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.

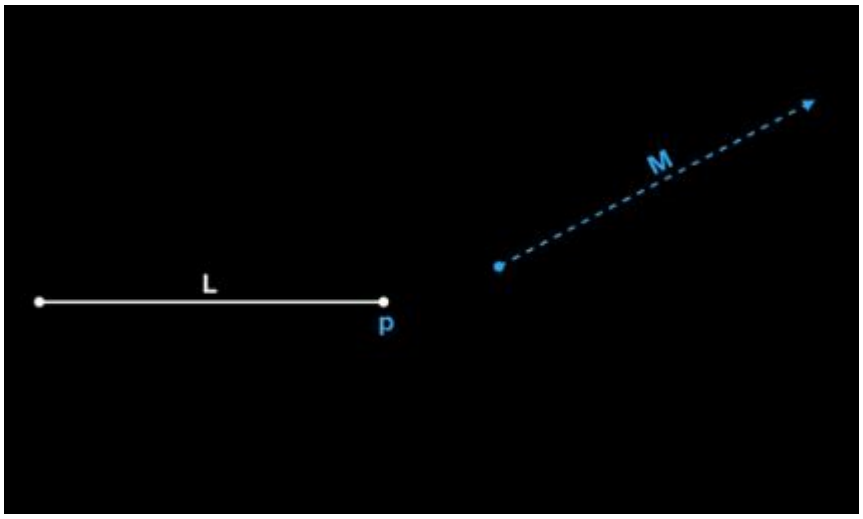

Figura 3.2: Duas curvas que não são contínuas

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

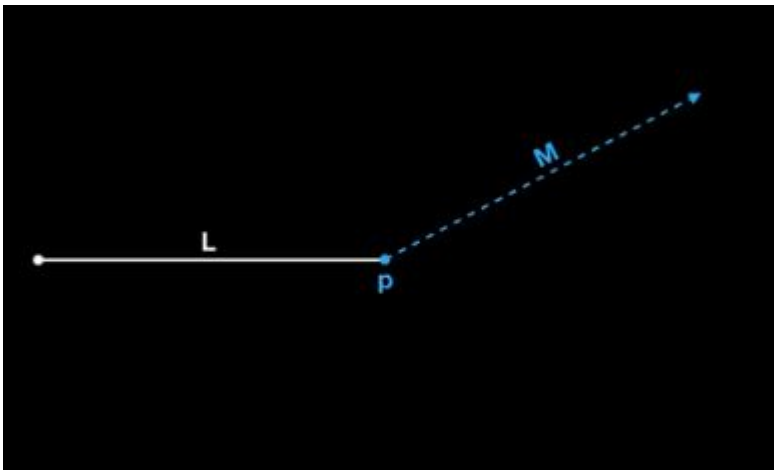

Figura 3.3: Continuidade C^0

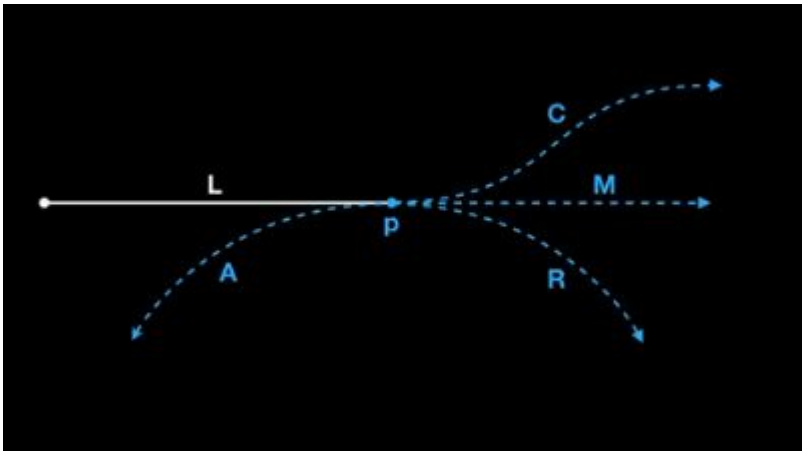

Figura 3.4: Continuidade C^1

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.

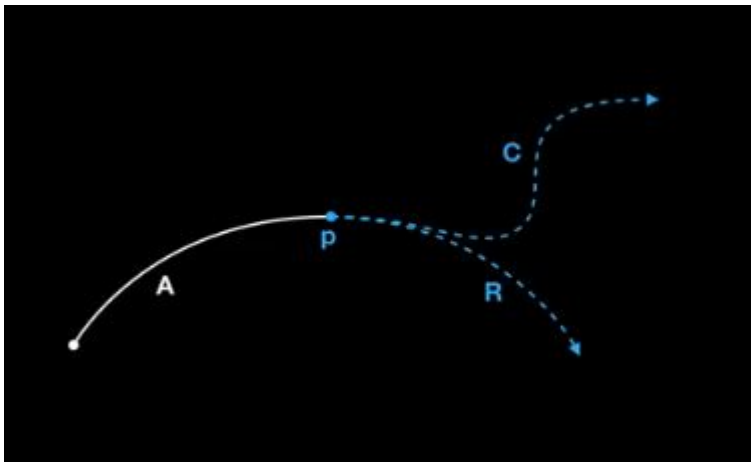

Figura 3.5: Continuidade C^2

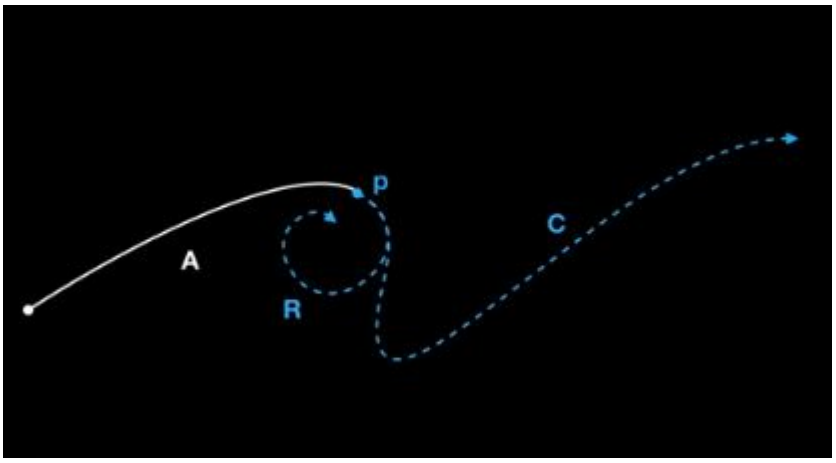

Figura 3.6: Continuidade C^3

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-

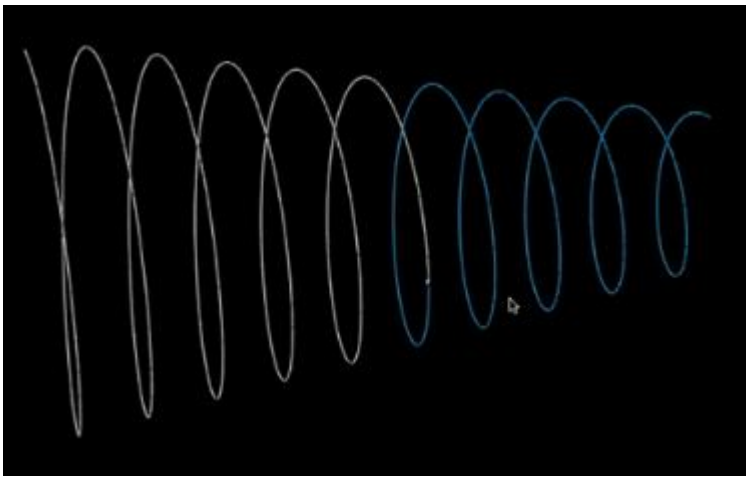

Figura 3.7: Continuidade C^4

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]).

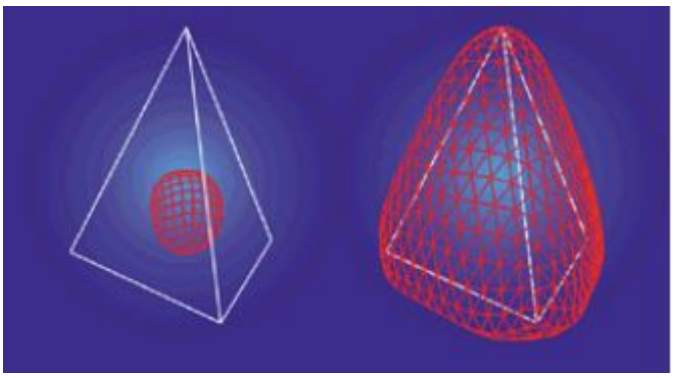

Figura 3.8: Interpolativo vs aproximativo[Sharp, 2000]

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.

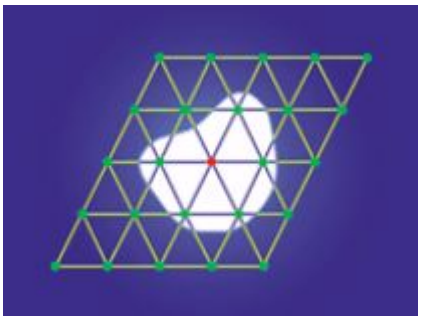

Figura 3.9: Máscara para um algoritmo hipotético de subdivisão [Sharp, 2000]

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.

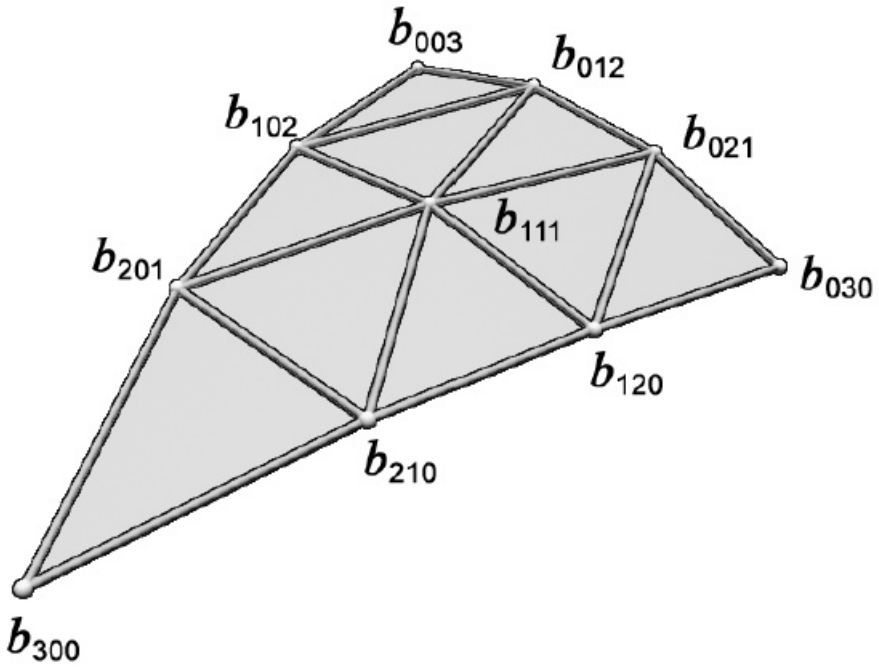

Figura 3.10: Pontos de controle do patch de geometria [Vlachos et al., 2001]

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:

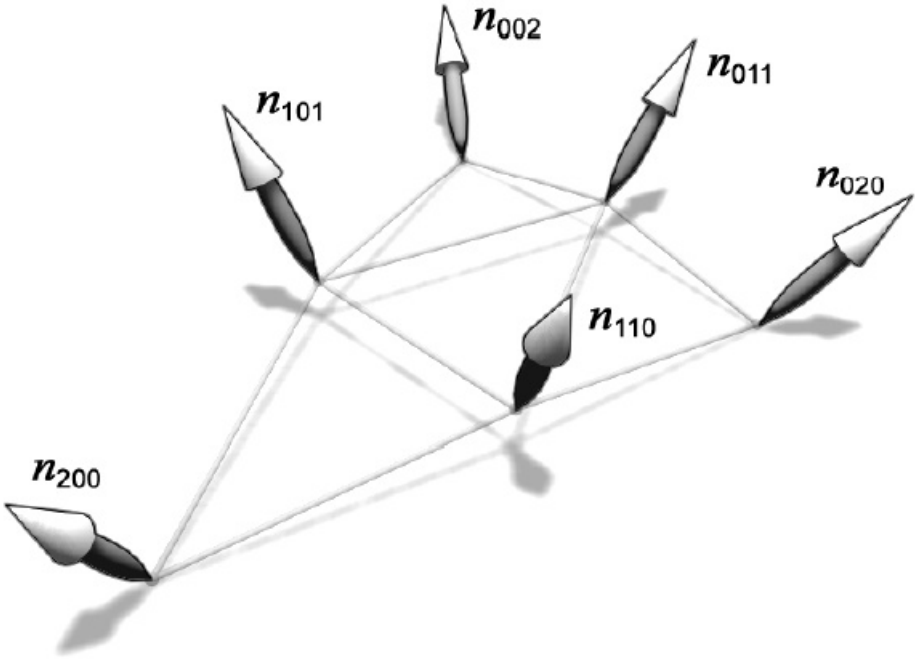

Figura 3.11: Pontos de controle do patch de normal [Vlachos et al., 2001]

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

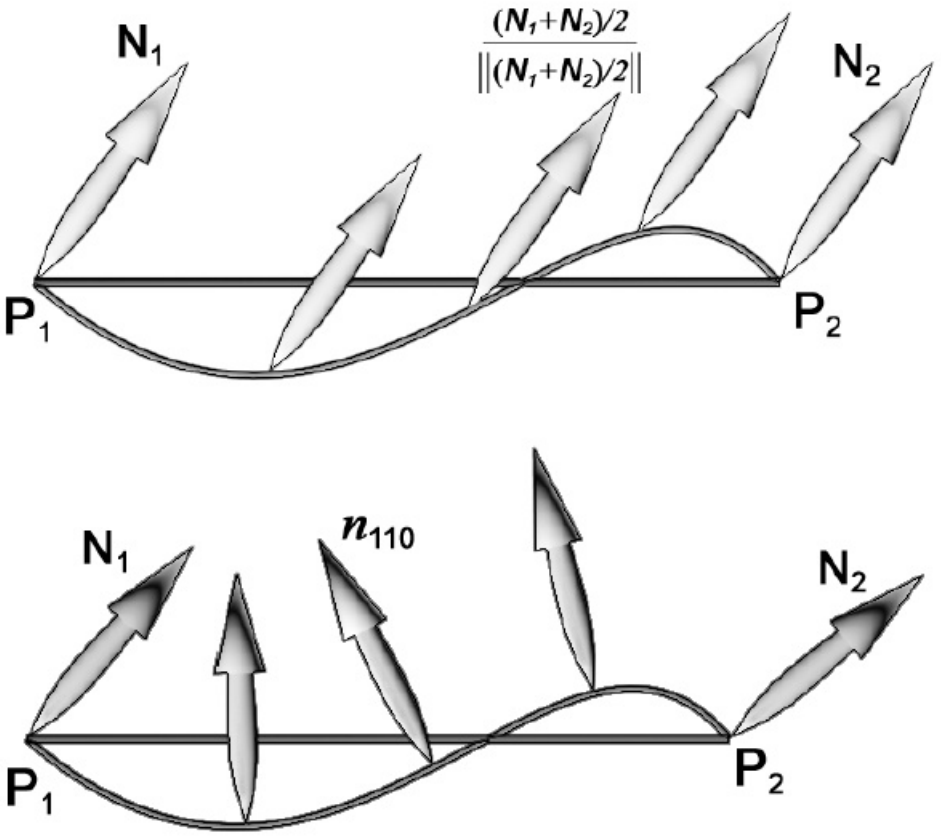

Figura 3.12: Interpolação linear das normais(acima) e variação quadrática(em baixo) [Vlachos et al., 2001]

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:

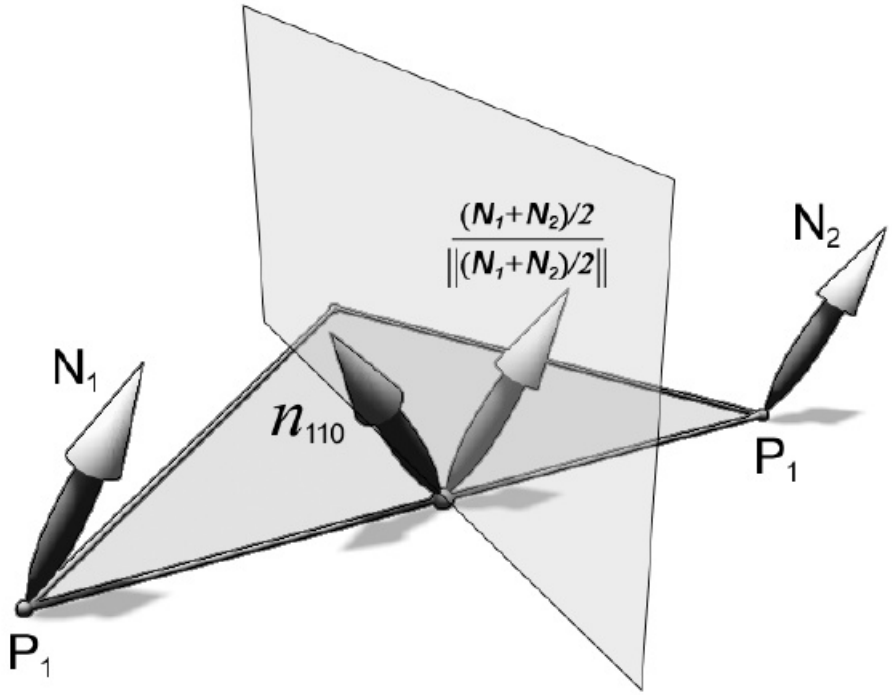

Figura 3.13: Reflexão da normal no meio da aresta pelo plano perpendicular a ela [Vlachos et al., 2001]

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

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

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

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

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

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

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

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.

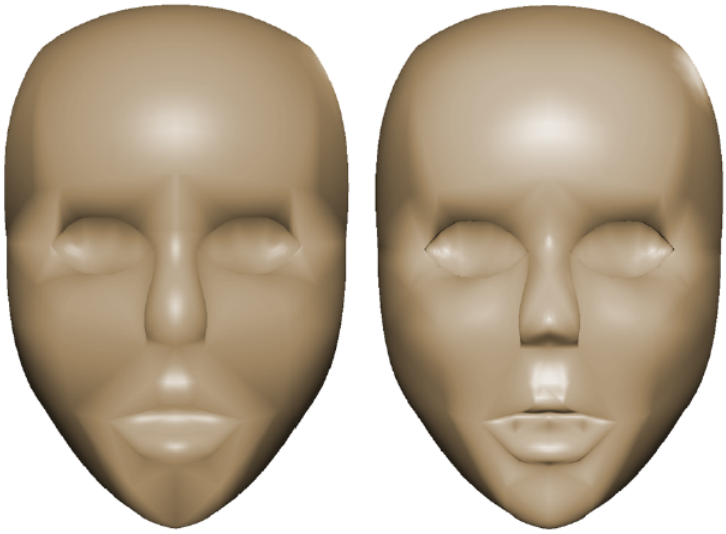

Figura 3.14: Normais variando linearmente (esquerda) e quadraticamente (direita) [Vlachos et al., 2001]

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:

$p : \Re^2 \rightarrow \Re^3$, 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:

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

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

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.

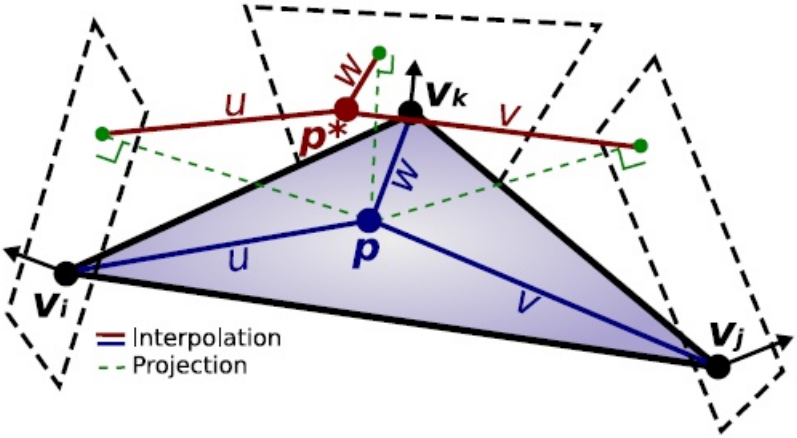

Figura 3.15: Projeções e interpolações do Phong Tessellation [Boubekeur and Alexa, 2008]

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

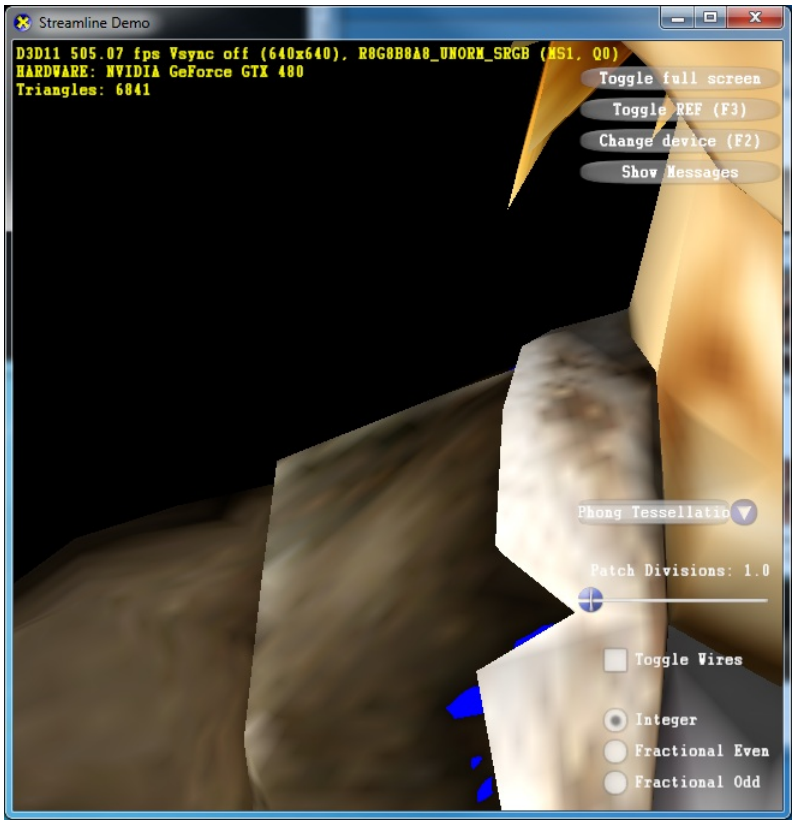

Figura 3.16: Modelo original sem tecelagem

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.

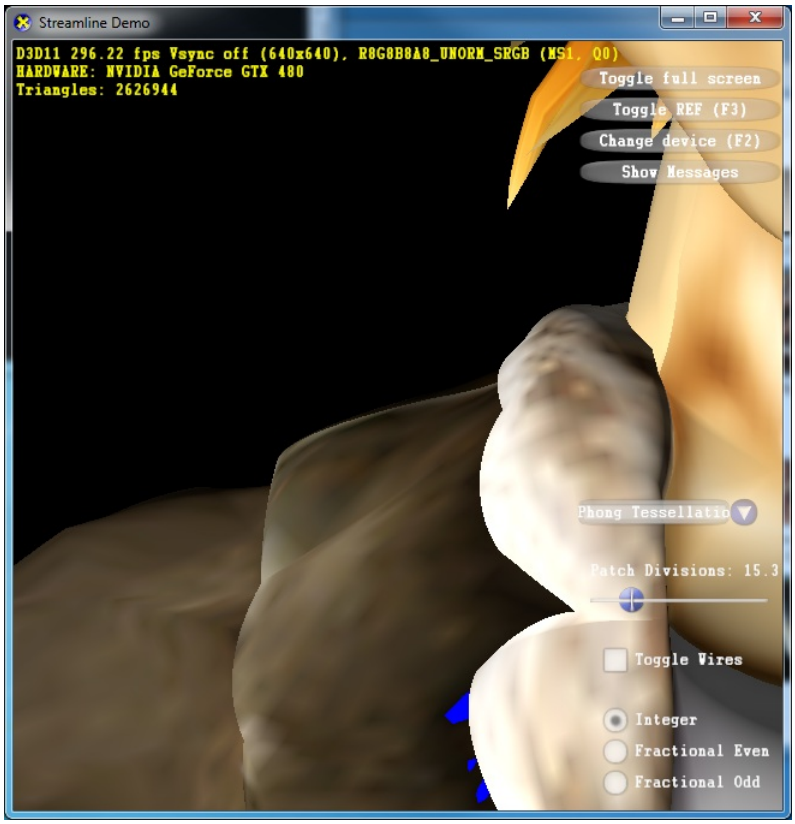

Figura 3.17: Modelo usando o algoritmo Phong Tessellation

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

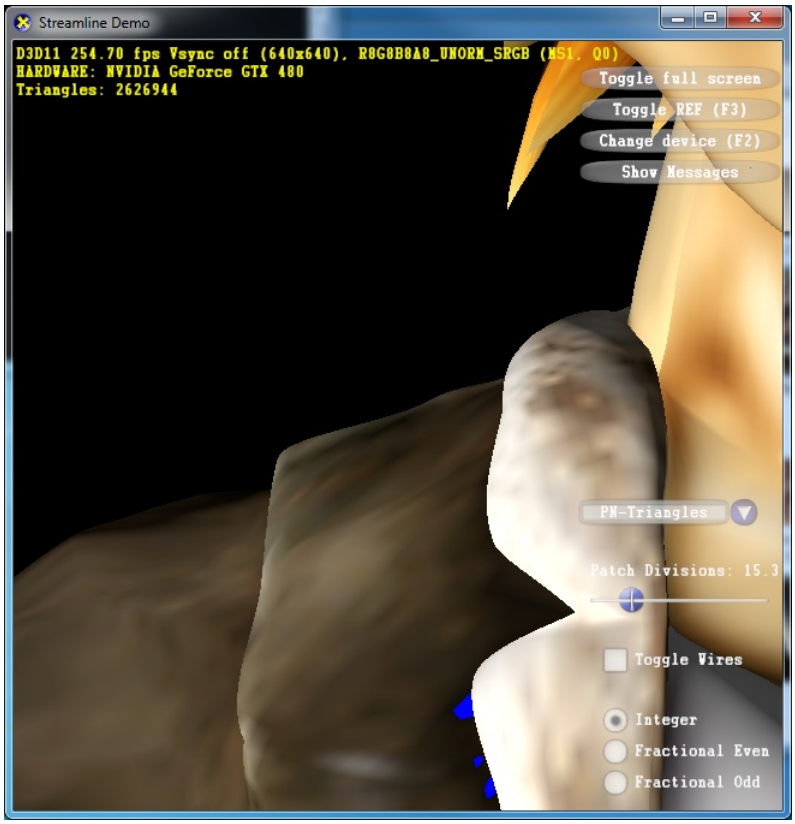

Figura 3.18: Modelo usando o algoritmo PN-Triangles e normais quadráticas

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

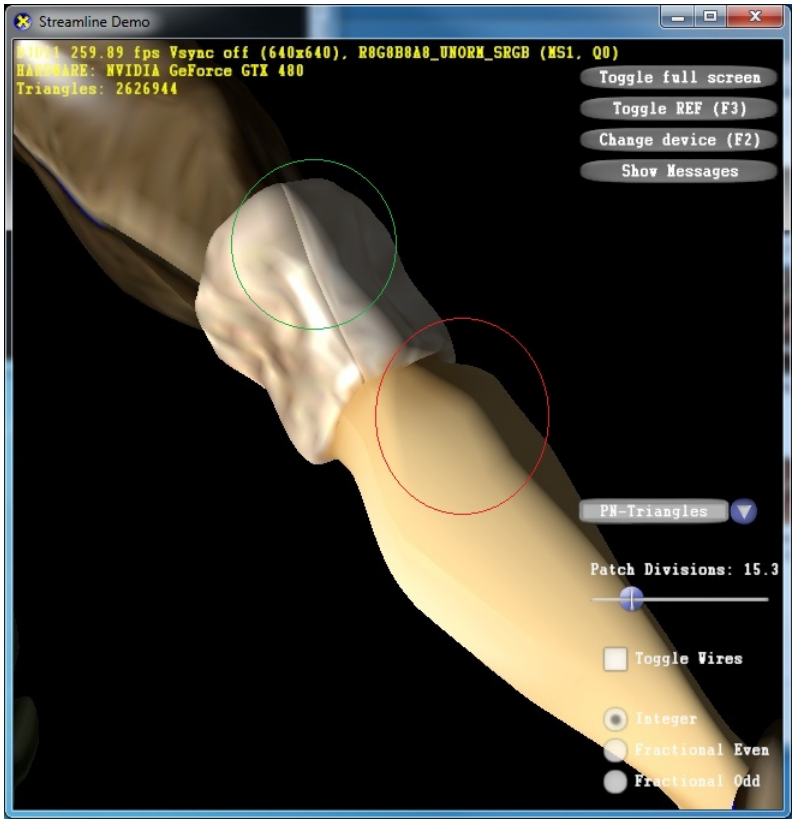

Figura 3.19: PN-Triangles com interpolação linear das normais

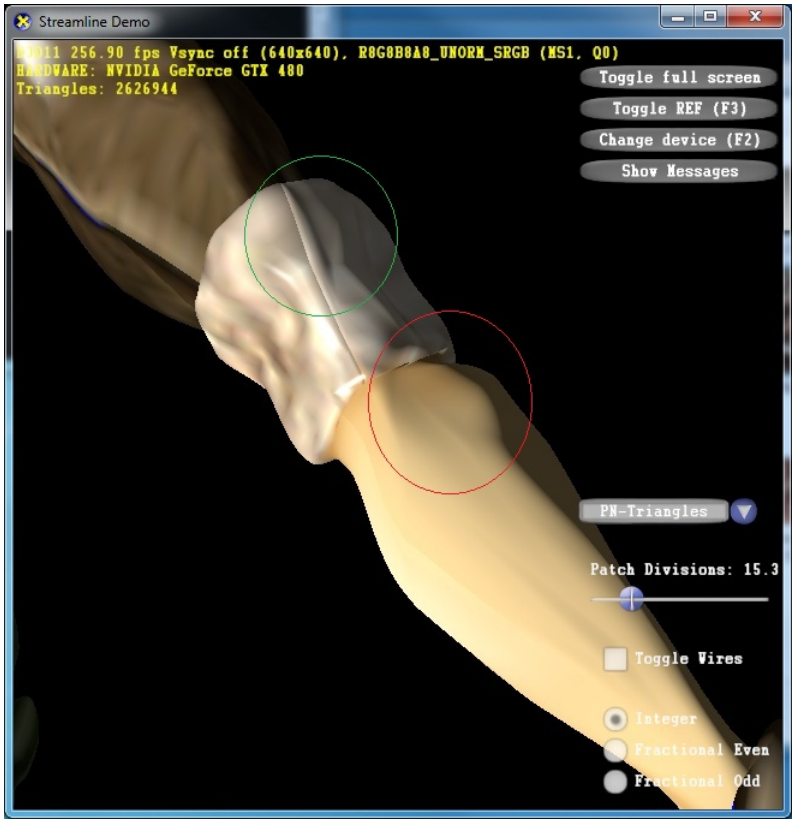

Figura 3.20: PN-Triangles com interpolação quadrática das normais

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

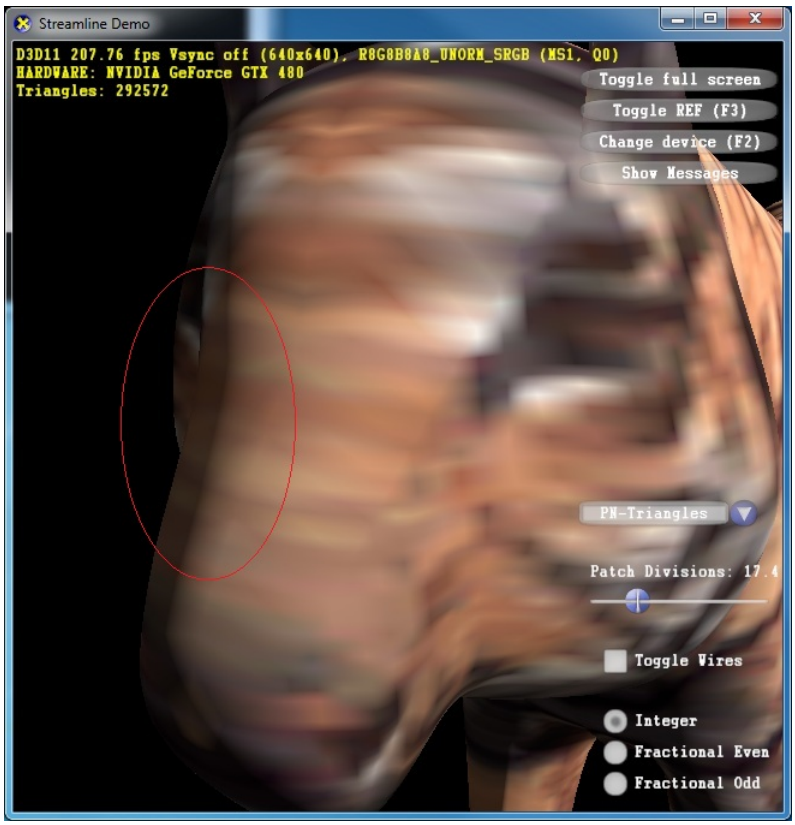

Figura 3.24: PN-Triangles com interpolação linear das normais

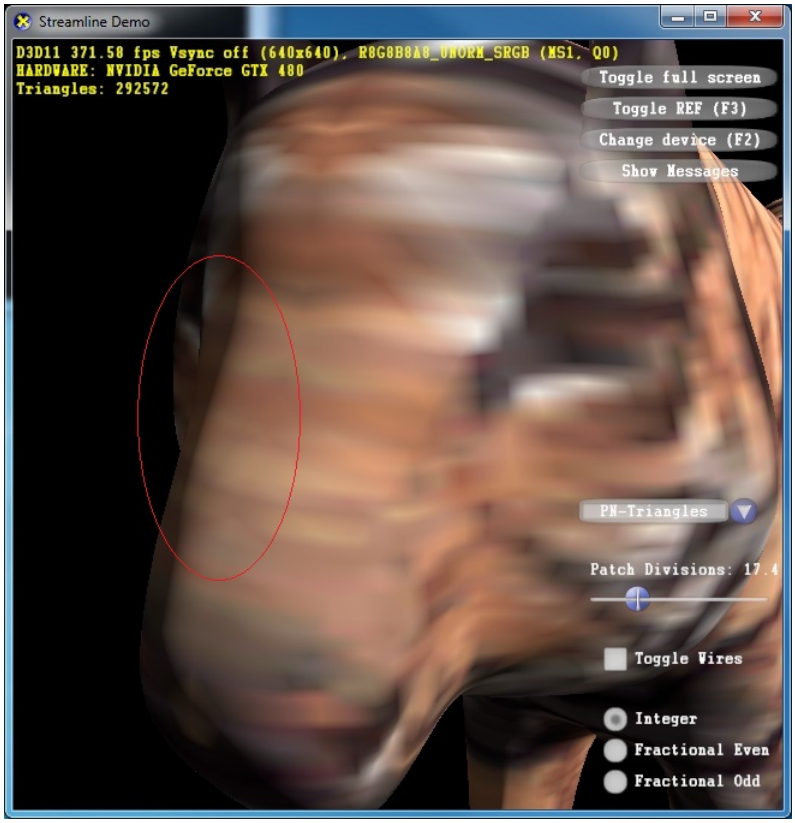

Figura 3.25: PN-Triangles com interpolação quadrática das normais

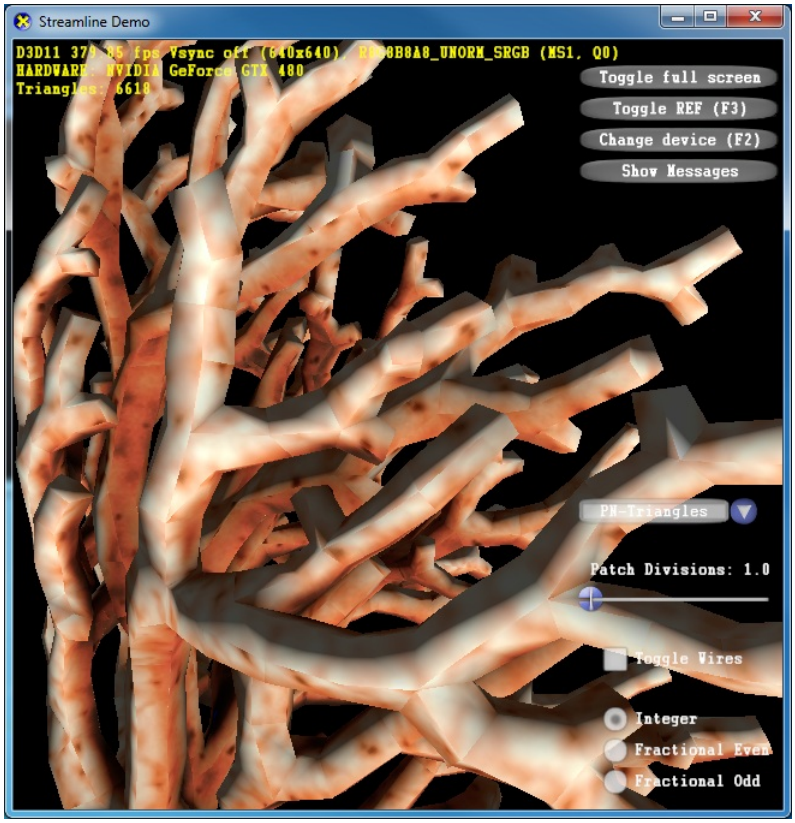

Figura 3.26: Modelo original sem tecelagem

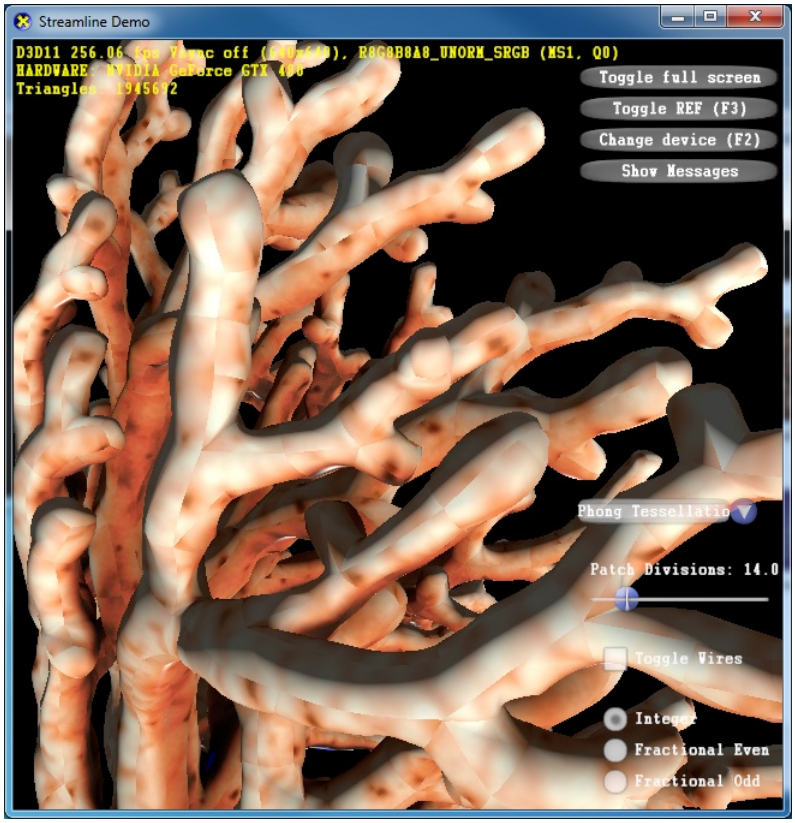

Figura 3.27: Modelo usando o algoritmo Phong Tessellation

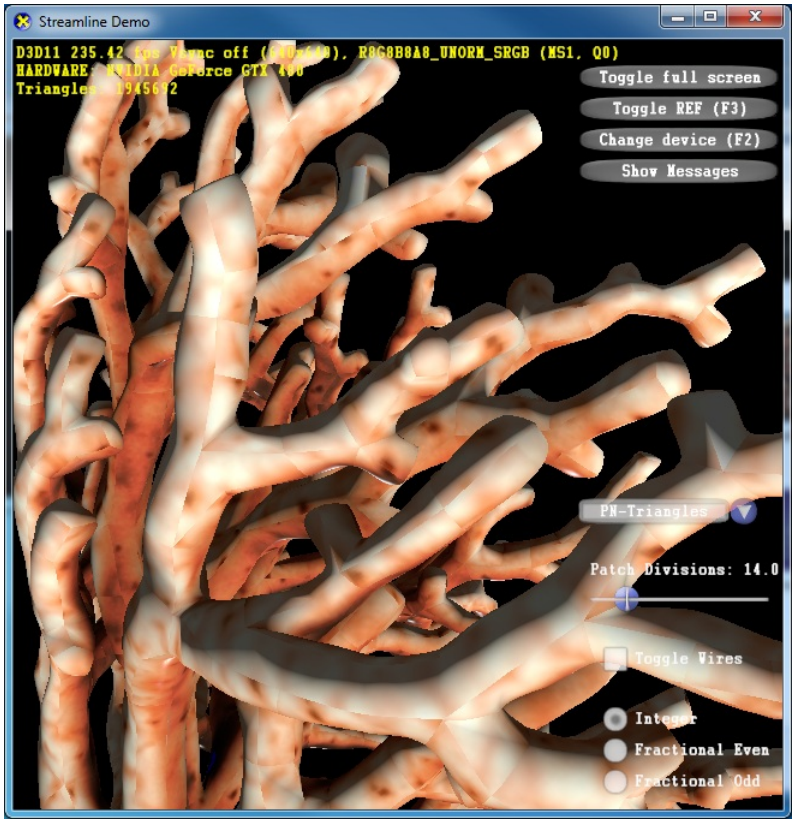

Figura 3.28: Modelo usando o algoritmo PN-Triangles e normais quadráticas

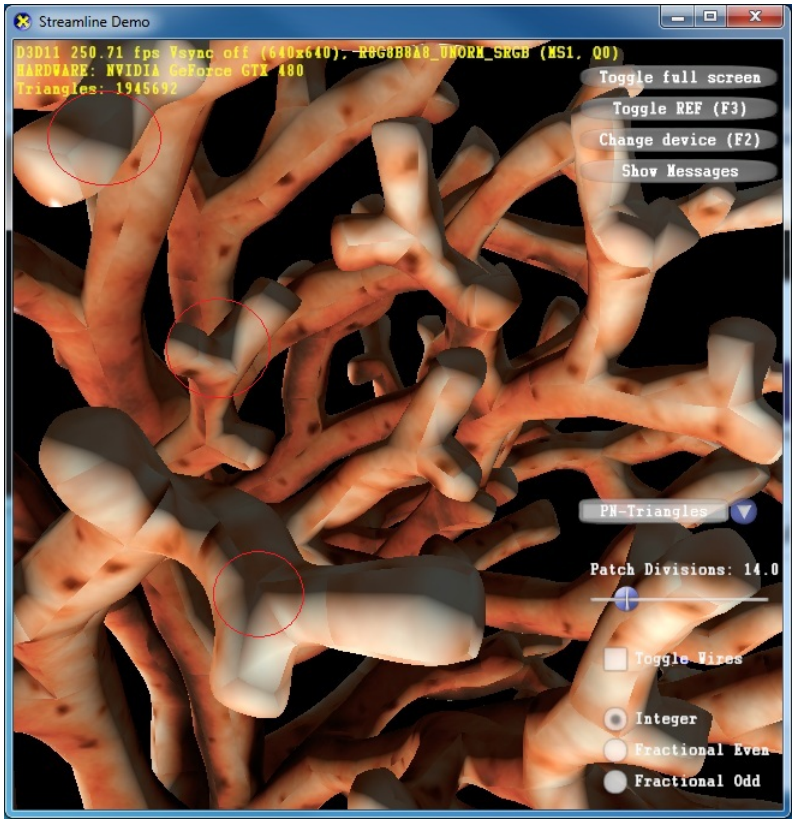

Figura 3.29: PN-Triangles com interpolação linear das normais

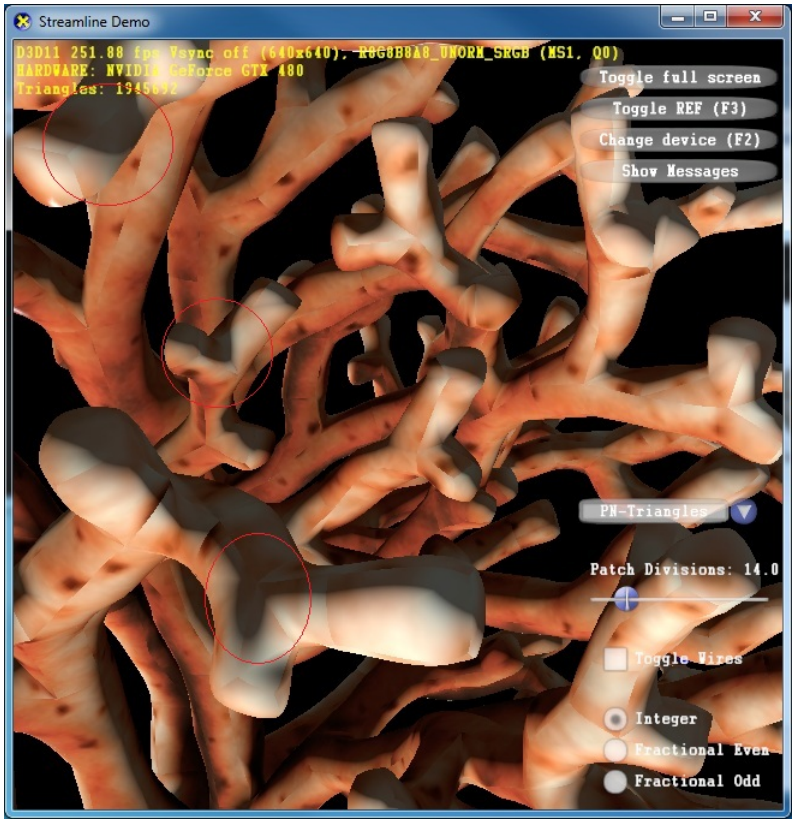

Figura 3.30: PN-Triangles com interpolação quadrática das normais

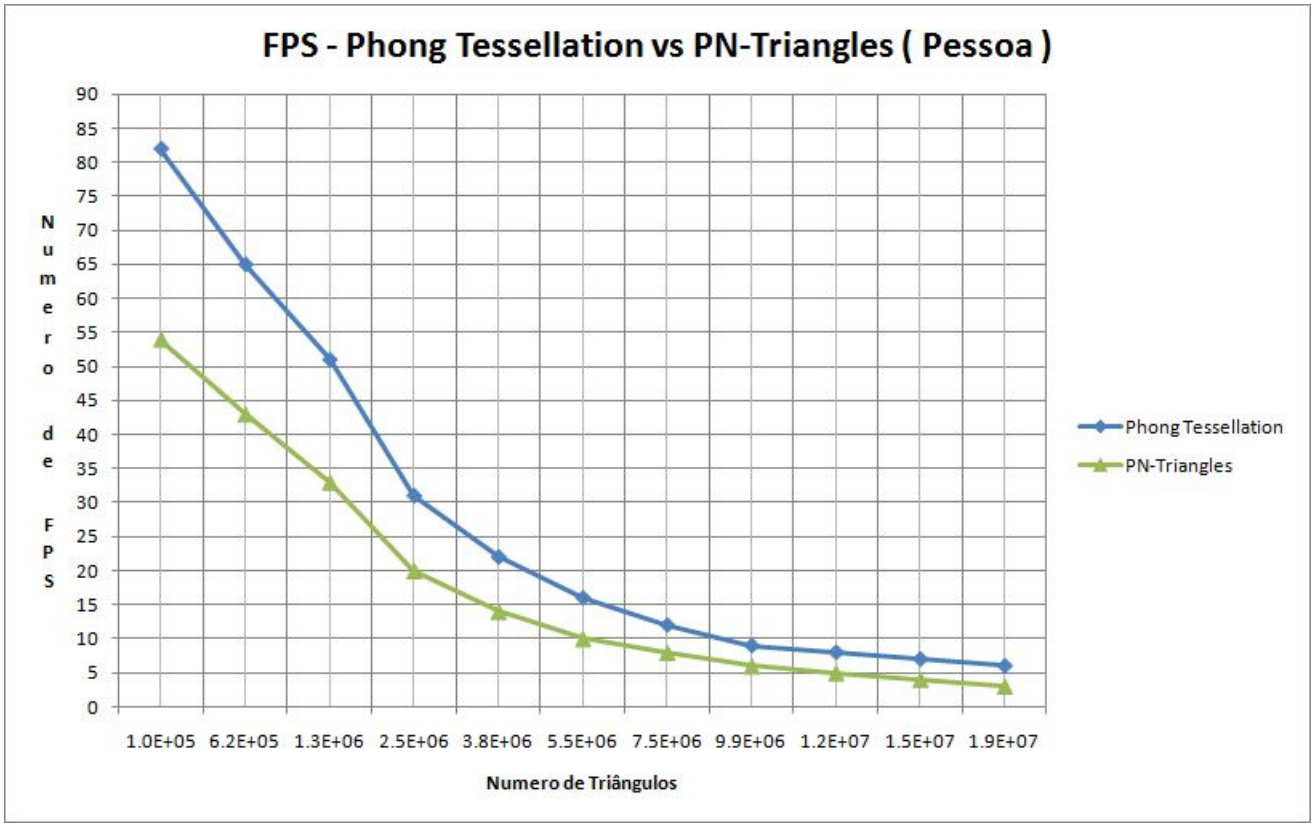

Figura 3.31: Demonstrativo da diferença de performance entre o Phong Tessellation e o PN-Triangles para o modelo da Pessoa

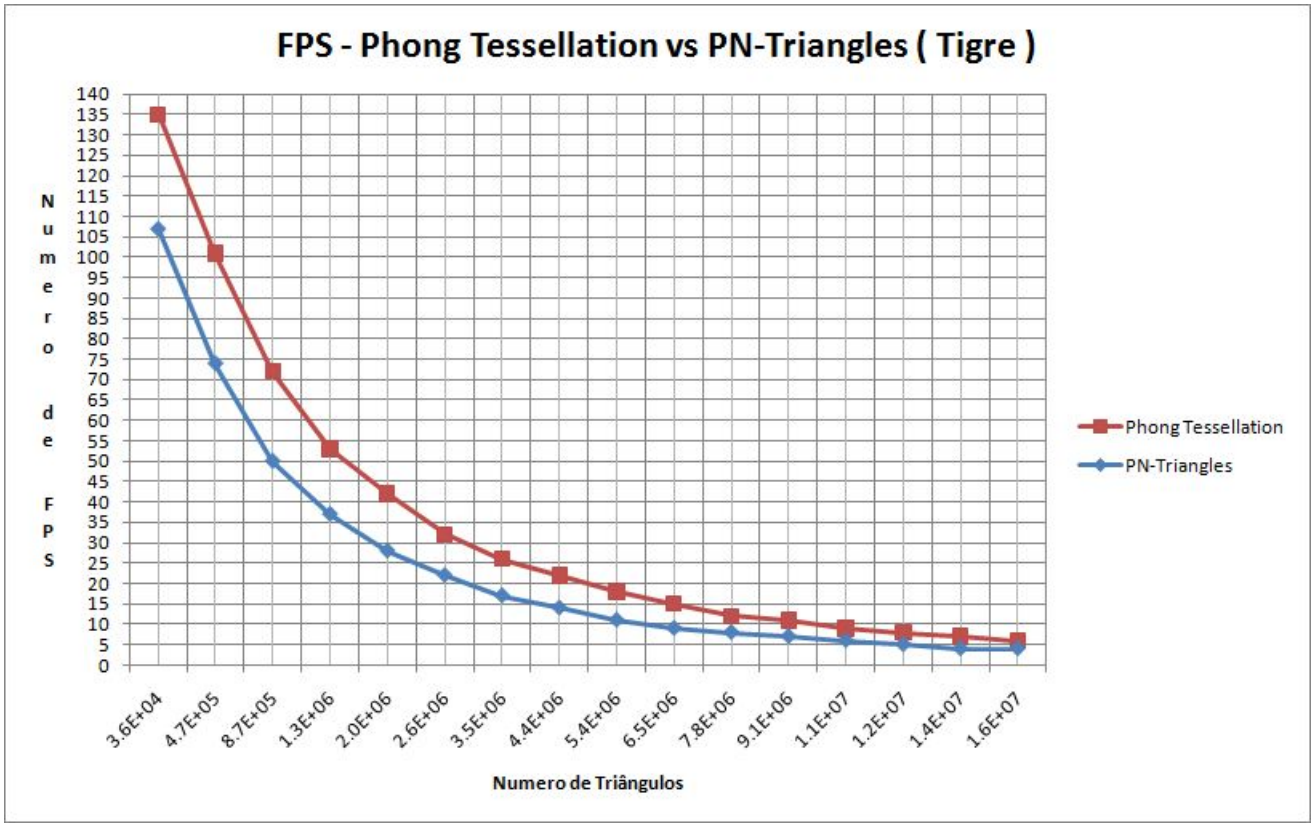

Figura 3.32: Demonstrativo da diferença de performance entre o Phong Tessellation e o PN-Triangles para o modelo do Tigre