

Smooth GPU Tessellation

CS284 Final Project

Brandon Wang

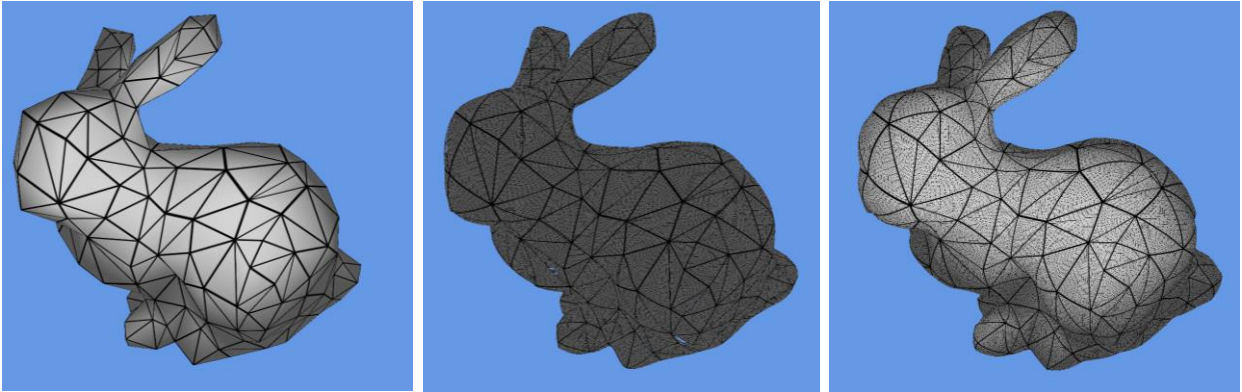


Figure 1: An input coarse mesh (left) is subjected to tessellation using Curved PN Triangles (middle) and Phong Tessellation (right).

Abstract

Models used in real-time graphics are commonly rendered using triangular boundary representations. Triangulation approximates flat surfaces very well, but requires many fine triangles to accurately represent curved surfaces. This project implements two methods for smoothing coarse input meshes using GPU tessellation shaders to obtain smooth surfaces.

1 Introduction

Triangular boundary representations are common for many 3D models. Because triangles are planar, they represent flat surfaces with hard edges well. When attempting to represent a curved surface, such as a sphere, many triangles are needed, when compared to a flat surface, such as a parallelogram. Because of the additional space required, it is common to represent curved surfaces with jagged edges, as seen in the left picture of Figure 1. This project implements two methods to remedy this situation, in the context of real-time applications.

2 Background

Real-time applications commonly use OpenGL or Direct3D to interface with a graphics processing unit, or GPU, a specialized electronic circuit designed to rapidly render images for display. Modern commodity GPUs have a rendering pipeline that involves multiple programmable stages for per-vertex operations and per-fragment operations. This project utilizes the programmable stage built around the GPU's hardware tessellator.

The number of triangles affects the speed of rendering in modern real-time GPU-based approaches in two ways. First, the GPU must rasterize every triangle. Second, the CPU must pass the triangles into graphics memory. Because these constraints are kept in mind, many models are coarsely triangulated, to reduce their rendering cost. However, when an object is large on a screen, the triangulation's approximation artifacts are very visible, shown as jagged edges on what should be a smooth surface, shown in the left picture in Figure 1.

Subdivision surfaces can represent curved surfaces by utilizing a coarse control point mesh. These subdivision surfaces are commonly defined recursively [Catmull and Clark 1978]. To be implemented in a GPU tessellator, a limit-surface representation [Stam 1998] is common, as new points can be generated in a single pass. However, a limit-surface method involves pre-computation on the mesh before tessellation, has issues at extraordinary vertices, and can be quite slow when implemented on a GPU [Loop and Schaefer 2008]. Furthermore, meshes used with subdivision schemes are typically modeled with the scheme in mind, and may generate vastly inconsistent meshes with differing subdivision methods [Stam and Loop 2003].

Representations other than boundary representations can exhibit curved surfaces. However, for real-time applications, these representations are often triangulated to interface with the existing GPU rasterization pipeline that typically accepts only planar geometry. Alternative real-time approaches exist, such as real-time ray-tracing [Parker et al. 2010] that may handle more representations of geometry. This project, however, will focus on using the GPU rasterization pipeline.

2 Tessellation

Tessellation, in its most basic form, is breaking down polygons into finer pieces. A square can be tessellated into two triangles by splitting the shape along its diagonal. Although tessellation can be done on the CPU, doing so would increase the number of polygons that must be transmitted to the GPU. This may impact performance when using a large number of polygons, as the memory bandwidth between the CPU and GPU is finite. Additionally, GPUs are typically designed with parallelism in mind. Because tessellation is independent per-polygon, doing many tessellations in parallel will speed up the computation. The actual speed-up can be seen in the results section.

The current rasterization pipeline used in commodity GPUs is shown in Figure 2. The input to the pipeline are polygons, and typically specified by a CPU program. The polygons then undergo per-vertex operations in a vertex program, such as multiplication with transformation matrices. These vertices are then optionally tessellated, to yield additional vertices. The vertices are then assembled into primitives, and rasterized. Finally, a fragment program determines the color at each pixel, and an image then typically shown on a display.

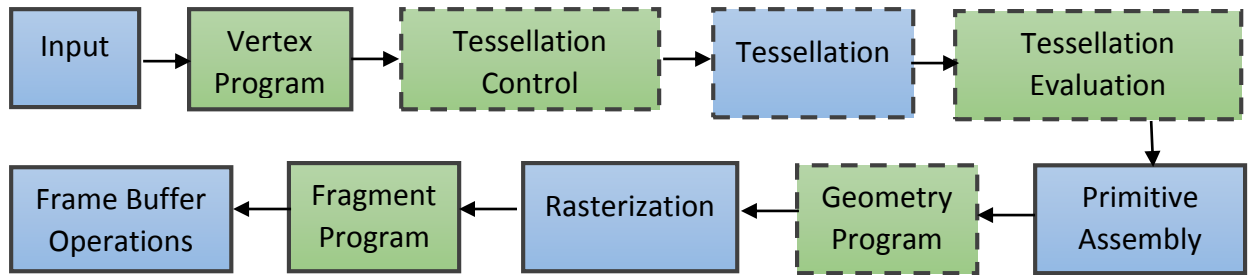


Figure 2: A modern GPU rasterization pipeline. Steps shown in blue are fixed-function. Steps shown in green are programmable. Steps with a dotted border are optional.

The Tessellation Control Program (TCP) controls the number of additional polygons to generate, per input polygon. The tessellation step of actually generating the additional vertices is not programmable. The Tessellation Evaluation Program (TEP) controls the location of additional points within an input polygon.

The TCP operates on a per-vertex level. We will revisit this program in adaptive tessellation. For uniform tessellation, this can simply set a user-defined variable for the number of new triangles to generate.

The tessellator will generate additional vertices within a given polygon. These additional vertices are then passed to the TEP, along with barycentric coordinates of the generated vertex relative to the input polygon. We smooth our input geometry by moving the generated vertices in the TEP.

3 Smooth Tessellation

By manipulating the location of an additional vertex in the TEP, we can smooth the model by more closely approximating a smooth surface. This project implement two existing methods, and I discuss a third possible option. For the two methods implemented, placement of additional vertices depend on the input polygon vertices $V_1, V_2, V_3 \in \mathbb{R}^3$, their associated normal vectors $N_1, N_2, N_3 \in \mathbb{R}^3$, and barycentric coordinates of the newly generated vertex $u, v, w \in \mathbb{R}, u + v + w = 1$.

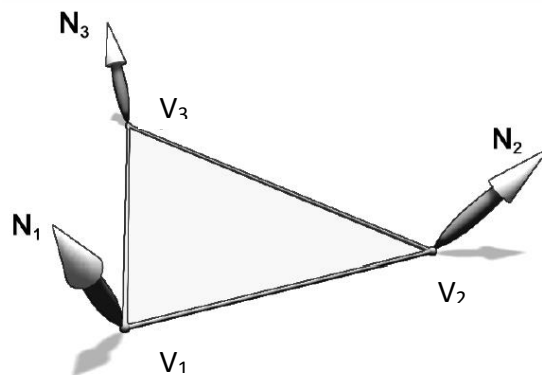


Figure 3: The inputs to a smooth Tessellation Evaluation Program. Image from Vlachos et al. 2001

3.1 Curved PN Triangles

From the input vertices and normals, Curved PN Triangles [Vlachos et al. 2001] generates control points to represent a smooth, cubic Bezier triangle that interpolates across the three input vertices.

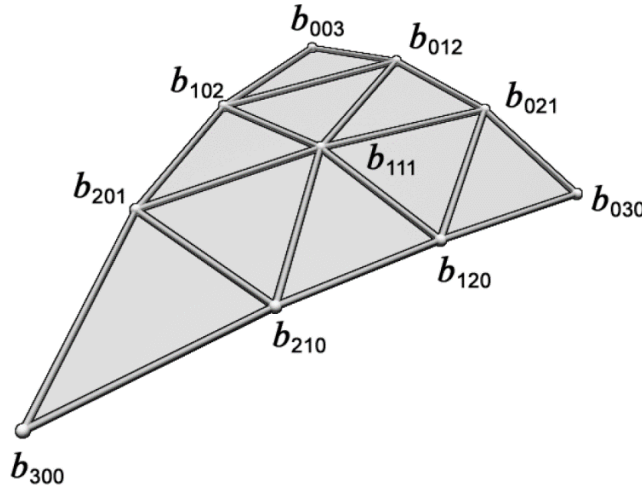


Figure 4: Generated control points for a cubic Bezier triangle patch. Image from Vlachos et al. 2001.

The cubic Bezier triangle can be evaluated given these control points with the following formula [Vlachos et al. 2001]

$$p^*(u, v, w) = b_{300}w^3 + b_{030}u^3 + b_{003}v^3 + 3b_{210}w^2u + 3b_{120}wu^2 + 3b_{201}w^2v + 3b_{021}u^2v + 3b_{102}wv^2 + 3b_{012}uv^2 + 6b_{111}wuv$$

The control points are generated by interpolating geometric vertex locations, and projecting them along their nearest normal, and in the case of the center control point, the averaged normal. The formulae for generating these control points are given in the appendix.

3.2 Phong Tessellation

Phong Tessellation draws inspiration from Phong Shading, a method that shades boundary representations smoothly by interpolating normals used for lighting calculations. Phong Tessellation [Boubekeur and Alexa 2008] generates tangent planes for each input vertex, aligned to each input normal, and performs barycentric interpolation within the three defining tangent planes of a triangle to find a newly generated point.

This process is illustrated in Figure 5. A newly generated vertex p is projected onto the three tangent planes, shown by the dotted green lines. The new vertex location p^* is found by barycentric interpolation of the tangent plane projections.

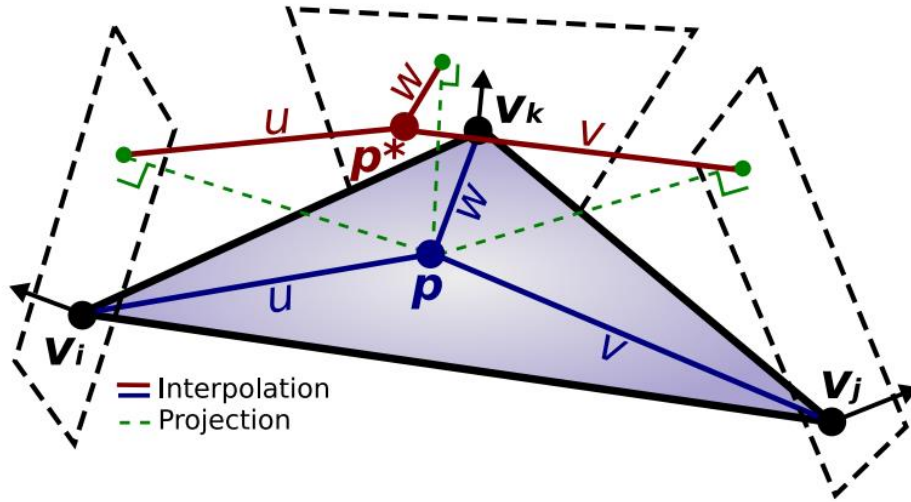


Figure 5: Phong Tessellation principle. The interpolation of vertex tangent planes defines a curve geometry for each triangle. Image from Boubekeur and Alexa 2008.

We can define the projection operation of a point q onto the tangent plane associated with vertex v_i and normal n_i , following Boubekeur and Alexa 2008.

$$\pi_i(q) = q - ((q - v_i) \cdot n_i)n_i$$

Given this projection, we can find the new location p^* of an input vertex p by a simple matrix multiplication.

$$p^*(u, v, w) = (u, v, w) \begin{pmatrix} \pi_1(p(u, v, w)) \\ \pi_2(p(u, v, w)) \\ \pi_3(p(u, v, w)) \end{pmatrix}$$

3.3 Subdivision Surfaces

Although this project does not implement subdivision surfaces, it is possible to evaluate subdivision surfaces in a tessellation program. Because each stage in the tessellation pipeline is evaluated only once, we cannot utilize a recursive subdivision strategy without doing multiple render passes, which will involve multiple evaluations of the other portions of the rasterization pipeline. A limit surface evaluation of subdivision surfaces can be used, such as Stam's method [Stam 1998]. However, computation of Stam's method on extraordinary vertices will exhibit poor performance, and much research is done to approximate subdivision surfaces at extraordinary vertices [Loop and Schaefer 2008]. Subdivision surfaces also require adjacency information of the input mesh [Catmull and Clark 1978], which may not be readily available in some 3D meshes, and also require a pre-computation pass if implemented in a tessellation program.

4 Adaptive Tessellation

Phong Shading will shade shapes smoothly in lighting calculations. This smoothness is interrupted at edges of the geometry, due to coarse triangulation. Phong Tessellation and Curved PN Triangles address the issue at silhouette edges by generating a smooth tessellation over the surface of each triangle. However, the tessellation levels needed to generate a smooth curve do not need to be uniform throughout the mesh. The approach this project takes to adaptively tessellate geometry considers both the distance from the camera and the normal of the polygon.

The tessellation levels can be specified by heuristic metrics in the TCP, which control the number of vertices to generate by tessellation.

Intuitively, we can see errors due to triangulation the closer an object is to the camera. Therefore, we only need to tessellate the triangles which are closer to the camera. A further extension on this metric would be to consider the screen-space projected triangle size.

This project also utilizes Boubekur et al.'s contour metric to find a function to find polygons facing almost parallel to the camera.

Combining these two metrics, we can obtain the following measure to control tessellation

$$d_i = \left(1 - \frac{|n_i^T(c - v_i)|}{|c - v_i|} \right) \frac{m}{z}$$

With c as the camera position, v_i and n_i as the vertices and normals corresponding to the input polygon, m as the user-defined maximum tessellation value, and z as a measure of zoom.

Adaptively tessellating very coarse meshes can result in visible tears in the model. Although a continuous field is generated at every edge, the number of samples from the two bordering triangles can generate a hole in the mesh. The tessellation level difference between two neighboring triangles must be minimized to reduce the holes generated in the mesh.

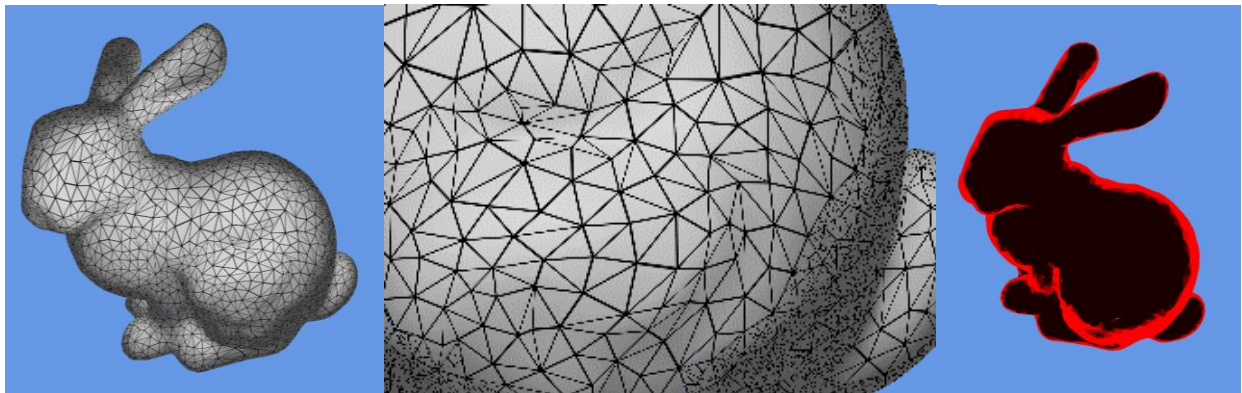


Figure 6: Left: Adaptive tessellation of the Stanford bunny. Middle: Close-up view. Right: Visualization of areas with high tessellation

5 Results

Example outputs of Curved PN Triangles and Phong Tessellation can be seen in Figure 1 and Figure 6. We can measure the speed of doing these computations using a tessellation shader, versus doing the computations on the CPU. It is clear that doing these computations on the GPU maintains a higher frame rate, which is important for real-time applications.

Adaptive tessellation is not included on the table, as no noticeable frame rate difference occurs until about 9000x triangle tessellation, due to the smaller number of triangles that must be tessellated.

	CPU Averaging	Averaging	Curved PN Triangles	Phong Tessellation
6x	60	60	60	60
54x	40	60	60	60
96x	22	43	35	39
150x	14	37	32	35
216x	10	17	14	16
294x	7	19	16	18
384x	6	14	12	13
486x	5	12	10	11
600x	4	8	7	8
726x	3	8	7	8

Table 1: Frames-per-second values (capped at 60) of various tessellation values on different tessellation methods. Horizontally, the table shows differing methods. Vertically, the table shows a multiplier on number of triangles generated. Measured on a 10,000 triangle Stanford Dragon model, on a computer running a NVIDIA GeForce GTX 680 GPU.

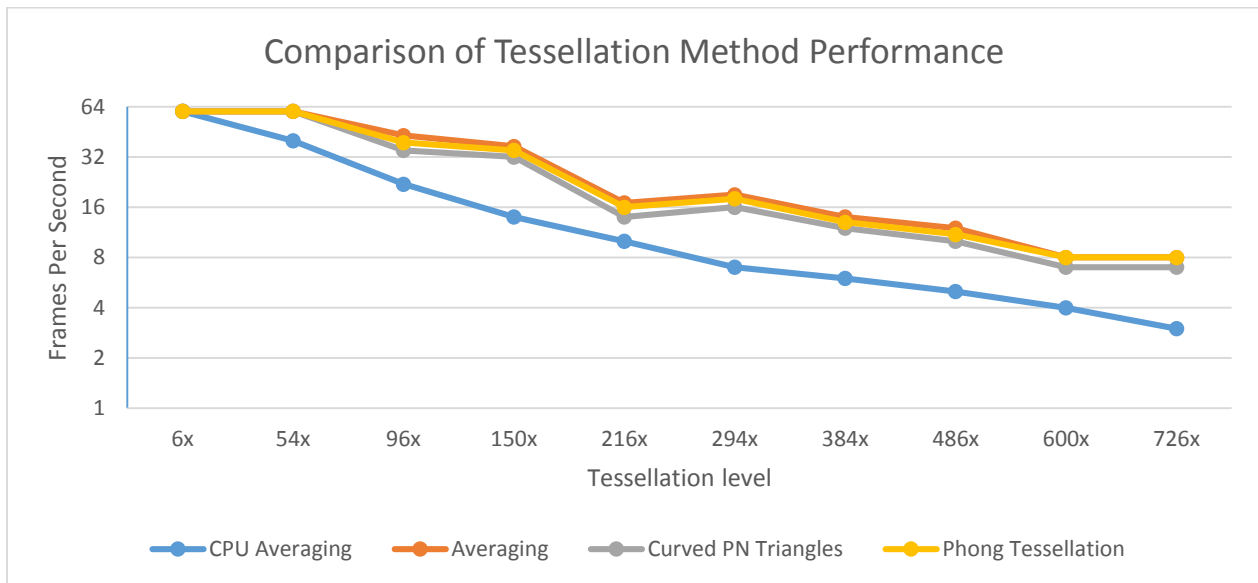


Figure 7: Chart of Table 1. Vertical FPS axis is logarithmic.

6 Conclusion

This project presents results and comparisons between various smooth tessellation methods. Although these curved surfaces do not have as strong a guarantee of continuity as subdivision surfaces [Boubekeur and Alexa 2008, Vlachos et al. 2001], they are very fast to compute. I believe the project was a success, as the speed of these methods are well-suited for real-time applications, as can be seen in Table 1.

7 References

- Boubekeur, T. and Alexa, M. 2008. Phong Tessellation. *Proceedings of SIGGRAPH Asia 2008*.
- Catmull, E. and Clark, J. 1978. Recursively Generated B-Spline Surfaces on Topological Meshes. *Computer-Aided Design Vol. 10*.
- Loop, C. and Schaefer S. 2008. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Transaction on Graphics, Vol. 27*.
- Parker, S., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robinson, A., and Stitch, M. 2010. Optix: A General Purpose Ray Tracing Engine. *Proceedings of SIGGRAPH 2010*.
- Stam, J. 1998. Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. *Proceedings of SIGGRAPH 1998*.
- Stam, J. and Loop, C. 2003. Quad/Triangle Subdivision. *Computer Graphics Forum Vol. 22*.
- Vlachos A., Peters J., Boyd C., and Mitchell J. 2001. Curved PN Triangles. *Symposium on Interactive 3D Graphics 2001*.

8 Appendix

8.1 Curved PN Triangle Weights

$$w_{ij} = (P_j - P_i) \cdot N_i \in R$$

$$b_{300} = P_1$$

$$b_{030} = P_2$$

$$b_{003} = P_3$$

$$b_{210} = \frac{1}{3}(2P_1 + P_2 - w_{12}N_1)$$

$$b_{120} = \frac{1}{3}(2P_2 + P_1 - w_{21}N_2)$$

$$b_{021} = \frac{1}{3}(2P_2 + P_3 - w_{23}N_2)$$

$$b_{012} = \frac{1}{3}(2P_3 + P_2 - w_{32}N_3)$$

$$b_{102} = \frac{1}{3}(2P_3 + P_1 - w_{31}N_3)$$

$$b_{201} = \frac{1}{3}(2P_1 + P_3 - w_{13}N_1)$$

$$E = \frac{1}{6}(b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201})$$

$$V = \frac{1}{3}(P_1 + P_2 + P_3)$$

$$b_{111} = E + (E - V)/2$$

