Pontifícia Universidade Católica
do Rio de Janeiro

Fabrício Cardoso da Silva

Detalhamento de Superfícies Utilizando Tesselação em Hardware

DISSERTAÇÃO DE MESTRADO

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio.

Orientador: Alberto Barbosa Raposo

Rio de Janeiro
Junho de 2012

Pontifícia Universidade Católica
do Rio de Janeiro

Fabrício Cardoso da Silva

Detalhamento de Superfícies Utilizando Tesselação em Hardware

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio.

Prof. Alberto Barbosa Raposo
Orientador
Departamento de Informática – PUC-Rio

Prof. Marcelo Gattass
Departamento de Informática – PUC-Rio

Prof. Waldemar Celes Filho
Departamento de Informática – PUC-Rio

Prof. José Eugênio Leal
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 29 de junho de 2012

Fabrício Cardoso da Silva

Graduado em Engenharia da Computação na Universidade Federal do Pará em 2009, atualmente trabalha no TecGraf, instituto de pesquisa associado à PUC-Rio, tendo como área de concentração Computação Gráfica.

Ficha Catalográfica

Silva, Fabrício Cardoso da

Detalhamento de superfícies utilizando tesselação em hardware / Fabrício Cardoso da Silva ; orientador: Alberto Barbosa Raposo. – 2012.

62 f : il. (color.) ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2012.

Inclui bibliografia

1. Informática – Teses. 2. Detalhamento de superfícies. 3. Programação em GPU. 4. Visualização em tempo real. I. Raposo, Alberto Barbosa. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Aos meus jovens irmãos Abner e Esther.

Agradecimentos

Ao Tecgraf, por custear minha pesquisa na íntegra.

Ao professor Alberto Raposo, por ter me orientado desde o início desta caminhada em "terras estranhas," sempre muito paciente e motivador, estimulando-me continuamente a prosseguir apesar de minhas limitações, momentos de teimosia e sumiços.

À minha família, em especial minha mãe e meus irmãos Abner e Esther, pelo apoio incondicional, apesar de ocasionais dificuldades e desentendimentos.

À minha querida Daria, que, em sua distinta delicadeza, sempre proferiu palavras maravilhosas para me dar conforto e motivação para seguir em frente, apesar da distância que nos separa.

Aos meus grandes amigos Pedro Luchini e Renato Prado, pela incrível e incomparável amizade e hospitalidade desde os momentos iniciais desta jornada.

Aos meus amigos Markus, Samuel e, especialmente, Daniel, Pandu e Renano, os quais, mesmo tendo conhecido há apenas alguns meses, proporcionaram momentos preciosos que me permitiram manter grande tranquilidade durante a etapa final deste trabalho.

Aos excelentíssimos senhores Bruno Baère, Chrystiano Araújo, Eduardo Ceretta e Marcelo Arruda, com quem pude compartilhar muitos conhecimentos durante este programa de mestrado.

Aos antigos e atuais integrantes da equipe do Environ com quem tive o privilégio de trabalhar e aprender, pelos ótimos momentos proporcionados dentro e fora do ambiente de trabalho.

Resumo

Silva, Fabrício Cardoso; Raposo, Alberto Barbosa. **Detalhamento de Superfícies Utilizando Tesselação em Hardware.** Rio de Janeiro, 2012. 63p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Técnicas de mapeamento de rugosidade são amplamente utilizadas para simular detalhes estruturais de superfícies tridimensionais com o intuito de aumentar a qualidade visual e compensar o baixo detalhamento geométrico usualmente aplicado aos modelos enviados à GPU por questões de desempenho. Avanços recentes no *pipeline* de renderização permitiram a geração massiva de vértices no hardware gráfico através do recurso de tesselação, oferecendo aos desenvolvedores uma poderosa ferramenta para controle do nível de detalhes de objetos. Este trabalho apresenta uma técnica para o detalhamento geométrico de modelos utilizando tesselação em hardware, baseada tanto em mapas pré-computados quanto em dados de deslocamento gerados inteiramente na GPU por meio de técnicas de texturas procedimentais. Análises de desempenho e qualidade visual demonstram as vantagens do método proposto em relação a uma técnica de detalhamento baseada em imagens que é utilizada frequentemente em jogos eletrônicos para enriquecimento da qualidade visual de seus ambientes.

Palavras-chave

Detalhamento de superfícies; programação em GPU; visualização em tempo real

Abstract

Silva, Fabrício Cardoso; Raposo, Alberto Barbosa (advisor). **Surface Detailing Using Hardware Tessellation.** Rio de Janeiro, 2012. 63p. MSc. Dissertation– Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Bump mapping techniques are widely used to simulate structural details of tridimensional surfaces in order to improve visual quality and compensate for the low geometric detailing generally applied to models sent to the GPU due to performance issues. Recent advances in the rendering pipeline enabled the massive generation of vertex data in the graphics hardware by means of the tessellation feature, providing developers with a powerful tool to control the meshes' level of details. The present work proposes a technique for geometric detailing of models using hardware tessellation, both based on pre-computed depth maps and on displacement data generated entirely on the GPU through procedural textures techniques. Performance and visual quality analysis demonstrates the advantages of the proposed method in relation to an image-based technique commonly used in videogames for enhancing the visual quality of the environments.

Keywords

Surface detailing; GPU programming; real time visualization

Sumário

Lista de Figuras

Lista de Tabelas

Tudo quanto vive, vive porque muda; muda porque passa; e porque passa, morre. Tudo quanto vive perpetuamente se torna outra coisa, constantemente se nega, se furta à vida.
Fernando Pessoa

1
Introdução

O constante avanço das tecnologias de *hardware* gráfico propicia o desenvolvimento de um grande número de algoritmos para enriquecimento de ambientes virtuais, com o claro intuito de aproximá-los ao máximo da realidade. A visualização de ambientes complexos em tempo real requer um alto nível de sofisticação desses algoritmos de forma que seja mantida uma qualidade visual aceitável sem prejudicar a interação do usuário com a aplicação.

No entanto, um gargalo bastante comum em aplicações de computação gráfica é a geometria a ser renderizada, relacionada ao conjunto de vértices que constituem o modelo. Tal problema está diretamente relacionado às limitações impostas pela largura de banda da interface entre CPU e GPU (Owens, et al., 2008), que leva os desenvolvedores a reduzirem a quantidade de informações enviadas para a placa gráfica, sendo uma prática comum optar pela renderização de modelos com baixa quantidade de polígonos para favorecer o desempenho da aplicação.

Uma das formas mais comuns de contornar essa limitação é a utilização de algoritmos baseados em imagens para adição de detalhes sobre superfícies. Introduzido por Blinn (Blinn, 1978), o *bump mapping* permite o acréscimo de detalhes de rugosidade a um objeto através de uma implementação simples e de baixo custo computacional, mas, por ser apenas um truque de iluminação, não é capaz de representar efeitos mais complexos, como auto-sombreamento ou oclusões causados por interpenetrações (*self-shadowing* e *self-occlusion*), correção de perspectiva e detalhes da silhueta do objeto.

Displacement mapping, introduzido por Cook (Cook, 1984), possibilita a definição de detalhes geométricos através do deslocamento de pontos da superfície com base em informações de um mapa de altura. Dessa forma, garante-se a correta representação de detalhes das silhuetas e dos efeitos de *self-shadowing* e *occlusion*, ao custo de uma vigorosa subdivisão da malha a ser

renderizada em micropolígonos, o que torna esta técnica inadequada para aplicações em tempo real.

A partir disso, foi desenvolvido um número de técnicas de detalhamento envolvendo diferentes abordagens, com o intuito de aprimorar os efeitos do *bump mapping* e *displacement mapping* sem comprometer o desempenho. Algumas técnicas são baseadas em *ray tracing* (Pharr & Hanrahan, 1996) (Heidrich & Seidel, 1998) (Smits, Shirley, & Stark, 2000), distorção de imagens durante o processo de mapeamento para correção de perspectiva (Oliveira, Bishop, & McAllister, Relief texture mapping, 2000) e pré-computação de informações de visibilidade (Wang, et al., 2003) (Wang, et al., 2004). Os métodos baseados em *ray tracing*, devido ao alto custo computacional, são inadequados para aplicações em tempo real, enquanto os restantes, apesar de interativos, apresentam alto consumo de memória devido ao pré-processamento de um grande volume de informações.

Técnicas mais recentes que fazem uso de programação em GPU são capazes de reproduzir efeitos de sombreamento, oclusão e interpenetrações com maior desempenho e menores requerimentos de memória. Tais técnicas se baseiam principalmente em deslocamentos na imagem mapeada utilizando programas de *pixel* do *pipeline* gráfico (Kautz & Seidel, 2001) (Brawley & Tatarchuk, 2004) (Hirche, Ehlert, Guthe, & Doggett, 2004) (Policarpo, Oliveira, & Comba, 2005) (McGuire & McGuire, 2005) (Tatarchuk, 2006).

No entanto, apesar de reproduzirem detalhes de forma bastante convincente, tais técnicas não realizam modificações na geometria do objeto, não sendo adequadas a casos genéricos em que objetos podem ser vistos de diferentes ângulos e distâncias. Tal limitação resulta na representação incorreta de silhuetas, podendo afetar a qualidade da imagem final. Algumas abordagens propõem a geração de silhuetas através de deslocamentos de acordo com o ponto de vista do observador (Wang, et al., 2003) (Wang, et al., 2004) ou adequação de superfícies quádricas sobre o objeto para descrever, em nível de *pixel*, a curvatura da superfície (Oliveira & Policarpo, 2005), mas sofrem com problemas de *aliasing*.

Com as recentes atualizações na arquitetura dos *hardwares* gráficos, marcadas com os lançamentos das interfaces de programação (API) DirectX 11[1]

[1]SDK (junho de 2010) disponível para download em:
http://www.microsoft.com/download/en/details.aspx?id=6812.

(Valdetaro, Nunes, Raposo, & Feijó, 2010) e OpenGL 4.0[2], foram introduzidos dois novos estágios programáveis no *pipeline*, correspondentes aos *tessellation shaders*, que possibilitam a criação de grande volume de vértices na GPU. Como esse dispositivo é dotado de grande capacidade de processamento em paralelo, a geração massiva de primitivas pelo *hardware* gráfico pode ser utilizada para suprir a deficiência geométrica do modelo que é enviado à GPU, além de oferecer melhores meios para controlar o seu nível de detalhes, amenizando o gargalo provocado pela transferência de muitas informações para a placa gráfica. Um exemplo de aplicação de tesselação pode ser visto no jogo *Tom Clancy's H.A.W.X. 2*, que foi um dos primeiros jogos a utilizar este recurso por meio da API DirectX 11, apresentando terrenos bastante realistas com controle dinâmico do nível de detalhes. Uma comparação entre os detalhes de terrenos do jogo com tesselação desativada e ativada pode ser vista na Figura 1.

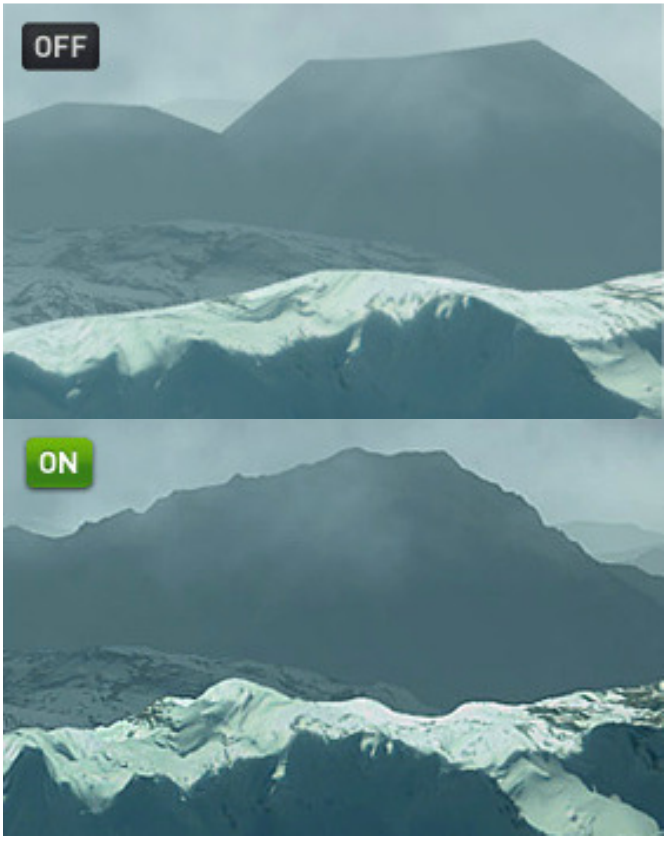


Figura 1: Tesselação de terrenos no jogo Tom Clancy's H.A.W.X. 2 (HAWX 2, 2010).

Com isso, surge a possibilidade de exploração deste *pipeline* para geração de detalhes sobre superfícies através da utilização de mapas de deslocamento na

[2] Informações sobre a versão mais recente disponíveis em http://www.opengl.org/.

GPU para realizar mudanças geométricas nas malhas subdivididas durante o estágio de tesselação. Tal procedimento não sofreria com as limitações dos métodos baseados em imagens, permitindo a visualização correta dos detalhes do objeto independente de ângulo ou distância de visualização.

A proposta desta dissertação, portanto, consiste em realizar um aparato sobre algumas técnicas de detalhamento baseadas em imagens comumente utilizadas em aplicações de tempo real e apresentar uma implementação que toma vantagem dos recursos recém-introduzidos no *pipeline* de renderização para adicionar detalhes geométricos autênticos a superfícies. Resultados experimentais demonstram vantagens em termos de desempenho e qualidade visual em relação a uma técnica de detalhamento baseada em imagens que é utilizada frequentemente em jogos eletrônicos para enriquecimento do ambiente virtual, a qual também foi implementada para que pudessem ser verificados os resultados deste trabalho. A análise dos resultados tem como principal objetivo identificar vantagens e desvantagens de cada abordagem, bem como avaliar propostas de trabalhos futuros que porventura aprimorem os resultados apresentados.

O Capítulo 2 apresenta rapidamente um conjunto de técnicas baseadas em imagens cujos objetivos são similares aos deste trabalho. O Capítulo 3 descreve os fundamentos do novo *pipeline*, tendo como foco o recurso de tesselação, e uma breve introdução à geração de texturas procedimentais. O Capítulo 4 revisita de forma mais detalhada algoritmos de algumas técnicas de detalhamento baseadas em imagens e expõe os critérios que levaram à escolha da técnica de *parallax occlusion mapping* para avaliar comparativamente os resultados deste trabalho. O Capítulo 5 expõe em detalhes a implementação proposta neste trabalho, apresentando as características de cada *shader* utilizado para tesselação e deslocamento de vértices. O Capítulo 6 apresenta uma análise dos resultados obtidos, realizando uma comparação do desempenho e da qualidade visual oferecida por cada implementação. O Capítulo 7, por fim, apresenta as conclusões e propostas de trabalhos futuros relacionados com o método descrito.

2
Trabalhos relacionados

Com o objetivo de aperfeiçoar os efeitos do *bump mapping* e *displacement mapping*, foram desenvolvidos diversos métodos que buscam explorar a programação em GPU em favor do acréscimo de detalhes estruturais a superfícies de objetos em aplicações interativas.

Kaneko, et al. (Kaneko, et al., 2001) introduz uma técnica simples para simular o efeito de paralaxe (*motion parallax*) – a aparente movimentação de partes do objeto quando da mudança do ponto de vista do observador – através do deslocamento, para cada *pixel*, das coordenadas de textura do objeto. Esse deslocamento é feito traçando-se um raio sobre a superfície para definir qual *texel* do mapa de altura corresponde ao ponto de interseção, que é, por fim, utilizado para definir as coordenadas corretas do *pixel* na imagem final. Apesar de ser simples e atingir bom desempenho, este método apresenta grandes distorções sobre a superfície quanto menor for o ângulo de visualização com a superfície.

Tais limitações foram amenizadas por Welsh (Welsh, 2004), que propôs a ideia de limitação do deslocamento das coordenadas de textura de forma que elas nunca sejam superiores ao valor de altura obtido do mapa. Essa atualização reduz a percepção das protuberâncias da superfície, mas também as flutuações de textura em ângulos rasantes.

Wang et al. (Wang, et al., 2003) propõem um algoritmo capaz de renderizar irregularidades no interior de superfícies junto aos efeitos de sombreamento, oclusão e silhuetas, sem modificações na geometria, por meio do armazenamento da distância dos pontos deslocados vistos de múltiplas direções. Esse grande volume de informações pré-computadas é codificado em uma função de cinco dimensões e requer decomposição e compressão dos dados a serem enviados à GPU. Essa técnica apresenta resultados bons, porém limitados. Além de suportar apenas superfícies fechadas, há a ocorrência de *aliasing* e imprecisões sobre as faces do objeto, que podem ser agravados quanto mais próximo estiver o observador da superfície devido ao número fixo de resoluções pré-computadas.

Enquanto o *aliasing* pode ser contornado por meio de *mip-maping*, a obtenção de resultados mais precisos só pode ser atingida através do aumento do número de amostras realizadas em cada direção, tendo implicações diretas sobre o consumo de memória.

Como uma extensão da técnica anterior, Wang et al. (Wang, et al., 2004) apresentaram uma abordagem capaz de renderizar com maior precisão superfícies fechadas ou abertas, sem que necessariamente tenham seus detalhes estruturais descritos por mapas de altura. A representação através de uma versão atualizada da função de cinco dimensões, no entanto, ainda mantém limitações em termos do número de amostras a serem computadas em favor de um menor consumo de memória, tornando esta técnica pouco adequada para visualizações muito próximas.

Brawley e Tatarchuk (Brawley & Tatarchuk, 2004) realizam uma abordagem semelhante à de Kaneko (Kaneko, et al., 2001), adicionando também suporte à geração de auto-sombreamento e oclusão. No entanto, ao invés de traçar o raio diretamente sobre a superfície, este é lançado sobre o perfil de sua curvatura, que é armazenado em um mapa de profundidade, e, a partir do ponto de interseção, é calculado o deslocamento para as coordenadas de textura. Tal técnica apresenta um conjunto de limitações que prejudicam demasiadamente a qualidade da imagem final, em especial quando da visualização de ângulos rasantes, resultando em uma perda severa das características da superfície devido ao método impreciso de amostragem do mapa de altura ao longo do raio.

Policarpo et al. (Policarpo, Oliveira, & Comba, 2005) seguem um princípio muito semelhante para realizar o deslocamento das coordenadas de textura, mas apresentam uma forma mais precisa de traçar o raio sobre o perfil do mapa de profundidade. O processo é iniciado com uma busca linear, com o objetivo de encontrar o primeiro ponto abaixo do perfil de curvatura descrita pelo mapa. Em seguida, uma busca binária é efetuada para determinar precisamente o ponto de interseção. A realização de filtragem da textura contendo o mapa de profundidade torna a curvatura da superfície contínua, amenizando o *aliasing* e permitindo a visualização de pontos mais próximos à superfície sem a ocorrência de distorções significativas. Apesar disso, além da ausência de silhuetas, também é perceptível um achatamento dos detalhes da curvatura quando da visualização de ângulos rasantes.

Posteriormente, essa técnica foi estendida por Oliveira e Policarpo (Oliveira & Policarpo, 2005) para dar suporte à geração de silhuetas. Através de uma deformação local da superfície descrita pelo mapa de profundidade, é possível determinar e descartar os raios que não interceptam o perfil da curvatura. Para isto, uma superfície quádrica é acomodada aos vértices do modelo poligonal em um estágio de pré-processamento para deformar a *bounding box* do modelo, tornando o mapa de altura "adequado" à curvatura da superfície, sendo o ponto de interseção determinado quando o raio interceptar o perfil desse mapa no interior da *bounding box* deformada. Este procedimento resulta na geração de silhuetas convincentes, mas não só possui problemas de *aliasing*, como não é adequado a qualquer tipo de superfície sem a aplicação de um fator de correção de profundidade.

Tatarchuk (Tatarchuk, 2006) propôs melhorias ao algoritmo original do *parallax occlusion mapping* (Brawley & Tatarchuk, 2004) melhorando a qualidade das sombras geradas pelas interpenetrações e aumentando a precisão da captura de feições da superfície pelo mapa de altura por meio do aperfeiçoamento do método de interseção do raio com a superfície. A imagem final apresenta efeitos de sombreamento bastante convincentes e correção de perspectiva melhorada, mas além de não dar suporte a silhuetas, também sofre com problemas de *aliasing* quanto mais próximo estiver o observador da superfície e apresenta distorções em ângulos de visualização pequenos.

Neste trabalho, o algoritmo de *parallax occlusion mapping* (Tatarchuk, 2006) foi implementado com o objetivo de ser usado como base para comparação dos resultados obtidos com o recurso de tesselação. Esta técnica será examinada com mais detalhes no Capítulo 4, juntamente com a técnica de *parallax mapping* (Kaneko, et al., 2001) (Welsh, 2004), da qual o *parallax occlusion mapping* deriva parte de seus princípios e os aprimora para produzir resultados com maior qualidade visual.

3
Fundamentos

Este capítulo apresenta uma visão geral do *pipeline* de renderização da versão 4.0 da API OpenGL, tendo como foco os recém-introduzidos estágios de tesselação. Em seguida, é realizada uma abordagem introdutória sobre texturas procedimentais com o objetivo de mostrar alguns aspectos sobre síntese de texturas como alternativa aos tradicionais métodos de obtenção de valores por meio de imagens.

3.1
Pipeline de renderização em OpenGL

Com a introdução de dois novos estágios programáveis chamados *tessellation control shader* e *tessellation evaluation shader* e um estágio fixo chamado *primitive generator*, o *pipeline* de renderização da OpenGL para a atual geração de hardwares gráficos pode ser caracterizado conforme mostra a Figura 2. O processo de tesselação é realizado sobre um novo tipo de primitiva criada especificamente para este fim, intitulada *patch*, a qual consiste em um conjunto de vértices ordenados, mas sem uma topologia definida, já que esta é definida apenas ao final do processo de subdivisão. Cada vértice de um *patch* possui seus atributos próprios, bem como atributos globais associados ao *patch* a que pertencem.

Cada *patch* possui um número fixo de vértices, que deve ser definido pelo programador antes das chamadas aos métodos de renderização que fazem uso de *shaders* de tesselação. Este número não pode exceder a constante especificada pela OpenGL para a máxima quantidade de vértices contidos em um *patch*.

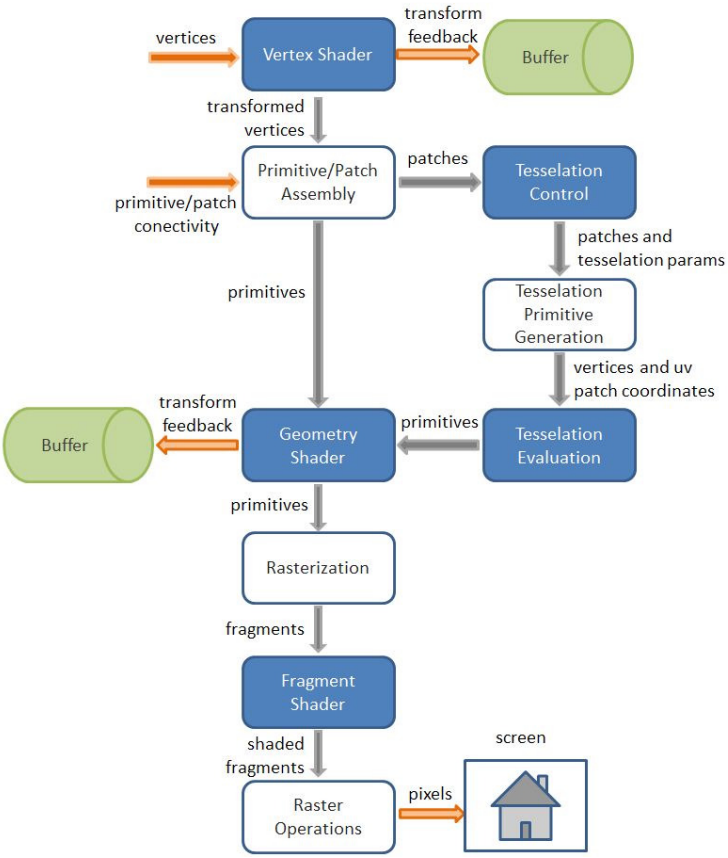


Figura 2: Pipeline de renderização da OpenGL (GL Pipeline, 2011).

3.1.1. Vertex Shader

O *vertex shader* recebe da aplicação o conjunto de vértices que constituem o modelo original e é responsável por transformá-los conforme necessário. Caso exista um *shader* de tesselação ativo no *pipeline*, a operação de transformação da posição do vértice atual para o espaço de projeção por meio do produto com as matrizes *model*, *view* e *projection* se torna desnecessária nesta etapa. Assim, a transformação para o espaço de projeção deve ser computada no *tessellation evaluation shader*, quando todos os vértices da malha tiverem sido gerados.

A forma mais interessante de utilização deste estágio em conjunto com os de tesselação é no caso de aplicações que possuem modelos com animações. Nesse cenário, transformações que realizam deformações na malha (como *skinning* e *morphing*) são realizadas diretamente sobre os pontos de controle do *patch* da malha de baixa resolução no *vertex shader* antes de serem enviados aos estágios de subdivisão (Ni & Castano, 2009). Com isso, as operações de animação

são realizadas em uma frequência reduzida e os vértices gerados durante o estágio de tesselação serão interpolados de acordo com as posições dos vértices do *patch* já transformado e estarão em suas posições desejadas para um dado quadro da animação, conforme mostra a Figura 3.

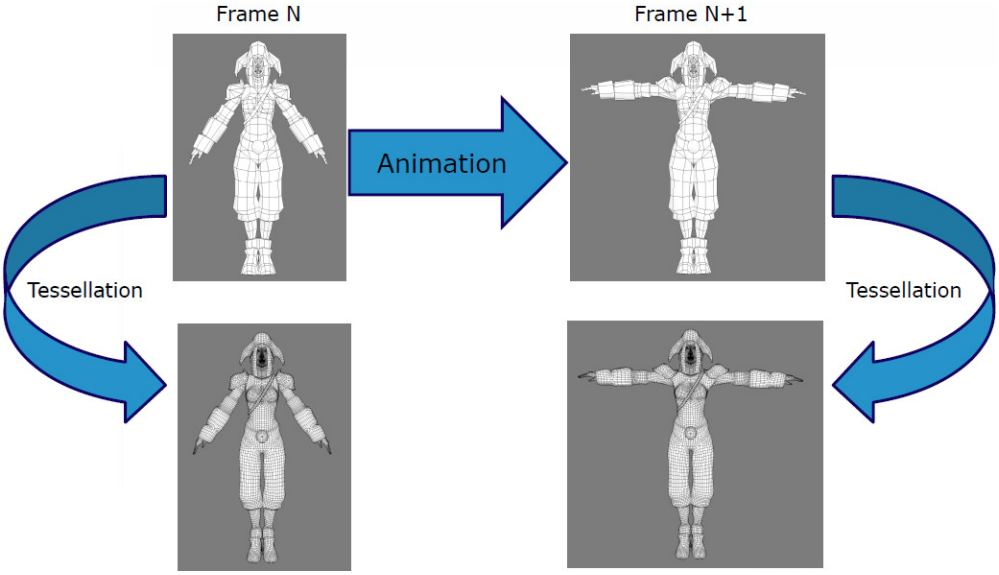


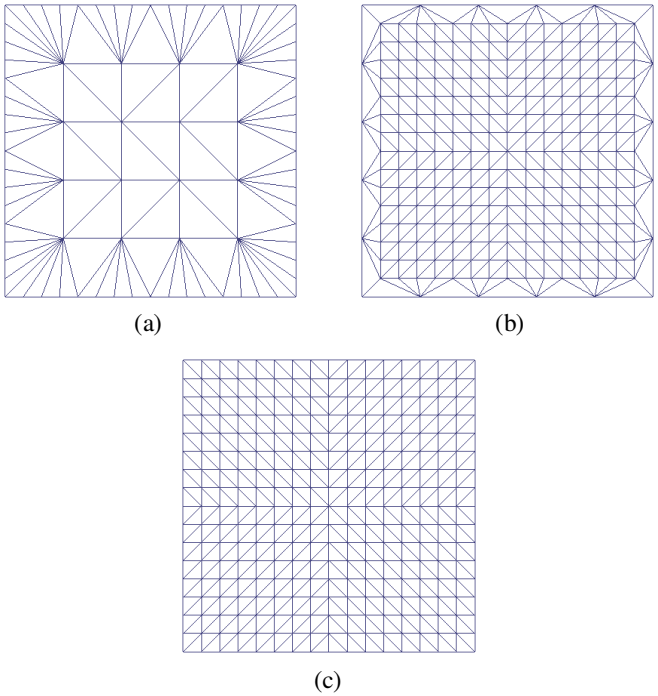
Figura 3: Morphing no vertex shader (Tatarinov, 2008).

3.1.2. Tessellation Control Shader

O *tessellation control shader* recebe um *patch* de entrada e é executado para cada vértice pertencente a ele. Este estágio é responsável pela computação dos atributos do vértice atual e atributos adicionais do *patch* de saída. Entre estes atributos estão os níveis de tesselação interno e externo e o número de vértices de saída.

O nível de tesselação interno é composto por dois valores de ponto flutuante e utilizado pelo *primitive generator* para aproximar o número de segmentos nas arestas internas da primitiva, com o objetivo de oferecer um aspecto homogêneo para a subdivisão. O nível de tesselação externo é composto por até quatro valores de ponto flutuante que definem o número de segmentos de cada aresta externa da primitiva a ser gerada. O número de vértices produzidos pelo *tessellation control shader* deve ser especificado por meio de uma declaração de *layout* para o *patch* de saída. Este valor corresponde ao número de vezes que este *shader* será executado e não pode exceder o valor máximo definido na aplicação para o tamanho do *patch*. O efeito da variação dos níveis de tesselação sobre a primitiva

subdividida é ilustrado na Figura 4. Um nível de tesselação externo maior que o interno resulta na primitiva observada na Figura 4(a), enquanto um nível interno maior resulta na observada na Figura 4(b); por fim, níveis de tesselação interno e externo iguais resultam na primitiva exibida na Figura 4(c).



(a) (b)

(c)

Figura 4: Efeito da variação dos níveis de tesselação interno e externo.

3.1.3. Primitive Generator

O *tessellation primitive generator* é executado se houver um *tessellation evaluation shader* ativo. Este estágio recebe o número de vértices definido no estágio anterior e realiza a subdivisão do *patch* em um conjunto de pontos, linhas ou triângulos, de acordo com os níveis especificados no *tessellation control shader* e com a declaração do *layout* de entrada do *tessellation evaluation shader*, que será discutida adiante.

A cada vértice da primitiva resultante são associadas coordenadas do tipo (*u, v*) ou (*u, v, w*) definidas em um espaço paramétrico. Para triângulos, são as coordenadas baricêntricas (*u, v, w*) indicando a posição do vértice gerado em relação às coordenadas do triângulo original do *patch* de entrada. Vértices de *isolines* ou *quads*, por outro lado, recebem coordenadas do tipo (*u, v*) que indicam as posições horizontal e vertical, relativas à primitiva do *patch* original.

3.1.4.
Tessellation Evaluation Shader

O *tessellation evaluation shader* recebe os vértices da primitiva subdividida pelo *primitive generator* e gera cada vértice da primitiva de saída com suas respectivas posições e atributos associados. Este *shader* é executado de forma independente para cada vértice da primitiva de saída, mas permite ao programador o acesso a todos os vértices do *patch* de entrada. Além disso, o programador pode acessar diretamente a posição do vértice atual em relação à primitiva subdividida, isto é, as coordenadas (*u, v*) ou (*u, v, w*) mencionadas anteriormente.

Nesta etapa, cabe ao programador definir o tipo de subdivisão realizada pelo *primitive generator* através de uma declaração de *layout* do *patch* de entrada para o *tessellation evaluation shader*, a qual também estabelece o método usado para derivar o número e o espaçamento entre os segmentos de acordo com o nível de tesselação estabelecido no *tessellation control shader*, além da orientação dos triângulos gerados nos modos *triangles* e *quads*, que pode ser em sentido horário ou anti-horário. Resultados de subdivisões simples utilizando diferentes métodos de segmentação e espaçamento podem ser vistos na Figura 5.

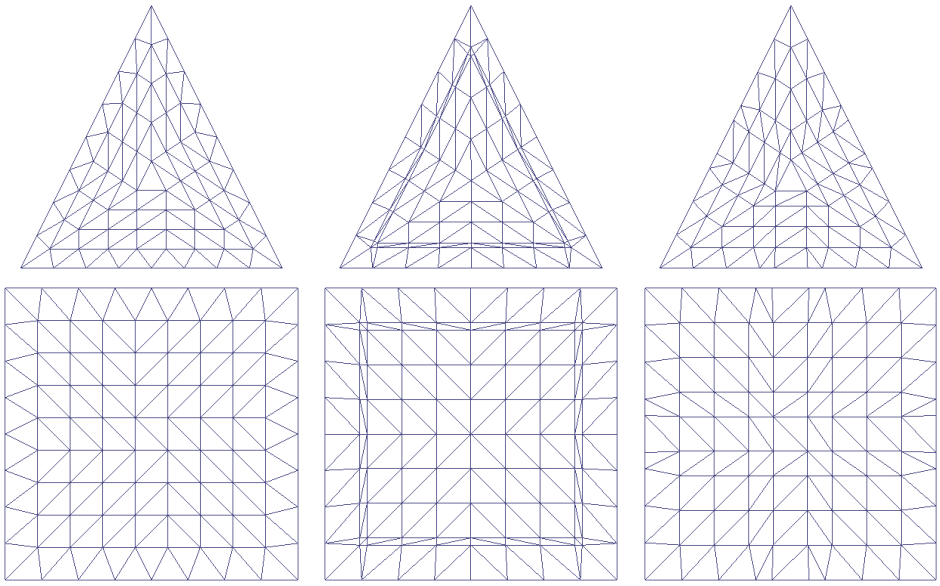


Figura 5: Diferentes modos de espaçamento na subdivisão de triângulos e quads.

3.1.5. Geometry Shader

O *geometry shader*, introduzido com o lançamento das interfaces de programação DirectX 10 e OpenGL 3.2, ofereceu a programadores os primeiros meios de criação de geometria em GPU, apesar de bastante limitados principalmente em termos do tamanho da saída a cada instância desse *shader*. Este estágio fica localizado após o *primitive assembly* e é executado para cada primitiva (cujos vértices são recebidos diretamente do *vertex shader*, caso não existam *shaders* de tesselação ativos), emitindo uma ou mais primitivas de saída do mesmo tipo, descartando, em seguida, a primitiva recebida originalmente.

Neste estágio, são disponibilizadas informações de adjacência da primitiva de entrada, tornando o *geometry shader* adequado para algoritmos que requerem coleta de informações da superfície sendo renderizada. Alguns exemplos de aplicações que utilizam *geometry shaders* são volumes de sombra (Stich, Wächter, & Keller, 2007), simplificação e controle do nível de detalhe de acordo com o ponto de vista do observador (Hu, Sander, & Hoppe, 2010) (DeCoro & Tatarchuk, 2007) e extração de iso-superfícies em GPU (Tatarchuk, Shopf, & DeCoro, 2007).

3.1.6. Fragment Shader

O *fragment shader* é responsável pelas operações aplicadas sobre os fragmentos resultantes da rasterização das primitivas e seus dados associados, sendo utilizado fundamentalmente para atualizar o *framebuffer* ou a memória de textura. Neste estágio, é possível ler variáveis de entrada correspondentes aos fragmentos gerados, as quais podem ser internas (como a posição do fragmento), ou definidas pelo usuário nos estágios que antecedem o *fragment shader* (as quais são interpoladas de acordo com a primitiva rasterizada), mas não é possível acessar dados de fragmentos vizinhos.

Neste estágio, também é possível atribuir variáveis de saída, que podem ser utilizadas para operações por fragmento subsequentes, como a escrita de dados em um *framebuffer object* ou o teste de profundidade com o *depth buffer* para determinar se determinado fragmento deve seguir pelo *pipeline* ou ser descartado.

O conjunto de operações e testes realizados por fragmento na etapa anterior à escrita no *framebuffer* não mostrados, em ordem, na Figura 6.

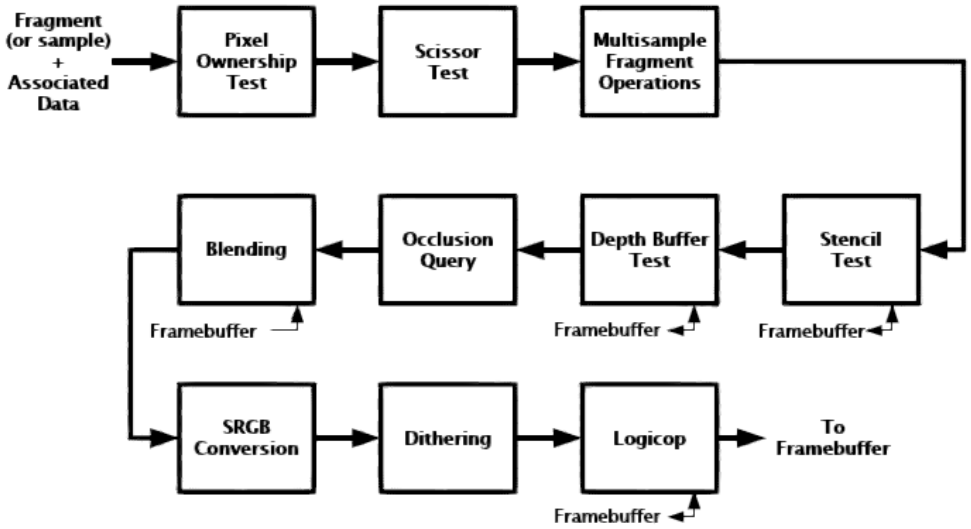


Figura 6: Ordem de operações por fragmento (OpenGL 4.2, 2012).

3.2
Introdução a texturas procedimentais

A geração texturas procedimentais é um campo de grande interesse em computação gráfica em virtude da possibilidade da criação de imagens que representem a natureza aleatória de diferentes tipos de materiais através de funções que codificam efeitos visuais de acordo com um determinado algoritmo, constituindo uma alternativa ao tradicional acesso a texturas codificadas em imagens para obtenção de informações. Não restrita apenas a casos bidimensionais, a síntese procedimental de texturas também é interessante no caso de texturas volumétricas (que possuem um alto custo de armazenamento no caso pré-computado) ou mesmo de geometria (Ebert, et al., 2002), sendo possível gerar uma variedade de formas complexas encontradas na natureza, como a vegetação ilustrada na Figura 7, renderizada através de *ray tracing*. A maioria dos efeitos gerados procedimentalmente são baseados no algoritmo chamado *Perlin Noise*, proposto por Ken Perlin (Perlin, 1985), que posteriormente também apresentou uma versão aperfeiçoada desse algoritmo em termos de desempenho e qualidade do ruído gerado (Perlin, 2002) (Perlin, 2004).

Figura 7: Vegetação gerada procedimentalmente (Ebert, et al., 2002).

A síntese de texturas na GPU para aplicações em tempo real possui um conjunto de vantagens e desvantagens que devem ser levadas em consideração ao se avaliar o efeito que se deseja renderizar. É mais comum a utilização balanceada entre texturas armazenadas em imagens e texturas geradas procedimentalmente, em que o primeiro caso cobre a maioria dos objetos existentes no mundo real, como fachadas de prédios, pinturas ou pavimentos, enquanto o segundo cobre elementos que possuem aspectos de cunho mais aleatório, como nuvens, tecidos ou terrenos.

Uma das razões para a efetividade de texturas procedimentais é a incorporação de elementos aleatórios de uma forma controlada. O algoritmo fundamental para geração de ruído consiste em aproximar o que seria obtido por meio do ruído branco e realizar uma filtragem das frequências além de um determinado ponto de corte. Essa aproximação pode ser avaliada para pontos arbitrários sem a necessidade de pré-computação de dados relacionados ao volume que contém esses pontos.

Comparadas a texturas armazenadas em imagens, texturas procedimentais tem um baixo custo de memória, já que são representadas apenas por algoritmos que avaliam determinadas funções, proporcionando uma notável vantagem em comparação a texturas 2D, a qual é ainda maior para texturas 3D. Por serem

definidas de forma algorítmica ao invés de amostragem de valores, essas texturas também não apresentam limitações de área ou resolução e podem ser aplicadas sobre objetos de qualquer escala sem que sejam visualizadas replicações, redução de detalhes ou artefatos de amostragem. Além disso, algoritmos para geração de texturas procedimentais são escritos com base em parâmetros que podem ser modificados com facilidade, possibilitando a criação de uma variedade de efeitos a partir do mesmo código.

Por outro lado, dependendo da complexidade do algoritmo ou da superfície, a avaliação da função para cada ponto de uma determinada superfície pode se tornar custosa, resultando em um desempenho menor que o do acesso a uma textura pré-computada. Em geral, texturas procedimentais também requerem algum tipo de filtragem para evitar artefatos causados por *aliasing*, os quais podem ser demasiadamente acentuados, dependendo do efeito implementado. Uma abordagem apresentada por Cook e DeRose (Cook & DeRose, 2005) propõe um método baseado na teoria de *wavelets* (Chui, 1992) para reduzir o efeito de *aliasing* com um pequeno aumento no custo de avaliação da função.

Uma aplicação em particular para texturas procedimentais é a combinação das texturas sintetizadas com a técnica de *bump mapping* para a simulação de detalhamento tanto regular quanto aleatório sobre superfícies. É importante ressaltar que, como as outras técnicas baseadas em imagens, por se tratar apenas de um efeito de iluminação, o *bump mapping* procedimental não realiza modificações sobre a geometria do objeto, sendo, portanto, incapaz de representar corretamente as suas silhuetas. Tal limitação torna este mapeamento adequado apenas para casos em que as irregularidades não são acentuadas em relação à superfície propriamente dita.

4
Detalhamento de superfícies baseado em imagens

Este capítulo realiza um breve estudo sobre algumas técnicas propostas para detalhamento de superfícies baseado em imagens. Serão explicados os fundamentos de cada abordagem e apontados os critérios que levaram à seleção da técnica de *parallax occlusion mapping* como base para comparação dos resultados deste trabalho.

4.1
Parallax mapping

A técnica de *parallax mapping* (Kaneko, et al., 2001) foi proposta com o objetivo de acrescentar à simulação de rugosidade o *motion parallax*, que consiste no aparente deslocamento de partes de um objeto de acordo com mudanças no ponto de vista do observador (sendo normalmente observado através do bloqueio entre protuberâncias da superfície), por meio de uma aproximação simples do efeito de um mapa de altura. Isto não pode ser alcançado através do algoritmo original de *bump mapping* por não ser levada em consideração a informação de profundidade do modelo.

4.1.1.
Algoritmo original

A superfície mostrada na Figura 8 é representada por um polígono sobre o qual foi apenas mapeada uma textura. Quando observada da direção especificada pelo vetor ***eye***, é visualizado o ponto A, que corresponde a uma representação incorreta da curvatura real da superfície, a qual, por consequência, parece ter sofrido um achatamento. Nesta situação, o ponto B corresponde à correta representação dessa curvatura, mas requer algum tipo de correção para que possa ser visualizado adequadamente.

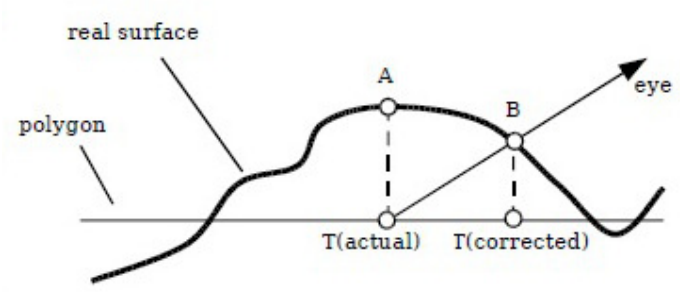


Figura 8: Visualização incorreta da curvatura da superfície (Welsh, 2004).

O algoritmo proposto para tal correção realiza o efeito de paralaxe através da distorção dinâmica da textura mapeada, de forma que ela corresponda à curvatura do objeto representado. Essa distorção não é realizada diretamente sobre a textura, mas através do deslocamento das coordenadas de textura de acordo com um mapa de altura, como mostra a Figura 9. A quantidade de deslocamento depende do ângulo de visão com a superfície e do valor de altura obtido, descrito apenas como um escalar no intervalo [0.0, 1.0].

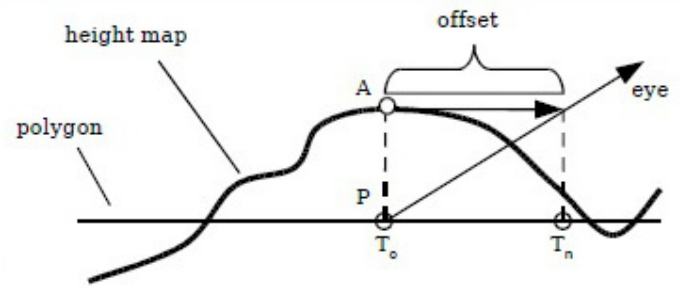


Figura 9: Deslocamento de coordenadas de textura (Welsh, 2004).

O deslocamento das coordenadas de textura para o ponto *P* é calculado com base no vetor ***eye*** normalizado, no valor de altura *h* obtido do mapa nas coordenadas de textura originais T_o. Traça-se um vetor paralelo ao polígono partindo do ponto *A*, pertencente ao perfil de curvatura da superfície e localizado diretamente sobre o ponto *P*, até ***eye***, representado pelas componentes horizontais, eye_x e eye_y e pela altura em relação à superfície, descrita pela componente no eixo z, eye_z. Conforme mostra a Equação 4-1, esse vetor pode ser somado diretamente às coordenadas originais T_o para produzir as coordenadas deslocadas T_n.

$$T_n = T_o + \frac{h \cdot eye_{xy}}{eye_z} \tag{4-1}$$

4.1.2. Limitação de deslocamento

A aproximação simples apresentada na Seção 4.1.1 resulta em efeitos de profundidade razoáveis quando as irregularidades da superfície não são acentuadas, mas introduz deformações consideráveis à medida que o observador se aproxima da superfície. Isto ocorre porque, em ângulos de visualização pequenos, o valor definido para o deslocamento tende a infinito, resultando em um mapeamento que em nada reflete o aspecto da superfície desejada.

Para contornar essa situação, Welsh (Welsh, 2004) propôs a ideia de limitar o deslocamento[3] em torno do ponto em que ocorre a interseção do vetor de visualização com o polígono da superfície. A solução consiste apenas em limitar o valor máximo do deslocamento de forma que ele nunca seja superior à altura obtida pelo mapa para esse ponto, como mostra a Figura 10.

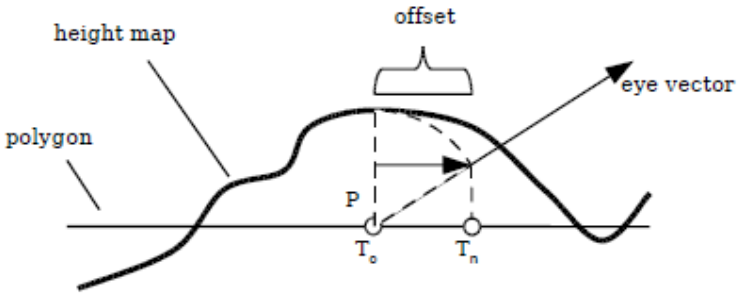


Figura 10: Parallax mapping com limitação de deslocamento (Welsh, 2004).

Observa-se também pela Figura 10 que esta abordagem simplificada pode resultar em uma maior percepção do efeito de achatamento da superfície, já que as novas coordenadas de textura podem não corresponder exatamente ao ponto em que o vetor de visualização intercepta a curvatura da superfície, mas isto evita o descontrole da quantidade de deslocamento quanto menor for o ângulo de visualização com o polígono e melhora a qualidade visual oferecida pela técnica. O novo procedimento para o cálculo para as novas coordenadas de textura é mostrado na Equação 4-2.

$$T_n = T_o + h \cdot eye_{xy} \tag{4-2}$$

[3] *Offset limiting* na literatura original.

4.2
Parallax occlusion mapping

O *parallax occlusion mapping* (POM) (Brawley & Tatarchuk, 2004) (Tatarchuk, 2006) surgiu junto a um número de algoritmos para detalhamento de superfícies desenvolvidos no mesmo período, mas por grupos de pesquisa diferentes, os quais também nomearam suas abordagens com nomes distintos: *relief mapping* (Policarpo, Oliveira, & Comba, 2005) e *steep parallax mapping* (McGuire & McGuire, 2005). Na prática, esses algoritmos realizam abordagens muito similares para representar irregularidades em superfícies, mantendo o efeito de *parallax* e adicionando a possibilidade de simular *self-shadowing* e *self-occlusion* em um modelo visto de diferentes ângulos. O princípio dessas técnicas será descrito na Seção 4.2.1 e o algoritmo para amostragem apresentado Tatarchuk (Tatarchuk, 2006) será examinado na Seção 4.2.2.

4.2.1.
Visão geral

A ideia dos algoritmos apresentados aqui é baseada na amostragem de uma textura ao longo de um raio projetado sobre a superfície, a qual é descrita por um mapa de profundidade (em que o polígono pertencente à superfície define o valor base na escala [0.0, 1.0]), tendo como objetivo a determinação do ponto de interseção entre raio e perfil de curvatura que esteja mais próximo ao observador, o qual corresponde ao mais próximo ponto visível da superfície, como mostra a Figura 11.

O cálculo do ponto é realizado por meio de uma busca linear pela primeira amostra localizada abaixo do mapa de profundidade. Essa amostra é usada junto à amostra anterior, localizada acima do perfil desse mapa, para resolver um problema de determinação de raiz que define o ponto de interseção desejado.

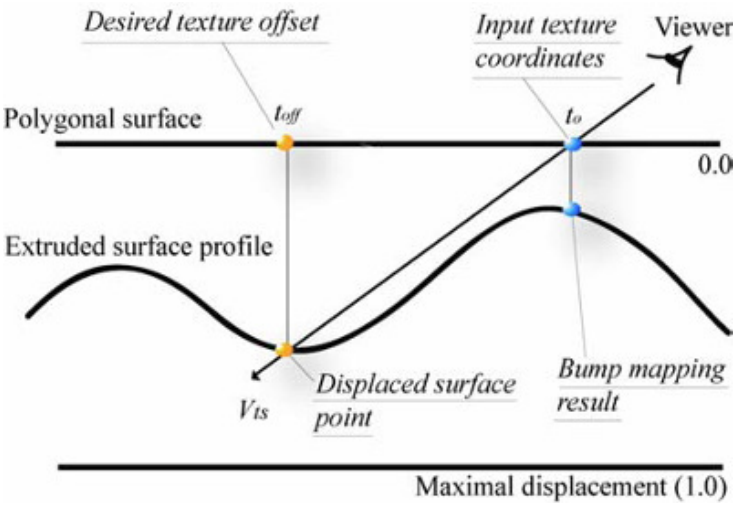


Figura 11: Determinação do ponto de interseção mais próximo (Tatarchuk, 2006).

Para a determinação da raiz, Tatarchuk (Tatarchuk, 2006) utiliza um passo do método secante, permitindo o uso mais eficiente do hardware gráfico, dado que as amostragens da textura podem ser feitas em paralelo. Policarpo (Oliveira & Policarpo, 2005), por outro lado, realiza uma busca binária entre os dois pontos, a qual resulta em menos acessos à textura, mas sofre com a latência causada pela dependência entre cada acesso. Szirmay-Kalos e Umenhoffer (Szirmay-Kalos & Umenhoffer, 2008) realizam um bom estudo sobre esses e outros métodos de busca por interseções.

4.2.2. Amostragem do mapa de profundidade

O procedimento para determinação do ponto de interseção entre raio e mapa de profundidade é iniciado com o cálculo do vetor de deslocamento de *parallax* (*parallax offset vector*) ***P***, o qual determina a direção em que deve ser feita a amostragem e a maior quantidade de deslocamento a ser realizado durante o cálculo da correção das coordenadas de textura. O cálculo do vetor é realizado com base nos vetores de visualização, tangente, binormal e normal para cada vértice pertencente à superfície. Todas as operações são realizadas no espaço tangente, sendo necessário, portanto, transformar para este espaço e normalizar cada vetor de interesse. O procedimento detalhado para o cálculo do vetor de deslocamento de *parallax* é apresentado por Brawley e Tatarchuk (Brawley & Tatarchuk, 2004).

O vetor de visualização no espaço tangente $\boldsymbol{V_{TS}}$ é então traçado ao longo de $\boldsymbol{P}$ para determinar o ponto de interseção. Conforme discutido na Seção 4.2.1, este procedimento é executado por meio de uma busca linear sobre o perfil do mapa de profundidade, que é aproximado através de uma curva linear segmentada[4], conforme mostra a Figura 12. A computação do ponto de interseção permite determinar a quantidade de deslocamento t_{off} que deve ser somada às coordenadas de textura do ponto original t_o para corrigir a visualização dos detalhes da superfície.

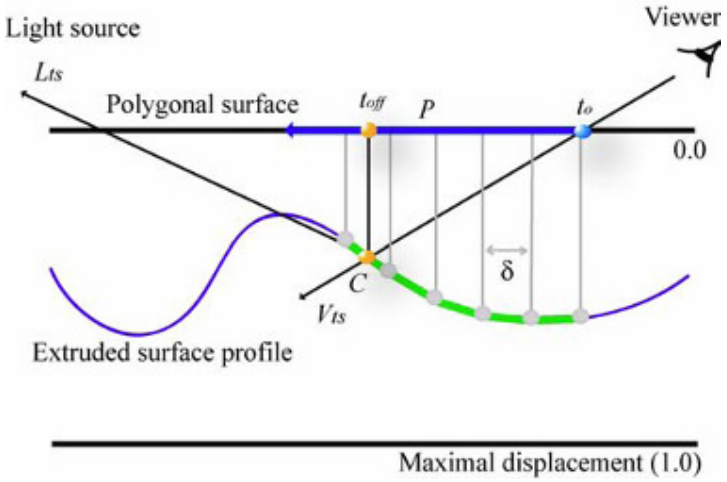


Figura 12: Amostragem do mapa de profundidade (Tatarchuk, 2006).

O intervalo entre cada amostra é de grande importância para a precisão do método, em especial no caso de mapas de altura de alta frequência, e deve assumir valores menores quanto menor o ângulo de visualização para evitar subamostragens. Nestas situações, todas as técnicas mencionadas sofrem com a ocorrência de artefatos causados por imprecisões da amostragem sobre a superfície, os quais são facilmente observados nas regiões de silhueta.

4.3 Conclusões

Mesmo com essas limitações, POM apresenta resultados bastante convincentes enquanto não aplicado a uma superfície muito complexa. O método de Policarpo (Oliveira & Policarpo, 2005) aplica à quantidade de deslocamento um fator de correção de profundidade que possibilita a sua aplicação sobre

[4] *Piecewise linear curve* na literatura original.

modelos arbitrários, mas esse fator resulta no achatamento das feições da superfície em direção ao horizonte. O *parallax mapping* original é uma técnica de implementação bastante simples e apresenta bons resultados para superfícies com protuberâncias pequenas, mas as distorções são claramente perceptíveis para mapas de altura com altas frequências ou quando a superfície é vista de ângulos rasantes.

Observa-se atualmente a utilização cada vez mais frequente do POM em jogos eletrônicos, que também já vem sendo suportado nativamente em *engines* bastante conhecidos, como CryEngine 3 (Crytek, 2012). A Figura 13 mostra uma comparação entre POM e *normal mapping* no ambiente do jogo Crysis 2 (Crysis 2, 2011), desenvolvido com o *engine* mencionado. Portanto, pelas razões apresentadas aqui, optou-se pela implementação do POM como base para uma análise comparativa dos resultados da implementação utilizando tesselação em *hardware* que é apresentada neste trabalho.

Figura 13: Renderização de cena com normal mapping e parallax occlusion mapping (imagem do jogo Crysis 2) (Crysis 2 DX11, 2011).

5
Detalhamento de superfícies com tesselação em GPU

Este capítulo descreve o procedimento realizado para a geração de detalhes em GPU usando tesselação em *hardware*. Inicialmente, será dada uma visão geral de cada método e, em seguida, serão apresentados detalhes de suas implementações, as quais têm como objetivo a geração e aplicação de detalhes estruturais sobre superfícies paramétricas, construídas a partir de um único *quad* enviado à GPU na forma de um *patch*. Para aprimorar a qualidade da imagem gerada por meio deste procedimento, também é aplicada a técnica de *normal mapping* sobre a superfície. Os resultados de cada método exposto neste capítulo são apresentados e analisados no Capítulo 6, tanto em termos de qualidade visual quanto desempenho.

5.1
Definição das superfícies paramétricas

Para os experimentos descritos neste capítulo, foram utilizados três tipos de superfícies: esfera, cilindro e *torus*, cujas coordenadas são computadas de acordo com as equações 5-1 a 5-9, as quais recebem como parâmetros as coordenadas *s* e *t*, definidas no domínio [0.0, 1.0].

$$x_{sphere} = r\sin(s)\cos(\mathrm{t}) \tag{5-1}$$

$$y_{sphere} = r\sin(s)\sin(t) \tag{5-2}$$

$$z_{sphere} = r\cos(s) \tag{5-3}$$

onde $s \in [0, \pi], t \in [0, 2\pi]$ e *r* é o raio da esfera.

$$x_{cylinder} = r\cos(s),\ s \in [0, 2\pi], t \in [0, 1] \tag{5-4}$$

$$y_{cylinder} = r\sin(s),\ s \in [0, 2\pi], t \in [0, 1] \tag{5-5}$$

$$z_{cylinder} = h\,t - (0.5h),\ t \in [0, 1] \tag{5-6}$$

onde $s \in [0, 2\pi], t \in [0, 1]$, *r* é o raio da base e *h* é a altura do cilindro.

$$x_{torus} = (R + r\cos(t))\cos(\mathrm{s}),\ s \in [0, 2\pi], t \in [0, 2\pi] \quad (5\text{-}7)$$

$$y_{torus} = (R + r\cos(t))\sin(\mathrm{s}),\ s \in [0, 2\pi], t \in [0, 2\pi] \quad (5\text{-}8)$$

$$z_{torus} = r\sin(\mathrm{t}),\ s \in [0, 2\pi], t \in [0, 2\pi] \quad (5\text{-}9)$$

onde $s \in [0, 2\pi], t \in [0, 2\pi]$, r é o raio do tubo e R é o raio do centro do torus ao centro do tubo.

5.2
Detalhamento utilizando mapas pré-computados

5.2.1.
Visão geral

A geração de detalhes por meio de mapas pré-computados obedece a um princípio similar ao do *displacement mapping* (Cook, 1984), no sentido de que utiliza informações codificadas em um mapa de altura para realizar diretamente o deslocamento da geometria do modelo. Neste caso, no entanto, os vértices da superfície deslocada são gerados na GPU através da subdivisão do *patch* de entrada. Uma vez subdividido, esse *patch* é transformado através de equações paramétricas, como as apresentadas na Seção 5.1, para gerar a topologia da superfície de interesse.

Além do *patch*, são enviadas para o *hardware* gráfico duas texturas: a primeira contém as informações de cor da superfície e a segunda codifica os mapas de normais (utilizado apenas no *fragment shader* para melhorar a qualidade da imagem renderizada) e altura, o qual constitui um escalar que é armazenado diretamente no canal alfa do mapa de normais. Exemplos dos mapas utilizados para este procedimento são mostrados na Figura 14.

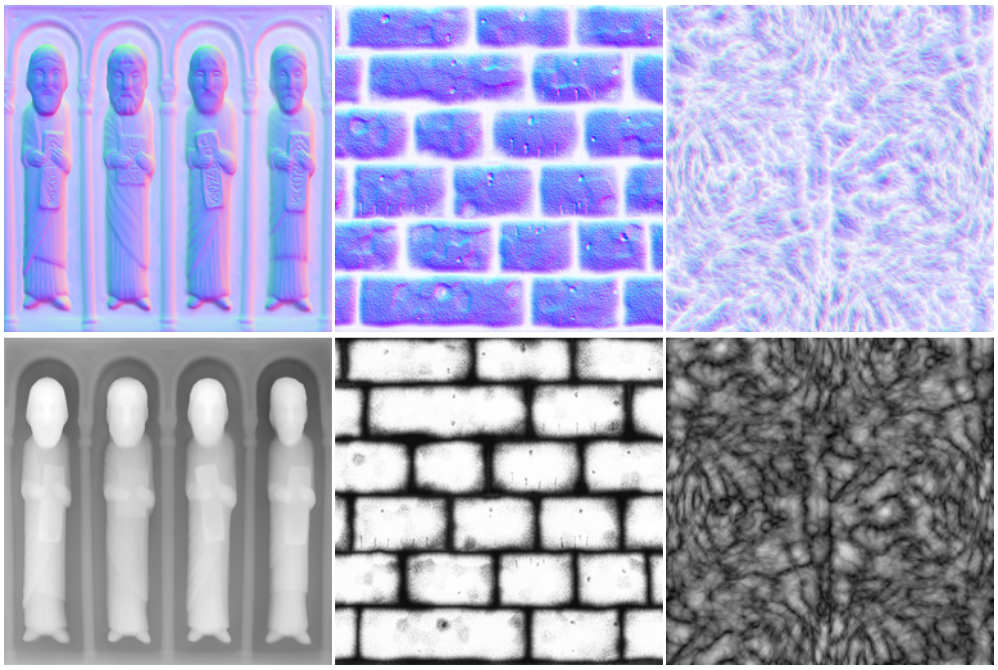


Figura 14: Mapas de normais (linha superior) e altura (linha inferior) pré-computados.

O mapa de altura é utilizado após a etapa de tesselação para deslocar os vértices da superfície. Esse processo é efetuado no espaço do objeto para cada vértice produzido e, caso esse mapa não esteja alinhado com a superfície, necessita do vetor normal deste vértice para definir a correta direção de deslocamento. Esse vetor, computado durante a etapa de subdivisão, é definido por meio do produto vetorial entre os vetores tangente e binormal associados ao vértice sendo transformado, os quais são determinados a partir dos gradientes das equações paramétricas apresentadas na Seção 5.1. Por fim, junto à textura de cor, que é mapeada diretamente sobre o objeto, o mapa de normais enviado à GPU é usado durante os cálculos de iluminação para perturbar as normais da superfície por *pixel*, processo que consiste no tradicional *normal mapping*, utilizado unicamente para melhorar o detalhamento na área interna à superfície renderizada.

5.2.2. Implementação

Na implementação deste trabalho, todas as transformações sobre vértices são calculadas no *tessellation evaluation shader*, após o estágio de geração de primitivas, quando a topologia da malha é definida. Dessa forma, o *vertex shader*

é responsável apenas pela propagação dos pontos de controle para o próximo estágio.

Como visto na Seção 3.1.2, o *tessellation control shader* recebe da aplicação os níveis de tesselação interno e externo. Visto que a primitiva de interesse é um *quad*, os dois valores correspondentes ao nível de tesselação interno e os quatro correspondentes ao nível externo são atribuídos. Também é definido nesta etapa o qualificador de *layout* do *patch* de saída, mostrado na Listagem 1, especificado através do identificador `vertices`, que recebe o número de vértices que constituem a primitiva a ser subdividida.

```
layout(vertices = 4) out;
```

Listagem 1: Declaração do layout de saída no tessellation control shader.

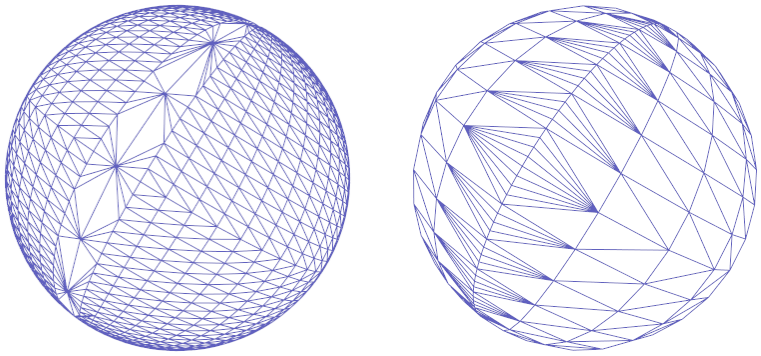
Nesta etapa, é aplicada uma pequena simplificação quando da atribuição dos níveis de tesselação, que é realizada apenas uma vez para o primeiro vértice de cada *patch*. Este controle é realizado por meio da variável interna `gl_InvocationID`, que armazena o índice no *patch* de entrada do vértice sendo processado atualmente, como mostra a Listagem 2. Na verdade, em virtude do número de *patches* utilizado aqui, esta modificação não possui impacto significativo sobre o desempenho, mas em casos de malhas mais complexas, pode evitar algumas operações redundantes.

```
if(gl_InvocationID == 0)
{
      // Atribuição dos níveis de tesselação.
      gl_TessLevelInner[0] = tessInner;
      gl_TessLevelInner[1] = tessInner;
      gl_TessLevelOuter[0] = tessOuter;
      gl_TessLevelOuter[1] = tessOuter;
      gl_TessLevelOuter[2] = tessOuter;
      gl_TessLevelOuter[3] = tessOuter;
}
```

Listagem 2: Atribuição dos níveis de tesselação no tessellation control shader.

Como a subdivisão é realizada sobre apenas um *patch*, a influência da variação do nível externo sobre a topologia da superfície não é tão significante quanto a do nível interno. Isto ocorre porque as mudanças no nível externo modificam apenas as arestas que “fecham” a primitiva (aquelas localizadas na borda do *patch* subdividido). Sobre as superfícies paramétricas apresentadas aqui, este efeito é perceptível apenas nos pontos pertencentes às arestas que delimitam a superfície, como mostra a Figura 15(a), que mostra uma superfície com níveis de

tesselação interno maior que o de tesselação externo. O efeito da variação do nível interno, por outro lado, tem influência mais direta sobre o nível de detalhamento desejado, como visto na Figura 15(b), onde o nível de tesselação interno é menor que o externo.



(a) (b)

Figura 15: Níveis de tesselação sobre superfície paramétrica.

O *tessellation evaluation shader* recebe a primitiva subdividida e é responsável pela computação das posições e atributos finais dos vértices antes do processo de iluminação. Conforme exposto na Seção 3.1.4, a declaração do *layout* de entrada utiliza três identificadores para definir o tipo de subdivisão, espaçamento e orientação da nova primitiva, os quais, para o *shader* de tesselação apresentado aqui, são: `quads`, `equal_spacing` e `ccw`, respectivamente. A declaração do layout utilizando estes identificadores pode ser vista na Listagem 3.

```
layout(quads, equal_spacing, ccw) in;
```

Listagem 3: Declaração do layout de entrada no tessellation evaluation shader.

As posições dos vértices da superfície são computadas por meio de um conjunto de funções que mapeiam as equações 5-1 a 5-9. Estas funções recebem como parâmetros de entrada as posições dos vértices que pertencem ao *patch* subdivido (as quais são interpoladas por uma função auxiliar) vindo do *primitive generator* e retornam a posição do vértice na superfície, bem como os vetores tangente e binormal associados (retornados como variáveis de saída da função paramétrica).

No caso de um ou mais *patches* de entrada com maior nível de complexidade, a interpolação garante que os vértices estarão em suas posições corretas no interior de cada nova primitiva e, consequentemente, da malha inteira. No caso de apenas um *quad*, também seria possível utilizar diretamente as

coordenadas de tesselação (*u, v*), obtidas através da variável interna `gl_TessCoord`, já que essas coordenadas são definidas no mesmo domínio da primitiva subdividida, isto é, no intervalo [0.0, 1.0].

A quantidade de deslocamento necessária para adequar o vértice à curvatura da superfície é obtida por meio de um acesso a textura, que obtém apenas a componente alfa do *texel* do mapa de altura nas coordenadas (*u, v*). Como este valor é escalar, a direção de deslocamento é determinada através da multiplicação da quantidade de deslocamento com o vetor normal do respectivo vértice (calculado durante a etapa de subdivisão) no espaço do objeto. Finalmente, o vetor contendo o valor de deslocamento devidamente ajustado é adicionado às coordenadas do vértice após a definição da topologia básica da superfície. Este processo gera as posições dos vértices no espaço do objeto. Assim, a etapa final consiste em realizar a transformação para as coordenadas de *clip* através do produto entre as coordenadas do vértice e as matrizes *model*, *view* e *projection*. A seção principal do código do *tessellation evaluation shader* para definição da topologia da superfície com deslocamento geométrico é mostrado na Listagem 4.

```
uniform mat4 modelView;
uniform mat4 projection;
uniform int primitive_type;
uniform sampler2D normalDepthMap;

// Função auxiliar para interpolação
vec4 interpolate(
            in vec4 v0, in vec4 v1,
            in vec4 v2, in vec4 v3)
{
      vec4 a = mix(v0, v1, gl_TessCoord.x);
      vec4 b = mix(v3, v2, gl_TessCoord.x);
      return mix(a, b, gl_TessCoord.y);
}

void main()
{
      vec4 pos;
      vec3 normal, tangent, binormal;
      float disp;

      pos = interpolate(
      gl_in[0].gl_Position, gl_in[1].gl_Position,
      gl_in[2].gl_Position, gl_in[3].gl_Position);

      // Definição da topologia da superfície.
      switch(primitive_type)
      {
            case SPHERE:
                  pos = sphere(pos.xy, tangent, binormal);
                  break;
            case CYLINDER:
```

```
                pos = cylinder(pos.xy, tangent, binormal);
                break;
            case TORUS:
                pos = torus(pos.xy, tangent, binormal);
                break;
    }

    tangent = normalize((tangent));
    binormal = normalize((binormal));
    normal = normalize(cross(tangent, binormal));

    // Ajuste da direção de deslocamento.
    vec3 adj =
    normal * texture(normalDepthMap, gl_TessCoord.xy).a;

    // Aplicação do deslocamento geométrico.
    pos.xyz = pos.xyz + adj;

    // Transformação para o espaço de clip.
    gl_Position = (projection * modelView) * pos;
}
```

Listagem 4: Deslocamento geométrico no tessellation evaluation shader.

Além disso, é necessário propagar para o próximo estágio as coordenadas de textura e os vetores de luz (*light*) e visualização (*view*) derivados durante esta etapa para cada vértice gerado durante a tesselação, onde a iluminação será calculada. Como os domínios de tesselação e parametrização são coincidentes, os valores das coordenadas de textura correspondem às coordenadas de tesselação especificadas pela variável `gl_TessCoord`. Os vetores *light* e *view* são propagados no espaço tangente (ou espaço da textura), onde são realizadas as computações do *normal mapping* no *fragment shader*, o que implica na necessidade de transformá-los por meio do produto com a matriz de mudança de base definida na Equação 5-10, a qual é construída com os vetores tangente (*t*), binormal (*b*) e normal (*n*), calculados neste estágio.

$$\begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \tag{5-10}$$

Por fim, o *fragment shader* realiza os cálculos de iluminação utilizando uma implementação simples do modelo de Phong, que produz boa qualidade visual a um baixo custo computacional, e da técnica de *normal mapping*. Este procedimento é codificado na função mostrada na Listagem 5, a qual recebe as coordenadas de textura e os vetores *light* e *view* normalizados no espaço tangente. As normais utilizadas neste cálculo são aquelas obtidas do mapa enviado à GPU

(já codificado no espaço tangente), amostradas com as coordenadas de textura recebidas do *tessellation evaluation shader*.

```
uniform vec4 matAmbientColor;
uniform vec4 matDiffuseColor;
uniform vec4 matSpecularColor;

uniform sampler2D colorTexture;
uniform sampler2D normalDepthMap;

vec4 computeIllumination(
            vec2 texCoord, vec3 light_ts, vec3 view_ts)
{
      // Amostragem do mapa de normais e textura de cor.
      vec3 normal_ts = normalize(
            texture(normalDepthMap, texCoord) * 2 - 1).rgb;

      vec4 baseColor = texture(colorTexture, texCoord);

      // Computação da componente difusa:
      vec3 light_tsAdj =
            vec3(light_ts.x, -light_ts.y, light_ts.z);

      vec4 diffuse =
            clamp(dot(normal_ts, light_tsAdj), 0.0, 1.0) *
            matDiffuseColor;

      // Computação da componente especular:
      vec4 specular = vec4(0.0, 0.0, 0.0, 0.0);
      vec3 half = normalize(view_ts + light_ts);
      float spec = clamp(dot(normal_ts, half), 0.0, 1.0);

      specular = pow(spec, 75) * matSpecularColor;

      // Composição da cor final:
      vec4 finalColor = ((matAmbientColor + diffuse) * baseColor +
specular);

      return finalColor;
}
```

Listagem 5: Método para iluminação da superfície no fragment shader.

5.3 Detalhamento procedimental

5.3.1. Visão geral

Conforme discutido na Seção 3.2, a geração de texturas procedimentais permite a codificação de uma variedade de efeitos para enriquecimento de ambientes virtuais. Uma das aplicações de texturas procedimentais baseadas em ruído no detalhamento de superfícies é o chamado *bump mapping* procedimental,

onde os mapas de normais utilizados para perturbar a superfície são gerados procedimentalmente. Esta aplicação, no entanto, também constitui apenas um efeito de iluminação que apresenta as mesmas limitações das técnicas baseadas em imagens apresentadas no Capítulo 4.

O uso do recurso de tesselação associado a técnicas de texturas procedimentais permite a simulação de diferentes efeitos de caráter aleatório sobre superfícies. Como na abordagem anterior, os vértices são gerados diretamente na GPU a partir de um *quad*, mas agora deslocados de acordo com um efeito procedimental baseado em ruído, o qual também é calculado na GPU.

É possível enviar uma textura gerada na CPU codificando o efeito desejado, substituindo, assim, o amplo volume de operações necessárias para computar os efeitos de ruído na GPU por algumas instruções de acesso à textura. No entanto, este experimento tem como objetivo avaliar o impacto da geração de detalhamento geométrico quando implementado inteiramente no hardware gráfico.

Para realizar o deslocamento dos vértices, foram desenvolvidos três efeitos procedimentais baseados em funções de ruído como *stripes*, *turbulence* e *fractal* (Ebert, et al., 2002): *marble*, *lumpy* e *ridge*. A parametrização da superfície ocorre da mesma forma daquela exposta na Seção 5.2, utilizando as mesmas funções para o cálculo da posição do vértice e computação dos vetores binormal e tangente. Em seguida, a quantidade de deslocamento é calculada através da chamada a uma função que codifica o efeito de detalhamento desejado, sendo então adicionada à posição do vértice após a devida correção da direção de deslocamento ao longo da normal.

Uma vez definida a topologia da malha, é realizado o processo de iluminação da superfície. Visto que não foi enviada à GPU uma textura contendo o mapa de normais, o cálculo destes vetores também é efetuado por meio de funções de ruído. O cálculo das normais *on the fly* também proporciona a liberdade de escolha do espaço onde elas podem ser computadas, por não serem recebidas pré-codificadas no espaço tangente. Dessa forma, optou-se por realizar os cálculos de iluminação diretamente no espaço do olho, evitando transformações desnecessárias entre espaços de coordenadas.

5.3.2. Implementação

Nesta abordagem, todos os efeitos procedimentais foram gerados inteiramente na GPU por meio do *tessellation evaluation shader* e *fragment shader*. O primeiro é responsável pelo deslocamento da geometria e atributos de iluminação de acordo com o efeito desejado, enquanto o segundo é responsável pelo cálculo da iluminação propriamente dita com base no efeito procedimental aplicado no estágio anterior. Portanto, os códigos para *vertex shader* e *tessellation control shader* seguem a ideia apresentada na Seção 5.2

No *tessellation evaluation shader*, as instruções de acesso à textura para obtenção da quantidade de deslocamento foram substituídas por uma chamada à função responsável pela geração do efeito de deslocamento desejado, as quais são mostradas nas Listagem 6. Observa-se que todas culminam com uma ou mais chamadas à função de ruído `noise()`, que segue a implementação apresentada por Perlin (Perlin, 2004).

```
// Funções para computação do efeito MARBLE
float marble(in vec3 pos)
{
      return (0.04f *
              stripes(pos.x + 2 *
              turbulence(pos.x,pos.y,pos.z), 1.6));
}

float turbulence(in float x, in float y, in float z) {
   float t = -0.5;
   float i = 128.0/12.0;
   for(float f = 1.0; f <= i; f *= 2)
      t += abs(noise(vec3(f*x,f*y,f*z)) / f);
   return t;
}

float stripes(in float x, in float f) {
   float t = 0.5 + 0.5 * sin(f * x * 2 * PI);
   return (t * t - 0.5);
}

// Função para computação do efeito LUMPY
float lumpy(in vec3 pos)
{
      return 0.2 * noise(2.0 * pos);
}

// Funções para computação do efeito RIDGE
float ridge(in vec3 pos)
{
      return ridgedmf(pos, 8);
}
```

```
float ridgedmf(in vec3 p, in int octaves)
{
      float lacunarity = 2.0;
      float gain = 0.5;
      float offset = 1.0;

      float sum = 0.0;
      float freq = 2.0, amp = 0.5;
      float prev = 1.0;

      for(int i = 0; i < octaves; i++)
      {
            float n = ridgeoff(noise(p*freq), offset);
            sum += n * amp * prev;
            prev = n;
            freq *= lacunarity;
            amp *= gain;
      }
      return sum;
}

float ridgeoff(in float h, in float offset)
{
    h = abs(h);
    h = offset - h;
    h = h * h;
    return h;
}
```

Listagem 6: Funções para cálculo de dos efeitos de deslocamento.

Uma vez obtida a quantidade de deslocamento, sua direção é ajustada de acordo com o vetor normal e adicionada à posição do vértice, que é multiplicada pelas matrizes *model*, *view* e *projection* para definir a topologia da superfície. Por fim, são propagadas as variáveis necessárias para a iluminação no *fragment shader*: os vetores normal, *light* e *view* (os dois últimos definidos no espaço do olho).

Em uma primeira abordagem, os vetores tangente, binormal e normal foram calculados para cada vértice no *tessellation evaluation shader* e propagados para o seguinte. Entretanto, apenas a interpolação realizada pela GPU não foi capaz de fornecer o detalhamento desejado e a qualidade visual dos efeitos resultantes não foi satisfatória, resultando em imagens borradas que pouco se assemelhavam ao efeito que se desejava obter, e, por esta razão, optou-se pelo cálculo desses vetores diretamente no *fragment shader*. Para isto, foram propagadas as coordenadas de tesselação obtidas no *tessellation evaluation shader* para realizar uma nova computação dos valores de deslocamento, que agora seriam utilizados para computar as normais em nível de *pixel*.

Inicialmente, foram calculados os vetores tangente e binormal a partir de três diferentes valores de deslocamento, em um procedimento análogo ao realizado no estágio anterior para definir a topologia da superfície. Os parâmetros de entrada nas três ocorrências, no entanto, correspondem às coordenadas de tesselação originais inalteradas, perturbadas na componente *x* e perturbadas na componente *y* com base em um fator constante. Calculados os três deslocamentos, os vetores tangente e binormal foram definidos como a diferença entre o valor de deslocamento com coordenadas inalteradas e as perturbadas nos eixos *x* e *y*, respectivamente. Em seguida, calculou-se a normal por meio do produto vetorial entre os dois vetores computados para, por fim, realizar a iluminação da superfície em um procedimento semelhante ao apresentado na Seção 5.2. A Listagem 7 mostra um excerto do código do *fragment shader*, que realiza as computações apresentadas para uma esfera detalhada com o efeito *marble*.

```
void main()
{
      vec4 matAmbientColor = vec4(0.125, 0.125, 0.25, 1.0);
      vec4 matDiffuseColor = vec4(0.25, 0.25, 0.5, 1.0);
      vec4 matSpecularColor = vec4(1.0, 1.0, 1.0, 1.0);

      float u0 = In.TexCoords.x;
      float v0 = In.TexCoords.y;

      // Perturbação das coordenadas de tesselação em x e y
      float u1 = (In.TexCoords.x + 0.0001);
      float v1 = (In.TexCoords.y + 0.0001);

      // Computação dos valores de deslocamento
      vec3 frag, frag_dx, frag_dy;
      frag    = sphere(vec2(u0, v0));
      frag_dx = sphere(vec2(u1, v0));
      frag_dy = sphere(vec2(u0, v1));

      float disp, disp_dx, disp_dy;
      disp    = marble(frag);
      disp_dx = marble(frag_dx);
      disp_dy = marble(frag_dy);

      // Computação dos valores base para tangente e binormal
      frag    = frag    + normalize(frag)    * disp;
      frag_dx = frag_dx + normalize(frag_dx) * disp_dx;
      frag_dy = frag_dy + normalize(frag_dy) * disp_dy;

      // Cálculo do vetor normal
      vec3 tangent  = normalize(frag_dx - frag);
      vec3 binormal = normalize(frag_dy - frag);
      vec3 normal   = normalize(cross(tangent, binormal));

      // Vetores light & view no espaço do olho
      vec3 light_es = normalize(In.Light);
      vec3 view_es  = normalize(In.View);
```

```
      // Componentes difusa e especular
      float diff =
            max(clamp(dot(normal, light_es), 0.0, 1.0), 0.0);
      float spec = 0.0;
      if(diff > 0.0)
      {
            vec3 half = normalize(view_es + light_es);
            spec = max(clamp(dot(normal, half), 0.0, 1.0), 0.0);
            spec = pow(spec,200);
      }

      // Cor final do fragmento
      FragColor =
            matAmbientColor +
            diff * matDiffuseColor +
            spec * matSpecularColor;
}
```

Listagem 7: Fragment shader para iluminação de superfície com base em deslocamento procedimental.

Os resultados obtidos com as implementações apresentadas aqui serão apresentados no Capítulo 6. Serão apontados os critérios utilizados durante os testes e feitas inferências acerca da qualidade visual e do desempenho de cada abordagem por meio de imagens, tabelas e gráficos, examinando a viabilidade de aplicação desses métodos em aplicações que requerem alto desempenho.

6
Resultados

Este trabalho foi implementado usando a API OpenGL versão 4.2 (OpenGL 4.2, 2012) em conjunto com a OpenGL Shading Language 4.20 (GLSL 4.20, 2011). Os testes foram realizados para diferentes tipos de superfícies paramétricas e, no caso procedimental, também para diferentes tipos de algoritmos com complexidades variáveis. Na abordagem de mapas pré-computados, foram utilizadas texturas de normais e valores de profundidade com resoluções diferentes, a fim de verificar possíveis efeitos da amostragem de mapas variados sobre o desempenho.

O algoritmo do POM, apresentado na Seção 4.2, é uma solução baseada em imagens que fornece bons resultados, mas ainda é limitada em termos de qualidade visual. No entanto, conforme mostrado na Seção 4.3, a qualidade da imagem associado ao bom desempenho proporcionado por esta técnica foi o principal fator de influência na escolha desta técnica como referência para comparação dos resultados obtidos com a implementação usando tesselação, os quais serão apresentados nas seções seguintes.

Todos os experimentos foram avaliados utilizando um computador Intel Core 2 Duo 2.93 GHz com 4 GB RAM, GPU NVIDIA GeForce GTX 560 1 GB RAM no sistema operacional Windows 7 Professional, em janelas de resolução 800 x 600 *pixels*.

6.1
Qualidade de renderização

6.1.1.
Mapas pré-computados

Em relação à qualidade visual, foi avaliada a capacidade do método apresentado em representar com precisão as feições descritas pelo mapa de altura em relação aos resultados obtidos com a implementação do POM. Para a análise

da qualidade visual, foram utilizados valores semelhantes de escala de profundidade e taxa de repetição da textura. Em ambos os casos, foram realizadas 64 subdivisões sobre a primitiva original, resultando em 8192 triângulos por superfície tesselada, para garantir o maior nível de detalhamento disponível (isto é, o máximo nível de tesselação permitido atualmente pela OpenGL).

A Figura 16 mostra comparações em pares das primitivas geométricas obtidas pela implementação baseada em tesselação (vistas na imagem superior de cada par) e pelo POM (as imagens inferiores de cada par). Observa-se que a qualidade resultante do método desenvolvido neste trabalho é superior em todos os casos, especialmente em função da correta representação de silhuetas, efeitos de oclusão causados por interpenetrações geometricamente corretas e ausência de distorções quando da observação de ângulos rasantes (que podem ser visualizadas nas regiões próximas às silhuetas das superfícies geradas com POM).

6.1.2. Abordagem procedimental

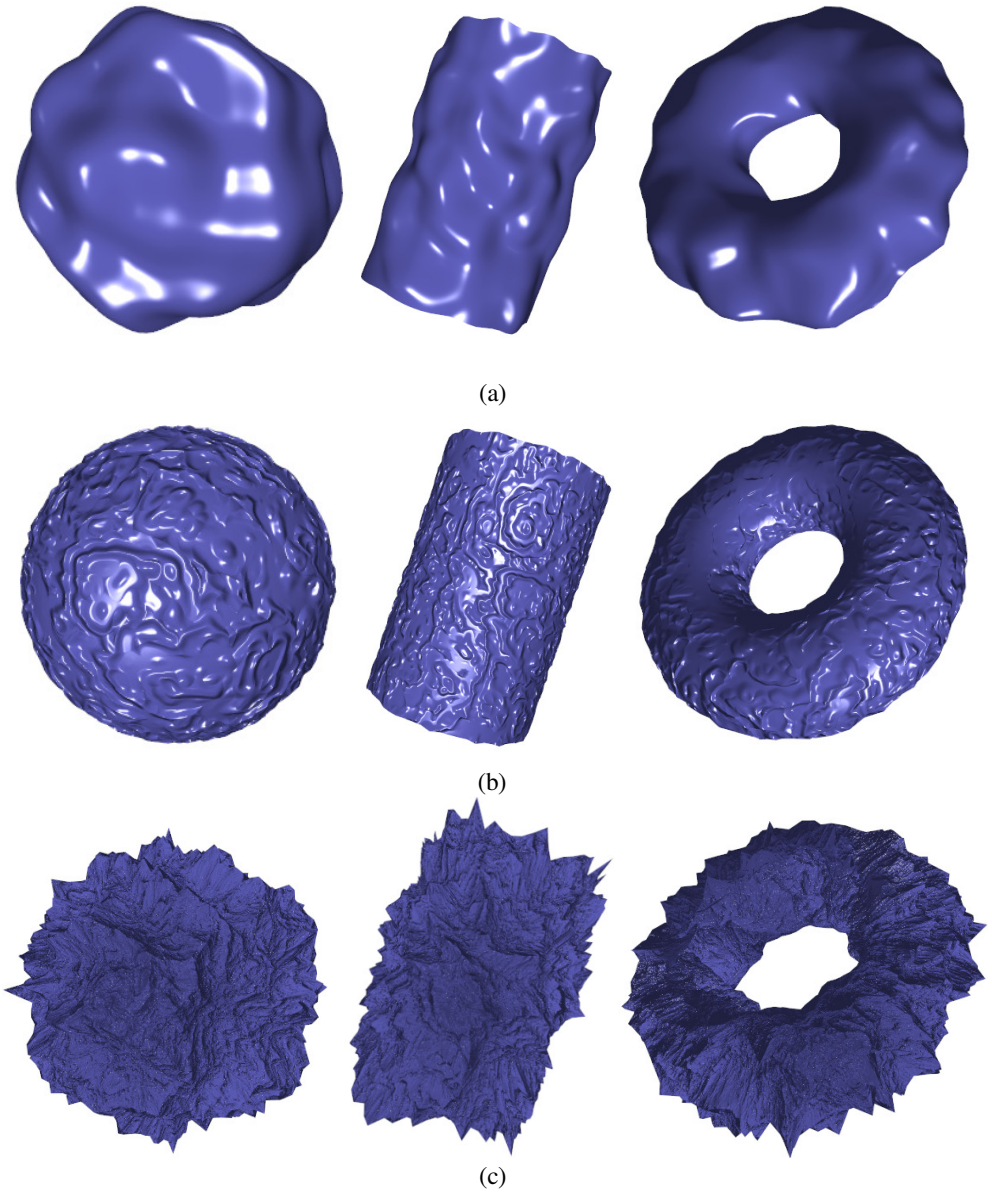
No caso das malhas cujos detalhes foram gerados procedimentalmente, não foi possível encontrar um parâmetro de comparação, já que este foi um experimento sobre a possibilidade de geração de detalhamento inteiramente na GPU, o que não era possível em versões anteriores do *pipeline* gráfico. Nesta abordagem, também foi usado o máximo nível de tesselação disponível para verificar o maior nível de detalhes que pode ser atingido com apenas uma primitiva subdividida.

A Figura 17 mostra os três efeitos procedimentais implementados neste trabalho – *lumpy*, *marble* e *ridge* – aplicados sobre as primitivas geométricas utilizadas como base. Observa-se a qualidade da computação dos detalhes internos da superfície mesmo com a ausência de uma textura de cor. Além disso, assim como na abordagem da Seção 5.2, observa-se claramente a computação das silhuetas dos objetos.

(a) (b) (c)

(d) (e) (f)

Figura 16: Comparação visual entre técnicas de detalhamento por meio de tesselação (linha superior de cada par) e simulação por pixel com POM (linha inferior). Os mapas utilizados em cada par foram intitulados rocks (a), tiles (b), parametric (c), saint (d), wall (e) e noise (f).



(a)

(b)

(c)

Figura 17: Superfícies detalhadas com os efeitos lumpy (a), marble (b) e ridge (c), gerados procedimentalmente na GPU.

6.2
Desempenho

6.2.1.
Mapas pré-computados

Uma comparação de desempenho entre ambas as abordagens é apresentada na Tabela 1, cujos dados foram transcritos no gráfico da Figura 18 para uma melhor visualização. Esta tabela relaciona o desempenho, medido em quadros por

segundo, obtido pela abordagem baseada em mapas de altura codificados em texturas com aquele obtido com a técnica de POM utilizando os mesmos mapas em cada caso, os quais, como indicado na tabela, possuem resoluções variáveis. O cálculo de sombras suaves (*soft shadows*), suportado pelo algoritmo do POM, foi desabilitado para que fossem obtidas as máximas medidas de desempenho. Além disso, o nível de tesselação para esta técnica foi reduzido a 16, resultando em apenas 512 triângulos por superfície, enquanto para a abordagem deste trabalho foram mantidas as 64 subdivisões. Neste cenário, o desempenho do método implementado aqui foi superior ao do POM em todos os casos, com uma média de ganho de aproximadamente 217,5%. É importante ressaltar que o POM é uma técnica baseada em traçado de raios cujo desempenho pode ser afetado pelo tamanho do objeto na janela de renderização. A fim de manter os testes de ambas as técnicas em condições semelhantes, todos os experimentos foram realizados sobre objetos com os mesmos valores paramétricos e a mesma distância para a câmera, de forma que tanto a malha subdividida em hardware quanto a renderizada com POM tenham os mesmos aspectos visuais em termos de tamanho ocupado na janela.

Textura	Superfície	Desempenho (FPS)		
		Tesselação	POM	Ganho (%)
Tiles (128 x 128) Figura 16(a)	Esfera	2327.615	1756.981	132.478
	Cilindro	2117.050	1472.409	143.781
	Torus	1916.711	1180.841	162.317
Rocks (128 x 128) Figura 16(b)	Esfera	2262.092	1348.397	167.762
	Cilindro	2093.285	1139.604	183.685
	Torus	1865.839	824.037	226.427
Saint (256 x 256) Figura 16(c)	Esfera	2336.967	1138.093	205.341
	Cilindro	2054.839	838.655	245.016
	Torus	1826.842	785.028	232.710
Parametric (256 x 256) Figura 16(d)	Esfera	2293.867	1046.563	219.181
	Cilindro	2004.270	755.918	265.144
	Torus	1904.460	719.364	264.742
Wall (512 x 512) Figura 16(e)	Esfera	2219.343	961.075	230.923
	Cilindro	2042.461	783.094	260.819
	Torus	1854.488	824.610	224.893
Noise (1024 x 1024) Figura 16(f)	Esfera	2148.558	894.761	240.126
	Cilindro	2025.543	748.155	270.738
	Torus	1880.545	785.809	239.313

Tabela 1: Comparação de desempenho entre detalhamento por meio de tesselação e simulação por pixel.

Por fim, a média do desempenho para cada mapa de altura é apresentada no gráfico da Figura 19, onde se observa a diferença obtida por ambas as técnicas quando do aumento da resolução dos mapas de altura (sem que tenham sido realizadas quaisquer alterações nos níveis de subdivisão). Nota-se também que, com o aumento da resolução dos mapas, ocorre uma sensível queda de desempenho na abordagem de tesselação, a qual se acentua a partir do mapa de 512x512, enquanto o desempenho obtido pelo POM se mantém mais estável para resoluções maiores.

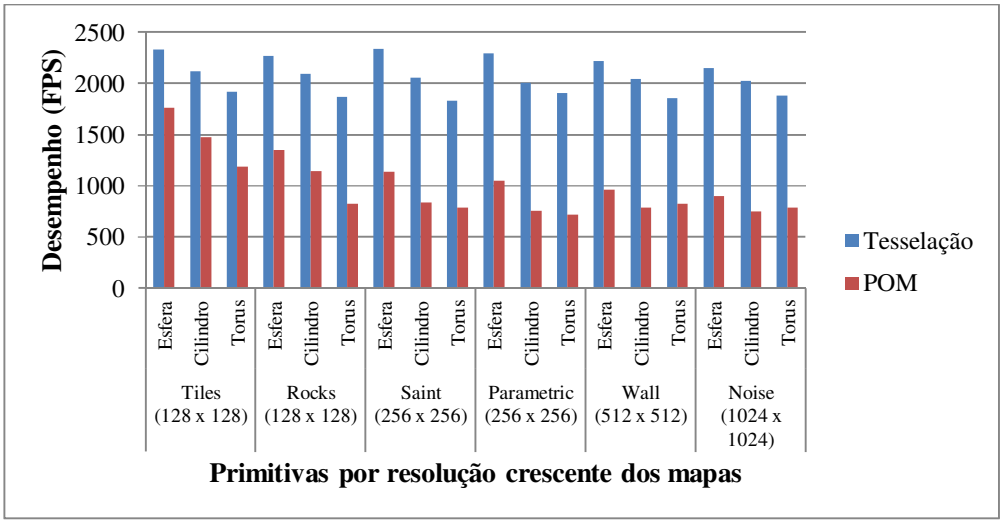


Figura 18: Relação de desempenho por primitiva entre tesselação e POM.

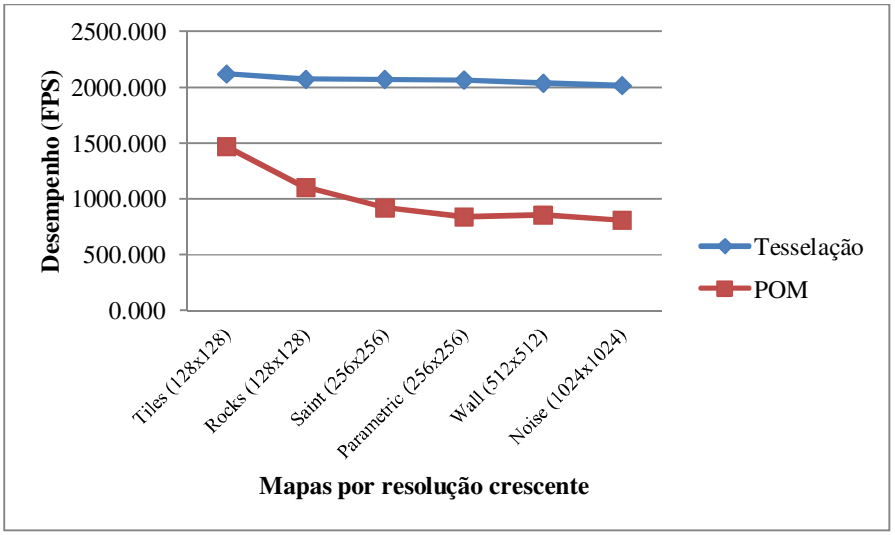


Figura 19: Média de desempenho por resolução dos mapas aplicados.

6.2.2. Abordagem procedimental

Como afirmado anteriormente, não foi possível encontrar uma técnica que pudesse ser usada como referência para comparação de resultados, então os valores fornecidos foram usados apenas para avaliação da factibilidade de implementação de tal tipo de detalhamento na GPU. A Tabela 2 mostra o desempenho obtido na computação procedimental dos efeitos *lumpy*, *marble* e *ridge* para diferentes números de instâncias simultâneas da superfície renderizada. Observa-se mais claramente com o auxílio da Figura 20 o efeito dessas múltiplas computações sobre o desempenho.

Número de Instâncias	Superfície	Desempenho (FPS)		
		Lumpy Figura 17(a)	Marble Figura 17(b)	Ridge Figura 17(c)
1	Esfera	2370.959	1185.179	495.697
	Cilindro	1798.420	711.913	345.395
	Torus	1827.016	757.204	317.355
10	Esfera	1977.952	882.853	273.843
	Cilindro	1506.645	593.815	243.994
	Torus	1494.529	614.571	216.904
100	Esfera	652.108	233.381	43.117
	Cilindro	511.597	204.167	48.974
	Torus	456.251	193.620	40.789
1000	Esfera	84.057	27.974	4.592
	Cilindro	66.791	26.947	5.462
	Torus	57.052	24.971	4.491
10000	Esfera	7.451	2.870	0.459
	Cilindro	7.170	2.813	0.552
	Torus	6.042	2.575	0.454

Tabela 2: Desempenho da abordagem de detalhamento procedimental.

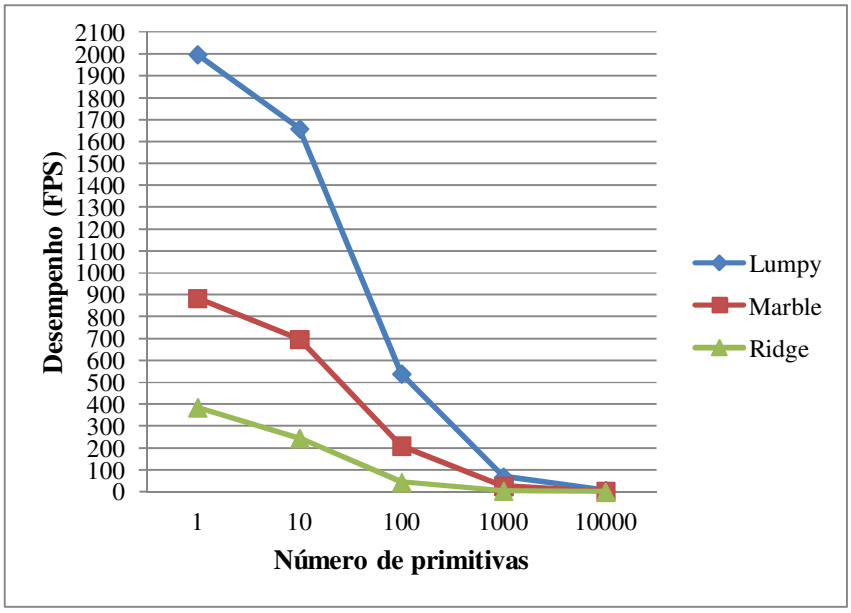


Figura 20: Desempenho médio por efeito procedimental.

Pela Listagem 7, apresentada na Seção 5.3.2, observa-se que existe um grande volume de operações realizadas por fragmento na computação dos efeitos procedimentais. O efeito *lumpy*, por exemplo, é baseado apenas na avaliação direta da função `noise()`, apresentando o maior desempenho dentre os tipos implementados.

Observa-se também pela Listagem 7 que a computação do efeito *ridge* adiciona sucessivamente novas avaliações da função de ruído, chamadas *octaves*, as quais sempre possuem o dobro da frequência da avaliação anterior e possuem influência direta na qualidade do ruído utilizado para a computação do efeito. Os valores apresentados na Tabela 2 para o desempenho proporcionado pelo efeito *ridge* foram obtidos através de oito iterações durante a computação dos *octaves*, as quais resultaram no efeito observado na Figura 17(c). A redução desse valor proporciona um desempenho maior ao custo da qualidade do ruído gerado, o que, por consequência, influencia na qualidade do efeito resultante.

Por fim, essas medidas mostram a eficiência do recurso de tesselação para detalhamento procedimental de superfícies, que, conforme mostra a Figura 17, produz resultados de alta qualidade visual apenas com o uso do alto poder de processamento da GPU. A variação de desempenho mostrada no gráfico da Figura 20 apenas ressalta a importância da utilização balanceada de métodos procedimentais e não procedimentais quando da renderização de ambientes complexos, conforme foi discutido na Seção 3.2.

7
Conclusão e trabalhos futuros

Esta dissertação investigou um método para geração de detalhamento geométrico sobre superfícies utilizando tesselação em hardware por meio de duas abordagens para deslocamento de vértices. Inicialmente, foi resumido um número de técnicas de detalhamento baseadas em imagens que realizam deslocamentos em nível de *pixel* para simular irregularidades com base em mapas de altura. Mostrou-se, no entanto, que o nível de detalhamento apresentado por essas técnicas não é capaz de representar completamente a estrutura da superfície, especialmente porque não existem modificações na geometria do objeto. Em seguida, foram apresentados detalhes da implementação realizada deste trabalho.

A primeira abordagem foi baseada em mapas de normais e altura pré-computados, os quais foram usados para deslocar os vértices das primitivas geradas na GPU com o recurso de tesselação. Este método produz modelos com correta representação de silhuetas, do efeito de *motion parallax* e sem deformações causadas por mudanças na perspectiva – problemas comumente observados nos métodos de deslocamento por *pixel*. Os resultados obtidos foram comparados em termos de qualidade visual e desempenho com a técnica de *parallax occlusion mapping*, tendo sito superiores em todos os casos apresentados, demonstrando a viabilidade de utilização plena do recurso de tesselação para este fim.

A segunda abordagem computou os dados de deslocamento procedimentalmente na GPU, baseada em um conjunto de funções de ruído implementadas em *shaders*. Este método dispensa, portanto, o uso de mapas pré-computados e permite a geração de um amplo conjunto de efeitos de detalhamento geométrico de alta qualidade. A complexidade das operações para computação de ruído apresentou uma queda de desempenho em experimentos com um grande número de primitivas, mas as taxas apresentadas ainda mostram a factibilidade do método apresentado para o detalhamento geométrico de superfícies.

É importante ressaltar que a maior contribuição deste trabalho foi apresentar a possibilidade de geração de detalhamento complexo sobre superfícies por meio da recente atualização do *pipeline* de renderização. Os resultados apresentados para a abordagem de mapas pré-computados mostram a possibilidade da plena utilização deste método para enriquecimento de ambientes virtuais a altas taxas de desempenho, enquanto os da abordagem procedimental apontam um novo caminho que, uma vez inviável em virtude de limitações tecnológicas, introduz um amplo conjunto de possibilidades a serem exploradas.

7.1 Trabalhos futuros

Visto que neste trabalho foram testados apenas casos de tesselação sobre superfícies paramétricas, surge a proposta de avaliar os efeitos da aplicação de tesselação tanto procedimental quanto baseada em mapas de profundidade sobre malhas arbitrárias. De fato, essa é a proposta original do recurso de tesselação, que ainda está sendo incorporado lentamente a engines de jogos eletrônicos, sendo geralmente utilizado apenas em elementos genéricos e de pouco destaque no cenário.

A implementação de métodos tesselação adaptativa na GPU implica no refinamento progressivo e controle automático do nível de detalhes de um objeto. Um método adaptativo para subdivisão em função de fatores como distância do objeto à câmera, visibilidade ou triângulos em regiões de silhueta pode oferecer um grande ganho em termos de desempenho sem queda perceptível na qualidade visual.

Modelos CAD geralmente apresentam um aspecto “plástico” que, apesar de muitas vezes suficiente para os usuários de softwares de engenharia, resulta em uma qualidade visual bastante degradada que pode ser melhorada por meio da tesselação em *hardware*. A relativa falta de interesse no detalhamento desse tipo de modelo decorre de outra preocupação fundamental nesta área: modelos CAD normalmente possuem uma quantidade massiva de polígonos que requerem diversos tipos de otimizações tanto de renderização quanto de memória para que sejam apenas desenhados na tela a taxas interativas (Raposo, et al., 2009).

Em muitos casos, esses modelos são descritos por um grande número de segmentos paramétricos, os quais, conforme visto neste trabalho, podem ser gerados com grande eficiência com o recurso de tesselação em *hardware*. Além disso, a geração de detalhamento procedimental pode ser uma aplicação interessante para aumentar a qualidade visual desses modelos, já que não existe a necessidade de utilizar imagens para este fim, dispensando o consumo de memória adicional com texturas. A utilização de possíveis algoritmos de tesselação adaptativa em conjunto com efeitos procedimentais pode proporcionar um ganho significativo de qualidade ao custo de um conjunto de operações adicionais de detalhamento na GPU.

8
Bibliografia

Blinn, J. F. (1978, August). Simulation of wrinkled surfaces. *Proceedings of the 5th annual conference on Computer Graphics and Interative Techniques*, pp. 286-292.

Brawley, Z., & Tatarchuk, N. (2004). Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing. *Shader X3: Advanced Rendering with DirectX and OpenGL*, pp. 135-154.

Chui, C. K. (1992). *An introduction to wavelets.* San Diego, CA, USA: Academic Press Professional, Inc.

Cook, R. L. (1984). Shade Trees. *Proceedings of the 11th conference on Computer graphics and interactive techniques*, pp. 223-231.

Cook, R. L., & DeRose, T. (2005). Wavelet Noise. *ACM Transactions of Graphics (SIGGRAPH 2005), vol. 24*, pp. 803-811.

Crysis 2. (2011). Retrieved from http://www.crytek.com/games/crysis2/overview

Crysis 2 DX11. (2011). *Crysis 2 DX11 Ultra Upgrade.* Retrieved from http://www.mycrysis.com/sites/default/files/support/download/c2_dx11_ultra_upgrade.pdf

Crytek. (2012). *CryENGINE 3.* Retrieved from http://www.crytek.com/cryengine

DeCoro, C., & Tatarchuk, N. (2007). Real-time mesh simplification using the GPU. *Proceedings of the 2007 symposium on Interactive 3D graphics and games, I3D '07*, pp. 161-166.

Doggett, M., & Hirche, J. (2000). Adaptive view dependent tessellation of displacement maps. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 59-66.

Ebert, D., Musgrave, F. K., Peachey, D., Perlin, K., Worley, S., Mark, B., & J., H. (2002). Texture & Modeling: A Procedural Approach. Morgan Kaufmann.

GL Pipeline. (2011). *OpenGL Pipeline - Lighthouse 3D.* Retrieved from http://www.lighthouse3d.com/2011/03/opengl-4-1-pipeline/

GLSL 4.20. (2011). *The OpenGL® Shading Language (Version 4.20.11 December 12, 2011).*

HAWX 2. (2010). *Tom Clancy's H.A.W.X. 2*. Retrieved from http://www.geforce.com/games-applications/pc-games/tom-clancys-hawx-2

Heidrich, W., & Seidel, H. P. (1998). Ray-tracing procedural displacement shaders. *Graphics Interface*, pp. 8-16.

Hirche, J., Ehlert, A., Guthe, S., & Doggett, M. (2004). Hardware accelerated per-pixel displacement mapping. *Graphics Interface*, pp. 153-158.

Hu, L., Sander, P. V., & Hoppe, H. (2010). Parallel View-Dependent Level-of-Detail Control. *IEEE Transactions on Visualization and Computer Graphics*, pp. 718-728.

Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T., & Tachi, S. (2001). Detailed Shape Representation with Parallax Mapping. *Proceedings of the ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence)*, pp. 205-208.

Kautz, J., & Seidel, H.-P. (2001). Hardware accelerated displacement mapping for image based rendering. *Proceedings of Graphics Interface 2001*, pp. 61-70.

McGuire, M., & McGuire, M. (2005). Steep Parallax Mapping. *Poster at ACM Symposium on Interactive 3D Graphics and Games (I3D 2005)*.

Moule, K., & McCool, M. (2002). Efficient bounded adaptive tesselation of displacement maps. *Proc. Graphics Interface*, pp. 171-180.

Ni, T., & Castano, I. (2009). Efficient Substitutes for Subdivision Surfaces. *SIGGRAPH 2009 Course Notes*.

Oliveira, M. M., & Policarpo, F. (2005). An Efficient Representation for Surface Details. *UFRGS Technical Report RP-351*.

Oliveira, M. M., Bishop, G., & McAllister, D. (2000). Relief texture mapping. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 359-368.

OpenGL 4.2. (2012). *The OpenGL® Graphics System: A Specification (Version 4.2 (Core Profile) – April 27, 2012).*

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J., & Phillips, J. (2008). GPU Computing. *Proceedings of the IEEE, 96*, pp. 879-899.

Perlin, K. (1985). An Image Synthesizer. *Proceedings of SIGGRAPH 85. In Computer Graphics (1985), vol. 19*, pp. 287-296.

Perlin, K. (2002). Improving Noise. *ACM Transactions on Graphics (SIGGRAPH 2002), vol. 21*, pp. 681-682.

Perlin, K. (2004). Implementing improved Perlin Noise. *GPU Gems*, pp. 73-85.

Pharr, M., & Hanrahan, P. (1996). Geometry caching for raytracing displacement maps. *Eurographics Rendering Workshop 1996*, pp. 31-40.

Policarpo, F., Oliveira, M. M., & Comba, J. L. (2005). Real-time relief mapping on arbitrary polygonal surfaces. *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 155-162.

Raposo, A., Santos, I., Soares, L., Wagner, G., Corseuil, E., & Gattass, M. (2009). Environ: Integrating VR and CAD in Engineering Projects. *IEEE Computer Graphics & Applications, v.29*(n.6), pp. 91-95.

Smits, B., Shirley, P., & Stark, M. M. (2000). Direct ray tracing of displacement mapped triangles. *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pp. 307-318.

Stich, M., Wächter, C., & Keller, A. (2007). Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders. In H. Nguyen, *GPU Gems 3* (pp. 239-255). Addison-Wesley Professional.

Szirmay-Kalos, L., & Umenhoffer, T. (2008). Displacement Mapping on the GPU — State of the Art. *Computer Graphics Forum 27*.

Tatarchuk, N. (2006). Dynamic parallax occlusion mapping with approximate soft shadows. *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 63-69.

Tatarchuk, N., Shopf, J., & DeCoro, C. (2007). Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline. *ACM SIGGRAPH courses, SIGGRAPH '07*, pp. 122-137.

Tatarinov, A. (2008). Instanced tessellation in DirectX10. *GDC '08: Game Developers' Conference 2008*.

Valdetaro, A., Nunes, G., Raposo, A., & Feijó, B. (2010). Understanding Shader Model 5.0 with DirectX 11. *SBGames 2010 – IX Simpósio Brasileiro de Jogos e Entretenimento Digital*, pp. 1-18.

Wang, L., Wang, X., Tong, X., Lin, S., Hu, S., Guo, B., & Shum, H.-Y. (2003). View-dependent displacement mapping. *ACM Trans. Graph., 22*, pp. 334-339.

Wang, X., Tong, X., Lin, S., Hu, S., Guo, B., & Shum, H.-Y. (2004). Generalized displacement maps. *Eurographics Symposium on Rendering 2004*, pp. 227-233.

Welsh, T. (2004). Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces. Infiscape Corp.