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.

PUC-Rio - Certificação Digital Nº 0921335/CA

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 dividos 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*,

PUC-Rio - Certificação Digital Nº 0921335/CA

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]

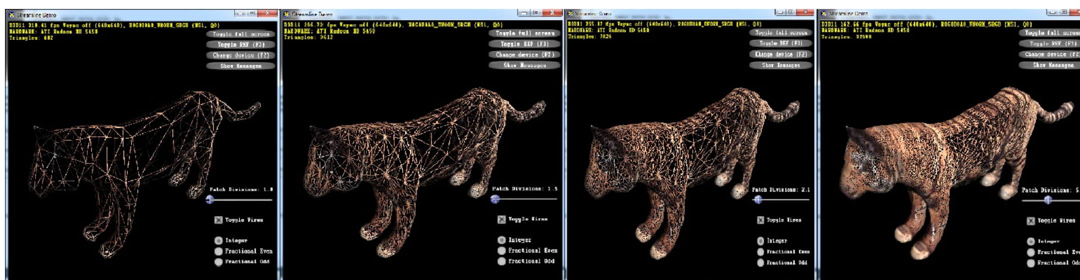

Figura 2.2: Modelo com diferentes níveis de detalhe produzidos na GPU

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

PUC-Rio - Certificação Digital Nº 0921335/CA

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.

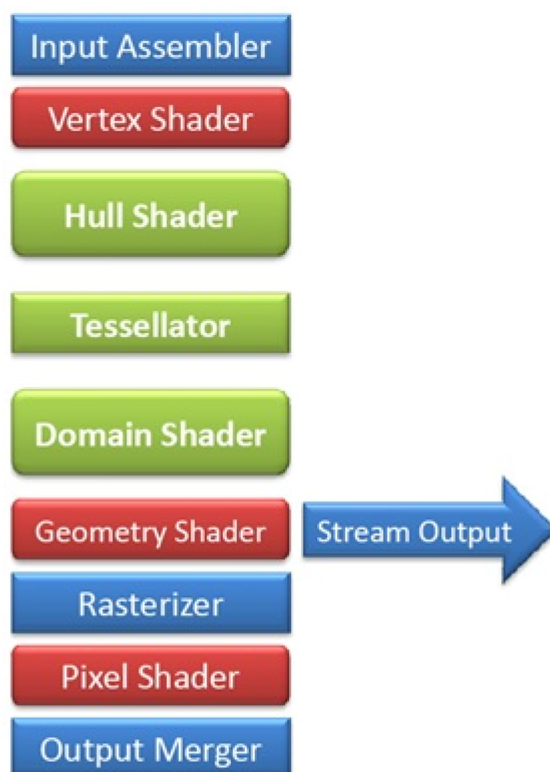

Figura 2.3: O novo pipeline gráfico[Rocco, 2010]

PUC-Rio - Certificação Digital Nº 0921335/CA

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.

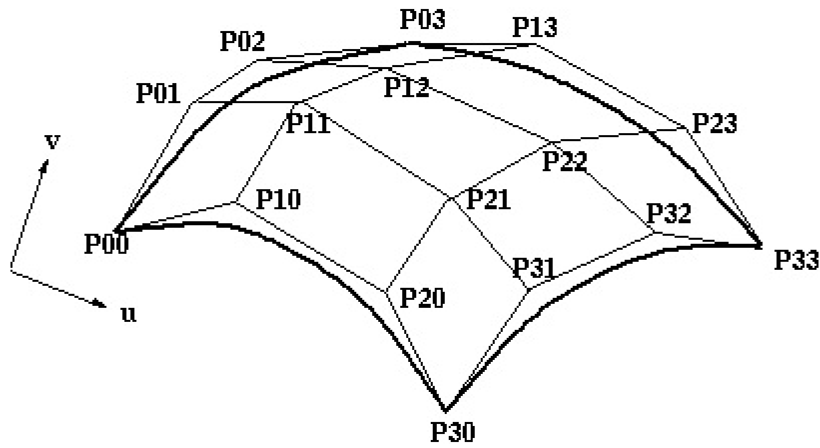

Figura 2.4: Patch com 16 pontos de controle representando uma superfície de Bézier

PUC-Rio - Certificação Digital Nº 0921335/CA

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

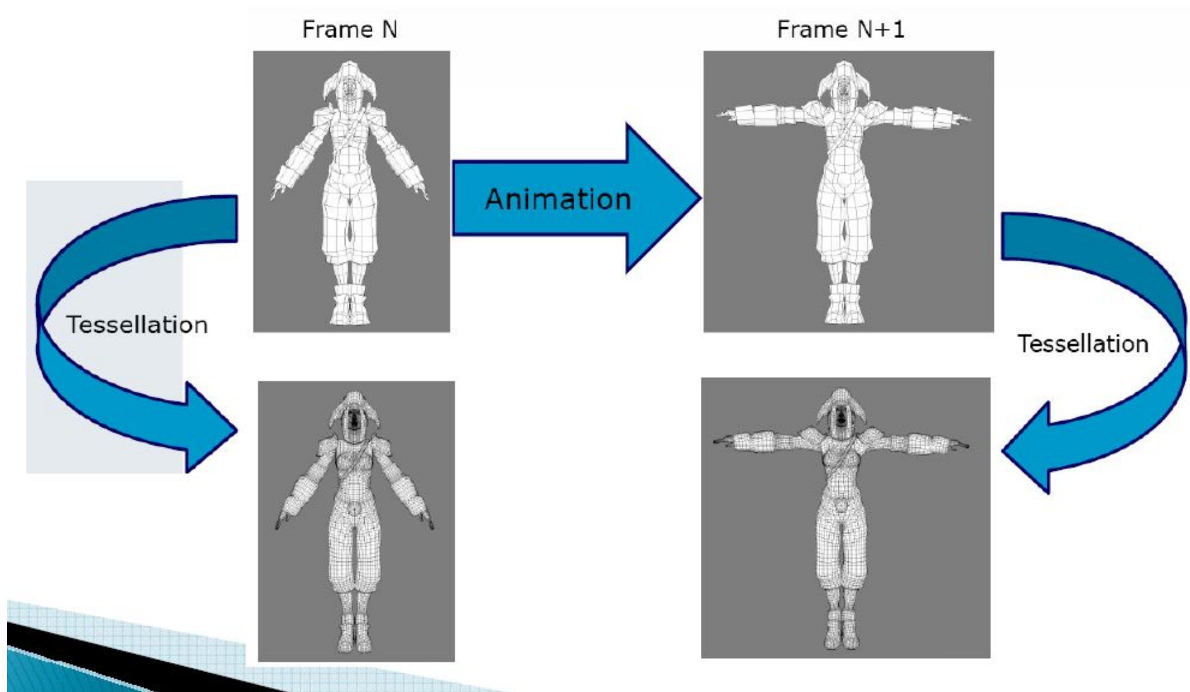

Figura 2.5: Economia ao animar modelo em baixa frequência [Tatarinov, 2008]

PUC-Rio - Certificação Digital Nº 0921335/CA

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.

PUC-Rio - Certificação Digital Nº 0921335/CA

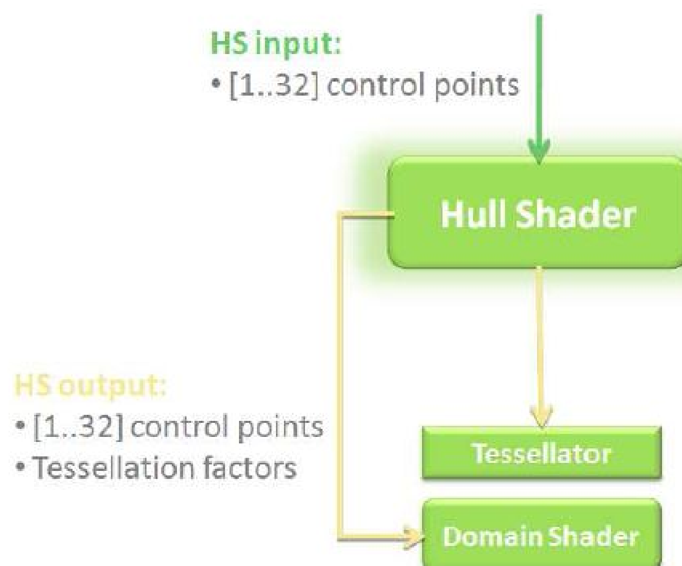

Figura 2.6: O Hull Shader [Ni et al., 2009]

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

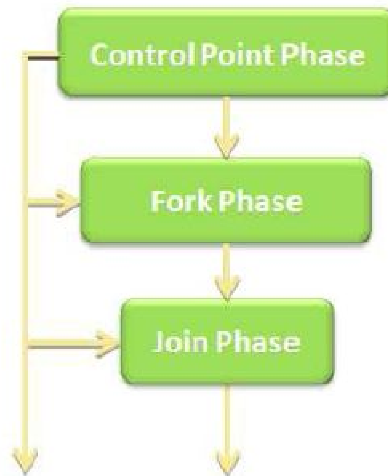

Figura 2.7: Fases do Hull Shader [Ni et al., 2009]

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

PUC-Rio - Certificação Digital Nº 0921335/CA

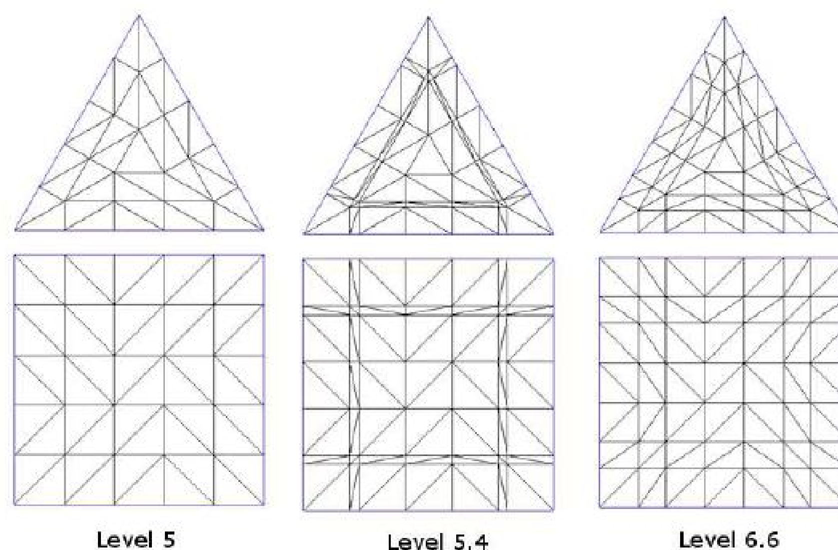

Figura 2.8: Exemplos de uso do Tessellator em diferentes domínios

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

PUC-Rio - Certificação Digital Nº 0921335/CA

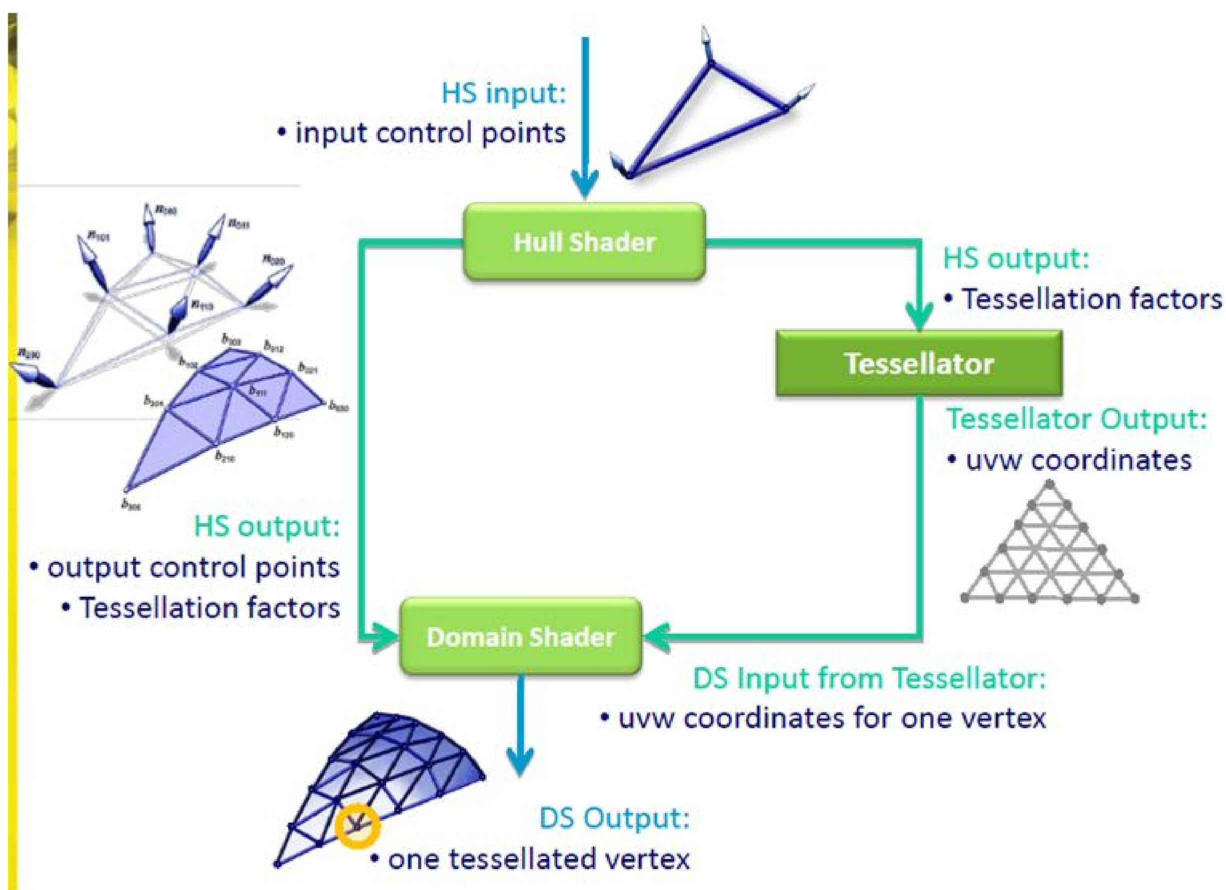

Figura 2.9: Fluxograma da nova parte do pipeline [Tariq, 2009]

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

PUC-Rio - Certificação Digital Nº 0921335/CA

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;
```

PUC-Rio - Certificação Digital Nº 0921335/CA

```
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;
```

PUC-Rio - Certificação Digital Nº 0921335/CA

```
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 }
```

PUC-Rio - Certificação Digital Nº 0921335/CA

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.

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

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

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

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

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);
```

PUC-Rio - Certificação Digital Nº 0921335/CA

```
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),
```

PUC-Rio - Certificação Digital Nº 0921335/CA

```
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.

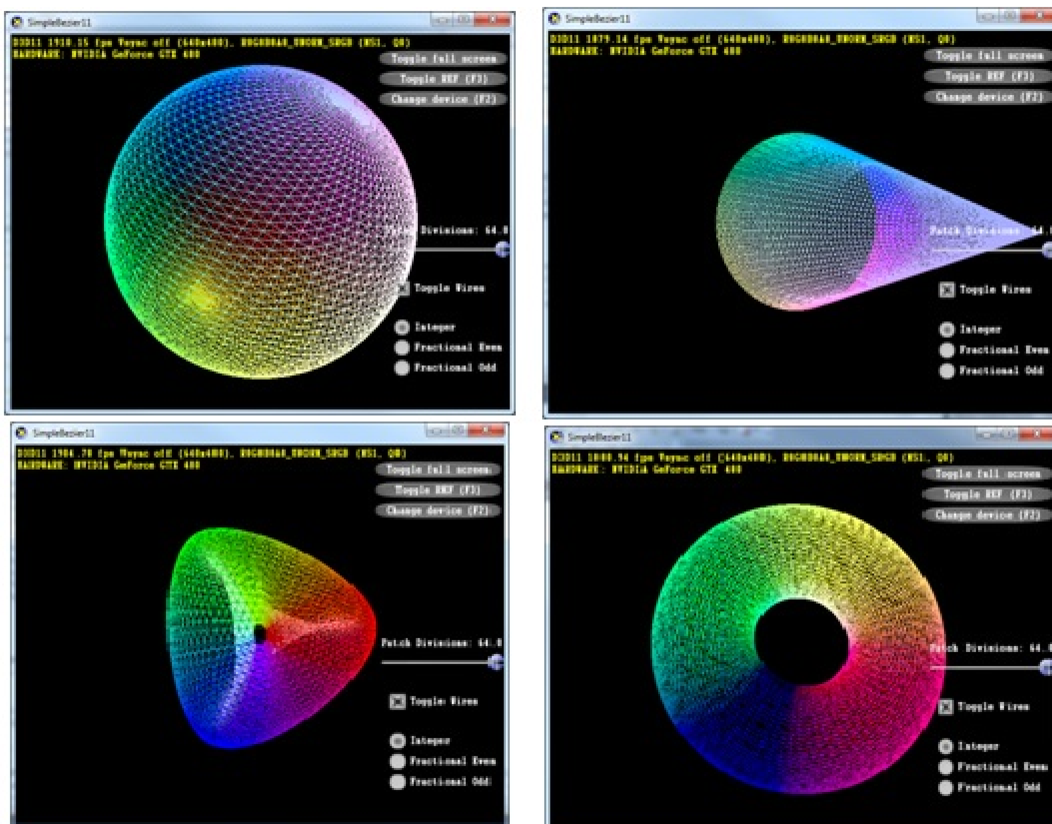

Figura 2.10: Superfícies paramétricas geradas em GPU

PUC-Rio - Certificação Digital Nº 0921335/CA

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

PUC-Rio - Certificação Digital Nº 0921335/CA

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.

PUC-Rio - Certificação Digital Nº 0921335/CA

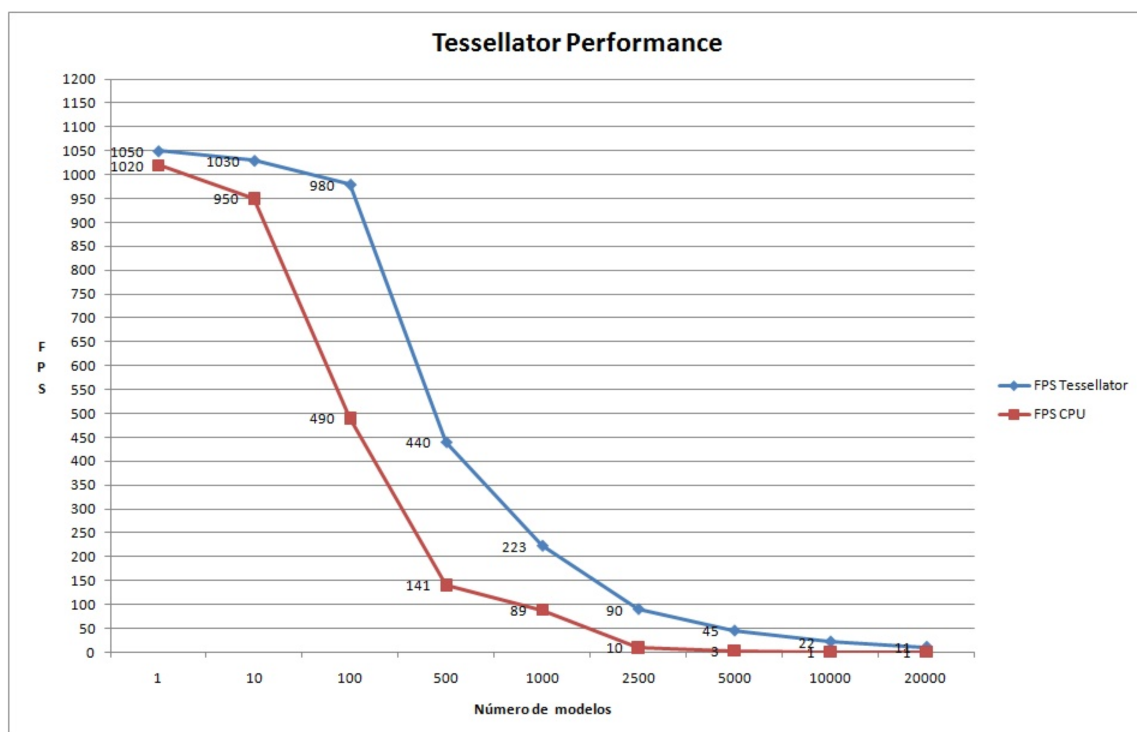

Figura 2.11: Gráfico representando o ganho de performance do Tessellator

PUC-Rio - Certificação Digital Nº 0921335/CA