# Ganho de Performance e Economia de Largura de Banda com o uso do Tessellator

Gustavo Nunes
Rodrigo Braga
Alexandre Valdetaro
Alberto Raposo
Bruno Feijó
Pontifícia Universidade Católica do Rio de Janeiro - Departamento de Informática

## Resumo

Um dos maiores gargalos do pipeline gráfico é a largura de banda disponível entre a GPU e a CPU. Uma vez que largura de banda é limitada pelo hardware, o foco das novas APIs gráficas foi a criação de funcionalidades de para otimizar o uso da banda. As APIs gráficas do DirectX11 e OpenGL4 adicionaram um excelente mecanismo para a redução do volume de tranferência entre CPU-GPU, o Tessellator, que permite a criação de vértices em massa diretamente na GPU. O presente artigo tem por objetivo fazer uma análise de performance do Tessellator para avaliar o ganho de performance devido à economia de largura de banda que se têm ao criar vértices na placa de vídeo em comparação com a abordagem antiga de usar o barramento para transmitir os vértices para a GPU. Nossos resultados mostraram uma melhora de performance de até 8x com o uso do Tessellator em alguns cenários.

**Keywords::** Tessellator, Shader Model 5.0, OpenGL4, DirectX11, Memory Bandwidth, HLSL, GLSL

**Author's Contact:**

{gustavo,rodrigo,alexandre}@xtunt.com
{bfeijo,abraposo}@inf.puc-rio.br

## 1 Introdução

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

Para ajudar na elaboração destes novos algoritmos, o pipeline gráfico vem se tornando cada vez mais flexível. Em 2006, foi lançado o DirectX10 com suporte ao SM 4.0(Shader Model 4.0). Nele foi introduzido um novo estágio no pipeline, o Geometry Shader. Pela primeira vez, era possível criar vértices em GPU. Os programadores tinham acesso a cada triângulo transferido pela placa gráfica, incluindo suas adjacências. Muitos pensaram que poderiam criar milhões de vértices em tempo de execução na placa gráfica. Porém, apesar de o Geometry Shader ser útil para uma série de algoritmos, ele é um estágio lento. O controle de fluxo ou criação de muitos vértices neste estágio faz com que a aplicação tenha uma queda de desempenho drástica, tornando inviável a renderização em tempo real. Vale citar também a técnica de Geometry Instancing introduzida no Shader Model 3.0. Esta técnica permite a replicação de malhas apenas com parâmetros diferentes. Apesar de ser um ótima solução para muitas aplicações, o Geometry Instancing não é uma otimização genérica para o gargalo de transferência CPU-GPU já que é utilizado para duplicar malhas; e não para alterá-las fisicamente e independentemente.

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

Por outro lado, um dos recursos mais baratos no pipeline é a alta capacidade de processamento da GPU. Sendo assim, uma abordagem interessante é poder trocar a transferência de memória por operações aritméticas na GPU. Com a troca, é possível aumentar o desempenho do pipeline gráfico já que são trocados recursos custosos por recursos mais baratos. Com o intuito de atender a essa necessidade, o Shader Model 5.0 [Drone et al. 2010] traz um novo pipeline gráfico, este com duas novas partes programáveis e uma parte fixa. Estes novos estágios servem para a criação de primitivas em massa, na GPU, a partir de uma quantidade relativamente pequena de dados de entrada transferidos pela CPU. Embora essa operação ainda tenha um custo adicional de processamento na GPU, é possível observar a redução no uso do barramento CPU-GPU. Em Setembro de 2009, chegavam ao mercado as primeiras placas com suporte ao novo pipeline, a série HD5xxx da ATI. Mais tarde, em abril de 2010, a Nvidia lançou a série Geforce 4xx com suporte nativo ao novo pipeline.

Neste trabalho será avaliado o ganho de performance e a economia de largura de banda com o uso do novo pipeline gráfico.

## 2 Trabalhos Relacionados

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 1997], [Hoppe 1996], [Hoppe 1998], [Hu et al. 2010], diminuem o número de vértices de um modelo, dinamicamente, fazendo com que não seja necessário transferir todos os vértices para a GPU quando a câmera está muito longe de uma malha. Porém, quando o modelo está muito próximo da câmera é inevitável transferir todos os vértices. Billboards e point sprites também são usados para evitar a renderização de geometria [Fernando 2003]. No entanto, geralmente, modelos complexos têm o realismo visual comprometido ao serem representados por billboards perto da câmera ou em suas silhuetas.

O uso do Geometry Shader também foi explorado para evitar a sobrecarga no barramento de transferência para a placa gráfica [Lorenz and Döllner 2008], detecção de silhuetas [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. De Toledo e Lévy [Toledo and Lévy 2004], [Toledo and Lévy 2008] propõem o uso de ray-casting no pixel shader para estender o pipeline gráfico e criar novas primitivas. Apesar de apresentar bons resultados visuais, os trabalhos mencionados pelos autores só propõem a criação de primitivas até a quarta ordem, além do fato do ray-casting ser muito intenso em operações aritméticas.

Na literatura, alguns trabalhos já exploram o Tessellator para desenvolver novos algoritmos. [Loop and Schaefer 2008] usa patches bi-cúbicos para a subdivisão de Catmull-Clark[Catmull and Clark 1978]. [Loop et al. 2009] usa patches gregorianos para tirar proveito do Tessellator em domínios triangulares. [Valdetaro et al. 2010] utiliza o Tessellator em um sistema de renderização de terrenos com patches avaliados localmente.

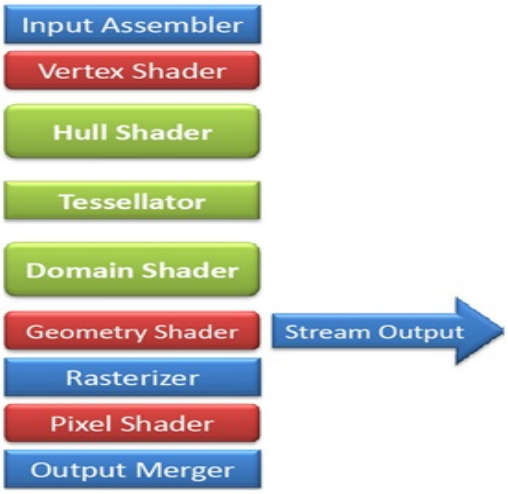


**Figura 1:** *O novo pipeline gráfico*

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 transferindo apenas um vértice. O maior objetivo é avaliar o ganho de performance entre a criação destas geometrias em CPU (transferindo todos os vértices para a GPU) em comparação com nossa abordagem em GPU. No melhor do conhecimento dos autores, não existe trabalho na literatura que compare explicitamente o ganho de performance do Tessellator em comparação com a abordagem tradicional.

# 3 O novo pipeline gráfico

A Figura 1 representa todos os estágios disponíveis nas novas placas de vídeo. Nesta seção vamos detalhar rapidamente os estágios novos do pipeline (Hull Shader, Tessellator e Domain Shader) e seus papéis.

## 3.1 Hull Shader

Depois do Vertex Shader, o Hull Shader é invocado para cada primitiva transferida. 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 controles declarados. 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 sairão dele 16 pontos de controle por primitiva para fazer a mudança de base de um quad para uma Bi-Cúbica de Bézier. 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).

## 3.2 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 definidos pelo Hull Shader. A partir do domínio selecionado (triângulos, quads ou linha), os vértices são criados e as suas coordenadas paramétricas UV/UVW são transferidas para o Domain Shader já 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 dão a impressão visual de estarem 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 é reduzido.

Outra característica importante do Tessellator é a possibilidade de atribuir valores distintos para cada aresta e para o interior do domínio. Desta maneira, duas primitivas vizinhas, mesmo com fatores de tecelagem diferentes para seus interiores não causarão descontinuidades na malha se a aresta comum entre elas tiver o mesmo fator de tecelagem para ambas.

## 3.3 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. Esse estágio é invocado para cada um dos vértices gerados pelo Tessellator. O Tessellator transfere 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 papel do Tessellator é, unicamente, subdividir um domínio (triângulo ou quad) para cada patch que é enviado ao pipeline. Assim, baseado nas coordenadas paramétricas UV/UVW dos vertices de saída do tessellator, e nas informações topológicas de entrada da CPU, os vértices criados devem ser deslocados para sua esperada coordenada de mundo, ou seja, à malha 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.

# 4 Criando superfícies paramétricas na GPU

O objetivo desta seção é fazer uma aplicação simples usando o Tessellator, que permita medir a performance da troca de transferência entre CPU e GPU por operações ALU na GPU. Esse algoritmo segue o trabalho de De Toledo e Lévy[Toledo and Lévy 2004], [Toledo and Lévy 2008] que estende os tipos de primitivas do pipeline gráfico criando superfícies paramétricas na GPU utilizando ray-casting. A diferença é que a presente abordagem requer menos operações aritméticas na GPU do que a abordagem do ray-casting e permite mais flexibilidade, pois permite criar superfícies paramétricas de qualquer ordem. No trabalho de De Toledo é proposta a criação de superfícies até quarta ordem. Vale ressaltar que o tipo de abordagem proposta por De Toledo requer a avaliação de equações complexas para renderizar primitivas acima de quarta ordem, isto exigiria ainda mais operações ALU na GPU, tornando-se pouco aplicável.

A aplicação consiste em renderizar um número grande de torus com e sem o uso do Tessellator e depois avaliar a performance e a economia da largura de banda. Para o Input Assembler será enviado apenas um vértice por torus. O vertex shader irá transferir apenas os dados recebidos pelo o Input Assembler, o Hull Shader deverá especificar quantos triângulos serão gerados para cada torus. Para o presente artigo, serão 8192 triângulos por torus. Além disso, o Hull Shader também precisa especificar em qual domínio o Tessellator irá gerar os vértices. Para o presente artigo, o domínio é quadrado. O papel do Domain Shader é avaliar o torus de acordo com a equação 4-1 e posicionar os vértices em espaço de mundo. Para o Pixel Shader serão transferidas as coordenadas dos vértices em espaço de tela. O Código 1 mostra o Domain Shader utilizado.

$$\begin{aligned} T : (x, y, z) = ((M + Ncos\theta) * cos\varphi, \\ (M + Ncos\theta) * sin\varphi, \\ N * sin\theta)) \\ \theta, \varphi \in [0, 2\pi]; M, N \in \Re \end{aligned} \tag{4-1}$$

onde $M$ e $N$ são os raios: interno e externo.

## 5 Performance do Tessellator

Visto que, na criação de cada primitiva paramétrica, foi transferido apenas 1 vértice da CPU para a GPU, pode-se considerar que esta aplicação mostra-se bastante apropriada para medir o quanto a troca da 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 transferida para a GPU e os FPS foram medidos nas duas abordagens. Os torus da CPU foram transferidos em um único vertex buffer para evitar que muitas chamadas de renderização(*draw calls*) virassem o gargalo da aplicação.

A Tabela 1 e a Figura 2 mostram o ganho de performance em número de FPS com o uso do Tessellator em contraste com o mesmo número de modelos transferidos pela CPU.

A Tabela 2 mostra a economia de memória (em MB) da abordagem feita em CPU 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).

## 6 Conclusão e Trabalhos Futuros

Analisando os gráficos e tabelas com os resultados, 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 1 e pela Figura 2 é notável a brusca queda de desempenho da aplicação feita em CPU após a faixa de 20 milhões de triângulos. Enquanto isso, com o uso do Tessellator, é possível manter taxas interativas de FPS usando até 163 milhões de triângulos por frame. Para taxas interativas de FPS ( $\geq$ 10 ), o máximo que a abordagem em CPU 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 dos vértices criados na CPU serem transferidos para GPU, normalmente eles são alocados na memória principal. A Tabela 2 mostra um gasto de memória de até 5.8GB ao usar a abordagem da CPU. Evitar o consumo desta quantidade de memória em uma aplicação em tempo real implica em um ganho considerável.

Vale lembrar que o Tessellator não é simplesmente um Geometry Shader com mais desempenho, são estágios diferentes do pipeline e que podem se complementar. Uma série de algoritmos ainda necessitam 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.

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 retornar para a CPU a geometria gerada 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.

Como trabalhos futuros pretende-se extender os trabalhos [Toledo and Lévy 2008] e [Toledo et al. 2008] que usam o pixel shader para criar novas primitivas e visualizar estruturas industriais e modelos CAD. O uso do Tessellator nesses trabalhos pode impactar em um ganho significativo.

## Agradecimentos

Os autores gostariam de agradecer ao CNPq, CAPES, FAPERJ, Tecgraf, ICAD/VisionLab e Petrobras pelo suporte financeiro.

| Quantidade de Torus | Triângulos (milhões) | FPS Tessellator | FPS CPU |
|---|---|---|---|
| 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 1:** *Ganho de performance do Tessellator*

| Quantidade de Torus | Memória c/ CPU | Memória c/ GPU |
|---|---|---|
| 20000 | 5898.24 | 0.24 |
| 10000 | 2949.12 | 0.12 |
| 5000 | 1479.56 | 0.06 |
| 2500 | 737.28 | 0.03 |
| 1000 | 294.91 | 0.012 |
| 500 | 147.45 | 0.006 |
| 100 | 29.49 | 0.0012 |
| 10 | 2.94 | 0.00012 |
| 1 | 0.29 | 0.000012 |

**Tabela 2:** *Tabela mostrando a economia de memória com o uso do Tessellator (em Megabytes)*

## Referências

BHATT, A. V. 2004. Pci-express specification. In *Intel Whitepaper*.

CATMULL, E., AND CLARK, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design 10*, 350–355.

DECORO, C., AND TATARCHUK, N. 2007. Real-time mesh simplification using the GPU. In *Symposium on Interactive 3D Graphics (I3D)*, vol. 2007, 6.

DOSS, J. 2008. Inking the cube: Edge detection with direct3d10. In *Gamasutra Article*.

DRONE, S., LEE, M., AND ONEPPO, M. 2010. Direct3d 11 tessellation. In *Microsoft Gamefest 2008*.

DRONE, S. 2007. Real-time particle systems on the gpu in dynamic environments. In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, SIGGRAPH '07, 80–96.

FERNANDO, R., 2003. The cg tutorial: The definitive guide to programmable real-time graphics.

HOPPE, H. 1996. Progressive Meshes. In *SIGGRAPH96*, ACM Press/ACM SIGGRAPH, New York, H. Rushmeier, Ed., 99–108.

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

HOPPE, H. 1998. Efficient implementation of progressive meshes. *Computers and Graphics 22*, 1, 27–36.

HU, L., SANDER, P. V., AND HOPPE, H. 2010. Parallel view-dependent level-of-detail control. *IEEE Transactions on Visualization and Computer Graphics 16*, 718–728.

JIM BREWER, J. S. 2004. Pci-express technology. In *Intel Whitepaper*.

LOOP, C., AND SCHAEFER, S. 2008. Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph. 27* (March), 8:1–8:11.

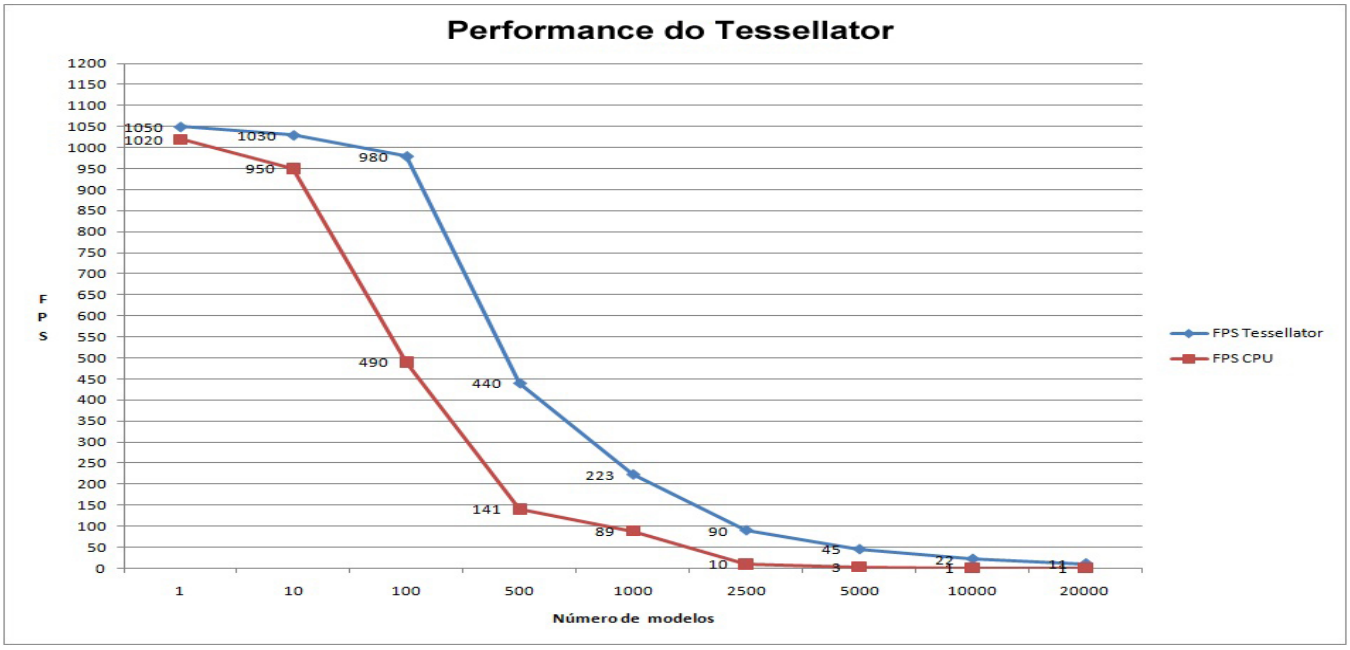


**Figura 2:** *Gráfico representando o ganho de performance do Tessellator*

LOOP, C., SCHAEFER, S., NI, T., AND CASTAÑO, I. 2009. Approximating subdivision surfaces with gregory patches for hardware tessellation. *ACM Trans. Graph. 28* (December), 151:1–151:9.

LORENZ, H., AND DÖLLNER, J. 2008. Dynamic mesh refinement on gpu using geometry shaders. In *16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG) - Full Papers*, 97–104.

OWENS, J., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J., AND PHILLIPS, J. May, 2008. Gpu computing. In *Proceedings of the IEEE*, 96–98.

SDK, D. 2007. Cubemapgs sample. In *Microsoft DirectX SDK*.

TATARCHUK, N., SHOPF, J., AND DECORO, C. 2007. Real-time isosurface extraction using the gpu programmable geometry pipeline. In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, SIGGRAPH '07, 122–137.

TOLEDO, R. D., AND LÉVY, B. 2004. Extending the graphic pipeline with new gpu-accelerated primitives. In *International gOcad Meeting, Nancy, France*.

TOLEDO, R. D., AND LÉVY, B. 2008. Visualization of industrial structures with implicit gpu primitives. In *ISVC (1)*, 139–150.

TOLEDO, R. D., LÉVY, B., AND PAUL, J.-C. 2008. Reverse engineering for industrial-plant cad models. In *TMCE, Tools and Methods for Competitive Engineering, 2008*, 1021–1034.

VALDETARO, A., NUNES, G., RAPOSO, A., FEIJO, B., AND TOLEDO, R. D. 2010. Lod terrain rendering by local prallel processing on gpu. *IX Brazilian symposium on computer games and digital entertainment*.

**Código 1:** *Domain Shader que avalia a equação paramétrica do Torus*

```
[domain("quad")]
DS_OUTPUT DS(HS_CONSTANT_DATA_OUTPUT input,
             float2 UV : SV_DomainLocation,
             const OutputPatch<HS_OUTPUT,
             OUTPUT_PATCH_SIZE> inputPatch)
{
 DS_OUTPUT Output;

 float3 position = float3(0.0,0.0,0.0);
 float pi2 = 6.28318530;
 float M = 1;
 float N = 0.5;
 float cosS, sinS;

 sincos( pi2 * UV.x, sinS, cosS);
 float cosT, sinT;
 sincos( pi2 * UV.y, sinT, cosT);
 position = float3((M + N * cosT) * cosS,
 (M + N * cosT) * sinS, N * sinT);

 Output.vColor = float3(normalize(position));
 Output.vPosition = mul(float4(position,1),
                        g_mViewProjection);
 Output.vWorldPos = Output.vPosition;

 return Output;
}
```