Gustavo Bastos Nunes

Aplicações para geração de vértices em GPU

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–graduação em Informática

Rio de Janeiro
Agosto de 2011

Pontifícia Universidade Católica
do Rio de Janeiro

Gustavo Bastos Nunes

Aplicações para geração de vértices em GPU

Dissertação de Mestrado

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

Orientador : Prof. Alberto Barbosa Raposo
Co–Orientador: Prof. Bruno Feijó

Rio de Janeiro
Agosto de 2011

Pontifícia Universidade Católica
do Rio de Janeiro

Gustavo Bastos Nunes

Aplicações para geração de vértices em GPU

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

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

Prof. Bruno Feijó
Co–Orientador
Departamento de Informática — PUC-Rio

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

Prof. Rodrigo Penteado Ribeiro de Toledo
Instituto de Matemática — UFRJ

Dr. Luciano Pereira Soares
Tecgraf — PUC-Rio

José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico — PUC-Rio

Rio de Janeiro, 16 de Agosto de 2011

Gustavo Bastos Nunes

Graduou-se em Engenharia de Computação na Pontifícia Universidade Católica do Rio de Janeiro em 2008. De 2008 a 2010 trabalhou no laboratório de Computação Gráfica da PUC-RIO (TecGraf) desenvolvendo sistemas de realidade virtual e visualização científica. De Dez/2010 a Maio/2010 trabalhou na T&T desenvolvendo simuladores virtuais e serious games. A partir de Outubro/2010 começará na Microsoft(Redmond) como Software Development Engineer in Test na equipe do Microsoft SharePoint.

Ficha Catalográfica

Nunes, Gustavo Bastos

Aplicações para geração de vértices em GPU / Gustavo Bastos Nunes; orientador: Prof. Alberto Barbosa Raposo; co–orientador: Prof. Bruno Feijó. — Rio de Janeiro : PUC-Rio, Departamento de Informática, 2011.

v., 111 f: il. ; 29,7 cm

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

Inclui referências bibliográficas.

1. Informática – Tese. 2. Hardware Tessellation. 3. Programação em GPU. 4. Geração de vértices. 5. DirectX11. 6. OpenGL4. 7. Shader Model 5.0. I. Raposo, Alberto Barbosa. II. Feijó, Bruno. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

À Deus que me deu força sempre.

Agradecimentos

Aos meus pais, Antonio e Maria, por me proporcionarem um ensino de qualidade. Às minhas irmãs, Silvia e Flavia. À Marcela por sempre me apoiar. Aos meus amigos Alexandre e Rodrigo por partilharem seus conhecimentos comigo. Aos amigos do Siviep, Thiago, Pablo, Siviano, Henrique, Galak, Mariano e Galinha Velha, pelos bons momentos de TecGraf. Aos Professores Alberto e Bruno pela motivação a cada idéia que surgia na minha cabeça. À professora Karin pela apoio em toda minha vida acadêmica. Ao professor Carlos Tomei por estar sempre disponível. Ao Professor Rodrigo pelas idéias e motivações. Aos professores Gattass e Waldemar, pelas excelentes aulas de Computação Gráfica que pude desfrutar. Aos demais professores do DI pela qualidade de ensino proporcionada. Aos meus amigos de Niterói, Falcão, Decão, Bê de óculos, Costela, Lopes, Dudu, Augusto, Daniel, André e respectivas pelas faltas nos churrascos.

Resumo

Nunes, Gustavo Bastos; Raposo, Alberto Barbosa; Feijó, Bruno. **Aplicações para geração de vértices em GPU**. Rio de Janeiro, 2011. 111p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Um dos maiores gargalos do pipeline gráfico hoje é largura de banda ainda insuficiente disponível entre a GPU e CPU. Para evitar esse gargalo novas funcionalidades programáveis foram inseridas nas placas de vídeos. Com o Geometry Shader foi possível criar vértices em GPU, porém, este estágio da pipeline apresenta uma performance baixa. Com o lançamento das novas APIs gráficas (DirectX11 e OpenGL4) em 2009, foi adicionado o Tessellator, que permite a criação de vértices em massa na GPU. Esta dissertação estuda este novo estágio da pipeline, bem como implementa algoritmos clássicos (PN-Triangles e Phong Tessellation) que originalmente foram feitos para CPU e propõe novos algoritmos (Renderização de Tubos e Terrenos em GPU) para tirar proveito deste novo paradigma.

Palavras–chave

Hardware Tessellation. Programação em GPU. Geração de vértices. DirectX11. OpenGL4. Shader Model 5.0.

Abstract

Nunes, Gustavo Bastos; Raposo, Alberto Barbosa; Feijó, Bruno. **Applications for real time vertex generation in GPU**. Rio de Janeiro, 2011. 111p. MSc Thesis — Department of Computer Science, Pontifícia Universidade Católica do Rio de Janeiro.

One of the main bottlenecks in the graphics pipeline nowadays is the still insufficient memory bandwidth between the CPU and the GPU. To avoid this bottleneck programmable features were inserted into the video cards. With the Geometry Shader launch it was possible to create vertices in the GPU, however, this pipeline stage has a low performance. With the new graphic APIs (DirectX11 and OpenGL4) a Tessellator stage that allows massive vertex generation inside the GPU was created. This dissertation studies this new pipeline stage, as well as implements classic algortihms (PN-Triangles and Phong Tessellation) that were originally designed for CPU and proposes new algorithms (Tubes and Terrain rendering in the GPU) that takes advantage of this new paradigm.

Keywords

Hardware Tessellation. GPU Programming. Vertex generation. DirectX11. OpenGL4. Shader Model 5.0.

Sumário

Lista de figuras

Lista de tabelas

Quando alguém pergunta "Qual o caminho?".
O Zen simplesmente responde "Caminhe".

Ditado Budista, *Autor Desconhecido.*

1
Introdução

A área de renderização em tempo real sempre proporcionou grandes desafios aos pesquisadores interessados. Elaborar algoritmos capazes de processar uma cena foto-realística em uma taxa de quadros por segundo interativa não é tarefa fácil.

Apesar da grande evolução do hardware gráfico ao longo dos anos, a demanda por uma visualização mais imersiva é constante. Essa imersão significa, na maioria das vezes, mais realismo. Ou seja, texturas maiores, simulações físicas mais realísticas, modelos com grande quantidade de vértices e algoritmos mais complexos, entre outras consequências.

Para ajudar na elaboração de tais algoritmos, as placas de vídeo vêm evoluindo constantemente nos últimos anos. Até 2000 as placas de vídeo tinham um pipeline fixo que era apenas configurável [Wikipedia, 2010]. Ou seja, os algoritmos de computação gráfica eram restringidos a usar alguns recursos pouco flexíveis das GPUs.

A partir daí começou a surgir o pipeline programável. Em 2001 foi lançado o DirectX8 com suporte a programação de vertex shaders e pixel shaders em uma linguagem assembly. Esta versão tinha disponibilidade de registradores e acesso a texturas limitadas, mas foi a primeira versão que permitia programar o hardware gráfico. Foi chamado de Shader Model 1.0. Em 2002 foi lançado o DirectX9 com suporte ao Shader Model 2.0 e introdução de uma linguagem de shader de alto-nível chamada HLSL (High Level Shader Language). No Shader Model 2.0 foi introduzido o conceito de múltiplos alvos de renderização (multiple render-targets), texturas com precisão de ponto flutuante e mais registradores de acesso [Wikipedia, 2010].

Em 2004 o DirectX9 sofreu uma atualização grande, chamada de DirectX9.0c. Nesta atualização houve a introdução do Shader Model 3.0, permitindo uma série de instruções novas, incluindo acesso a textura no Vertex Shader. Porém, neste Shader Model ainda existiam apenas dois estágios programáveis no pipeline. Era possível apenas manipular vértices ou pixels, não era possível ainda criar informação em tempo de execução na GPU.

Em 2006 foi lançado o DirectX10 com suporte ao SM 4.0(Shader Model

4.0) [Limaye, 2009]. Foi introduzido um novo estágio no pipeline, o Geometry Shader. Pela primeira vez era possível criar vértices na GPU. Os programadores tinham acesso a cada triângulo que passava na placa gráfica, incluindo suas adjacências. Muitos pensaram que poderiam criar milhões de vértices em tempo de execução na placa gráfica. Porém, apesar de o Geometry Shader ser útil para uma série de algoritmos, ele é um estágio lento. O controle de fluxo ou criação de muitos vértices neste estágio faz com que a aplicação tenha uma queda de desempenho drástica, tornando inviável a renderização em tempo real.

Apesar da recente migração da interface de I/O de AGP para PCI-Express [Jim Brewer, 2004][Bhatt, 2004], o recurso mais caro em todo o pipeline gráfico hoje ainda é a largura de banda [Owens et al., 2008]. Enviar milhares de vértices a cada quadro para a placa de vídeo é um processo custoso. Por isso muitas aplicações de visualização 3D em tempo real ainda usam modelos com baixa quantidade de polígonos para garantir uma taxa de quadros por segundo alta.

Por outro lado, um dos recursos mais baratos no pipeline é a alta capacidade de processamento da GPU. Uma única placa de vídeo hoje tem até 512 processadores trabalhando em paralelo [NVIDIA, 2010], além do fato que múltiplas placas de vídeo podem ser combinadas montando clusters poderosos. Desta maneira, fica claro que uma troca de transferência de memória por operações aritméticas na GPU seria muito benéfica para a melhora de desempenho do pipeline gráfico.

Com base nisto, em 2008 foi proposto um novo pipeline gráfico baseado no DirectX11 [Drone et al., 2010] que introduz o Shader Model 5.0 com duas novas partes programáveis e uma parte fixa, permitindo assim a criação de primitivas em massa na GPU. Em Setembro de 2009 chegava ao mercado as primeiras placas com suporte ao DirectX11, foi a série HD5xxx da ATi. Mais tarde, em abril de 2010, a Nvidia lançou a série Geforce 4xx com suporte nativo ao DirectX11. Um gráfico da evolução do pipeline baseado nas placas da Nvidia e no DirectX é apresentado na Figura 1.1.

A proposta deste trabalho consiste em implementar algoritmos que tirem proveito da geração dinâmica de vértices na GPU e que anteriormente eram feitos em CPU, além de propor algoritmos novos que explorem o potencial do novo pipeline gráfico, e sempre avaliando o ganho de desempenho do Tessellator criando vértices na GPU ao invés de enviá-los da CPU para GPU. Esta dissertação usa como nomenclatura os termos da API do DirectX, porém, tudo descrito aqui pode ser feito com a API do OpenGL, apenas as nomenclaturas de alguns termos dessas APIs são diferentes.

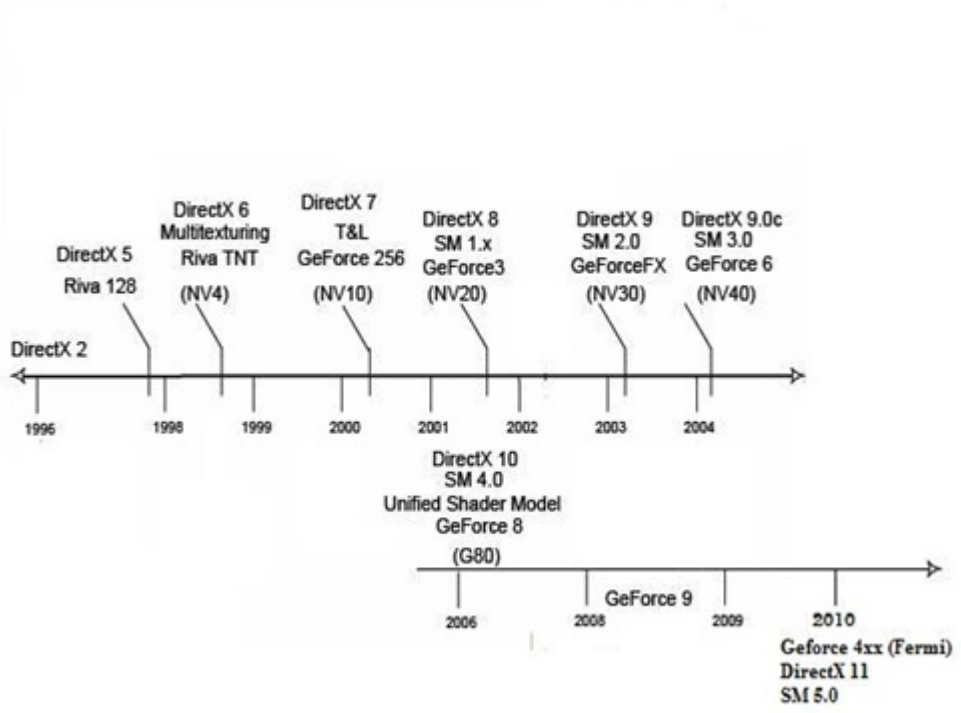

Figura 1.1: Linha do tempo da evolução do pipeline baseado nas placas Nvidia

A primeira parte do trabalho avalia o desempenho do Tessellator em comparação com o envio dos dados dos vértices pela CPU. Depois analisaremos o desempenho do PN-Triangles e do Phong Tessellation no Tessellator. Estes dois algoritmos têm como característica o uso de informações locais para o refinamento do malha. Em seguida são propostas duas novas soluções que usam o Tessellator: Renderização de Tubos e Terrenos em GPU.

Este documento está organizado como se segue. No capítulo que se segue, será apresentado o novo pipeline gráfico e discutida sua performance e economia de largura de banda. No capítulo 3 o novo pipeline será usado para implementar 2 algoritmos de subdivisão de superfícies: PN-Triangles e Phong Tessellation, que serão comparados entre si em termos qualitativos e quantitativos. Nos capítulos seguintes, serão mostrados dois novos algoritmos desenvolvidos para o novo pipeline: primeiramente um algoritmo para renderização de tubos com anti-aliasing e LOD contínuo e, no capítulo seguinte, um algoritmo para renderização de terrenos. Finalmente, o capítulo 6 apresenta as conclusões e trabalhos futuros.

2
O Novo Pipeline Gráfico

Este capítulo tem o objetivo de apresentar o novo pipeline gráfico e avaliar sua perfomance ao gerar vértices em GPU, em comparação com a abordagem antiga de passar todos os vértices da CPU para GPU. A tentativa de desafogar a parte de geometria do pipeline gráfico substituindo malhas por outras representações já foi vastamente explorada. Técnicas de Level-of-Detail de modelos [Hoppe, 1996], [Hoppe, 1998], [Hu et al., 2010a], [Hoppe, 1997] diminuem o número de vértices de um modelo dinamicamente, fazendo com que não seja necessário passar para a GPU todos os vértices quando a câmera está muito longe de uma malha. Porém, quando o modelo está muito próximo da câmera é inevitável passar todos os vértices. Billboards e point sprites também são usados para evitar a renderização de geometria [Fernando, 2003], mas modelos complexos ficam visualmente muito ruins ao serem representados por billboards perto da câmera. O uso do Geometry Shader também foi explorado para evitar a sobrecarga no barramento de transferência para a placa gráfica [Lorenz and Döllner, 2008], contudo, este estágio da pipeline gráfica é muito lento e o desempenho deixa a desejar. Além disso, o Geometry Shader só tem a capacidade de fazer um LOD discreto da malha, isso acarreta nos chamados *poppings*. O Tessellator oferece suporte nativo a uma transição visual contínua(geomorphing, [Hoppe, 1996]). De Toledo e Levy [de Toledo and Levy, 2004], [de Toledo and Lévy, 2008a] propõem o uso de ray-casting no pixel shader para estender o pipeline gráfico e criar novas primitivas. Apesar de apresentar bons resultados visuais, o ray-casting é muito intenso em operações aritméticas e nos trabalhos mencionados os autores só propõem a criação de primitivas até a quarta ordem.

A seguir vamos apresentar o novo pipeline gráfico e criar uma aplicação simples que explora a criação de vértices na GPU e que permite criar qualquer superfície paramétrica na placa gráfica passando apenas um vértice. O maior objetivo da seção é avaliar o ganho de performance entre a abordagem de passar todos os dados dos vértices da CPU para GPU em comparação com a geração de vértices em GPU.

2.1 Possibilidades no novo pipeline gráfico

Antes do Shader Model 4.0, não havia possibilidade de fazer manipulações na GPU por primitiva, também não era possível adicionar ou remover nenhuma primitiva no pipeline gráfico. Vertex e pixel shaders só podiam executar seus programas em dados que já estavam na memória. Porém, placas compatíveis com o DirectX10 adicionaram um novo estágio no pipeline gráfico chamado *Geometry Shader*. Vários algoritmos tiraram proveito desse novo estágio da pipeline: detecção de silhouetas [Doss, 2008], cube-mapping com uma única passada [SDK, 2007], refinamento de malhas [Lorenz and Döllner, 2008], entre outros. Porém, o Geometry Shader pode gerar apenas uma quantidade limitada de primitivas e as operações de adicionar/remover primitivas nesse estágio são consideravelmente custosas.

Para entender o propósito do novo pipeline gráfico, um dos principais gargalos da renderização em tempo real precisa ser analisado: a transferência de modelos com grande quatidade de primitivas da CPU para a GPU, essa questão será brevemente explicada nesta seção.

Modelos que se propõem a representar corpos reais devem ter superfícies suaves e contínuas. Para criá-las, existe uma série de algoritmos propostos. Esses algoritmos podem ser divididos grosseiramente em dois grupos: o primeiro grupo contém os métodos que têm uma malha grosseira como seu domínio, e uma malha refinada como sua imagem. Alguns exemplos são: superfícies de Bézier, superfícies de Subdvisão de Catmull-Clark e malhas com diferentes níveis de detalhes. O segundo grupo contém os algoritmos que já possuem uma malha refinada como seu domínio, e também uma malha refinada como sua imagem, de modo que seu domínio só recebe transformações, em contraste com o primeiro grupo, que refina e transforma seu domínio. Alguns exemplos desse segundo grupo são: superfícies paramétricas, mapas de altura de terrenos, oceanos, entre outros.

Sem a possibilidade de adicionar ou remover primitivas fica obviamente impossível de executar qualquer algoritmo do primeiro grupo inteiramente em GPU, visto que o único lugar que a malha poderia ser refinada é na CPU, o que pode afetar muito a performance da aplicação. Além desta impossibilidade, ainda existe o problema de transferir uma malha densa para a placa de vídeo, o que é necessário antes da execução do segundo grupo de algoritmos no pipeline gráfico. Como mencionado anteriormente, a largura de banda entre a CPU e a GPU é o fator limitante na renderização deste grupo de algoritmos.

O novo pipeline foi criado na tentativa de melhorar a performance para essas questões. Foram criados novos estágios no pipeline, *Hull Shader*,

Tessellator, e *Domain Shader*, sendo o primeiro e o último programáveis. Existe também um novo tipo de primitiva chamada *patch*, que consiste de um número de vértices ou pontos de controle (até 32). Apesar de a criação de uma primitiva que aceite 32 pontos de controle já ser util por si só, o que realmente faz a diferença no novo pipeline é o *Tessellator*. Esse novo estágio da pipeline gráfica pode criar até 8192 triângulos para cada primitiva que ele recebe como input. O número exato de primitivas a serem criadas é passado por parâmetro. Todos esses novos triângulos criados podem ser transformados programaticamente em um estágio da pipeline onde o programador pode acessar não só os dados da primitiva originalmente enviada, mas também as coordenadas dos vértices gerados em GPU. Essa possibilidade de criar uma quantidade massiva de triângulos na placa gráfica e poder manipulá-los com certa facilidade faz da criação deste novo pipeline um novo paradigma na renderização em tempo real.

Figura 2.1: Modelo com grande quantidade de polígonos[Castano, 2008]

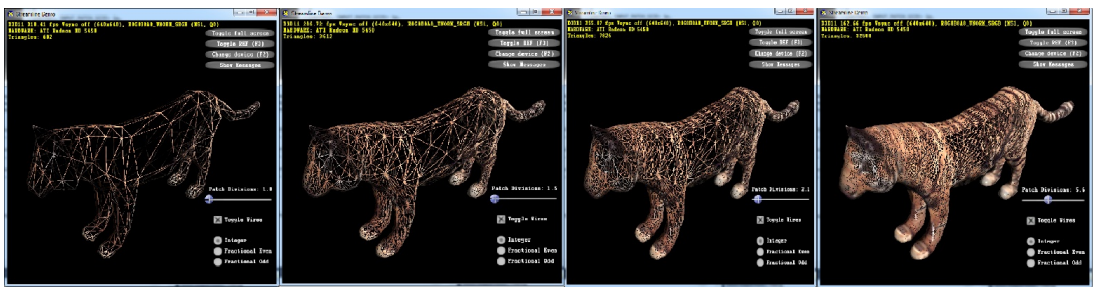

Figura 2.2: Modelo com diferentes níveis de detalhe produzidos na GPU

A programação dos novos shaders é extremamente flexível e feita para criar malhas complexas(Figura 2.1) em tempo de execução, além de suportar

algoritmos de níveis de detalhe com *geomorphing* na GPU. A Figura 2.2 mostra um modelo com diferente níveis de detalhe construídos na GPU.

2.2 Introdução ao novo pipeline

A Figura 2.3 representa todos os estágios disponíveis nas novas placas de vídeo. Nesta seção vamos detalhar cada estágio e seus papeis no novo pipeline.

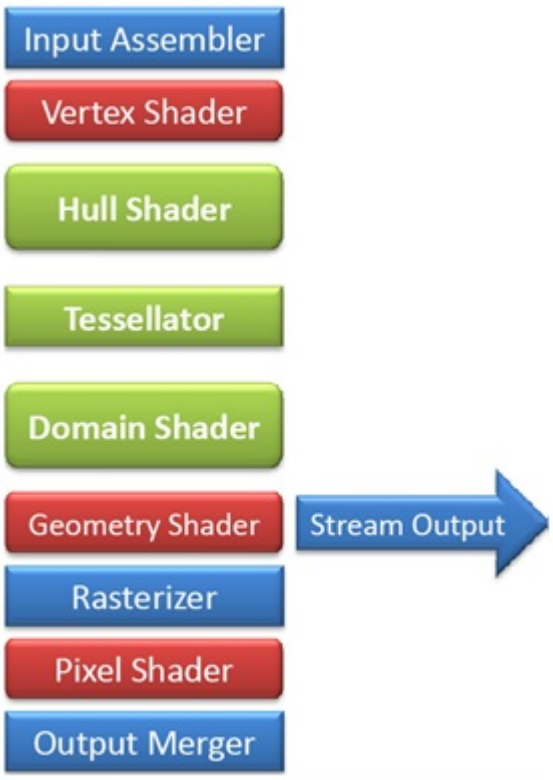

Figura 2.3: O novo pipeline gráfico[Rocco, 2010]

2.2.1 Input Assembler

O Input Assembler recebe do programa o tipo de primitiva que será passada pelo pipeline. Anteriormente existiam basicamente 4 tipos de primitivas, triângulos, quads, linhas e pontos. Neste novo pipeline foi introduzido um novo tipo de primitiva chamada *patch*. Quando o Tessellator está habilitado, o pipeline só aceita patches como entrada do Input Assembler.

Um patch pode ter de 1 até 32 pontos de controle. Não existe topologia implícita ao se declarar um patch. Cabe ao programador decidir que tipo de topologia aquele patch tem. Por exemplo, um patch com 3 pontos de controle pode representar tanto um triângulo quanto dados para a tecelagem de uma linha. Um patch com 16 control points poderia representar um quad com

curvatura, uma bi-cúbica de Bézier por exemplo. Ou seja, o programador pode usar os pontos de controle para passar qualquer tipo de informação necessária para a renderização. A Figura 2.4 mostra um patch com 16 pontos de controle representando uma superfície de Bezier.

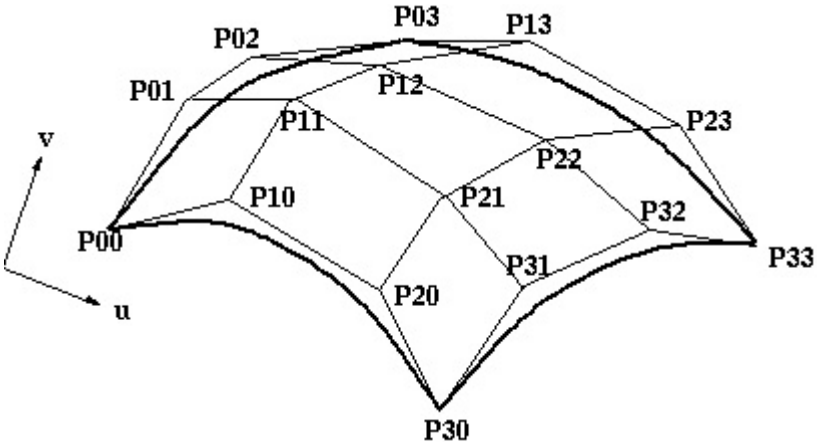

Figura 2.4: Patch com 16 pontos de controle representando uma superfície de Bézier

2.2.2 Vertex Shader

O Vertex Shader recebe os pontos de controle de cada patch e pode deslocá-los para qualquer lugar no espaço de mundo. Neste estágio a tecelagem ainda não ocorre. No caso de um modelo 3D, o Vertex Shader estará deslocando apenas a *gaiola de controle*. Isso é muito importante, pois uma das operações aritméticas mais caras era animar modelos que tinham muitos triângulos no Vertex Shader. Com o novo pipeline, anima-se apenas os poucos vértices da gaiola de controle e o modelo tecelado sai animado no final do pipeline. A Figura 2.5 representa a animação da gaiola de controle e a tecelagem que ocorre após. Além disso, não é mais no Vertex Shader que se transforma de espaço de mundo para espaço de tela (multiplicando pelas matrizes de View e Projection), esta tarefa cabe agora ao Domain Shader.

2.2.3 Hull Shader

Depois do Vertex Shader, o Hull Shader é invocado para cada patch com todos os vértices transformados pelo estágio anterior. No Hull Shader deve ser declarado quantos pontos de controle vão sair deste estágio. O Hull Shader será invocado baseado na quantidade de pontos de controle declarada. Isto serve basicamente para uma mudança de base. Por exemplo, o Input Assembler pode receber 4 pontos de controle por primitiva, o Hull Shader pode declarar que

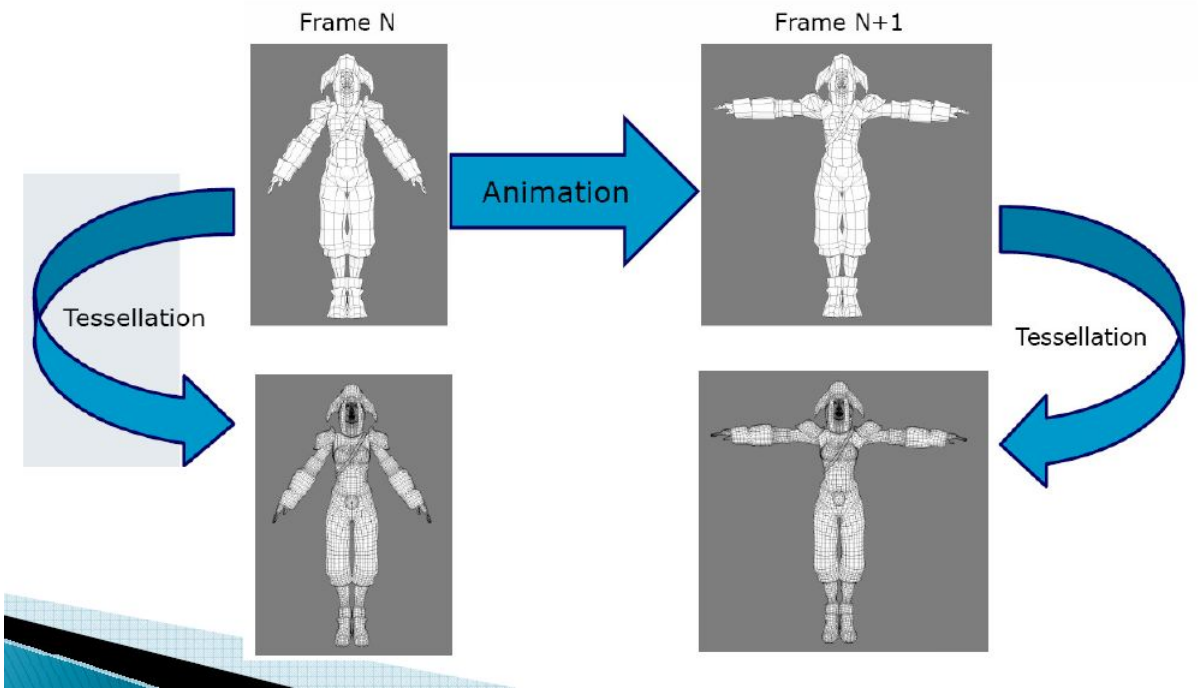

Figura 2.5: Economia ao animar modelo em baixa frequência [Tatarinov, 2008]

sairão dele 16 pontos de controle por primitiva para fazer a mudança de base de quad para Bi-Cúbica de Bezier. Outra tarefa do Hull Shader é computar os fatores de tecelagem tanto das arestas quanto do interior da primitiva. Estes fatores indicam o quanto o Tessellator deve subdividir cada primitiva. Também deve ser explicitado no Hull Shader qual é o domínio de subdivisão que o Tessellator usará (triângulos, quads ou linhas). A Figura 2.6 ilustra o fluxograma do Hull Shader.

Para executar estas tarefas, o Hull Shader precisa ser paralelizado explicitamente. Isto é, ao invés de ter uma thread por patch para computar todos os pontos de controle e fatores de tecelagem, o Hull Shader é dividido em três fases paralelas representadas pela Figura 2.7. Cada uma dessas fases é composta por um número de threads igual ao número de pontos de controle que sairá deste estágio (entre 1 e 32, definido pelo programador). Essas threads não podem se comunicar entre si, mas cada fase pode ver a saída da fase anterior.

Isto permite ao programador, por exemplo, computar os pontos de controle na primeira fase, *Control Point Phase*, baseado nesses control points computar os fatores de tecelagem das arestas na segunda fase, *Fork Phase*, e finalmente baseado nos fatores de tecelagem das arestas computar a tecelagem do interior da primitiva na última fase, *Join Phase*.

Para simplificar a programação, a segunda e a terceira fase não são disponíveis explicitamente. O usuário as programa na mesma função e é papel do compilador separá-las para a paralelização.

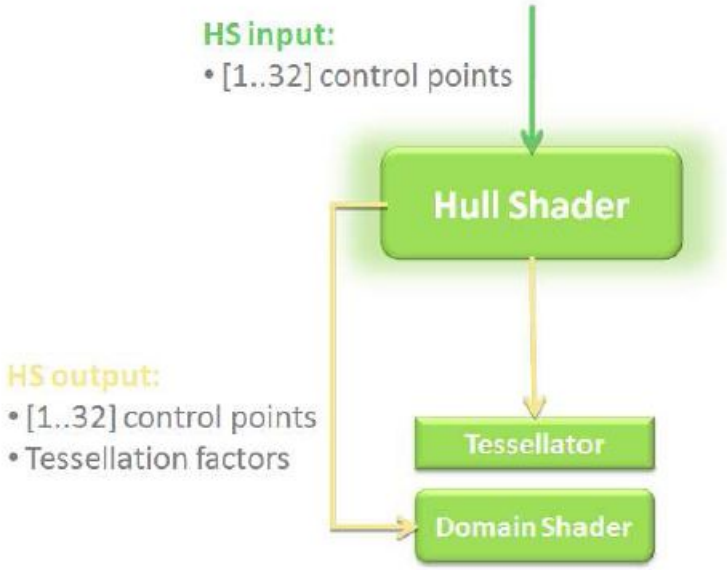

Figura 2.6: O Hull Shader [Ni et al., 2009]

2.2.4 Tessellator

O Tessellator não é uma parte programável do pipeline, ele é apenas configurável. O papel dele é gerar os vértices de acordo com os fatores de tecelagem passados pelo Hull Shader. De acordo com o domínio selecionado (triângulos, quads ou linha) ele cria os vértices e passa para o Domain Shader coordenadas paramétricas UV/UVW que são normalizadas no espaço do domínio. Com essas coordenadas o Domain Shader sabe exatamente onde foram criados os vértices e pode deslocá-los para onde for necessário.

Os fatores de tecelagem para as arestas e para o interior do domínio variam no intervalo [1..64]. O Tessellator suporta os métodos de tecelagem inteiro e fracionário. O método inteiro, como o nome sugere, cria vértices somente com fatores de tecelagem dos números inteiros no intervalo [1..64]. Este tipo de criação dos vértices pode resultar em *popping* das malhas, ou seja, as malhas darem a impressão visual que estão claramente sendo modificadas em tempo de execução. Para resolver esta questão, o Tessellator também suporta o método fracionário, onde os vértices são criados de maneira contínua com uma transição visual (geomorphing [Hoppe, 1996]). Desta maneira o efeito de popping fica bem reduzido.

Outra característica importante do Tessellator é a possibilidade de atribuir valores distintos para cada aresta e para o interior do domínio. Isto garante a flexibilidade de ter duas primitivas vizinhas com fatores de tecelagem diferentes sem discontinuidades na malha. A Figura 2.8 ilustra alguns padrões de

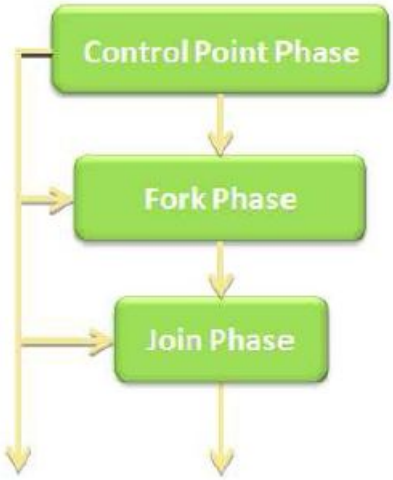

Figura 2.7: Fases do Hull Shader [Ni et al., 2009]

tecelagem do Tessellator tanto no domínio de triângulos quanto no de quads.

2.2.5 Domain Shader

É neste estágio onde ocorre a avaliação do domínio tecelado. O Domain Shader pode ser visto como um Vertex Shader após a tecelagem. Cada invocação deste estágio corresponde a um vértice gerado pelo Tessellator. O Tessellator passa as coordenadas UV/UVW em espaço normalizado no intervalo [0..1] e é papel do Domain Shader posicionar os vértices gerados no mundo. Vale lembrar que o que o Tessellator faz é unicamente subdividir um domínio (triângulo ou quad) para cada patch que é enviado ao pipeline, cabe ao programador usar as coordenadas UV/UVW e deslocar os vértices criados para se adequar à gaiola de controle de um modelo por exemplo.

Caso o Geometry Shader não esteja habilitado, é papel do Domain Shader colocar os vértices em espaço de tela para a rasterização. A Figura 2.9 mostra um resumo do fluxograma das novas partes do pipeline gráfico.

2.2.6 Geometry Shader e StreamOutput

O Geometry Shader continua com as mesmas características anteriores, porém, seu uso para tecelagem, que era lento, fica agora ainda mais sem propósito. Contudo, o Geometry Shader ainda é útil para algoritmos que requerem extração ou inserção de informação por triângulo. O Tessellator é útil quando se pretende fazer uma tecelagem global do domínio e não um acréscimo de um ou dois triângulos por primitiva, esse tipo de trabalho ficaria

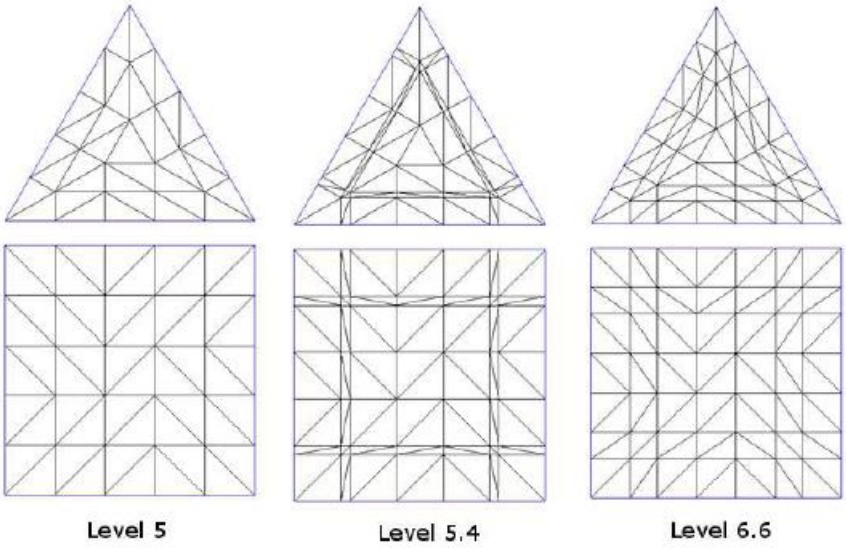

Figura 2.8: Exemplos de uso do Tessellator em diferentes domínios

para o Geometry Shader. A propriedade deste estágio de selecionar para qual Render Target o triângulo vai ser rasterizado ainda é util, por exemplo, para algoritmos que usam um cube map.

Uma série de algoritmos ainda necessita do geometry shader para serem executados como: simplificações de malha na GPU [DeCoro and Tatarchuk, 2007], extração de iso-superfícies na GPU [Tatarchuk et al., 2007], controle de sistema de partículas na GPU [Drone, 2007], entre outros. Portanto, o Tessellator não é simplismente um Geometry Shader com mais desempenho, são estágios do pipeline diferentes e que se complementam.

O StreamOutput é outra funcionalidade interessante do pipeline que pode vir a ser ainda mais usado com o Tessellator. Com ele o usuário é capaz de mandar para a CPU as geometrias geradas na GPU, por exemplo, o programador pode usar o Geometry Shader para visualizar uma iso-superfície e querer enviá-la para a CPU para ser salva posteriormente num arquivo de malha, possibilitando um designer alterar a iso-superfície em um editor 3D.

2.2.7 Rasterizer, Pixel Shader e Output Merger

Depois que os triângulos passam pelo estágio de geometria do pipeline (Tessellator/Geometry Shader), eles são enviados para o rasterizador que gera fragmentos para cada pixel. Fragmentos são candidatos a pixels, ou seja, muitos fragmentos podem cair no mesmo pixel e um teste com o buffer de profundidade é necessário para saber quais fragmentos serão descartados.

Uma característica interessante e útil do novo pipeline é a possibilidade

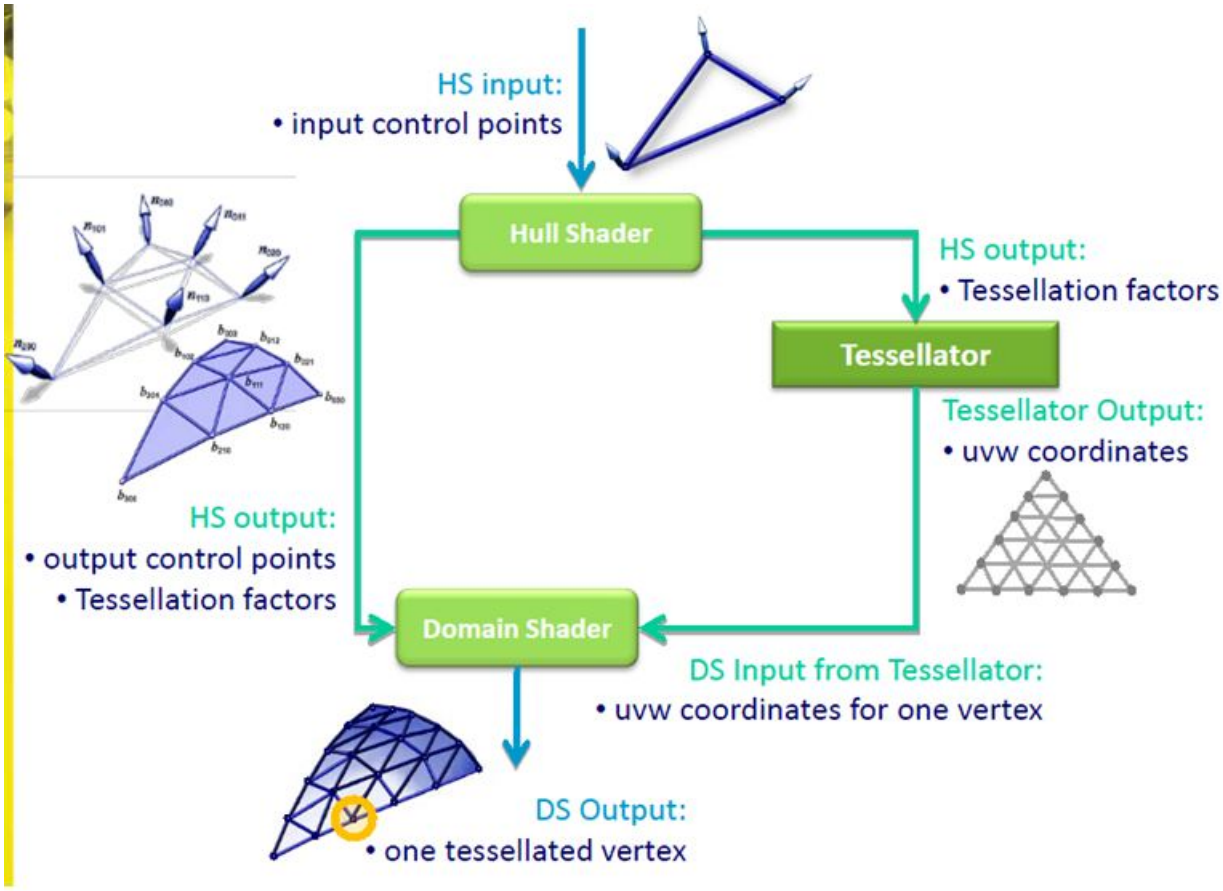

Figura 2.9: Fluxograma da nova parte do pipeline [Tariq, 2009]

de não só ler, mas, escrever em estruturas no Pixel Shader. Isto permite o uso de listas encadeadas para saber quais fragmentos fazem parte de um determinado pixel em tempo de execução. Esta propriedade possibilita a solução de um problema antigo na computação gráfica: ordenação de malhas transparentes [Yang et al., 2010].

2.3 Performance do Tessellator

Para avaliar a permonce do tessellator, criaremos uma aplicação que uma aplicação simples usando o Tessellator, que permita medir a performance da troca de transferência entre CPU e GPU por operações aritméticas na GPU.

2.3.1 Criando superfícies paramétricas na GPU

Esse algoritmo segue o trabalho de de Toledo e Levy [de Toledo and Levy, 2004], [de Toledo and Lévy, 2008a] que estende os tipos de primitivas do pipeline gráfico criando superfícies paramétricas na GPU utilizando ray-casting. A diferença é que a presente abordagem requer menos operações

aritméticas na GPU do que a abordagem do ray-casting e é muito mais flexível, podendo criar superfícies paramétricas de qualquer ordem. No trabalho de de Toledo e Levy é proposta a criação de superfícies até quarta ordem.

Configurando o Input Assembler

Queremos criar toda a superfície na GPU apenas com a equação paramétrica. Um domínio de quad será tecelado pelo Tessellator e o Domain Shader deslocará os vértices de acordo com a equação paramétrica desejada. Vamos passar para o Input Assembler um patch com apenas 1 vértice, apenas para habilitar o pipeline e fazer uma chamada de renderização (*drawcall*). O código 2.1 mostra a chamada do DirectX11 para configurar o Input Assembler de modo a receber apenas um vértice por primitiva.

Código 2.1: Configurando o Input Assembler

```
1
2 pd3dImmediateContext->IASetPrimitiveTopology(
3 D3D11_PRIMITIVE_TOPOLOGY::
4 D3D11_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST);
```

Vertex Shader

O Vertex Shader apenas passará a informação adiante para o Hull Shader. O código 2.2 mostra o Vertex Shader que repassa a informação e as estruturas de input e output do Vertex Shader. No mesmo código também são mostradas as variáveis constantes por frame: matriz de view e projection e o fator de tecelagem que vem da CPU e será passado para o Tessellator.

Código 2.2: Vertex Shader passando a informação adiante

```
1
2 cbuffer cbPerFrame : register( b0 )
3 {
4     matrix g_mViewProjection;
5     float  g_fTessellationFactor;
6 };
7 struct VSIn
8 {
9     float3 vPosition        : POSITION;
10 };
11 struct VSOut
12 {
13     float3 vPosition        : POSITION;
```

```
14 };
15 VS_CONTROL_POINT_OUTPUT BezierVS( VSIn Input )
16 {
17     VSOut Output;
18     Output.vPosition = Input.vPosition;
19     return Output;
20 }
```

Hull Shader

O Hull Shader precisa passar para o Tessellator o quanto o domínio deverá ser subdividido. Esse parâmetro vem da CPU (*g_fTessellationFactor*). Para um domínio de um quad existem 6 valores, 4 para as arestas e dois para o interior. A parte constante do Hull Shader é responsável pela passagem desses parâmetros para o Tessellator, enquanto a parte principal do Hull Shader apenas repassará a informação. O código 2.3 mostra a parte constante e principal do Hull Shader. As estruturas de entrada e saída também são mostradas.

Código 2.3: Hull Shader com sua parte constante e principal e também suas estruturas de entrada/saída.

```
1
2  struct HS_CONSTANT_DATA_OUTPUT
3  {
4      float Edges[4]          : SV_TessFactor;
5      float Inside[2]         : SV_InsideTessFactor;
6  };
7
8  struct HS_OUTPUT
9  {
10     float3 vPosition        : POSITION0;
11 };
12
13 HS_CONSTANT_DATA_OUTPUT ConstantHS( InputPatch<VSOut, 1> ip,
14                                      uint i : SV_PrimitiveID )
15 {
16     HS_CONSTANT_DATA_OUTPUT Output;
17
18         Output.Edges[0] = g_fTessellationFactor;
19         Output.Edges[1] = g_fTessellationFactor;
20         Output.Edges[2] = g_fTessellationFactor;
```

```
21          Output.Edges[3] = g_fTessellationFactor;
22          Output.Inside[0] = g_fTessellationFactor;
23          Output.Inside[1] = g_fTessellationFactor;
24
25      return Output;
26  }
27
28  [domain("quad")]
29  [partitioning("fractional_odd")]
30  [outputtopology("triangle_cw")]
31  [outputcontrolpoints(1)]
32  [patchconstantfunc("ConstantHS")]
33  HS_OUTPUT HS( InputPatch<VS_CONTROL_POINT_OUTPUT, 1> p,
34                uint i : SV_OutputControlPointID,
35                uint PatchID : SV_PrimitiveID )
36  {
37      HS_OUTPUT Output;
38      Output.vPosition = p[i].vPosition;
39      return Output;
40  }
```

Domain Shader

O Domain Shader será o responsável por deslocar os vértices gerados pelo Tessellator de acordo com a equação paramétrica desejada. Neste exemplo disponibilizamos várias equações diferentes que são escolhidas através de um #DEFINE passado pela CPU. As equações 2-1, 2-2, 2-3 e 2-4 listam as superfícies paramétricas apresentadas neste exemplo.

$$Esfera: \quad (x,y,z) = (rsin\varphi cos\theta, rsin\varphi sin\theta, rcos\varphi), \quad \theta \in [0,2\pi] \quad e \quad \varphi \in [0,\pi] \tag{2-1}$$

$$Cone: \quad (x,y,z) = (\frac{h-u}{h}rcos\theta, \frac{h-u}{h}rsin\theta, u), \quad u \in [0,h] \quad e \quad \theta \in [0,2\pi] \tag{2-2}$$

$$Torus: \quad (x,y,z) = ((M+Ncos\theta)*cos\varphi, (M+Ncos\theta)sin\varphi, N*sin\theta), \quad \theta,\varphi \in [0,2\pi] \tag{2-3}$$

$$SteinersRoman: \quad (x,y,z) = (\frac{a^2(cosv)^2 sin2u}{2}, \frac{a^2 sinusinv}{2}, \frac{a^2 cosu*sin2v}{2}), \quad u,v \in [0,pi] \tag{2-4}$$

O código 2.4 lista o Domain Shader usado. Uma cor arbitrária é atribuida a cada vértice para poder visualizar melhor as superfícies.

Código 2.4: Domain Shader com algumas opções de superfícies paramétricas

```
1
2  [domain("quad")]
3  DS_OUTPUT DS( HS_CONSTANT_DATA_OUTPUT input,
4                        float2 UV : SV_DomainLocation,
5                        const OutputPatch<HS_OUTPUT,
6                                OUTPUT_PATCH_SIZE> inputPatch )
7  {
8
9      DS_OUTPUT Output;
10
11     float3 position = float3(0.0,0.0,0.0);
12
13     /****** Sphere ******/
14 #if defined(SPHERE)
15     float pi2 = 6.28318530;
16     float pi = pi2/2;
17     float R = 1.0;
18     float fi = pi*UV.x;
19     float theta = pi2*UV.y;
20     float sinFi,cosFi,sinTheta,cosTheta;
21     sincos( fi, sinFi, cosFi);
22     sincos( theta, sinTheta,cosTheta);
23     position = float3(R*sinFi*cosTheta, R*sinFi*sinTheta,
24                                     R*cosFi);
25     Output.vColor = float3(normalize(position) + 0.6);
26     /******* End Sphere *******/
27
28     /****** Cone ******/
29 #elif defined(CONE)
30     float pi2 = 6.28318530;
31     float pi = pi2/2;
32     float R = 0.5;
33     float H = 1.5;
34     float S = UV.y;
35     float T = UV.x;
36     float theta = pi2*T;
37     float sinTheta,cosTheta;
38     sincos( theta, sinTheta,cosTheta);
```

```
39      position = float3(((H-S*H)/H)*R*cosTheta,
40                        ((H-S*H)/H)*R*sinTheta, S*H);
41      Output.vColor = float3(normalize(position) + 0.4 );
42      /****** End Cone ******/
43
44  #elif defined(TORUS)
45      /****** Torus ******/
46      float pi2 = 6.28318530;
47      float M = 1;
48      float N = 0.5;
49      float cosS, sinS;
50      sincos( pi2 * UV.x, sinS, cosS);
51      float cosT, sinT;
52      sincos( pi2 * UV.y, sinT, cosT);
53      position = float3((M + N * cosT) * cosS,
54                        (M + N * cosT) * sinS, N * sinT);
55      Output.vColor = float3(normalize(position) + 0.4);
56      /****** End Torus ******/
57
58
59  #elif defined(STEINERS_ROMAN)
60      /****** Steiners Roman ******/
61      float pi2 = 6.28318530;
62      float pi = pi2/2;
63      float u = UV.x*pi;
64      float v = UV.y*pi;
65      float sinu,cosu,sinv,cosv,sin2v,sin2u;
66      sincos( u, sinu, cosu );
67      sincos( v, sinv, cosv );
68      sin2v = sin( 2*v );
69      sin2u = sin( 2*u );
70      float r = 1;
71      float a = 1;
72      position = float3(a*a*cosv*cosv*sin2u/2,
73                        a*a*sinu*sin2v/2,a*a*cosu*sin2v/2);
74      Output.vColor = float3(normalize(position) + 0.4f);
75      /****** End Steiners Roman ******/
76
77  #endif
78
79      Output.vPosition = mul( float4(position,1),
```

```
80                                  g_mViewProjection );
81          Output.vWorldPos = Output.vPosition;
82
83      return Output;
84  }
```

Pixel Shader

O Pixel Shader apenas retorna a cor de cada vértice como mostra o código 2.5.

Código 2.5: Pixel Shader simples que retorna a cor de cada vértice

```
1
2  float4 PS( DS_OUTPUT Input ) : SV_TARGET
3  {
4      return float4(Input.vColor, 1);
5  }
```

A Figura 2.10 mostra as superfícies paramétricas geradas com este programa.

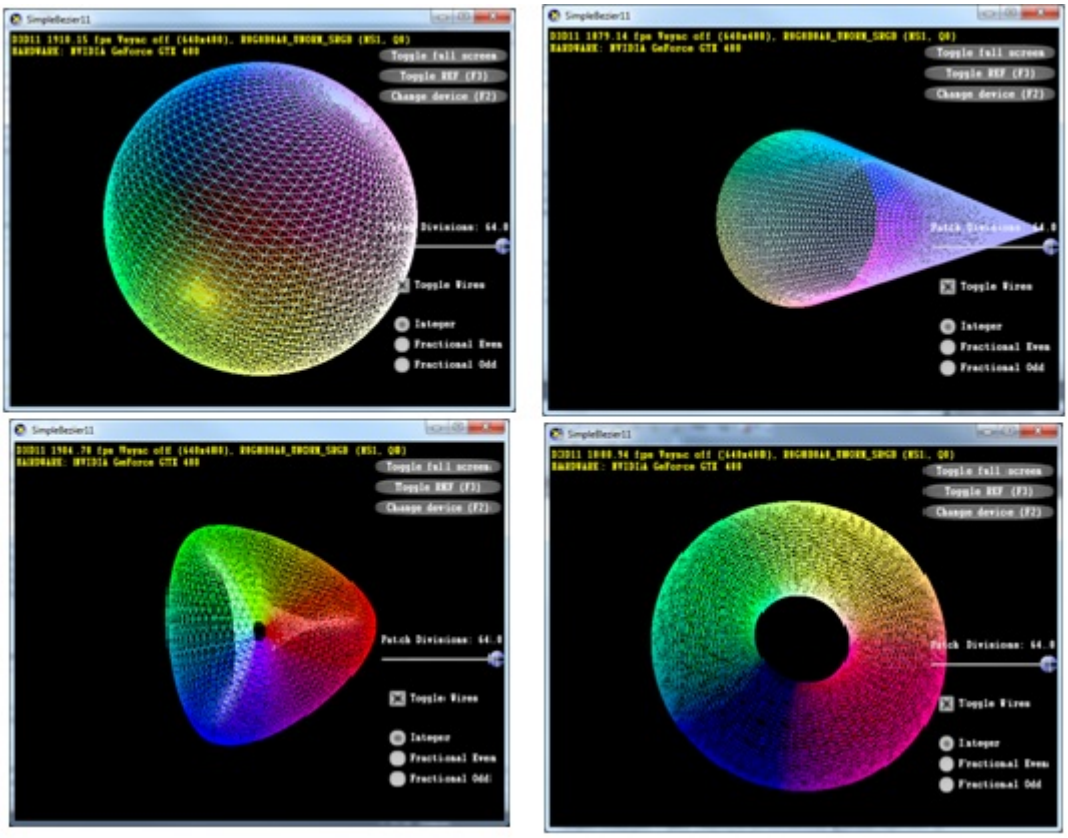

Figura 2.10: Superfícies paramétricas geradas em GPU

Quantidade de Torus	Triângulos(em milhões)	FPS Tessellator	FPS sem Tessellator
20000	163.84	11	1
10000	81.92	22	1
5000	40.96	45	3
2500	20.48	90	10
1000	8.192	223	89
500	4.096	440	141
100	0.8192	980	490
10	0.08192	1030	950
1	0.008192	1050	1020

Tabela 2.1: Ganho de performance do Tessellator

2.3.2 Avaliando a performance

Visto que nós passamos apenas 1 vértice da CPU para a GPU na criação de cada primitiva paramétrica, esta aplicação se mostra bastante apropriada para medir o quanto a troca de transferência de dados da CPU para a GPU por operações na GPU pode ser vantajosa em termos de desempenho.

O teste foi realizado em uma máquina com Processador Core i7 920 2.66Ghz, 6GB RAM e placa de vídeo NVIDIA Geforce 480GTX. O teste consistiu na criação de diversos torus em GPU como descrito na seção anterior. Cada torus possui 8192 triângulos. A mesma quantidade de torus foi criada na CPU e passada para a GPU e os FPS foram medidos nas duas abordagens. Os torus da CPU foram passados em um único vertex buffer para evitar que muitas chamadas de renderização (*draw calls*) virassem o gargalo da aplicação.

A Tabela 2.1 e a Figura 2.11 mostram o ganho de performance em número de FPS e em porcentagem do uso do Tessellator em contraste com o mesmo número de modelos passados pela CPU.

A Tabela 2.2 mostra a economia de memória (em MB) da abordagem sem geração de vértices em GPU em comparação com o uso do Tessellator. Para os cálculos de uso da memória foi considerado que cada vértice contém apenas a informação de posição (12 bytes).

2.4 Discussão

Analisando os gráficos e tabelas mostrados na seção anterior fica evidente o grande salto de performance que o uso do Tessellator proporciona para aplicações que necessitam de uma grande quantidade de vértices. Pela Tabela 2.1 e pela Figura 2.11 é notável a brusca queda de desempenho da aplicação

Uso de memória sem o Tessellator	Uso de memória com o Tessellator
5898.24	0.24
2949.12	0.12
1479.56	0.06
737.28	0.03
294.91	0.012
147.45	0.006
29.49	0.0012
2.94	0.00012
0.29	0.000012

Tabela 2.2: Tabela mostrando a economia de memória com o uso do Tessellator (em Megabytes)

feita sem o uso do Tessellator após a faixa de 20 milhões de triângulos. Enquanto isso, com o uso do Tessellator, fomos capazes de manter taxas interativas de FPS usando até 163 milhões de triângulos por frame. Para taxas interativas de FPS (≥ 10), o máximo que a abordagem sem o uso do Tessellator conseguiu alcançar foi a renderização de 2500 torus por frame. O tessellator foi capaz de aumentar esse número em 8 vezes (20000 torus/frame).

O benefício deste novo pipeline não se limita em apenas desafogar o barramento de transferência. Antes de serem transferidos para GPU os vértices criados na CPU normalmente são alocados na memória principal. A Tabela 2.2 mostra um gasto de memória de até 5.8GB ao usar a abordagem sem o uso do Tessellator. Evitar o consumo desta quantidade de memória em uma aplicação em tempo real implica em um ganho considerável.

Com esse novo paradigma, no decorrer deste trabalho, vamos analisar aplicações antigas que se adequam bem ao novo pipeline e também propor novos algoritmos que podem fazer uso da geração massiva de vértices em GPU.

No próximo capítulo abordaremos dois algoritmos de refinamento de malha. Avaliaremos seus desempenhos com o uso do Tessellator e faremos uma comparação qualitativa e quantitativa de ambos.

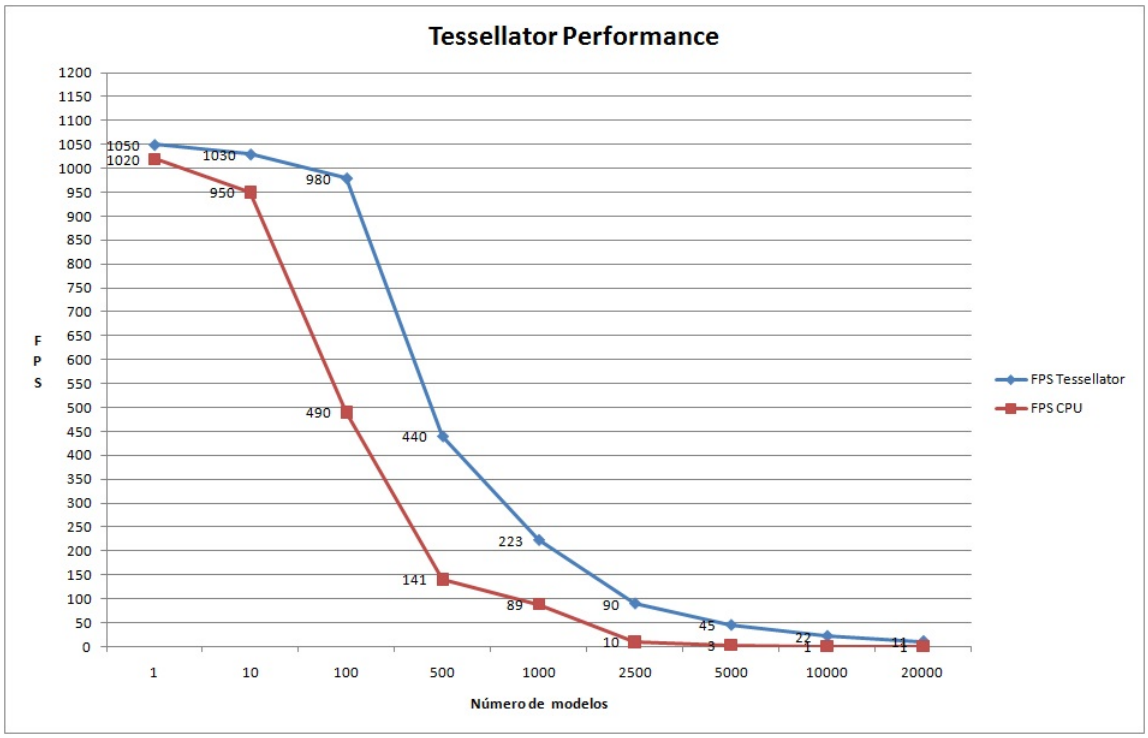

Figura 2.11: Gráfico representando o ganho de performance do Tessellator

3
PN-Triangles vs Phong Tessellation

3.1
Introdução

A área de subdivisão de superfícies é um campo que recebe bastante atenção desde 1978 quando E. Catmull e J. Clark propuseram um dos primeiros algoritmos relacionados [Catmull and Clark, 1978]. Basicamente, cada superfície a ser subdividida possui uma malha original, chamada malhada de controle ou gaiola de controle. Após a superfície ser subdividida, os novos vértices gerados são movidos de acordo com uma série de regras que variam dependendo do algoritmo utilizado. A superfície que será gerada (chamada de superfície limite) tem a mesma topologia da gaiola de controle.

Apesar de algoritmos de subdivisão de superfícies serem o padrão de representação geométrica em renderização off-line, até o início da década de 90, esses algoritmos não tinham muita atenção da indústria de renderização em tempo real, especialmente a indústria de jogos. Isto se deve ao fato dos computadores da época serem fracos em poder de processamento e os algoritmos propostos possuírem um custo computacional elevado. Porém, no final da década de 90, com o lançamento das placas de vídeo 3D, os consumidores começaram a ter configurações de seus computadores pessoais cada vez mais discrepantes. Com isso, a indústria de jogos se viu com a necessidade de reduzir o trabalho no pipeline de arte. Um jogo, em máquinas poderosas, poderia ter modelos com milhares de polígonos. Já em máquinas mais limitadas, o número de polígonos dos modelos deveria ser baixo. A possibilidade de criar modelos com poucos polígonos (low-poly), animá-los e ter, ao final da construção, um modelo animado com qualquer nível de detalhe seria um grande alívio para os designers.

Embora os softwares de modelagem da época já permitissem o uso de algoritmos de subdivisão de superfícies, era inviável colocar em um jogo dezenas de modelos com níveis de detalhes diferentes para atender a maior parte de configurações de computadores possível. O tamanho da mídia era limitado, assim como o espaço em disco dos usuários. A solução ideal seria a geração desses

níveis de detalhe em tempo de execução. As placas aceleradoras 3D começaram a ter uma grande capacidade de processamento aliviando bastante o uso da CPU para outras tarefas. Porém, as placas 3D têm a sua arquitetura voltada para o paralelismo, ou seja, toda a manipulação de vértices é feita em paralelo. Esse fator ia de encontro aos algoritmos de subdivisão de superfícies existentes, pois todos eram baseados em recursão e usavam informações das adjacências para posicionar os vértices criados. Em 2001 Vlachos [Vlachos et al., 2001] propôs um algoritmo de subdivisão de superfície puramente local, chamado de PN-Triangles. A idéia era disponibilizar esse algoritmo implementado no hardware para que as produtoras de jogos pudessem simplesmente passar um modelo que seria refinado pela placa gráfica. Esta solução foi implementada, porém, como era específica de um vendedor de hardware ela acabou não sendo utilizada pela indústria que desejava atingir o maior número de consumidores possíveis.

Em 2008 Boubekeur e Alexa [Boubekeur and Alexa, 2008] propuseram outro algoritmo de subdivisão de superfície, denominado Phong Tessellation, que tornou-se outro bom candidato para implementação em hardware, já que utiliza informação unicamente local.

Apesar dos grandes avanços em renderização em tempo real, o problema de modelos low-poly persiste até hoje. Um exemplo disso é o jogo MAFIA II(Figura 3.1) lançado em agosto de 2010. Uma das propostas da tecelagem em hardware é a solução para este problema. O objetivo deste capítulo é avaliar o desempenho e a qualidade visual dos algoritmos PN-Triangles e Phong Tessellation. Ambos são puramente locais, interpolativos e precisam de poucas operações aritméticas para a avaliação da superfície, sendo assim, apropriados para implementação no novo pipeline de tecelagem e utilização em motores 3D de tempo real. Além disso, será avaliado o impacto na economia da largura de banda na implementação em GPU em comparação com a implementação em CPU.

3.2 Trabalhos Relacionados

Várias técnicas foram desenvolvidas para geração de subdivisão de superfícies, porém muitas delas ainda se mostram pouco eficazes para implementação nos motores de jogos 3D. Isto se deve ao fato da grande maioria usar informação da vizinhança para fazer a subdivisão. Implementar esses algoritmos em hardware requer um esforço maior, são necessários várias passadas e a avaliação da superfície é muito custosa. Boubekeur et al. [Boubekeur et al., 2005] estenderam a técnica PN-Triangles colocando três valores escalares em

Figura 3.1: Foto do jogo MAFIA II ([2KGames, 2010])

cada vértice, posssibilitando a criação de um mapa de displacement procedural que aumenta a qualidade de detalhes da geometria permitindo construir superfícies pontiagudas/afiadas. Porém, este método acrescenta um overhead no pipeline artístico, já que o designer deve setar 3 valores adicionais para cada vértice da geometria. Com a falta de feedback visual dos programas de modelagem para suportarem essa técnica, ela acabou sendo pouco usada.

As soluções que usam informação da vizinhança são mais diversas. Catmull e Clark [Catmull and Clark, 1978] usam uma B-spline bi-cúbica uniforme em seu esquema de subdivisão. Doo e Sabin [Doo, 1978; Doo and Sabin, 1978] baseiam seu método em B-splines bi-quadráticas. Loop [Loop, 1987] propõe um algoritmo que gera superfícies limite com continuidade C2 em todo lugar exceto em vértices extraordinários que apresentam continuidade C1. Kobbelt [Kobbelt, 2000] propôs uma idéia que oferece um refinamento adaptativo natural quando preciso. Zorin et al. [Zorin et al., 1996] propuseram um esquema para geração de superfícies suaves de malhas de triângulos irregulares. Ni et al. [Yeo et al., 2009] apresentaram um método que imita o formato das superfícies de Catmull-Clark usando splines bi-cúbicos e uma nova classe de patches chamada de c-patches. Boubekeur e Schlick [Boubekeur and Schlick, 2007] evitam a recursão em seu método, que é mais rápido, porém, geometricamente ele só garante superfícies com continuidade C0. Mais recentemente, Loop e Schaefer [Loop and Schaefer, 2008] usaram superfícies quárticas com campos de normais separados para aproximar as superfícies de Catmull-Clark. Este esquema proposto por Loop e Schaefer trabalha apenas

com quads. Mais tarde, Loop et al. [Loop et al., 2009] propuseram um esquema que usa patches gregorianos para aproximar a subdivisão de superfícies também com triângulos.

Todas essas técnicas expostas usam o mesmo princípio básico: cada polígono é substituído por um patch polinomial que é avaliado em sequência. A única exceção é o Phong Tessellation que não cria um patch explicitamente.

3.3 Continuidade de superfícies

Uma característica de qualquer esquema de subdivisão é a sua continuidade. Esquemas se referem como tendo continuidade C^n onde n define quantas derivadas são contínuas. Para exemplificar, vamos analisar a continuidade de curvas ao invés de superfícies pois é mais fácil visualizar e a correspondência é a mesma. A Figura 3.2 mostram duas curvas que não são contínuas.

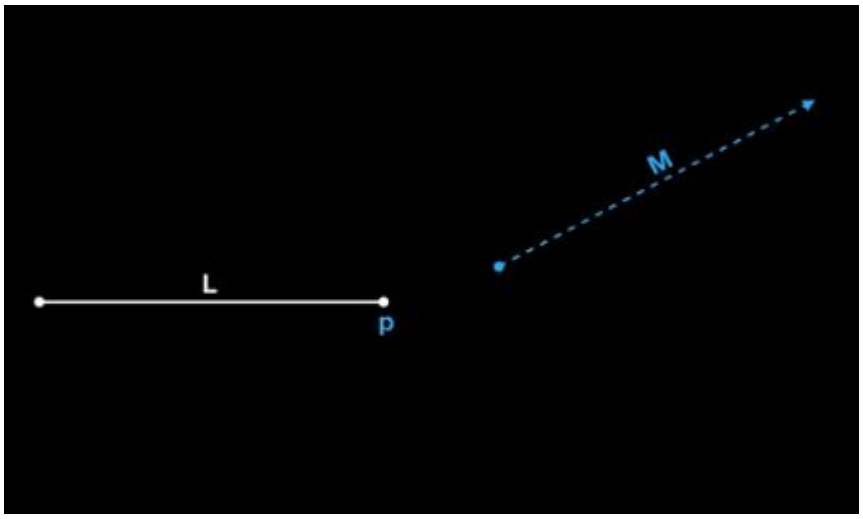

Figura 3.2: Duas curvas que não são contínuas

Na Figura 3.3 a curva L têm continuidade C^0 com a curva M no ponto **p**. As curvas simplesmente tocam no ponto **p** e depois seguem em qualquer direção.

Na Figura 3.4 todas as curvas têm continuidade C^1 no ponto **p**. Elas tocam no ponto **p** e se movem na mesma direção. Isso implica que a primeira derivada de todas as curvas tem o mesmo valor no ponto **p**.

Na Figura 3.5 as curvas tocam, vão na mesma direção e têm o mesmo raio no ponto onde elas se encontram. Isso condiz com a primeira e a segunda derivada serem iguais no ponto **p**. Portanto as curvas têm continuidade C^2 no ponto em questão.

Na Figura 3.6 a primeira, segunda e terceira derivada são iguais no ponto **p**. As curvas R e C tem continuidade C^3 no ponto **p**. Elas tocam, vão na mesma

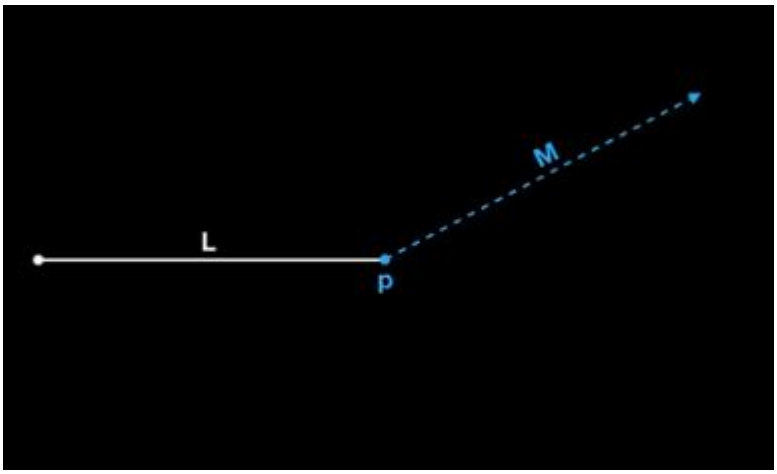

Figura 3.3: Continuidade C^0

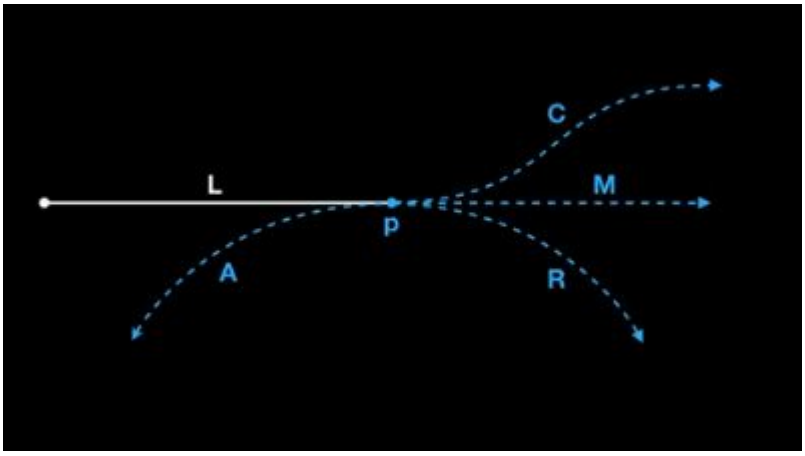

Figura 3.4: Continuidade C^1

direção, tem o mesmo raio no ponto de contato e o raio está acelerando com a mesma taxa no ponto de contato.

Já na Figura 3.7 todas as quatro primeiras derivadas são iguais. Isso implica que as curvas tocam, vão na mesma direção, tem o mesmo raio no ponto de contato, o raio está acelerando com a mesma taxa no espaço 3D (aceleração torsoidal).

Nos esquemas que usam informação da vizinhança a maioria tem continuidade de superfície C^1 em todos os pontos. Alguns tem continuidade C^2 em alguns lugares, mas todos têm áreas que o melhor que eles podem assegurar é a continuidade C^1. Já no Phong Tessellation e PN-Triangles ambos só garantem continuidade C^0, isto é, só é garantido que os pontos das superfícies se tocam e formam uma superfície limite contínua. Na teoria isso pode proporcionar uma superfície com arestas pontiagudas e superfícies se juntando, porém caminhando em direções diferentes. Contudo, na prática, o resultado de suavidade visual é considerado bom.

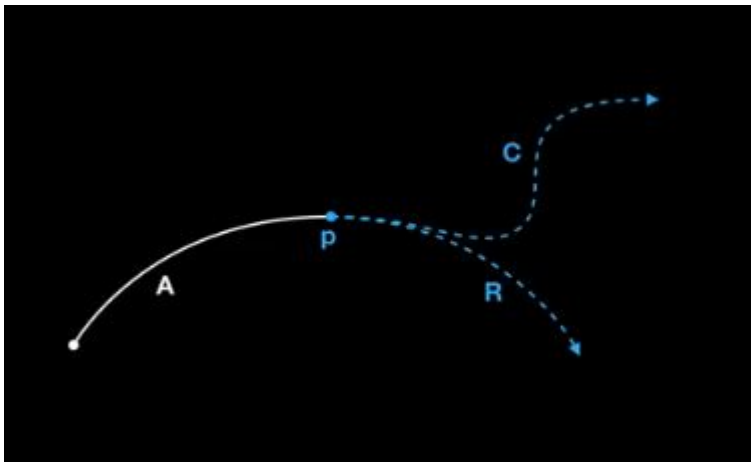

Figura 3.5: Continuidade C^2

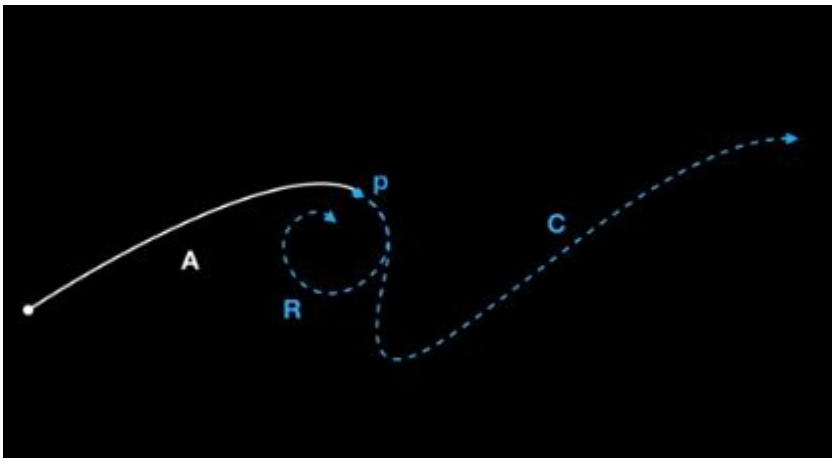

Figura 3.6: Continuidade C^3

3.4 Métodos Aproximativos vs Interpolativos

Os métodos de subdivisão também são classificados como interpolativos(e.g., [Kobbelt, 2000; Zorin et al., 1996]) e aproximativos(e.g., [Catmull and Clark, 1978; Loop, 1987]). Se o esquema é aproximativo, após cada subdivisão os vértices gerados e os vértices existentes da malha de controle se movem mais para perto da superfície limite. Isso quer dizer que os vértices da malha de controle não permanecem na superfície. Isso implica tanto em uma vantagem quanto em uma desvantagem. A vantagem é que este tipo de esquema evita ondulações e deformidades na superfície limite. A desvantagem fica por conta de ser difícil prever só com a malha de controle qual será a superfície limite gerada. Já no esquema interpolativo, os vértices iniciais da malha de controle fazem parte da superfície limite. Ou seja, independente de quantos vértices forem gerados, o método interpolativo sempre garante que a superfície tocará a malha de controle. A Figura 3.8 mostra na esquerda um tetraedro sendo sub-

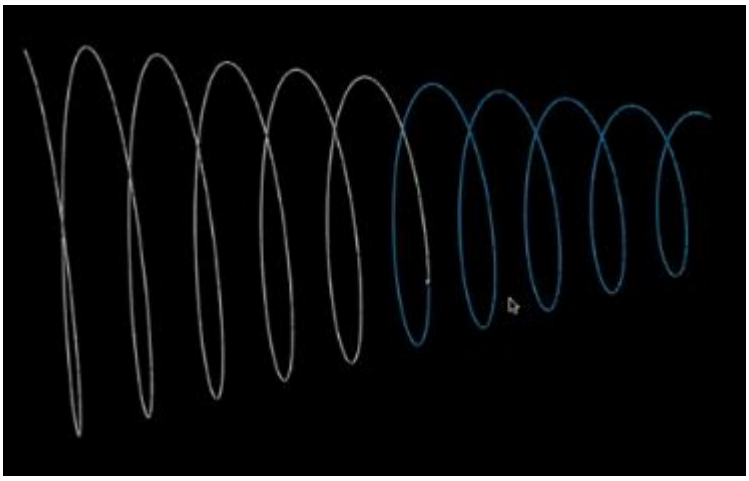

Figura 3.7: Continuidade C^4

dividido com um método aproximativo (Catmull-Clark [Catmull and Clark, 1978]). Na direita é mostrado o mesmo tetraedro usando um método interpolativo (Modified-Butterfly [Zorin et al., 1996]).

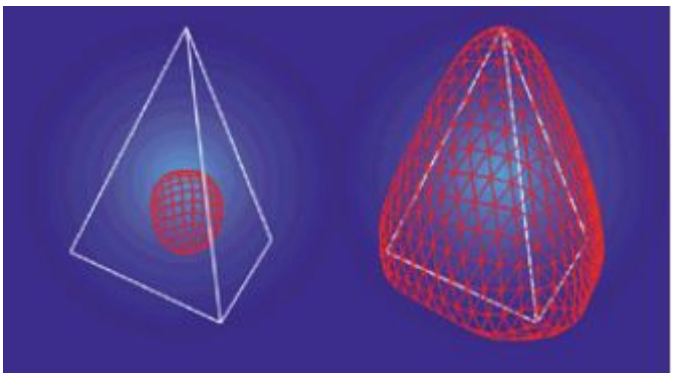

Figura 3.8: Interpolativo vs aproximativo[Sharp, 2000]

Ambos os métodos (Phong Tessellation e PN-Triangles) estudados nessa dissertação são interpolativos.

3.5 Avaliação da superfície

Para a superfície ser avaliada os algoritmos usam o conceito de "máscara". A máscara de um algoritmo indica quais vértices da vizinhança precisam ser levados em conta para posicionar o vértice em questão. Por exemplo, a Figura 3.9 mostra uma máscara hipotética onde os vértices da região branca devem ser levados em conta para o cálculo de posicionamento do vértice vermelho. Tanto o algoritmo PN-Triangles quanto o Phong Tessellation usam máscaras locais, ou seja, para posicionar um vértice que foi gerado dentro de um triângulo só são usadas as informações dos vértices deste triângulo.

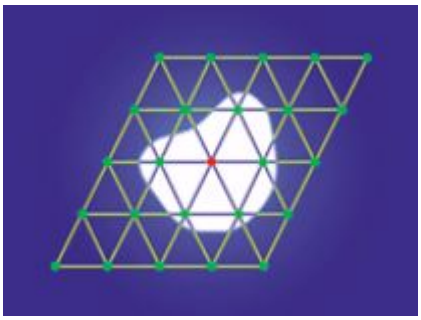

Figura 3.9: Máscara para um algoritmo hipotético de subdivisão [Sharp, 2000]

3.6 PN-Triangles

3.6.1 Patch de Bézier

A principal característica do algoritmo é a construção de um patch cúbico com informações locais de um triângulo. O patch b é definido da seguinte maneira:

$b : \Re^2 \rightarrow \Re^3$, para $w = 1 - u - v$, $u, v, w \geq 0$

$$\begin{aligned} b(u,v) &= \sum_{i+j+k=3} b_{ijk} \frac{3!}{i!j!k!} u^i v^j w^k, \\ &= b_{300}w^3 + b_{030}u^3 + b_{003}v^3 \\ &+ b_{210}3w^2u + b_{120}3wu^2 + b_{201}3w^2v \\ &+ b_{021}3u^2v + b_{102}3wv^2 + b_{012}3uv^2 \\ &+ b_{111}6wuv. \end{aligned} \tag{3-1}$$

As normais podem ser definidas de duas maneiras: uma simples interpolação linear ou uma função quadrática n avaliada do seguinte modo:

$n : \Re^2 \rightarrow \Re^3$, para $w = 1 - u - v$, $u, v, w \geq 0$

$$\begin{aligned} n(u,v) &= \sum_{i+j+k=2} n_{ijk} u^i v^j w^k, \\ &= n_{200}w^2 + n_{020}u^2 + n_{002}v^2 \\ &+ n_{110}wu + n_{011}uv + n_{101}wv. \end{aligned} \tag{3-2}$$

As figuras 3.10 e 3.11 mostram os pontos de controle b_{ijk} e n_{ijk} relativos a cada patch.

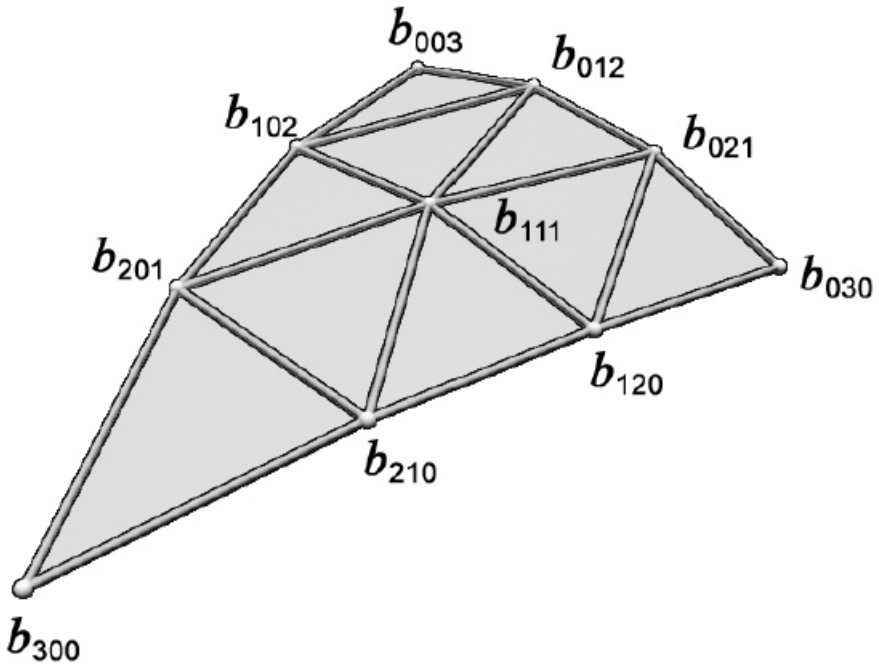

Figura 3.10: Pontos de controle do patch de geometria [Vlachos et al., 2001]

3.6.2 Escolhendo os pontos de controle da geometria

Dadas as posições $P_1, P_2, P_3 \in \Re^3$ e as normais $N_1, N_2, N_3 \in \Re^3$ de um triângulo, a escolha dos pontos de controle B_{ijk} é feita da seguinte forma:

1. Coloque os coeficientes b_{ijk} nas posições intermediárias $(iP_1 + jP_2 + kP_3)/3$.

2. Deixe cada vértice do triângulo em seu ponto de controle correspondente (e.g., $b_{300} = P_1, b_{030} = P_2, b_{003} = P_3$).

3. Para cada canto do triângulo projete os dois coeficientes mais próximos a este canto no plano tangente definido pela normal do canto.

4. Mova o coeficiente do centro da sua posição intermediária V para a média dos pontos $b_{012}, b_{102}, b_{120}, b_{210}, b_{201}, b_{021}$ e continue seu deslocamento na mesma direção por 1/2 da distância já deslocada.

Seguindo a descrição acima os pontos de controle são:

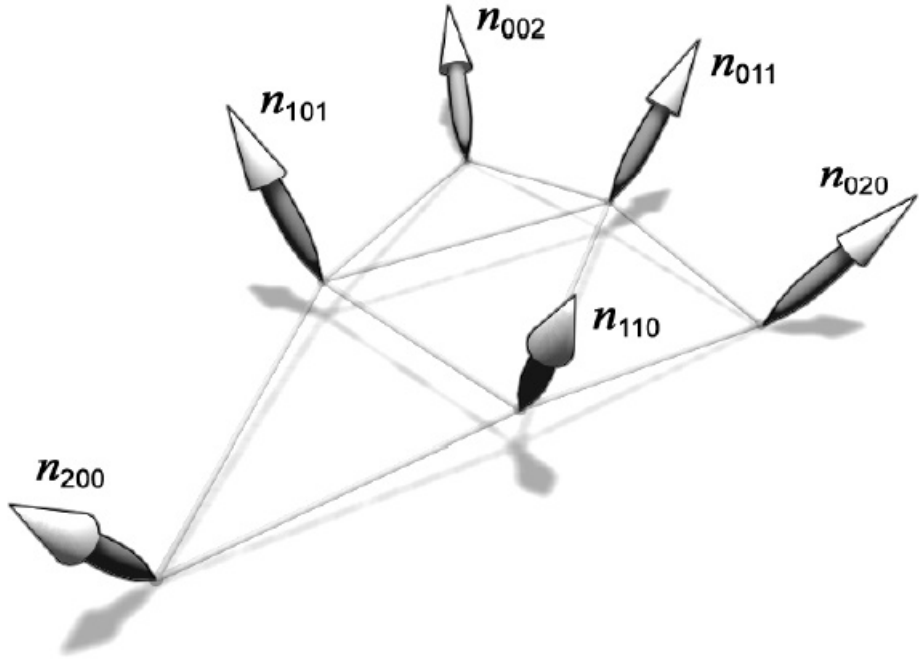

Figura 3.11: Pontos de controle do patch de normal [Vlachos et al., 2001]

$$b_{300} = P_1, \tag{3-3}$$

$$b_{030} = P_2, \tag{3-4}$$

$$b_{003} = P_3, \tag{3-5}$$

$$w_{ij} = (P_j - P_i) \cdot N_i \in \Re, \tag{3-6}$$

$$b_{210} = (2P_1 + P_2 - w_{12}N_1)/3, \tag{3-7}$$

$$b_{120} = (2P_2 + P_1 - w_{21}N_2)/3, \tag{3-8}$$

$$b_{021} = (2P_2 + P_3 - w_{23}N_2)/3, \tag{3-9}$$

$$b_{012} = (2P_3 + P_2 - w_{32}N_3)/3, \tag{3-10}$$

$$b_{102} = (2P_3 + P_1 - w_{31}N_3)/3, \tag{3-11}$$

$$b_{201} = (2P_1 + P_3 - w_{13}N_1)/3, \tag{3-12}$$

$$E = (b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201})/6, \tag{3-13}$$

$$V = (P_1 + P_2 + P_3)/3, \tag{3-14}$$

$$b_{111} = E + (E - V)/2. \tag{3-15}$$

3.6.3
Escolhendo os coeficientes das normais

As normais da geometria do PN-Triangles geralmente não variam continuamente de triângulo para triângulo. No algoritmo é sugerido uma interpolação linear ou uma variação quadrática. O problema da interpolação linear é que ela ignora inflexões como mostra a Figura 3.12.

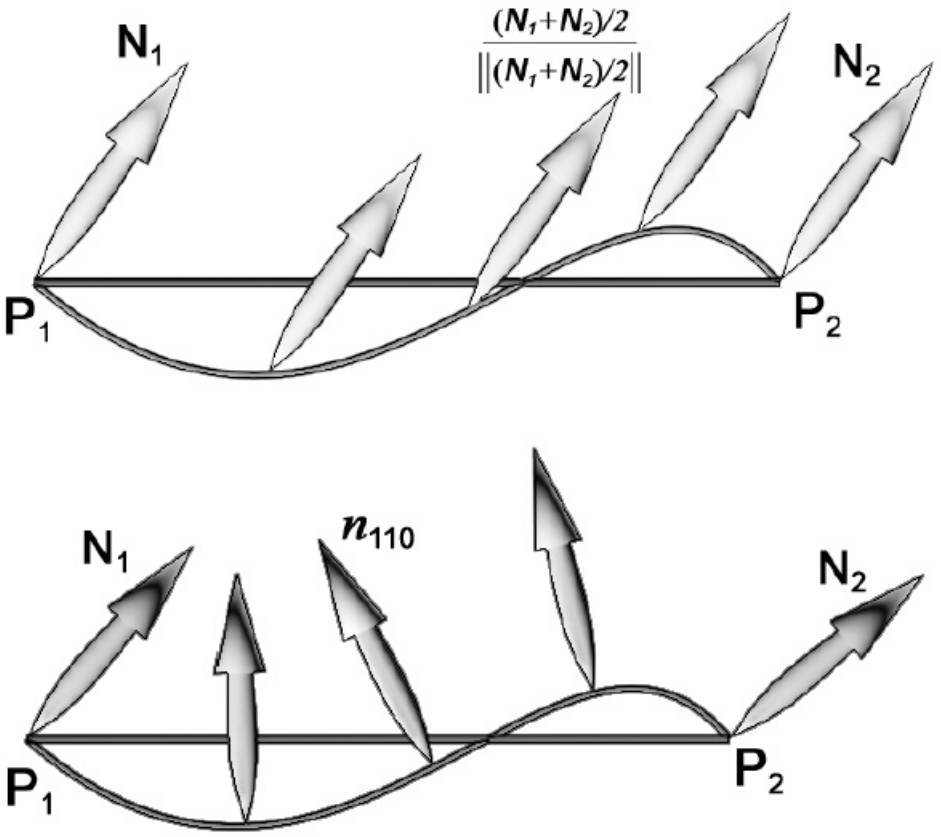

Figura 3.12: Interpolação linear das normais(acima) e variação quadrática(em baixo) [Vlachos et al., 2001]

Para capturar as inflexões, um coeficiente no meio de cada aresta é calculado para a avaliar a superfície n. A média das normais de cada vértice de uma aresta é calculada e refletida no plano perpendicular a aresta como mostra a Figura 3.13.

Seguindo a descrição acima os pontos de controle para as normais assumindo $\|N_i\| = 1$ são:

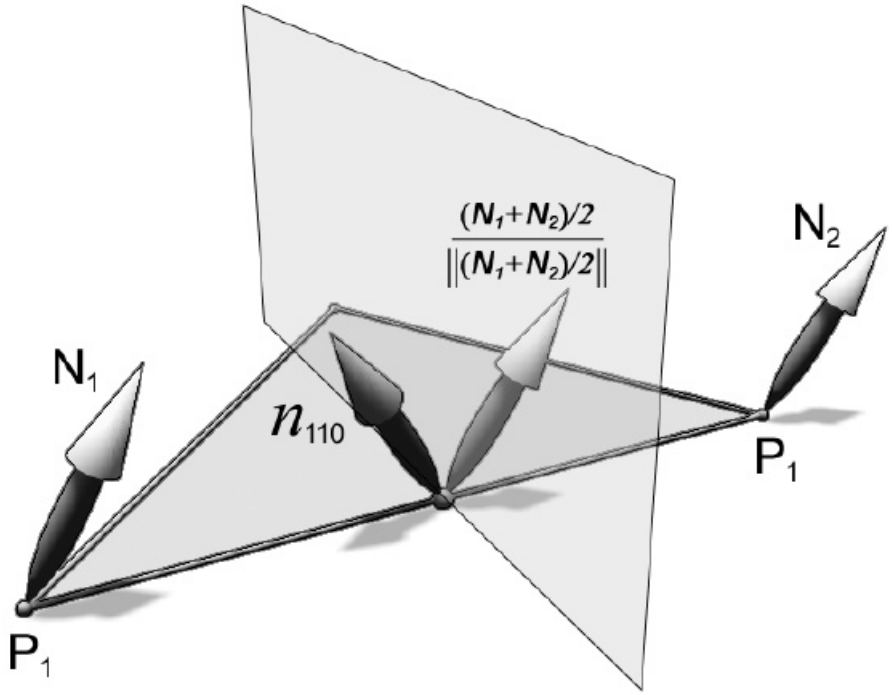

Figura 3.13: Reflexão da normal no meio da aresta pelo plano perpendicular a ela [Vlachos et al., 2001]

$$n_{200} = N_1, \tag{3-16}$$

$$n_{020} = N_2, \tag{3-17}$$

$$n_{002} = N_3, \tag{3-18}$$

$$v_{ij} = 2\frac{(P_j - P_i) \cdot (N_i + N_j}{(P_j + P_i) \cdot (P_j - P_i)} \in \Re, \tag{3-19}$$

$$n_{110} = h_{110} / \left\| h_{110} \right\|, h_{110} = N_1 + N_2 - v_{12}(P_2 - P_1), \tag{3-20}$$

$$n_{011} = h_{011} / \left\| h_{011} \right\|, h_{011} = N_2 + N_3 - v_{23}(P_3 - P_2), \tag{3-21}$$

$$n_{101} = h_{101} / \left\| h_{101} \right\|, h_{101} = N_3 + N_1 - v_{31}(P_1 - P_3). \tag{3-22}$$

A Figura 3.14 mostra a diferença entre as normais variando linearmente e quadraticamente.

3.6.4 Implementação

A implementação do PN-Triangles se encaixa bem no novo pipeline. Basicamente deveremos calcular os pontos de controle do campo de geometria e normal no Hull Shader e avaliar a superfície no Domain Shader.

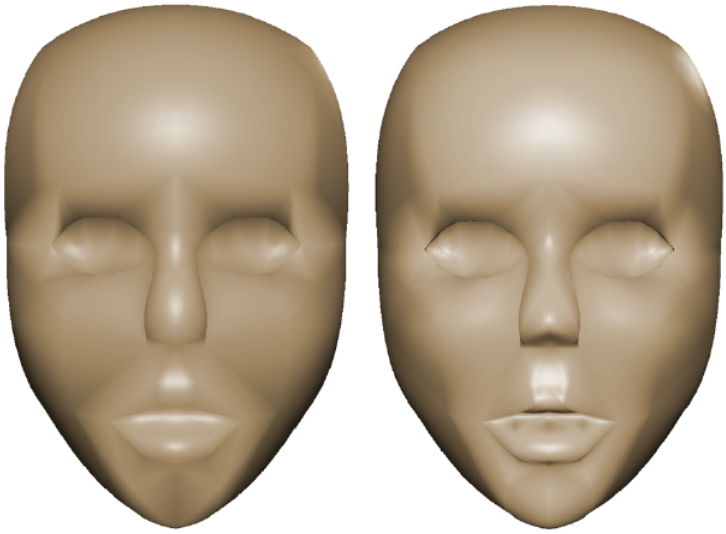

Figura 3.14: Normais variando linearmente (esquerda) e quadraticamente (direita) [Vlachos et al., 2001]

O Vertex Shader simplesmente repassa a informação para o Hull Shader, ele receberá os dados de cada vértice (Posição, Normal e Textura) e mandará para o Hull Shader. Abaixo temos o código do Vertex Shader e sua struct de entrada.

Código 3.1: Vertex Shader do PN-Triangles

```
1  struct VS_Input
2  {
3      float3 Position    : POSITION;
4      float3 Normal      : NORMAL;
5      float2 TexCoord    : TEXCOORD;
6  };
7  VS_Input VS_PassThrough( VS_Input input )
8  {
9      VS_Input output;
10     output.Position =input.Position;
11     output.Normal = input.Normal;
12     output.TexCoord =input.TexCoord;
13     return output;
14 }
```

A parte constante do Hull Shader receberá o output do Vertex Shader e como ela é executada por patch e existem registradores suficientes para passar as informações do algoritmo, ela passará essas informações neste output. Além disso ela deve setar os fatores de tecelagem do triângulo. Deixaremos estes

fatores de acordo com um parâmetro constante vindo da CPU.

Código 3.2: Hull Shader Constante do PN-Triangles

```
1  struct HS_Input
2  {
3      float3 Position    : POSITION;
4      float3 Normal      : NORMAL;
5      float2 TexCoord    : TEXCOORD;
6  };
7
8  struct HS_ConstantOutput
9  {
10     // Fatores de tecelagem
11     float fTessFactor[3]     : SV_TessFactor;
12     float fInsideTessFactor : SV_InsideTessFactor;
13
14     // Pontos de controle da geometria
15     float3 B210     : POSITION3;
16     float3 B120     : POSITION4;
17     float3 B021     : POSITION5;
18     float3 B012     : POSITION6;
19     float3 B102     : POSITION7;
20     float3 B201     : POSITION8;
21     float3 B111     : CENTER;
22
23     // Pontos de controle das normais
24     float3 N110     : NORMAL3;
25     float3 N011     : NORMAL4;
26     float3 N101     : NORMAL5;
27 };
28
29 HS_ConstantOutput HS_Constant(InputPatch<HS_Input, 3>input)
30 {
31     HS_ConstantOutput output = (HS_ConstantOutput)0;
32
33     //Todos fatores de tecelagem iguais
34       output.fTessFactor[0] = output.fTessFactor[1] =
35     output.fTessFactor[2] = output.fInsideTessFactor =
36         g_cpuTessFactor;
37
38     //Pontos de controle das posições e
39     //normais dos corners são os mesmos do triangulo
```

```
40      float3 B003 = input[0].Position;
41      float3 B030 = input[1].Position;
42      float3 B300 = input[2].Position;
43      float3 N002 = input[0].Normal;
44      float3 N020 = input[1].Normal;
45      float3 N200 = input[2].Normal;
46
47      //Computa os pontos de controle restantes da geometria
48      output.B210 = ( ( 2.0f * B003 ) + B030 -
49          ( dot( ( B030 - B003 ), N002 ) * N002 ) ) / 3.0f;
50      output.B120 = ( ( 2.0f * B030 ) + B003 -
51          ( dot( ( B003 - B030 ), N020 ) * N020 ) ) / 3.0f;
52      output.B021 = ( ( 2.0f * B030 ) + B300 -
53          ( dot( ( B300 - B030 ), N020 ) * N020 ) ) / 3.0f;
54      output.B012 = ( ( 2.0f * B300 ) + B030 -
55          ( dot( ( B030 - B300 ), N200 ) * N200 ) ) / 3.0f;
56      output.B102 = ( ( 2.0f * B300 ) + B003 -
57          ( dot( ( B003 - B300 ), N200 ) * N200 ) ) / 3.0f;
58      output.B201 = ( ( 2.0f * B003 ) + B300 -
59          ( dot( ( B300 - B003 ), N002 ) * N002 ) ) / 3.0f;
60      // Ponto de Controle central
61      float3 E = ( output.B210 + output.B120 + output.B021
62          + output.B012 + output.B102 + output.B201 ) / 6.0f;
63      float3 V = ( B003 + B030 + B300 ) / 3.0f;
64      output.B111 = E + ( ( E - V ) / 2.0f );
65
66      // Computa os pontos de controle
67          //para variação quadratica das normais
68      float V12 = 2.0f * dot( B030 - B003, N002 + N020 )
69          / dot( B030 - B003, B030 - B003 );
70      output.N110 = normalize( N002 + N020 - V12 *
71          ( B030 - B003 ) );
72      float V23 = 2.0f * dot( B300 - B030, N020 + N200 )
73          / dot( B300 - B030, B300 - B030 );
74      output.N011 = normalize( N020 + N200 - V23 *
75          ( B300 - B030 ) );
76      float V31 = 2.0f * dot( B003 - B300, N200 + N002 )
77          / dot( B003 - B300, B003 - B300 );
78      output.N101 = normalize( N200 + N002 - V31 *
79          ( B003 - B300 ) );
80
```

```
81      return output;
82  }
```

Como os pontos de controle foram computados na parte constante, a parte principal do Hull Shader que é invocada por ponto de controle do patch somente repassará os valores para o Domain Shader.

Código 3.3: Hull Shader principal do PN-Triangles

```
1
2   struct HS_Output
3   {
4       float3      Position      : POSITION;
5       float3      Normal        : NORMAL;
6       float2      TexCoord      : TEXCOORD;
7   };
8
9   [domain("tri")]
10  [partitioning("integer")]
11  [outputtopology("triangle_cw")]
12  [patchconstantfunc("HS_Constant")]
13  [outputcontrolpoints(3)]
14  HS_Output HS_Main( InputPatch<HS_Input, 3> input,
15  uint i : SV_OutputControlPointID )
16  {
17      HS_Output output = (HS_Output)0;
18      // Apenas repassa os dados
19      output.Position = input[i].Position;
20      output.Normal = input[i].Normal;
21      output.TexCoord = input[i].TexCoord;
22      return output;
23  }
```

Com os pontos de controle calculados e os novos vértices gerados pelo Tessellator, o Domain Shader avalia a superfície de acordo com as equações 3-1 e 3-2.

Código 3.4: Domain Shader do PN-Triangles

```
1
2
3   struct DS_Output
4   {
5       float4 Position    : SV_Position;
6       float2 TexCoord    : TEXCOORD0;
```

```
7           float3 Normal           : NORMAL0;
8   };
9
10  [domain("tri")]
11  DS_Output DS( HS_ConstantOutput HSC,
12  const OutputPatch<HS_Output, 3> input,
13  float3 UVW : SV_DomainLocation )
14  {
15      DS_Output output = (DS_Output)0;
16
17      // Avalia a posição baseado nos pontos de controle
18          //e nas coordenadas baricentricas
19      float3 Position = input[0].Position * UVW.z*UVW.z*UVW.z+
20      input[1].Position * UVW.x * UVW.x * UVW.x +
21      input[2].Position * UVW.y * UVW.y * UVW.y +
22      HSC.B210 * 3 * UVW.z * UVW.z * UVW.x +
23      HSC.B120 * UVW.z * 3 * UVW.x * UVW.x +
24      HSC.B201 * 3 * UVW.z * UVW.z * 3 * UVW.y +
25      HSC.B021 * 3 * UVW.x * UVW.x * UVW.y +
26      HSC.B102 * UVW.z * 3 * UVW.y * UVW.y +
27      HSC.B012 * UVW.x * 3 * UVW.y * UVW.y +
28      HSC.B111 * 6.0f * UVW.y * UVW.x * UVW.z;
29
30      // Avalia as normais
31      float3 Normal =
32          input[0].Normal * UVW.z * UVW.z +
33      input[1].Normal * UVW.x * UVW.x +
34      input[2].Normal * UVW.y * UVW.y +
35      HSC.N110 * UVW.z * UVW.x +
36      HSC.N011 * UVW.x * UVW.y +
37      HSC.N101 * UVW.z * UVW.y;
38
39      // Normaliza
40      output.Normal = normalize( Normal );
41
42      // Interpola as coordenadas de textura linearmente
43      output.TexCoord = input[0].TexCoord * UVW.z
44          + input[1].TexCoord * UVW.x
45          + input[2].TexCoord * UVW.y;
46
47      // Transforma para espaço de tela
```

```
48      output.Position = mul( float4( Position.xyz, 1.0 ),
49          g_ViewProjection );
50
51      return output;
52  }
```

3.7 Phong Tessellation

O algoritmo do Phong Tessellation foi pensado de uma maneira a complementar o Phong Shading. Computacionalmente o Phong Shading avaliado por pixel não é custoso e faz um bom trabalho na iluminação do interior do modelo, porém, fica evidente a baixa quantidade de polígonos quando se olha para a silhueta de um modelo low-poly. Em termos de operações aritméticas, o Phong Tessellation é o algoritmo mais barato de subdivisão de superfícies disponível hoje em dia.

Uma tecelagem linear com interpolação baricêntrica pode ser definida da seguinte maneira:

$p : \Re^2 \to \Re^3$, para $w = 1 - u - v, \qquad u, v, w \in [0, 1]$

$$p(u, v) = (u, v, w)(p_i, p_j, p_k)^T \tag{3-23}$$

A interpolação linear das normais que ocorre entre o Domain Shader e o Pixel Shader ocorre da mesma maneira, só é preciso normalizar o resultado no Pixel Shader, este processo é amplamente usado no Phong Shading:

$n' : \Re^2 \to \Re^3$, para $w = 1 - u - v, \qquad u, v, w \in [0, 1]$

$$\begin{aligned} n'(u, v) &= (u, v, w)(n_i, n_j, n_k)^T, \\ n(u, v) &= n' / \left\| n' \right\| \end{aligned} \tag{3-24}$$

Em torno de cada vértice o plano tangente definido pela normal do vértice aponta a informação correta da posição apropriada para a geometria localmente. O algoritmo projeta um triângulo $\mathbf{t}$ no plano tangente definido pelo vértice $\mathbf{v_i}$ e faz uma interpolação baricêntrica com as informações dos planos tangentes dos outros dois vértices do triângulo para definir a geometria na vizinhança de $\mathbf{v_i}$. A geometria relativa aos outros dois vértices do triângulo $\mathbf{v_j}$ e $\mathbf{v_k}$ é definida de forma similar.

A avaliação dos pontos gerados para cada triângulo pode ser simplificada pelo seguinte procedimento:

1. Faça a tecelagem linearmente

2. Projete cada vértice gerado ortogonalmente nos três planos tangentes definidos pelos vértices do triângulo

3. Faça a interpolação baricêntrica dos três pontos conseguidos através das projeções. Esta é a posição final do vértice.

A Figura 3.15 ilustra o procedimento descrito acima. O algoritmo cumpre o que promete em termos de operações de aritméticas. É necessário apenas 3 projeções e duas interpolações lineares.

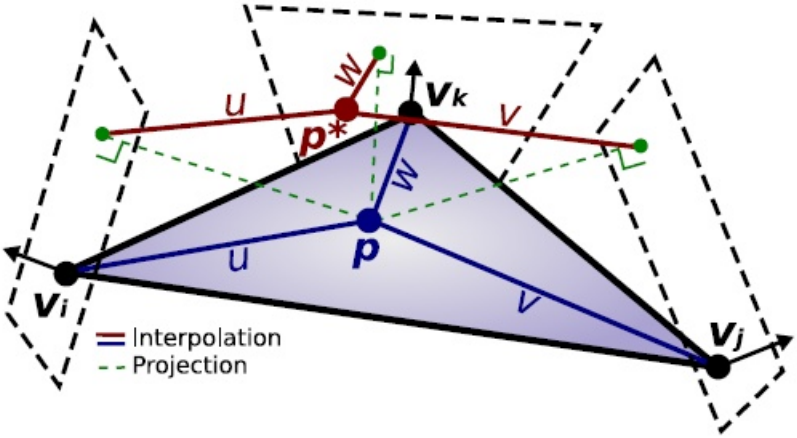

Figura 3.15: Projeções e interpolações do Phong Tessellation [Boubekeur and Alexa, 2008]

Seja $\pi_{\mathbf{i}}(\mathbf{q}) = \mathbf{q} - ((\mathbf{q} - \mathbf{p_i})^{\mathbf{T}}\mathbf{n_i})\mathbf{n_i}$ a projeção ortogonal de $\mathbf{q}$ no plano definido por $\mathbf{p_i}$ e $\mathbf{n_i}$. O Phong Tessellation é definido como:

$$p^*(u,v) = (u,v,w) \begin{pmatrix} \pi_i(\mathbf{p}(u,v)) \\ \pi_j(\mathbf{p}(u,v)) \\ \pi_k(\mathbf{p}(u,v)) \end{pmatrix} \tag{3-25}$$

3.7.1 Implementação

Assim como o PN-Triangles, o Phong Tessellation não requer nenhuma alteração no pipeline de arte para usá-lo. Modelos que não foram feitos com o Phong Tessellation em mente podem ser usados sem nenhum problema. No artigo original, [Boubekeur and Alexa, 2008] sugerem o uso de um fator α

por vértice do modelo caso a tecelagem final não fique conforme o desejado. Porém, caso o resultado da subdivisão não tenha ficado bom visualmente e seja necessário um controle do fator α por vértice do modelo, os artistas terão um grande re-trabalho. Este é um dos motivos porque técnicas como [Boubekeur et al., 2005] não foram bem aceitas pela indústria.

O Phong Tessellation também se encaixa bem com a estrutura do novo pipeline. Basicamente todo o trabalho do algoritmo é feito no Domain Shader. O Vertex Shader é o mesmo do PN-Triangles, simplesmente um vertex shader que repassa as informações. A parte principal do Hull Shader também fica igual ao PN-Triangles pois não alteraremos nenhum fator por ponto de controle. Já o Hull Shader constante muda um pouco, pois no Phong Tessellation apenas passaremos o fator de tecelagem para o Tessellator:

Código 3.5: Hull Shader constante do Phong Tessellation

```
1
2  HS_CONSTANT_DATA_OUTPUT BezierConstantHS
3  ( InputPatch<VS_CONTROL_POINT_OUTPUT,
4  INPUT_PATCH_SIZE> ip,
5  uint PatchID : SV_PrimitiveID )
6  {
7      HS_CONSTANT_DATA_OUTPUT Output;
8
9      Output.Edges[0] = Output.Edges[1] =
10     Output.Edges[2] =  Output.Inside = g_fTessellationFactor;
11
12     return Output;
13 }
```

Todo o trabalho do algoritmo fica no Domain Shader. Primeiro criamos uma função auxiliar que define uma projeção ortogonal em um plano. Ela recebe a normal do plano de projeção, um ponto no plano, o ponto a ser projetado e retorna a projeção:

Código 3.6: Função auxiliar para projeção ortogonal

```
1
2  float3 projIntoPlane(float3 planeNormal,
3  float3 planePoint,
4  float3 pointToProject)
5  {
6      float3 res;
7      res = pointToProject -
8      dot(pointToProject - planePoint, planeNormal)*planeNormal;
```

```
9
10      return res;
11  }
```

No Domain Shader seguimos os passos descritos na seção 3.7. Uma interpolação baricêntrica linear no plano do triângulo, três projeções ortogonais usando a função auxiliar descrita acima e interpola novamente baseado nas projeções para achar a posição final do vértice. As normais e coordenadas de textura seguem uma interpolação linear baricêntrica.

Código 3.7: Phong Tessellation Domain Shader

```
1
2   struct DS_OUTPUT
3   {
4       float4 vPosition     : SV_POSITION;
5           float3 vNormal           : NORMAL0;
6           float2 vTexCoord       : TEXCOORD0;
7   };
8
9   [domain("tri")]
10  DS_OUTPUT Bezier( HS_CONSTANT_DATA_OUTPUT input,
11                    float3 UV : SV_DomainLocation,
12                    const OutputPatch<HS_OUTPUT,
13                                      OUTPUT_PATCH_SIZE> patch )
14  {
15      DS_OUTPUT Output;
16
17      //interpolação linear no plano do triângulo
18      float3 p = UV.x*patch[0].vPosition +
19                 UV.y*patch[1].vPosition +
20                 UV.z*patch[2].vPosition;
21
22      //três projeções ortogonais nos planos tangentes
23      float3 pProjU =
24      projIntoPlane(patch[0].vNormal, patch[0].vPosition, p);
25      float3 pProjV =
26      projIntoPlane(patch[1].vNormal, patch[1].vPosition, p);
27      float3 pProjW =
28      projIntoPlane(patch[2].vNormal, patch[2].vPosition, p);
29
30      //interpola de novo para achar a posição final
31      float3 pNovo = UV.x*pProjU + UV.y*pProjV + UV.z*pProjW;
```

```
32
33      //Output em espaço de tela para o rasterizador
34      Output.vPosition=mul(float4(pNovo,1),g_mViewProjection);
35      //Normal interpolada linearmente
36      Output.vNormal = normalize(UV.x*patch[0].vNormal +
37                                 UV.y*patch[1].vNormal +
38                                 UV.z*patch[2].vNormal);
39      //Coordenada de Textura interpolada linearmente
40      Output.vTexCoord = UV.x*patch[0].vTexCoord +
41      UV.y*patch[1].vTexCoord +
42      UV.z*patch[2].vTexCoord;
43
44      return Output; }
```

3.8 Resultados

Nesta seção avaliaremos os dois algoritmos tanto qualitativamente quanto no aspecto de performance.

3.8.1 Qualidade

Os autores do Phong Tessellation argumentam que o patch quadrático de normais criado pelo PN-Triangles é pouco útil na prática. Isto vai de encontro com a Figura 3.14 que mostra uma diferença bem acentuada entre as abordagens com interpolação linear e com a construção do patch quadrático de normais. Vamos avaliar alguns modelos com o mesmo fator de tecelagem para ambos os algoritmos. Porém, usaremos tanto o patch quadrático quanto a abordagem linear para geração das normais no caso do PN-Triangles e comparar com as normais do Phong Tessellation que são geradas linearmente.

Apesar de ambos os algoritmos terem apenas continuidade C_0 garantidas, a Figura 3.18 (PN-Triangles) mostra uma continuidade bem mais suave do que a Figura 3.17 (Phong Tessellation) com relação à malha original (Figura 3.16).

As Figuras 3.19 e 3.20 mostram uma diferença significante na qualidade da sombra gerada pela iluminação. Os círculos vermelhos apontam uma sombra facetada na figura com interpolação linear das normais, enquanto pode-se perceber uma sombra mais suave no caso da interpolação quadrática. Os círculos verdes mostram um caso característico de ponto de inflexão das normais de um triângulo, onde o tecido da manga do modelo aparenta estar

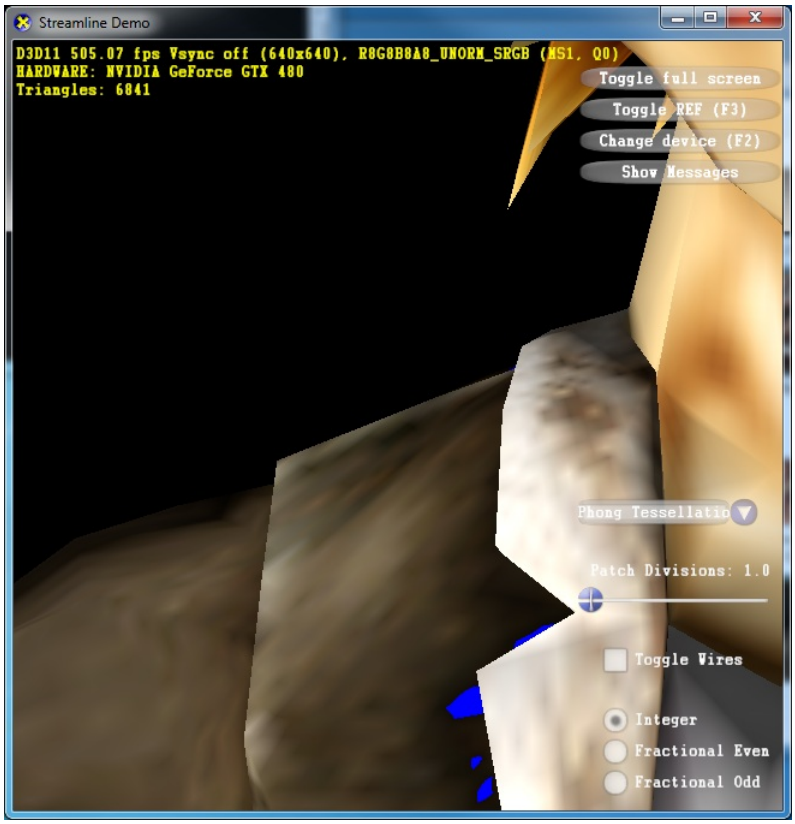

Figura 3.16: Modelo original sem tecelagem

dobrado no caso da interpolação quadrática. Já no caso da interpolação linear essa inflexão não é levada em conta e a iluminação não mostra essa dobra.

As Figuras 3.22 e 3.23 mostram claramente a suavidade maior do algoritmo PN-Triangles em relação à malha original (Figura 3.21).

Como este modelo possui pouca curvatura, a interpolação linear não sofreu grande perda de qualidade em comparação com a interpolação quadrática das normais. Nota-se uma pequena diferença na área apontada pelos círculos vermelhos nas Figuras 3.24 e 3.25.

Novamente, olhando as figuras 3.26, 3.27 e 3.28 percebe-se que os triângulos criados pelo Phong Tessellation apresentam continuidade menos suave que os criados pelo algoritmo do PN-Triangles. Porém, por este modelo ser de um objeto orgânico, a deformação do Phong Tessellation ficou agradável visualmente.

Mais uma vez os círculos vermelhos na Figura 3.29 mostram uma sombra facetada para a interpolação linear de normais. Já os mesmos círculos na figura 3.30 apresentam sombras mais suaves.

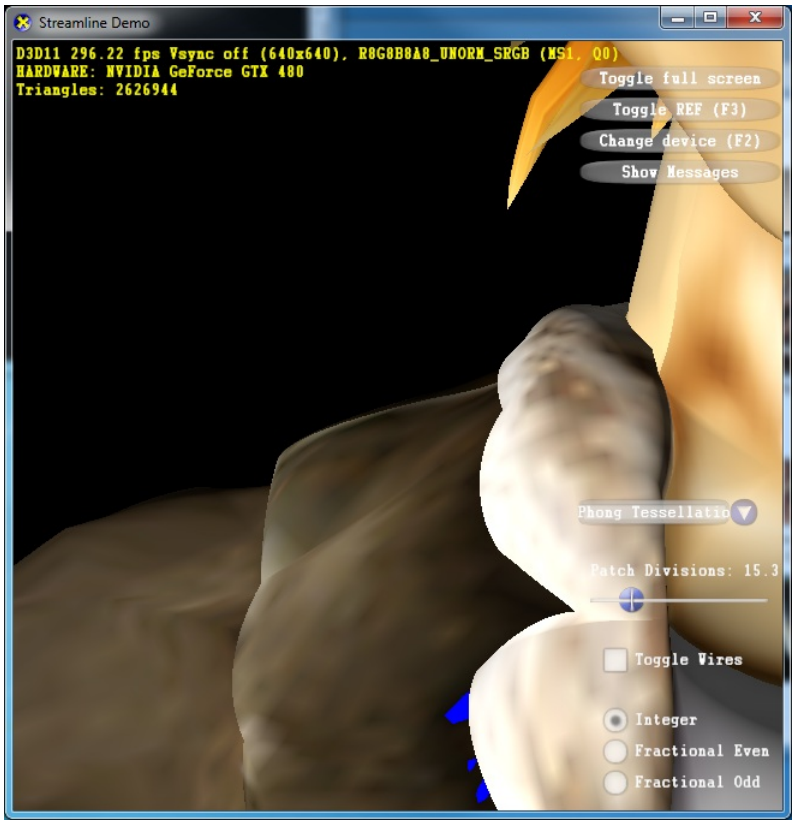

Figura 3.17: Modelo usando o algoritmo Phong Tessellation

3.8.2 Performance

A máquina utilizada nos testes foi um Core2Quad Q6600 com 4GB de RAM e placa gráfica RadeonHD 8450. Foi testada uma cena com vários modelos de um mesmo tipo (Tigre ou Pessoa) sem utilização de instanciação de geometria. Evitou-se colocar apenas um modelo só com um alto fator de tecelagem para não acontecer uma sobrecarga no rasterizador pela criação de triângulos muito pequenos (micro-polígonos). A iluminação também foi desligada para evitar sobrecarga no pixel shader. O intuito foi proporcionar que o maior gargalo fosse as operações aritméticas de ambos os algoritmos.

As Tabelas 3.1 e 3.2 mostram os FPS e a quantidade de triângulos para os modelos da Pessoa e do Tigre respectivamente.

As Figuras 3.31 e 3.32 mostram os resultados expressos nas tabelas 3.1 e 3.2.

As Tabelas 3.4 e 3.3 mostram o ganho tanto percentual quanto absoluto (em FPS) do algoritmo Phong Tessellation em comparação com o PN-Triangles.

Pessoa (Número de Triângulos em milhões)	Phong Tessellation (FPS)	PN-Triangles(FPS)
0.103	82	54
0.616	65	43
1.33	51	33
2.46	31	20
3.80	22	14
5.54	16	10
7.49	12	8
9.85	9	6
12.40	8	5
15.40	7	4
18.60	6	3

Tabela 3.1: FPS do modelo da Pessoa usando o Phong Tessellation e o PN-Triangles

Tigre (Número de Triângulos em milhões)	Phong Tessellation (FPS)	PN-Triangles(FPS)
0.036	135	107
0.47	101	74
0.87	72	50
1.30	53	37
2.00	42	28
2.60	32	22
3.50	26	17
4.40	22	14
5.40	18	11
6.50	15	9
7.80	12	8
9.1	11	7
11.0	9	6
12.0	8	5
14.0	7	4
16.0	6	4

Tabela 3.2: FPS do modelo do Tigre usando o Phong Tessellation e o PN-Triangles

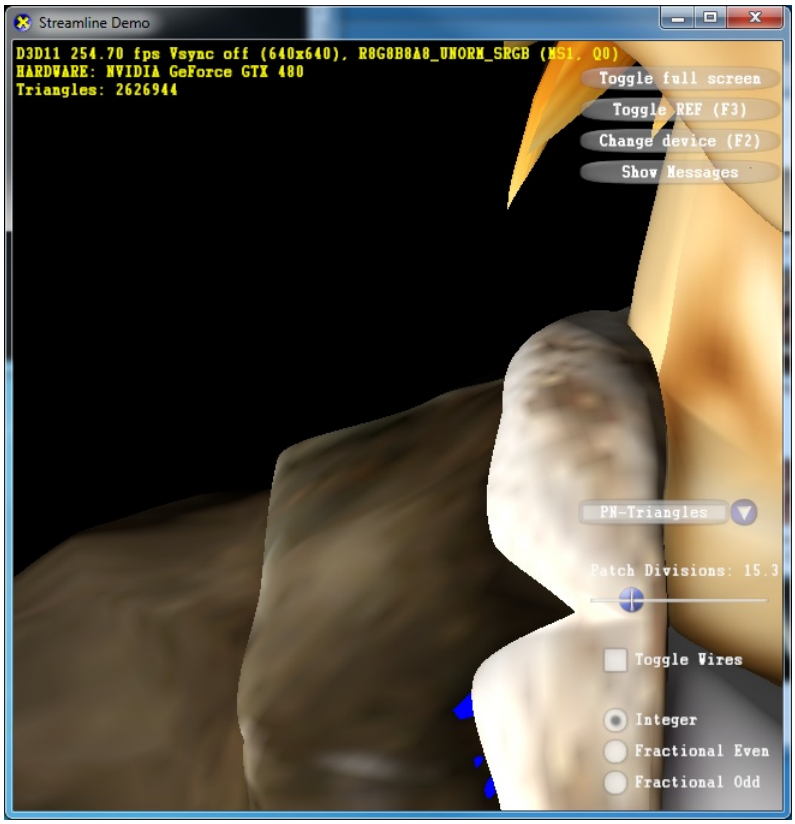

Figura 3.18: Modelo usando o algoritmo PN-Triangles e normais quadráticas

3.8.3 Discussão

Pelos gráficos e resultados apresentados na seção anterior, nota-se que o Phong Tessellation oferece um ganho de performance considerável quando comparado ao algoritmo do PN-Triangles (em média cerca de 40%). Porém, as imagens 3.17 e 3.18 mostram uma qualidade da subdivisão muito superior do método PN-Triangles. Além disso, em alguns casos (Figuras 3.20 e 3.30) a interpolação quadrática das normais também apresentou resultados visuais melhores. Para jogos ou aplicações muito dinâmicas e de alta performance onde o usuário não vai se reter a observações delicadas o Phong Tessellation mostra-se uma excelente opção, principalmente para o refinamento de silhueta, onde um leve aumento da tecelagem resolveria casos como o da Figura 3.1. Já para jogos com menos dinamismo ou aplicações de visualização o PN-Triangles deve ser considerado pela melhor qualidade visual em seu método de subdivisão.

Tigre (Número de Triângulos em milhões)	Ganho (em %)	Ganho (em FPS)
0.036	26.17%	28
0.47	36.49%	27
0.87	44.00%	22
1.30	43.24%	16
2.00	50.00%	14
2.60	45.45%	10
3.50	52.94%	9
4.40	57.14%	8
5.40	63.64%	7
6.50	66.67%	6
7.80	50.00%	4
9.1	57.14%	4
11.0	50.00%	3
12.0	60.00%	3
14.0	75.00%	3
16.0	50.00%	2

Tabela 3.3: Ganho percentual e absoluto do Phong Tessellation sobre o PN-Triangles para o modelo do Tigre

Pessoa (Número de Triângulos em milhões)	Ganho (em %)	Ganho (em FPS)
0.103	51.85%	28
0.616	51.16%	22
1.33	54.55%	18
2.46	55.00%	11
3.80	57.14%	8
5.54	60.00%	6
7.49	50.00%	4
9.85	50.00%	3
12.40	60.00%	3
15.40	75.00%	3
18.60	100.00%	3

Tabela 3.4: Ganho percentual e absoluto do Phong Tessellation sobre o PN-Triangles para o modelo da Pessoa

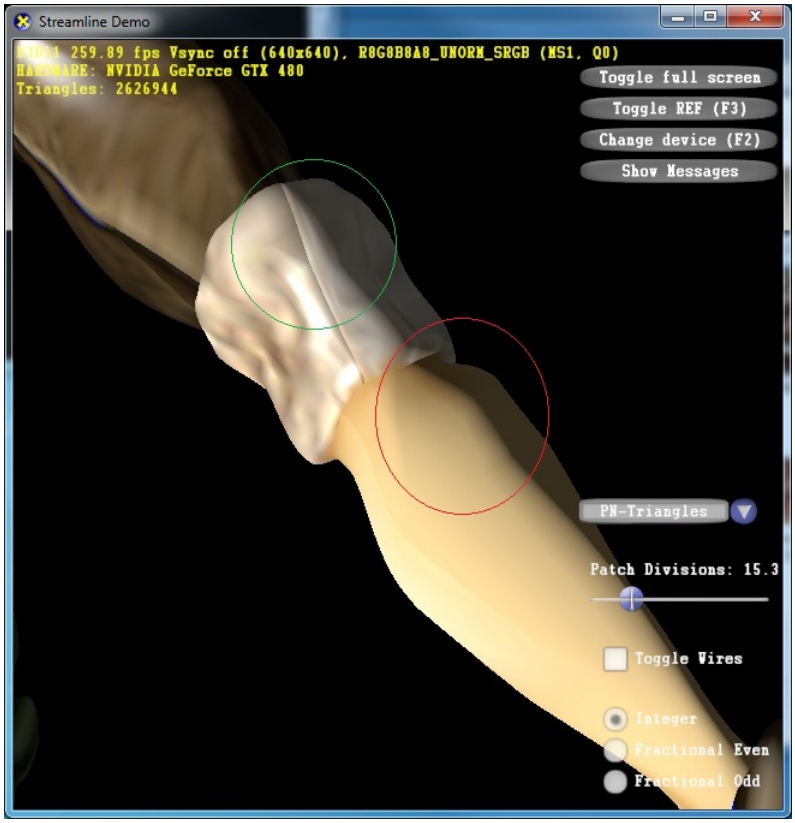

Figura 3.19: PN-Triangles com interpolação linear das normais

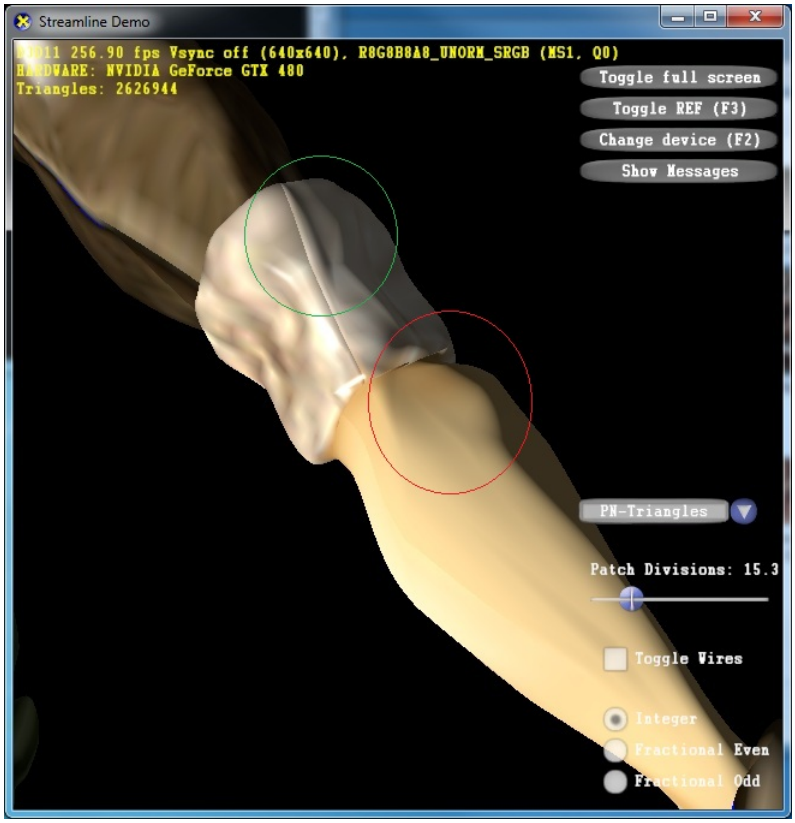

Figura 3.20: PN-Triangles com interpolação quadrática das normais

Figura 3.21: Modelo original sem tecelagem

Figura 3.22: Modelo usando o algoritmo Phong Tessellation

Figura 3.23: Modelo usando o algoritmo PN-Triangles e normais quadráticas

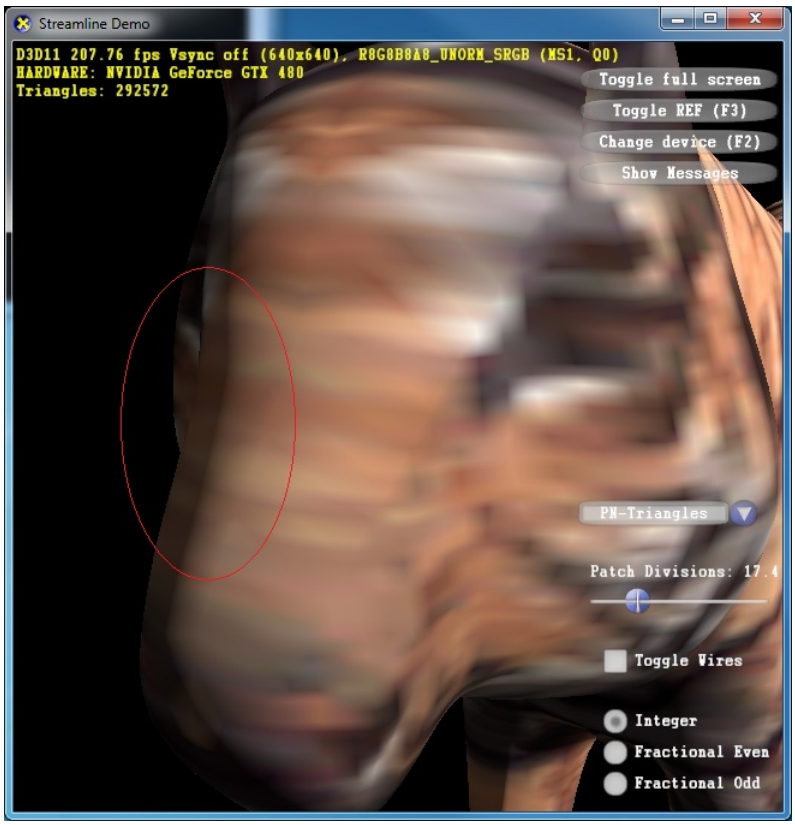

Figura 3.24: PN-Triangles com interpolação linear das normais

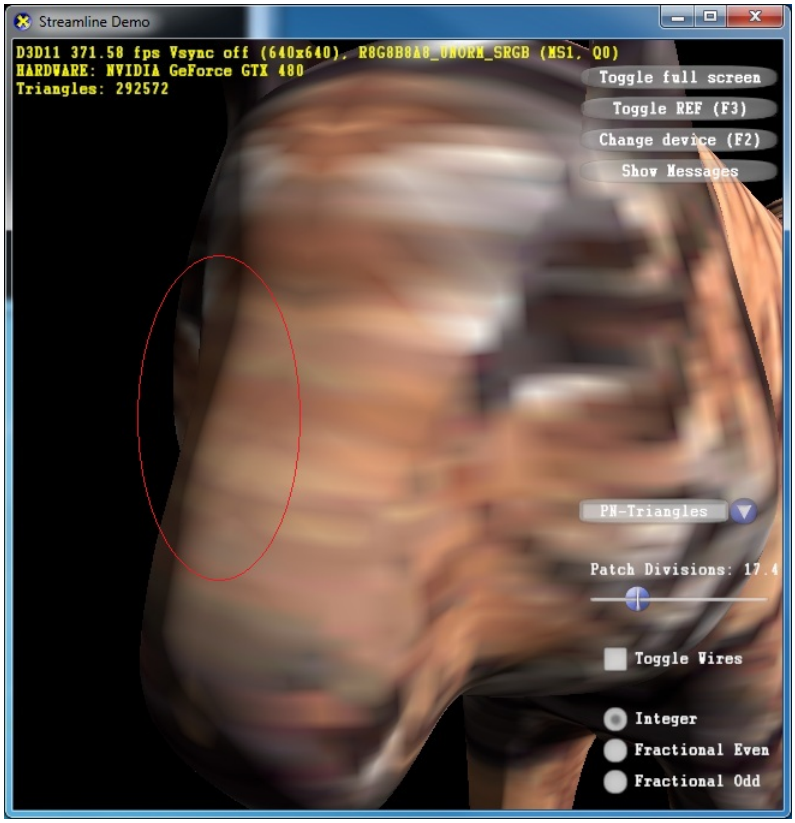

Figura 3.25: PN-Triangles com interpolação quadrática das normais

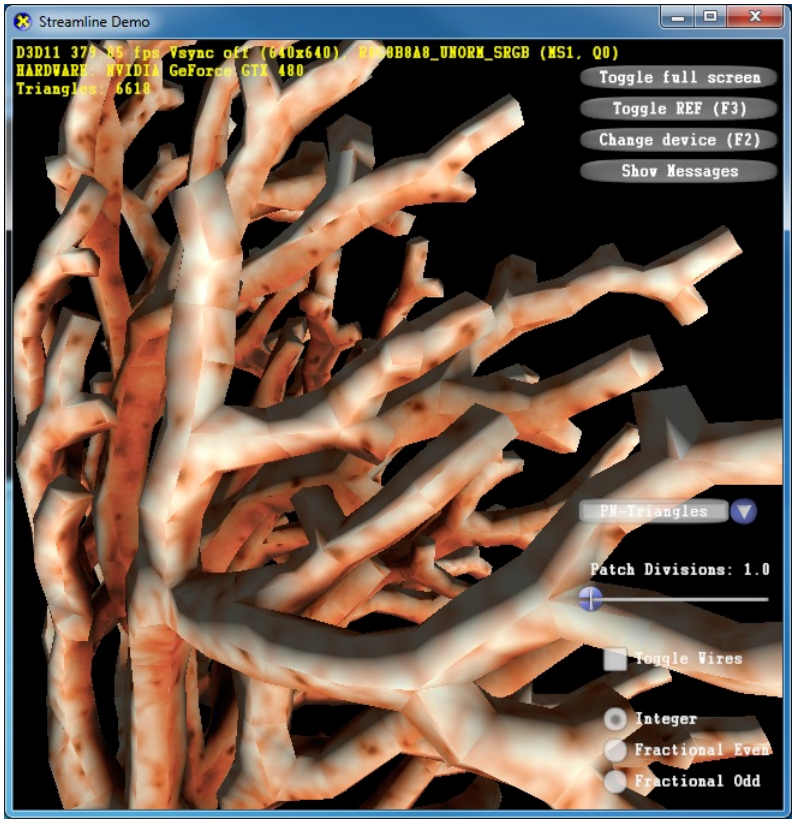

Figura 3.26: Modelo original sem tecelagem

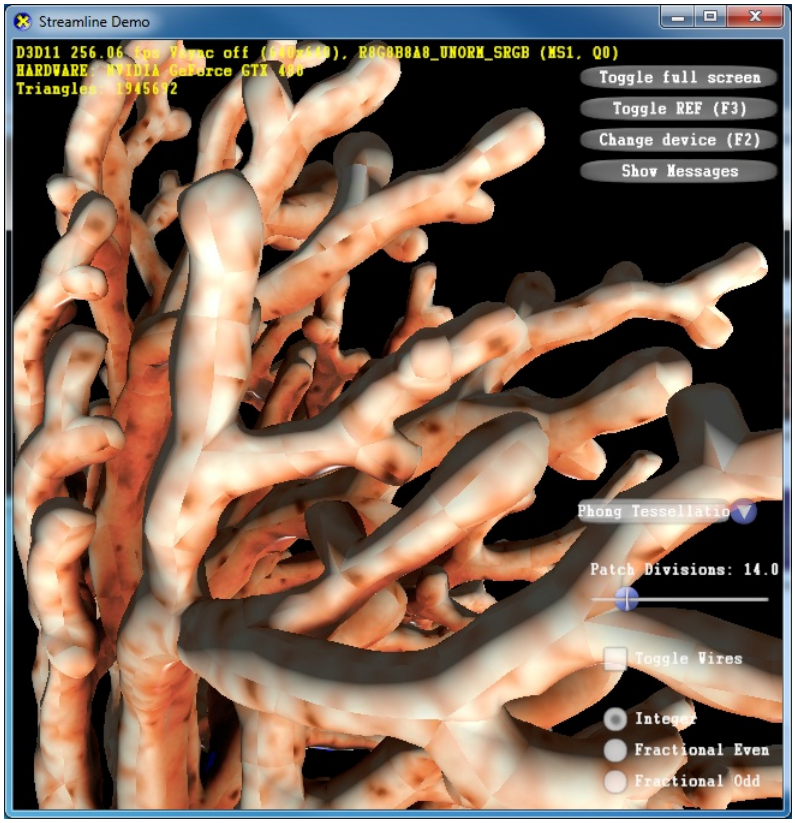

Figura 3.27: Modelo usando o algoritmo Phong Tessellation

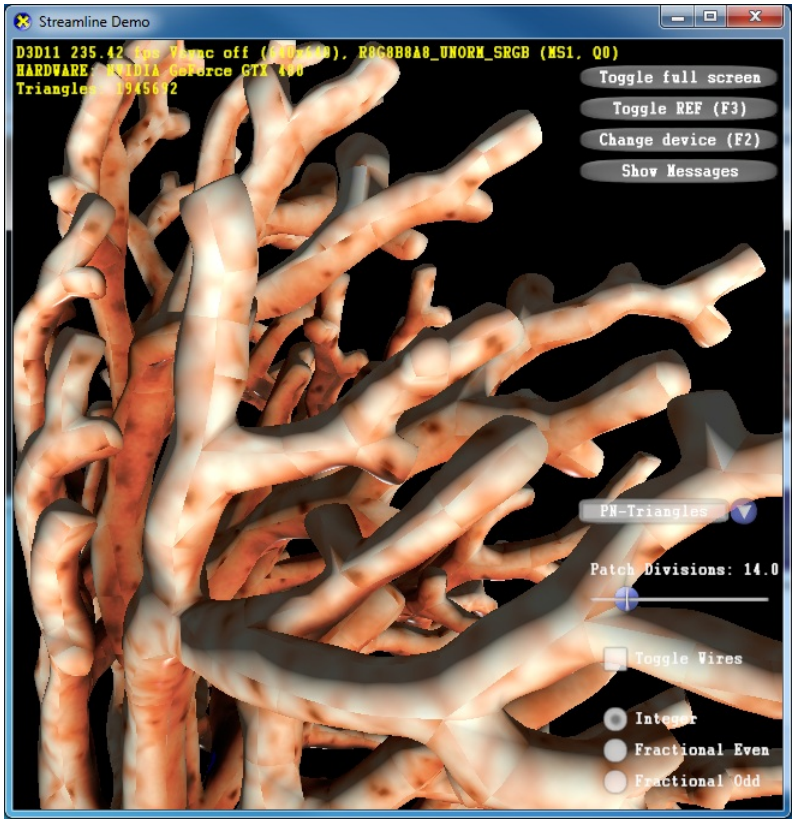

Figura 3.28: Modelo usando o algoritmo PN-Triangles e normais quadráticas

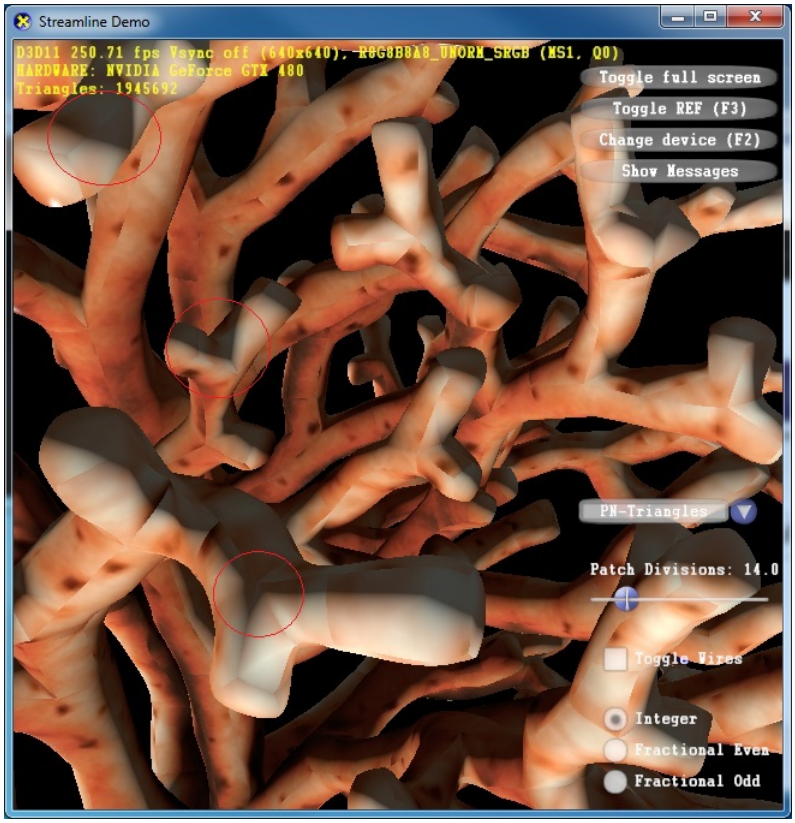

Figura 3.29: PN-Triangles com interpolação linear das normais

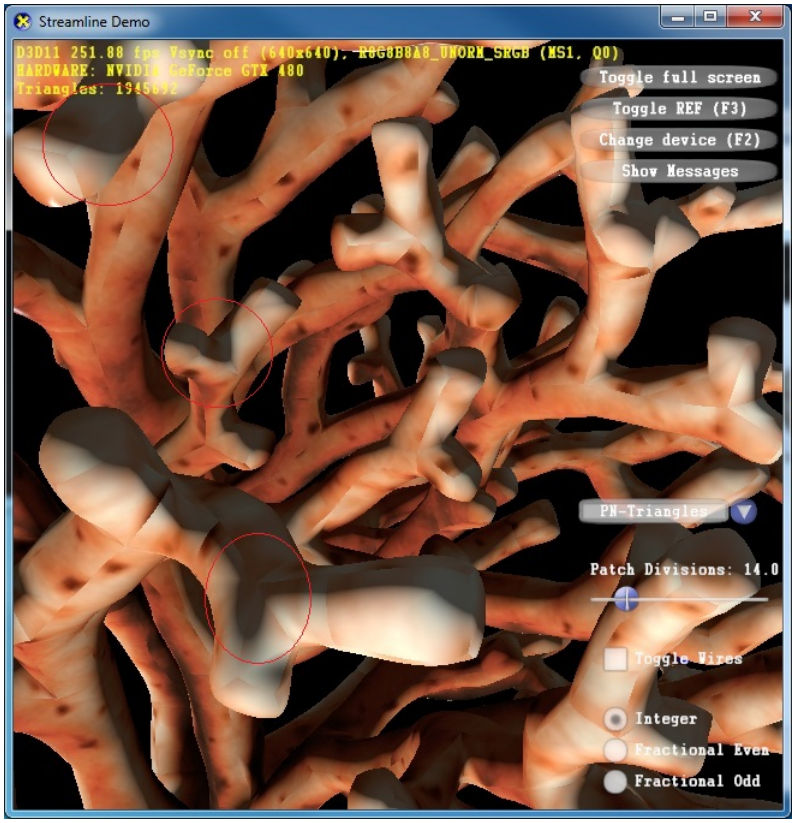

Figura 3.30: PN-Triangles com interpolação quadrática das normais

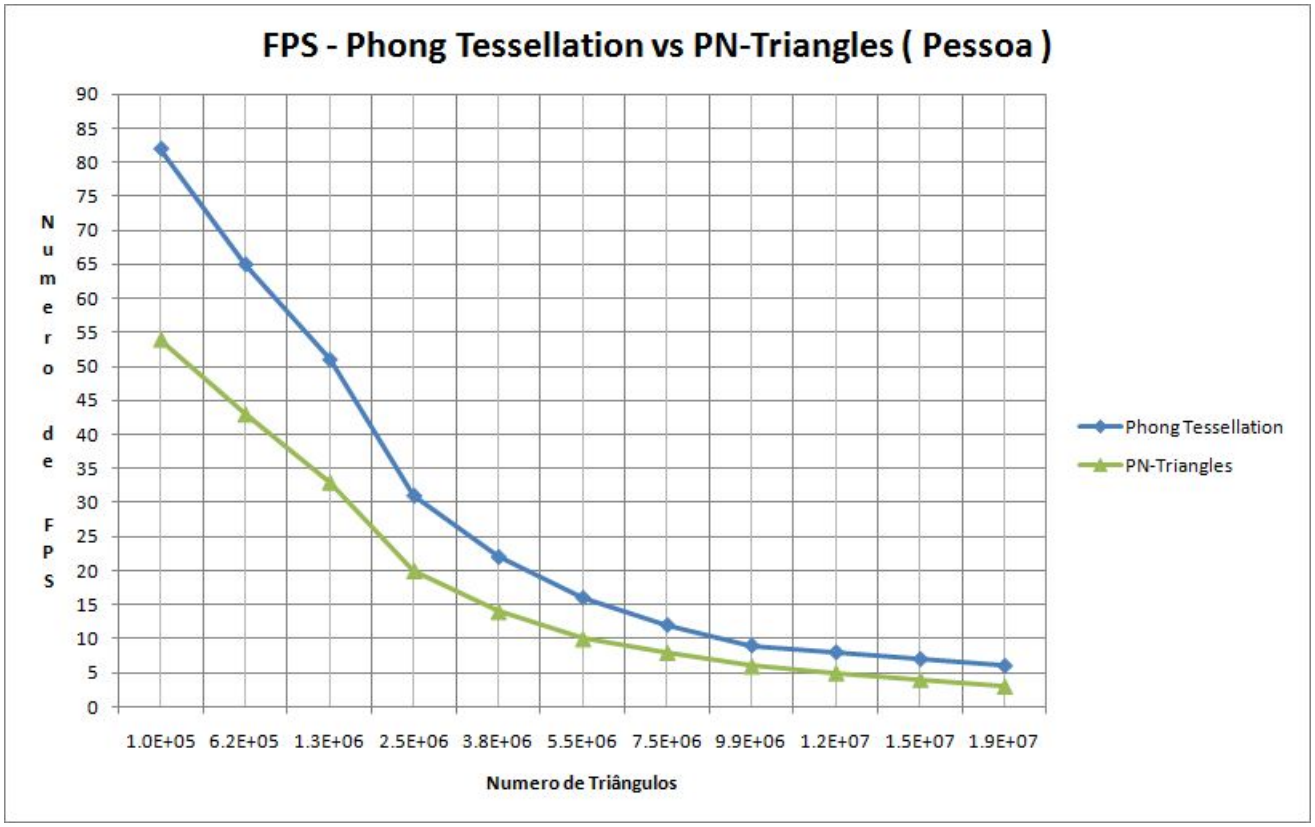

Figura 3.31: Demonstrativo da diferença de performance entre o Phong Tessellation e o PN-Triangles para o modelo da Pessoa

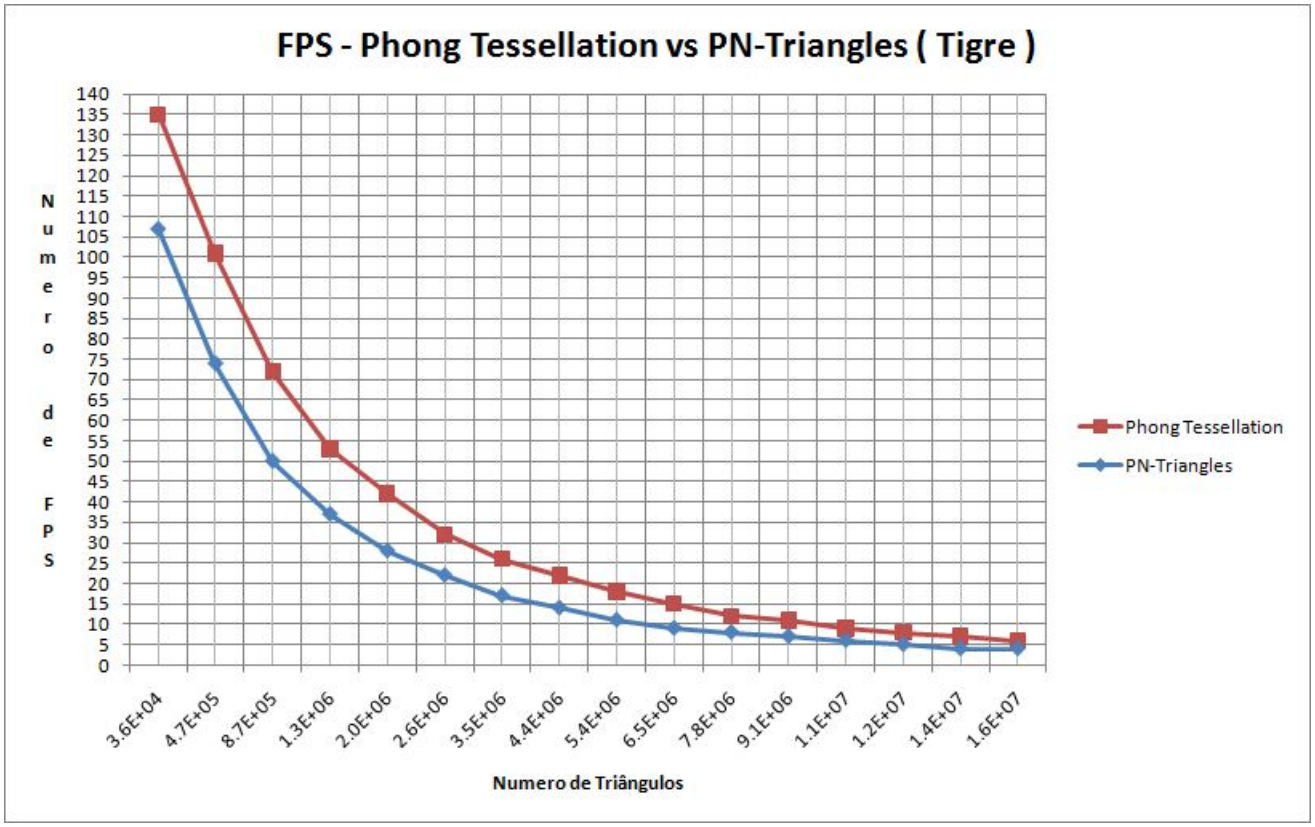

Figura 3.32: Demonstrativo da diferença de performance entre o Phong Tessellation e o PN-Triangles para o modelo do Tigre

4
Renderizando Tubos a partir de Curvas Discretas com Anti-Aliasing e LOD Contínuo usando Tecelagem em Hardware

4.1
Introdução

Tubos 3D podem ser a solução para uma série de problemas em diversas áreas. Para ilustrar esses casos vamos passar por alguns exemplos. Um caso apropriado é a visualização de partículas no tempo na forma de linhas de fluxo. A visualização de simulações de fluxo é outro exemplo deste tipo de aplicação [Stoll et al., 2005]. Na medicina, existem várias aplicações que podem fazer uso de tubos, por exemplo, visualização de tratos da substância branca [Merhof et al., 2006] e fibrilação cardíaca [Kondratieva et al., 2005]. Na física, campos vetoriais também podem ser visualizados como grupos de tubos. Na industria de óleo e gás, os tubos podem ser usados em simulações 3D e para a renderização de objetos específicos. Tais objetos, como poços e risers, podem ser visualizados como tubos 3D e foram a principal motivação para o desenvolvimento desta técnica, Figura 4.1.

Como mencionado acima, a principal motivação para esta solução veio de um problema ao visualizar tubos 3D. Nós nos deparamos com uma quantidade massiva de poços e dutos petróleo sendo representados por pontos discretizados em um software de GIS. Foi desenvolvida uma primeira abordagem sem o uso do novo pipeline, e a qualidade visual ficou exatamente como desejada. Contudo, houve um gargalo muito grande no visualizador com a transferência de banda da CPU para a GPU quando renderizávamos milhares de dutos, poços e risers.

Os tubos eram criados na CPU e passados para a GPU a cada frame. Além disso, em aplicações CAD é importante poder visualizar esses tubos mesmo quando a câmera está posicionada longe deles. Esse requisito implica em um aliasing indesejável. Foi desenvolvida então uma segunda proposta, usando o novo pipeline, que cria as malhas dinamicamente, melhorando o problema do aliasing. Além disso, nossa abordagem exercita todos os novos estágios do pipeline programável e é simples e direta.

Os tubos 3D são representados por um conjunto de pontos no R3 e um número escalar para o raio. No pipeline gráfico convencional, é preciso construir uma malha baseada nesta representação antes de enviá-la para a GPU. Nesta proposta, nós exploramos o novo pipeline das GPUs, para que as malhas sejam criadas no último momento possível dentro das GPUs. Desta maneira, nós aliviamos a transferência da CPU para GPU, que é um dos gargalos mais comuns para aplicações de renderização em tempo real que necessitam de representações com muitos vértices. Adicionalmente, nós usamos os parâmetros de renderização do frame corrente para gerar malhas melhores, controlando continuamente o nível de detalhe (LOD) e melhorando a qualidade visual.

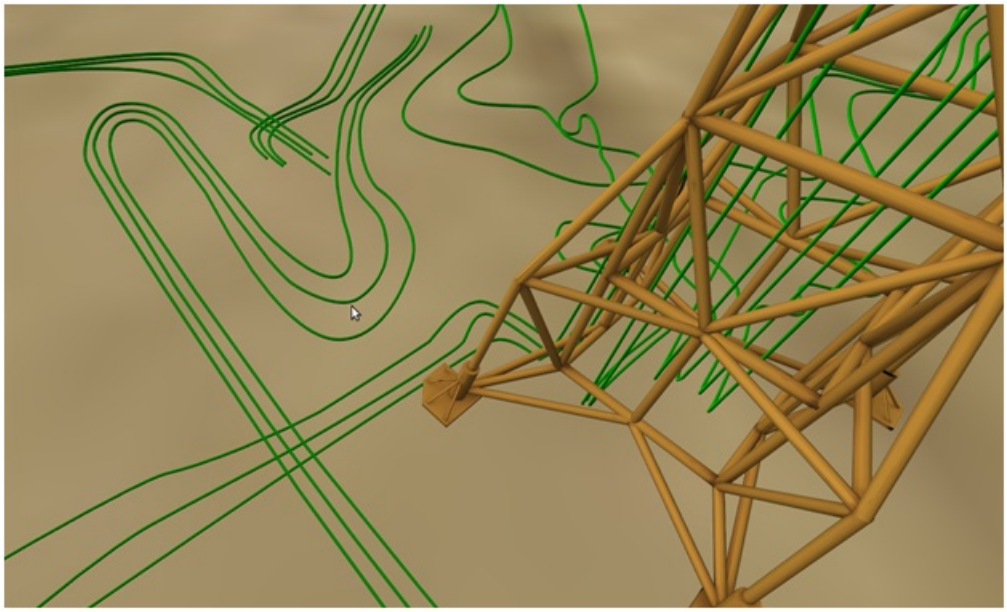

Figura 4.1: Tubos 3D renderizados com a técnica proposta em um visualizador de campos de petróleo.

Existem também abordagens que usam o ray-casting para renderizar tubos 3D, porém, esse método é muito intenso em operações aritméticas ao gerar milhares de tubos e também podem produzir alguns artefatos indesejáveis. A abordagem mais simples de renderizar os tubos como simples linhas é razoável do ponto de vista de performance, mas a qualidade visual deixa muito a desejar.

Nós consideramos duas maneiras de construir os tubos 3D a partir de uma série de pontos: eles podem estar diretamente conectados ou serem usados como pontos de controle de uma spline. Em ambos os casos, só são passados para a GPU uma série de pontos e algumas informações auxiliares por seção do tubo que será explicado mais adiante. Nosso método demonstrou ser uma solução bem balanceada para a renderização de tubos 3D em aplicações CAD. Houve ganhos consideráveis nos quatro principais pontos na renderização em

tempo real: performance, qualidade de imagem, simplicidade de implementação e consumo de memória.

4.2 Trabalhos Relacionados

Desde que as GPUs se tornaram programáveis, vários esforços foram feitos para renderizar tubos 3D de maneiras mais eficientes. Restritos aos primeiros estágios programáveis (vertex e pixel shaders), alguns trabalhos combinaram ray-casting com rasterização [Merhof et al., 2006], [Stoll et al., 2005]. Esta técnica, também conhecida como primitivas de GPU estendidas [de Toledo and Levy, 2004], pode apresentar alguns artefatos no caso de tubos 3D.

As primitivas de GPU estendidas necessitam de um rasterizador para executar o algoritmo de ray casting, e quadstrips são a escolha natural para cilindros. Contudo, em áreas com alto grau de curvatura, quando alinhado ao ângulo de visão da câmera, as quadstrips trocam de direção, impedindo um ray casting correto. Existem dois métodos propostos em trabalhos anteriores para resolver esta limitação: o uso de um círculo de sprite extra [Merhof et al., 2006] e uma resolução de quad-strip aumentada localmente [Stoll et al., 2005]. Em ambos os casos, uma passada extra é necessária para aliviar o problema visual. Em nosso trabalho nós não usamos nenhum tipo de algoritmo de ray casting e os tubos são renderizados somente como malhas poligonais.

Mais recentemente, com o pipeline novo, é possivel criar vértices dentro da GPU para acelerar a visualização. No entanto, até onde pudemos pesquisar, não encontramos implementação para geração de tubos usando o novo pipeline.

4.3 O algoritmo de geração de tubos

O algoritmo consiste basicamente em 3 fases: preparação dos dados com um pré-processamento linear, criação da geometria e melhora do aliasing. Estas etapas serão detalhadas a seguir.

4.3.1 Preparando os dados para a renderização

Definido os tubos 3D como sequências de pontos

Os tubos 3D são considerados cilindros gerais com uma área de seção constante. Eles são guiados por uma curva dada como seu caminho no espaço 3D. Nosso algoritmo cria uma geometria que reconstrói esses cilindros gerais.

Para criar a geometria é necessário obter a sequência de pontos que define o caminho do tubo. Como os pontos são sequenciais, podemos assumir que cada ponto é conectado com seu vizinho por uma segmento de reta. Logo, cada ponto deve ter uma derivada numérica. Contudo, as sequências de pontos devem ser interpretadas de duas maneiras diferentes dependendo do caso. Existe um caso bom, onde a sequência de pontos é uma curva, representando o tubo propriamente dito (linha azul na Figura 4.2). Porém existe um caso ruim onde a sequência de pontos é um mero caminho onde o cilindro deve tocar (linha preta na Figura 4.2). Este caso indesejável pode ser facilmente convertido para um caso bom através de um pequeno pré-processamento que será descrito no próximo paragráfo.

Figura 4.2: Grupo de pontos em preto representando o caminho do tubo. Em azul a linha central do tubo.

Assegurando a corretude na sequência de pontos

Precisamos transformar os pontos esparsados em uma série de pontos mais refinados que definem bem a linha central do caminho que o cilindro deve seguir, como mostrado na Figura 4.2. Melhoramos a precisão da sequência de pontos interpolando suavemente com o uso de splines de Catmull-Rom [Catmull and Rom, 1974]. O incremento do fator de interpolação requerido para a geração da spline decide quantos pontos deverão ser gerados.

Fazendo uma boa aproximação

Nosso algoritmo aproxima um cilindro geral contínuo por uma série de cilindros retos, menores e discretizados. Naturalmente, a quantidade de pontos melhora a precisão da reconstrução do tubo. Além disso, lugares que contêm uma derivada muito alta, por exemplo, curvas acentuadas e joelhos necessitam de mais pontos (Figura 4.3).

O problema é simples, nós devemos cortar os pontos que têm uma derivada baixa e manter os pontos com derivada alta. Este filtro é aplicado aos pontos navegando pela sequência e selecionando os pontos relevantes sempre que o grau de desvio da curva atingir uma certa tolerância.

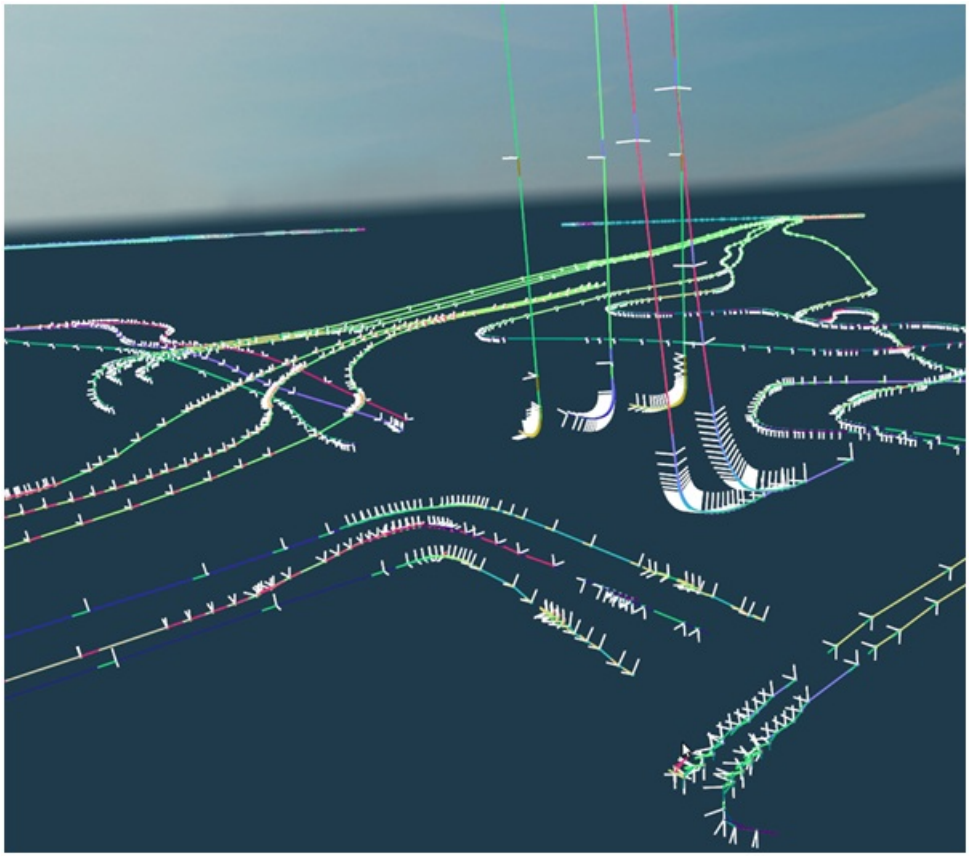

Figura 4.3: Seleção de pontos com as tangentes, normias e bi-normais associadas ao longo da curva. Áreas com derivada numérica alta requerem mais pontos para a reconstrução precisa

A quantidade de geometria desejada determina o número de pontos a serem selecionados. Nós descreveremos a geração da geometria nas seções subsequentes.

Tangentes, Normais e Binormais

O algoritmo desenvolvido precisa de uma normal e binormal para transformar a geometria gerada em um tubo 3D, como explicado na próxima seção. A normal $\vec{N_i}$ de um ponto P_i é qualquer vetor ortogonal a tangente $\vec{T_i}$ da curva em P_i. A binormal $\vec{B_i}$ de P_i é qualquer vetor ortogonal a ambos os vetores $\vec{N_i}$ e $\vec{T_i}$. A tangente $\vec{T_i}$ é a derivada numérica no ponto P_i. Para cada ponto P_i que selecionamos na seção anterior, nós precisamos armazenar os vetores $\vec{N_i}$ e $\vec{B_i}$ correspondentes. A Figura 4.3 mostra várias tangentes normais e binormais ao longo das curvas dos tubos.

4.3.2
Criando a geometria

A geometria da malha do tubo é gerada pelo Tessellator, resultando em uma renderização muito eficiente. A largura de banda CPU-GPU é usada somente para invocar o pipeline e passar alguma informação topológica. Como explicado anteriormente, o Tessellator pode gerar um conjunto de vértices em um domínio escolhido. Em nosso caso foi escolhida uma topologia de quads, logo nossos vértices são criados em um domínio UV[0..1], como será mostrado na Figura 4.4E. O objetivo é tranformar o grid 2D e um conjunto de pontos em um tubo 3D.

Vamos isolar o problema para um único ponto P_i da curva, por enquanto, considere que nosso domínio é uma linha reta L_i (Figura 4.4F) composta por um conjunto discreto de pontos $p_0, ..., p_n$. Nosso objetivo é transformar L_i em um corte ortogonal circular do tubo em P_i (Figura 4.4H). Para converter os pontos de L_i para a seção de corte nós devemos tranformar cada ponto $p_0, ..., p_n$. de uma linha 2D para os pontos $p'_0, ..., p'_n$. de um círculo 3D (Figura 4.4G) no plano $y = 0$. Fazemos isso transformando a posição de p_i relativa ao comprimento total de L_i para um ângulo de acordo com a equação 4-1

$$\theta = 2\pi \frac{p_i}{p_n} \tag{4-1}$$

Considere p_i/p_n como o tamanho de p_i ao londo de L_i. Agora que os pontos estão mapeados para um círculo simples no plano $y = 0$, o problema é transformar o círculo para a seção de corte do tubo em P_i.

Essa transformação é feita de acordo com o método de [Bloomenthal, 1990], onde a posição final dos pontos $p''_0, ..., p''_n$.(Figura 4.4H) em espaço de mundo é dado pela equação 4-2.

$$F_p = (P_{ix} + p'_x N_x + p'_z B_x, P_{iy} + p'_x N_y + p'_z B_y, P_{iz} + p'_x N_z + p'zB_z) \tag{4-2}$$

Na equação 4-2 N e B são, respectivamente, a Normal e Binormal no ponto P_i. Detalhes posteriores são explicados na seção de implementação mais adiante.

Com o algoritmo para converter a linha em uma seção do tubo em coordenadas de mundo, falta apenas um pequeno passo para ser possível renderizar o tubo inteiro. Como o tessellator fornece um grid regular de vértices, basta transformar cada linha deste grid em uma seção do tubo como

ilustrado na Figura 4.4. O raio do círculo é escalado adaptativamente como explicado anteriormente, mas por enquanto podemos assumir um valor fixo.

Nós precisamos de um grid regular com 64 linhas em um eixo. Portanto, é preciso manter o fator de tecelagem de um dos eixos e de uma parte interna fixo em 64. Para o outro eixo e a outra parte interna, não existe valor fixo, o único requisito é apenas serem iguais, pois precisamos de um grid regular. Logo, nós fazemos esses valores diretamente dependentes da distância para a câmera para o quad, resultando em um controle de LOD contínuo.

4.3.3 Removendo o Aliasing

O efeito de aliasing é esperado quando renderizamos uma linha, pois uma de suas dimensões é muitas vezes menor do que a outra. Em aplicações CAD é importante ver os tubos mesmo quando eles estão longe da câmera.

Nossa abordagem para remover o aliasing é simples e bastante efetiva e utiliza o mesmo princípio de [de Toledo and Lévy, 2008b]. Para cada ponto P_i do conjunto de pontos que representam o tubo 3D, nós calculamos o tamanho de uma aresta de pixel em espaço de mundo na posição de P_i. Se o raio r da seção de corte em P_i é menor que a aresta de um pixel, nós setamos o valor de r com o tamanho da aresta do pixel correspondente. Essa abordagem combinada com o LOD descrito na seção anterior resolve o problema do aliasing.

4.4 Implementação

A implementação consiste em duas partes. A primeira de pré-processamento onde devemos calcular as normais e bi-normais que serão utilizadas no shader. A segunda parte consiste na configuração do Tessellator para a rendereização dos tubos baseando-se nos dados pré-processados.

4.4.1 Pré-processamento

Para passar as informações de normal, bi-normal e ponto central para cada secção de corte do tubo, nós precisamos fazer um pré-processamento em $O(n)$ como explicado anteriormente. Essas informações são passadas para a GPU como informações dos pontos de controle para serem usadas pelo domain shader. Cada invocação do tessellator constrói um quad que é transformado em um parte de um tubo 3D pelo domain shader. O tessellator é capaz de subdividir um quad até 64 vezes em cada eixo.

Como demonstramos na Figura 4.4E, um eixo do domínio do quad tem o fator de tecelagem máximo. Cada linha deste eixo é transformada para seguir a seção de corte definida pela sequência de pontos que representa o caminho do tubo. Os valores de tecelagem do outro eixo é flexível para uso de LOD como mostramos na Figura 4.4E.

4.4.2 Configurando o Input Assembler

Em um domínio de quad o tessellator é capaz de criar 64 linhas em cada eixo. Para ter o máximo de aproveitamento da capacidade do tessellator nós devemos passar 64 posições, normais e bi-normais diferentes para o tessellator para cada quad subdividido. Porém, existe uma limitação do Input Assembler que só aceita 32 pontos de controle por primitiva. Esse problema pode ser contornado passando duas posições, normais e bi-normais por ponto de controle.

Então o Input Assembler é configurado para aceitar um patch com 32 pontos de controle e cada um desses pontos de controle deve carregar a informação para posicionar corretamente duas seções de corte. O código 4.1 mostra a informação que precisamos para cada ponto de controle.

Código 4.1: Informações de cada ponto de controle

```
1 struct VS_CONTROL_POINT_INPUT
2 {
3     float3 vPosition   : POSITION;
4     float3 vBinormal   : TEXCOORD0;
5     float3 vNormal     : NORMAL0;
6     float3 vPosition2  : TEXCOORD1;
7     float3 vBinormal2  : TEXCOORD2;
8     float3 vNormal2    : TEXCOORD3;
9 };
```

Esses pontos de controle são os únicos dados que são transmitidos via vertex buffer para a GPU. Escolhendo um domínio de quad o tessellator é capaz de criar 8192 triângulos. Além disso, um parâmetro que determina o raio pode opcionalmente ser passado.

4.4.3 A parte constante do Hull Shader

A parte constante do Hull Shader é o lugar no novo pipeline onde o fator de tecelagem deve ser passado para o tessellator, além disso, algoritmos de tecelagem adaptativa também devem ser calculados aqui. O fator de tecelagem

está no intervalo [1..64]. Existem seis fatores de tecelagem no domínio de um quad, quatro deles são as arestas e dois para o interior do domínio (um para cada eixo do quad).

O algoritmo proposto transforma cada linha do domínio do quad em um círculo, como explicado anteriormente. Para tirar o máximo de proveito do poder de tecelagem, é preciso maximizar o número de linhas em um eixo do domínio. Consequentemente, para atingir este objetivo, é preciso setar o valor de tecelagem de duas arestas e um eixo do interior para 64. Essa configuração garante que transformaremos o máximo de linhas em seções de tubo. Os três parâmetros restantes podem ser variados adaptativamente através da distância da câmera ou o tamanho da aresta em espaço de tela. Além disso, a posição da câmera deve ser passada como um parâmetro constante para o shader. A Figura 4.5 mostra o mesmo tubo com dois LODs diferentes configurados pelo Hull Shader.

O código 4.2 mostra a parte constante do Hull Shader.

Código 4.2: Parte constante do Hull Shader

```
1  HS_CONSTANT_DATA_OUTPUT ConstantHS(
2  InputPatch<VS_CONTROL_POINT_OUTPUT,
3  INPUT_PATCH_SIZE> ip,
4  uint PatchID : SV_PrimitiveID )
5  {
6     HS_CONSTANT_DATA_OUTPUT Output;
7  //these factors should be configured according
8  //to an adaptive tessellation
9  //parameter(camera or edge screen-space size)
10 Output.Edges[0]= Output.Edges[2]= Output.Inside[1] =
11 g_fTessellationFactor;
12 //maximum tessellator factor for one axis
13 Output.Edges[1] = Output.Edges[3] = Output.Inside[0] = 64;
14
15 return Output;
16 }
```

4.4.4 A parte principal do Hull Shader

Uma boa vantagem de gerar a malha do tubo na GPU é que o raio pode ser setado dinamicamente. Com esse recurso é possível resolver os problemas de aliasing. A parte principal do Hull Shader é executada uma vez por ponto de controle, é nesta parte que controlamos dinamicamente o raio de cada parte

do nosso quad (que será transformado posteriormente em uma seção do tubo no Domain Shader).

Seja P_i o ponto central de uma seção (existem 2 desses pontos por ponto de controle como explicado anteriormente). Primeiro nós precisamos achar a profundidade corrente de P_i em espaço de tela. Multiplicando P_i pela matriz de *ViewProjection* obtemos P_i em espaço de tela (P_{screen}). A profundidade é obtida dividindo a coordenada Z de P_{screen} pela coordenada W do mesmo. Isso é mostrado pelas equações 4-3 e 4-4.

$$P_{screen} = P * ViewProjection \tag{4-3}$$

$$P_{depth} = P_{screen_z}/P_{screen_w} \tag{4-4}$$

Para fins de simplicidade assumimos um canvas com aspecto de 1. Seja res a resolução do canvas. $S_1 = (0, 0, P_{depth}, 1)$ e $S_2 = (2/res, 0, P_{depth}, 1)$ representam o comprimento de um pixel em espaço de tela. Nosso objetivo é achar o tamanho do pixel em espaço do mundo na profundidade P_{depth}. Multiplicando S_1 e S_2 pela inversa da $ViewProjection$ e dividindo pela coodenada W conseguimos os dois pontos em espaço de mundo. O tamanho do pixel em espaço de mundo é o comprimento do vetor formado por esses dois pontos, como mostra a equação 4-5.

$$\left\| \frac{S_2 * ViewProj^{-}1}{S_{2w}} - \frac{S_1 * ViewProj^{-}1}{S_{1w}} \right\| \tag{4-5}$$

O código 4.3 mostra a parte principal do Hull Shader.

Código 4.3: Parte principal do Hull Shader

```
1  [domain("quad")]
2  [partitioning("integer")]
3  [outputtopology("triangle_cw")]
4  [outputcontrolpoints(32)]
5  [patchconstantfunc("ConstantHS")]
6  HS_OUTPUT HS( InputPatch<VS_CONTROL_POINT_OUTPUT, 32> p,
7                uint i : SV_OutputControlPointID,
8                uint PatchID : SV_PrimitiveID )
9  {    HS_OUTPUT Output;
10     Output.vPosition = p[i].vPosition;
```

```
11      Output.vBinormal = p[i].vBinormal;
12      Output.vNormal = p[i].vNormal;
13      Output.vPosition2 = p[i].vPosition2;
14      Output.vBinormal2 = p[i].vBinormal2;
15      Output.vNormal2 = p[i].vNormal2;
16      float4 pScreen = mul(float4(p[i].vPosition,1),
17                                  g_mViewProjection);
18      float pDepth = pScreen.z/pScreen.w;
19      float4 S2 = float4(1.0f/res,0,pDepth,1);
20      float4 S1 = float4(0,0,pDepth,1);
21      float4 worldPixel2 = mul(S2,g_mInvViewProjection);
22      float4 worldPixel1 = mul(S1,g_mInvViewProjection);
23      worldPixel2.xyz = worldPixel2.xyz/worldPixel2.www;
24      worldPixel1.xyz = worldPixel1.xyz/worldPixel1.www;
25      float PixelSizeWorld = length(worldPixel2.xyz
26                                          - worldPixel1.xyz);
27
28      if(radius <= 2* PixelSizeWorld)
29          radius = 2* PixelSizeWorld;
30      Output.vRadius = radius;
31      return Output;
32  }
```

4.4.5 O Domain Shader

O Domain Shader recebe coordenadas UV no intervalo $[0..1]^2$. Essas coordenadas especificam onde, no domínio do quad, cada vértice gerado pelo tessellator está localizado. Cada invocação do Domain Shader corresponde a um vértice do quad tecelado. Primeiramente, precisamos transformar cada linha do quad em um círculo. Seja p'_i um ponto do círculo no plano $y = 0$ e r o raio do tubo(o raio deve vir como saída do Hull Shader). A transformação é feita de acordo com as equações 4-6 e 4-7.

$$\theta = 2\pi V \tag{4-6}$$

$$p'_i = (rcos\theta, 0, rsin\theta) \tag{4-7}$$

Agora a posição (P_i), normal ($\vec{N}$) e binormal ($\vec{B}$) devem ser obtidas através dos dados passados por ponto de controle. Como cada ponto de controle contém a informação de duas seções de tubo, nós devemos pegar a informação correta para cada invocação do Domain Shader. Para isso, deve-se fazer uma multiplicação da coordenada U por 63 e uma escolha através do operador de módulo.

Agora deve-se posicionar e orientar a seção de corte C do tubo. Converte-se p'_i para p''_i de acordo com a equação 4-2. Depois p''_i é transformado pelas matrizes de view e projection e setado como uma das saídas do Domain Shader.

Outra saída do Domain Shader é a normal do vértice gerado, $\vec{N}$. A normal pode ser facilmente achada usando a equação 4-8

$$\vec{N} = (F_p - P) / \|F_p - P\| \tag{4-8}$$

O código 4.4 mostra o Domain Shader.

Código 4.4: Implementação do Domain Shader

```
1  [domain("quad")]
2  DS_OUTPUT DS( HS_CONSTANT_DATA_OUTPUT input,
3                       float2 UV : SV_DomainLocation,
4                       const OutputPatch<HS_OUTPUT, 32> p)
5  {
6      DS_OUTPUT Output;
7      float v = UV.y;
8      float u = UV.x;
9      float pi2 = 6.28318530;
10     float theta = v*pi2;
11     float sinTheta, cosTheta;
12     sincos(theta, sinTheta, cosTheta);
13     int index = 63*u;
14     float3 N,B,P;
15     if( index % 2 != 0 )
16     {
17         index = (int)(index/2.0f);
18         N = p[index].vNormal2;
19         B = p[index].vBinormal2;
20         P = p[index].vPosition2;
21     }
22     else
23     {
```

```
24            index = (int)(index/2.0f);
25            N = p[index].vNormal;
26            B = p[index].vBinormal;
27            P = p[index].vPosition;
28        }
29
30        int radius = p[splineIndex].vRadius;
31        float3 C = float3(raio*cosTheta,0,raio*sinTheta);
32        float3 worldPos;
33
34        worldPos.x = P.x + C.x*N.x + C.z*B.x;
35        worldPos.y = P.y + C.x*N.y + C.z*B.y;
36        worldPos.z = P.z + C.x*N.z + C.z*B.z;
37
38        float3 normal = normalize(worldPos - cylinderPos);
39
40        Output.vPosition = mul( float4(worldPos,1),
41                                g_mViewProjection );
42        Output.vNormal = normal;
43        Output.vWorldPos = float4(worldPos,1);
44
45        return Output;
46    }
```

4.5 Resultados

Nossos testes foram executados em um Intel Core i7 920 com uma Nvidia GTX480 e 6GB de RAM. Foi feita uma comparação entre abordagens onde todos os triângulos foram passados da CPU para a GPU com o tessellator desabilitado contra nosso método proposto usando o tessellator. Nenhuma estrutura de aceleração ou algoritmo de otimização como frustum culling foi usado. O objetivo dos testes foi medir a eficiência obtida com a troca de largura de banda por operações aritméticas na GPU. É importante ressaltar que otimizações posteriores como evitar a subdivisão de patches virados de costas para a câmera e tecelagem adaptativa de acordo com a aresta em espaço de tela para evitar triângulos menores que um pixel são possíveis no pipeline do tessellator.

A Tabela 4.1 mostra os resultados comparando o frame rate com e sem a técnica proposta, tanto usando a nossa proposta de melhora do aliasing quanto não o utilizando. Os resultados mostram que o algoritmo proposto

Milhões de Triângulos	FPS - técnica CPU	FPS - Tessellator com AA	FPS - Tessellator sem AA
184,3	0	10	11
28,6	1	59	60
12,3	64	102	104
11	71	111	114
9,8	79	124	130
8,6	89	136	147
7,4	99	145	167
6,1	120	178	194
4,9	141	210	231
3,7	181	247	286
2,5	242	320	373
1,2	335	418	500
0,61	518	621	720
0,3	730	835	911
0,12	930	999	1050
0,061	970	1050	1112
0,0061	1100	1111	1135

Tabela 4.1: Comparação de FPS das técnicas

apresenta um ganho significativo sobre a abordagem sem o uso do tessellator. Além disso, fomos capazes de renderizar até 184 milhões de triângulos em uma taxa interativa de quadros por segundo. Usando a abordagem de CPU com informações de pelo menos posição e normal por vértice resultaria em um vertex buffer de tamanho proibitivo (13.2GB) que seria passado da CPU para a GPU enquanto nossa abordagem usa apenas 184.32MB para o mesmo número de triângulos como ilustra a Tabela 4.2. Não obstante, nossa solução para redução do aliasing se mostrou bastante eficiente com uma pequena queda de performance comparada com a solução com aliasing. A Figura 4.7 mostra um gráfico com a porcentagem de ganho do nosso algoritmo contra a abordagem tradicional em CPU.

4.6 Melhorando o consumo de memória por frame ainda mais

A técnica explicada acima é um método direto e balanceado de se renderizar tubos 3D. Apesar de não ser nosso caso, uma aplicação pode requerer um consumo de memória por frame ainda menor. Ao invés de passar todos os dados (Posição, Normal, Bi-Normal) através dos pontos de controle, pode-se usar somente um ponto de controle por domínio a ser tecelado e mapear uma textura com as informações de posição, normal e bi-normal. O único ponto de controle por quad traria codificada a posição onde a textura deveria

Memória técnica CPU (em MB)	Nossa técnica(em MB)
13270	184,32
2059	28,8
886	12,16
792	10,88
706	9,6
619	8,32
533	7,68
439	6,4
353	5,12
266	3,84
180	2,56
86	1,28
44	0,64
22	0,32
9	0,16
4	0,08
0	0,00

Tabela 4.2: Comparação consumo de largura de banda CPU-GPU

ser consultada para obter os dados. Essa abordagem iria reduzir o consumo de memória em um fator de 64. De acordo com as Tabelas 4.1 e 4.2, esse método não traria muito impacto a partir da segunda linha até o fim da tabela, porém, ele pode trazer um aumento de performance caso a aplicação necessite renderizar mais de centenas de milhões de triângulos.

4.7 Limitações

A maior limitação desta técnica é que ela só é compatível com as placas de vídeo que suportam o DirectX11 e OpenGL4. Pode-se pensar em implementar a técnica usando instanciamento de geometria nas placas antigas. Essa abordagem funciona, porém existe um problema grave de performance com o rasterizador. O controle de LOD através de instanciamento de geometria não é tão flexível quanto o pipeline novo e caso a aplicação comece a criar muitos triângulos menores do que um pixel, o rasterizador torna-se rapidamente o gargalo da aplicação.

Outro problema é que a solução para evitar o aliasing aumenta o raio dos tubos em espaço de mundo. Caso se tenha uma cena que não é composta apenas de tubo, como a Figura 4.1, um ajuste de raio máximo permitido é necessário para evitar a criação de tubos fora de proporção. Esta última limitação não é severa, fomos facilmente capazes de achar valores de raio que previnem o aliasing e mantém a cena com tamanhos proporcionais.

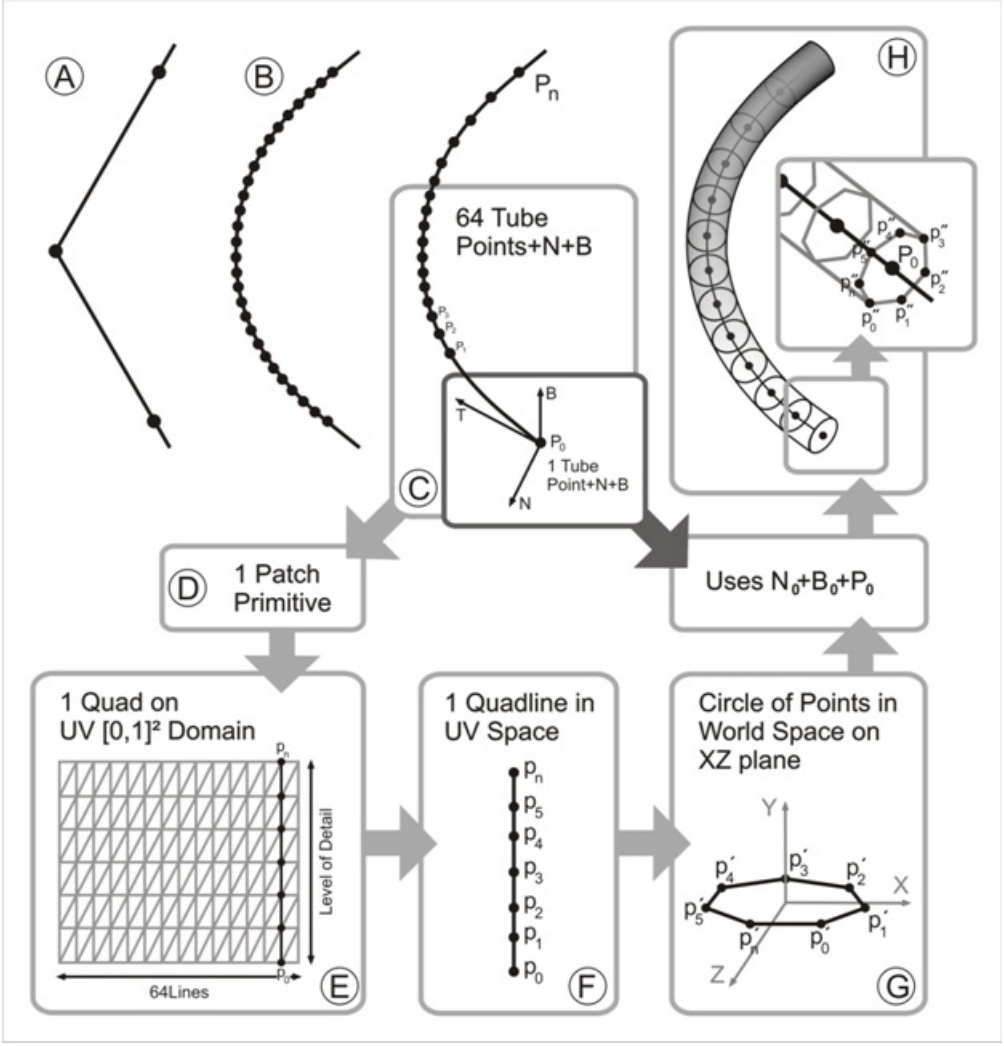

Figura 4.4: A) Um caso de sequência de pontos ruim. B) A sequência ruim após a aplicação da interpolação de Catmull-Rom. C) A seleção de pontos da sequência e cálculo das tangentes, normais e bi-normais. D) Cada 64 pontos implica em uma primitiva(patch). E) O patch é uma primitiva do tipo quad com 64xLOD linhas. F) Uma linha do quad com o número de pontos definido pelo LOD. G) Cada ponto da linha é transformado em um círculo no espaço 3D no plano y=0. H) Usando as normais e bi-normais cada ponto é transformado do círculo para a seção de corte do tubo.

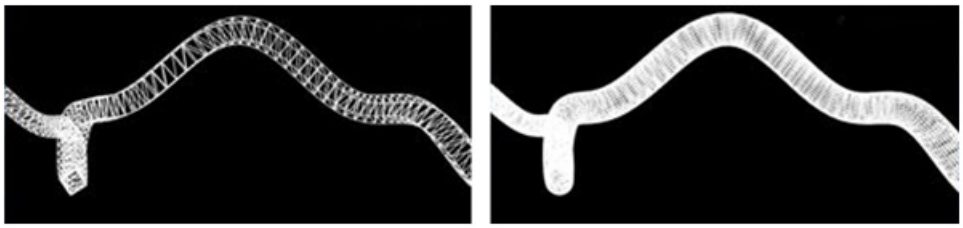

Figura 4.5: Esquerda - Tubo com LOD baixo. Direita - Tubo com LOD alto.

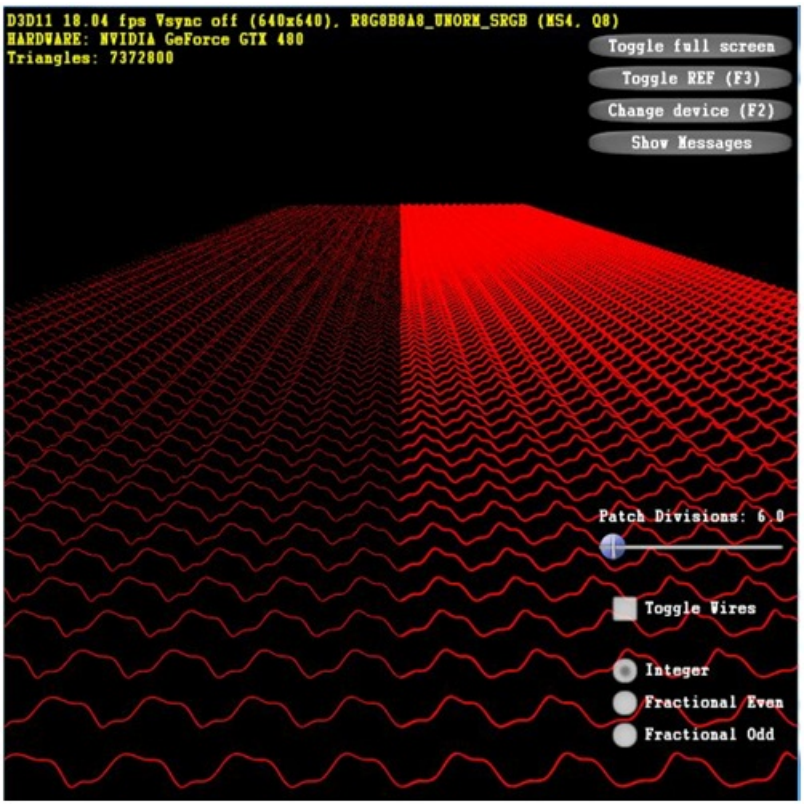

Figura 4.6: Mesma cena: na esquerda com 16x GPU anti-aliasing. Na direita com a correção de aliasing proposta.

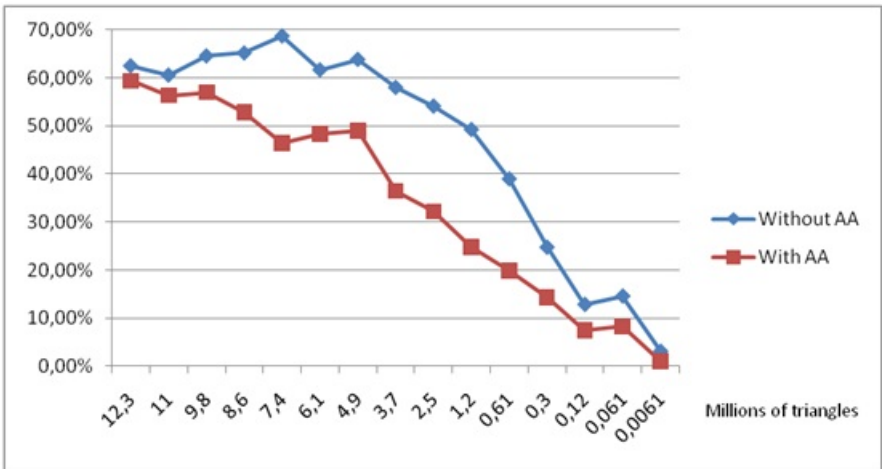

Figura 4.7: Gráfico mostrando a porcentagem de ganho em FPS do nosso algoritmo com e sem a correção de aliasing proposta comparado com a abordagem em CPU sem anti-aliasing.

5
Renderização de terreno usando processo local paralelo em GPU

5.1
Introdução

Renderização de terrenos com taxas interativas é essencial para aplicações GIS, simuladores de vôo, simuladores de veículos terrestres e jogos. Contudo, apesar dos avanços na última década, alcançar o equilíbrio entre alta qualidade de imagem e bom tempo de processamento é ainda um desafio. Um estudo recente [Asirvatham and Hoppe, 2005] atesta uma taxa de 90 fps em um vôo interativo sobre um mapa de altura com mais de 20 bilhões de amostras (216.000 x 93.600), porém, severas restrições são aplicadas. Extrema fidelidade visual e LOD automático são alguns dos recursos disponibilizados pelo tessellator para a nova geração de placas gráficas, porém, o uso do tessellator não é trivial e técnicas novas de LOD são necessárias.

Terrenos são tipos de modelos representados por mapas de altura - um grupo de amostragens de altura sobre um domínio plano. Uma forma simples de renderização de terreno é criar uma malha de polígonos regulares e mover seus vértices de acordo com um mapa de altura, o que não é eficiente nem escalável.

Algoritmos para uma renderização eficiente de terrenos já foram propostos por mais de uma década, a maioria envolvendo estruturas hierárquicas associadas com LOD e técnicas de descarte de geometria. Bons estudos sobre o assunto podem ser encontrados em [Floriani et al., 1996] e [Pajarola and Gobbetti, 2007].

A avaliação é necessária localmente, pois uma parte do terreno pode ter uma altura constante e não é preciso subdividi-la. Contudo, os vértices devem ser reduzidos de forma adequada, pela possibilidade de porções de terreno apresentarem variação de altura em alta frequência - o que exige uma malha refinada para uma representação confiável. A malha do terreno deve ser adaptada automaticamente para ser refinada ou não, de acordo com o ponto de vista da câmera. Isso exige um controle de LOD de acordo com o ponto de

vista do observador e que mantenha a conectividade consistente (ou seja, sem quebras na malha) e animação suave (sem artefatos salientes ou saltitantes). Porém, a maioria das técnicas de divisão de vértices e colapso de arestas (predominante em técnicas de LOD) exigem algoritmos de CPU sequenciais para atualização de suas estruturas de dados e, como consequência, são difícies de serem implementadas eficientemente, apesar de algumas exceções, como em [Losasso, 2004]. Há também a possibilidade de utilizar o Geometry Shader, que tem acesso aos vértices vizinhos das primitivas e pode atualizar estruturas de dados que estão na memória de vídeo. Todavia, essas abordagens são muito complexas, pois a continuidade da malha precisa ser preservada.

Controle de LOD paralelo de acordo com o observador baseado no uso de Geometry Shaders foi proposto recentemente para malhas arbitrárias [Hu et al., 2010b]. Esse controle pode ser adaptado para terrenos, mas a complexidade da solução não é reduzida significativamente.

Escalabilidade é outro ponto crítico quando se trata de grandes mapas de altura. Nesse caso, durante uma navegação aproximada a malha deve ter o máximo de refinamento possível (um quadrado para cada amostra), causando um overhead crítico. Algoritmos de renderização de terrenos devem apresentar alta escalabilidade. É indispensável a utilização de alguma estrutura hierárquica para navegar em um terreno com escalas altamente variáveis. Entretanto, processos de construção e acesso a essas estruturas de dados significam menor eficiência computacional e, muitas vezes, um obstáculo para a comunicação entre CPU e GPU. Outra forma de escalabilidade se refere à capacidade de algoritmos paralelos usarem o máximo do número de unidades de processamento que se tornaram disponíveis em consecutivas gerações de GPU.

Neste capítulo apresentamos uma nova técnica para uma eficiente renderização de terrenos usando uma técnica de LOD contínua e dependente do observador com base no uso do tessellator. Nossa técnica é baseada em um processamento local paralelo, ou seja, os resultados de um patch do terreno não dependem dos resultados obtidos em outros patches. A técnica proposta não usa estrutura hierárquica para fácil implementação em GPU.

5.2 Trabalhos Relacionados

Técnicas de multirresolução para renderização de terrenos podem ser classificadas através de três classes básicas:

1. (i) Modelos de Multirresolução através de malhas irregulares [Floriani et al., 1996], e

2. (ii) Modelos de multirresolução que exploram certa semi-regularidade de dados [Pajarola and Gobbetti, 2007].

3. (iii) Modelos de multirresolução híbrida com estrutura regular para malhas irregulares [Toledo et al., 2001].

Seguindo a classificação proposta por [Pajarola and Gobbetti, 2007], a segunda classe de modelos pode ser agrupada em três linhas:

1. Triangulações diretas [Falby et al., 1993] e malhas regulares sobrepostas [Losasso, 2004] e [Asirvatham and Hoppe, 2005],

2. Triangulações baseadas em árvores [Lindstrom et al., 1996] e [Duchaineau et al., 1997].

3. Triangulações agrupadas em cluster [Schneider and Westermann, 2006] e [Levenberg, 2002].

A primeira linha utiliza grids regulares, o que a torna escalável, simples de implementar e feita sob medida para hardwares gráficos. A segunda linha utiliza malhas com conectividade semi-regular, que se baseia em estruturas de dados mais poderosas. A terceira trabalha com porções contínuas da malha, regulares ou semi-regulares, formando patches quadrados ou triangulares. Essas partes são teceladas dentro da GPU com uma comunicação mínima com a CPU.

O modelo proposto neste artigo utiliza um grid 2D regular de patches quadrados (chamados tiles) que são tecelados pela GPU de forma independente. Esta abordagem classifica o modelo proposto como uma triangulação em cluster, porém com características de simplicidade encontradas em triangulações diretas. A principal diferença entre nosso trabalho e todas as outras técnicas é que nosso sistema é o primeiro a fazer renderização de terrenos com LOD em tempo real sem utilizar nenhum tipo de estrutura hierárquica de dados.

Estratégias de LOD não específicas para terrenos também são aplicadas para casos em terrenos. Muitas dessas estratégias contam com o Geometry Shader, e podem ser muito eficientes. Há métodos baseados em GPU [Hu et al., 2010b],[Decoro and Tatarchuk, 2007] e [Ji et al., 2006]. Apesar dessas estratégias baseadas em GPUs serem muito eficientes, elas contam com estruturas hierárquicas complexas e condições para atualizar a malha e evitar fraturas nela.

A maioria das novas propostas para modelos baseados em multirresolução na GPU evitam a fase de pré-processamento, mas ainda exigem o percorrimento de toda a estrutura a cada frame. Uma notável exceção é o trabalho

realizado por [Hu et al., 2010b], que, de qualquer forma, usa uma estrutura de dados hierárquica - o que faz dele mais complexo do que nosso sistema. Percorrer toda a estrutura de dados a cada frame não é apropriado quando o terreno muda dinamicamente - um problema crítico em jogos 3D. Em nosso modelo, o terreno pode ser deformado dinamicamente por manipulação direta do mapa de altura sem nenhuma sobrecarga extra para o sistema.

Apesar de todos os avanços prévios de tecelagem em hardware, pesquisadores de técnicas de LOD para terrenos não parecem estar cientes dos problemas enfrentados pelos desenvolvedores de jogos para utilizar isso funcionalmente. Acreditamos que a razão para esta falta de popularidade são os problemas popping de artefatos que são causados devido ao uso trivial e direto do hardware de tecelagem. Nossa técnica propõe um modelo para superar estas dificuldades através de controles simples e critérios de erro. Podemos encontrar alguma similaridade de nosso critério com técnicas propostas por [Tatarchuk et al., 2010]. No entanto, nossa abordagem é mais ampla do que a encontrada nesta referência, pois calculamos uma subdivisão mínima e máxima para as arestas e também para o centro de cada patch.

Nosso modelo estabelece um framework claro para o uso de renderização de terrenos com LOD por processo local paralelo.

No melhor de nosso conhecimento, nenhum outro trabalho atual na literatura propõe um meio de implementar renderização de terreno robusta e eficiente com LOD e sem uso de estruturas de dados hierárquicas. Além disso, nenhum outro trabalho estabelece uma estrutura clara para processamento local paralelo na renderização de terrenos.

5.3 Visão Geral

Processar tiles de forma contínua e em uma abordagem view-dependent é um problema desafiador, especialmente se estas malhas não puderem ser refinadas do mesmo modo. Devemos garantir a continuidade da malha. Arestas de tiles adjacentes devem ter o mesmo número de subdivisões, sem levar em consideração o número de sub-quads que tem cada tile adjacente.

Consequentemente, a idéia é tecelar o tile independentemente dos resultados dos outros tiles, estabelecendo um verdadeiro processamento local paralelo.

Podemos alcançar essa independência definindo uma malha regular na qual uma aresta compartilhada entre dois tiles tem o mesmo valor do fator de tecelagem (chamado TessFactor).

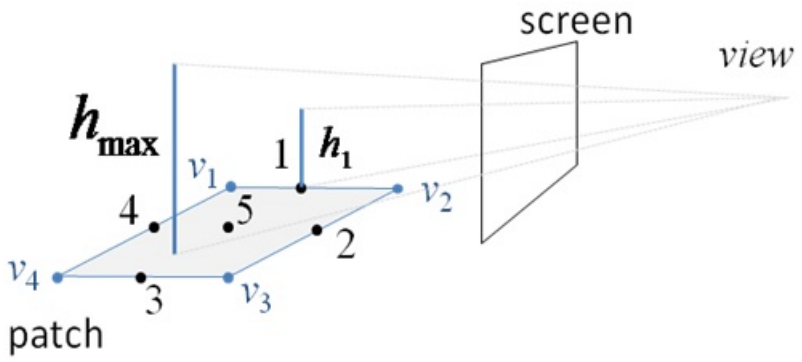

Figura 5.1: Pontos de referência de mosaico (quatro pontos no meio dos contornos e um no centro do pedaço). V_i é um vértice do patch.

Definimos pontos de referência de tecelagem Figura 5.1 e o fator de tecelagem para estes pontos, dado por:

$$T_i = f(view, \nabla^2 h) \quad i = 1, .., 4 \tag{5-1}$$

onde h_i é o valor de altura correspondendo ao ponto $i, view$ é a posição da câmera e $\nabla^2 h$ é a derivada de segunda ordem do mapa de altura $h(u, v)$ correspondendo ao tile. A posição de um ponto de referência na aresta corresponde ao centro da aresta. O quinto ponto está no centro do tile e sua tecelagem é dada por: $T_5 = f(h_{max}, view, \nabla^2 h)$

onde h_{max} é o valor máximo de altura para os modelos dentro do tile.

Um dos princípios por trás da equação 5-1 é que a relação entre h_i e view deve estabelecer um valor mínimo de TessFactor (T_{min}) que garante uma tecelagem precisa do ponto de vista da câmera. Por exemplo, se a câmera está muito longe do patch e h_i está baixo, nenhuma subdivisão será necessária em torno do ponto i. Outro princípio que ronda a equação 5-1 é que $\nabla^2 h$ pode ser utilizado para estabelecer um limite superior para o TessFactor (T_{max}) e garantir que os patches não sejam divididos em excesso. Por exemplo, nenhuma subdivisão é necessária para um terreno plano ($\nabla^2 h = 0$). Em outro aspecto, penhascos íngremes exigem altos fatores de tecelagem. De acordo com esses princípios, a equação 5-1 deve calcular um valor de TessFactor no intervalo $[T_{min}, T_{max}]$.

O cálculo do TessFactor de uma aresta depende da função $\nabla^2 h$ do tile adjacente. Contudo, esta restrição não destrói a natureza do processo local no método proposto. Além disso, consideramos algumas simplificações na equação 5-1 que transformam o cálculo de influência de $\nabla^2 h$ em um simples

procedimento.

5.4 Análise do mapa de altura

5.4.1 Idetinficação de fraturas

Para ser representado, um terreno plano necessitaria apenas de uma única primitiva. No entanto, terrenos reais têm irregularidades (como em penhascos íngremes) que necessitam de mais geometria para serem renderizados corretamente. Neste documento, chamamos estas irregularidades de "fratura".

O processo para encontrar as fraturas consiste em fazer a média das derivadas do mapa de altura nas direções horizontais e verticais Figura (5.2). O cálculo Laplaciano $\nabla^2 h$ não é aceitável porque a Laplaciana é extremamente sensível a ruídos. Além disso, desejamos um modo mais simplificado para levar em conta as fraturas do terreno. Propomos uma combinação de dois operadores Sobel. A magnitude do vetor gradiente calculado pelo operador Sobel sobre o mapa de altura representa a variação de altura em um pixel dado. Contudo, o gradiente não é suficiente para determinar uma fratura.

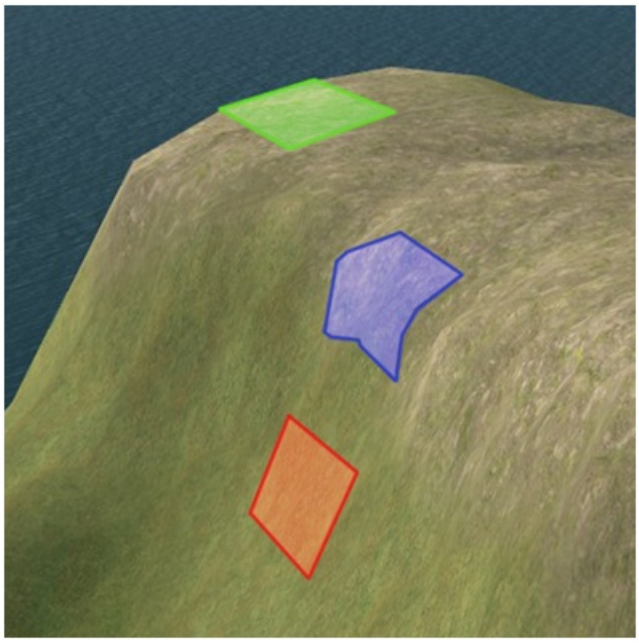

Figura 5.2: A área no topo possui primeira e segunda derivada pequenas. A parte inferior tem maior valor de primeira derivada e menor de segunda derivada. Já a área do meio apresenta maior segunda derivada. A segunda derivada denota locais que exigem refinamento.

Locais que devem ser interpretados como fraturas apresentam mudança na variação de altura. No algoritmo proposto, estimamos este valor aplicando

um operador Sobel sobre o campo de gradiente obtido através da aplicação do primeiro operador Sobel. Os novos valores de inclinação podem ser visualizados como um mapa de aceleração da altura (height acceleration map) ou HAM, que pode ser utilizado para identificar fraturas. Na fase de renderização, nosso algoritmo precisa consultar acelerações de altura com o objetivo de determinar o refinamento máximo que será detalhado mais adiante.

5.4.2 Tamanho do tile

A dimensão do tile é determinada pelo usuário, que decide quantas amostras do mapa de altura um sub-tile deve cobrir. O caso mais refinado seria quando tivermos um quad para cada amostra. Em casos reais, um mapa de altura nem sempre exige que toda amostra tenha um quad para representá-la. Porém, se a densidade de divisões é maior que a densidade de vértices da área tecelada, podem ocorrer algumas impressões indesejadas e artefatos. Estes efeitos indesejados sempre ocorrem quando a frequência de amostragem (vértices do quad) é muito diferente da frequência de sinal (densidade de texels).

Em nosso modelo, eliminamos esses problemas considerando que um quad sempre cobre um texel.

5.5 View-dependent LOD

5.5.1 Refinamento máximo: T_{max}

Nossa primeira preocupação é não tecelar demais partes que não necessitam de refinamento. Intuitivamente, um terreno plano não precisa de subdivisão, logo, todo tile que contêm uma parte quase plana do terreno não deve ser subdividido quando a câmera se aproxima. Partes planas podem ser identificadas pelo mapa de aceleração de altura (HAM). Além disso, precisamos identificar fraturas associada aos tiles também consultando o HAM. Nesses dois casos, nós não precisamos de uma expressão exata para a equação 5-1, mas somente uma proproção razoável entre a aceleração e o TessFactor. Propomos a equação 5-2.

$$T_{max} = g(HAM_{max}) \tag{5-2}$$

Onde HAM_{max} é o valor máximo para aceleração de altura associada ao tile e g pode ser uma função logaritma ou a função de raiz quadrada. A Figura 5.3 ilustra a equação 5-2. Na prática, HAM_{max} está dentro do intervalo que não contém os extremos do domínio

A equação 5-2 não oferece nenhum obstáculo à complexidade do algoritmo proposto. Quando o ponto de referência de tecelagem está em uma aresta, o algoritmo seleciona o valor de T_{max} entre dois tiles adjacentes.

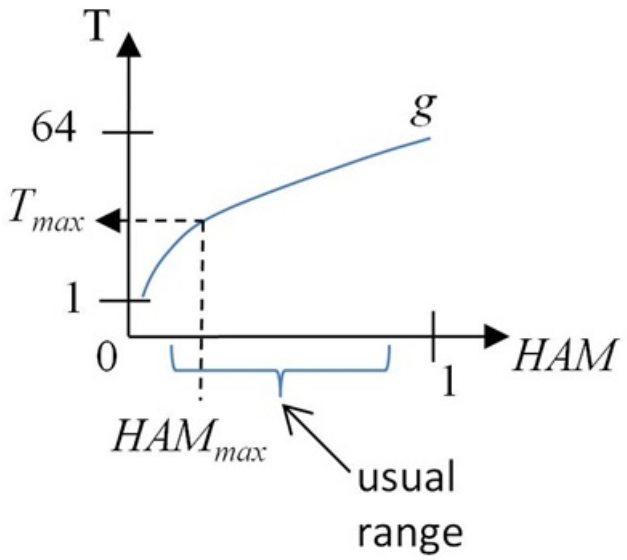

Figura 5.3: TessFactor como uma função de HAM. g não é definido por $T < 1$, porque o intervalo do TessFactor é $[1, 64]$.

5.5.2 Refinamento mínimo: T_{min}

Outro problema a ser resolvido é o refinamento minimo requerido quando se está a uma longa distância. Um exemplo real é uma parte do terreno que contém um pico. Um pico não pode desaparecer no horizonte, mesmo que a câmera esteja muito longe dele. Portanto, precisamos estabelecer um refinamento mínimo por tile (T_{min}) baseado no ângulo entre a câmera e a normal do tile.

T_{min} é obtido a partir do controle do erro de aproximação. É comum na literatura avaliar o erro de aproximação de uma malha tesselada através de seu desvio no vértice. [Duchaineau et al., 1997] e [Lindstrom and Pascucci, 2002] mediram este desvio no espaço projetado. Nós também utilizamos o espaço projetado, mas de um modo diferente. A Figura **??** ilustra o processo proposto, onde sucessivamente interpolamos o ponto que deve corresponder à maior altura h abrigada pelo tile. Devemos ressaltar que este valor máximo

(h) não necessariamente ocorre em um dos pontos de referência de tecelagem Figura 5.4. Em cada fator de tecelagem, o ponto move um pequeno passo em direção ao valor correto. O desvio é monitorado no espaço projetado e o processo acaba quando o desvio está menor do que o erro permitido ϵ . Se o processo é recursivo, a profundidade da recursão é o valor de T_{min}.

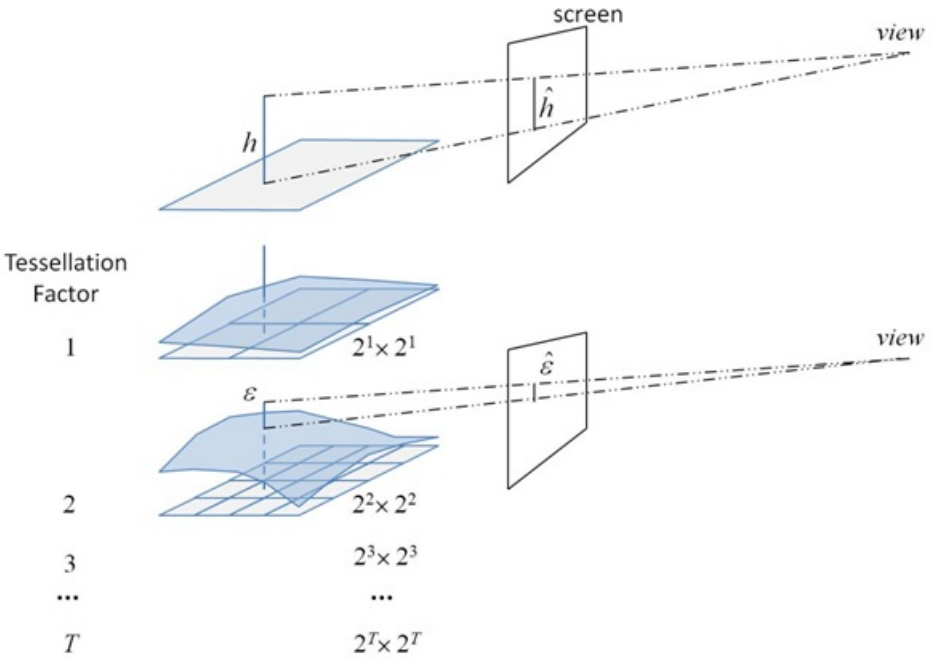

Figura 5.4: Processo de cálculo do T_{min} que para quando o erro projetado E é igual ao erro permitido ϵ. h é o valor máximo de altura.

De qualquer forma, o processo recursivo não é necessário, pois uma boa estimativa de T_{min} pode ser obtida observando que $\epsilon/\hat{h}$ é proporcional a 2^T . Sem perda de generalidade podemos assumir que

$$\epsilon/\hat{h} = 2^T \tag{5-3}$$

E substituir a equação 5-1 pela seguinte equação:

$$T_{min} = \log_2 \epsilon/\hat{h} \tag{5-4}$$

5.5.3 Distância da câmera

Agora, o valor de TessFactor para um ponto de referência de tecelagem i é obtido a partir do cálculo baseado na distância da câmera d, mantendo-

os dentro do limite entre valores máximos e mínimos. Aqui assumidos uma proporcionalidade inversa entre TessFactors e distâncias da câmera. Além disso, supomos que há um valor de distância de câmera (d_{max}) que se $d < d_{max}$, então T deve ser igual a T_{max}. Considerando essas suposições, propomos a seguinte equação para substituir a equação 5-1.

$$T = \frac{T_{max} {d_{max}}^2}{d^2} \qquad se \;\; d \in [d_{max}, d_{min}] \tag{5-5}$$

$$T = T_{max} \qquad se \;\; d < d_{max} \tag{5-6}$$

$$d_{min} = \frac{\sqrt{T_{max}}}{\sqrt{T_{min}}} d_{max} \tag{5-7}$$

A figura 5.5 ilustra as equações 5-5, 5-6, 5-7.

5.6 Implementação

Para ser tecelada posteriormente, uma malha não refinada precisa ser transferida para a GPU. O tamanho do grid é calculado com base no tamanho do tile como relatado anteriormente.

Além da posição, precisamos também transferir as coordenadas de textura do height map correspondendo a cada vértice do quad.

5.6.1 Mapa de aceleração de altura

Dado um terreno representado por um mapa de altura H, nosso primeiro objetivo é encontrar a taxa de mudança de cada texel em H. Para obter este valor, utilizamos uma abordagem baseada no operador [Sobel and Feldman, 1968].

Dois Kernels 3x3 são convoluídos com H para calcular aproximações de derivativas - um para mudanças horizontais e outro para verticais. Assumindo que e são as aproximações das derivadas horizontais e verticais correspondentes, o cálculo é como segue:

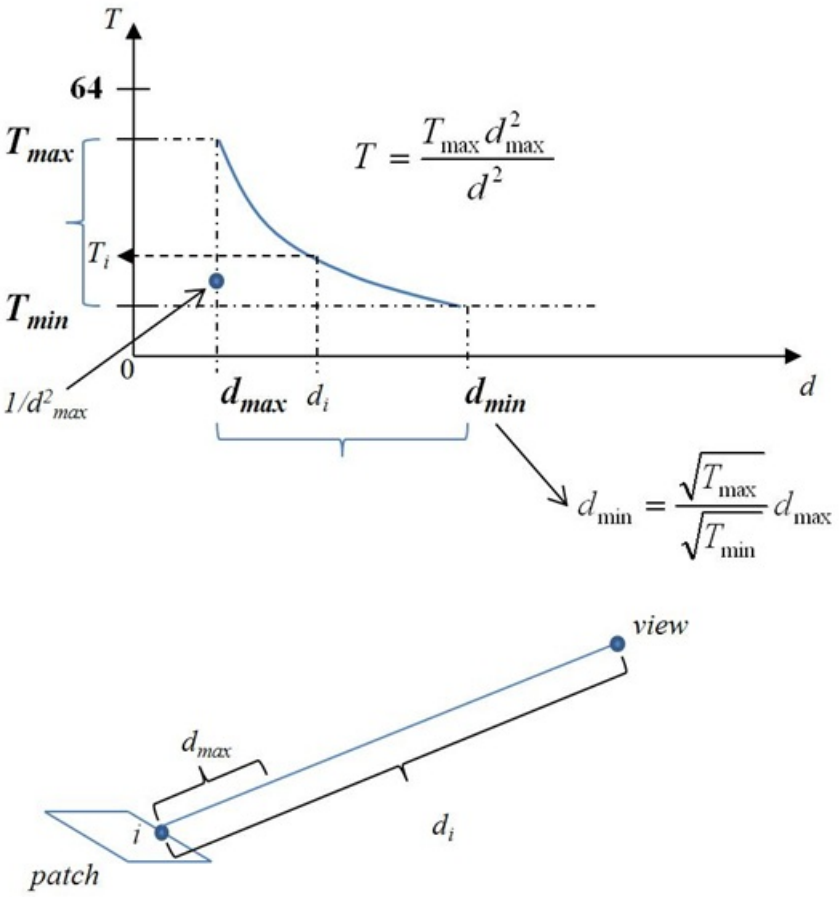

Figura 5.5: TessFactor como uma função de distância da câmera d (equação 5-1)

$$D_x = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * H$$

$$D_y = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * H$$

onde * representa a operação de convolução em duas dimensões. Em cada ponto na imagem, a magnitude do gradiente é obtida através da combinação das duas aproximações:

$$D = \sqrt{{D_x}^2 + {D_y}^2} \tag{5-8}$$

Um novo mapa M representando a magnitude do gradiente é criado. Cada Texel de M possui o valor de D correspondente ao respectivo ponto em H. O mesmo processo é repetido com M para calcular aproximações da segunda

derivada que chamaremos de height acceleration map (HAM). O processo de geração do HAM é similar a outros algoritmos de processamento de imagem e pode ser facilmente paralelizado na GPU usando pixel shaders [Mitchell, 2002].

5.6.2 Transferindo o HAM para o GPU

A nova pipeline baseada no Shader Model 5.0, trabalha com o conceito de patches, que em nosso caso é um tile do terreno. Os fatores de tecelagem devem ser determinados por patch no estágio Hull Shader.

Cada patch exige quatro valores informativos (cada um em um canal de textura): valor de HAM_{max}, máxima altura h, a posição do texel x de h e a posição do texel y de h. Em seguida, uma textura H com quatro canais de dados é criada. O primeiro canal recebe o HAM_{max}. O segundo canal recebe o máximo valor de altura h. Já o terceiro e quarto canais recebem a posição (x,y) do texel correspondente.

5.6.3 Deslocamento de terreno

Após a malha ser tesselada, o próximo estágio de processamento é o Domain Shader, que é executado para cada novo vértice gerado pelo Tessellator. O Domain Shader recebe os quatro vértices do patch inicial v1, v2, v3, v4 e duas coordenadas normalizadas ([0..1]), u e v, que representam os vértices gerados para este patch. Para encontrar a posição universal do novo vértice, nós linearmente interpolamos os vértices do patch, utilizando u e v como fatores de interpolação. O mesmo processo é feito com as coordenadas de textura. Após isso, o movimento vertical do vértice é obtido a partir do mapa de altura utilizando as coordenadas de textura do vértice.

5.7 Resultados

Nossos testes foram executados em um Intel Core i7 920 com uma Nvidia GTX480 e 6GB de RAM. Iniciamos com um mapa de altura de 2048x2048, que é equivalente ao mapa do Rio de Janeiro ($4,4\text{x}10^{10}m^2$) com um detalhe máximo de $100m^2$. Também foi feito um frustum culling em GPU para descartar os patches que não estão no frustum da câmera (basta setar o TessFactor para 0).

As figuras 5.6 e 5.8 mostram os resultados para o caso de 2048x2048 (Figuras 5.7c e 5.7d). Os resultados mostraram que apenas por ser capaz de explorar o Tessellator sem aplicar nenhum algoritmo LOD, em termos de fps

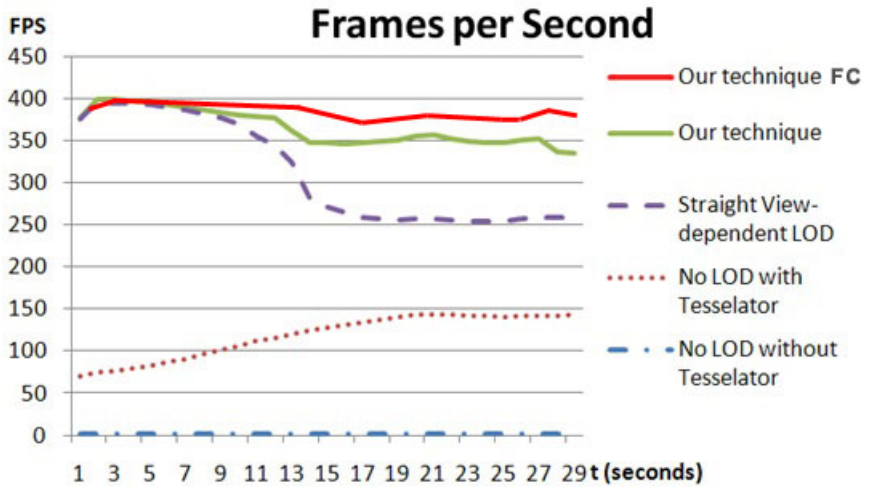

Figura 5.6: Frames por segundo (fps) para o caso das Figuras 5.7c e 5.7d. "Straight View-dependent LOD" não usa T_{max}, "No Lod with Tessellator" utiliza o TessFactor 64 para toda a malha, "No LOD, without Tessellator" é apenas uma referência (o caso onde toda a malha é transferida do CPU para o GPU). "Our technique FC" shows performance with GPU frustum culling turned on.

nossa renderização já é muito superior a uma abordagem simples de CPU para terrenos, como pode ser visto nas curvas mais baixas da Figura 5.6. Nesse caso, ambas as figuras apresentaram o mesmo número de triângulos. Comparando um tradicional algoritmo de LOD view-dependent com nossa técnica, somos capazes de manter valores consistentes de frame independente da posição do visualizador. Além disso, o número de triângulos foi reduzido significantemente com nossa técnica Figura 5.8.

A Figura 5.9 apresenta valores de fps para o caso de um height map 65536x65536, que é equivalente ao mapa do Brasil ($8x10^{12}m^2$ com um detalhe máximo de $40m^2$). O resultado mostra que nosso sistema é capaz de promover um rendering consistente para um caso de terreno massivo. Como os resultados indicam, nossa técnica é aplicável para a maioria dos mapas de altura disponíveis atualmente.

5.8 Discussão

Apresentamos uma nova técnica para renderização de terrenos utilizando tecelagem em hardware, sem estruturas hierárquicas. Nossa técnica é capaz de renderizar grandes mapas de alturas com frame rates interativos, além de ser muito simples de implementar. Apresentamos também um verdadeiro processo local paralelo, no sentido de que os resultados de cada patch de terreno não dependem dos resultados já obtidos em patches adjacentes ou outros patches.

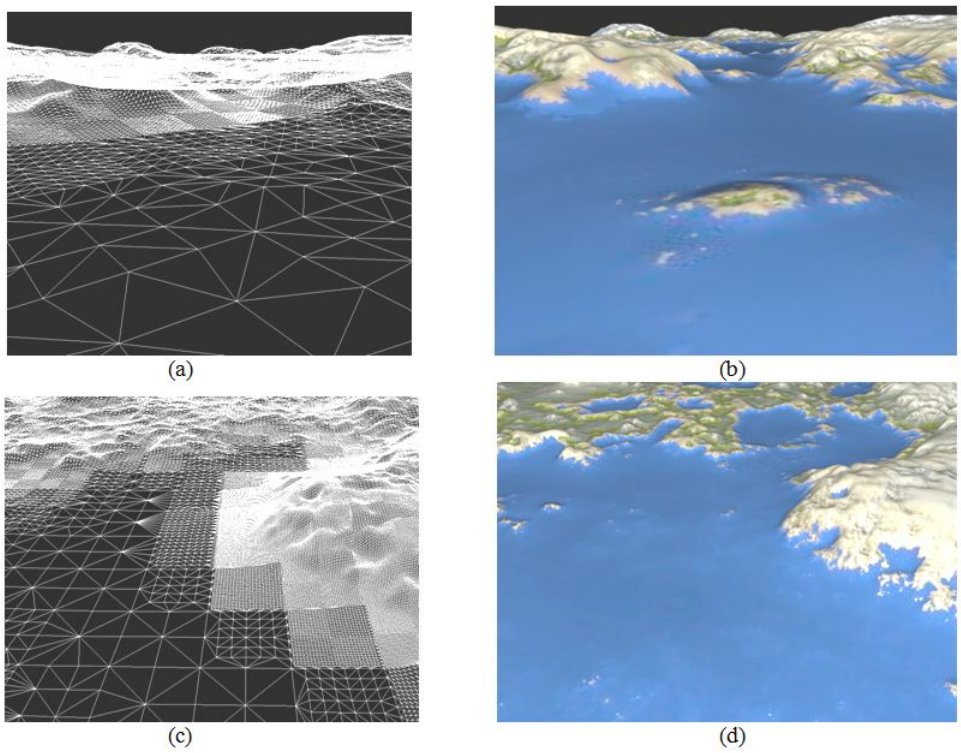

Figura 5.7: Modelos de terreno em wireframe e visualização final correspondente gerados pela técnica proposta rodando em uma Nvidia GTX480 e um Intel core i7. (a) e (b): mapa de altura de 65536x65536, área de $8\text{x}10^{12}m^2$ com precisão de $40m\text{x}40m$, à 52-109 fps. (c) e (d): mapa de altura de 2048x2048, area de $4.4\text{x}10^{10}m^2$ com precisão de $100m\text{x}100m$, à 347-399 fps.

Com esta abordagem somos capazes de determinar, quando necessário, o mínino de tecelagem exigido para um mínimo de erro em espaço projetado. Mantendo o menor refinamento possível, podemos reduzir overheads de subdivisão. Consequentemente, em áreas onde um refinamento profundo é exigido, podemos usar o tessellator em todo seu potencial. Fazendo o melhor uso deste hardware, nós drasticamente minimizamos o overhead do barramento CPU-GPU com apenas um pequeno grupo de primitivas sendo transferido. Nosso algoritmo é linearmente escalável, enquanto alguns outros modelos apresentam algoritmos hierárquicos logarítmicos. Contudo, nossa constante é muito pequena devido ao uso extensivo do tessellator. Em nosso testes, usando o tessellator podemos renderizar modelos em torno de 10^8 triângulos com frame rates interativos e erros menores que 3 pixels. Em qualquer outra abordagem de CPU com LOD este caso corresponderia a um exorbitante número de 5.0 GB de dados sendo transferidos em cada frame entre CPU e GPU. O contínuo fotorrealismo do terreno sem erros é garantido pelo estável geomorphing e edge-splitting através do estágio fixo do tessellator na GPU.

Para futuros trabalhos, planejamos criar uma hierarquia de tiles em terreno que seja capaz de manter a continuidade das divisões. Com tal

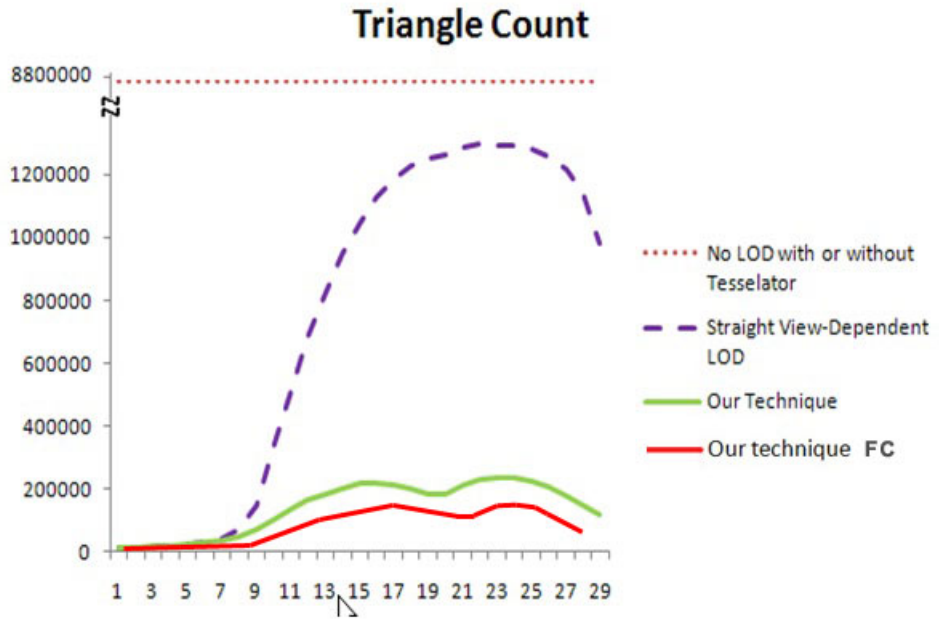

Figura 5.8: Contagem de triângulos para o mesmo caso da Figura 5.6.

estrutura, o sistema estaria apto a selecionar por toda parte do terreno qual nível de hierarquia utilizar. Portanto, o número de primitivas poderia ser reduzido drasticamente para terrenos massivos.

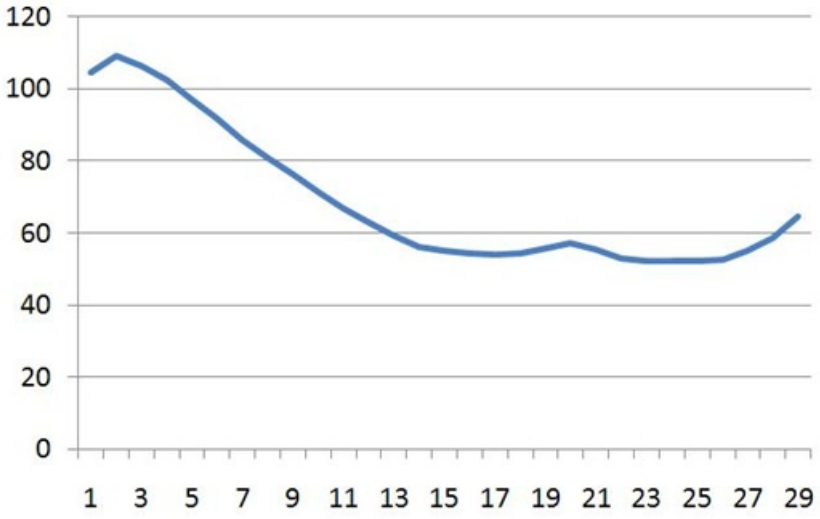

Figura 5.9: Frames por segundo (fps) para o caso 65536x65536 nas Figuras 5.7a e 5.7b.

6
Conclusão

Neste trabalho foram apresentadas avaliações de performance do novo estágio do pipeline gráfico, chamado Tessellator. Também foram implementados algoritmos clássicos neste novo pipeline e foi feita uma comparação qualitativa e quantitativa entre os algoritmos. Além disso, foram propostos dois novos algoritmos que fazem uso do Tessellator, um para renderização de terrenos e outro para renderização de tubos 3D.

No segundo capítulo foi apresentado uma introdução ao novo estágio do pipeline e foram feitos testes analisando a economia na largura de banda ao gerar primitivas na GPU ao invés de gerá-las na CPU. Com o uso do Tessellator fomos capazes de renderizar 163 milhões de triângulos com taxas interativas de FPS.

No terceiro capítulo foi implementado um algoritmo clássico utilizando o Tessellator (PN-Triangles) e outro algoritmo proposto recentemente (Phong Tessellation). Esses dois algoritmos podem ser fortes candidatos a virarem padrão nas aplicações de renderização em tempo real que requerem tecelagem dinâmica. Isto se deve pelo fato de ambos serem puramente locais, ou seja, não usam informação da vizinhança para fazer a subdivisão. Esta propriedade os torna muito propícios para o uso na GPU.

No quarto capítulo foi proposto um novo algoritmo que faz uso do Tessellator para gerar tubos 3D. Fomos capazes de aumentar o número de tubos renderizados em uma cena em mais de uma ordem de grandeza se comparado com a mesma abordagem em CPU. Além disso, este método exercita de maneira simples todos os estágios do novo pipeline gráfico tornando-se um algoritmo bastante didático para passar o conhecimento do novo pipeline gráfico.

Por último, foi proposta uma nova abordagem para renderização de terrenos com o uso do Tessellator. Foi usado um processo local paralelo em GPU para fazer uma tecelagem adaptativa do terreno em tempo real.

6.1 Contribuições

No melhor do nosso conhecimento, não existe na literatura um trabalho sobre o funcionamento do novo pipeline gráfico com a abrangência do que apresentamos aqui. A comparação entre o PN-Triangles e o Phong Tessellation expõe claramente os prós e os contras de cada abordagem, auxiliando na escolha entre os dois algoritmos.

Uma das principais contribuições deste trabalho foi o algoritmo para renderização de Tubos 3D que foi implantado com sucesso em um visualizador de campos de petróleo. Além disso, a abordagem de renderização de terrenos permite renderizar com taxas de FPS interativas terrenos de tamanhos grandes com precisão satisfatória.

Por último, foi demonstrado com clareza qual é o real ganho de performance do Tessellator se comparado a uma abordagem em CPU.

Como resultado deste trabalho, publicamos um artigo no SBGames 2010 [Valdetaro et al., 2010] e outros estão em processo de avaliação.

6.2 Trabalhos Futuros

A chegada do Tessellator abre um novo paradigma na renderização em tempo real. Vários algoritmos novos irão surgir nos próximos anos. No melhor do nosso conhecimento, não existe um trabalho que compare o desempenho de esquemas de subdivisão que não são simplesmente locais. Estes esquemas (e.g Superfícies de Catmull-Clark [Catmull and Clark, 1978]) produzem malhas visualmente melhores e um comparativo de performance entre esses esquemas e esquemas locais seria válido.

Na renderização de terreno planejamos criar um esquema que possibilite fazer uma hierarquia de tiles, sendo capaz de manter a malha contínua e sem quebras.

Geração de malhas em GPU e análise de tecelagens adaptativas são outros dois temas interessantes para o uso do Tessellator.

Referências Bibliográficas

[2KGames, 2010] 2KGames (2010). "http://www.mafia2game.com/". 3.1

[Asirvatham and Hoppe, 2005] Asirvatham, A. and Hoppe, H. (2005). "Chapter 2 Terrain Rendering Using GPU-Based Geometry Clipmaps". 5.1, 1

[Bhatt, 2004] Bhatt, A. V. (2004). "PCI-Express Specification". In *Intel Whitepaper*. 1

[Bloomenthal, 1990] Bloomenthal, J. (1990). "Graphics gems". chapter Calculation of reference frames along a space curve, pages 567–571. Academic Press Professional, Inc., San Diego, CA, USA. 4.3.2

[Boubekeur and Alexa, 2008] Boubekeur, T. and Alexa, M. (2008). "Phong Tessellation". *ACM Trans. Graphics (Proc. SIGGRAPH Asia)*, 27, pp. 139–143. 3.1, 3.15, 3.7.1

[Boubekeur et al., 2005] Boubekeur, T.; Reuter, P.; and Schlick, C. (2005). "Scalar Tagged PN Triangles". In *EUROGRAPHICS 2005 (Short Papers)*. Eurographics. 3.2, 3.7.1

[Boubekeur and Schlick, 2007] Boubekeur, T. and Schlick, C. (2007). "QAS: Real-Time Quadratic Approximation of Subdivision Surfaces". *Computer Graphics and Applications, Pacific Conference on*, 1, pp. 453–456. 3.2

[Castano, 2008] Castano, I. (2008). "Tessellation of Displaced Subdivision Surfaces in DX11". In *GameFest 2008*. (document), 2.1

[Catmull and Clark, 1978] Catmull, E. and Clark, J. (1978). "Recursively generated B-spline surfaces on arbitrary topological meshes". *Computer Aided Design*, 10, pp. 350–355. 3.1, 3.2, 3.4, 6.2

[Catmull and Rom, 1974] Catmull, E. and Rom, R. (1974). "A class of local interpolating splines". In Barnhill, R. and Riesenfeld, R., editors, *Computer Aided Geometric Design*, pages 317–326. Academic Press. 4.3.1

[de Toledo and Levy, 2004] de Toledo, R. and Levy, B. (2004). "Extending the graphic pipeline with new GPU-accelerated primitives". In *International gOcad Meeting, Nancy, France*. 2, 2.3.1, 4.2

[de Toledo and Lévy, 2008a] de Toledo, R. and Lévy, B. (2008a). "Visualization of Industrial Structures with Implicit GPU Primitives". In *ISVC (1)*, pages 139–150. 2, 2.3.1

[de Toledo and Lévy, 2008b] de Toledo, R. and Lévy, B. (2008b). "Visualization of Industrial Structures with Implicit GPU Primitives". In *ISVC (1)*, pages 139–150. 4.3.3

[Decoro and Tatarchuk, 2007] Decoro, C. and Tatarchuk, N. (2007). "Abstract Real-time Mesh Simplification Using the GPU". In *Proceedings of the 2007 Symposium on Interactive 3D Graphics ans Games*. 5.2

[DeCoro and Tatarchuk, 2007] DeCoro, C. and Tatarchuk, N. (2007). "Real-time Mesh Simplification Using the GPU". In *Symposium on Interactive 3D Graphics (I3D)*, volume 2007, page 6. 2.2.6

[Doo, 1978] Doo, D. (1978). "A subdivision algorithm for smoothing down irregularly shaped polyhedrons". In *Int'l Conf. Ineractive Techniques in Computer Aided Design*, pages 157–165, Bologna, Italy. IEEE Computer Soc. 3.2

[Doo and Sabin, 1978] Doo, D. and Sabin, M. (1978). "Behaviour of recursive division surfaces near extraordinary points". *Computer Aided Design*, 10(6), pp. 356–360. 3.2

[Doss, 2008] Doss, J. (2008). "Inking the Cube: Edge Detection with Direct3D10". In *Gamasutra Article*. 2.1

[Drone, 2007] Drone, S. (2007). "Real-time particle systems on the GPU in dynamic environments". In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 80–96, New York, NY, USA. ACM. 2.2.6

[Drone et al., 2010] Drone, S.; Lee, M.; and Oneppo, M. (2010). "Direct3D 11 Tessellation". 1

[Duchaineau et al., 1997] Duchaineau, M.; Wolinsky, M.; Sigeti, D. E.; Miller, M. C.; Aldrich, C.; and Mineev-Weinstein, M. B. (1997). "ROAMing terrain: real-time optimally adapting meshes". In *Proceedings of the 8th conference on Visualization '97*, VIS '97, pages 81–88, Los Alamitos, CA, USA. IEEE Computer Society Press. 2, 5.5.2

[Falby et al., 1993] Falby, J. S.; Zyda, M. J.; Pratt, D. R.; and Mackey, Y. L. (1993). "NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulation". *Computers and Graphics*, 17, pp. 65–69. 1

[Fernando, 2003] Fernando, R. (2003). "The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics". 2

[Floriani et al., 1996] Floriani, L. D.; Marzano, P.; and Puppo, E. (1996). "Multi-resolution Models for Topographic Surface Description". In *Proceedings of The Visualization'95*. 5.1, 1

[Hoppe, 1996] Hoppe, H. (1996). "Progressive Meshes". In Rushmeier, H., editor, *SIGGRAPH96*, pages 99–108, New York. ACM Press/ACM SIGGRAPH. 2, 2.2.4

[Hoppe, 1997] Hoppe, H. (1997). "View-dependent refinement of progressive meshes". In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 189–198, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co. 2

[Hoppe, 1998] Hoppe, H. (1998). "Efficient implementation of progressive meshes". *Computers and Graphics*, 22(1), pp. 27–36. 2

[Hu et al., 2010a] Hu, L.; Sander, P. V.; and Hoppe, H. (2010a). "Parallel View-Dependent Level-of-Detail Control". *IEEE Transactions on Visualization and Computer Graphics*, 16, pp. 718–728. 2

[Hu et al., 2010b] Hu, L.; Sander, V.; and Hoppe, H. (2010b). "Parallel View-Dependent Level-of-Detail Control.". *IEEE Transactions on Visualization and Computer Graphics*, 16, pp. 718–728. 5.1, 5.2

[Ji et al., 2006] Ji, J.; Wu, E.; Li, S.; and Liu, X. (2006). "View-dependent refinement of multiresolution meshes using programmable graphics hardware". *Vis. Comput.*, 22, pp. 424–433. 5.2

[Jim Brewer, 2004] Jim Brewer, J. S. (2004). "PCI-Express Technology". In *Intel Whitepaper*. 1

[Kobbelt, 2000] Kobbelt, L. (2000). "sqrt(3)-subdivision". In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 103–112, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co. 3.2, 3.4

[Kondratieva et al., 2005] Kondratieva, P.; Krüger, J.; and Westermann, R. (2005). "The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization". In *Proceedings IEEE Visualization 2005*. 4.1

[Levenberg, 2002] Levenberg, J. (2002). "Fast view-dependent level-of-detail rendering using cached geometry". In *Proceedings of the conference on Visualization '02*, VIS '02, pages 259–266, Washington, DC, USA. IEEE Computer Society. 3

[Limaye, 2009] Limaye, J. (2009). "Evolution of DirectX". 1

[Lindstrom et al., 1996] Lindstrom, P.; Koller, D.; Ribarsky, W.; Hodges, L. F.; Faust, N.; and Turner, G. A. (1996). "Real-ti continuous level of detail rendering of height fields". In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 109–118, New York, NY, USA. ACM. 2

[Lindstrom and Pascucci, 2002] Lindstrom, P. and Pascucci, V. (2002). "Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization". *IEEE Transactions on Visualization and Computer Graphics*, 8, pp. 239–254. 5.5.2

[Loop, 1987] Loop, C. (1987). "Smooth Subdivision Surfaces Based on Triangles". In *PhD Thesis - University of Utah*. 3.2, 3.4

[Loop and Schaefer, 2008] Loop, C. and Schaefer, S. (2008). "Approximating Catmull-Clark subdivision surfaces with bicubic patches". *ACM Trans. Graph.*, 27, pp. 8:1–8:11. 3.2

[Loop et al., 2009] Loop, C.; Schaefer, S.; Ni, T.; and Castaño, I. (2009). "Approximating subdivision surfaces with Gregory patches for hardware tessellation". *ACM Trans. Graph.*, 28, pp. 151:1–151:9. 3.2

[Lorenz and Döllner, 2008] Lorenz, H. and Döllner, J. (2008). "Dynamic Mesh Refinement on GPU using Geometry Shaders". In *16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG) - Full Papers*, pages 97–104. 2, 2.1

[Losasso, 2004] Losasso, F. (2004). "Geometry clipmaps: terrain rendering using nested regular grids". *ACM Transactions on Graphics*, 23, pp. 769–776. 5.1, 1

[Merhof et al., 2006] Merhof, D.; Sonntag, M.; Enders, F.; Nimsky, C.; Hastreiter, P.; and Greiner, G. (2006). "Hybrid Visualization for White Matter Tracts using Triangle Strips and Point Sprites". *IEEE Transactions on Visualization and Computer Graphics*, 12, pp. 1181–1188. 4.1, 4.2

[Mitchell, 2002] Mitchell, J. L. (2002). "Image Processing with 1.4 Pixel Shaders in Direct3D". In Engel, W., editor, *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware, Plano, Texas. 5.6.1

[Ni et al., 2009] Ni, T.; Castaño, I.; Peters, J.; Mitchell, J.; Schneider, P.; and Verma, V. (2009). "Efficient substitutes for subdivision surfaces". In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, pages 13:1–13:107, New York, NY, USA. ACM. 2.6, 2.7

[NVIDIA, 2010] NVIDIA (2010). "Geforce GTX 580 specification". In *NVIDIA whitepaper*. 1

[Owens et al., 2008] Owens, J.; Houston, M.; Luebke, D.; Green, S.; Stone, J.; and Phillips, J. (May, 2008). "GPU Computing". In *Proceedings of the IEEE*, page 96. 1

[Pajarola and Gobbetti, 2007] Pajarola, R. and Gobbetti, E. (2007). "Survey of semi-regular multiresolution models for interactive terrain rendering". *Vis. Comput.*, 23, pp. 583–605. 5.1, 2, 5.2

[Rocco, 2010] Rocco, D. (2010). "GPU Tessellation". In *University of Pennsilvannia Lecture Slides*. (document), 2.3

[Schneider and Westermann, 2006] Schneider, J. and Westermann, R. (2006). "GPU-friendly High-quality Terrain Rendering". In *WSCG*. 3

[SDK, 2007] SDK, D. (2007). "CubeMapGS Sample". In *Microsoft DirectX SDK*. 2.1

[Sharp, 2000] Sharp, B. (2000). "Implementing Subdivision Surfaces". In *Gamasutra Article*. 3.8, 3.9

[Sobel and Feldman, 1968] Sobel, I. and Feldman, G. (1968). "A 3x3 isotropic gradient operator for image processing". In *Presentation for Stanford Artificial Project*. 5.6.1

[Stoll et al., 2005] Stoll, C.; Gumhold, S.; and Seidel, H.-P. (2005). "Visualization with stylized line primitives". *Visualization Conference, IEEE*, 0, pp. 88. 4.1, 4.2

[Tariq, 2009] Tariq, S. (2009). "D3D11 Tessellation". In *http://developer.amd.com/*. 2.9

[Tatarchuk et al., 2010] Tatarchuk, N.; Barczak, J.; and Bilodeau, B. (2010). "Programming for real-time tessellation GPU". In *AMD whitepaper*. 5.2

[Tatarchuk et al., 2007] Tatarchuk, N.; Shopf, J.; and DeCoro, C. (2007). "Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline".

In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 122–137, New York, NY, USA. ACM. 2.2.6

[Tatarinov, 2008] Tatarinov, A. (2008). "Instanced Tessllation". In *NVIDIA whitepaper*. 2.5

[Toledo et al., 2001] Toledo, R.; Gattass, M.; and Velho, L. (2001). "QLOD: A Data Structure for Interative Terrain Visualization". VISGRAF Laboratory TR-2001-13, IMPA. 3

[Valdetaro et al., 2010] Valdetaro, A.; Nunes, G.; Raposo, A.; Feijo, B.; and de Toledo, R. (2010). "LOD terrain rendering by local parallel processing on GPU". *IX Brazilian symposium on computer games and digitak entertainment*. 6.1

[Vlachos et al., 2001] Vlachos, A.; Peters, J.; Boyd, C.; and Mitchell, J. L. (2001). "Curved PN triangles". In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 159–166, New York, NY, USA. ACM Press. 3.1, 3.10, 3.11, 3.12, 3.13, 3.14

[Wikipedia, 2010] Wikipedia (2010). "Wikipedia - DirectX - http://en.wikipedia.org/wiki/DirectX". 1

[Yang et al., 2010] Yang, J.; Hensley, J.; Grun, H.; and Thibieroz, N. (2010). "Real-Time Concurrent Linked List Construction on the GPU". In *In Rendering Techniques 2010: Eurographics Symposium on Rendering*, Eurographics '10. 2.2.7

[Yeo et al., 2009] Yeo, Y. I.; Ni, T.; Myles, A.; Goel, V.; ; and Peters, J. (2009). "Parallel Smoothing of Quad Meshes". *The Visual Computer*, 25(8), pp. 757–769. 3.2

[Zorin et al., 1996] Zorin, D.; Schroder, P.; and Sweldens, W. (1996). "Interpolating Subdivision for Meshes with Arbitrary Topology". In *SIGGRAPH96*, pages 189–192. 3.2, 3.4