

Smooth Surfaces for Video Game Development

by

Ashish Amresh

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2011 by the
Graduate Supervisory Committee:

Anshuman Razdan, Co-Chair

Gerald Farin, Co-Chair

Peter Wonka

Dianne Hansford

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

The video game graphics pipeline has traditionally rendered the scene using a polygonal approach. Advances in modern graphics hardware now allow the rendering of parametric methods. This thesis explores various smooth surface rendering methods that can be integrated into the video game graphics engine. Moving over to parametric or smooth surfaces from the polygonal domain has its share of issues and there is an inherent need to address various rendering bottlenecks that could hamper such a move. The game engine needs to choose an appropriate method based on in-game characteristics of the objects; character and animated objects need more sophisticated methods whereas static objects could use simpler techniques. Scaling the polygon count over various hardware platforms becomes an important factor. Much control is needed over the tessellation levels, either imposed by the hardware limitations or by the application, to be able to adaptively render the mesh without significant loss in performance. This thesis explores several methods that would help game engine developers in making correct design choices by optimally balancing the trade-offs while rendering the scene using smooth surfaces. It proposes a novel technique for adaptive tessellation of triangular meshes that vastly improves speed and tessellation count. It develops an approximate method for rendering Loop subdivision surfaces on tessellation enabled hardware. A taxonomy and evaluation of the methods is provided and a unified rendering system that provides automatic level of detail by switching between the methods is proposed.

DEDICATION

To my parents Latha and Amresh, and to my loving wife kiran

ACKNOWLEDGEMENTS

During these long years as a PhD student at Arizona State University, I have had the opportunity of working with many admirable fellow researchers, students and faculty, who have been instrumental in building my research. I am ever so grateful to all your support and guidance.

Foremost, I would like to thank my advisors Dr. Gerald Farin and Dr. Anshuman Razdan for continued guidance, support and extended patience with my research. I would like to thank my committee members Dr. Peter Wonka and Dr. Dianne Hansford for their timely advise, review and feedback at various stages of my research. I would like to thank all my fellow researchers at Arizona State University labs, PRISM, Decision Theater, Gaming and I3DEA, and specifically Ryan Anderson, Christoph Funzig, Kerstin Mueller and John Femiani for directly impacting my research and help shape it into ways I never thought was possible. Mark Buchignani at THQ Game Studios, Phoenix for helping out with the visual fidelity tests and providing constant feedback.

I would like to thank my family members and friends who have always been there for me over these long years and have supported me all the way. Above all my parents for having the faith in me and letting me pursue my dreams and my wife Kiran for being there from day one and never getting tired of this never-ending journey.

TABLE OF CONTENTS

	Page
TABLE OF CONTENTS	iv
LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER	1
1 INTRODUCTION	1
1.1 Smooth Surfaces for Video Game development	1
1.2 Intellectual Merit	2
1.3 Contributions of This Work	3
1.4 List of Publications	3
2 RELATED WORK	5
2.1 Related Work in Polygon Tessellation	5
2.2 Related Work in Rendering Smooth Surfaces	6
3 GRAPHICS PIPELINE IN GAMES	9
3.1 Introduction	9
3.2 Surfaces in Games	9
3.2.1 Polygonal Rendering Pipeline	10
3.2.2 Tessellation by Instancing	11
3.2.3 Hardware Tessellation	12
3.2.4 Asset Production Pipeline	13
3.3 Background for Rendering Smooth Surfaces	15
3.3.1 Notation	15
3.3.2 Continuity	16
3.3.3 Reflection Lines	17
3.3.4 Continuous Tangent Frames	18
3.4 Locally Defined Parametric Surfaces	18
3.4.1 Triangular Bézier Patches	18

Chapter		Page
3.4.2	G^1 Continuity of Quartic Bézier Triangles	20
3.4.3	PN-Triangles	21
3.4.4	Triangular Gregory Patches	22
3.4.5	Bi-Cubic Bézier Patches	23
3.5	Subdivision Surfaces	24
3.5.1	Catmull-Clark Subdivision Surface	24
3.5.2	Approximating Catmull-Clark Surfaces	25
3.5.3	The Loop Subdivision Surface	30
3.5.4	Approximating Loop Subdivision Surfaces	31
4	ADAPTIVE TESSELLATION	39
4.1	Introduction	39
4.2	Semi-Uniform, 2-Different Tessellation	39
4.2.1	Edge Tessellation Factors based on Vertex/Edge Criteria	39
4.2.2	Edge Tessellation Factors for the 2-Different Case	40
4.2.3	Adaptive Tessellation Pattern with 2-Different Factors	41
4.3	Results	42
5	UNIFIED RENDERING FRAMEWORK	48
5.1	Introduction	48
5.2	Unified Rendering Framework	49
5.2.1	Control Shader	51
5.2.2	Evaluation Shader	51
5.3	Results	52
5.3.1	Visual Fidelity Test	52
5.3.2	Performance Results	53
6	CONCLUSION AND FUTURE DIRECTIONS	57
6.1	Summary	57
6.2	Conclusions and Future Work	57

Chapter	Page
REFERENCES	59
APPENDIX	63
A SD-2 PSEUDO CODE	63
B CONTROL, EVALUATION AND FRAGMENT SHADER IMPLEMENTA- TIONS	65
BIOGRAPHICAL SKETCH	70

LIST OF TABLES

Table	Page
4.1 Performance for silhouette refinement with max tessellation factor 7 and triangle strips	44
4.2 Performance for silhouette refinement with max tessellation factor 7 and not using triangle strips	44
5.1 Shader properties for the three methods	52
5.2 Frame rate comparison for the three methods	53
5.3 Geometric error reported by Metro	53

LIST OF FIGURES

Figure	Page
1.1 Contribution Area - 1	1
1.2 Contribution Area - 2	1
1.3 Contribution - 1	2
1.4 Contribution - 2	3
3.1 Polygon rendering pipeline	10
3.2 Tessellation using instancing	12
3.3 New graphics pipeline with the tessellator unit	13
3.4 Asset production	14
3.5 Cross platform rendering using CGFX	14
3.6 Difference between G^1 and C^1 continuity	16
3.7 Smooth normals and reflection lines	17
3.8 Cubic Bézier triangle	19
3.9 de Casteljau algorithm	19
3.10 G^1 continuity for quartic Bézier	20
3.11 PN-points	21
3.12 PN-normals	21
3.13 Triangular Gregory control polygon	22
3.14 Control points of a bi-cubic Bézier patch.	24
3.15 Catmull-Clark masks	25
3.16 One-ring neighborhood of a Catmull-Clark vertex	26
3.17 Catmull-Clark to bi-cubic Bézier masks	26
3.18 Quad Gregory control points	27
3.19 Tangent fields along the boundary edge	28
3.20 Approximate Catmull-Clark (ACC) subdivision	31
3.21 ACC limitations	32
3.22 Loop subdivision masks	33

Figure	Page
3.23 Loop limit tangents	35
3.24 Loop cross tangents	35
3.25 Rendering comparison	36
3.26 Rendering reflection lines and normals	37
3.27 Tetrahedron with long skinny triangles	38
4.1 SD-2 tessellation pattern	43
4.2 Results of SD-2 refinement for improving the silhouette on the Sphere and Bunny models	45
4.3 Comparison of SD-2 and stitching methods using reflection lines.	46
4.4 SD-2 with continuous LOD	47
5.1 Unified control points	49
5.2 Half edge mesh structure	50
5.3 Unified rendering stages	50
5.4 Hardware tessellation of PN-triangles	52
5.5 Visual fidelity results - 1	54
5.6 Visual fidelity results - 2	54
5.7 Rendering results for Monster Frog	56
5.8 Approximation error	56
A.1 SD-2 pseudo code	64
B.1 Reflection line shader	66
B.2 Gregory control shader	67
B.3 Gregory evaluation shader	68
B.4 Gregory evaluation shader-continued	69

INTRODUCTION



Figure 1.1: [Approximate Catmull-Clark patches rendered in Valve Software’s game: Team Fortress 2] The black lines indicate patch boundaries and the green lines provide crease support

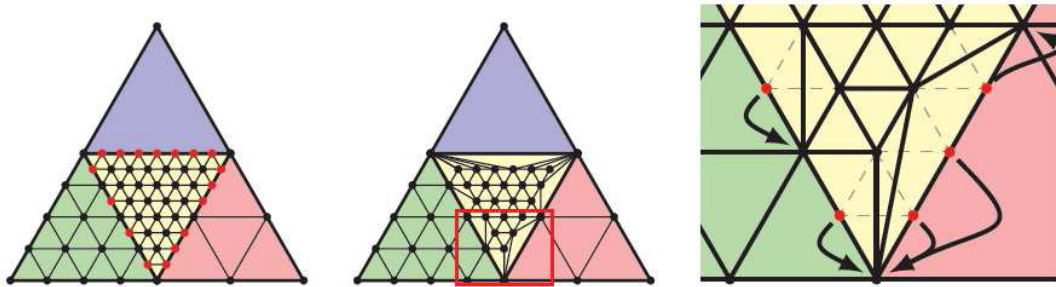


Figure 1.2: [A semi-uniform adaptive tessellation method using vertex snapping] Figure courtesy of (Dyken et al., 2009)

1.1 Smooth Surfaces for Video Game development

In this work, we present research that specifically addresses the following questions:

1. *What are the various methods for rendering smooth surfaces using current generation GPUs?*
2. *How can the rendering performance in terms of speed and quality be vastly improved?*
3. *What strategies can an application developer employ for optimizing the graphics pipeline?*

4. *How does an application developer change the asset production pipeline to accommodate for a shift from polygonal to parametric rendering?*

1.2 Intellectual Merit

This research builds upon various techniques, algorithms and methods and proposes a unified rendering system that would help application developers improve their rendering pipeline. Specifically it provides a comprehensive strategy for rendering smooth parametric surfaces using modern day GPUs and provides strategies for moving from polygonal to parametric rendering. It achieves the above by:

- *Identifying various game assets for proper application of the rendering method.*
- *Providing implementation strategies for various parametric approaches.*
- *Applying these strategies into an adaptive tessellation framework.*

It also develops two novel techniques for improving the rendering of triangular parametric surfaces. In the first technique, an adaptive tessellation algorithm that vastly improves the performance and visual quality of current methods is presented and in the second, an approximate rendering method for Loop subdivision surfaces is developed.

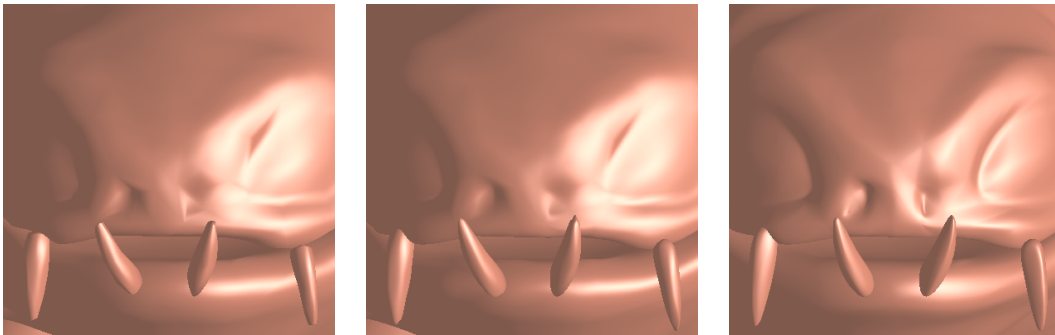


Figure 1.3: [Contribution 1:] A unified rendering framework for approximate Loop subdivision

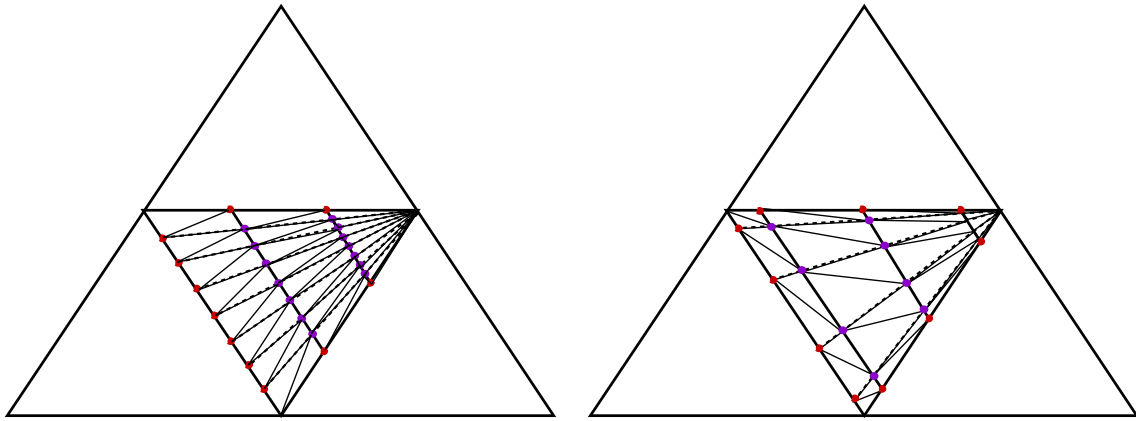


Figure 1.4: [Contribution 2:] A non-uniform adaptive tessellation algorithm

1.3 Contributions of This Work

This thesis presents the following original contributions by the author

- A unified rendering framework has been developed which enables simultaneous rendering of various smooth rendering algorithms in a single pass while using a minimal set of input points. This framework would be important for application developers interested in moving their architecture from polygonal to parametric or smooth surface rendering. See Figure 1.3.
- The ACC method (Loop et al., 2009) has been adapted and a variant for approximate rendering of Loop subdivision surfaces has been proposed.
- A new tessellation algorithm for triangular meshes is developed that takes advantage of the observation that all three edges of a triangle never get different tessellation factors. The algorithm is developed in such a manner that it optimizes the triangles into strips and achieves faster performance while tessellating into greater number of triangles. See Figure 1.4.

1.4 List of Publications

The following publications and submissions have been accomplished in last two years under the scope of this research plan.

- **Ashish Amresh**, Christoph Fünfzig, *Semi-uniform, 2-different tessellation of triangular parametric surfaces*, Proceedings of the 6th international conference on Advances in visual computing-Volume Part I, 2010
- **Ashish Amresh**, John Femiani, Christoph Fünfzig, *A Taxonomy and Evaluation of Methods for Approximating Loop Subdivision Using Tessellation Enabled GPUs*, submitted to The Journal of Graphics and Game Tools.

Chapter 2

RELATED WORK

2.1 Related Work in Polygon Tessellation

Rendering of parametric surfaces has a long history (Sfarti et al., 2006). Early approaches consist of direct scanline rasterization of the parametric surface (Schweitzer and Cobb, 1982). Modern GPUs have the ability to transform, light, rasterize surfaces composed of primitives like triangles and more recently also to tessellate them. Tessellation in particular can save bus bandwidth for rendering complex smooth surfaces on the GPU. The proposed triangle tessellation approaches can be distinguished either by the hardware stage they employ or by the geometric tessellation pattern. Approaches originating from triangular subdivision surfaces use a 1-to-4 split of the triangle, as seen in (Bóo et al., 2001) and (Settgast et al., 2004). For greater flexibility of the refinement, factors are assigned to the edges of each triangle, which results in three different tessellation factors for an arbitrary triangle.

Tessellation patterns for this general case go back to several authors after (Moreton, 2001), for example, in (Chung and Kim, 2003) optimized for a hardware implementation, and in (Schwarz and Stamminger, 2009) for a fast CUDA implementation.

Non-uniform, fractional tessellation (Munkberg et al., 2008) adds reverse projection to the evaluation of the tessellation pattern to make it non-uniform in parameter-space but more uniform in screen-space. In a recent work, (Dyken et al., 2009) keep a uniform tessellation pattern and snap missing vertices on the border curve to one of the existing vertices at the smaller tessellation factor. This is a systematic way to implement gap-filling but it is restricted to power-of-two tessellation factors and 1-to-4 splits of the parameter domain. In recent years hardware tessellation has become a reality and significant work has been done to enable direct rendering of parametric surfaces by providing a hardware tessellator unit (Gee, 2008).

Graphics API's DirecX11 and OpenGL 4.0 provide support for this tessellator unit by introducing three new stages in the rendering pipeline, see Section 3.2.3 in Chapter 3 for more details. In Chapter 4, we propose a novel tessellation algorithm that is non-uniform and is applied when the tessellation factors of a triangle are at most different by two. We show that in most cases this is true and our method improves the performance in terms of speed as well as number of triangles generated.

2.2 Related Work in Rendering Smooth Surfaces

The basic idea behind subdivision can be traced as far back as to the late 40s and early 50s when G. de Rham used corner cutting to describe smooth curves. In recent times the application of subdivision surfaces has grown in the field of computer graphics and computer aided geometric design (CAGD) mainly because it easily addresses the issues raised by multiresolution techniques and challenges raised for modeling complex geometry. The subdivision schemes introduced by Catmull and Clark (Catmull and Clark, 1978a) and Doo and Sabin (Doo and Sabin, 1978) set the tone for other schemes to follow and schemes like Loop (Loop, 1987), Butterfly (Dyn et al., 1990) and Modified Butterfly (Zorin et al., 1996), Kobbelt (Kobbelt, 1996) have become popular. These schemes are classified as either approximating, where the original vertices are not retained at newer levels of subdivision, or interpolating, where subdivision makes sure that the original vertices are carried over to the next level of subdivision. The Doo-Sabin, Cattmull-Clark and Loop schemes are approximating and Butterfly, Modified Butterfly and Kobbelt schemes are interpolating.

Real-time rendering of subdivision surfaces goes back to (Bolz and Schröder, 2002) where the next-step subdivision patterns are precomputed and stored inside a texture and (Bolz and Schröder, 2004) where it is performed as a linear combination of basis functions in the fragment shader. (Shiue et al., 2005) further refines it and preserves the patterns locally per mesh to reduce memory fetch time. Hardware implementation of subdivision surface was first proposed in (Pulli and Segal, 1996)

for Loop subdivision where they organized the triangles into a set of patches. In (Bischoff et al., 2000) hardware acceleration is performed by procedurally subdividing the meshes using a breadth-first or a depth-first approach. Direct evaluation can be performed on subdivision surfaces by implementing Stam's methods (Stam, 1998a,b), however this is not suitable for modeling sharp features and can also be computationally intensive.

The idea of rendering parametric triangular surfaces in realtime dates back to the introduction of PN-triangles (Vlachos et al., 2001), developed to improve the surface quality of low polygonal meshes. The method renders the surface by passing the control points and normals for the input mesh and evaluating a cubic triangular Bézier patch and a quadratic normal patch. Quadratic approximation surface (QAS) (Boubekeur and Schlick, 2007a,b) used the idea of rendering a PN-patch for limit Loop points and normals. In PNG1-triangles (Fünfzig et al., 2008) this approach was further improved to satisfy G^1 continuity across each edge. Similarly (Walton and Meek, 1996) provides a method for G^1 continuous surface when the boundary curves are known. (Loop and Schaefer, 2008) first introduced the idea of approximating Catmull-Clark subdivision surfaces using bi-cubic Bézier patches and then further refined it in (Loop et al., 2009) for hardware tessellation using Gregory Patches. In both these methods the resulting surface is C^2 continuous for regular meshes and G^1 continuous for irregular. (Kovacs et al., 2009) adds creases and corners to this method. These methods use 16 control points for the regular patches and 20 control points for the irregular ones. This method is implemented in two passes with the regular and the irregular patches rendered separately. (Li et al., 2010) provides a method for approximating the Loop surface by interpolation and their method uses 29 input points per patch. This method is implemented using two passes for the regular and the irregular triangles. Our methods for approximating the Loop subdivision surface, described in Chapter 5, are implemented using a single pass and also provide the

ability to switch from one method to the other at runtime.

Chapter 3

GRAPHICS PIPELINE IN GAMES

3.1 Introduction

Fast rendering of polygons has been an established process for over a decade and it has been primarily driven by the processing power of the GPU. The common graphics libraries DirectX and OpenGL expose this power and developers reconfigure their applications to deliver increased realism by implementing rendering techniques Luebke and Humphreys (2007); Kautz (2004); Akenine-Moller and Haines (2002) like normal mapping, bloom lighting, shadows, and animation techniques Kry et al. (2002); Baran and Popović (2007) like skinning, rigging and facial animation. In most cases the polygons are rendered as triangles and animation is performed using a skeleton, which transforms the vertices of the triangle in realtime.

We have come a long way since the days of *doom*, this landmark game did many firsts including the creation of the game engine. The game engine was an effort to modularize the content creation and display mechanism in games so it can be separated out from the core game logic, which included game play, features, mechanics and actions pertinent to the game. In doing so developers were able to use the same engine or a revision of it for many games simultaneously. This helped streamline the production process and bring efficiency to the game industry. The *Quake* engine by ID software was the first major example where multiple games were produced using the same engine. In this chapter we describe the evolution of the graphics engine in games and how that ties to the asset creation process.

3.2 Surfaces in Games

In this section we describe the process of rendering inside a game engine and the steps that need to be taken for content to move from the creation tool to the screen. We also describe relevant methods that provide for smooth rendering of surfaces or patches. We describe two ways of achieving tessellation, a necessary condition for rendering

patches, one using the hardware tessellator and other by instancing a tessellation pattern using vertex and index buffer objects. We then describe current methods for rendering triangle as well as quad patches.

3.2.1 Polygonal Rendering Pipeline

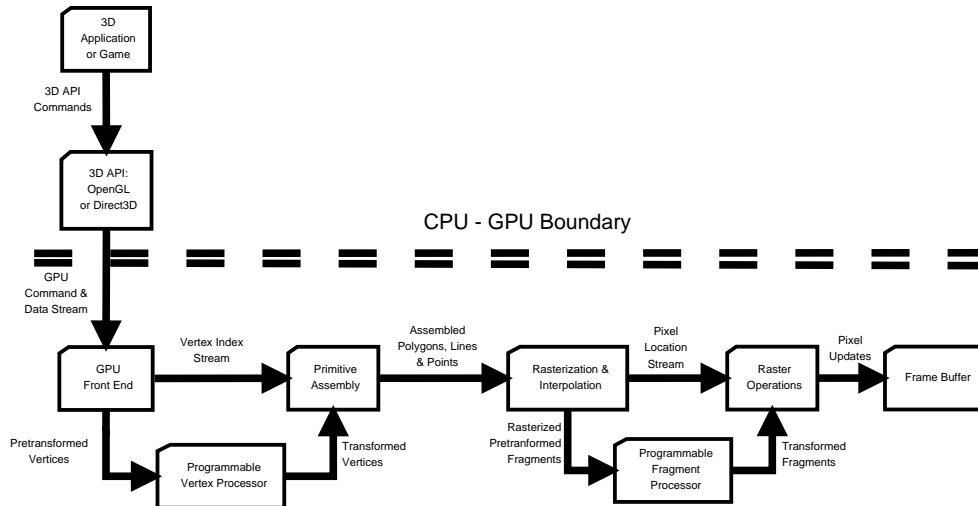


Figure 3.1: Polygon rendering pipeline

The central component of any game engine is the graphics core known as the renderer. The renderer closely connects to the graphics processing unit (GPU) at work and uses either DirectX or OpenGL to ultimately paint the picture on the screen. What is ultimately seen on the screen is a combination of various stages in the graphics pipeline and depends on many factors including the GPU, the type of the game, the audience and the controls the game provides. Most engines however are broad in their capabilities and are benchmarked on how they perform with respect to the current state of the art in 3D rendering. Figure 3.1 describes the polygonal rendering pipeline, the most prevalent pipeline in game engines today. The major stages of the pipeline are as follows:

- Determine what objects are in the scene, locate their assets e.g. textures, animations, geometry and locate the camera properties.

- Engine performs visibility and culling operations on the set of objects to determine which ones to draw and with what properties.
- The engine passes this subset to the renderer which in turn performs back face culling and transforms the objects from world to screen space in the vertex shader.
- The primitive assembly organizes the vertices into points, lines or triangles.
- The pixel shader performs the per fragment operations and sends the transformed fragments to the rasterizer, which in turn sends the final per pixel values to the frame buffer.

In this setup the 3D objects are rendered as polygons and the detail is captured and stored in bump or normal maps Policarpo et al. (2005). There are several tools that can reduce the polygon count for a high resolution model and store the detail in a normal map, the most common one being ZBrush Spencer (2008). In this approach the fragment shader does a texture lookup from the normal map and on a per pixel basis performs the lighting calculation to give the appearance of a highly detailed surface.

3.2.2 Tessellation by Instancing

The idea was first proposed in Boubekeur and Schlick (2005) and mainly simulates tessellation by instancing. With hardware and API support developed based on this idea the game engine can perform patch tessellation and render parametric surfaces using this feature. As shown in Figure 3.2 the CPU stores a coarse mesh and a refinement pattern is preserved inside vertex and index buffer objects and applied for each triangle in the coarse mesh during render time. Displacement data can be looked up and applied after performing a barycentric evaluation on the tessellated mesh to provide additional detail to the surface. Graphics APIs provide a mechanism to store the pattern and invoke it in the vertex shader at render time. One of the limitations is

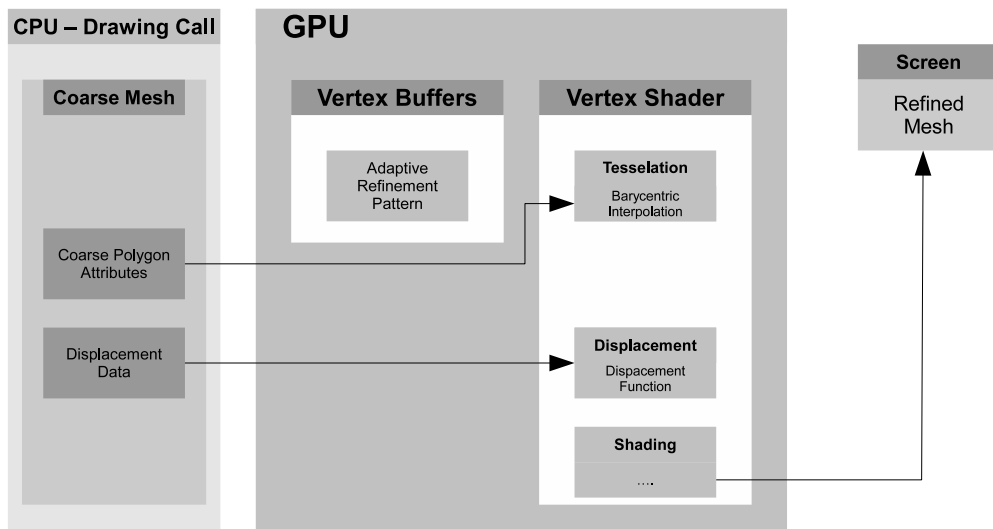


Figure 3.2: Tessellation using instancing

that level of detail cannot be changed on the fly as the refinement patterns are static. Several solutions have been proposed for adaptive tessellation using instancing; Boubekeur and Schlick (2008) use a matrix of refinement patterns and constrain the tessellation factors so that the size of this matrix is manageable. Dyken et al. (2009) use a snap function to collapse triangles and patch the adjacent faces correctly. In Chapter 4 we provide a new tessellation algorithm that is semi-uniform, local and does not require matrix storage or patching.

3.2.3 Hardware Tessellation

Subdivision surfaces have been available in modeling and animation tools and most asset production pipelines rely heavily on them. This is true in the case of games where they are the primary mode of rendering high quality cut-scenes. With the addition of a hardware tessellator unit to the graphics pipeline in Direct3D11 and OpenGL 4.0, the asset production pipeline can be simplified to a greater detail by having the content creators model both the cut-scenes and application level assets using subdivision surfaces. The tessellator unit adds two new programmable stages (See Figure 3.3), the hull and domain in Direct3D11 and control and evaluation in OpenGL 4.0, to the graphics pipeline. The control shader takes in the control points

for the input mesh, performs geometric operations, and passes a regular mesh to the tessellator unit. The tessellator adds edge tessellation factors for this input mesh and constructs a triangulated pattern (See Figure 3.3) and sends the parameter values to the evaluation shader. The evaluation shader evaluates the surface for u, v, w values passed from the tessellator and the controls points from the control shader. The evaluation shader is responsible for performing the appropriate parametric calculations for an input mesh of given control points. The main advantage is in the fact that we now store and manipulate only the control points for rendering and animation purposes. With this process any parametric surface can be rendered using hardware tessellation.

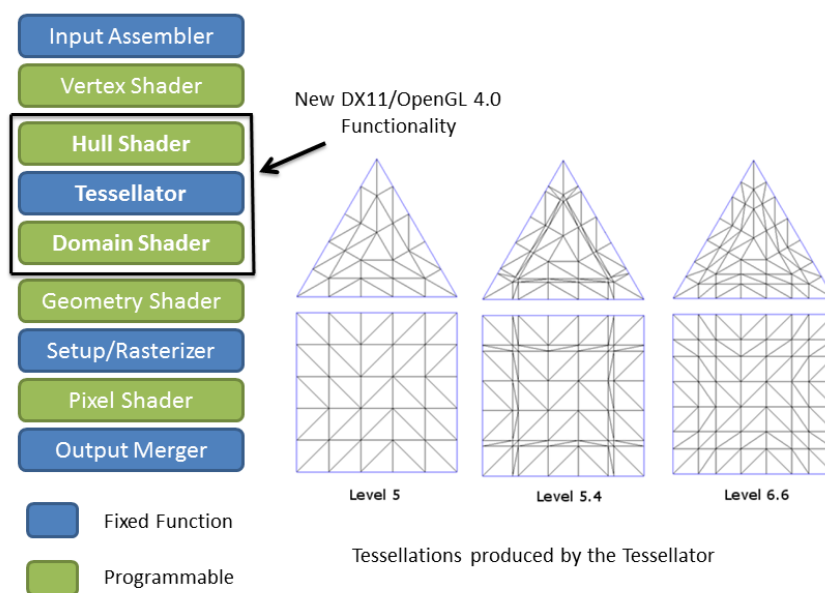


Figure 3.3: New graphics pipeline with the tessellator unit

3.2.4 Asset Production Pipeline

Game assets include everything that is not code Arnaud (2010). The game asset pipeline is the path that all models, textures, sound effects, levels, animations and other assets follow to go from the tool in which they were created to the actual game.

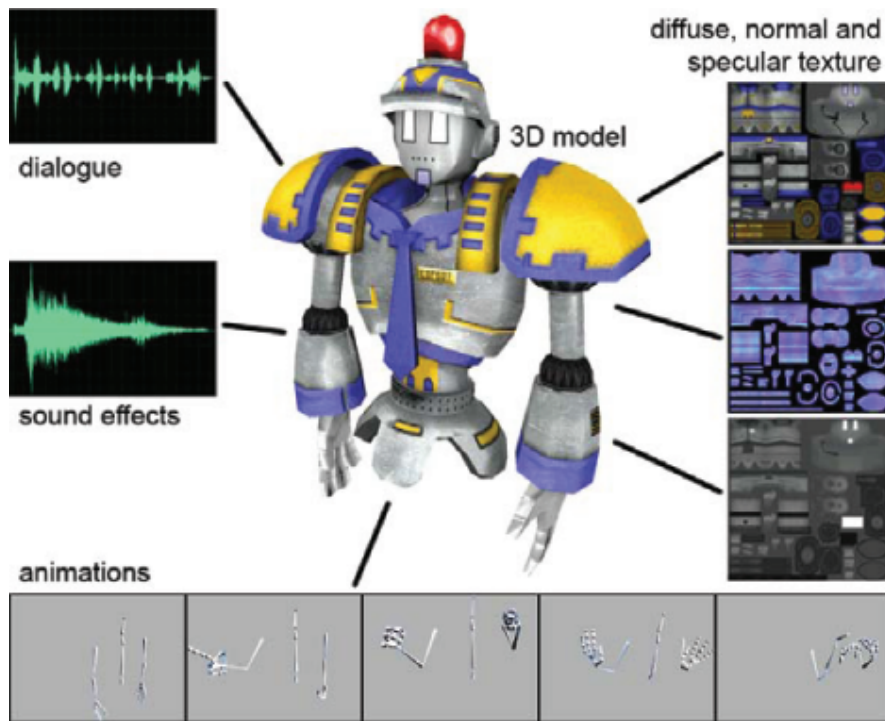


Figure 3.4: Asset production pipeline, model courtesy of Joost van Dongen

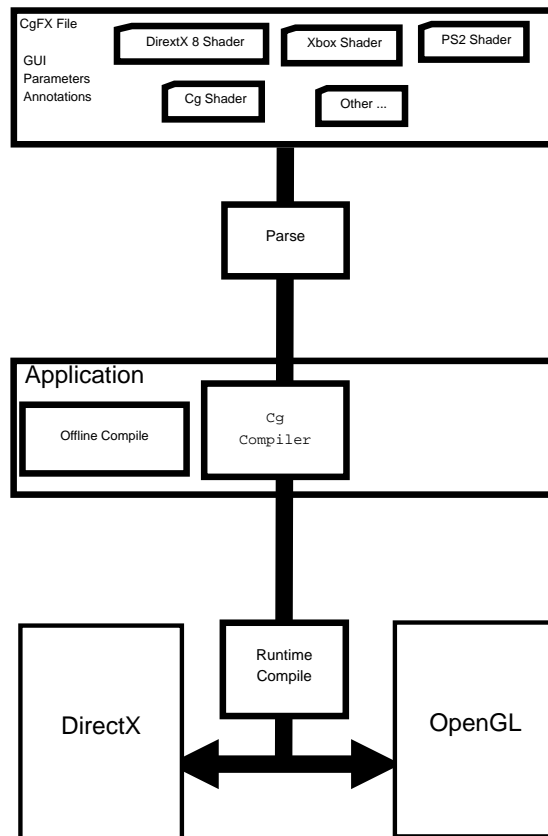


Figure 3.5: Cross platform rendering using CGFX

This path can include export, optimization, checking for correctness, pre-calculating data and the content management system that stores this data to keep track of versions. The main component of this pipeline is the 3D modeling, texturing and animation of game objects. As shown in Figure 3.4, a model has several assets associated with it including geometry, textures, sounds and animations. Most game engines interface with the asset creation software by the way of an intermediate production tool or platform that consolidates all the assets to be engine ready. For the purposes of rendering 3D models by polygonal or parametric methods, preprocessing of the original asset needs to be performed and represented in a uniform manner. One way of standardizing this process is by defining an effect file that contains the relevant rendering information for the model. We use the popular CGFX effect format described in the CG tutorial Fernando and Kilgard (2003) for our implementation discussions in Chapter 5. Figure 3.5 shows the cross platform rendering ability of the CGFX file format. The format helps illustrate multiple effects by breaking them into several techniques to achieve multi-pass rendering for the model. It also allows the engine to specify non-programmable rendering states such as blending and depth-test.

3.3 Background for Rendering Smooth Surfaces

In this section we define the building blocks that are necessary during several stages of our research. Since the tessellation hardware needs to construct patches that are locally defined and since our goal is to achieve smooth looking surfaces, methods are needed that convert common subdivision algorithms into patches. We discuss the background necessary for achieving this conversion and present some existing approaches that are currently the state of the art.

3.3.1 Notation

This thesis uses the following notation:

- All control points and vectors are denoted by boldface letters, for example \mathbf{b}_{00}

denotes a control point of a Bézier surface.

- All scalars are denoted by non boldface letters.
- All points that are intermediate and not part of the control mesh are denoted by capital non boldface letters.
- All functions, including those that generate surfaces are denoted by capital boldface letters.

3.3.2 Continuity

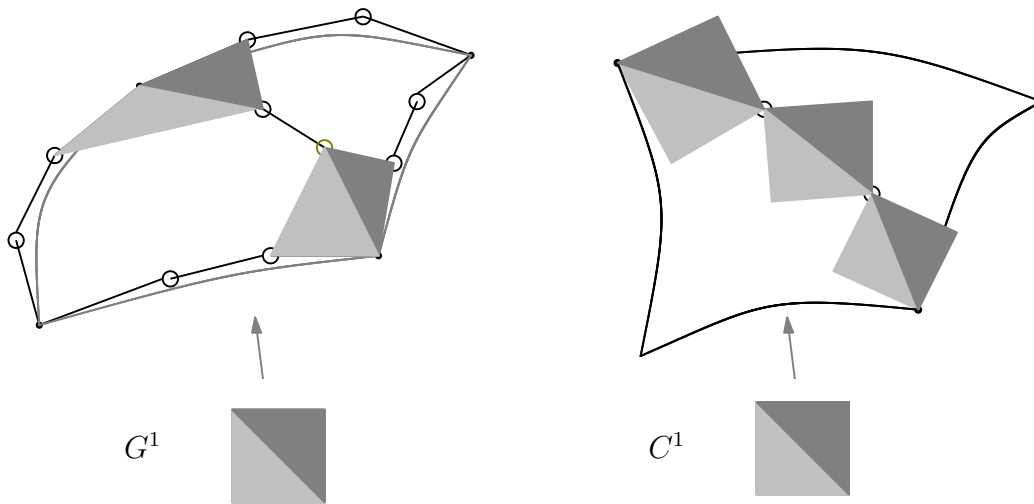


Figure 3.6: Difference between G^1 and C^1 continuity

Surfaces are said to exhibit G^0 continuity when any two faces on the surface share a common edge except at the boundary. G^0 surfaces that are differentiable once have a unique tangent plane and are called G^1 or geometrically continuous surfaces. These exhibit a unique normal \mathbf{n} at every point on the surface and it is computed as the cross product of two independent tangents in the tangent plane. In order for two adjacent patches to appear visually smooth, see Figure 3.7, they need to be G^1 , which means that for any point on the boundary curve the tangent along the edge and the tangents pointing towards the inside of each patch must be on the same plane. For surfaces to be C^1 continuous, they have to satisfy geometric continuity and the tangent

planes need to be an affine map of the domain triangles. Figure 3.6 shows two adjacent cubic triangles that are G^1 and C^1 continuous. For more detailed explanation on continuity conditions see Peters (2002). When the boundary curves for any two adjacent patches are known, Piper (1987) shows that cubics do not have enough degrees of freedom to enforce tangent plane continuity across the shared boundary. Therefore a quartic needs to be constructed for ensuring G^1 continuity using Bézier triangles. The tangent plane continuity conditions are described in Section 3.4.

3.3.3 Reflection Lines

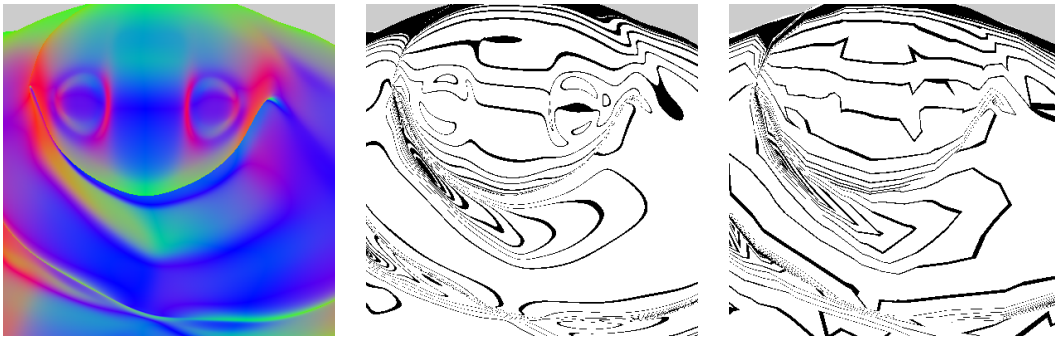


Figure 3.7: Smooth normals rendered as color for a G^1 surface (left), reflection lines: G^1 (middle) and G^0 (right)

A well known surface interrogation technique Hagen et al. (1992) is by rendering reflection lines. Even and continuous flow in these lines indicate a smooth surface. In terms of application they are useful in comparing approximation techniques with the original methods by visually representing how close the lines match. Figure 3.7 compares the reflection lines for a G^1 surface with a G^0 surface. Reflection lines are implemented by a technique described in Klass (1980), wherein parallel light lines are projected on to the surface that can be seen from a fixed view-point. Let $\mathbf{X}(u, v)$ be a surface and \mathbf{n} be the normal on the surface and a light line can be defined for a parameter t as $\mathbf{L}(t) = L_0 + \mathbf{l}_d * t$, where L_0 is the light position and \mathbf{l}_d is the light direction vector. Then the reflection line is the projection of $\mathbf{L}(t)$ on $\mathbf{X}(u, v)$ as seen from a fixed eye point A . Appendix B shows the

implementation of reflection lines in the fragment shader.

3.3.4 Continuous Tangent Frames

In addition to rendering smooth surfaces at a significantly less cost, moving to the parametric domain has other advantages for game developers. Tangent frame is common term given to tangent space basis vectors used for calculations during normal mapping. Tangent frame for each vertex has to be pre-calculated and then skinned to produce the needed detail during normal mapping. With the parametric approach vertex operations such as skinning are performed before tessellation on a small number of control points. The mesh is tessellated, smooth normals are then calculated per vertex relative to the animation operations. This reduces the complexity of the normal mapping pipeline and improves shading quality. For a detailed explanation of the interplay between the tangent frame, animation pipeline and normal mapping refer Lengyel (2004).

3.4 Locally Defined Parametric Surfaces

We split our discussion into methods that deal with parametric patches and those that deal with subdivision surfaces. Most realtime rendering applications and especially games render the polygons as triangles. The focus of this dissertation is therefore on rendering smooth triangles and leads to the development of a unified rendering framework presented in Chapter 5. However from a modeling standpoint quads are easier to represent smooth surfaces and we discuss methods that deal with smooth rendering of quads in the next section. In the next few sections we describe the construction of triangular and tensor product Bézier surfaces and methods for approximating subdivision surfaces by constructing Bézier patches.

3.4.1 Triangular Bézier Patches

A parametric triangular patch in Bézier form is defined by:

$$\mathbf{P}(u, v, w) = \sum_{i+j+k=n} \frac{n!}{i!j!k!} u^i v^j w^k \mathbf{b}_{ijk} \quad (3.1)$$

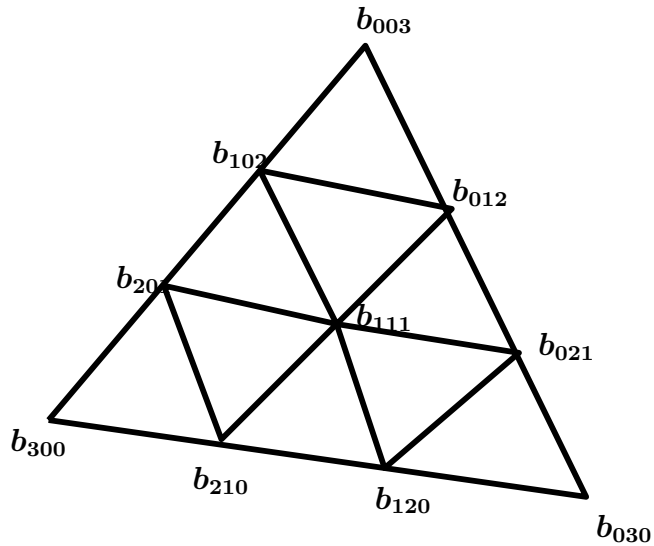


Figure 3.8: Control points of a cubic Bézier patch

where b_{ijk} are the control points of the Bézier triangle and $(u, v, w = 1 - u - v)$ are the barycentric coordinates with respect to the triangle Farin (2002). Setting $n = 3$ gives a cubic Bézier triangle as used in the following construction. see Figure 3.8.

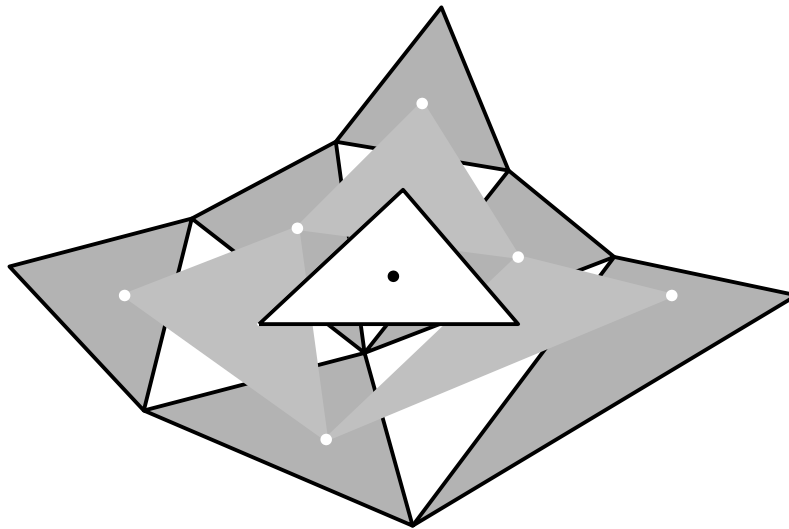


Figure 3.9: Point on the surface and the tangent plane (white triangle) calculated using the de Casteljau algorithm

The de Casteljau algorithm for triangular patches is a n recursive linear interpolation algorithm where n is the degree of the patch. Figure 3.9 shows the construction for a cubic Bézier triangle and a point on the surface is obtained by repeating the interpolation process for 3 times. Another important benefit of this

algorithm is that it provides the tangent plane at $n - 1$ recursion level. So for the cubic case the tangent plane is shown by the white triangle in Figure 3.9. One can therefore simultaneously evaluate the point and normal at any given parameter value using the de Casteljau algorithm.

3.4.2 G^1 Continuity of Quartic Bézier Triangles

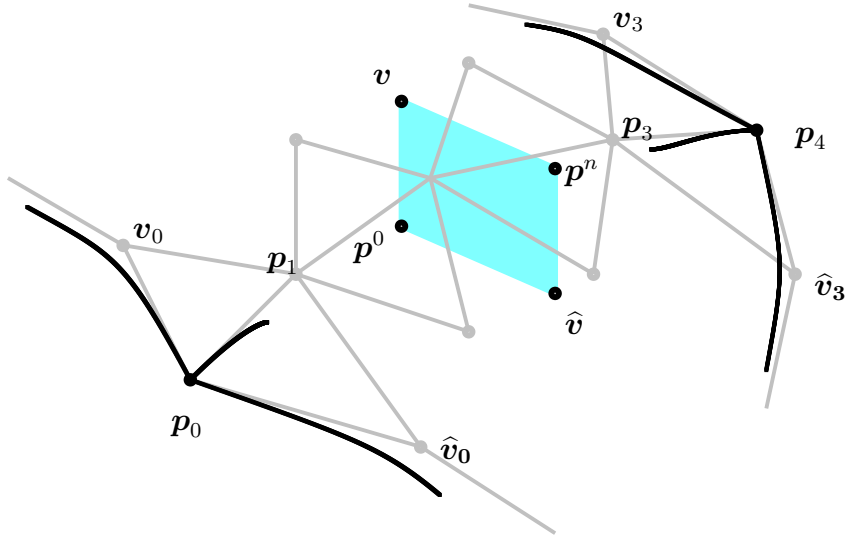


Figure 3.10: G^1 Continuity for a quartic Bézier patch boundary , the blue plane is the result of applying de Casteljau algorithm on the G^1 constraints

Consider two quartic triangular patches that share an edge as shown in Figure 3.10 and let $\mathbf{p}(t)$ be a point on the boundary curve common to the two patches. This point can be constructed using de Casteljau algorithm for both these patches. The algorithm also provides the tangent planes for this point, calculated both times. For the two patches to be G^1 the two planes have to be coplanar for all values of t . The points $\mathbf{p}^0, \mathbf{v}, \mathbf{p}^n, \hat{\mathbf{v}}$ have to be on the same plane. This is shown in Figure 3.10. Which means that the lines $\overline{\mathbf{p}^0\mathbf{p}^n}$ and $\overline{\mathbf{v}\hat{\mathbf{v}}}$ intersect. Therefore for all t Farin (2002), there are functions $\lambda(t)$ and $\mu(t)$ such that:

$$(1 - \lambda(t))\mathbf{v}(t) + \lambda(t)\hat{\mathbf{v}}(t) = (1 - \mu(t))\mathbf{p}^0(t) + \mu(t)\mathbf{p}^n(t) \quad (3.2)$$

Figure 3.10 shows the shared edge of a quartic Bézier patch and for enforcing G^1 across the two patches, we only need to consider the two parallel rows of control points for each patch.

3.4.3 PN-Triangles

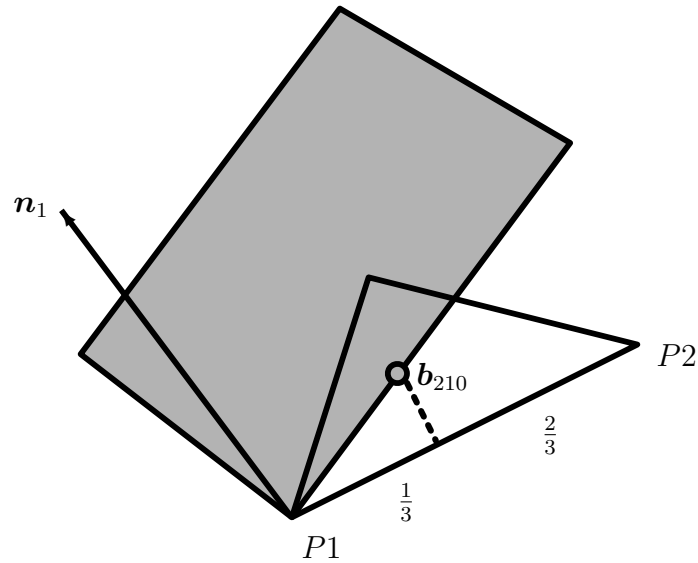


Figure 3.11: The edge point b_{210} of the Bézier triangle is calculated by projecting $(2P_1 + P_2)/3$ into the tangent plane at P_1

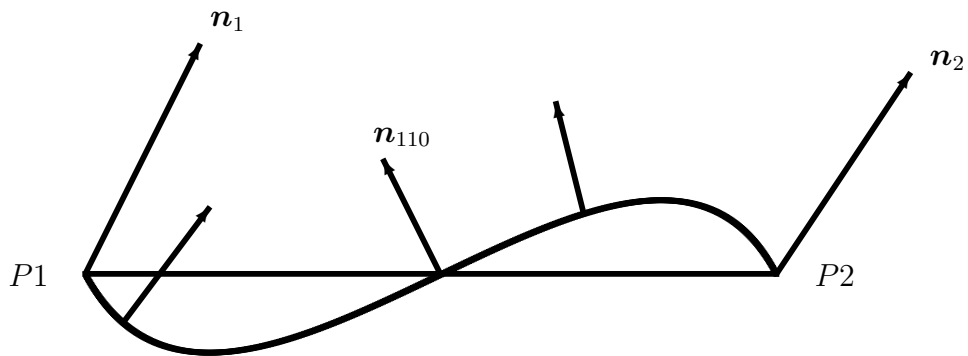


Figure 3.12: Quadratic interpolation of the normals is needed to pick up inflection points

Vlachos et al. Vlachos et al. (2001) propose *curved PN-triangles* for interpolating a triangle mesh by a parametric, piecewise cubic surface. This established technique generates a G^0 continuous surface, stays near the original control polygon and thus avoids self-interference. At first, the PN scheme places the

intermediate points B_{ijk} at the positions $(ib_{300} + jb_{030} + kb_{003})/3, i + j + k = 3$, leaving the three corner points unchanged. Then, each control point b_{ijk} on the border is constructed by projecting the intermediate point B_{ijk} into the plane defined by the nearest corner point and its normal, see Figure 3.11. Finally, the central control point b_{111} is constructed by moving the point B_{111} halfway in the direction $M - B_{111}$ where M is the average of the six control points computed on the borders as described above. The construction uses only data local to each triangle: the three triangle vertices and its normals. This makes it especially suitable for a triangle rendering pipeline. Quadratic interpolation of the normals is performed, see Figure 3.12, for picking up shape variations. The mid-edge normal is calculated by reflecting the average of the two end normals across a plane perpendicular to the edge.

3.4.4 Triangular Gregory Patches

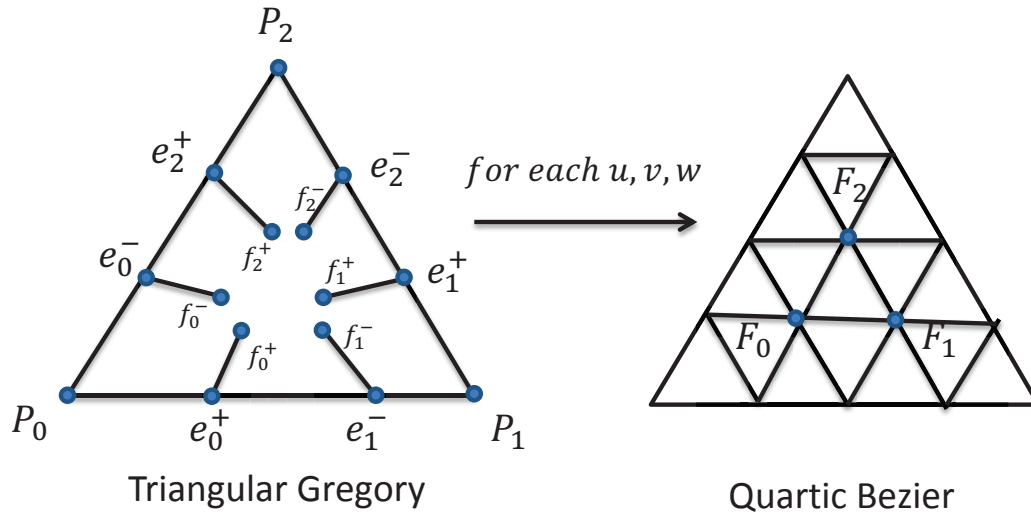


Figure 3.13: Control points for triangular Gregory and its relation with quartic Bézier

A triangular Gregory patch Gregory (1974); Chiyokura et al. (1990) is a modified form of a quartic Bézier triangle where in the boundary curves are cubic and the interior surface is a quartic. However the inner control points of the patch are dependent on the $(u, v, w = 1 - u - v)$ parameter values on the domain. This leads to

the formulation of interior points represented by F_0, F_1, F_2 in Figure 3.13 for each parameter value in the domain. This construction was introduced by Gregory to ensure that a pair of patches meeting at a shared edge are tangent plane continuous across that edge. The patch is evaluated by the following equation:

$$\begin{aligned} \mathbf{T}(u, v, w) = & u^3 \mathbf{p}_0 + v^3 \mathbf{p}_1 + w^3 \mathbf{p}_2 + \\ & 3uv(u+v)(u\mathbf{e}_0^+ + v\mathbf{e}_1^-) + \\ & 3vw(v+w)(v\mathbf{e}_1^+ + w\mathbf{e}_2^-) + \\ & 3wu(w+u)(w\mathbf{e}_2^+ + u\mathbf{e}_0^-) + \\ & 12uvw(uF_0 + vF_1 + wF_2) \end{aligned}$$

where

$$\begin{aligned} F_0 &= \frac{w\mathbf{f}_0^- + v\mathbf{f}_0^+}{v+w}, \\ F_1 &= \frac{u\mathbf{f}_1^- + w\mathbf{f}_1^+}{w+u}, \\ F_2 &= \frac{v\mathbf{f}_2^- + u\mathbf{f}_2^+}{u+v} \end{aligned} \quad (3.3)$$

Figure 3.13 shows the control point labeling for the Gregory Patch and how it maps to a quartic Bézier patch.

3.4.5 Bi-Cubic Bézier Patches

Smooth surface rendering is possible for quad patches and can be determined locally by converting the content data to bi-cubic Bézier patches. A n-degree parametric quadrilateral patch in Bézier form is defined by

$$\mathbf{P}(u, v) = \sum_{i=0}^n \sum_{j=0}^n \mathbf{b}_{i,j} h_i(u) h_j(v), \quad h_k(t) = \frac{n!}{(n-k)!k!} (1-t)^{n-k} t^k \quad (3.4)$$

where $\mathbf{b}_{i,j}$ are the control points of the Bézier quad and (u, v) are the parameter values with respect to the quad Farin (2002). Setting $n = 3$ gives a bi-cubic Bézier quad as used in the following construction. see Figure 3.14.

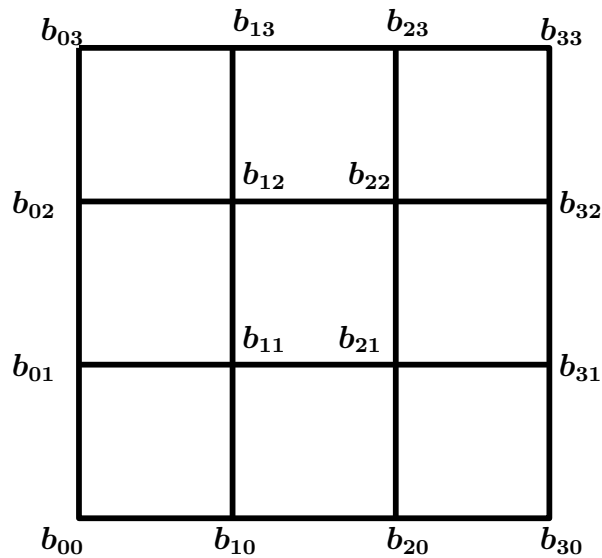


Figure 3.14: Control points of a bi-cubic Bézier patch.

3.5 Subdivision Surfaces

In these section we describe the implementation details for various subdivision surface rendering methods and how they can be approximated using Bézier and Gregory patches. By starting with an input mesh consisting of polygons, regular polygons are directly converted to Bézier patches and irregular polygons are approximately rendered using Gregory patches. The methods described in this section perform operations on the one-ring neighborhood of the input mesh to derive the various patch control points.

3.5.1 Catmull-Clark Subdivision Surface

The Catmull-Clark scheme was described in Catmull and Clark (1978b). It is based on the tensor product bi-cubic spline. The masks are shown in Figure 3.15. The scheme produces surfaces that are C^2 continuous everywhere except at extraordinary vertices, where they are C^1 continuous.

The limit point and limit tangents for the Catmull-Clark scheme to include both quad and triangles was derived in Loop et al. (2009) and can be calculated by the following equations:

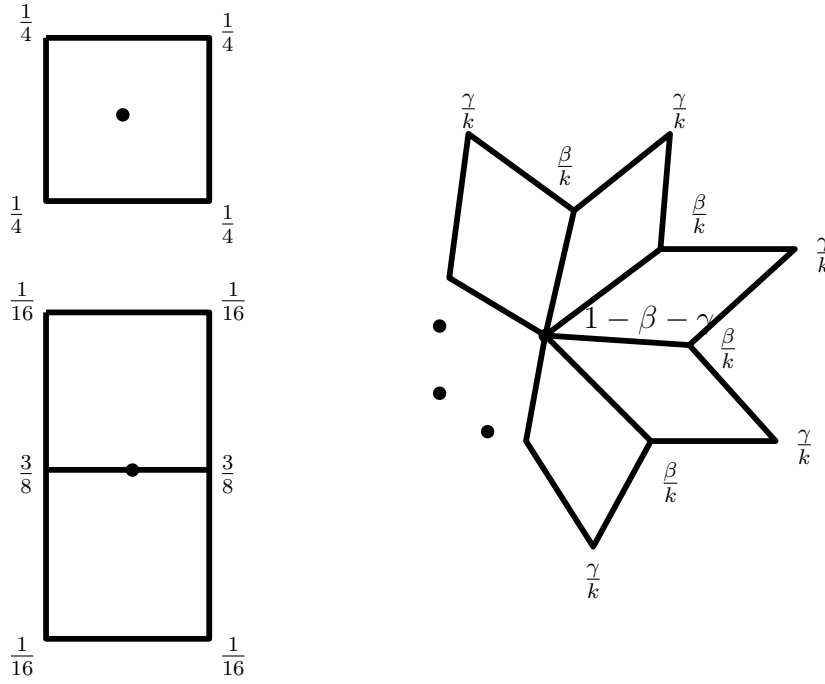


Figure 3.15: Masks for calculating the face(top-left), edge(bottom-right) and vertex(right) points using Catmull-Clark subdivision, $\beta = 3/2k$ and $\gamma = 1/4k$ and k is the valence at the vertex

$$\mathbf{p}_0^\infty = \frac{n-3}{n+5} \mathbf{p}_0 + \frac{4}{n(n+5)} \sum_{i=0}^{n-1} (M_i + C_i) \quad (3.5)$$

$$\mathbf{t}_{0,i} = \frac{2}{n} \sum_{i=0}^{n-1} \left(\left(1 - \sigma \cos\left(\frac{\pi}{n}\right) \right) \cos\left(\frac{2\pi * i}{n}\right) M_i + 2\sigma \cos\left(\frac{2\pi * i + \pi}{n}\right) C_i \right) \quad (3.6)$$

M_i and C_i are the midpoints and centroids of the i^{th} edge surrounding \mathbf{p}_0 . This is shown in Figure 3.16.

It naturally follows that the limit normal can be calculated by the cross product of two tangent vectors at the limit point. Direct evaluation for the surface at arbitrary u, v values is possible and presented in Stam (1998a).

3.5.2 Approximating Catmull-Clark Surfaces

In this section we describe the Approximate Catmull-Clark (ACC) subdivision method that works with meshes that are quad-dominant, implying that the mesh has mostly

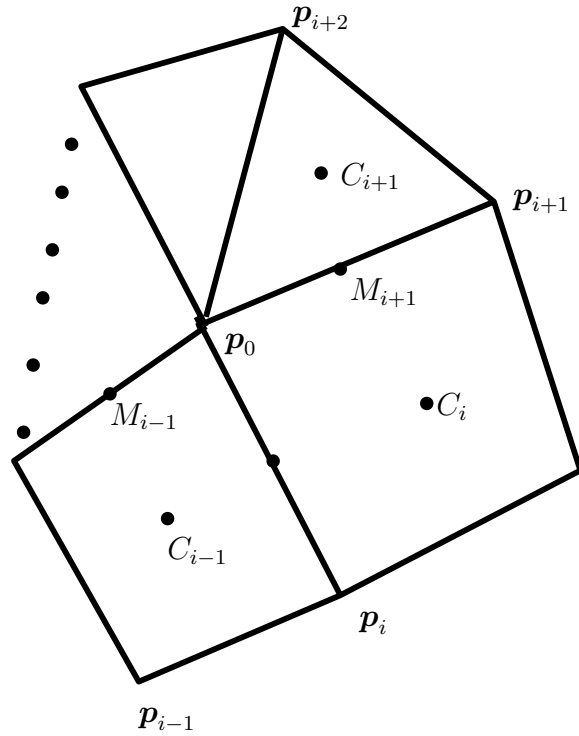


Figure 3.16: The one ring neighborhood of p_0 showing C_i and M_i

quads and a few triangles. Rendering quads has been available for game engines in past few iterations of the graphics hardware, however application have been slow to adapt. With the availability of the hardware tessellation unit and the ability to merge the asset production pipeline to create both high quality cut scene content and in-game content, game engines will be ready for incorporating Catmul-Clark surfaces.

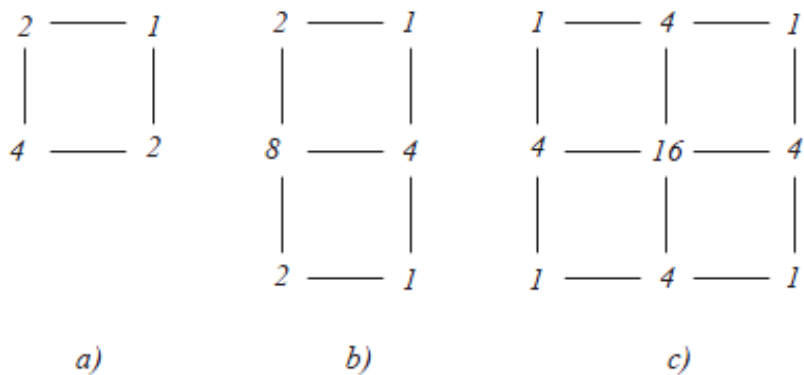


Figure 3.17: Masks for converting a regular Catmull-Clark quad into a bi-cubic Bézier control mesh

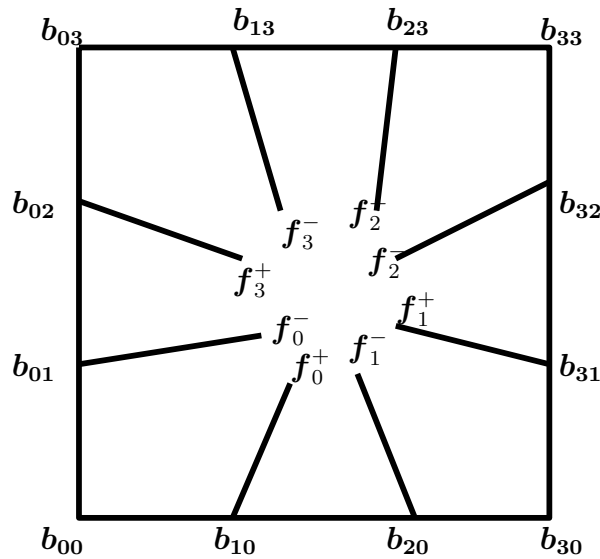


Figure 3.18: The quad Gregory control polygon

The fastest method to approximate Catmull-Clark surfaces is by applying the point-normal strategy to quads in a manner similar to PN-triangles Vlachos et al. (2001). The limit Catmull-Clark points and normals can be passed to the control shader and a bi-cubic Bézier patch can be constructed from the boundary curves. In Loop et al. (2009) a method for approximating Catmull-Clark surfaces using a combination of bi-cubic Bézier and Gregory patches is proposed. The Bézier patches are used for regular quads, all vertices have a valence of 4, and the irregular quads and triangles are approximated by using quad and tri Gregory patches. For regular quads the inner Bézier points are calculated by the masks shown in in Figure 3.17, a) calculates the four inner points, b) the edge points and c) the corner points. The construction of the tri Gregory patch is shown in Section 3.4.4 and Figure 3.18 shows the construction for quads. The evaluation is performed using Equation 3.4, the regular case uses 16 control points calculated by the masks shown in Figure 3.17 and for irregular quads the 20 Gregory control points are calculated using the limit points in Equation 3.5 and the limit tangents in Equation 3.6 and the inner control points are calculated for each u, v by the following equation:

$$\begin{aligned}
\mathbf{b}_{11} &= \frac{u\mathbf{f}_0^- + v\mathbf{f}_0^+}{v+u}, & \mathbf{b}_{21} &= \frac{(1-u)\mathbf{f}_1^- + v\mathbf{f}_1^+}{1-u+v}, \\
\mathbf{b}_{12} &= \frac{(1-u)\mathbf{f}_2^+ + (1-v)\mathbf{f}_2^-}{2-u-v}, & \mathbf{b}_{22} &= \frac{u\mathbf{f}_3^- + (1-v)\mathbf{f}_3^+}{1+u-v}
\end{aligned} \tag{3.7}$$

Similarly Equations 3.15 and 3.16 are used for the triangular Gregory patch. The last step is to calculate the inner Gregory points or the f points shown in Figures 3.13 and 3.18. For tangent plane continuity the versal (along the edge) and the two traversal (pointing to the interior) derivatives are linearly dependent. See Figure 3.19.

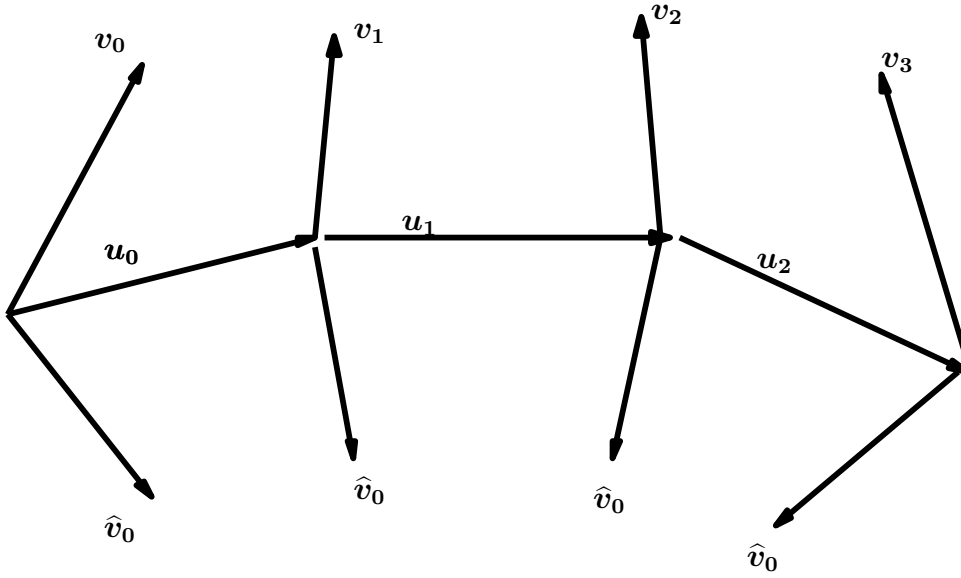


Figure 3.19: The three tangent fields that satisfy the continuity constraints

$$\begin{aligned}
\mathbf{u}(t) &= \mathbf{B}^2(t) \cdot [\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2] \\
\mathbf{v}(t) &= \mathbf{B}^3(t) \cdot [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] \\
\hat{\mathbf{v}}(t) &= \mathbf{B}^3(t) \cdot [\hat{\mathbf{v}}_0, \hat{\mathbf{v}}_1, \hat{\mathbf{v}}_2, \hat{\mathbf{v}}_3]
\end{aligned} \tag{3.8}$$

The values of \mathbf{u} vectors and $\mathbf{v}_0, \hat{\mathbf{v}}_0, \mathbf{v}_3, \hat{\mathbf{v}}_3$ are known. The remaining \mathbf{v} vectors can be written in terms of the f points. Value of d is set to 3 for quads and 4 for triangles. The notation is shown in Figure 3.19.

$$\begin{aligned}\mathbf{v}_1 &= d(\mathbf{f}_0^+ - \mathbf{e}_0^+) \\ \mathbf{v}_2 &= d(\mathbf{f}_1^- - \mathbf{e}_1^-)\end{aligned}\quad (3.9)$$

The condition for tangent plane continuity is given by:

$$((1-t)c_0 - tc_1)\mathbf{u}(t) = 1/2(\mathbf{v}(t) + \hat{\mathbf{v}}(t)) \quad (3.10)$$

where $c_i = \cos(2\pi/n_i)$, $i = 0, 1$ and n_0, n_1 are the valence at $t = 0$ and $t = 1$. Note that this condition takes the same form as Equation 3.2 by choosing $\mu(t)$ as $((1-t)c_0 - tc_1)$ and $\lambda(t)$ as 0.5.

Substituting $t = 1/3$ in the above equation

$$\frac{2}{3}c_0\mathbf{u}_1 - \frac{1}{3}c_1\mathbf{u}_0 = 1/2(\mathbf{v}_1 + \hat{\mathbf{v}}_1) \quad (3.11)$$

Equation 3.11 will be satisfied by constructing a traversal vector \mathbf{r} such that

$$\begin{aligned}\mathbf{v}_1 &= \frac{2}{3}c_0\mathbf{u}_1 - \frac{1}{3}c_1\mathbf{u}_0 + \mathbf{r} \\ \hat{\mathbf{v}}_1 &= \frac{2}{3}c_0\mathbf{u}_1 - \frac{1}{3}c_1\mathbf{u}_0 - \mathbf{r}\end{aligned}\quad (3.12)$$

The value of \mathbf{r} is chosen in such a manner that it satisfies the case when the patch is regular, i.e. $n = 4$.

$$\mathbf{r}_0^+ = \frac{1}{3}(M_{i+1} - M_{i-1}) + \frac{2}{3}(C_i - C_{i-1}) \quad (3.13)$$

Equating 3.9 with 3.12 the f point is obtained as:

$$\mathbf{f}_0^+ = \frac{1}{d}(c_1\mathbf{p}_0 + (d - 2c_0 - c_1)\mathbf{e}_0^+ + 2c_0\mathbf{e}_1^- + \mathbf{r}_0^+) \quad (3.14)$$

Figure 3.20 shows the results of rendering this method on a tessellation enabled GPU. The red regions represent irregular patches rendered using Gregory patches. The resulting surface is G^1 continuous as indicated by the smooth normals. Figure 3.21 shows some of the limitations of the method. The tangents at the corners do not align with the boundary edges when there are skinny and long faces in the mesh. The rendered surface has wells and undulations when the algorithm is applied to a triangulated version of the same mesh, it is evident from the rendering that the method does not provide visually pleasing results for triangles. In the next section we develop a variant of the above approach for approximating the Loop subdivision scheme on triangular meshes.

3.5.3 The Loop Subdivision Surface

The Loop subdivision scheme Loop (1987); Amresh et al. (2002) is a face split scheme based on triangular box splines Seidel (1992) and at every subdivision step it calculates a new vertex for each existing one and a new vertex for each edge. The masks for these are shown in Figure 3.22. A limit point is the result of applying infinite number of subdivision steps to the input vertex or in our case the input control points. Given a vertex \mathbf{p}_0 and its 1-ring of adjacent vertices $\mathbf{p}_i, i = 1, \dots, n$, we can calculate the limit point and limit tangents for the Loop control mesh directly Li et al. (2010) as shown by Equations 3.15 and 3.16:

$$\mathbf{p}_0^\infty = \frac{\omega_n}{n + \omega_n}\mathbf{p}_0 + \frac{1}{n + \omega_n} \sum_{i=1}^n \mathbf{p}_i$$

where $\omega_n = \frac{3n}{8\alpha_n}$ and $\alpha_n = \frac{5}{8} - \left(\frac{3}{8} + \frac{2}{8} \cos \frac{2\pi}{n}\right)^2$ (3.15)

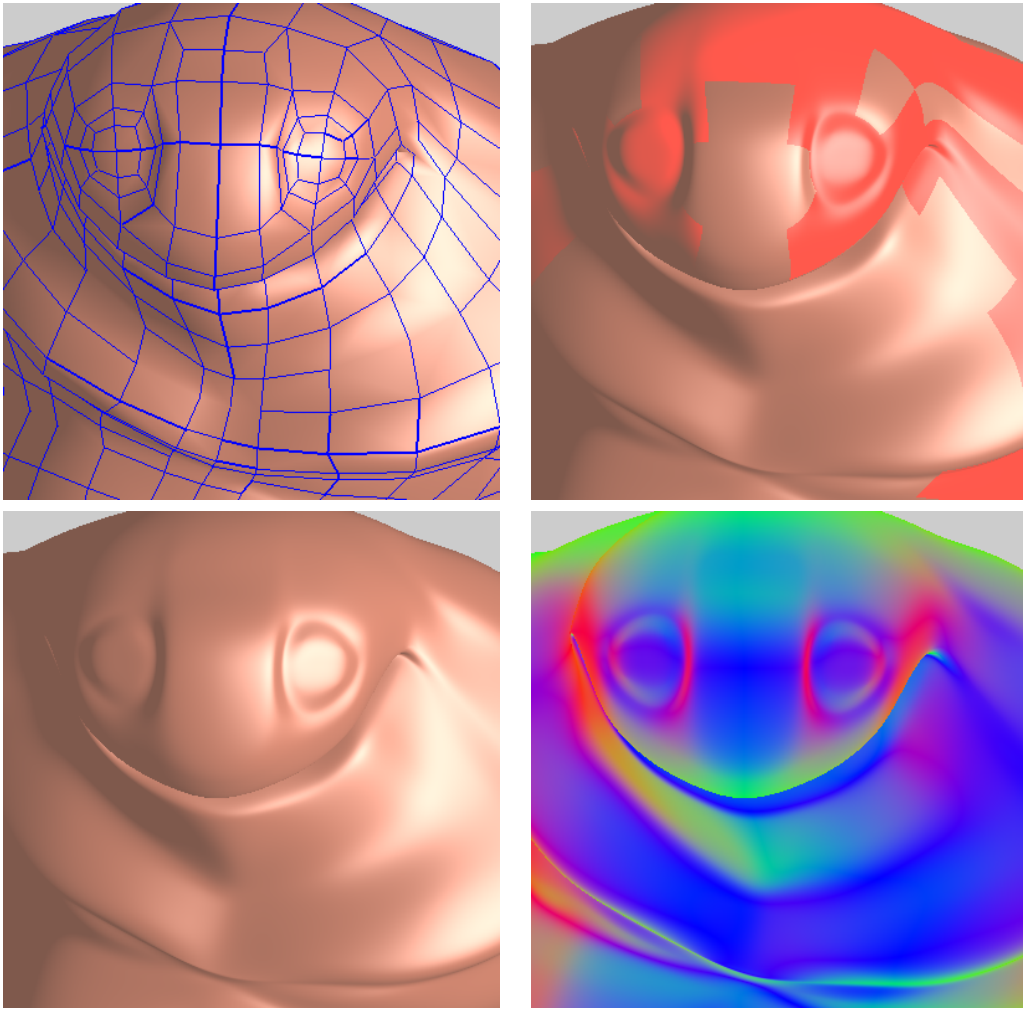


Figure 3.20: ACC Subdivision, top row: original mesh, irregular patches in red, bottom row: rendered surface and surface normals

$$\mathbf{t}_{0,i} = \sum_{j=0}^{n-1} \frac{2}{n} \cos \frac{2\pi j}{n} \mathbf{p}_{1+\text{mod}(i+j-1,n)}, \quad i = 1, \dots, n \quad (3.16)$$

It naturally follows that the limit normal can be calculated by the cross product of two tangent vectors at the limit point. Direct evaluation for the loop scheme at arbitrary u, v, w values is possible and presented in (Stam, 1998b).

3.5.4 Approximating Loop Subdivision Surfaces

In this section we will illustrate how to approximate the Loop subdivision surface by using the points, normals and tangents obtained in Section 3.5. We start by introducing methods that require less input control points and then move on to those

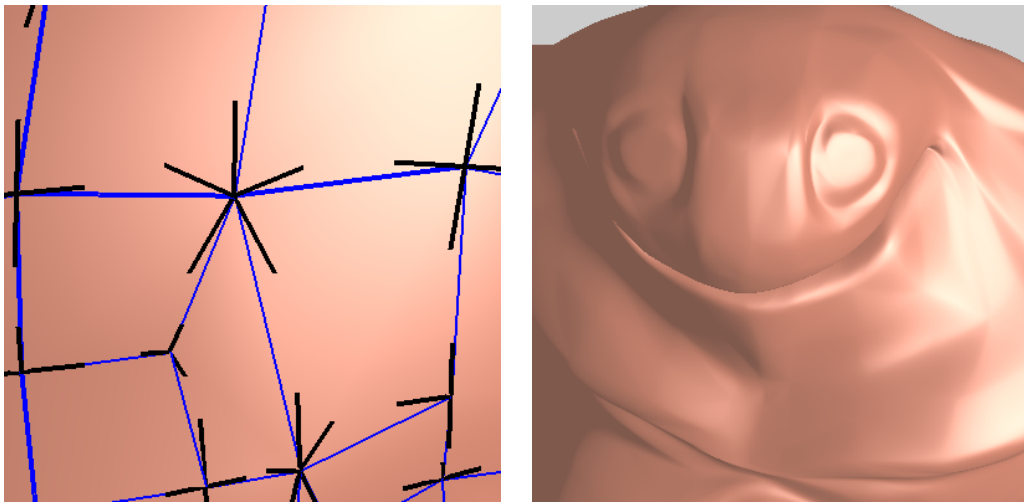


Figure 3.21: Limitations: Improper alignment of the tangents (left) and tri patch rendering (right)

that would require more. We also discuss alternate implementation details in the control shader and provide performance trade-offs.

PN-Triangles

The PN-Triangles method is described in Section 3.4. The construction uses only data local to each triangle: the three triangle vertices and its normals. This makes it especially suitable for a triangle rendering pipeline. Substituting the Loop limit points and normals from Equations 3.15 and 3.16, we get an approximate Loop representation using PN-Triangles.

Walton-Meek Triangles

Walton and Meek (Walton and Meek, 1996) proposed the WM method for achieving G^1 patch from the boundary curves of a triangular control net. Both the WM method and the Gregory Method, described in the next section, construct a triangular Gregory patch to evaluate the surface. The boundary curves are obtained in the same manner as PN-triangles and the inner points are calculated for each u, v, w , this ensures that any two neighboring triangles have a common tangent plane along their shared boundary. The main difference between the WM and Gregory method is based on where and how the calculations for the tangent plane continuity occur, the WM performs this

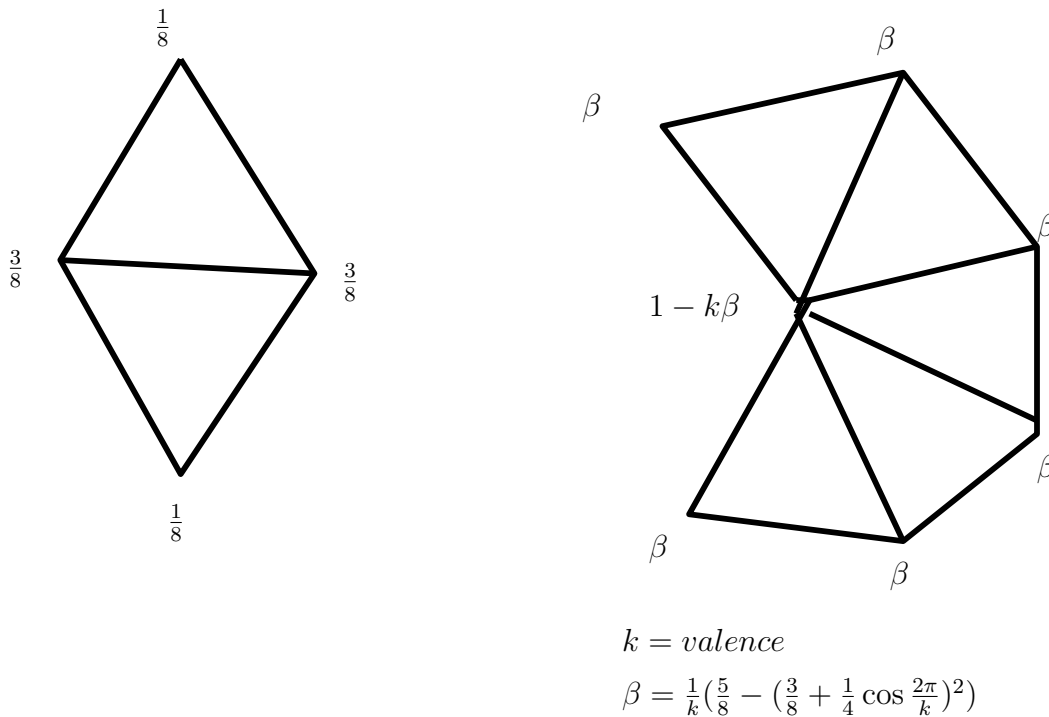


Figure 3.22: The masks for the Loop subdivision scheme, edge mask (left) and vertex mask (right)

inside the control shader while the Gregory processes this information and sends it to the control shader. We pass the limit Loop points (3) and normals (3)a total of 6 control points for this method. The WM method constructs the inner points based only on the boundary control points as the 1-ring neighborhood is not available inside the control shader. It is therefore not as smooth as the Gregory method but much smoother than the PN method. A simple explanation of the WM method is provided below, for details refer to (Walton and Meek, 1996):

- Degree elevate the cubic boundary curve, formed by the limit Loop points and normals, to a quartic and get the boundary control points.
- Since each interior point of a quartic (F_0, F_1, F_2) is associated with two boundary curves, it is determined twice to obtain the 6 interior Gregory control points.

- For each u, v, w evaluate the patch by blending the six Gregory points and obtaining the interior quartic points as shown in Equation 3.3.

Gregory Triangles

As shown in Figure 3.13, we need to calculate 15 control points for evaluating a triangular Gregory surface. For each u, v, w parameter value the boundary control points are degree elevated from cubic to quartic using the two corner and two edge points and the the inner quartic points F_0, F_1, F_2 are calculated from the inner six Gregory control points as shown in Equation 4. So an approximation of the Loop surface is possible by calculating these 15 control points, 3 corner, 6 edge and 6 inner points for the Gregory triangle.

The corner points are set to the limit Loop control points, see Equation 3.15, the edge points are calculated using the limit Loop tangents, see Equation 3.16, however we need to choose the right length for these tangents to get the edge points. We also know that the derivative at the end points of a Gregory patch is $3(e_0^+ - p_0)$, therefore we can solve e_0^+ by

$$e_0^+ = p_0 + \frac{2}{3}t_{0,1}\lambda \quad (3.17)$$

where $t_{0,1}$ is the Loop limit tangent from p_0 to p_1 and λ is chosen to be the subdominant eigenvalue of the Loop surface and is given by

$$\lambda = \frac{3}{8} + \frac{1}{4} \cos\left(\frac{2\pi}{n}\right) \quad (3.18)$$

We have to now construct the inner points so that the surface is tangent plane continuous across the edge. We use the method described in Section 3.5.2 of Chapter 3, we find that setting the traversal vector r to be equal to the Loop surface cross tangents (Figure 3.24), provides the best results. The traversal vector r is calculated by performing two levels of Loop subdivision on the original control mesh and evaluating the cross tangents at $1/4, 1/2, 3/4$ along each edge and then linearly

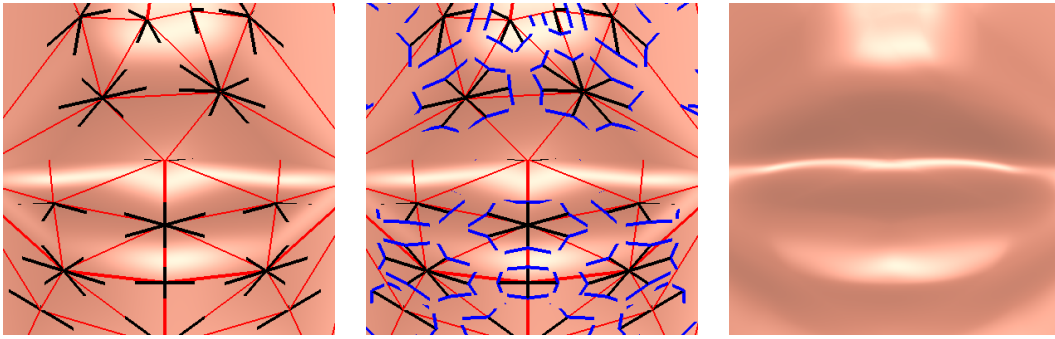


Figure 3.23: Lines showing Loop limit tangents in black, the original mesh in red, the inner points in blue and the result of rendering (right)

interpolating them to find the values at $1/3, 2/3$. This construction is not strictly G1 in all cases but, based on our experiments, perfectly adequate for gaming applications.

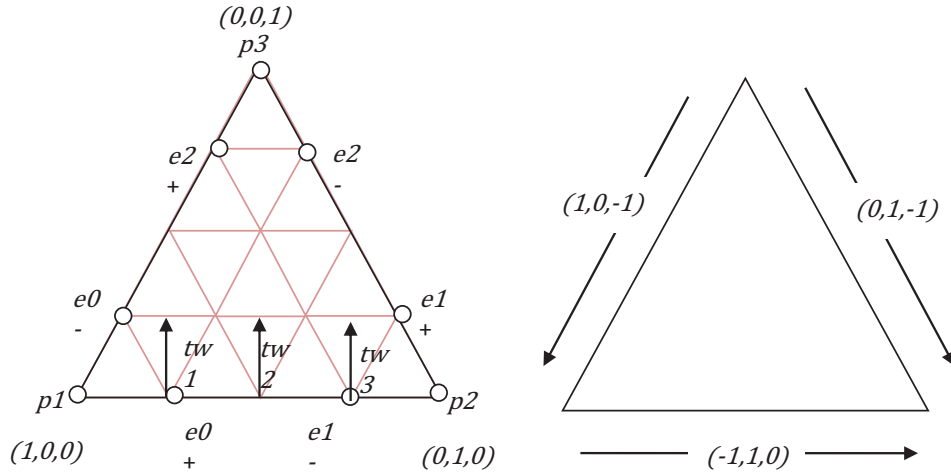


Figure 3.24: Loop Cross tangents at 0.25, 0.5 and 0.75 shown by tw_1, tw_2 and tw_3

The equation for the inner points is given by

$$\mathbf{f}_0^+ = \frac{1}{4} \left(c_1 \mathbf{p}_0 + (4 - 2c_0 - c_1) \mathbf{e}_0^+ + 2c_0 \mathbf{e}_1^- + \mathbf{r} \right) \quad (3.19)$$

where $c_0 = \frac{2\pi}{n_0}$ and $c_1 = \frac{2\pi}{n_1}$, n_0 and n_1 are the valence at \mathbf{p}_0 and \mathbf{p}_1

Figure 3.25 shows the the face, bunny and big guy models rendered using the three methods. Figure 3.26 shows the reflection lines and normals on the face mesh

for each of the three methods and the Loop scheme. Figure 3.27 shows the ability of this method to handle arbitrary meshes with long and skinny triangles.

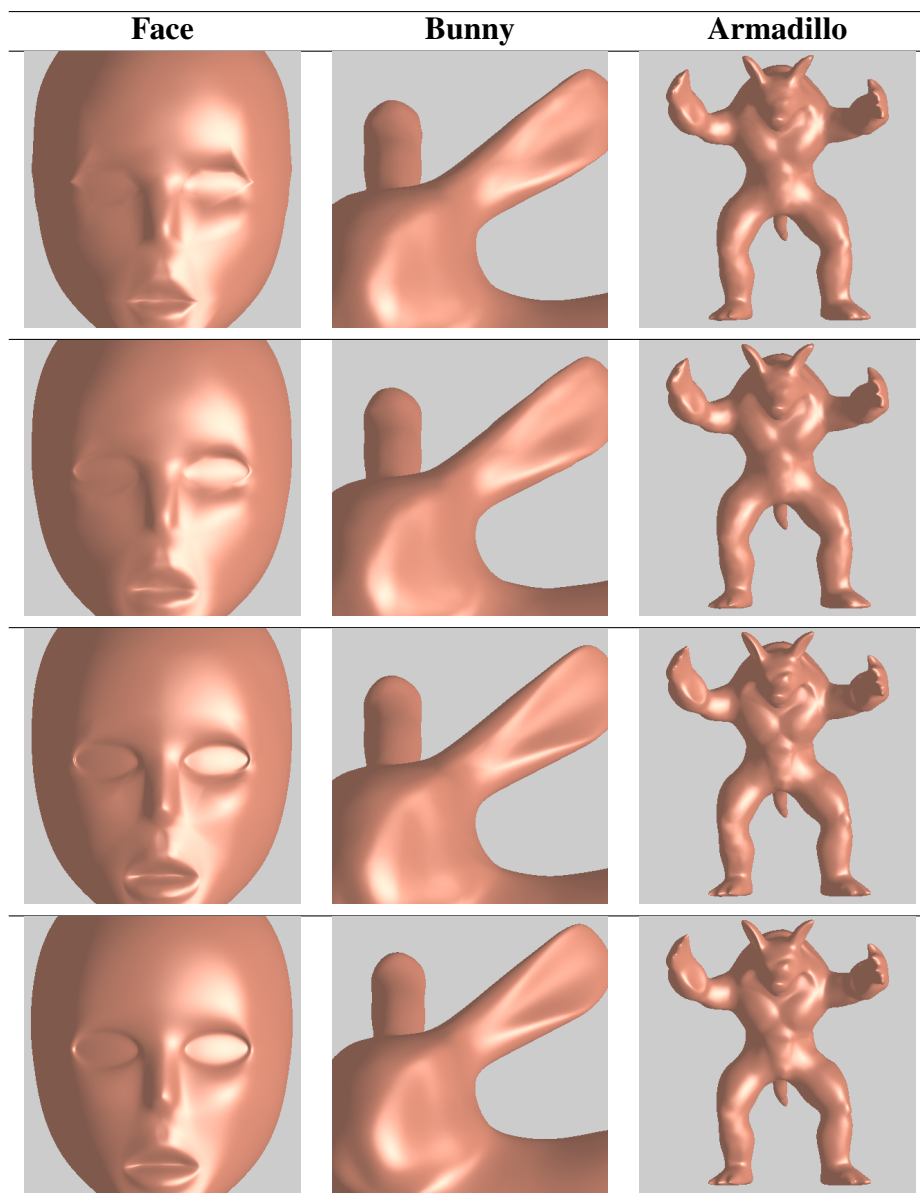


Figure 3.25: The Face, Bunny and Armadillo models rendered using PN, WM, Gregory and Original Loop methods

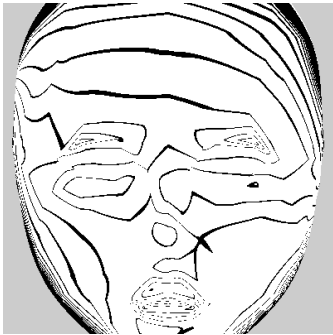
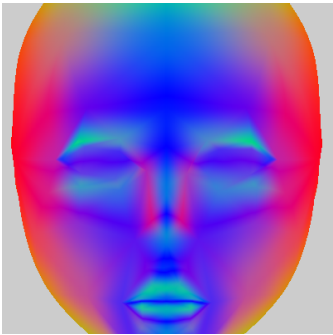
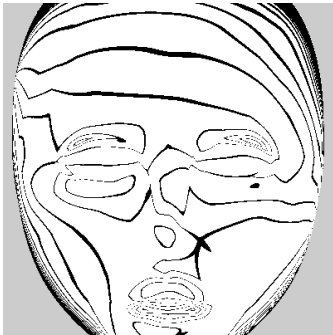
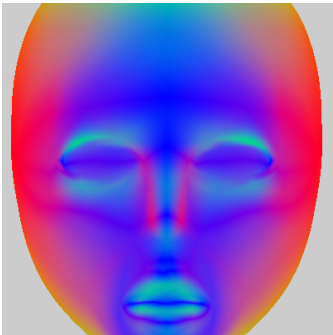
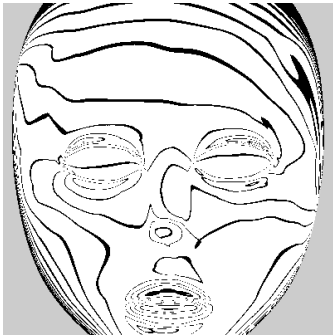
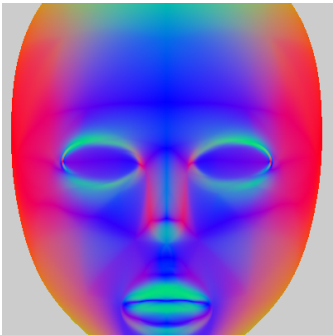

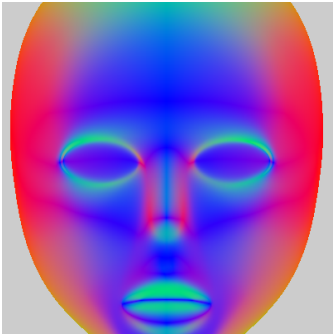
Method	Reflection Lines	Normals
PN		
WM		
Gregory		
Loop		

Figure 3.26: Reflections lines (left) and surface normals (right) for the Face model

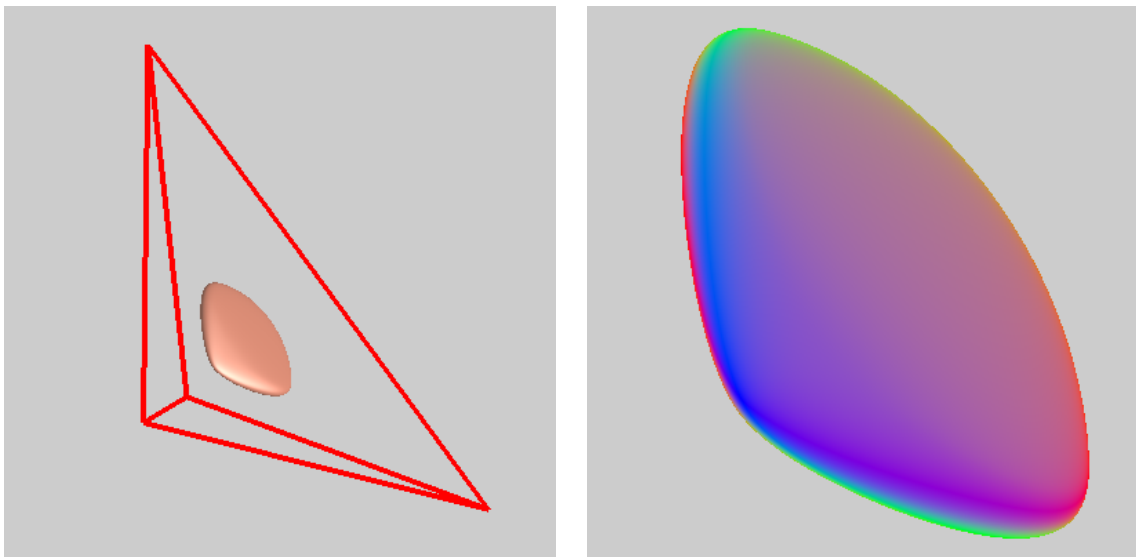


Figure 3.27: Ability to handle long skinny triangles as shown by the original control polygon in red and the corresponding smooth normals

ADAPTIVE TESSELLATION

4.1 Introduction

In this work, we restrict the edge tessellation factors in such a way that only two different tessellation factors can occur in each triangle. For such cases, a tessellation pattern can be used, which is much simpler and more regular than in the regular case. The tessellation code can output the resulting triangles as triangle strips, which makes this approach also suitable for an inside hardware implementation. We will demonstrate this advantage by implementing the pattern in a geometry shader (Section 4.2.3). In Section 4.2.2, we analyze which assignments are possible with two different factors and show that many important cases are contained, i.e., factors based on the distance to the camera eye plane and to the silhouette plane. We show visual results and GPU metrics obtained with our implementation in Section 4.3.

4.2 Semi-Uniform, 2-Different Tessellation

Semi-uniform 2-Different (SD-2) is our proposed pattern for adaptive tessellation where the tessellation factors on the three edges of a triangle are either all same or only two different values occur. As parametric triangular surfaces are composed of patches, it is necessary to do a tessellation of patches. In order to change the tessellation based on various criteria, the computation of suitable tessellation factors and an adaptive tessellation pattern for them is necessary. In the following sections, we describe both these components.

4.2.1 Edge Tessellation Factors based on Vertex/Edge Criteria

For adaptive tessellation of a triangular parametric, normally tessellation factors are computed per vertex or per edge. If computed per vertex then they are propagated to tessellation factors per edge. Given the three edge tessellation factors (f_u, f_v, f_w) , the subdivisions of the three border curves into line segments are given by

$p(0, i/f_u, (f_u - i)/f_u), i = 0 \dots \lfloor f_u \rfloor, p(j/f_v, 0, (f_v - j)/f_v), j = 0 \dots \lfloor f_v \rfloor,$ and

$p(k/f_w, (f_w - k)/f_w, 0)$, $k = 0 \dots \lfloor f_w \rfloor$. A gap-free connection to the mesh neighbors is guaranteed by a tessellation that incorporates these conforming border curves.

An assignment of edge tessellation factors can be based on criteria like vertex distance to the camera eye plane, edge silhouette property, and/or curvature approximations using normal cones Settgest et al. (2004).

4.2.2 Edge Tessellation Factors for the 2-Different Case

For our following tessellation pattern, we need that in the edge tessellation factor triple (f_u^*, f_v^*, f_w^*) only two different values $f_u^* = f_v^*$ and f_w^* occur.

Approximation of an arbitrary tessellation factor assignment (f_u, f_v, f_w) , $f_u \neq f_v$, $f_v \neq f_w$, $f_u \neq f_w$ by a 2-different one (f_u^*, f_v^*, f_w^*) is a non-local problem. Therefore, we avoid the general case and guarantee that the tessellation factor calculation never produces 3-different factors for the edges of a triangle. Then SD-2 can be used for a faster and simpler tessellation.

Let $D : V(M) \rightarrow \mathbb{R}$ be a *level function* on the vertices of the mesh, which means d is strictly monotone increasing on shortest paths $\{v_0 = x \in I, \dots, v_l = y\}$, i.e. $D(v_{i-1}) < D(v_i)$, from a vertex $x \in I$ in the set $I = \{x \in V(M) : \forall y d(x) \leq d(y)\}$ of minimum elements. Given a level function D , it is easy to derive a tessellation factor assignment f^* , which is only 2-different, as follows $f^*(\{s, e\}) := G(\min\{D(s), D(e)\})$ with a normally monotone, scalar function G . Note that the level function is only used to impose an order on the mesh vertices, which is easy to compute based on the vertex coordinates. We give examples of tessellation factor assignments below, which are constructed with the help of a level function as described.

Distance from the camera eye plane. The smallest distance d_p to a plane, for example, the camera eye plane, naturally is a level function, as defined above. A semi-uniform edge tessellation factor assignment (f_u^*, f_v^*, f_w^*) for a triangle then is

$f_{\text{edge}}^* := \mathbf{G}(\min\{\mathbf{D}(s_{\text{edge}}), \mathbf{D}(e_{\text{edge}})\})$ with a linear function

$\mathbf{G}_1(d) := f_{\text{max}} \frac{d_{\text{max}} - d}{d_{\text{max}} - d_{\text{min}}} + f_{\text{min}} \frac{d - d_{\text{min}}}{d_{\text{max}} - d_{\text{min}}}$ or a quadratic function

$\mathbf{G}_2(d) := \frac{f_{\text{max}} - f_{\text{min}}}{(d_{\text{min}} - d_{\text{max}})^2} (d - d_{\text{max}})^2 + f_{\text{min}}$ mapping the scene's depth range $[d_{\text{min}}, d_{\text{max}}]$

to decreasing tessellation factors in the range $[f_{\text{min}}, f_{\text{max}}]$.

Silhouette refinement. Silhouette classification is usually done based on a classification of the vertices into front-facing ($\mathbf{n}(E - \mathbf{v}) \geq 0$) and back-facing ($\mathbf{n}(E - \mathbf{v}) < 0$) using the vertex coordinates \mathbf{v} , vertex normal \mathbf{n} and the camera eye point E . The distance function \mathbf{D}_{silh} to the silhouette plane is a level function as defined above. But the function $\mathbf{n}(E - \mathbf{v})$ is easier to compute by just using the vertices and vertex normals of the mesh. It is not a level function though, but an edge is crossed by the silhouette plane in case the two incident vertices are differently classified. Each triangle can have exactly 0 or 2 such edges. An edge tessellation factor assignment with just two values, a for an edge not crossed by the silhouette, and b for an edge crossed by the silhouette, can be used to refine the silhouette line and present full geometric detail at the silhouette.

The edge tessellation factor assignments based on a level function can be directly computed inside a GPU shader. We show examples of this in Section 4.3. On the contrary, the edge tessellation factors obtained by a curvature approximation in the patch vertices can not be made 2-different for arbitrary meshes easily. Computing an approximation is possible though on the CPU.

4.2.3 Adaptive Tessellation Pattern with 2-Different Factors

In case of triangles with only two different edge tessellation factors $f_u = f_v$ and f_w , it is possible to tessellate in an especially simple way. Our tessellation pattern is composed of a bundle of parallel lines $w = i/f_u := w_i, i = 0, \dots, \lfloor f_u \rfloor$, which are intersected by a second bundle of radial lines from the tip vertex $(0, 0, 1)$ to $((f_w - j)/f_w, j/f_w, 0), j = 0, \dots, \lfloor f_w \rfloor$. The intersections with the parallel line i are in the points $((1 - w_i)(f_w - j)/f_w, (1 - w_i)j/f_w, w_i), j = 0, \dots, \lfloor f_w \rfloor$. This pattern

is very flexible as it works also with fractional factors $f_u = f_v$ and f_w . In the fractional case, the remainders $(f_u - \lfloor f_u \rfloor)/f_u$ and $(f_w - \lfloor f_w \rfloor)/f_w$ can be added as additional segments. In case the subdivision is symmetric to the mid-edge, it can be generated in arbitrary direction. We achieve this by shrinking the first segment and augmenting the last segment by the half fractional remainder $0.5(1/f_u - (f_u - \lfloor f_u \rfloor)/f_u)$ and $0.5(1/f_w - (f_w - \lfloor f_w \rfloor)/f_w)$ respectively. Otherwise, it has to be generated in a unique direction, for example from the nearest to the farthest vertex, which complicates things a lot. Concerning the distribution of lines, it is also possible to place them non-uniformly by a reverse projection according to Munkberg et al. (2008): $u' = \frac{u/z_1}{(1-u-v)/z_0+u/z_1+v/z_2}$, $w' = \frac{w/z_0}{w/z_0+u/z_1+(1-u-w)/z_2}$ where z_0, z_1, z_2 are the vertex depths of the triangle.

Figure 4.1 shows an example of the construction for fractional values $f_u = f_v = 2.5$, $f_w = 2.5$, and for integer factors $f_u = f_v = 3$, $f_w = 3$ and $f_u = f_v = 3$, $f_w = 5$.

It is possible to output all triangles between two adjacent radial lines or two adjacent parallel lines as a triangle strip, which reduces vertex repetition considerably and is beneficial on some hardware architectures. This property becomes a great advantage at the silhouettes where the triangles have edge tessellation factors $(f_u = f_v \gg f_w)$ or $(f_u = f_v \ll f_w)$ and the tessellation can be emitted with a minimum number of strips. For edge tessellation factors $(f_u = f_v, f_w)$, we give the pseudo code for barycentric coordinates $(u, v, w = 1 - u - v)$ on triangle strips along radial lines in Appendix A.

4.3 Results

The method can use any triangular parametric surface, however we have chosen the PN triangles scheme. We compare SD-2 with uniform tessellation of the PN patches as well as the stitching pattern methods described in (Moreton, 2001), (Chung and Kim, 2003) and (Schwarz and Stamminger, 2009). We compare frame rate and

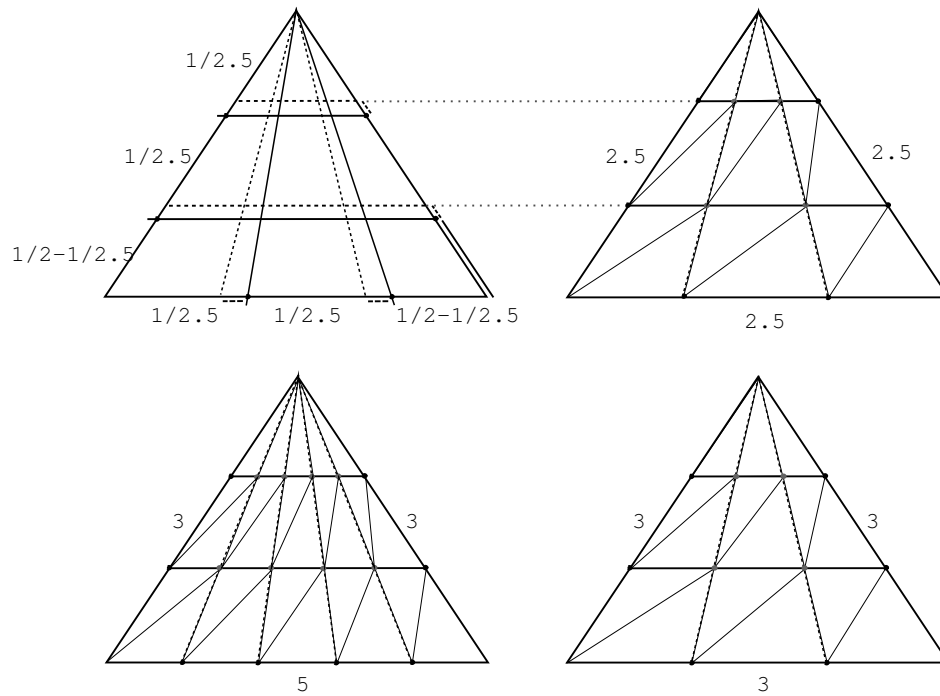


Figure 4.1: Tessellation pattern for 2-different factors, which is composed of a bundle of parallel lines, intersected by a second bundle of radial lines towards the tip vertex. In the top row, for a fractional example: $f_u = f_v = 2.5$, $f_w = 2.5$; in the bottom row, for integer examples: $f_u = f_v = 3$, $f_w = 3$ and $f_u = f_v = 3$, $f_w = 5$.

number of primitives generated on the GPU. For demonstration purposes, we have implemented all methods as geometry shaders. The SD-2 method clearly outperforms the other two and it can be clearly seen that there is a significant boost in frame rate for SD-2 by switching over to triangle strips. See Tables 4.1 and 4.2 for the concrete values on a PC with Windows Vista, 32bit, and NVIDIA 9800 GTX graphics. For surface interrogation, we render a series of reflection lines on the final surface and look at the smoothness of these lines. In general, smoother reflection lines indicate better surface quality. Reflection lines are much smoother for SD-2 in the adaptive region compared to the stitching method, see Figure 4.3.

Table 4.1: Performance for silhouette refinement with max tessellation factor 7 and triangle strips.

Model Name	Base Mesh Triangles	Uniform PN $f = 7$ FPS/Primitives	PN SD-2 FPS/Primitives	PN Stitch FPS/Primitives
Sphere	320	121.0/15680	153.0/12952	122.0/12184
Violin Case	2120	19.5/103880	24.5/83932	19.9/75446
Cow	5804	7.0/284396	9.5/222586	7.4/200383

Table 4.2: Performance for silhouette refinement with max tessellation factor 7 and not using triangle strips.

Model Name	Base Mesh Triangles	Uniform PN $f = 7$ FPS/Primitives	PN SD-2 FPS/Primitives	PN Stitch FPS/Primitives
Sphere	320	60.0/15680	65.0/12952	65.0/12184
Violin Case	2120	7.5/103880	16.0/83932	16.0/75446
Cow	5804	3.2/284396	4.0/222586	4.0/200383

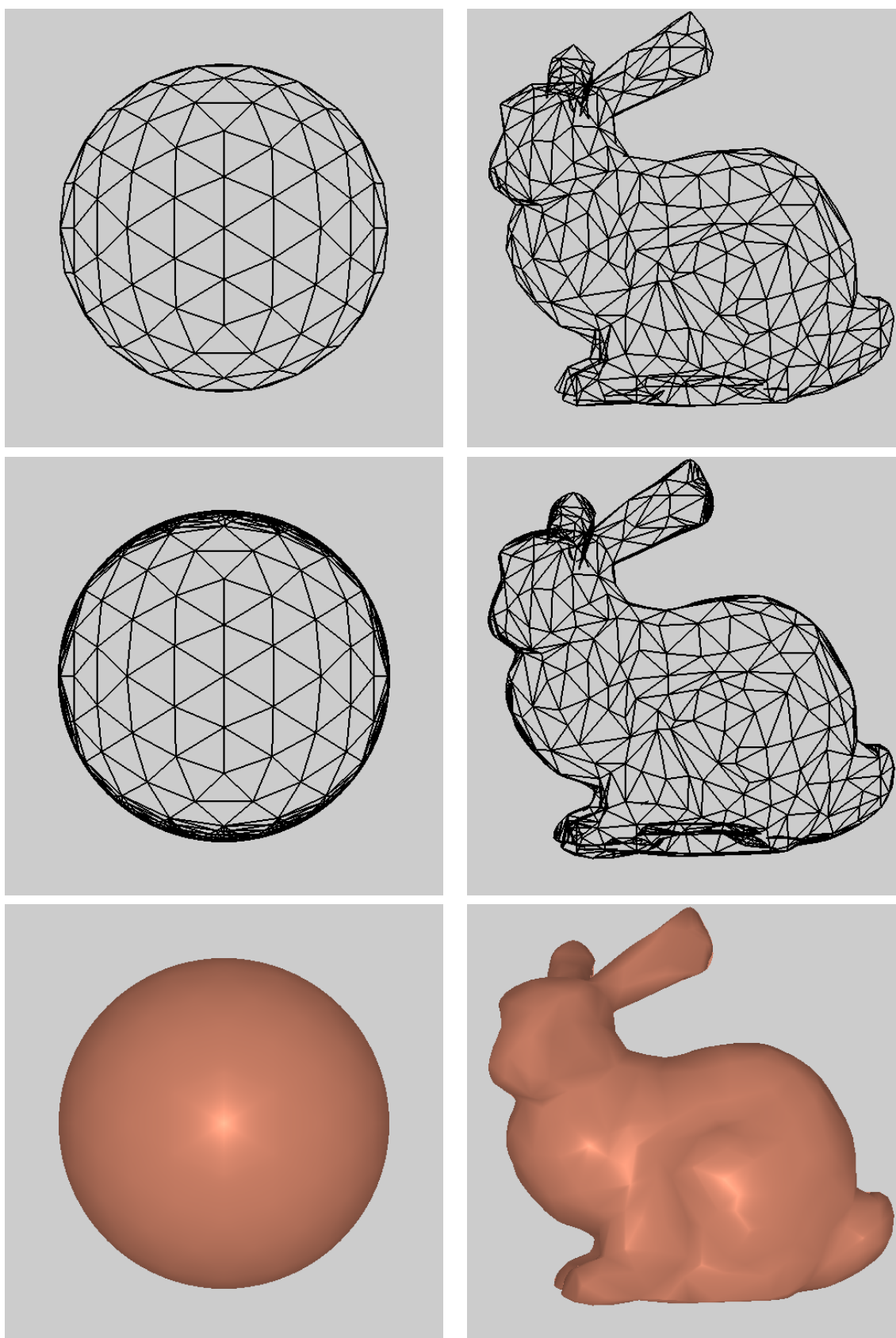


Figure 4.2: Results of SD-2 refinement for improving the silhouette on the Sphere and Bunny models

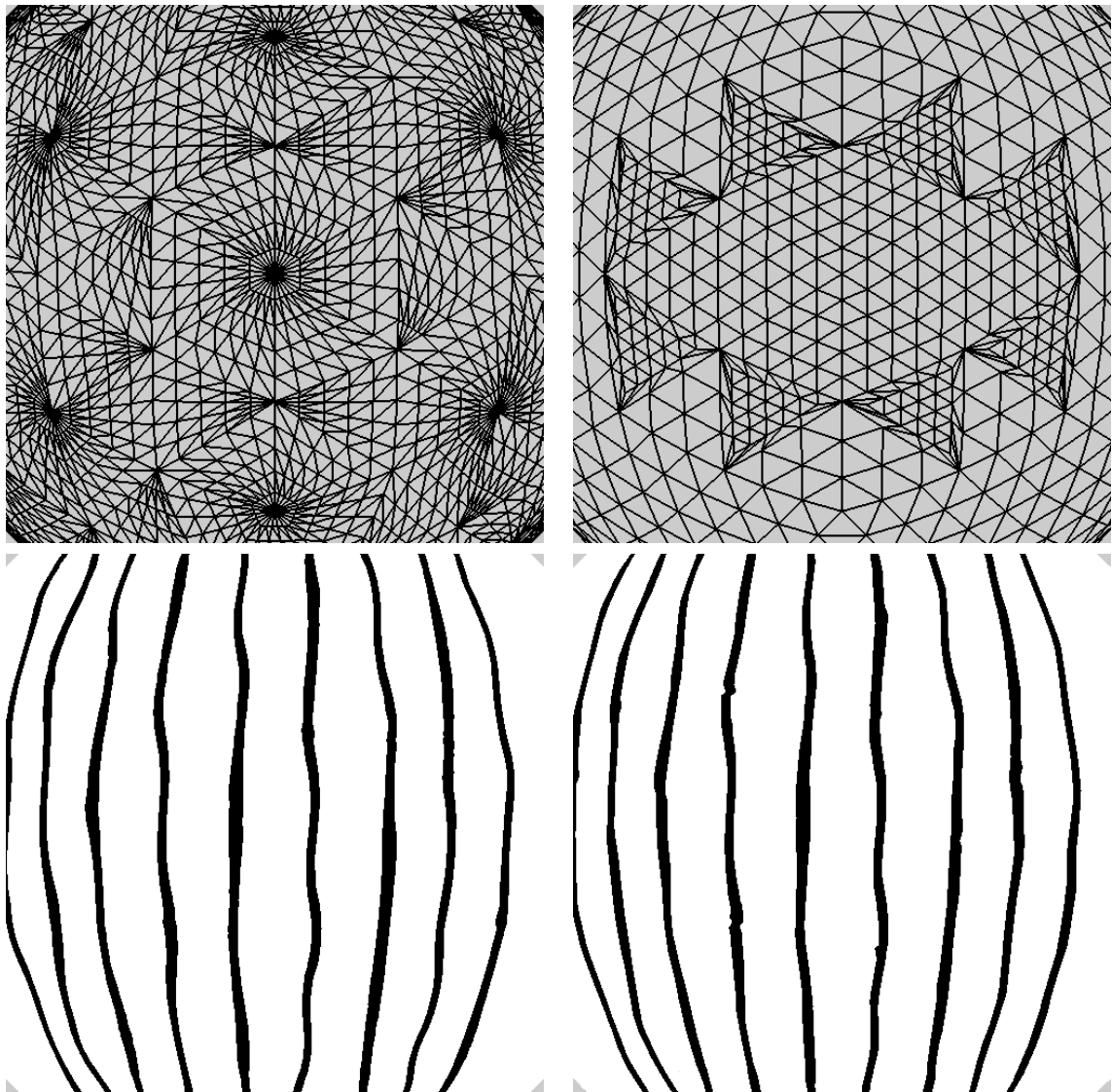


Figure 4.3: Comparison of SD-2 and stitching methods using reflection lines.

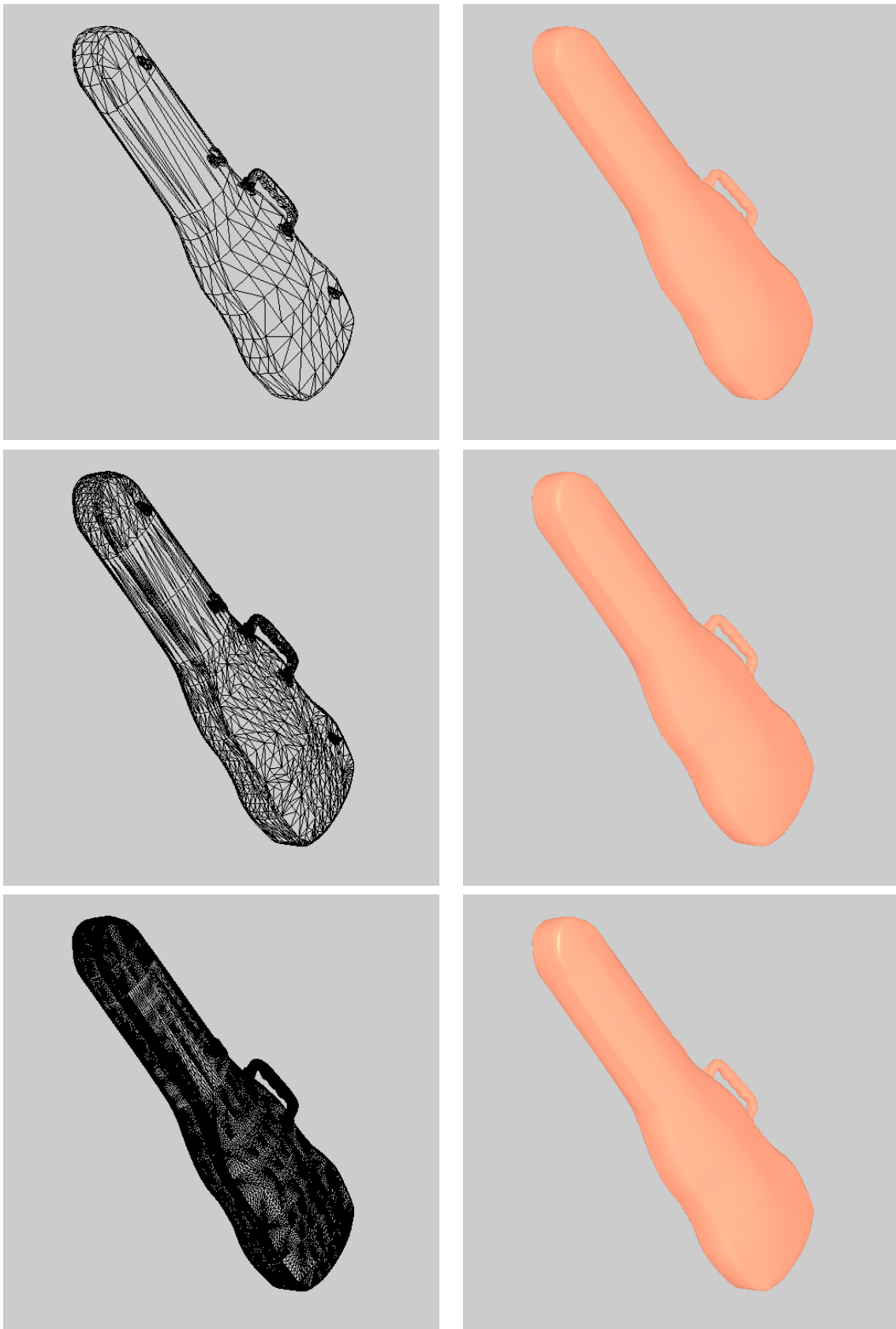


Figure 4.4: Results of SD-2 refinement for continuous LOD based on the distance to the camera eye plane. From left to right, the original mesh, and adaptive tessellations generated by a linear level function with $f_{\max} = 7$, $f_{\min} = 2$, $d_{\max} = 1$ and $d_{\min} = 0$.

Chapter 5

UNIFIED RENDERING FRAMEWORK

5.1 Introduction

In principle, direct evaluation can be performed on subdivision surfaces by implementing Stam's methods Stam (1998a,b) however it fails to achieve a reasonable framerate Loop and Schaefer (2008); Loop et al. (2009). It is therefore necessary to find methods that can take advantage of the tessellator unit and approximately render subdivision surfaces. The main factors that limit the performance of the GPU is the number of control points in the input mesh and number of calculations performed in the control shader to calculate the control points. This is mainly because the GPU is bound by its memory bandwidth and its speed is directly dependent on the number of memory fetches. Our research also found that algorithms that use many instructions to derive the control points in the control shader can severely limit performance even though their initial memory foot print is low. We therefore choose three methods based on these observations. PN method has 6 control points and 13 operations in the control shader, WM method has 6 control points and 70 operations in the control shader and Gregory method has 15 control points and 0 operations in the control shader. We render various objects using the three methods and describe the performance in terms of their visual quality and speed. We find that the speed is also dependent on the number of models in the scene invoking a particular method and therefore the methods need to be carefully chosen in order to optimize the scene. To validate the hypothesis, a visual fidelity test, described in Section 5.3, was performed at THQ Digital Studios, Phoenix, Arizona. The test shows that there is a need for developing an unified rendering framework, that can combine all three methods at run time. Our goal is to provide application developers the information necessary for migrating from polygonal domain to smooth rendering of triangles. By applying the unified rendering framework, described in Section 5.2, developers can perform

automatic level-of-detail(LOD) calculations for their meshes. The number of control points in the input mesh then becomes synonymous with the LOD level. This gives the ability to have a two step system for LOD, one computes the control points of the input mesh and the other computes the edge tessellation factors.

5.2 Unified Rendering Framework

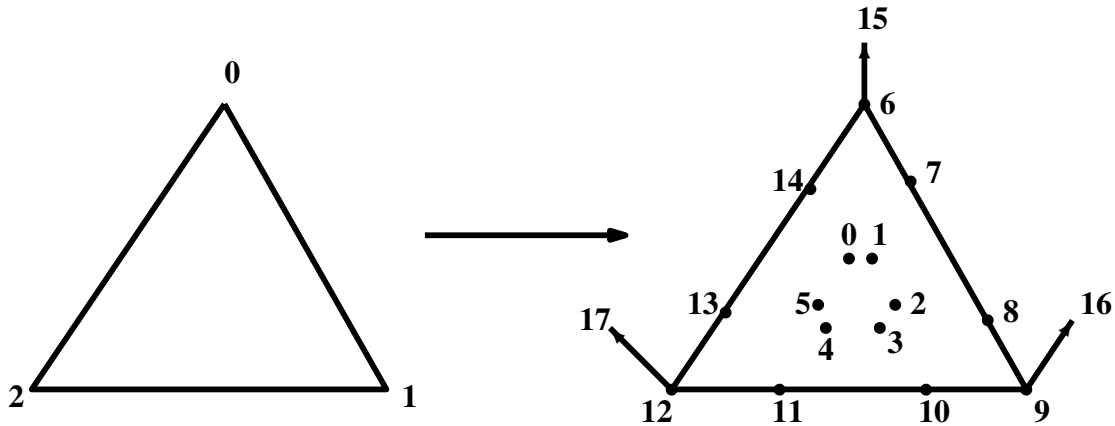


Figure 5.1: Indexing of the 18 control points calculated for every input triangle

We propose a unified rendering framework that involves three stages, preprocessing, CPU switching and rendering. We suggest that during the preprocessing stage all the control point calculations for each triangle in the input mesh be done irrespective of the method chosen. This means that we need to calculate for each triangle, 3 limit points, 3 limit normals, 6 edge points and 6 inner points, resulting in a total of 18 control points. The application can then dynamically pass the appropriate number of control points at run time on a case by case basis. This gives the developer an automatic one-pass algorithm to switch the methods at run time to manage quality vs. performance or connect it into existing LOD structures. In order to manage the control point mesh efficiently we use a half edge mesh structure Weiler (1988) and is shown in Figure 5.2. The half edge splits each edge down its length with each half (blue lines) representing one direction, therefore the two half edges of an edge are in opposite direction. Each half edge references the face it borders, so unlike the edge, the half edge is part of a unique face. It also stores references to the vertex it

originates from, the next half edge in the face that it belongs to and the opposite half edge that it pairs with to form the edge.

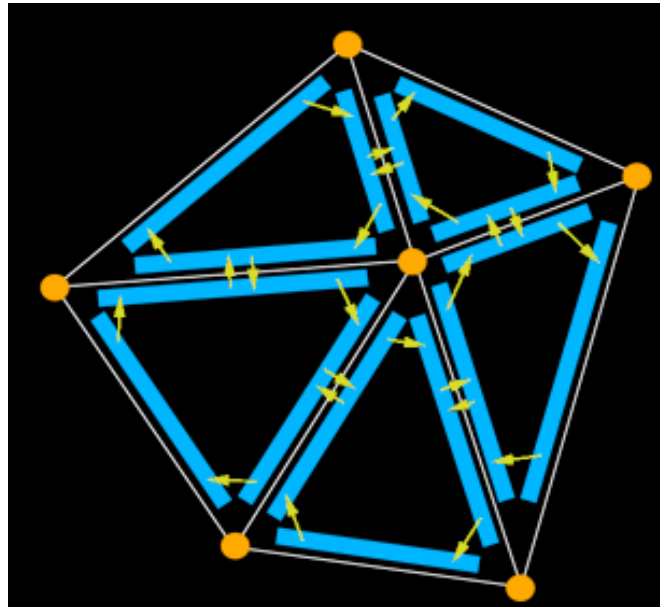


Figure 5.2: The half edge mesh structure, image courtesy of Max McGuire

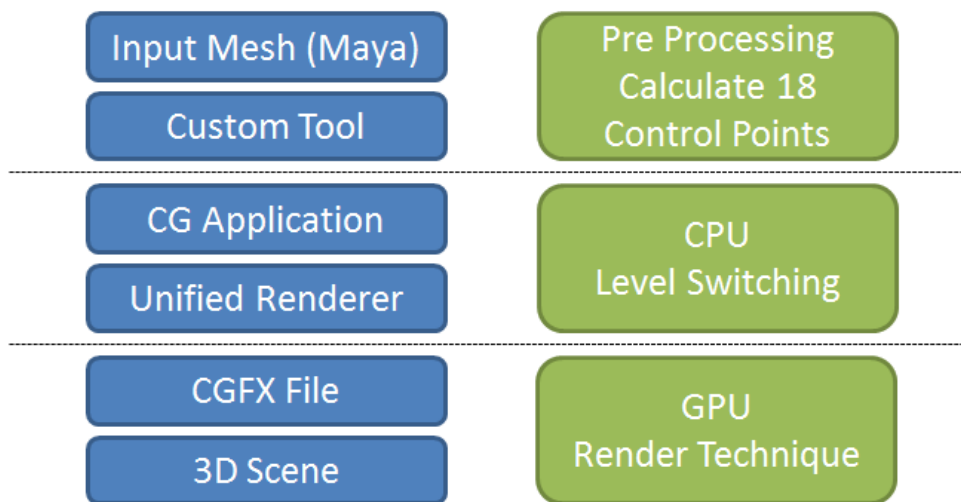


Figure 5.3: Various stages of the unified rendering framework

As shown in Figure 5.3, there are three main stages for the unified rendering framework. In the first stage, control polygon from asset production packages like

Maya is read by a custom patch generation tool and processed into the half edge mesh structure, then the mesh is evaluated for generating the limit points, tangents and normals and the 18 control points are indexed and stored for rendering. Figure 5.1 shows the indexing of these points. In the second stage the application creates custom draw calls for all the three methods and performs CPU based decisions to choose the appropriate method.

5.2.1 Control Shader

The control shader takes the input control mesh based on the method chosen, and performs tessellation calculations to determine each edge tessellation factor. Also depending on the case it can perform other calculations before sending the output to the evaluation shader. For example, in the PN method the 10 control points for the cubic Bézier triangle and the 6 normals for quadratic interpolation are calculated in the control shader. The implementation details are provided in Appendix B.

5.2.2 Evaluation Shader

The Evaluation shader receives the $u, v, w = 1 - u - v$ values from the tessellator and the output from the control shader and calculates the surface point and normal at that parameter set. The PN method evaluates the point using Equation 3.1 and the normal by quadratic interpolation. The WM method evaluates the point by Equation 3.3, and the normal by quadratic interpolation. The Gregory method transforms Equation 3.3 into a quartic Bézier by degree elevating the boundary cubic curves and calculating F_0, F_1, F_2 as described in Section 3.5.4. This allows for simultaneous calculation of the point and normal using the de Casteljau algorithm. Table 5.1 shows the the number of calculations in the control and evaluation shaders for the three methods. The last stage involves creating the effect file using the CGFX file format with the methods implemented as separate techniques. Control and Evaluation shaders need to be written and are provided as CGFX files. Figure 5.4 shows the various steps involved in the rendering the PN method using hardware tessellation. In principle this

process remains the same across all methods with changes happening in the type and size of data fed to the control shader, calculations required for processing this data and the calculations required for evaluating the method in the evaluation shader. The implementation details are provided in Appendix B.

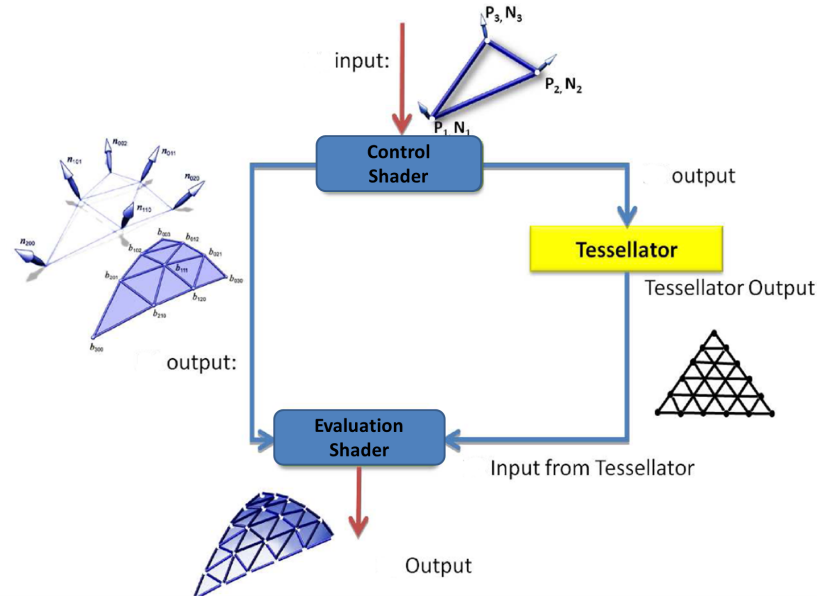


Figure 5.4: Rendering PN-Triangles using hardware tessellation, PN images courtesy of Vlachos et al. (2001)

Table 5.1: Shader properties for the three methods

Method Name	Control Shader	Evaluation Shader	Control Points
PN	13	10	6
WM	70	10	6
Gregory	0	20	15

5.3 Results

5.3.1 Visual Fidelity Test

A team of 15 video game artists, graphics programmers and designers ranked the methods applied to the *bunny*, *monster frog* and *face* models on scale of 1 to 10 with 10 being the best. The camera was up-close and they were also shown the actual Loop rendering for comparison. In the next step they were asked to again rank these models at mid-distance, with camera moved away and the models occupying only 50 percent

of the screen space, and at far-distance, with camera moved further away and the models occupying less than 25 percent of screen space. Finally they were asked to rank at close range by considering the frame rate for each of the renderings along with the visual quality. Figure 5.6, left, shows that for all models the Gregory is visually pleasing over WM and PN; however this difference diminishes quite rapidly at mid and far distances (Figure 5.6, right). Figure 5.5 shows that different models rate differently at near, mid and far distances. It is not necessary that Gregory should be the chosen method at all times and when performance numbers are known it is less likely to be preferred for most models. Looking closely at Figure 5.5 we can also see that except the face model, Gregory is not the preferred method when frame rate is taken into consideration. This proves that there is a need for having multiple smooth rendering methods and it would greatly benefit if they can be switched at run time. The unified rendering system provides this ability for developers to either use it during run time or inside export tools that perform asset conversion during production time.

5.3.2 Performance Results

Table 5.2: Frame rate comparison for the three methods

Model Name	Base Mesh Tris/Verts	Tess Level/ Object Count	PN FPS	WM FPS	Gregory FPS
Face	102/200	5/1	1850	1180	1780
Bunny	502/1000	5/1	1297	822	1190
Big Guy	1754/2900	5/1	940	635	870
Face	102/200	5/50	479	379	266
Bunny	502/1000	5/50	63	52	32
Big Guy	1754/2900	5/50	53	39	22

Table 5.3: Geometric error reported by Metro for the three methods

Method Name	Max error	mean error	RMS error
PN	0.5618	0.019	0.030
WM	0.3913	0.013	0.022
Gregory	0.1746	0.009	0.013

In this section we measure the speed and quality of the surface for the three

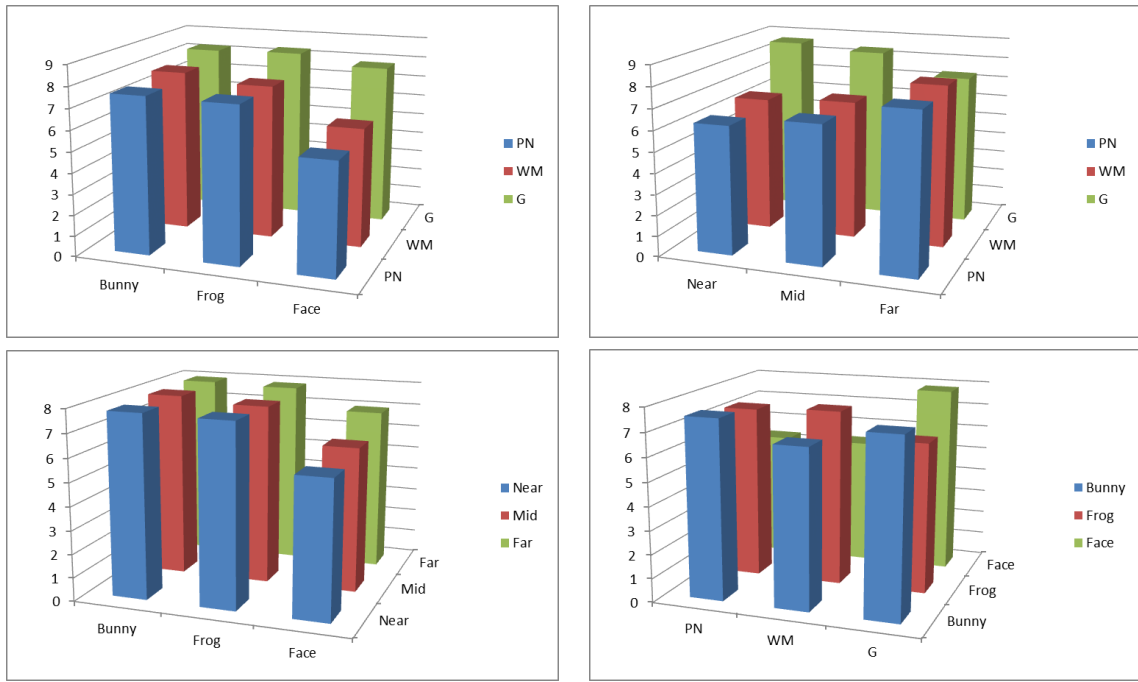


Figure 5.5: Results of a visual fidelity test performed on the bunny, monster frog and face models, top left: compares PN, WM and Gregory for the three models, top right: compares the methods at three distances, bottom left: shows how various models compare at different distances and bottom right: comparison when performance numbers are known

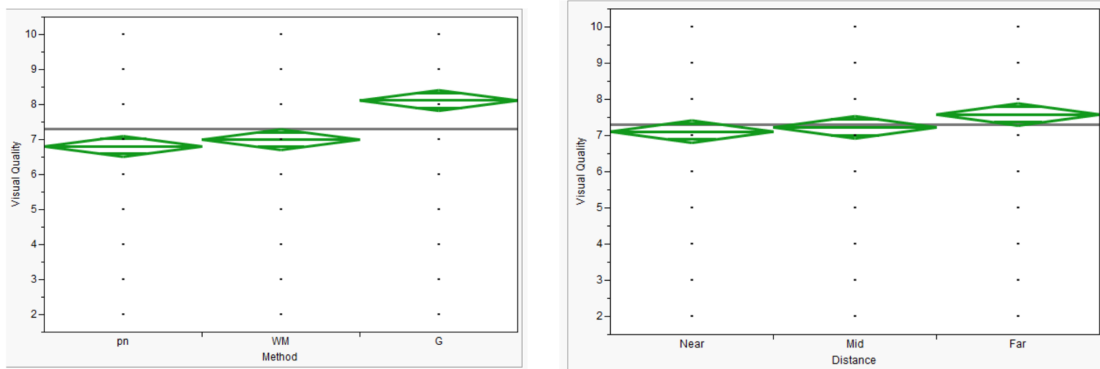


Figure 5.6: Left: Average visual quality values for the three methods and Right: Average values obtained for all methods at near, mid and far distances

methods described in this paper. With all conditions being the same we test the frame rate at various tessellation levels. We also render the reflection lines for each method to test the quality of the surface. Our tests were performed on PC with Windows 7 32bit and NVIDIA Quadro5000 graphics card. Table 5.2, shows the frame rate

comparisons for the face, bunny and big guy models at tessellation level 5 and also compares the numbers by changing the number of models drawn from 1 to 50. It is seen that PN performs best under all situations and Gregory outperforms the WM method only when few models use it. This confirms that performance depends on the number of operations in the hull shader and the number of control points that need to be stored in video memory for a base mesh triangle. Figure 5.7 shows the monster frog model rendered using the three methods, PN, WM, Gregory and the original Loop method. Table 5.3 shows the geometric error calculated by Metro Cignoni et al. (1998) by comparing the triangle meshes generated by Loop subdivision to the ones generated by the three methods. Figure 5.8 shows this error mapped to vertex colors and we can observe that except at extreme extraordinary points (valence greater than 7) the error produced by the Gregory method is minimal. To compensate for this limitation the original control mesh would need to avoid having such points at model time.

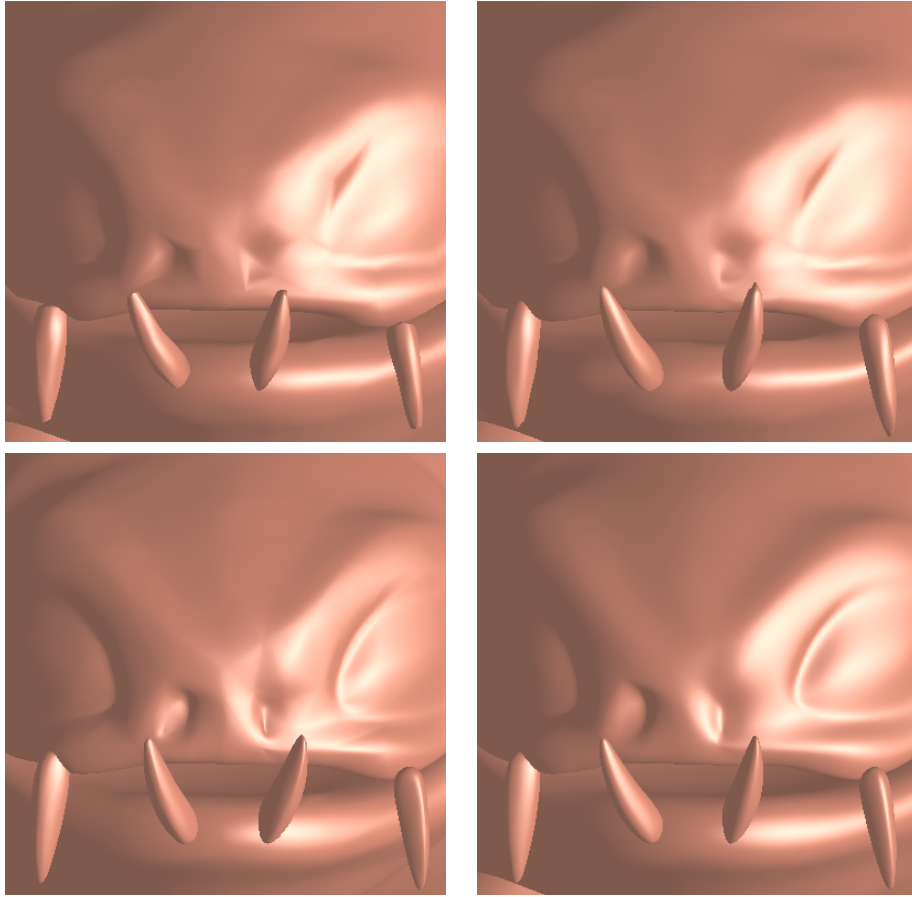


Figure 5.7: Monster frog rendered using PN (top-left), WM (top-right), Gregory (bottom-left) and Original Loop (bottom-right) methods

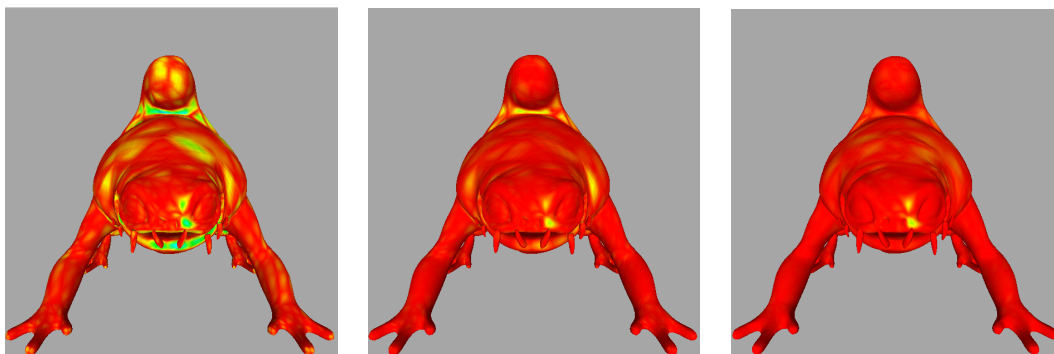


Figure 5.8: Geometric approximation error calculated by Metro for PN, WM and Gregory methods, the error is represented as a color ramp between red and blue with red representing no error, the error range is 0.0 to 0.57 with the bounding box diagonal measuring 41.25 units.

Chapter 6

CONCLUSION AND FUTURE DIRECTIONS

6.1 Summary

In this thesis, we have described, SD-2, a tessellation pattern for fractional edge tessellation factors with only two different values per triangle. Under continuous changes of the tessellation factors, the pattern fulfills all the requirements on the continuity of tessellation changes. This is especially important for the sampling of geometry and applied texture/displacement maps during animations. The scheme is especially simple to implement, and it is suitable for triangle output in the form of triangle strips. In terms of adaptivity, it can cover the most important cases, where the edge tessellation factors are derived from a level function on the mesh vertices. Then, the edges can be directed and 2-different edge tessellation factors can be assigned based on the minimum (or maximum) vertex on each edge. We have shown results in terms of speed and quality with an implementation of the pattern and the edge tessellation factor assignment in the geometry shader of the GPU. This shows that the approach is also suitable for a future hardware implementation.

We have also presented three methods for approximating Loop subdivision surfaces using tessellation enabled hardware. Developers can unify the asset production pipeline and provide automatic switching between these methods at runtime. The maximum number of control points required per patch is 18, the PN and WM methods use 6 while the Gregory uses 15.

6.2 Conclusions and Future Work

SD-2 has potential to be implemented in hardware or be used via instanced tessellation in cases where hardware tessellation is not available. SD-2 works for any parametric surface rendering method and in future we would like to compare its performance with other smooth rendering methods. We would also like to further refine the pattern so that it produces even tessellation triangles around the 1-ring of

each vertex and minimizes skinny triangles.

Our observations with the unified rendering setup lead to the following conclusions, the Gregory method is best suited for characters and fluid assets that incorporate complex animations such as human characters, the WM method is best suited for dynamic objects with simple animations such as weapons, vehicles and breakable objects, while the PN method works best for static objects like trees, environments and terrain. This observation stems from the visual fidelity test described in Section 5.3 and the information in Table 5.2 based on the performance of rendering these methods with single and multiple objects as well as at lower and higher tessellations. Even though the unified rendering setup is shown for triangles by approximating the Loop method, it can similarly be applied for quads using the Catmull-Clark subdivision method. In future work, we plan to improve the unified rendering setup to include meshes consisting of arbitrary polygons and not just triangles. We also plan to develop newer algorithms that approximate subdivision surfaces while maintaining smoothness and performance metrics.

REFERENCES

- C. Dyken, M. Reimers, J. Seland, Semi-Uniform Adaptive Patch Tessellation, in: *Computer Graphics Forum*, vol. 28, Wiley Online Library, ISSN 1467-8659, 2255–2263, 2009.
- C. Loop, S. Schaefer, T. Ni, I. Castaño, Approximating subdivision surfaces with Gregory patches for hardware tessellation, *ACM Transactions on Graphics (TOG)* 28 (5) (2009) 1–9, ISSN 0730-0301.
- A. Sfarti, B. A. Barsky, T. Kosloff, E. Pasztor, A. Kozłowski, E. Roman, A. Perelman, Direct Real Time Tessellation of Parametric Spline Surfaces, in: *3IA Conference, Invited Lecture*, http://3ia.teiath.gr/3ia_previous_conferences_cds/2006, 2006.
- D. Schweitzer, E. S. Cobb, Scanline rendering of parametric surfaces, *SIGGRAPH Comput. Graph.* 16 (3) (1982) 265–271, ISSN 0097-8930, doi:[binfo{doi}{http://doi.acm.org/10.1145/965145.801289}](http://doi.acm.org/10.1145/965145.801289).
- M. Bóo, M. Amor, M. Doggett, J. Hirche, W. Strasser, Hardware support for adaptive subdivision surface rendering, in: *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ACM, New York, NY, USA, ISBN 1-58113-407-X, 33–40, doi:[binfo{doi}{http://doi.acm.org/10.1145/383507.383522}](http://doi.acm.org/10.1145/383507.383522), 2001.
- V. Settgast, K. Müller, C. Fünzig, D. Fellner, Adaptive Tessellation of Subdivision Surfaces, *Computers & Graphics* 28 (2004) 73–78.
- H. Moreton, Watertight tessellation using forward differencing, in: *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ACM, New York, NY, USA, ISBN 1-58113-407-X, 25–32, doi:[binfo{doi}{http://doi.acm.org/10.1145/383507.383520}](http://doi.acm.org/10.1145/383507.383520), 2001.
- K. Chung, L. Kim, Adaptive Tessellation of PN Triangle with Modified Bresenham Algorithm, in: *SOC Design Conference*, 102–113, 2003.
- M. Schwarz, M. Stamminger, Fast GPU-based Adaptive Tessellation with CUDA, *Comput. Graph. Forum* 28 (2) (2009) 365–374.
- J. Munkberg, J. Hasselgren, T. Akenine-Möller, Non-uniform fractional tessellation, in: *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, ISBN 978-3-905674-09-5, 41–45, 2008.
- K. Gee, Introduction to the Direct3D 11 graphics pipeline, in: *nvision '08: The World of Visual Computing*, Microsoft Corporation, ISBN 1-59593-364-6, 1–55, 2008.
- E. Catmull, J. Clark, Recursively generated B-spline surfaces on arbitrary topological meshes, *Computer-aided design* 10 (6) (1978a) 350–355, ISSN 0010-4485.
- D. Doo, M. Sabin, Behaviour of recursive division surfaces near extraordinary points, *Computer-Aided Design* 10 (6) (1978) 356–360, ISSN 0010-4485.

- C. Loop, Smooth Subdivision Surfaces Based on Triangles, Master's Thesis, University of Utah 1 (1987) 1–74, URL <http://research.microsoft.com/~cloop/>.
- N. Dyn, D. Levine, J. Gregory, A butterfly subdivision scheme for surface interpolation with tension control, ACM transactions on Graphics (TOG) 9 (2) (1990) 160–169, ISSN 0730-0301.
- D. Zorin, P. Schröder, W. Sweldens, Interpolating subdivision for meshes with arbitrary topology, in: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, ACM, ISBN 0897917464, 189–192, 1996.
- L. Kobbelt, Interpolatory subdivision on open quadrilateral nets with arbitrary topology, in: Computer Graphics Forum, vol. 15, Wiley Online Library, ISSN 1467-8659, 409–420, 1996.
- J. Bolz, P. Schröder, Rapid evaluation of Catmull-Clark subdivision surfaces, in: Proceedings of the seventh international conference on 3D Web technology, ACM, ISBN 1581134681, 11–17, 2002.
- J. Bolz, P. Schröder, Evaluation of subdivision surfaces on programmable graphics hardware, preprint URL <http://www.multires.caltech.edu/pubs/GPUSubD.pdf>.
- L. Shiue, I. Jones, J. Peters, A realtime GPU subdivision kernel, ACM Transactions on Graphics (TOG) 24 (3) (2005) 1010–1015, ISSN 0730-0301.
- K. Pulli, M. Segal, Fast rendering of subdivision surfaces, in: Proceedings of the eurographics workshop on Rendering techniques, vol. 96, Citeseer, 61–70, 1996.
- S. Bischoff, L. Kobbelt, H. Seidel, Towards hardware implementation of loop subdivision, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, Citeseer, 41–50, 2000.
- J. Stam, Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values, in: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, ACM, ISBN 0897919998, 395–404, 1998a.
- J. Stam, Evaluation of loop subdivision surfaces, in: SIGGRAPH'98 CDROM Proceedings, Citeseer, 1998b.
- A. Vlachos, J. Peters, C. Boyd, J. Mitchell, Curved PN triangles, in: Proceedings of the 2001 symposium on Interactive 3D graphics, ACM, ISBN 1581132921, 159–166, 2001.
- T. Boubekeur, C. Schlick, Approximation of subdivision surfaces for interactive applications, ACM Siggraph Sketch Program .
- T. Boubekeur, C. Schlick, QAS: Real-Time Quadratic Approximation of Subdivision Surfaces, in: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications, IEEE Computer Society, ISBN 0769530095, 453–456, 2007b.

- C. Fünfzig, K. Müller, D. Hansford, G. Farin, PNG1 triangles for tangent plane continuous surfaces on the GPU, in: GI '08: Proceedings of graphics interface 2008, Canadian Information Processing Society, Toronto, Canada, ISBN 978-1-56881-423-0, 219–226, 2008.
- D. Walton, D. Meek, A triangular G1 patch from boundary curves, *Computer-Aided Design* 28 (2) (1996) 113–123, ISSN 0010-4485.
- C. Loop, S. Schaefer, Approximating Catmull-Clark subdivision surfaces with bicubic patches, *ACM Transactions on Graphics (TOG)* 27 (1) (2008) 1–11, ISSN 0730-0301.
- D. Kovacs, J. Mitchell, S. Drone, D. Zorin, Real-time creased approximate subdivision surfaces, in: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, 155–160, 2009.
- G. Li, C. Ren, J. Zhang, W. Ma, Approximation of Loop Subdivision Surfaces for Fast Rendering, *IEEE Transactions on Visualization and Computer Graphics* ISSN 1077-2626.
- D. Luebke, G. Humphreys, How gpus work, *Computer* 40 (2) (2007) 96–100, ISSN 0018-9162.
- J. Kautz, Hardware lighting and shading: A survey, in: *Computer Graphics Forum*, vol. 23, Wiley Online Library, ISSN 0167-7055, 85–112, 2004.
- T. Akenine-Moller, E. Haines, *Real-time rendering*, AK, ISBN 1568811829, 2002.
- P. Kry, D. James, D. Pai, Eigenskin: real time large deformation character skinning in hardware, in: *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, ISBN 1581135734, 153–159, 2002.
- I. Baran, J. Popović, Automatic rigging and animation of 3d characters, in: *ACM SIGGRAPH 2007 papers*, ACM, 72, 2007.
- F. Policarpo, M. Oliveira, J. Comba, Real-time relief mapping on arbitrary polygonal surfaces, in: *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM, ISBN 1595930132, 155–162, 2005.
- S. Spencer, *ZBrush Character Creation: Advanced Digital Sculpting*, Sybex, ISBN 047024996X, 2008.
- T. Boubekur, C. Schlick, Generic mesh refinement on GPU, in: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, ISBN 1595930868, 99–104, 2005.
- T. Boubekur, C. Schlick, A flexible kernel for adaptive mesh refinement on GPU, in: *Computer Graphics Forum*, vol. 27, Wiley Online Library, ISSN 1467-8659, 102–113, 2008.

- R. Arnaud, The Game Asset Pipeline, *Game Engine Gems: Volume One* (2010) 11.
- R. Fernando, M. Kilgard, *The Cg Tutorial: The definitive guide to programmable real-time graphics*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, ISBN 0321194969, 2003.
- J. Peters, Geometric Continuity, in: *Handbook of Computer Aided Geometric Design*, Elsevier, 193–229, 2002.
- B. Piper, Visually smooth interpolation with triangular Bézier patches, *Geometric modeling: algorithms and new trends* (1987) 211–233.
- H. Hagen, S. Hahmann, T. Schreiber, Y. Nakajima, B. Wordenweber, P. Hollemann-Grundstedt, Surface interrogation algorithms, *Computer Graphics and Applications*, IEEE 12 (5) (1992) 53–60, ISSN 0272-1716.
- R. Klass, Correction of local surface irregularities using reflection lines, *Computer-Aided Design* 12 (2) (1980) 73–77, ISSN 0010-4485.
- E. Lengyel, *Mathematics for 3D game programming and computer graphics*, Cengage Learning, ISBN 1584502770, 2004.
- G. Farin, *Curves and Surfaces for Computer-Aided Geometric Design — A Practical Guide*, The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling, Morgan Kaufmann Publishers (Academic Press), 5th edn., 499 pages., 2002.
- J. Gregory, Smooth interpolation without twist constraints, *Computer Aided Geometric Design* (1974) 71–87.
- H. Chiyokura, T. Takamura, K. Konno, T. Harada, G1 surface interpolation over irregular meshes with rational curves, *NURBS for Curve and Surface Design* (1990) 15–34.
- E. Catmull, J. Clark, Recursively generated B-spline surfaces on arbitrary topological meshes, *Computer-Aided Design* 10 (1978b) 350–355.
- A. Amresh, G. Farin, A. Razdan, Adaptive subdivision schemes for triangular meshes, *Hierarchical and Geometric Methods in Scientific Visualization* .
- H. Seidel, Polar forms and triangular B-spline surfaces, *Computing in Euclidean Geometry* (1992) 235–286.
- K. Weiler, The radial edge structure: a topological representation for non-manifold geometric boundary modeling, *Geometric modeling for CAD applications* (1988) 3–36.
- P. Cignoni, C. Rocchini, R. Scopigno, Metro: measuring error on simplified surfaces, in: *Computer Graphics Forum*, vol. 17, Wiley Online Library, ISSN 1467-8659, 167–174, 1998.

Appendix A
SD-2 PSEUDO CODE

```

// reorder control points/normals (b000/n000, b030/n030, b003/n003)
// so that fu=fv on u,v isolines
void calcUVValues(float diff, float same) //fu=fv=same, fw=diff
{
float line; // counter that traverses along the diff edge
float line1; // counter that traverses along the same edge

// remainder for same fractional
float incS = 0.5*(1.0/same - (same - floor(same))/same);
// remainder for diff fractional
float incD = 0.5*(1.0/diff - (diff - floor(diff))/diff);

vec2 diff1, diff2; // pair of u,v values on the diff edge
float u, v;
float par; // keeps track of the current location on the same edge

for (line=0; line < diff; ++line)
{
diff1.u = (line -incD)/diff;
if (diff1.u < 0.0)
    diff1.u = 0.0;
diff1.v = 1.0 - diff1.u;
diff2.u = (line+1.0 -incD)/diff;
if (diff2.u > 1.0)
    diff2.u = 1.0;
diff2.v = 1.0 - diff2.u;

u = 0.0;
v = 0.0;
//Use PN/PNG1 evaluation code to calculate the point and the normal
//In GLSL use EmitVertex()
for (line1=0; line1<same; ++line1)
{
/* evaluate the point on the second radial line */
par = (line1+1.0 -incS)/same;
if (par > 1.0)
par = 1.0;
v = diff2.y*par;
u = diff2.x*par;
//Use PN/PNG1 evaluation code to calculate the point and the normal
//In GLSL use EmitVertex()

/* evaluate the point on the first radial line */
v = diff1.y*par;
u = diff1.x*par;
//Use PN/PNG1 evaluation code to calculate the point and the normal
//In GLSL use EmitVertex()
}
//Finish the triangle strip
//In GLSL call EndPrimitive()
}
}

```

Figure A.1: Pseudo code used for generating the barycentric coordinates in SD-2 tessellation

Appendix B

CONTROL, EVALUATION AND FRAGMENT SHADER IMPLEMENTATIONS

```

float4 fragmentReflection
(
    float4 position : TEXCOORD0, //screenspace position
    float3 normal    : TEXCOORD1, //screenspace normal
    uniform float4x4 modelView, // modelview matrix
    uniform float4x4 projection // projection matrix
) : flat COLOR // fragment color to output
{

    float3 normal_ = normalize(normal);
    float3 position_ = position.xyz;
    float4 color;

    int iter; //index to the number of light lines
    for(iter=0; iter<100; iter++)
    {
        float3 lightPos = float3(-100.,20.,-10.);
        lightPos = mul((float3x3)projection,lightPos);
        lightPos = mul((float3x3)modelView,lightPos); // light position transform to screen space
        lightPos.x += float(1*iter);
        float3 lightDir = normalize(lightPos); // light direction vector

        float t = (dot(position_,lightDir) - dot(lightPos,lightDir));
        // parameter value t on the light line
        float s = -1.0*(dot(normal_,lightDir));

        float3 r = normalize(lightPos + t*lightDir - position_); // projected look vector
        float3 q = normalize(normal_ + s*lightDir); // projected normal

        float angle = (dot(r,q)); // if angle between the two vectors is ~= 0 then color black else white
        if(angle > 0.9999)
        {
            color = float4(0.,0.,0.,1.);
            break;
        }
        else
        {
            color = float4(1.,1.,1.,1.);
        }

    }

    return color;
}

```

Figure B.1: Fragment shader implementation of reflection lines

```

gp5tcp PATCH_15
void main_gt
(
    in_ATTRIBArray<float3> position : POSITION,

    out accPatchData pd      : ATTR3, // 15 gregory control points
    out float  oEdgeTess[3]  : EDGETESS, // tessellation factors - outer
    out float  oInnerTess    : INNERTESS, // tessellation factors - inner

    uniform float2  innerTess, // CPU input
    uniform float4  outerTess // CPU input
)
{
    oEdgeTess[0] = outerTess.x;
    oEdgeTess[1] = outerTess.y;
    oEdgeTess[2] = outerTess.z;

    oInnerTess = innerTess.x;

    // copy the preprocessed input data over
    pd.p0 = position[0];
    pd.p1 = position[1];
    pd.p2 = position[2];
    pd.p3 = position[3];
    pd.p4 = position[4];
    pd.p5 = position[5];
    pd.p6 = position[6];
    pd.p7 = position[7];
    pd.p8 = position[8];
    pd.p9 = position[9];
    pd.p10 = position[10];
    pd.p11 = position[11];
    pd.p12 = position[12];
    pd.p13 = position[13];
    pd.p14 = position[14];
}

```

Figure B.2: Control shader for the Gregory method

```

gp5step PATCH_15 TRIANGLES ORDER_CCW
void main_gt
(
  in float2 uv          : UV, // u,v parameters from tessellator
  in accPatchData pd   : ATTR3, // control point input from control shader
  out float4 oPosition : POSITION,
  out float3 oNormal   : TEXCOORD1,
  out float2 oUV       : TEXCOORD2,
  out float3 oGradU    : TEXCOORD3,
  out float3 oGradV    : TEXCOORD4,
  uniform float4x4 projection, // projection matrix
  uniform float4x4 modelview  // model view matrix
)
{
  //Transformation from gregory to quartic control points
  //
  //
  //          6
  //          14 0|1 7      ==>          0
  //          13 5/4 3\2 8      2 1 5
  //          12 11 10 9      3 7 6 9
  //                                4 8 11 13 14

  //          11 10
  //          13/14 12
  // 2 4/3 9\8 6
  // 0 1 7 5

```

Figure B.3: Evaluation shader for the Gregory method

```

float3 F0, F1, F2; //inner quartic points
float3 pos, nor;
float u,v,w;
u=uv.x; v=uv.y; w=1-u-v;

float d0 = ( v*(1-u) + u*(1-v) ); if ( (d0<0.001) ) d0 = 1;
float d1 = ( w*(1-v) + v*(1-w) ); if ( (d1<0.001) ) d1 = 1;
float d2 = ( u*(1-w) + w*(1-u) ); if ( (d2<0.001) ) d2 = 1;

F0 = (v*(1-u)*pd.p1 +u*(1-v)*pd.p0)/d0;
F1 = (w*(1-v)*pd.p5 +v*(1-w)*pd.p4)/d1;
F2 = (u*(1-w)*pd.p3 +w*(1-u)*pd.p2)/d2;

float3 q[15];
float3 p[15];
q[0]= pd.p6; q[1]=(pd.p6+3*pd.p14)/4; q[2]=(pd.p14+pd.p13)/2; q[3]=(3*pd.p13+pd.p12)/4;
q[4]= pd.p12; q[5]=(3*pd.p7+pd.p6)/4; q[6]=F0; q[7]=F1; q[8]=(pd.p12+3*pd.p11)/4;
q[9]=(pd.p7+pd.p8)/2; q[10]=F2; q[11]=(pd.p11+pd.p10)/2; q[12]=(pd.p9+3*pd.p8)/4;
q[13]=(pd.p9+3*pd.p10)/4; q[14]= pd.p9;

// perform de Cateljau algorithm
uint j, k=0;
float s=w; float t= u; w=v;
for (j=0; j<4; j++) {
    p[k++]=s*q[j]+t*q[j+1]+w*q[j+5];
}
for (j=5; j<8; j++) {
    p[k++]=s*q[j]+t*q[j+1]+w*q[j+4];
}
for (j=9; j<11; j++) {
    p[k++]=s*q[j]+t*q[j+1]+w*q[j+3];
}
p[9]=s*q[12]+t*q[13]+w*q[14];

k=0;
for (j=0; j<3; j++) {
    q[k++]=s*p[j]+t*p[j+1]+w*p[j+4];
}
for (j=4; j<6; j++) {
    q[k++]=s*p[j]+t*p[j+1]+w*p[j+3];
}
q[5]=s*p[7]+t*p[8]+w*p[9];
for (j=0; j<2; j++) {
    p[j]=s*q[j]+t*q[j+1]+w*q[j+3];
}
p[2]=s*q[3]+t*q[4]+w*q[5];

pos=s*p[0]+t*p[1]+w*p[2];
oGradV = (p[2] - p[0]); // surface tangent
oGradU = (p[1] - p[0]); // surface tangent
oUV = uv;
oPosition = float4(pos,1);
oPosition = mul(modelview,oPosition);
oPosition = mul(projection,oPosition); // transformed position
oNormal = cross(oGradV,oGradU);
oNormal = mul((float3x3)modelview,oNormal);
oNormal = normalize(oNormal); / normalized transformed normal
}

```

Figure B.4: Evaluation shader for the Gregory method-continued

BIOGRAPHICAL SKETCH

Ashish Amresh was born in India and moved to the united states as a graduate student in computer science. Upon getting his masters degree he worked for Ronin Entertainment as a graphics software engineer. While working on his doctoral degree he built the decision theater prototype and launched the successful camp game summer program and the computer gaming certificate.