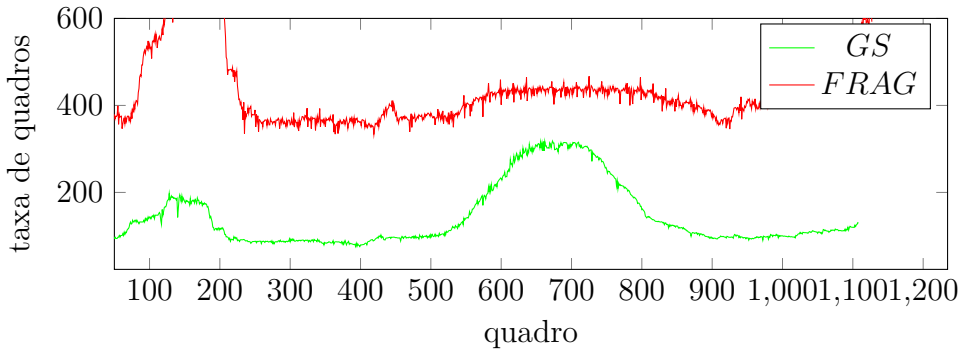


Figura 5.5 – Taxa de quadros dos algoritmos de construção de silhueta.

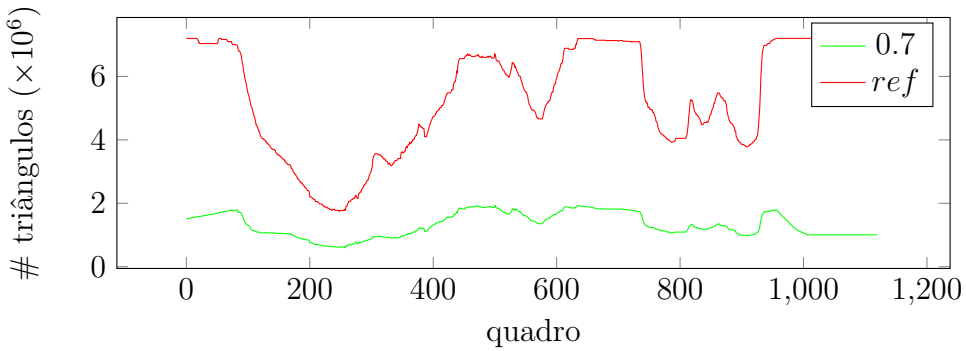
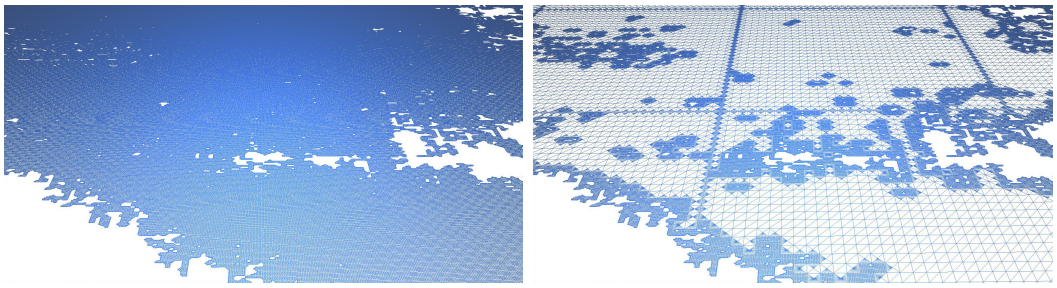
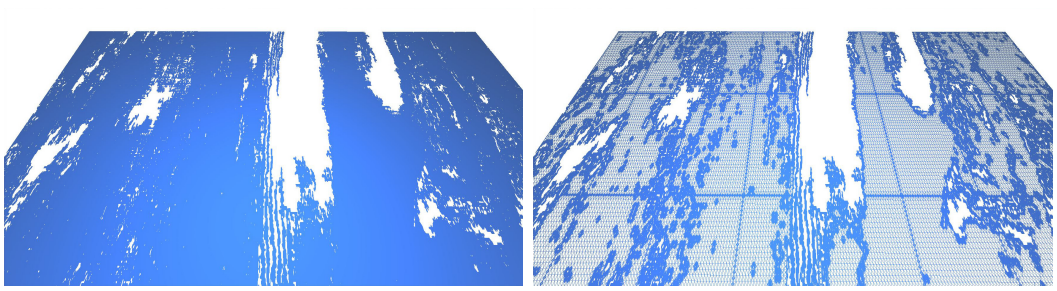


Figura 5.6 – Comparação entre os triângulos gerados pela malha completa(*ref*) e os gerados pelo algoritmo de LOD com uma tolerância de 0.7 pixels.



(a) Malha completa: 1.4M triângulos. Algoritmo de LOD: 0.3M triângulos



(b) Malha completa: 2.7M triângulos. Algoritmo de LOD: 0.9M triângulos

Figura 5.7 – Comparação entre a malha completa e o algoritmo de LOD. A esquerda as imagens mostram o *wireframe* da triangulação completa, e a direita uma triangulação com tolerância ao erro de 0.7 pixels. A diferença entre as imagens é nula.

6
Conclusão

Neste trabalho foi apresentada uma nova estratégia híbrida para desenhar grandes modelos digitais de elevação com imagens aéreas mapeadas como textura. O gerenciamento dos níveis de detalhe executam separadamente, para geometria (elevação) e textura (imagem aérea). Em CPU, uma implementação multi-thread é responsável por selecionar, carregar e transferir os dados dos ladrilhos para GPU. Os ladrilhos de geometria são subdivididos em *patches*, e os *tessellation shaders* são usados para determinar os níveis de triangulação para cada *patch*. O método proposto elimina possíveis *T-vértices* entre as interfaces dos *patches*. Os ladrilhos de geometria e textura são combinados sem necessidade de dados extras, mesmo nos casos onde o ladrilho de geometria é coberto por um conjunto de diferentes ladrilhos de textura.

Também foi proposta uma nova métrica de erro do objeto para dados de elevação contendo informações nulas. A métrica insere informação suficiente para o algoritmo de LOD triangular o terreno sem alterar visualmente a silhueta dos buracos. Foram propostas duas técnicas para descartar os triângulos dentro de áreas com ausência de informação, e gerar silhuetas melhor definidas.

Como trabalhos futuros está prevista uma extensão do algoritmo para lidar com terrenos que não possuam dimensões potência de 2 (2^n), o uso da GPU para acelerar o pré-processamento dos dados, e criar um cache dos dados em memória RAM para otimizar o carregamento para a GPU.

Referências Bibliográficas

[1] LOSASSO, F.; HOPPE, H.. **Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids**. In: ACM SIGGRAPH 2004 Papers, SIGGRAPH '04, p. 769–776, New York, NY, USA, 2004. ACM.

[2] CIGNONI, P.; GANOVELLI, F.; GOBBETTI, E.; MARTON, F.; PONCHIO, F. ; SCOPIGNO, R.. **Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM)**. In: PROCEEDINGS OF THE 14TH IEEE Visualization 2003 (VIS'03), VIS '03, p. 20–, Washington, DC, USA, 2003. IEEE Computer Society.

[3] MAGALHÃES, L.. **Multi-resolution of out-of-core terrain geometry**. PhD thesis, Pontfícia Universidade Católica do Rio de Janeiro - PUC-Rio, Rio de janeiro, Mar. 2006.

[4] LINDSTROM, P.; PASCUCCI, V.. **Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization**. IEEE Transactions on Visualization and Computer Graphics, 8(3):239–254, July 2002.

[5] CORREA, W. T.; KLOSOWSKI, J. T. ; SILVA, C. T.. **Visibility-Based Prefetching for Interactive Out-Of-Core Rendering**. In: PROCEEDINGS OF THE 2003 IEEE Symposium ON Parallel AND Large-Data Visualization AND Graphics, PVG '03, p. 2–, Washington, DC, USA, 2003. IEEE Computer Society.

[6] LINDSTROM, P.; PASCUCCI, V.. **Visualization of large terrains made easy**. In: VISUALIZATION, 2001. VIS'01. Proceedings, p. 363–574. IEEE, 2001.

[7] LINDSTROM, P.; KOLLER, D.; RIBARSKY, W.; HODGES, L. F.; FAUST, N. ; TURNER, G. A.. **Real-time, continuous level of detail rendering of height fields**. In: PROCEEDINGS OF THE 23RD ANNUAL CONFERENCE ON Computer GRAPHICS AND INTERACTIVE TECHNIQUES, p. 109–118. ACM, 1996.

[8] DUCHAINEAU, M.; WOLINSKY, M.; SIGETI, D.; MILLER, M.; ALDRICH, C. ; MINEEV-WEINSTEIN, M.. **ROAMing terrain: Real-time Op-**

timally Adapting Meshes. In: VISUALIZATION '97., Proceedings, p. 81–88, Oct. 1997.

[9] DICK, C.; KRÜGER, J. ; WESTERMANN, R.. **GPU Ray-Casting for Scalable Terrain Rendering**. In: PROCEEDINGS OF Eurographics 2009 - Areas Papers, p. 43–50, 2009.

[10] SCHÄFER, H.; SNER, M. N.; KEINERT, B.; STAMMINGER, M. ; LOOP, C.. **State of the Art Report on Real-time Rendering with Hardware Tessellation**. In: EUROGRAPHICS, p. 93–117, 2014.

[11] PAJAROLA, R.; GOBBETTI, E.. **Survey of semi-regular multiresolution models for interactive terrain rendering**. The Visual Computer, 23(8):583–605, 2007.

[12] CIGNONI, P.; GANOVELLI, F.; GOBBETTI, E.; MARTON, F.; PONCHIO, F. ; SCOPIGNO, R.. **BDAM — Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization**. Computer Graphics Forum, 22(3):505–514, Sept. 2003.

[13] FERNANDES, A.; OLIVEIRA, B.. **Gpu tessellation: We still have a lod of terrain to cover**. In: Cozzi, P.; Riccio, C., editors, OPENGL INSIGHTS, p. 143–160. CRC Press, July 2012. `http://www.openglinsights.com/`.

[14] CERVIN, A.. **Adaptive Hardware-accelerated Terrain Tessellation**. PhD thesis, Linköpings Universitet, Tekniska Högskolan, 2012.

[15] YUSOV, E.; SHEVTSOV, M.. **High-performance terrain rendering using hardware tessellation.** Journal of WSCG, 19(3):85–92, 2011.

[16] KANG, H.; JANG, H.; CHO, C.-S. ; HAN, J.. **Multi-resolution Terrain Rendering with GPU Tessellation**. Vis. Comput., 31(4):455–469, Apr. 2015.

[17] ULRICH, T.. **Rendering massive terrains using chunked level of detail**. ACM SIGGRAPH Course "Super-size it! Scaling up to Massive Virtual Worlds", 2000.

[18] USGS; OF WASHINGTON, T. U.. **Puget sound**, 2001.