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.

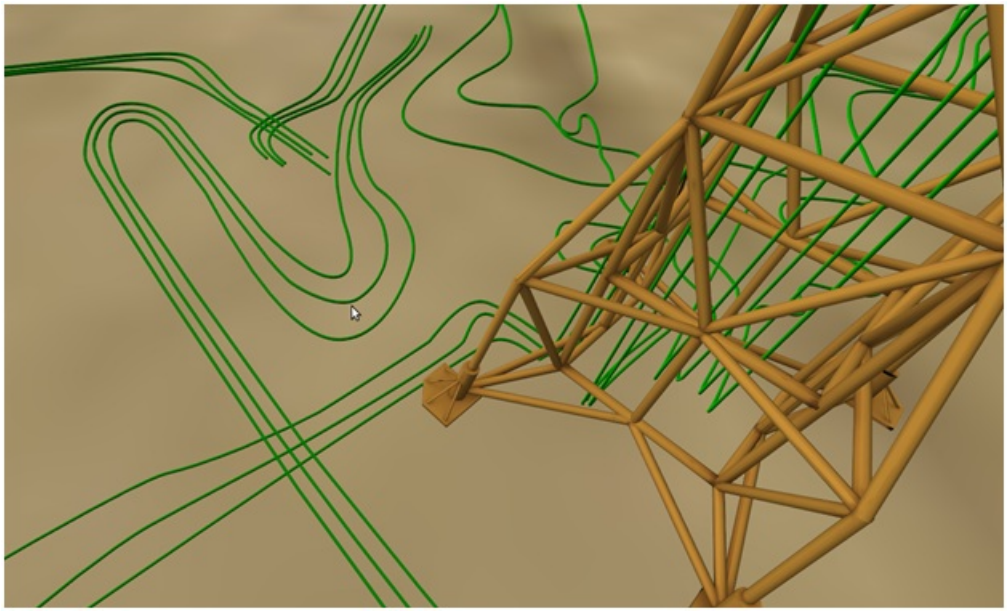

Figura 4.1: Tubos 3D renderizados com a técnica proposta em um visualizador de campos de petróleo.

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.

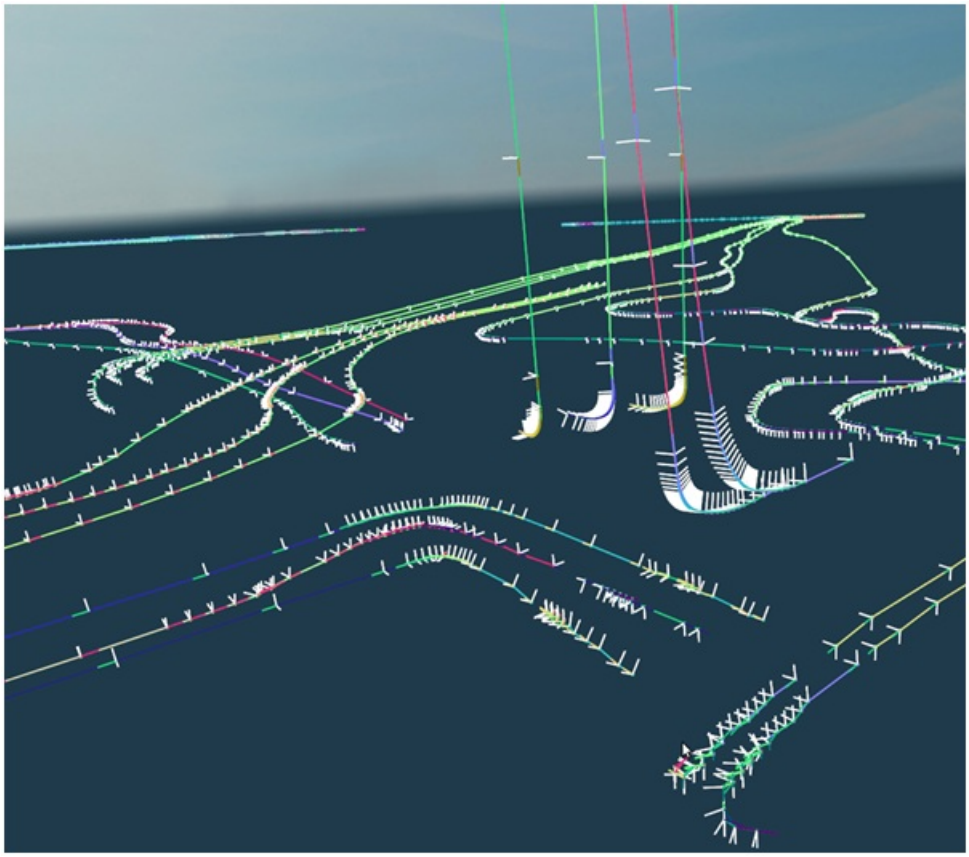

Figura 4.3: Seleção de pontos com as tangentes, normias e bi-normais associadas ao longo da curva. Áreas com derivada numérica alta requerem mais pontos para a reconstrução precisa

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.

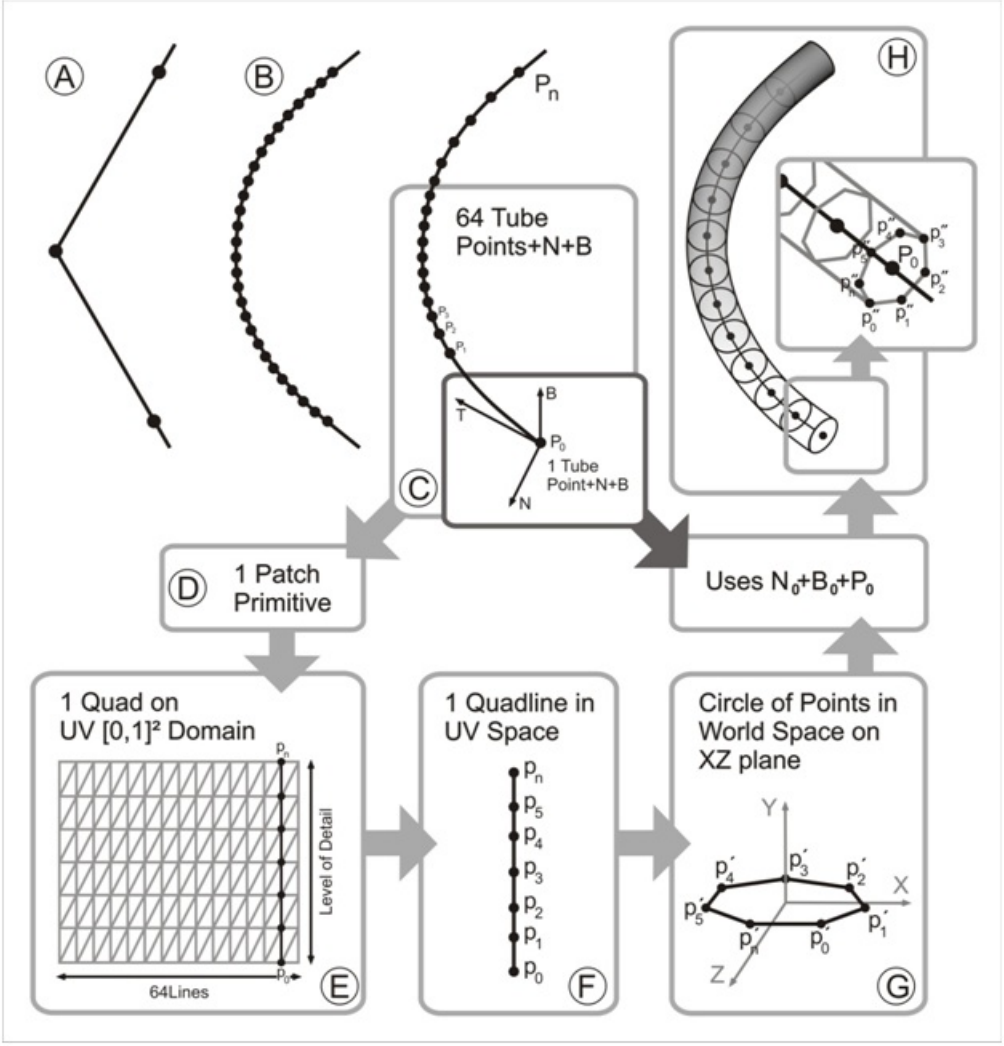

Figura 4.4: A) Um caso de sequência de pontos ruim. B) A sequência ruim após a aplicação da interpolação de Catmull-Rom. C) A seleção de pontos da sequência e cálculo das tangentes, normais e bi-normais. D) Cada 64 pontos implica em uma primitiva(patch). E) O patch é uma primitiva do tipo quad com 64xLOD 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.

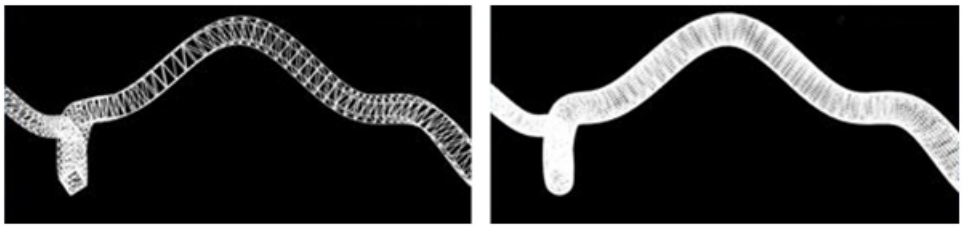

Figura 4.5: Esquerda - Tubo com LOD baixo. Direita - Tubo com LOD alto.

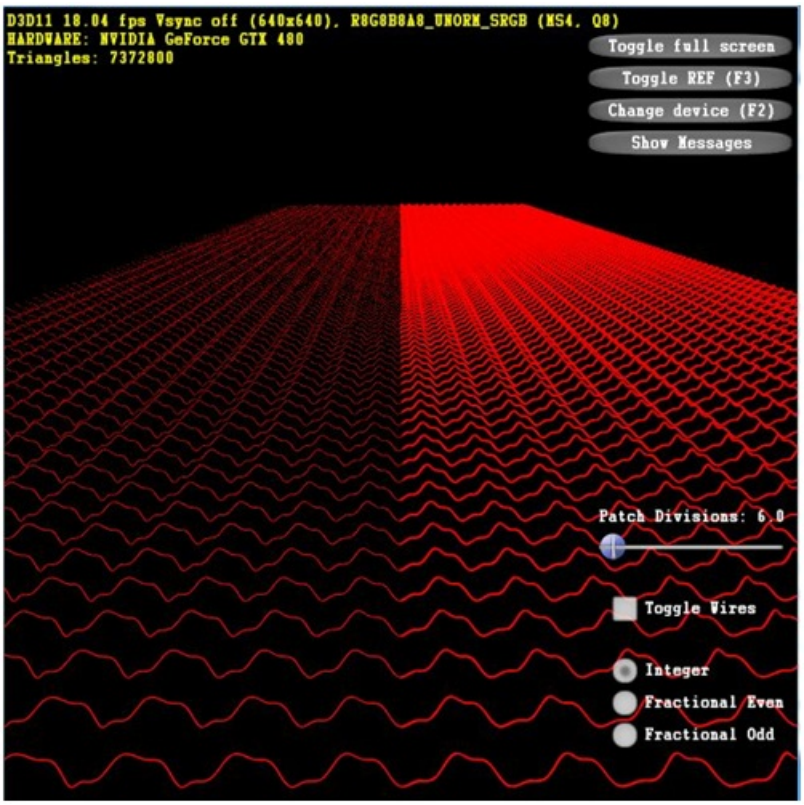

Figura 4.6: Mesma cena: na esquerda com 16x GPU anti-aliasing. Na direita com a correção de aliasing proposta.

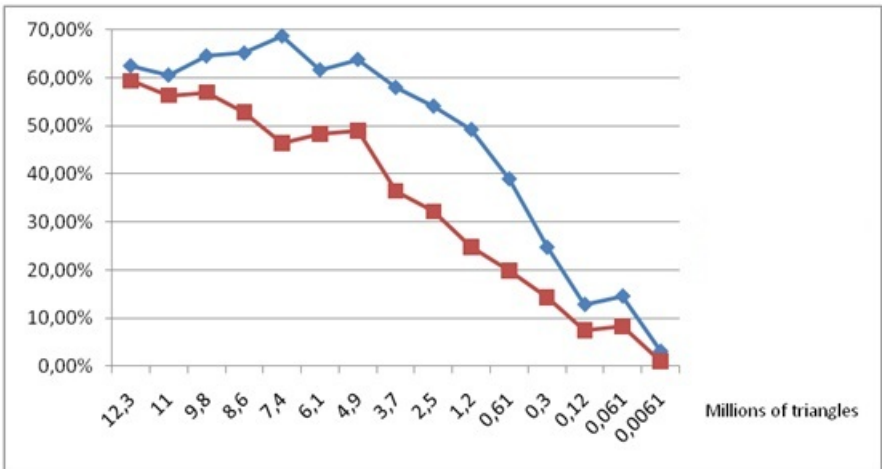

Figura 4.7: Gráfico mostrando a porcentagem de ganho em FPS do nosso algoritmo com (em vermelho) e sem (em azul) a correção de aliasing proposta comparado com a abordagem em CPU sem anti-aliasing.