Analysis and Implementation of Local Subdivision Algorithms in the GPU

Gustavo Nunes
Rodrigo Braga
Alexandre Valdetaro
Alberto Raposo
Bruno Feijó
Pontifical Catholic University of Rio de Janeiro (PUC-Rio) - Department of Informatics

Abstract

Being able to refine a mesh without comprimising the memory bandwidth is a powerful tool that might be explored with the new Tessellator part of the graphics pipeline. In this paper we provide a detailed implementation of two local subdivision algorithms - PN-Triangles and Phong Tessellation - using the new Tessellator pipeline. Moreover, a explicit quality and performance comparison between both algorithms is made. Our results showed that Phong Tessellation has a considerable performance gain in comparison with PN-Triangles when implemented in the current Tessellation hardware. However, the visual quality of PN-Triangles algorithm is visually smoother.

Keywords:: Tessellator, Shader Model 5.0, OpenGL4, DirectX11, Memory Bandwidth, Phong Tessellation, PN-Triangles, Subdivision Surfaces

Author's Contact:

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

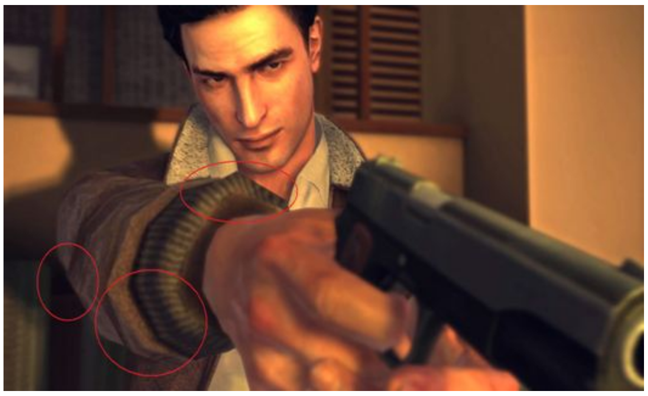


Figure 1: *Screenshot of the game MAFIA II ([2KGames 2010])*

1 Introduction

The subdivision surfaces area is a field that gained attention since 1978 when E. Catmull e J. Clark proposed one of the best works in the area[Catmull and Clark 1978]. Basicaly, each surface that is going to be subdivided has an original mesh, called control mesh or control cage. After the subdivision of the surface the new vertices are moved according to a series of rules that varies according to the subdivision surface algorithm. The surface that is generated (called limit surface) has the same topology of the control mesh.

Although subdivision surfaces algorithms are the standard geometric representation for off-line rendering, until the beginning of the 90s, they didn't got much attention from the real-time rendering industry, specially the games industry. This is due to the fact that computers at the time didn't have enough processing power and the proposed algorithms have a high computational cost. However, in the end of the 90s, with the launch of new 3D video cards, consumers started to have their PC capabilities increased. With this increasing capability it also came an increased contrast among the setups of consumer machines. A game in powerful machines could have high quality textures and models with a good amount of polygons. But in machines that were less capable the number of polygons should be low. The possibility to create a low-poly model, animate it and at the end of the construction having an animated model with any level-of-detail "(LOD)" would be a great help and relieve for the artists and the assets pipeline. Subdivision surfaces algorithms could offer that.

Despite of the modeling softwares allowing the use of subdivision surfaces algorithms it was not practical to insert in a game a single model with many LOD levels in order to attend a huge part of the consumers PCs setups. The media size were limited and also the hard drive space of the users. The ideal solution would be the generation of these LOD levels at execution time. Also, the 3D video cards started to have a great processing capacity and they were relieving the use of the CPU for other tasks. Nevertheless, the 3D video cards had their architecture towards parallelism, in other words, all the vertices manipulation is done in parallel. This factor headed against the existent subdivision surfaces algorithms, all of them were based in the adjacency information of the vertices. In 2001, Vlachos[Vlachos et al. 2001] proposed a purely local subdivision surface algorithm called PN-Triangles. The idea was to implement this algorithm in the video card hardware and the game producers would be able to pass a low-poly model that would be refined by the card. This solution was implemented, but, it was specific of a single graphics card vendor and it wasn't much used because the industry was focused in attend the majority of users and video cards.

In 2008, Boubekeur[Boubekeur and Alexa 2008] proposed another subdivision surface algorithm, named Phong Tessellation, it is also a good candidate for hadware implementation because it is also purely local.

Despite the major advances in the real-time rendering field, the problem with low-poly models still exists until today. An example of this is the game MAFIA released in August,2010. Although per-pixel lightning does a good job hiding the low polygon model at the parts inside the models, at the silhouette the user is able to see sharp edges(Figure 1). One of the goals of the new Tessellator pipeline[Drone et al. 2010] and the new graphics APIs(DirectX11 and OpenGL4) is the solution of this problem.

In this work, we propose to implement both algorithms in the GPU and analyse their quality and performance. Both algorithms are purely local, interpolatives and they need few hardware instructions to evaluate the surface, thus they are appropriate for the implementation in the new Tessellator pipeline and also are strong candidates to become an industry standard in a near future. To the best of our knowledge, there is not a previous work that explicitly compares the GPU implementation, performance and quality of both algorithms (PN-Triangles and Phong Tessellation).

2 Related Work

Many techniques were developed for generating subdvision surfaces, but many are still inefficient to be implemented in 3D game engines. This is due to the fact that the majority of the techniques requires information from the neighborhood of a vertex to apply the subdivision. Implementing this kind of algorithm in hardware requires a bigger effort, it can't be done in a single pass and the evaluation of the surface is very costly. Boubekeur[Boubekeur et al. 2005] extended the PN-Triangles technique placing three scalar values in each triangle vertex, in that way it is able to create a procedural displacement map that enhances the detail quality of the

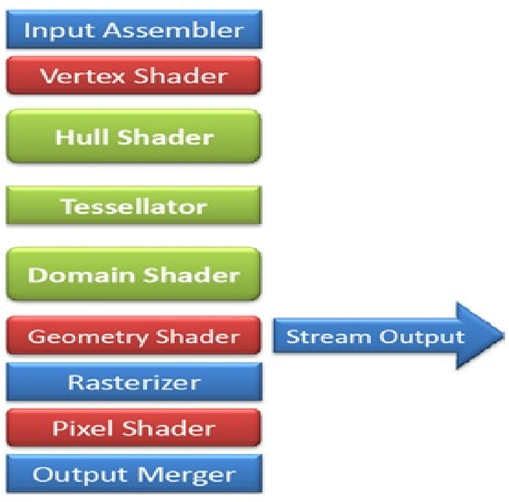


Figure 2: *The new graphics pipeline*

mesh allowing to build creases in the surface. However, this method adds an overhead to the artistic pipeline, because the designer has to setup three aditional values for each vertex in the mesh. With the lack of feedback from the modeling softwares to support this technique, it was not largely used.

The solutions that use adjacency information are more diverse. Catmull and Clark[Catmull and Clark 1978] use a uniform cubic B-Spline in their subdivision method. Doo and Sabin[Doo 1978; Doo and Sabin 1978] based their method in bi-quadratic B-Splines. Loop[Loop 1987] proposes an approximative algorithm that is based on triangle meshes and generates limit surfaces with C2 continuity everywhere except in extraordinary vertices which has C1 continuity. Kobbelt[Kobbelt 2000] proposes an idea that offers a natural adaptive refinement if needed. Zorin et al.[Zorin et al. 1996] proposed an interpolative method based on triangular meshes. Ni et al.[Yeo et al. 2009] presented a method that imitates the shape of Catmull-Clark surfaces using bi-cubic Splines and a new class of patches, called c-patches. Boubekeur and Schlick[Boubekeur and Schlick 2007] avoids the recursion in their method, that is fast, but, geometrically it only guarantees surfaces with C0 continuity. More recently, Loop and Schaefer[Loop and Schaefer 2008] used quartic surfaces with separate normal fields to approximate Catmull-Clark surfaces. This scheme proposed by Loop and Schaefer works only with quads. Later, Loop et al.[Loop et al. 2009] proposed a scheme that uses gregorian patches to approximate subdivision surfaces with triangles.

All those exposed techniques has a basic principle: each polygon is replaced by a polynomial patch that is evaluated later. The only exception is Phong Tessellation that doesn't creates a patch explicitly.

3 The new graphics pipeline

This section will present a quick review of the new graphics pipeline stages (Hull Shader, Tessellator and Domain Shader), for detailed information please refer to [Valdetaro et al. 2010]. Figure 2 represents all the available stages in the new video cards.

3.1 Hull Shader

After the Vertex Shader, the Hull Shader is invoked for each primitive transferred. In the Hull Shader it must be declared how many control points will be output by this stage, the Hull Shader will be invoked based in the amount of control points declared. This serves basically for a base change. For example, the Input Assembler may receive 4 control points per primitive and the Hull Shader may declare an output of 16 control points per primitive in order to do a basis change from a quad to a Bézier Bi-Cubic. Another task of the Hull Shader is to compute and output to the Tessellator the tessellation factors for each edge and also for the interior of the primitive. These factors indicates the amount of subdivision that the Tessellator must subdivide each primitive. It must also be declared in the Hull Shader which subdivision domain that the Tessellator will use (quads, triangles or lines).

3.2 Tessellator

The Tessellator is not a programmable part of the pipeline, it is only configurable. Its duty is to generate vertices according to the tessellation factors transferred by the Hull Shader. According to the selected domain(triangles, quads or lines) it creates vertices and passes its normalized parametric coordinates UV/UVW to the Domain Shader. With those coordinates the Domain Shader knows where the vertices were created and it may move them to where its necessary.

The tessellation factors for the edges and for the interior of the domain varies in the range $[1..64]$. The Tessellator supports integers and fractional tesselating methods. The integer method, as the name suggests, creates vertices only with integers values in the specified range. This kind method may result in *poppings* on the meshes, in other words, the meshes may give a clear impression that they are being modified in execution time. To solve this issue, the Tessellator also provides the fractional method, in this method the vertices are created in a continuos way with a smooth visual transition (see geomorphing [Hoppe 1996]). In this way the popping effect is greatly reduced.

Another important characteristic of the Tessellator is the possibility to attribute distinct values for each edge and for the interior of the domain. This guarantees the flexibility of two neighbor primitives having differents tessellation factors without having discontinuities in the mesh.

3.3 Domain Shader

In this stage occurs the evaluation of the tesselated domain. The Domian Shader may be seen as a Vertex Shader after the tessellation. Each invocation of this stage corresponds to a vertex generated by the Tessellator. The Tesselator passes the coordinates UV/UVW in normalized space in the interval $[0..1]$ and it is Domain Shader task to positionate the generated vertices in world space. It's good to remember that what the Tessellator does is only subdivide a domain for each patch that is pushed into the pipeline, it is the programmer task to use the coordinates UV/UVW and to positionate the vertices based on the control cage of a model for example.

If the Geometry Shader is not enabled, it is Domain Shader task to put the vertices in screen space for the Pixel Shader.

4 Algorithms Review

This section will present a brief review of both algorithms; for detailed explanation please refer do the original papers([Vlachos et al. 2001; Boubekeur and Alexa 2008]).

4.1 PN-Triangles

The main characteristic of the algorithm is the construction of a cubic patch using the information of the vertices of a triangle. The patch b is defined according to the equation 1.

$b : \Re^2 \to \Re^3$, for $w = 1 - u - v$; $u, v, w \geq 0$

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

The normals may be defined in two different ways: a simple linear interpolation or a quadratic function n evaluated according to equation 2.

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

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

Figures 3 and 4 shows the control points b_{ijk} and n_{ijk} related to each patch.

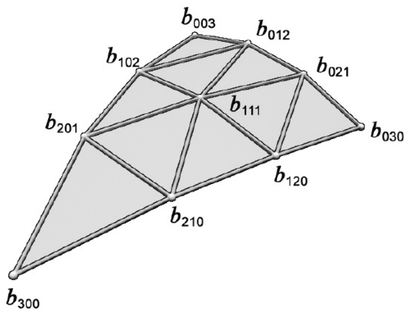


Figure 3: *Control points of the geometry patch[Vlachos et al. 2001]*

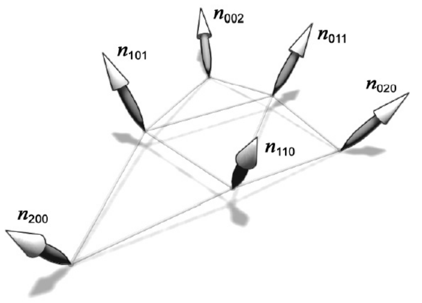


Figure 4: *Control points of the normal patch([Vlachos et al. 2001])*

Given the positions $P_1, P_2, P_3 \in \Re^3$ and the normals $N_1, N_2, N_3 \in \Re^3$ of a triangle, the control points b_{ijk} is made as follows:

1. Put the coefficients b_{ijk} at the intermediary positions $(iP_1 + jP_2 + kP_3)/3$.

2. Let the vertices of the triangle in its corresponding control point(i.e, $b_{300} = P_1, b_{030} = P_2, b_{003} = P_3$)).

3. For each triangle corner project the two coefficients closer to that corner in the tangent plane defined by the normal of the corner.

4. Move the coefficient of the center to the average of the points $b_{012}, b_{102}, b_{120}, b_{210}, b_{201}, b_{021}$ and continue its displacement in the same direction for $1/2$ of the lenght already displaced.

The geometry normals of the PN-Triangles generally don't vary continously from one triangle to another. In the algorithm it is suggested a linear interpolation or a quadratic variation. The problem of the linear interpolation is that it ignores inflections as Figure 5 shows.

To capture the inflexions a coefficient in the middle of each edge is calculated to evaluate the surface n. The average of the normals of each vertex of an edge is calculated and reflected in the plane perpendicular to the edge.

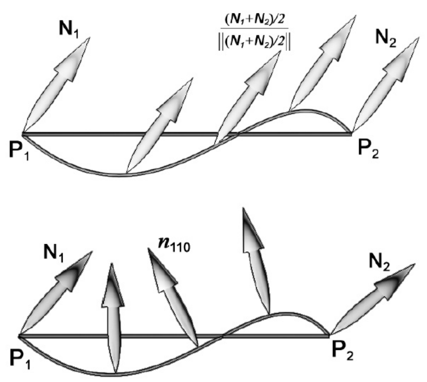


Figure 5: *Linear interpolation of normals(up) and quadratic variation(bottom)[Vlachos et al. 2001]*

4.2 Phong Tessellation

The Phong Tessellation algorithm was thought in a way to complement the Phong Shading. Computing the Phong Shading algorithm per pixel is not costly and it does a good job shading the interior of a mesh, however, it is clear to notice the low amount of polygons when one looks to the silhouette of a low-poly model. In terms of ALU operations the Phong Tessellation algorithm is very lightweight.

A linear tessellation with baricentric interpolation may be defined by the equation 3.

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

$$p(u, v) = (u, v, w)(p_i, p_j, p_k)^T \quad (3)$$

The linear interpolation of the normals that occurs between the Domain Shader and the Pixel Shader is realized in the same way. The result just needs to be normalized in the Pixel Shader, this process is largely used in the Phong Shading. Equation 4 represents this process.

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

$$\begin{aligned} n'(u, v) = (u, v, w)(n_i, n_j, n_k)^T, \\ n(u, v) = n' / \|n'\| \end{aligned} \quad (4)$$

Around each vertex the tangent plane defined by the normal of the vertex points indicates how the geometry should behave locally. The algorithm projects a vertex $v_i$in the tangent plane definied by its normal n_i and does a baricentric interpolation with the information of the other two vertices of the triangle to define the geometry around the neighborhood of v_i. The geometry related to the other two vertices of the triangle v_j and v_k is defined in the same way.

The evaluation of the points generated for each triangle may be simplified by the following process:

1. Do the tessellation linearly

2. Project each tessellated vertex orthogonally in the three tangent plans defined by the normals of the triangle

3. Do the baricentric interpolation of the projected points. This is the final position of the vertex.

Figure 6 show the process described above. The algorithm has few ALU operations, it is necessary only three projection and two linear interpolations to achieve the final position of a vertex.

Let $\pi_\mathbf{i}(\mathbf{q}) = \mathbf{q} - ((\mathbf{q} - \mathbf{p_i})^\mathbf{T} \mathbf{n_i})\mathbf{n_i}$ be the orthogonal projection of $\mathbf{q}$ in the plane defined by $\mathbf{p_i}$ e $\mathbf{n_i}$. Then Phong Tessellation may be defined as:

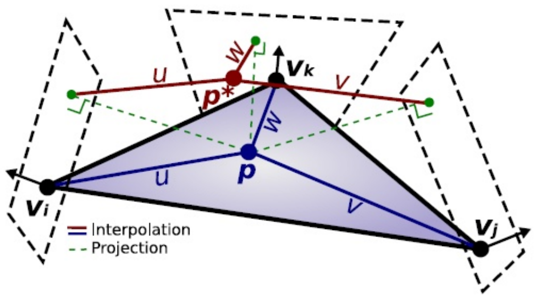


Figure 6: *Projections and interpolations of Phong Tessellation[Boubekeur and Alexa 2008]*

$$p^*(u,v) = (u,v,w)\begin{pmatrix}\pi_i(\mathbf{p}(u,v))\\ \pi_j(\mathbf{p}(u,v))\\ \pi_k(\mathbf{p}(u,v))\end{pmatrix} \qquad (5)$$

5 GPU Implementation

For GPU implementation, both algorithms don't need any pre-processing step or adaptation by the artists. Thus, any mesh that was created in the past is able to be used without additional work. The only requirement is that it must be a triangle mesh. For quad meshes PN-Triangles analogous please refer to [Peters 2008]. The Phong Tessellation may also be easily adapted for quad meshes.

5.1 PN-Triangles

The PN-Triangles implementation fits well in the new Tessellator pipeline. Basically we have to calculate the control points of the geometry and normal fields in the Hull Shader and evaluate the surface in the Domain Shader.

The Vertex Shader just repasses the information to the Hull Shader, it will receive the data for each vertex(Position, Normal and TexCoords) and will send to the Hull Shader. Due to space purpose the Vertex Shader code will be omitted in this paper. Of course that, if there are any animation to be made to the mesh it should be done by the Vertex Shader.

The Hull Shader will receive the Vertex Shader output and it will be executed once per primitive. The patch coefficients for the geometry and the normals should be calculated according to proceedings explained in the previous section. Also, the Hull Shader must output for the Tessellator the tessellation factors for each edge and for the interior of the domain.

Please refer to code listing 1 for the entire Hull Shader code.

With the calculated control points and the new vertices generated by the Tessellator, the Domain Shader evaluates the surface according to equations 1 and 2. Also, as there is no use of the Geometry Shader, the Domain Shader must output the vertices to screen space for rasterization. The texture coordinates are linearly interpolated. Please refer to code listing 2 for the entire Domain Shader code.

The Pixel Shader does a basic per-pixel lightning Phong Shading and, for this paper, it will be omitted.

5.2 Phong Tessellation

Just like PN-Triangles, the Phong Tessellation algorithm doesn't require any changes in the assets pipeline to use it. Models that weren't done with Phong Tessellation in mind may be used without problems. However, in case that the subdivision doesn't visually satisfy the author suggests a control factor α per vertex of the model, in such case the artists would had to do a big re-work and the algorithm would be less practical. The objective here is to analyse the raw algorithm, without any need of artistic intervention.

Phong Tessellation also fits well in the new Tessellator pipeline. Basically all the work will be done by the Domain Shader. The Vertex Shader, just like PN-Triangles, will just pass forward the data given by the Input Assembler. As there is no explicitly patch created by the algorithm, thus the Hull Shader just needs to output the Tessellation factor to the Tessellator. Please refer to code listing 3 for the entire Hull Shader code.

In The Domain Shader we will first create an auxiliary function that defines a orthogonal projection in a plan. The function receives the normal of the plane to project, a point in the plane, the point to be projected and it returns the point projected. Then we must follow the steps described in the previou section. First a baricentric linear interpolation in the triangle plane, three orthogonal projections using the auxiliar function described above and then another linear interpolation based in the projections to find the final vertex position. The normals and the texture coordinates follows a linear interpolation also. Please refer to code listing 4 for the auxiliary projection function and the listing 5 for the entire Domain Shader.

6 Results

In this section we are going to analyse the algorithms both in terms of image quality and performance.

6.1 Quality

All the figures in this section were taken without any anti-aliasing correction or image post-effect. Also, a per-pixel lightning pixel shader with Phong Shading was used to light the models presented. Both algorithms were compared with the same tessellation factors for the edges and the interior of a primitive. However, we used both quadratic patch and linear interpolation for normals in the PN-Triangles algorithm. For the Phong Tessellation algorithm we did a linear interpolation of the normals.

Although both algorithms have only guaranteed C_0 continuity, Figure 9 - using PN-triangles - shows a much more smooth continuity when compared do Figure 8 - using Phong Tessellation and also to the original mesh(Figure 7).

Figures 10 and 11 shows some difference between the quality of shadows generated by the lightning. The red circles shows more sharp shadows in the figure with linear interpolation of the normals and in the case of quadratic normals a more smooth shadow may be noticed. The green circles shows a characteristic case of a inflection point of the triangles normals. The cloth in the green circle seems to be slightly wrinkled in the quadratic normals case. However, in the linear interpolation case, this wrinkle is not taken into account.

Figures 14 and 13 shows clearly the better smooth of the PN-Triangles algorithm when compared to the Phong Tessellation and to the original mesh(Figure 12). Because this tiger model doesn't have much curvature, the linear interpolation didn't have any major loss when compared to the quadratic interpolation of the normals. A really minor deviation might be seen in the red circles of Figures 15 and 16.

Once again, looking at Figures 17, 18 and 19 we may see that the triangles created by Phong Tessellation present a less smooth continuity when compared to PN-Triangles algorithm. However, due to the fact that this is an organic model, the deformation made by Phong Tessellation provided a visually good result.

6.2 Performance

The PC setup that held the tests was an Intel Core2Quad Q6600 with 4GB of RAM and a RadeonHD 8450 video card. We tested a scene with many models of the same type(Tiger or Person) without the utilization of geometry instancing. We avoided to place a single model with a high tesselation factor to prevent an overload in the rastezirer due to the creation of micropolygons[Fatahalian 2010]. The lightning was also turned off to avoid a pixel shader overload.

Person - Number of triangles (million)	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

Table 1: *FPS of the Person mesh using both algorithms*

Person - Number of triangles (million)	Gain (%)	Gain (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

Table 2: *Percentage and Absolute gain of Phong Tessellation under PN-Triangles for person mesh*

The objective was to let the ALU operations of both algorithms be the main bottleneck.

Tables 1 and 3 shows the FPS and the amount of triangles for the Person and the Tiger models respectively.

Figures 20 and 21 shows the results expressed in tables 1 and 3.

Tables 4 and 2 shows the percentual and absolute gain (in FPS) of the Phong Tessellation algorithm against the PN-Triangles.

7 Conclusion and Future Works

This paper presented a detailed GPU implementation of two practical algorithms for Real-Time model refinement. The performance and quality analysis of both algorithms on modern GPUs poses a powerful tool for developers to choose which algorithm to use in a specific situation.

Tiger - Number of triangles (million)	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

Table 3: *FPS of the Tiger mesh using both algorithms*

Tiger - Number of triangles (million)	Gain (%)	Gain (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

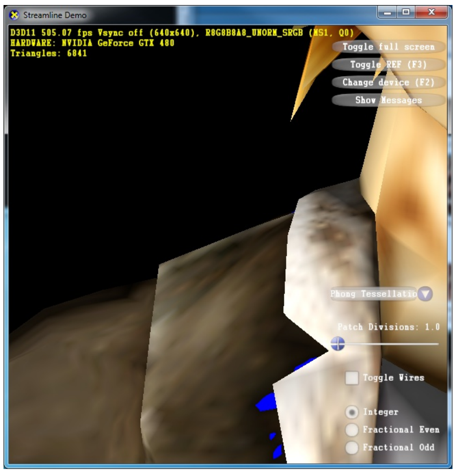
Table 4: *Percentage and Absolute gain of Phong Tessellation under PN-Triangles for tiger mesh*

Based on the Figures and the results presented in the previous section, it is noticed that the Phong Tessellation has a considerable performance gain when compared against the PN-Triangles algorithm (40% gain in average). However, Figures 8 and 9 shows a much better quality of the PN-Triangles method. Moreover, in some cases (Figure 11) the quadratic interpolation showed better results.

For games or very dynamic applications where the user won't be able to pay attention to minor details the Phong Tessellation algorithm might be a great option, specially for silhouette refinement, where a minor increase of the tessellation factor at the silhouette would solve cases like Figure 1. But for applications or games that require a better visual quality the PN-Triangles algorithm should be considered because of its high subdivision quality. One might even consider an option of using Phong Tessellation for a lower end game configuration and PN-Triangles for a higher game setting.

Acknowledgment

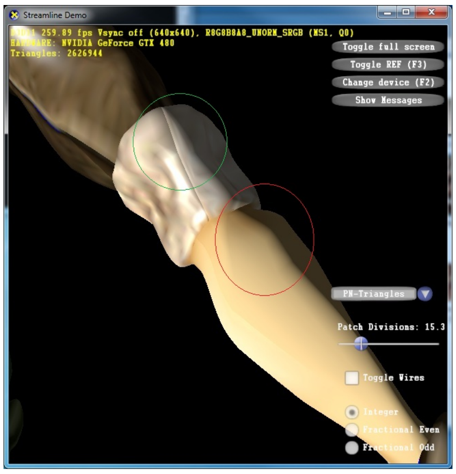
The authors would like to thank CNPq, CAPES, FAPERJ, Tecgraf, ICAD/VisionLab and Petrobras for the financing support.



```
D3D11 505.07 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 6841
```

Patch Divisions: 1.0

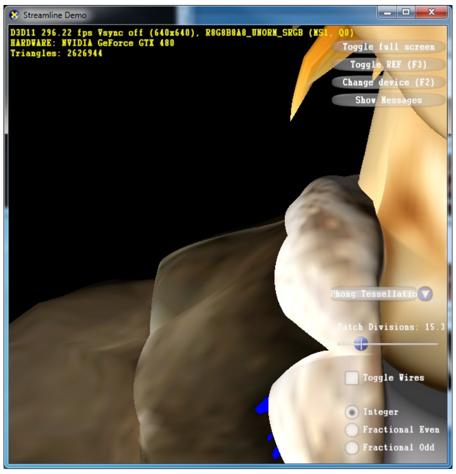
Figure 7: *Original model without any subdivision*



```
D3D11 259.89 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 2626944
```

Patch Divisions: 15.3

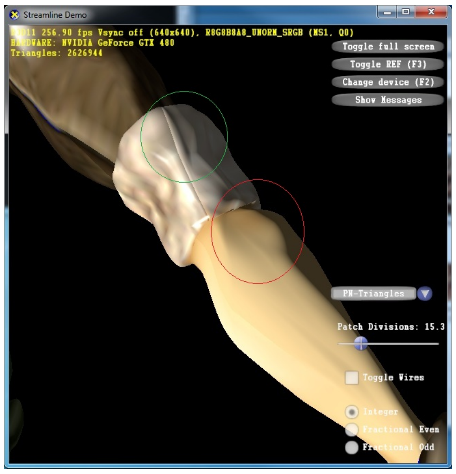
Figure 10: *PN-Triangles with linearly interpolated normals*



```
D3D11 296.22 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 2626944
```

Patch Divisions: 15.3

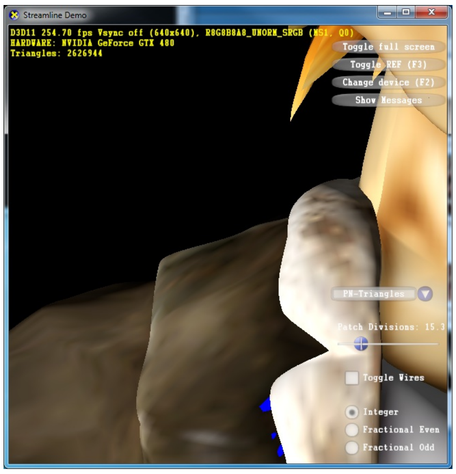
Figure 8: *Model using the Phong Tessellation algorithm*



```
D3D11 256.90 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 2626944
```

Patch Divisions: 15.3

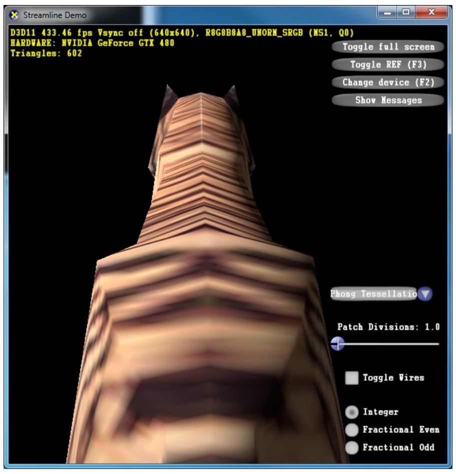
Figure 11: *PN-Triangles with quadratic normals interpolation*



```
D3D11 254.70 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 2626944
```

Patch Divisions: 15.3

Figure 9: *Model using the PN-Triangles algorithm*



```
D3D11 433.46 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 602
```

Patch Divisions: 1.0

Figure 12: *Original model without any subdivision*

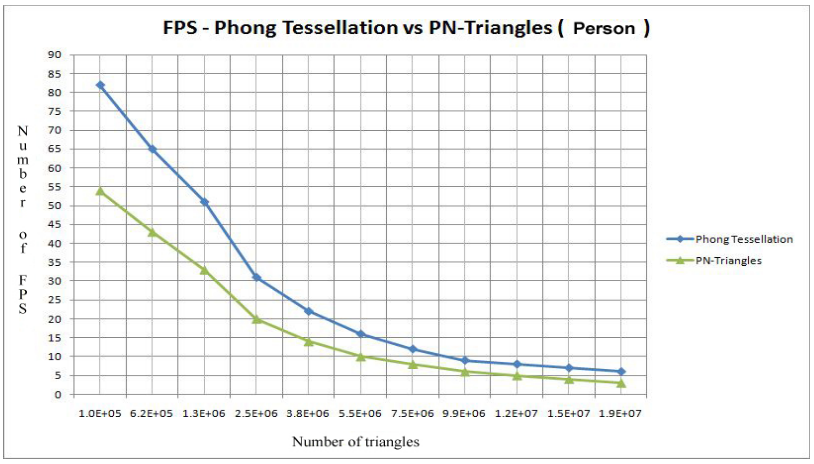


Figure 20: *Phong Tessellation vs PN-Triangles - Person mesh*

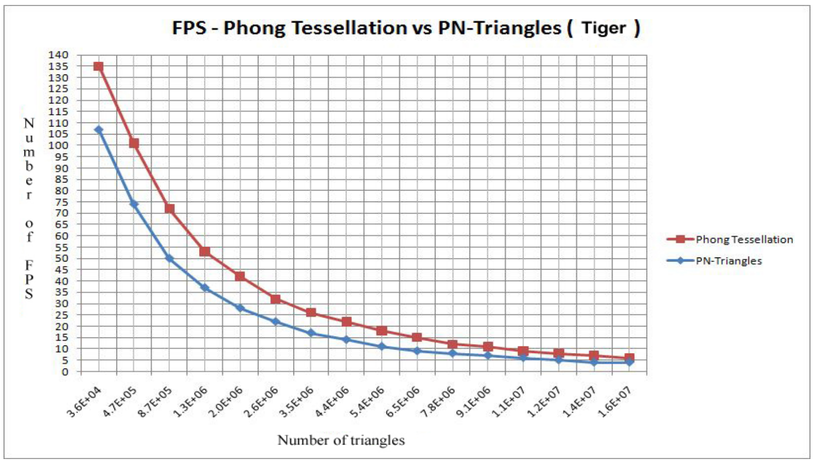
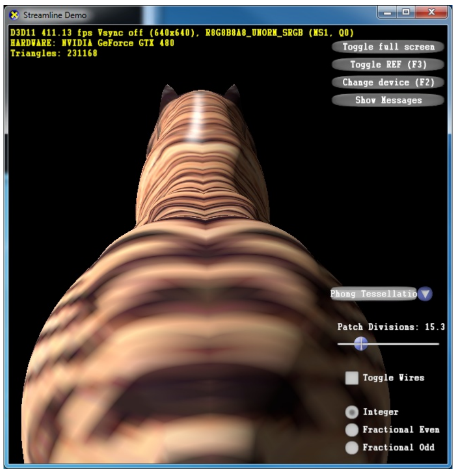


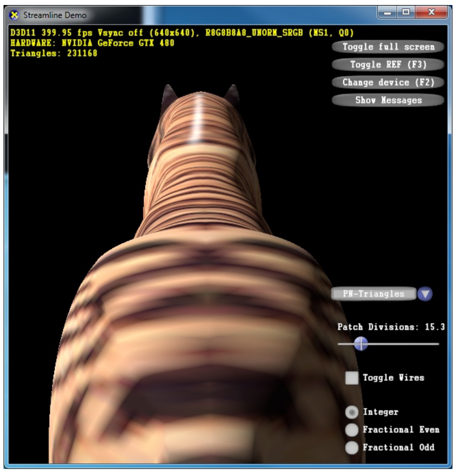
Figure 21: *Phong Tessellation vs PN-Triangles - Tiger mesh*



```
D3D11 411.13 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 231168
```

Phong Tessellation
Patch Divisions: 15.3

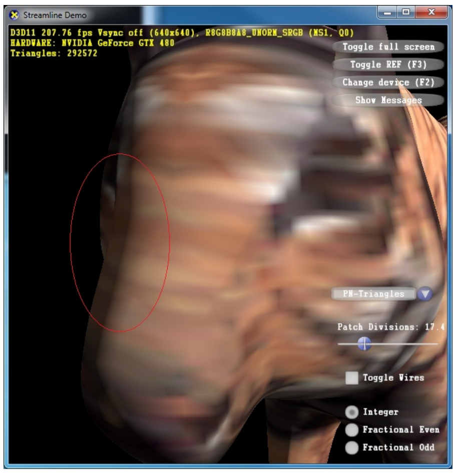
Figure 13: *Model using the Phong Tessellation algorithm*



```
D3D11 399.95 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 231168
```

PN-Triangles
Patch Divisions: 15.3

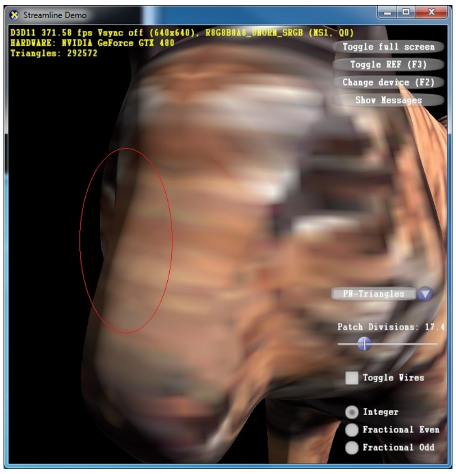
Figure 14: *Model using the PN-Triangles algorithm*



```
D3D11 207.76 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 292572
```

PN-Triangles
Patch Divisions: 17.4
Integer

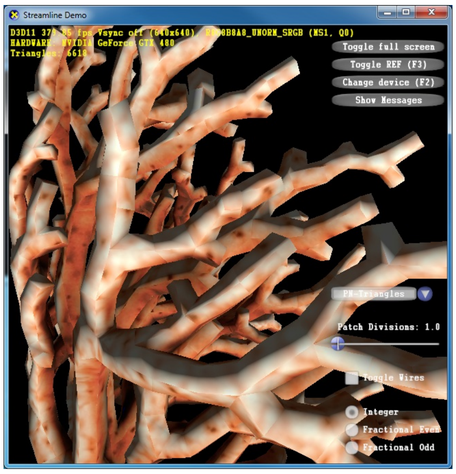
Figure 15: *PN-Triangles with linearly interpolated normals*



```
D3D11 371.58 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 292572
```

PN-Triangles
Patch Divisions: 17.4
Integer

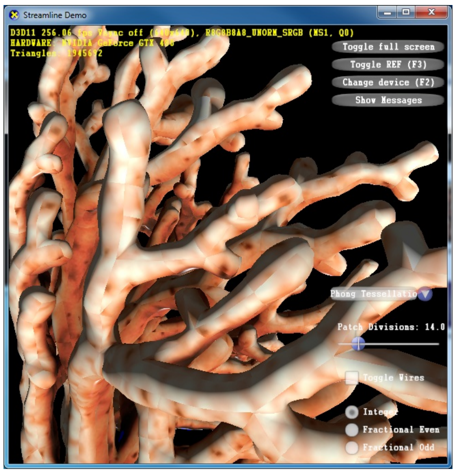
Figure 16: *PN-Triangles with quadratic normals interpolation*



```
D3D11 379.85 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 6618
```

PN-Triangles
Patch Divisions: 1.0
Integer

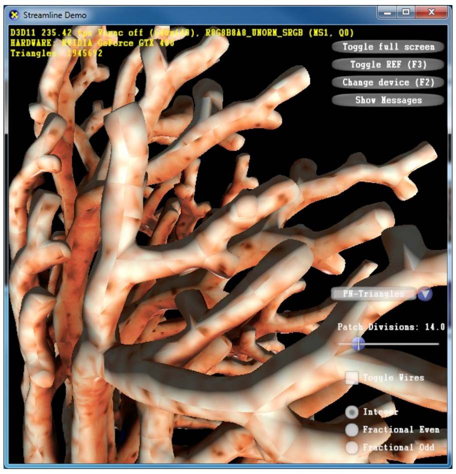
Figure 17: *Original model without any subdivision*



```
D3D11 256.06 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 1945692
```

Phong Tessellation
Patch Divisions: 14.0
Integer

Figure 18: *Model using the Phong Tessellation algorithm*



```
D3D11 235.42 fps Vsync off (640x640), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA GeForce GTX 480
Triangles: 1945692
```

PN-Triangles
Patch Divisions: 14.0
Integer

Figure 19: *Model using the PN-Triangles algorithm*

Listing 1: *PN-Triangles Hull Shader*

```
struct HS_Input
{
    float3 Position    : POSITION;
    float3 Normal      : NORMAL;
    float2 TexCoord    : TEXCOORD;
};

struct HS_ConstantOutput
{
    // Tessellation factors
    float fTessFactor[3]       : SV_TessFactor;
    float fInsideTessFactor  : SV_InsideTessFactor;

    // Geometry control points
    float3 B210      : POSITION3;
    float3 B120      : POSITION4;
    float3 B021      : POSITION5;
    float3 B012      : POSITION6;
    float3 B102      : POSITION7;
    float3 B201      : POSITION8;
    float3 B111      : CENTER;

    // Normal control points
    float3 N110      : NORMAL3;
    float3 N011      : NORMAL4;
    float3 N101      : NORMAL5;
};

HS_ConstantOutput HS_Constant(InputPatch<HS_Input,3>input)
{
    HS_ConstantOutput output = (HS_ConstantOutput)0;

    //Same Tessfactor
    output.fTessFactor[0] = output.fTessFactor[1] =
    output.fTessFactor[2] = output.fInsideTessFactor =
    g_cpuTessFactor;

   //Control points of positions and normals
   //of the corners are the same of the triangle
    float3 B003 = input[0].Position;
    float3 B030 = input[1].Position;
    float3 B300 = input[2].Position;
    float3 N002 = input[0].Normal;
    float3 N020 = input[1].Normal;
    float3 N200 = input[2].Normal;

    //Calculate the control points for the geometry
    output.B210 = ((2.0f*B003)+B030−
               (dot((B030−B003),N002)*N002))/3.0f;
    output.B120=((2.0f*B030)+B003−
               (dot((B003−B030),N020)*N020))/3.0f;
    output.B021=((2.0f*B030)+B300−
               (dot(B300−B030),N020)*N020))/3.0f;
    output.B012 = ((2.0f*B300)+B030−
               (dot((B030−B300),N200)*N200))/3.0f;
    output.B102 = ((2.0f*B300)+B003−
               (dot((B003−B300),N200)*N200))/3.0f;
    output.B201 = ((2.0f*B003)+B300−
               (dot((B300−B003),N002)*N002))/3.0f;
    // Central control point
    float3 E = (output.B210+output.B120+output.B021
               +output.B012+output.B102
               +output.B201)/6.0f;
    float3 V = (B003+B030+B300)/3.0f;
    output.B111 = E+((E−V)/2.0f);

    //Control points for the normal field
    float V12 = 2.0f*dot(B030−B003,N002+N020)
               / dot(B030−B003,B030−B003);
    output.N110 = normalize(N002+N020−V12*
               (B030−B003));
    float V23 = 2.0f*dot(B300−B030,N020+N200)
               /dot(B300−B030,B300−B030);
    output.N011 = normalize(N020+N200−V23*
               (B300−B030));
    float V31 = 2.0f*dot(B003−B300,N200+N002)
               / dot(B003−B300,B003−B300 );
    output.N101 = normalize(N200+N002−V31*
               (B003−B300));

    return output;
}
```

Listing 2: *PN-Triangles Domain Shader*

```
struct DS_Output
{
    float4 Position     : SV_Position;
    float2 TexCoord     : TEXCOORD0;
            float3 Normal                    : NORMAL0;
};

[domain("tri")]
DS_Output DS( HS_ConstantOutput HSC,
const OutputPatch<HS_Output, 3> input,
float3 UVW : SV_DomainLocation )
{
    DS_Output output = (DS_Output)0;

    //Evaluates the position based on control points
            //and on baricentric parameters
    float3 Position = input[0].Position *
            UVW.z*UVW.z*UVW.z+
    input[1].Position * UVW.x * UVW.x * UVW.x +
    input[2].Position * UVW.y * UVW.y * UVW.y +
    HSC.B210 * 3 * UVW.z * UVW.z * UVW.x +
    HSC.B120 * UVW.z * 3 * UVW.x * UVW.x +
    HSC.B201 * 3 * UVW.z * UVW.z * 3 * UVW.y +
    HSC.B021 * 3 * UVW.x * UVW.x * UVW.y +
    HSC.B102 * UVW.z * 3 * UVW.y * UVW.y +
    HSC.B012 * UVW.x * 3 * UVW.y * UVW.y +
    HSC.B111 * 6.0f * UVW.y * UVW.x * UVW.z;

    //Evaluate Normals
    float3 Normal =
            input[0].Normal * UVW.z * UVW.z +
    input[1].Normal * UVW.x * UVW.x +
    input[2].Normal * UVW.y * UVW.y +
    HSC.N110 * UVW.z * UVW.x +
    HSC.N011 * UVW.x * UVW.y +
    HSC.N101 * UVW.z * UVW.y;

    //Normalizes
    output.Normal = normalize( Normal );

    //Linearly interpolate texcoords
    output.TexCoord = input[0].TexCoord * UVW.z
            + input[1].TexCoord * UVW.x
            + input[2].TexCoord * UVW.y;

    //Send to screen space
    output.Position = mul( float4( Position.xyz, 1.0 ),
            g_ViewProjection );

    return output;
}
```

Listing 3: *Phong Tessellation Hull Shader*

```
HS_CONSTANT_DATA_OUTPUT BezierConstantHS
( InputPatch<VS_CONTROL_POINT_OUTPUT,
INPUT_PATCH_SIZE> ip,
uint PatchID : SV_PrimitiveID )
{
    HS_CONSTANT_DATA_OUTPUT Output;

  Output.Edges[0] = Output.Edges[1] =
  Output.Edges[2] =  Output.Inside =
  g_fTessellationFactor;

    return Output;
}
```

Listing 4: *Auxiliary function for orthogonal projection*

```
float3 projIntoPlane(float3 planeNormal,
float3 planePoint,
float3 pointToProject)
{
  float3 res;
  res = pointToProject −
  dot(pointToProject−planePoint,planeNormal)*planeNormal;

  return res;
}
```

Listing 5: *Phong Tessellation Domain Shader*

```
struct DS_OUTPUT
{
    float4 vPosition    : SV_POSITION;
    float3 vNormal      : NORMAL0;
    float2 vTexCoord    : TEXCOORD0;
};

[domain("tri")]
DS_OUTPUT Bezier( HS_CONSTANT_DATA_OUTPUT input,
                  float3 UV : SV_DomainLocation,
                  const OutputPatch<HS_OUTPUT,
                  OUTPUT_PATCH_SIZE> patch )
{
  DS_OUTPUT Output;

  //Linear interpolation of the triangle
  float3 p = UV.x*patch[0].vPosition +
             UV.y*patch[1].vPosition +
             UV.z*patch[2].vPosition;

  //Three orthogonal projection in the tangent plans
  float3 pProjU =
  projIntoPlane(patch[0].vNormal, patch[0].vPosition, p);
  float3 pProjV =
  projIntoPlane(patch[1].vNormal, patch[1].vPosition, p);
  float3 pProjW =
  projIntoPlane(patch[2].vNormal, patch[2].vPosition, p);

  //Another interpolation to find the final position
  float3 pNovo = UV.x*pProjU + UV.y*pProjV + UV.z*pProjW;

  //Screen Space output for the rasterizer
  Output.vPosition=mul(float4(pNovo,1), g_mViewProjection);
  //Linearly interpolared Normals
  Output.vNormal = normalize(UV.x*patch[0].vNormal +
                   UV.y*patch[1].vNormal +
                   UV.z*patch[2].vNormal);
  //Linearly interpolated texture coordinates
  Output.vTexCoord = UV.x*patch[0].vTexCoord +
  UV.y*patch[1].vTexCoord +
  UV.z*patch[2].vTexCoord;

  return Output;
}
```

References

2KGAMES, 2010. http://www.mafia2game.com/.

BOUBEKEUR, T., AND ALEXA, M. 2008. Phong tessellation. *ACM Trans. Graphics (Proc. SIGGRAPH Asia) 27*, 139–143.

BOUBEKEUR, T., AND SCHLICK, C. 2007. Qas: Real-time quadratic approximation of subdivision surfaces. *Computer Graphics and Applications, Pacific Conference on 1*, 453–456.

BOUBEKEUR, T., REUTER, P., AND SCHLICK, C. 2005. Scalar tagged pn triangles. In *EUROGRAPHICS 2005 (Short Papers)*, Eurographics.

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

DOO, D., AND SABIN, M. 1978. Behaviour of recursive division surfaces near extraordinary points. *Computer Aided Design 10*, 6, 356–360.

DOO, D. 1978. A subdivision algorithm for smoothing down irregularly shaped polyhedrons. In *Int'l Conf. Ineractive Techniques in Computer Aided Design*, IEEE Computer Soc., Bologna, Italy, 157–165.

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

FATAHALIAN, K. 2010. Evolving the Direct3D Pipeline for Real-time Micropolygon Rendering. SIGGRAPH 2010 Course. In *ACM SIGGRAPH 2010.*

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

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

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

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.

LOOP, C. 1987. Smooth subdivision surfaces based on triangles. In *PhD Thesis - University of Utah.*

PETERS, J. 2008. PN-Quads. Technical Report 2008-421. In *Dept CISE, University of Florida.*

VALDETARO, A., NUNES, G., RAPOSO, A., FEIJO, B., AND DE TOLEDO, R. 2010. Understanding shader model 5.0 with directx11. *IX Brazilian symposium on computer games and digital entertainment.*

VLACHOS, A., PETERS, J., BOYD, C., AND MITCHELL, J. L. 2001. Curved pn triangles. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, ACM Press, New York, NY, USA, 159–166.

YEO, Y. I., NI, T., MYLES, A., GOEL, V., , AND PETERS, J. 2009. Parallel smoothing of quad meshes. *The Visual Computer 25*, 8 (Aug), 757–769.

ZORIN, D., SCHRODER, P., AND SWELDENS, W. 1996. Interpolating subdivision for meshes with arbitrary topology. In *SIGGRAPH96*, 189–192.