Towards Adaptive Rendering of Smooth Primitives on GPUs

by

Jennifer Fung

B.Sc., The University of British Columbia, 2002

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

October 2005

Abstract

Higher order surface primitives enable artists to create smooth, complex objects by manipulating only a few control points and allow for the generation of smooth surfaces from a very compact representation. The implementation of higher order primitives on Graphics Processing Units (GPUs) has the potential to significantly reduce the bandwidth requirements across the graphics bus. Unfortunately, the GPU support for higher order primitives is still rudimentary.

We present an adaptive, depth-first tessellation algorithm for smooth surfaces. The algorithm takes a set of Bézier control points and tessellates them according to criteria such as screen-space edge length. Other representations, such as subdivision surfaces, can be handled through preprocessing. The algorithm is designed to provide consistent, hole-free tessellations of adjacent patches. In addition, the polygons generated by the tessellator reside on a space filling curve on the 2D manifold of the surface. This guarantees the good memory coherence for both framebuffer and texture memory access. While the current implementation of the method is purely CPU-based, we believe it is suitable for hardware implementation on future generations of GPUs.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Wolfgang Heidrich, for all his help and encouragement. His comments and insights have been invaluable to this research and it was a pleasure to work with him. I would also like to thank Alla Sheffer, my second thesis reader, and Jocelyn Smith, my my proofreader, for helping me write this thesis.

I would also like to thank everyone in the lab and at UBC for making my time in the graduate program such an enjoyable experience.

Last, but not least, I would like to thank my parents, William and Mary Fung, my brother, Jonathan Fung, and Allen Lin for all their support. I couldn't have done it without them.

Chapter 1

Introduction

Higher order surfaces such as splines or subdivision surfaces have long been the modeling primitive of choice in graphics applications. From a modeling point of view, their primary advantage is that artists are able to create smooth, complex objects by manipulating only a few control points. From a rendering point of view, higher order primitives can be beneficial as well. They occupy less memory on a GPU and also allow for the generation of smooth surfaces from a very compact representation, which is important particularly for streaming applications where bandwidth is an issue, such as transmission over a network or rendering using graphics hardware. Geometry transmission over the graphics bus is one of the major bottlenecks for real-time graphics systems. A compact, higher order surface representation could dramatically reduce the transfer cost, thereby speeding up the rendering process, if appropriate tessellators were available on the GPU side.

One problem that such tessellators face is keeping the triangulation free of cracks and T-vertices. There are essentially two ways to achieve this. On the one hand, one can use a subdivision scheme such as Loop [24], Catmull-Clark [7], or Doo-Sabin [14] as a surface representation. In this case, the tessellation requires knowledge of a complete local neighbourhood, which can contain arbitrarily many triangles (depending on the valences of the vertices involved). Because a whole neighbourhood is used for subdivision, the resulting triangulation can be made free of cracks, even if adaptive subdivision (say, based on screen area) is used. Unfortunately, the necessity to encode a whole neighbourhood of the surface does not map well to streaming architectures such as GPUs, and results in the same data being transmitted over the graphics bus multiple times.

The second possibility is to use independent and self-contained patches (such as tensor product Bézier patches or PN Triangles [36]) as the rendering primitive. In this case, the resulting triangulation can be made consistent by enforcing that the tessellation algorithm independently arrives at consistent subdivision decisions for any two neighbouring patches. The easiest way to achieve this is to fix the level of subdivision for all patches on an object. This is known as uniform subdivision.

In this thesis, we document research done towards the development of an adaptive GPU-friendly tessellation algorithm for higher-order patches.

1.1 Research Objectives

While an efficient implementation of any tessellation algorithm is not possible on current graphics hardware, future hardware will contain a tessellator unit specifically designed for these algorithms. Support for the tessellator unit is already available on DirectX 9.0. Given that the actual tessellation will take place in hardware, our goals were to develop a tessellation algorithm with the following properties:

- Provides hole-free adaptive surface tessellation without requiring any information on the neighbouring surfaces
- Has low GPU-to-graphic memory bandwidth requirements
- Requires little GPU memory

We use bi-cubic Bézier patches, but any other parametric self-contained patch representation could be substituted in. The patches could be pre-computed from another representation used for modeling, such as a subdivision surface with a polynomial limit surface.

To maintain consistency between patches during rendering, the subdivision criterion is purely based on patch boundaries, both top-level and tessellated, such that the algorithm will arrive at the same decision for two patches sharing an edge. Our algorithm provides an additional benefit that is important for high-performance rendering on GPUs: the tessellation is created in a depth-first order that causes the resulting polygons to lie on a space-filling curve covering the object surface. This way, we improve cache coherence for memory accesses to both textures (during texture read), and the framebuffer (while writing the final pixel result). Again, this feature addresses one of the common bottlenecks in GPU-based rendering, i.e., the bandwidth between the GPU and graphics memory. Finally, our algorithm leaves a small GPU memory footprint by storing vertex positions in parametric form when tessellating, instead of computing a new control mesh for each new tessellated surface.

1.2 Overview

In Chapter 2 we discuss related work. We then provide an overview of our method (Chapter 3), and describe the depth-first tessellation algorithm (Chapter 3.1), as well as a specific traversal order that creates polygons on a space-filling curve (Chapter 3.2). We conclude the paper with results in Chapter 4 and a summary with discussion of future work in Chapter 5.

Chapter 2

Related Work

An obvious way to render higher-order surfaces is to tessellate them into polygonal meshes in software, and to then transfer those to the GPU for rendering (e.g. [21] and many others), but the CPU-to-GPU transfer overhead for these methods grows exponentially with the level of detail. In parallel, there has also been quite a bit of work on scan-converting higher order primitives directly on the graphics hardware [2, 8, 18, 26]. Unfortunately, it is difficult to incorporate adaptive tessellation, and to prevent cracks between adjacent patches using these approaches.

One solution to this problem is to tessellate the patch borders and the patch surface separately [15, 28]. The patch borders are tessellated to a fixed level and the uniformly tessellated surfaces are joined to the borders using triangle fans. Although the individual patches may undergo different levels of tessellation, adaptive tessellation within a spline surface is not supported. The number of polygons used to tessellate a single spline still grows exponentially with each level of detail.

More recently, several groups have looked into the implementation of subdivision algorithms on GPUs [4, 6, 11, 33]. These approaches need to encode and transmit whole neighbourhoods of the control mesh in order to render a single surface patch. This results in the retransmission of surface data. To overcome this problem, PN triangles [36] were developed and implemented on recent ATI chips. They use normal information at the vertices instead of vertex position information from neighbouring triangles to define the subdivision rules. Thus, every patch is completely self-contained, and can be subdivided independently. However, PN-triangles do not guarantee even tangent plane continuity across patch boundaries (except at the vertices) and only allow for uniform subdivision to maintain a coherent triangulation.

In this chapter, we present an overview of previous work on rendering smooth surfaces, both in software and in hardware. We start with a discussion of Bézier polygon tessellation (Section 2.1) and scanline tessellation (Section 2.2.) In Section 2.3 we describe traditional mesh subdivision algorithms and discuss their hardware implementations in Section 2.4. The PN-triangles method is detailed in Section 2.5.

Finally, we provide an introduction to the space-filling curves, which we use to draw polygons in a order that maintains memory coherence, in Section 2.6.

2.1 Surface tessellation

Surface tessellation is the process of converting a parametric surface, such as a Bézier surface, into polygons or points for rendering on a GPU. The simplest, most obvious way to render Bézier curves is to tessellate the surfaces in software, and then feed the tessellation polygons into the hardware for rendering.

Although tessellating the surface in software makes it easy to perform adaptive tessellation thereby reducing the number of polygons in the tessellation without reducing the surface quality, software solutions suffer from one major problem: the high bandwidth requirements for transferring the polygons to the hardware. Software tessellation is especially taxing on GPU bandwidth because the number of polygons can increase exponentially with tessellation quality. Models with base meshes consisting of only hundreds of polygons may contain hundreds of thousands of polygons when fully tessellated. Many researchers have worked on developing software-based algorithms for reducing the CPU-to-GPU bandwidth load, ([1, 11, 16, 21, 32] to name a few,) but the results are generally geared towards high-end graphics processors. For instance, some algorithms make use of multiple parallel rendering pipelines which are not available on lower-end consumer-grade GPUs.

At the same time, hardware implementations of the adaptive tessellation algorithms are difficult because the adaptive algorithms require information on neighbouring surface patches to ensure the tessellations of neighbouring patches align to form a crack-free surface. This results in multiple transfers of the same data. In addition, unlike with uniform tessellation where the positions of the tessellation polygons are fixed, vertices generated by an adaptive tessellation algorithm may appear at arbitrary points on the Bézier surface. Therefore, a polygon must remain in memory until the polygon's entire neighbourhood is tessellated. This poses problems because GPU memory space is limited, but the number of polygons has the potential to grow exponentially with each additional level of detail.

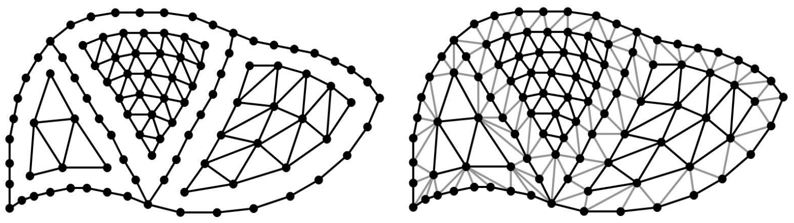


Figure 2.1: An example of tessellating a mesh using coving triangles. *Left:* The boundaries are tessellated to a fixed degree and the surfaces are individually tessellated until each patch meets the tessellation criteria. *Right:* Coving triangles join the tessellated patches to the boundaries.

Coving triangles [15, 28] is a way of implementing partially adaptive tessellation of a model in hardware. First, the patch boundaries are uniformly tessellated until the entire boundary meets the tessellation criteria, such as edge length or curvature. The boundary's level of tessellation is stored on the GPU memory. Then, each surface patch is independently uniformly tessellated until the individual patch surfaces meet the criteria. Finally, the tessellated surfaces are joined to the tessellated boundary using triangle fans or strips called coving triangles. Vertices are evenly spaced apart in uniform tessellation, making it possible to recreate a patch's boundary tessellation using only the patch geometry and the boundary's tessellation level. This allows us to create a crack-free surface without any information on the neighbouring patch tessellations (see Figure 2.1.)

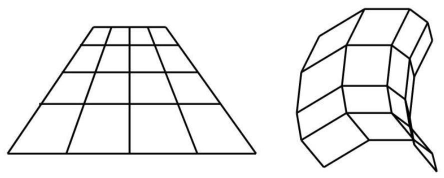


Figure 2.2: Examples of surfaces where coving triangles has problems.

The coving triangles tessellations are globally adaptive over the entire model surface, but are not locally adaptive within the individual surface patches. This global adaptivity is sufficient for models consisting of small to mid-sized low-curvature surface patches, where the tessellation polygons are roughly the same size. However, local adaptivity is needed for models where the tessellation polygons' sizes can vary significantly over the surface of a single patch. For instance, perspective foreshortening on large patches causes the polygons closest to the viewer to appear larger than polygons further away. Tessellation polygons can also vary greatly in size due to the differing angles of orientation between the polygons and the camera (see Figure 2.2.)

2.2 Scanline surface tessellation in hardware

Forward differencing was introduced as a hardware-friendly rendering alternative to software surface tessellation. In forward differencing, surfaces are not tessellated into polygons. Instead, points on the surface are evaluated for each pixel and then drawn directly into the framebuffer. Cracks on the surface will be unnoticeable because they are smaller than a single pixel. Thus, the positions of previous rendered points don't need to be stored in GPU memory to ensure a crack-free surface. The forward differencing algorithm is well known in mathematics literature. In this section, we describe the 2D version of the forward differencing algorithm for the sake of clarity. The 3D algorithm is similar.

Given a parametric curve $f(t)$, we can estimate the first and second derivatives of the curve using the following formulae.

$$\begin{aligned} f'(x) &\approx \tfrac{f(x+\Delta x)-f(x)}{\Delta x} \\ f''(x) &\approx \tfrac{f\prime(x+\Delta x)-f\prime(x)}{\Delta x} \end{aligned}$$

Let p_i and d_i be approximations of $f(i\Delta x)$ and $f\prime(i\Delta x)$ respectively. Starting with $p_0 = f(0)$ and $d_0 = f\prime(0)$, we can trace out the full curve by recursively evaluating the equations

$$\begin{aligned} p_{i+1} &= p_i + d_i \Delta x \\ d_{i+1} &= d_i + f//(x)\Delta x. \end{aligned}$$

Forward differencing is faster than simply evaluating the curve at the points $f(i\Delta x)$ because the second derivative f'' computes faster than $f(x)$ for polynomial functions. Unfortunately, forward differencing suffers from the accumulation of approximation and floating point errors, which may lead to cracks. Nevertheless, per-pixel forward differencing was implemented in the geometry engines of both the IRIS 1000 [9] and the Reality Engine [2].

Alleviating the approximation errors has been the subject of much research. Klassen [18] developed an integer-based forward differencing algorithm that does away with the floating point errors but it is still prone to approximation errors. A hole-free floating point-based algorithm was developed and implemented on NVIDIA graphics processing chips [26]. This method compensates for calculation errors accumulated along the way, but is slower than the integer-based algorithm.

Forward differencing is often used to render points on the surface directly to the framebuffer. Therefore, with uniform forward differencing, care must be taken to ensure the step size, Δx, is small enough that at least one point on the surface is evaluated per pixel. The fixed step-size, combined with the surface variability due to curvature, viewing angles, and perspective can lead to multiple surface points being evaluated within each pixel. Lien et al. [23] developed an adaptive forward-differencing algorithm that evaluates one point on the surface per pixel in screen space, thus avoiding evaluating and drawing unnecessary points. This method relies on the fact that connective geometry is unnecessary at the sub-pixel level and can be ignored. Because of this, adaptive forward differencing cannot generate tessellations involving polygons larger than a single pixel. This is especially wasteful for rendering low-curvature Bézier surfaces which may by adequately approximated with only a small number of flat polygons.

2.3 Mesh subdivision

Subdivision is a method of generating complex geometry by recursively refining and smoothing a coarse mesh representation based on a set of rules. Mesh edges and vertices can be tagged as “sharp”, causing them to appear as creases and points in the subdivision limit surface. The most common subdivision algorithms are the Doo-Sabin [14], Catmull-Clark [7], and Loop [24] algorithms. Since then, several other subdivision methods have been proposed [17, 19, 20, 34, 35, 38].

2.3.1 Doo-Sabin Subdivision

Intuitively, the Doo-Sabin algorithm recursively shaves off the mesh edges and vertices until it arrives at a smooth surface. The mesh faces “shrink” around

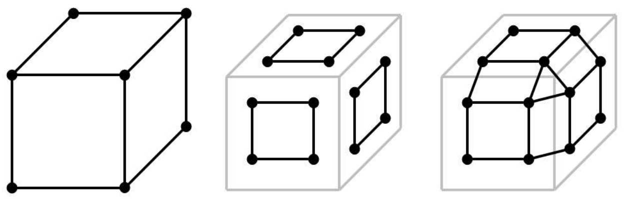


Figure 2.3: One iteration of Doo-Sabin subdivision. *Left:* the original mesh. *Centre:* shrinking the faces. *Right:* adding new polygons where the vertices and edges were pulled apart.

the face centroid and polygons are added where the vertices and edges joining the faces once were (see Figure 2.3.) During the shrinking phase, each mesh face is reduced by half while its centroid remains in the same place, so that the mesh vertices lie halfway between the centroid and the original vertex position. While simple, the Doo-Sabin algorithm may produce non-planar polygons over vertices with valence greater than 3.

2.3.2 Catmull-Clark Subdivision

The Catmull-Clark algorithm is a generalization of bi-cubic B-spline subdivision. During each subdivision step, new vertices are created at what are called the face, edge and vertex points. The new face points are placed at the centroids of each face and the new edge points are placed halfway between the centroids of each edge's adjacent faces. The new vertex point $\hat{s}$ is a convex combination of the following.

- q, the average of the new face points sharing the old vertex
- r, the average of the midpoints of all the old edges incident on the old vertex
- s, the old vertex position

The new vertex point is

$$\hat{s} = \frac{1}{N}q + \frac{1}{N}r + \frac{N-3}{N}s \tag{2.1}$$

where N is the old vertex valence. New subdivision quadrilaterals are created joining together a new face point, one of the face's new vertex points, and the two new edge points of the common edges, as in Figure 2.4.

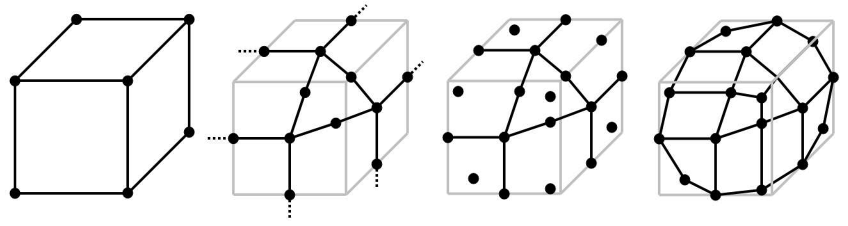


Figure 2.4: An example of one iteration of Catmull-Clark subdivision. *From left to right:* the original mesh, adding the face and edge points, placing the vertex points, connecting the points with faces to get the final subdivided mesh.

2.3.3 Loop Subdivision

The Loop subdivision algorithm produces a set of curvature continuous quartic Bézier patches in the limit. Unlike Doo-Sabin and Catmull-Clark, Loop subdivision only works on triangle meshes.

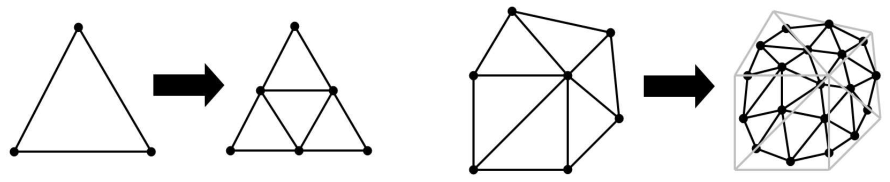
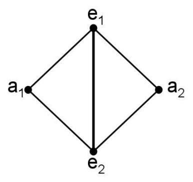


Figure 2.5: An example of one iteration of Loop subdivision. *Left:* each triangle is subdivided into 4 new triangles. *Right:* the original mesh and the subdivided mesh.

When creating the next level of subdivision, each triangle is divided into 4 new triangles and the entire mesh is smoothed, as in Figure 2.5. The vertices placed on the edge midpoints are called edge vertices. During the smoothing phase, the position of a new edge vertex $\hat{e}$ is



$$\hat{e} = \frac{3}{8}(e_1 + e_2) + \frac{1}{8}(a_1 + a_2) \tag{2.2}$$

where the e_1 and e_2 are the edge endpoints and a_1 and a_2 are the remaining vertices of the adjacent faces.

The new vertex positions $\hat{v}$ are

$$\hat{v} = \alpha(N)v + (1 - \alpha(N))q \tag{2.3}$$

where v is the old vertex position, N is its valence, q is the average of the positions of the old vertices that share an edge with v, and $\alpha(N)$ is defined as follows.

$$\alpha(N) = \frac{3}{8} + (\frac{3}{8} + \frac{1}{4}cos(\frac{2\pi}{N}))^2 \tag{2.4}$$

In Loop subdivision, vertices and edges may be tagged as sharp. Sharp vertices and edges are simply left alone during the smoothing stage, giving them a crisper, more defined appearance. Edges and vertices may also be given an integer sharpness value s where a sharpness value of ∞ means that the edge or point is infinitely sharp and a 0 means it is not sharp at all. Edges and vertices are left unsmoothed for s recursive iterations. Semi-sharp edges and vertices give a model a softer, more organic look, whereas perfectly sharp edges result in a crisp, artificial or industrial feel.

2.3.4 Discussion

Subdivision is an attractive way of generating smooth surfaces because of its simplicity of implementation and the compactness of the base representation. It also allows the user to specify varying levels of detail corresponding to the number of levels of subdivision the model is to undergo. Unfortunately, software implementations of subdivision algorithms are very slow and transferring the subdivided mesh to the graphics hardware requires a lot of bandwidth, even if an adaptive algorithm is used to lower the polygon count. We discuss hardware implementations in Section 2.4.

2.4 Mesh subdivision in hardware

Subdivision algorithms have the ability to produce a smooth, attractive model from a simple mesh representation. Not surprisingly, considerable research has gone into implementing subdivision schemes in hardware, so subdivision can be used in interactive and real-time applications. There are two main approaches to implementing subdivision in hardware: one based on pre-computed basis functions and another that utilizes displacement maps.

2.4.1 Pre-computed basis functions

Loop [24] and Catumull-Clark [7] subdivision transform each mesh polygon into a NURBS patch in the limit. In addition, each NURBS patch is defined only by the positions of the vertices in the 1-neighbourhood surrounding the polygon, which means that any point on a limit surface patch over a mesh polygon can be expressed as a convex combination of the vertices in the 1-neighbourhood around the polygon. Furthermore, corresponding points on polygons with similar connecting geometry use the same coefficients or basis functions. For example, the point defined by the barycentric coordinates (α, β, γ) on two triangles with the same vertex valences (see Figure 2.6 for example) will be defined by the same convex combination of vertices in their respective 1-neighbourhoods.

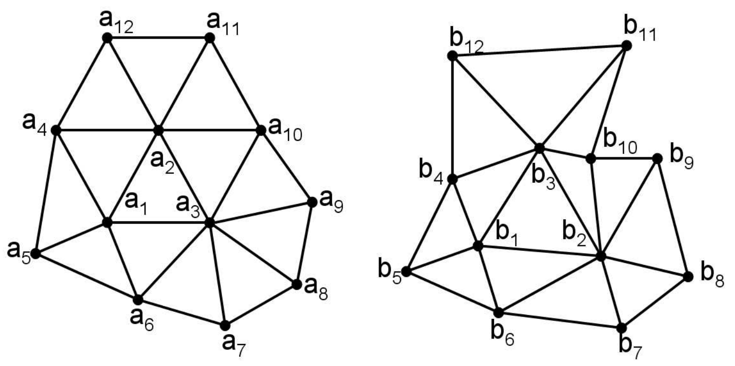


Figure 2.6: Triangles (a_1, a_2, a_3) and (b_1, b_2, b_3) have similar connecting geometry.

Some hardware subdivision algorithms [4, 6, 12] take advantage of this fact by precomputing all the coefficients or basis functions needed to represent the subdivision surface of a mesh and storing them in tables in GPU memory. Then, subdividing a polygon is a simple matter of transferring the polygon's 1-neighbourhood to the GPU, performing a table lookup to find the coefficients that match the vertex valences and applying the appropriate instructions in hardware to evaluate the vertex positions. The basis function method can be combined with forward differencing to speed up the calculation times [3].

Shiue et al. [33] take the basis function method one step further. They store the neighbourhood vertex positions in a "patch texture" and treat surface subdivision as a form of texture scaling. The scaling function is defined such that each pixel on the scaled texture corresponds to a vertex position on the subdivided mesh. Triangles with similar connective geometry can be stored in the same patch texture. The instructions for scaling the patch texture are stored in a lookup table.

Hardware subdivision algorithms that use pre-computed basis functions can be quite effective for meshes with limited vertex valence connectivity. In real applications, however, meshes may have arbitrary connectivity. The GPU memory must store one basis function for each polygon on the mesh with differing vertex valences. This leaves less memory available for storing other information important for rendering, such as texture maps and BRDFs.

2.4.2 Displacement mapping

Displacement mapping is a compact way of representing detailed meshes. A coarse mesh is used to approximate the mesh surface and the finer geometric detail is stored in a displacement map that stores information on how to displace the surface of the coarse mesh to achieve a better representation of the detailed model (see Figure 2.7). The displacement map is sent to the graphics hardware as a texture accompanying the coarse mesh and the displacement is performed in the hardware. Displacement mapping works well when representing fine details over a simple geometry. For instance, a model of an intricate relief carving on a flat table would be well suited to displacement mapping.

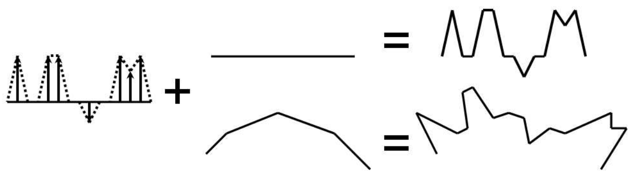


Figure 2.7: The displacement map (left) is applied to different simple surfaces (centre.) The resulting complex geometry is shown on the left.

More recently, displacement mapping has been used to represent mesh subdivision limit surfaces in order to quickly subdivide meshes in hardware. During rendering, the displacement map contains information on how the base mesh surface is to be displaced to achieve the subdivision limit surface [13, 22] Unfortunately, this method suffers from surface smoothness artifacts across the patch boundaries due to numerical round-off errors and texture stretching, making it unsuitable for generating high quality subdivision surfaces. Boo et al. [5] transmit the polygon neighbourhood, along with the actual polygon, to ensure smoothness across boundaries, but this technique leads to multiple transmissions of the same data, wasting precious GPU bandwidth. Also, a displacement map representation is less compact than a NURBS representation.

2.5 PN-Triangles

Vlachos et al. [36] developed PN-triangles to overcome the problem of having to transmit additional polygon neighbourhood information to the hardware in order to draw smooth surfaces. The PN-triangles method draws a smooth surface over a triangle based only on triangle vertex positions and normals, which would normally be transmitted to the hardware for a traditional rendering pass. This makes it easy to incorporate in projects where the artwork is already finalized or there are limited resources to hire "touch-up" artists.

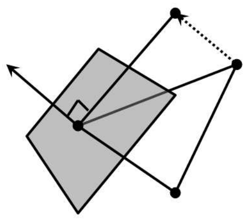


Figure 2.8: Projecting an edge onto a plane defined by one of the vertex normals.

A cubic triangular Bézier control mesh is generated over each triangle based only on its triangle vertex positions and normals. The corner control points are placed at the triangle vertices. The 6 control points on the boundaries are generated by projecting the triangle edges onto the plane perpendicular to the normal of the closest vertex (see Figure 2.8) and scaling the projected edge by

$\frac{1}{3}$. The centre control point is placed at the position

$$C_c + (C_c - C_t) \tag{2.5}$$

where C_t is the triangle centroid and C_c is the centroid of the six edge control vertices.

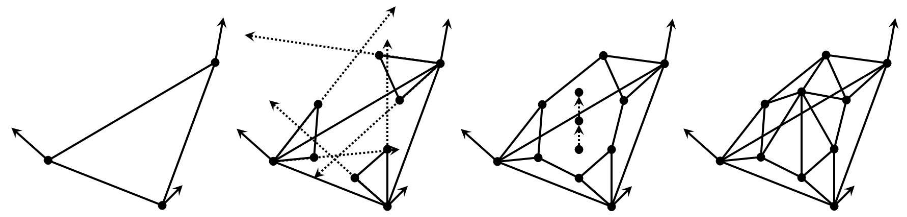


Figure 2.9: Generating a cubic Bézier control polygon over a triangle. *From left to right:* the triangle and its vertex normals, adding the edge control points, placing the centre control point, the resulting control polygon.

The resulting surface patches are smooth and hole-free, since adjacent triangles share the vertices that define the control mesh at the boundary, but it is impossible to guarantee smoothness between adjacent patches without neighbourhood information. In fact, adjacent patches are often only C^0 continuous to one another, meaning patch boundaries align but the tangent planes at the boundaries do not. This often results in sharp and disturbingly obvious discontinuities along the mesh edges. Some of the discontinuities can be hidden by interpolating the normals separately from the geometry using quadratic, as opposed to cubic, interpolation, but this creates a disparity between the surface geometry and lighting.

2.6 Space-filling curves

A space-filling curve is a recursively defined curve that covers an entire 2-dimensional area in its limit. We look at the Hilbert space-filling curve [30]

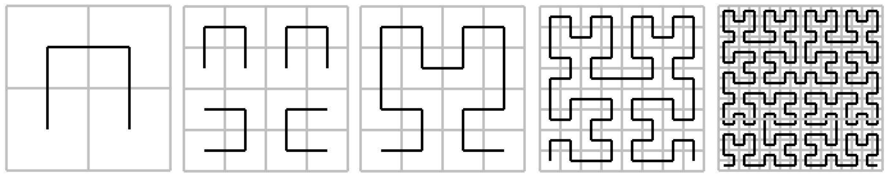


Figure 2.10: The first few iterations for generating a Hilbert space-filling curve.

as an example. The limit of a Hilbert curve is a square. To generate the curve, we start with a square divided into 4 equal-sized squares. The curve connects the centres of the four squares in the base level, as in Figure 2.10. This is known as the base shape. Every level of recursion, the subsquares are replaced with smaller versions of the base shape, and the curves are joined together. The base shapes are oriented such that the joining line segments only cross a single edge.

It has been known for some time that rasterizing polygons in the shape of a space-filling curve, such as the Hilbert curve [31], produce significantly improved texture and framebuffer cache when compared to standard scanline traversal [25, 37], at least when textures and the framebuffer are stored in tiled fashion instead of scanline order. Although specific information about the memory layout of commercial GPUs is difficult to come by, this seems to be the case in modern hardware.

A nice property of many space-filling curves is that they can be easily created by hierarchical depth-first traversal procedures like our tessellation algorithm described in Section 3.1. All that is required are a few local rules at every level that determine in which order the subnodes are created.

2.7 Summary

We discussed several methods for rendering smooth, complex surfaces from coarse representations, including polygon tessellation (Section 2.1), scanline surface tessellation (Section 2.2), mesh subdivision (Sections 2.3 and 2.4), and PN-Triangles (Section 2.5). Although many of these methods support some combination of real-time rendering, improved memory coherence, and local surface tessellation adaptivity, none of them support all three features.

Our proposed locally adaptive surface tessellation technique is suitable for implementation in hardware, making it possible to render smooth surfaces in real-time. We also use space-filling curves (Section 2.6) to improve performance through cache coherence.

Chapter 3

Tessellation

The algorithm presented in this thesis adaptively tessellates cubic tensor-product Bézier patches according to criteria that are only based on tessellation edge information. The algorithm is depth-first recursive and the polygons are generated in space-filling curve order.

Basing the tessellation criteria entirely on tessellation edge information ensures that subdivision decisions are consistent across neighbouring patches, preventing cracks. Our method differs from the coving triangles methods introduced by Rockwood [29] and Filip et al. [15] in that the tessellation criteria are evaluated for boundaries created by the tessellation as well as for the top-level patch boundaries.

Our tessellation is depth-first, which implies a logarithmic memory consumption (linear in the number of tessellation levels). The representation for each level is designed to optimize memory footprint: instead of explicitly generating and storing the control mesh for every level, we only store one control mesh for the whole patch. At every tessellation level, we store only the (u, v) parameter values for the corners of the of the subdivided regions. In addition to reduced memory cost, storing the patch corner parameter values also allows us to create both quadrilaterals and triangles in our tessellation algorithm. The latter helps simplify the crack-free adaptive tessellation process. This approach does come at the cost of having to recompute some vertex positions due to multiple polygons sharing the same vertex. However, the reduced memory footprint and the support for adaptive tessellations outweighs these costs.

In addition, we can generate the triangles in space-filling order at no extra cost. In a uniform subdivision setting, the center points of the resulting quadrilaterals reside on a Hilbert curve embedded in the 3D geometry. As shown by Voorhies [37], space-filling curves provide a dramatic improvement over scan-line traversal orders in terms of memory coherence if the image data (textures and the framebuffer) are stored in tiled rather than scanline order. As a result, our method simultaneously optimizes the accesses to graphics memory for both texture memory read and framebuffer read and write. This is similar in spirit to recent work on polygon scan conversion in Hilbert order [25].

We expand on our surface tessellation process and the tessellation order in the rest of this chapter.

3.1 Surface tessellation

In this section we describe the tessellation process in more detail. We start with uniform subdivision and then discuss the adaptive case. Let $S(u, v)$ with parameter values $(u, v)\epsilon[0, 1]x[0, 1]$ be the tensor-product Bézier patch that requires tessellating.

3.1.1 Uniform tessellation

Our algorithm uses a recursive depth-first method for generating uniform tessellations. The recursive property makes it easy to implement tessellations in a space-filling curve order. With uniform tessellation, surfaces are tessellated the same way for each level of detail regardless of what the surface geometry may look like. A level-of-detail variable ℓ specifies the number of tessellation steps a surface will undergo. This ℓ is analogous to the level-of-detail variable used in subdivision schemes.

The corners of the base quadrilateral (i.e. the $\ell = 0$ surface) are located at the surface's corners, i.e. the points at parameter values $S(0, 0)$, $S(1, 0)$, $S(1, 1)$, and $S(0,1)$. We generate the surface for the next tessellation level of detail by adding five new vertices, 1 on each of the 4 edges and 1 in the centre of the polygon, and subdividing each quadrilateral into four new quadrilaterals.

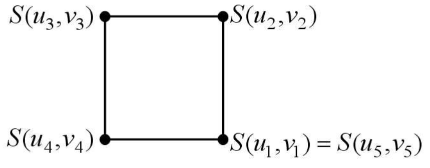


Figure 3.1: Base polygon vertex placement.

For example, given a quadrilateral with corner vertices at $S(u_i, v_i)$, $1 \leq i \leq 4$. The new boundary edge vertices are placed on the Bézier surface at points $S(\frac{1u_i+u_{i+1}}{2}, \frac{1}{v_i+v_{i+1}})$ where $(u_5, v_5) = (u_1, v_1)$ (see Figure 3.1.) These points correspond to the boundary midpoints. The new center vertex is placed at the surface's parametric centre $S(\frac{1}{4}\sum u_i, \frac{1}{4}\sum v_i)$. All vertices in the tessellation lie on the surface $S(u, v)$, as in Figure 3.2.

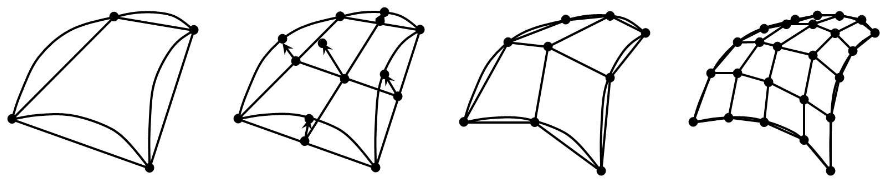


Figure 3.2: Tessellating a quadrilateral Bézier patch.

With uniform tessellation, a crack-free surface is guaranteed as long as neighbouring surface patches undergo the same level of tessellation. However, uniform recursive tessellation results in an exponential increase in the number of polygons at each level of detail, making the surface slow to render at higher levels of detail, even if the tessellation is performed in hardware. We use adaptive tessellation to reduce the number of polygons in the tessellation without lowering the final surface quality.

3.1.2 Adaptive tessellation

Adaptive tessellation allows us to avoid drawing unnecessary polygons, giving the algorithm more time to spend on areas that require further refinement. For example, surface areas that are flat or smaller than a single pixel don't need to be tessellated further, whereas areas that are curved or cover a significant amount of screen space may require more attention. In addition, top-level patches may be of drastically varying size, depending on the amount of detail necessary for representing a particular part of the object. Uniform subdivision of such models may result in extremely non-uniform polygon areas.

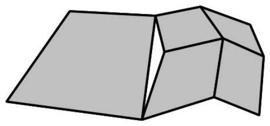


Figure 3.3: An example of a crack forming in the mesh due to differing levels of tessellation on adjacent patches.

With adaptive tessellation, cracks in the surface may appear if adjacent areas are subjected to different levels of tessellation (See Figure 3.3.) In CPU-based algorithms that have access to the complete model geometry (such as [19], [27], and others) a typical solution is to add some extra polygons to fill potential cracks. However, we limit our GPU-friendly algorithm to have access to the geometry of only a single surface at a time. This limited access avoids having to transmit entire neighbourhoods, which may be of arbitrarily large size. Because of this restriction, there is no way of knowing how the neighbouring surfaces will be tessellated, making it difficult to maintain consistency across patches. In particular, this rules out subdivision criteria based on surface curvature or area.

Our algorithm solves this problem by using tessellation criteria entirely based on boundary information. While the idea of tessellation criteria that completely ignore the surface interior may seem limiting, it works well in practice. Uniformly tessellating the surfaces a few levels-of-detail before applying adaptive tessellation creates boundaries in the patch interiors. So, when the tessellation criteria are applied, the criteria are actually basing their decisions on the patch inner geometry, even though the tessellation criteria themselves are entirely boundary-based. Pre-tessellating the surfaces 1 level-of-detail should be sufficient for cubic Bézier surfaces like the ones generated from subdivision surfaces, and 2 levels-of-detail will effectively deal with non-convex and non-concave surface patches.

The tessellation criteria are re-evaluated at every tessellation level for every new polygon boundary created, resulting in an adaptive tessellation of the entire spline surface. Assuming the Bézier patches are at least boundary continuous, the tessellated geometry of a shared boundary will remain the same, regardless of which patch is currently processed by the GPU. Therefore, since the decision whether or not a boundary needs tessellation is based solely on the boundary geometry, then a shared boundary will be divided the same number of times and at the same places in two adjacent patches and the boundary vertices will line up.

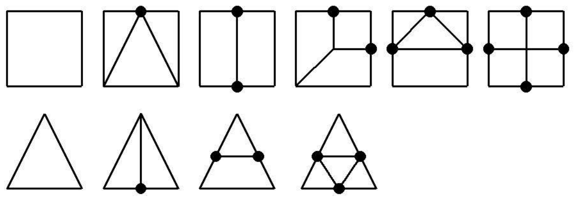


Figure 3.4: Possible tessellation cases in our algorithm.

How the interior of the surface of the patch is tessellated depends on which edges require further refinement. The tessellations must only introduce new vertices on edges that need refinement and must leave edges that don't need refinement intact. Figure 3.4 lists all possible configurations based on which edges need to be subdivided and which do not. When a boundary between $S(u_1, v_1)$ and $S(u_2, v_2)$ requires subdivision, the new boundary vertex is always placed at the point $S(\frac{u_1+u_2}{2}, \frac{v_1+v_2}{2})$. The centre vertex is placed at $S(\frac{1}{4}\sum u_i, \frac{1}{4}\sum v_i)$if needed.

Note that two of the quadrilateral cases actually create triangles rather than quadrilaterals, and that in one case the quadrilaterals are no longer aligned with the major coordinate axes. Since we now have triangular surface pieces as well, we also define subdivision rules for those (see Figure 3.4.) It is worth pointing out that the surfaces over those subdivided regions cannot be represented exactly as bi-cubic Bézier patches unless trimming curves are used. In our approach this is not a problem, however, since we do not explicitly represent the subdivided surface pieces. Rather, as pointed out in the overview of Chapter 3, we store the positions of the vertices only in parameter space, and use point evaluation to compute the corresponding 3D locations from the representation of the full (i.e. top-level) patch.

3.1.3 Edge tessellation criteria

This section describes the four different boundary tessellation criteria we use with our algorithm. This list of tessellation criteria is not exclusive. Other criteria may be used as long as only edge information is involved.

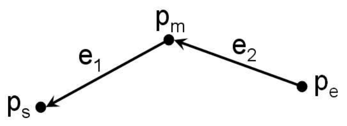


Figure 3.5: An illustration of the terms used by our edge criteria.

We first define a few terms, illustrated in Figure 3.5 for the sake of clarification. Let $p_s = S(u_s, v_s)$ and $p_e = S(u_e, v_e)$ be the start and end vertices of an edge of a tessellated face. Some of the tessellation criteria described in this section require evaluating the vertex at the center point $p_m = S(\frac{u_s+u_e}{2}, \frac{v_s+v_e}{2})$. Also, let $e_1 = p_s - p_m$ and $e_2 = p_m - p_e$ be the vectors between the boundary endpoints and its midpoint.

We define a boundary tessellation function $T(S, u_s, v_s, u_e, v_e) \in \{yes, no\}$ where if $T(S, u_s, v_s, u_e, v_e) = yes$ then the boundary between $S(u_s, v_s)$ and $S(u_e, v_e)$ needs tessellating. Otherwise, the boundary is left alone. The four possible tessellation functions we propose are the edge length function T_l, the arc length function T_a, the straightness function T_s, and the tangent function T_t.

Edge length criterion

The edge length criterion simply uses the distance between the points p_e and p_s as a coarse estimate of the the tessellation boundary arc length. The edge is tessellated if this distance is larger than the threshold δ_l.

$$T_l(u_s, v_s, u_e, v_e) = \begin{cases} yes, & \text{if } ||p_e - p_s|| < \delta_l \\ no, & \text{otherwise} \end{cases}$$

Calculating the distance $||p_e - p_s|| = \sqrt{(p_e - p_s) \cdot (p_e - p_s)}$ can be problematic because the square root operation takes a while to evaluate and may have significant round-off errors. We modified the criterion to speed it up as follows.

$$T_l(u_s, v_s, u_e, v_e) = \begin{cases} yes, & \text{if } ||p_e - p_s||^2 < \delta_l^2 \\ no, & \text{otherwise} \end{cases}$$

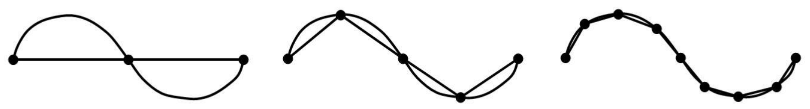


Figure 3.6: The edge length criterion becomes a better approximation of the surface arc length after a few iterations of the tessellation algorithm.

This criteria is the quickest to evaluate of the four we propose and is effective for low curvature surfaces. The edge length criterion is useful on splines that are relatively flat. While this is not common for model base surfaces, the tessellation boundaries become less rounded and the point p_m grows closer to the edge joining points p_s and p_e as the tessellation progresses. Therefore, the edge length criterion becomes a viable approximation of the arc length after a few iterations of the tessellation algorithm are performed with a different criterion.

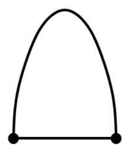


Figure 3.7: The edge length criterion would be a poor choice for a boundary curve such as this one.

Unfortunately, it is not view dependent and so may drastically over-estimate the screen-space arc-length when the boundary is viewed from certain angles. Also, it may be a poor estimator of the actual arc length of high-curvature boundaries. The boundary curve shown in Figure 3.7 is an example where this may be the case. The boundary endpoints lie close enough together that $T_\ell = no$ and the boundary is not tessellated further, even though it is a poor estimate of the boundary arc length and requires more tessellation. Incorporating the boundary midpoint p_m into the boundary length estimation as in the arc length criterion will give us a better approximation.

Arc length criterion

The arc length function uses additional information provided by p_m to estimate the arc length of the boundary curve in pixels when rendered. The boundary needs tessellating if the estimated arc length is greater than the threshold δ_a.

Instead of computing the actual arc length of the curve, we estimate the

Chapter 3. Tessellation 28

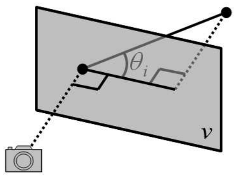


Figure 3.8: An illustration of the terms used by the arc length criterion.

screen distance between points p_s, p_m, and p_e. Let r be the screen resolution, d be the distance from the object to the camera, and v be the viewing direction (see Figure 3.8.) Also, let θ_i be the angle between edge e_i and the plane perpendicular to the viewing direction. Then, the approximate length $L(e_i)$ of the edge e_i in pixels is

$$L(e_i) = \frac{r}{d}||e_i|| \cos\theta_i.$$

The approximate arc length a of the boundary curve is

$$a = \frac{r}{d}(||e_1|| \cos\theta_1 + ||e_2|| \cos\theta_2).$$

We use the dot products $v \cdot e_i$ to avoid evaluating the cosine and square root functions.

$$\begin{aligned}
v \cdot e_i &= ||e_i|| \cos(\theta_i + 90) \\
&= ||e_i|| \sin(\theta_i) \\
(v \cdot e_i)^2 &= ||e_i||^2 \sin^2\theta \\
&= ||e_i||^2 (1 - \cos^2\theta) \\
\tfrac{(v \cdot e_i)^2}{||e_i||^2} &= 1 - \cos^2\theta \\
\cos^2\theta &= 1 - \tfrac{(v \cdot e_i)^2}{||e_i||^2}
\end{aligned}$$

Then, we derive the arc length approximation $\hat{a}$ as follows.

$$
\begin{aligned}
a &= \tfrac{r}{d}(||e_1|| \cos\theta_1 + ||e_2|| \cos\theta_2) \\
a^2 &= (\tfrac{r}{d})^2(||e_1||^2 \cos^2\theta_1 + 2||e_1||||e_2|| \cos\theta_1 \cos\theta_2 + ||e_2||^2 \cos^2\theta_2) \\
a^2 &\leq (\tfrac{r}{d})^2(||e_1||^2 \cos^2\theta_1 + ||e_2||^2 \cos^2\theta_2 + 2||e_1||||e_2||) \\
&\leq (\tfrac{r}{d})^2(||e_1||^2 \cos^2\theta_1 + ||e_2||^2 \cos^2\theta_2 + 2||e_1||^2||e_2||^2) \\
\hat{a}^2 &= (\tfrac{r}{c})^2(||e_1||^2(1 - \tfrac{(v\cdot e_1)^2}{||e_1||^2}) + ||e_2||^2(1 - \tfrac{(v\cdot e_2)^2}{||e_2||^2}) + (e_1 \cdot e_1)(e_2 \cdot e_2)) \\
&= (\tfrac{r}{c})^2(||e_1||^2 + ||e_2||^2 - (v \cdot e_1)^2 - (v \cdot e_2)^2 + (e_1 \cdot e_1)(e_2 \cdot e_2))
\end{aligned}
$$

Finally, the arc length criterion is as follows.

$$
T_a(u_s, v_s, u_e, v_e) = \begin{cases} yes, & \text{if } \hat{a}^2 < \delta_a^2 \\ no, & \text{otherwise} \end{cases}
$$

The arc length criterion works best when a per-pixel tessellation is desired, because it estimates the arc length in screen space. It also works well for point-splatting, where the model is rendered entirely in points instead of triangles, because it controls the distance between vertices.

The arc length criterion does nothing to ensure transitions between polygons are smooth and so the model may have a jaggy or faceted appearance in the lower levels of detail. The straightness and tangent tessellation criteria presented below are more appropriate if the desired final tessellation is to have fewer polygons.

Straightness criterion

A common way of deciding whether or not to continue tessellating a patch is to tessellate if the inner surface of the patch has a high curvature and to stop if it is relatively flat. The straightness criterion is similar in spirit, but since our tessellation criteria can only use patch boundary information, we use the boundary curvature instead.

The straightness function estimates how straight the boundary curve is. Let β be the angle between e_1, and e_2. The value of $\cos\beta$ is lowest when the

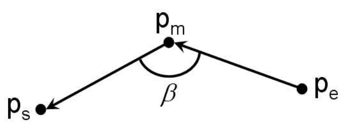


Figure 3.9: β is the angle between p_s, p_m, and p_e.

$\beta = 180°$ i.e. when the boundary is perfectly straight. Therefore, if the cosine of the angle is greater than the threshold δ_s, then the boundary is considered not straight enough and requires further tessellation.

$$T_s(u_s, v_s, u_e, v_e) = \begin{cases} yes, & \text{if } \cos\beta > \delta_s \\ no, & \text{otherwise} \end{cases}$$

We use an approximation to avoid evaluating the cosine function.

$$\cos\beta = \frac{e_1 \cdot e_2}{||e_1||||e_2||} \approx \frac{e_1 \cdot e_2}{||e_1||^2} = \frac{e_1 \cdot e_2}{e_1 \cdot e_1}$$

This assumes that e_1 and e_2 are approximately the same length, which is the case in many cubic Bézier surfaces.

$$T_s(u_s, v_s, u_e, v_e) = \begin{cases} yes, & \text{if } \frac{e_1 \cdot e_2}{e_1 \cdot e_1} > \delta_s \\ no, & \text{otherwise} \end{cases}$$

The straightness criterion is good at detecting large, flat areas. It works best on models with a large variety of surface curvatures, but can result in skinny triangles in the higher levels of detail.

Tangent criterion

The tangent criterion guarantees a certain amount of smoothness between adjacent patches at the patch corners by continuing to tessellate an edge if the angle between the edge and the surface tangent at its endpoints is too large. In many Bézier models, neighbouring Bézier patches are at least C^1 continuous to one another, i.e. tangents of adjacent surfaces are equal at the patch boundaries.

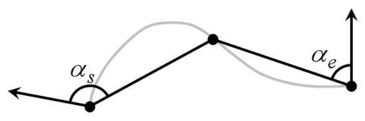


Figure 3.10: α_s and α_e are the angles between the tessellation and the surface normals.

Therefore, the tessellation will be completely smooth at the patch corners if it is parallel to the tangent planes at the corners.

Let n_s and n_e be the surface normals at points p_s and p_e. The tangent planes are perpendicular to the surface normals. α_s and α_e are the angles between the current tessellation and the surface normals n_s and n_e. The edge requires further tessellation if one of the angles α_s or α_e is too far away from 90° or 270°, i.e. if either $|cos(\alpha_s)|$ or $|cos(\alpha_t)|$ is too large. For a tangent cosine angle threshold value δ_t the tangent plane function is defined as follows.

$$T_t(u_s, v_s, u_e, v_e) = \begin{cases} yes, & \text{if } |\cos\alpha_s| > \delta_t \\ & \quad \text{or } |\cos\alpha_e| > \delta_t \\ no, & \text{otherwise} \end{cases}$$

After squaring both sides of the equation in order to avoid evaluating the cosine functions, we get the following:

$$\cos^2\alpha_i = \frac{(n_i \cdot e_i)^2}{||e||^2}$$

$$T_t(u_s, v_s, u_e, v_e) = \begin{cases} yes, & \text{if } \cos^2\alpha_s > \delta_t^2 \\ & \text{or } \cos^2\alpha_e > \delta_t^2 \\ no, & \text{otherwise} \end{cases}$$

The tangent criterion is best at obscuring the boundaries between patches. However, it is not view dependent, even though its calculations are heavily based on vectors which may change with the application of a perspective transformation. It is possible to transform the vectors before performing the calculations.

This introduces no computational overhead because the vectors must be transformed prior to rendering anyway.

Discussion

These edge tessellation criteria introduce very little computational overhead to the algorithm. The positions p_s and p_e should already be stored in memory from the previous recursive call. Determining p_m for evaluating the tessellation criteria does not introduce any significant overhead because p_m needs to be calculated anyway if the edge requires tessellation or if uniform tessellation is being performed. The screen resolution, distance from the object to the camera, and viewing direction only need to be computed once per frame. Also, the normals at p_s and p_e which are used in the tangent criterion need to be computed anyway, since they are required for shading.

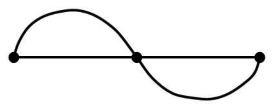


Figure 3.11: In some cases, the tessellation may be planar even when the surface has a high curvature.

With the arc length and straightness tessellation criteria, it is possible to drastically underestimate the arc length or overestimate the straightness of the curve if the points p_s, p_m, and p_e are almost linear when the curve is not (see Figure 3.11.) Adding more points to the calculation would result in a more accurate estimation but would require more operations and there would be additional overhead required to compute the new point positions. In the vast majority of cases this isn't a problem since Bézier polygons generated from a subdivision surface are generally either concave or convex, so the case where the points are linear when the surface is not does not occur very often.

3.2 Tessellation order

As mentioned in Section 3, it is possible to generate the polygons of the tessellation the order of a space-filling curve to improve the coherence of memory access patterns during texture mapping and writing to the framebuffer. This addresses the problems with data transfers between the GPU and graphics memory.

In the following, we describe how this can be achieved for both uniform and adaptive subdivision. While the number of different cases may at first seem daunting, Section 3.3 discusses an efficient and straightforward implementation using lookup tables.

3.2.1 Space-filling curves

The order in which our uniform quadrilateral tessellations are generated follows the shape of the Hilbert curve, shown in Figure 3.12.

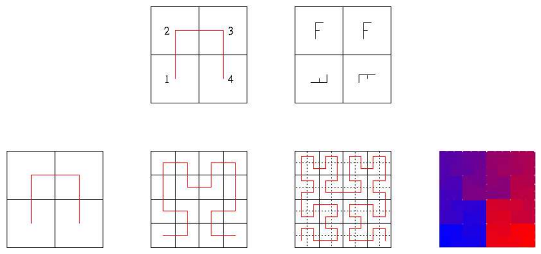


Figure 3.12: The Hilbert space-filling curve.

Our adaptive tessellation algorithm generates triangles as well as quadrilaterals, but the Hilbert curve only applies to rectangular domains. We therefore came up with the space filling scheme depicted in Figure 3.13.

Like the Hilbert curve, the triangular curve never self-intersects for all recursive levels. The proof goes as follows.

We partition the base shape into 4 identical convex partitions to show that the non-limit curves are not self-intersecting. The curve at the top-level passes

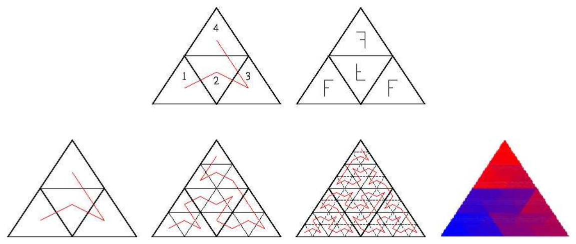


Figure 3.13: The triangular space-filling curve we use in our algorithm.

through the centres of each of the 4 partitions. It lies completely within the partitions and is not self-intersecting. At any level of refinement, the curve consists of non-intersecting copies of the original pattern contained within convex non-intersecting partitions, as well as the line segments connecting the copies in neighbouring partitions. In the limit, the endpoint of the curve segment in one partition and the start point of the segment in the next partition converge to a point that is shared by both partitions. Since the partitions are convex, the connecting line segments do not introduce self-intersections either, and the non-limit curves are not self-intersecting.

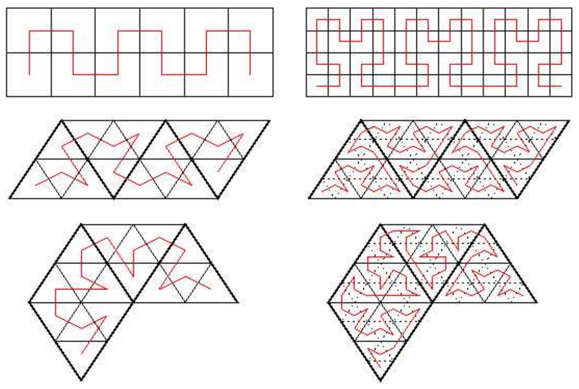


Figure 3.14: Coherence between top-level patches is maintained if they are specified in a quadrilateral strip, a triangle strip, or a triangle fan.

An interesting feature of this approach is that coherence can even be maintained from one top-level patch to the next if they are specified in an order resembling a quadrilateral strip (Figure 3.14). A similar result applies to triangular patches specified in triangle strip or triangle fan order.

3.2.2 Adaptive tessellation curves

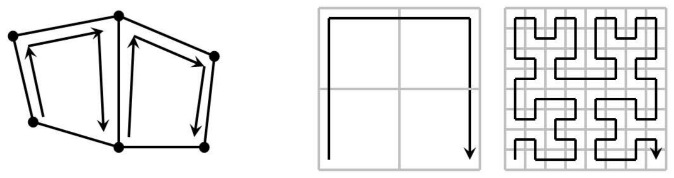


Figure 3.15: *Left:* The last vertex and the first vertex of two adjacent tessellated polygons are the same, maintaining coherence between adjacent tessellated polygons. *Right:* The first and last vertices traversed on the top-level Bézier patch are the same regardless of tessellation level-of-detail, maintaining inter-patch coherency.

The Hilbert rules cannot directly be used for adaptively tessellated surfaces because adaptive tessellation does not necessarily create 4 subpatches at every level. Instead, the polygons are drawn in an order that tries to preserve the cache coherent properties of the space-filling curve. Specifically, we preserve inter-patch and intra-patch coherence.

Inter-patch coherence is maintained by ensuring the last vertex of a tessellated polygon and the first vertex of the next tessellated polygon are the same. This way, the vertex ordering never skips back and forth between polygons needlessly. The vertices are presented in one continuous, non-intersecting chain.

We need to preserve intra-patch coherence or coherence between patches because the Bézier surfaces may be oriented to maintain coherency between top-level patches as mentioned in Section 3.2.1. Our algorithm respects this inter-patch coherency by organizing the tessellation such that the first and last vertices traversed on the entire Bézier patch are always the same, regardless of the tessellation level.

Preserving both inter- and intra-patch coherency results in a tessellation path that is continuous and not self-intersecting. In addition, the path will traverse all vertices in a subpatch before continuing onto a neighbouring patch. Therefore if the texture for two subpatches are contained in separate cache blocks in memory, the rasterization process will only require a single cache swap.

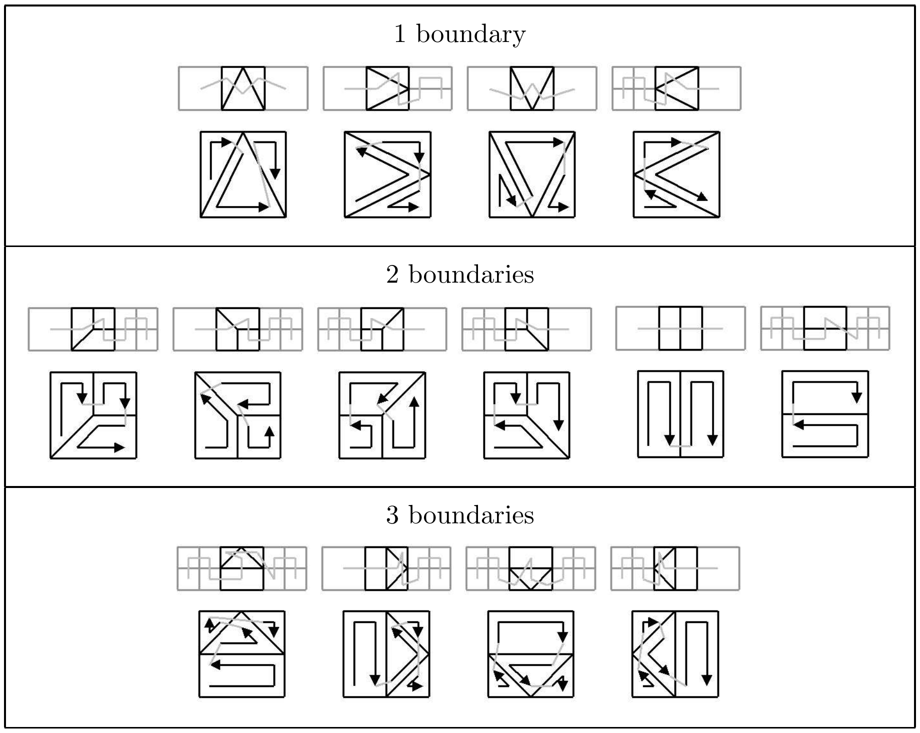


Table 3.1: Tessellation orders for all possible tessellation cases.

There is a different tessellation order for every adaptive case and for every possible direction. In Table 3.1, we list the individual cases. We try to draw the adaptive tessellation of a triangular patch in an order that preserves the memory coherency properties of the triangular space-filling curve in all tessellation cases. When only 1 boundary requires refinement on a quadrilateral patch, all the tessellations begin and end on the first and last vertex of the parent quadrilateral, allowing for easy integration with other tessellations. When 2 or 3 boundaries need to be refined, the curve may not necessarily end at the bottom right corner, resulting a slight loss of inter-patch coherence. However, in one of the 2-boundary conditions, the tessellation stops on an edge that will not be subdivided further, so loss of coherence will be small.

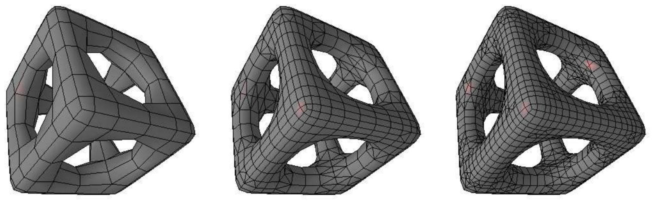


Figure 3.16: A model being adaptively tessellated using the arc length criterion.

3.3 Hardware-friendly implementation

Although the number of different cases for adaptive tessellation may seem intimidating at first, an implementation is actually not very complex. For every subpatch we create a bit vector describing which edges need to be subdivided and which don't. This bit vector references a lookup table with sixteen entries for the quadrilateral case and 8 entries for the triangular case.

Every table entry stores the subpatches that need to be created in the order required according to Section 3.2. Every subpatch is specified in terms of the vertex positions (in parameter space relative to the current level). Sub-patch orientation is implicit in the order that the vertices are listed in the table.

For example, suppose we are tessellating a quadrilateral patch where the first boundary requires tessellation and the second, third, and fourth boundaries do not. The corresponding bit vector for such a patch would be 1000 in binary notation or 8 in decimal notation. Entry 8 in the lookup table will contain information describing how the surface should be tessellated.

With this table-based approach, the evaluation of the surface points, computation of the adaptive subdivision criteria, and computation of the next subdivision level are easily implemented given the feature sets of vertex shaders for current GPUs. The fundamental missing feature from current PC-based GPUs is the ability to generate new triangles in a vertex program, although the graphics chips of some game consoles such as the Playstation 2 do support this. This feature will soon be making its way onto the next generation of PC-based GPUs and is already supported by DirectX 9.0.

Chapter 4

Results

To evaluate the performance of the proposed method, we compare the number of polygons generated by the different adaptive tessellation criteria with different threshold values. We also analyze spatial coherence of the polygon ordering both visually and quantitatively.

4.1 Polygon counts

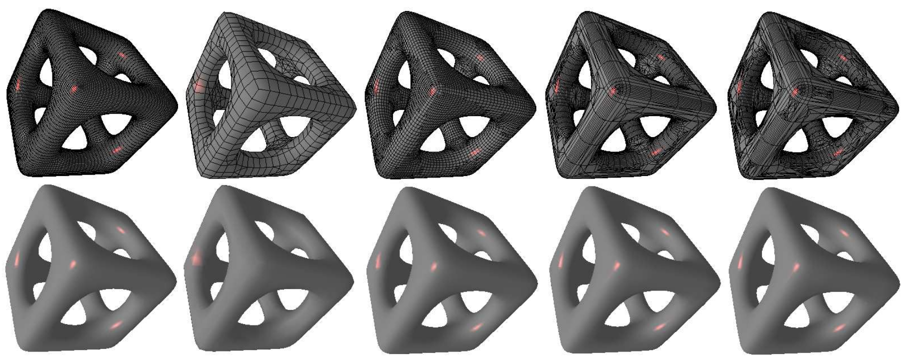


Figure 4.1: A comparison of a model tessellated using different tessellation criteria. The criteria used are, from left to right, no criteria (uniform tessellation), the edge length criterion, the arc length criterion, the straightness criterion, and the tangent criterion.

Table 4.1 shows a comparison of the number of polygons generated by the various methods for the cube with holes composed of 288 top-level patches (compare Figure 4.1). The columns of the table correspond to different threshold values for the different edge subdivision criteria. The $\delta = 0$ column corresponds to uniform tessellation. As expected, we can see that the adaptiveness of the subdivision can result in a drastically reduced number of polygons generated over uniform tessellation.

The straightness criterion performs best with this model because it contains a variety of flat and highly-curved surfaces. The tangent criteria also performs well, but does not evaluate the angle around the boundary midpoint until an additional level of tessellation is performed. This explains why the tessellation with $\delta = 10$ at tessellation level $\ell = 5$ using the tangent criteria contain approximately 4 times more polygons (i.e. undergo about 1 more level of tessellation) than the tessellations using the straightness criterion.

4.2 Texture accesses

Figure4.2 shows the resulting tessellations and polygon orders both in 3D and in the parameter domain for a simple blob object. From visual inspection, it is easy to see that the polygons created close together in time are also grouped together spatially. This is the property we are after for memory coherence during texture lookup and framebuffer writes.

To quantitatively assess the gain in coherence, we evaluated the number of changes between texture tiles as we render the generated tessellation in space-filling order (see Table 4.2.) The texture over each face consists of 4 texture tiles. We compare this against a uniform scanline order tessellation (leftmost column). The results confirm Voorhies' earlier findings of superior performance for space filling traversal orders [37].

	ℓ	$\delta = 0$	$\delta = 1$	$\delta = 2$	$\delta = 5$	$\delta = 10$
T_ℓ	1	1152	1152	1152	1152	792
	3	18432	18408	15992	5760	1464
	5	294912	120502	31674	6192	1464
T_a	1	1152	1152	1152	1152	1152
	3	18432	18432	18432	18432	10688
	5	294912	294912	294912	163982	17644
T_s	1	1152	1056	1056	936	840
	3	18432	13594	12164	7296	1896
	5	294912	134222	53684	9366	1896
T_t	1	1152	1056	1056	986	840
	3	18432	13038	11326	7484	2706
	5	294912	120354	41308	12582	4789

Table 4.1: Number of polygons generated for a model of a cube with holes in it consisting of 288 Bézier patches at different tessellation levels using the different tessellation criterion with varying thresholds. The threshold δ is in angles for criteria T_s and T_t, pixels for T_a, and percentage of model width for T_ℓ.

4.3 Point Splatting

Table 4.3 shows the number of vertices drawn to render the ball and cube with holes model with simple 5-pixel point splats. Both uniform and adaptive splatting the models using different tessellation criteria are compared. The tessellation stopping thresholds are adjusted to be the largest values possible without creating any gaps in the tessellation. Some of the results are shown in Figure 4.3. The tessellation created using the arc length criterion results in a relatively uniform distribution of points in screen space. This makes the arc length criterion better suited to point-splatting than the edge length criterion where the distribution is denser in the parts of the model further away from the

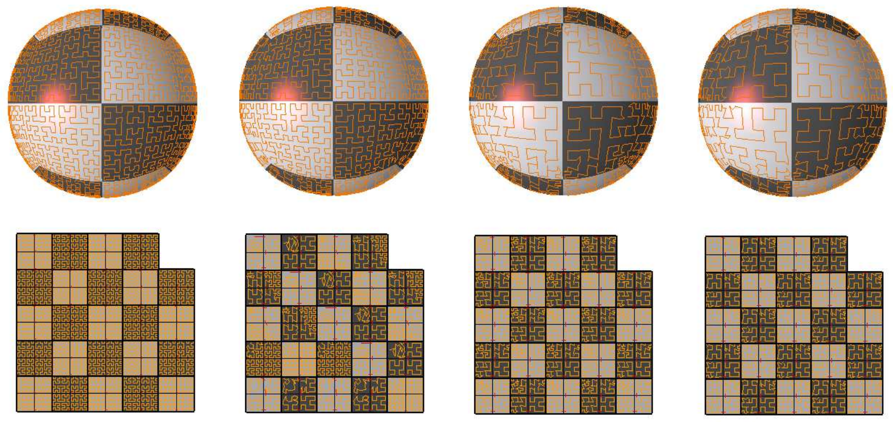


Figure 4.2: The space-filling curves drawn over a ball model tessellated with maximum tessellation level $\ell = 3$ (left) and the corresponding pattern of texture accesses in texture space (right). The model is tessellated using no (top), arc length (second), straightness (third), and tangent (bottom) tessellation criteria. Each square in the texture space represents one texture tile. Red line segments indicate texture accesses that require loading a new texture tile into memory.

camera.

The straightness and tangent criteria are clearly unsuitable for point splatting. On the cube with holes model, the flat surfaces are never tessellated under these criteria unless the threshold is set to the lowest possible value, 0. Adaptive tessellation with the arc length criterion performs better than adaptive tessellation with the edge length criterion and uniform tessellation because it takes into account perspective foreshortening (as seen in Figure 4.3).

4.4 Tessellations of complex models

Finally, we show several examples of more complex models tessellated with our method in Figures 4.4 and 4.5.

	ℓ	$\delta = 0$	$\delta = 1$	$\delta = 5$	$\delta = 10$
uniform	1	576	576	576	576
(top/bottom,left/right)	3	2880	2880	2880	2880
	5	6720	6720	6720	6720
uniform	1	95	95	95	95
(space-filling)	3	383	383	383	383
	5	1247	1247	1247	1247
T_a	1	72	72	72	56
	3	380	380	376	68
	5	1200	1200	928	68
T_s	1	72	72	72	72
	3	360	360	360	216
	5	1224	1224	1080	264
T_t	1	72	72	72	72
	3	360	360	360	360
	5	1224	1224	984	360

Table 4.2: Comparison of the number of times a new texture tile is loaded into memory when rendering a textured ball model using a uniform left-right, top-bottom tessellation order versus our algorithm at different subdivision levels. Low numbers indicate higher cache coherence.

model	ℓ	uniform	T_e	T_a	T_s	T_t
ball	5	98304	53567	28184	91152	88416
cube with holes	4	294912	155308	89786	294912	294912

Table 4.3: Comparison of the number of vertices drawn to render the ball and cube with holes model with simple 5-pixel point splats, while tessellating the models using different tessellation criteria. $\delta_\ell = 0.02$, $\delta_a = 3.11$, $\delta_s = 1.2$, and $\delta_t = 1.1$ for the ball model and $\delta_\ell = 0.15$, $\delta_a = 9.2$, $\delta_s = 0$, and $\delta_t = 0$ for the cube with holes model.

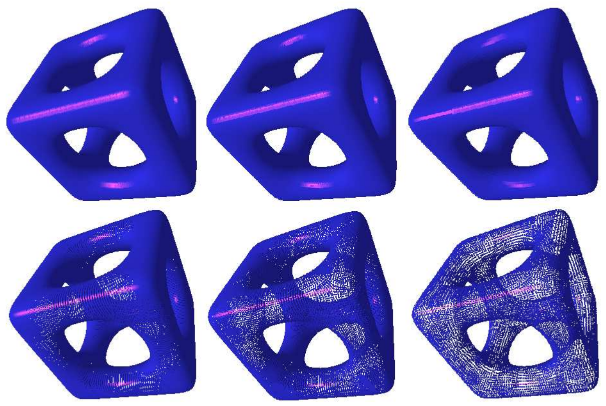


Figure 4.3: An example of a model rendered using simple point splats. *From left to right:* uniform tessellation, adaptive tessellation with the edge length criterion and with the arc length criterion. The model with 5-pixel point splats is shown in the top row and the same model with 2-pixel point splats is shown in the bottom row.

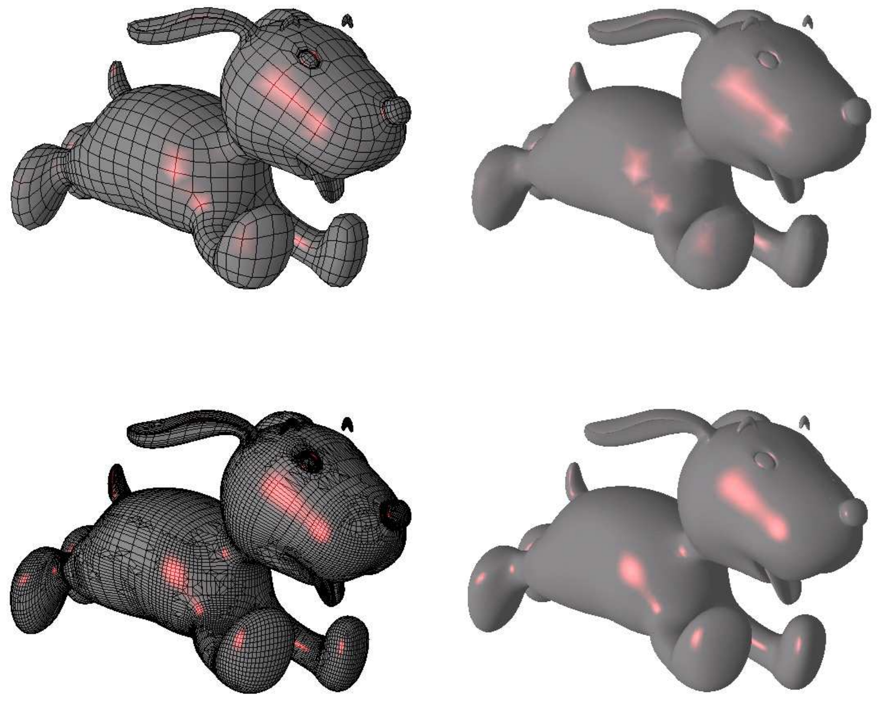


Figure 4.4: A cartoon dog modelled with Bézier surfaces. *Top:* The model at tessellation level $\ell = 0$. *Bottom:* The model at tessellation level $\ell = 3$.

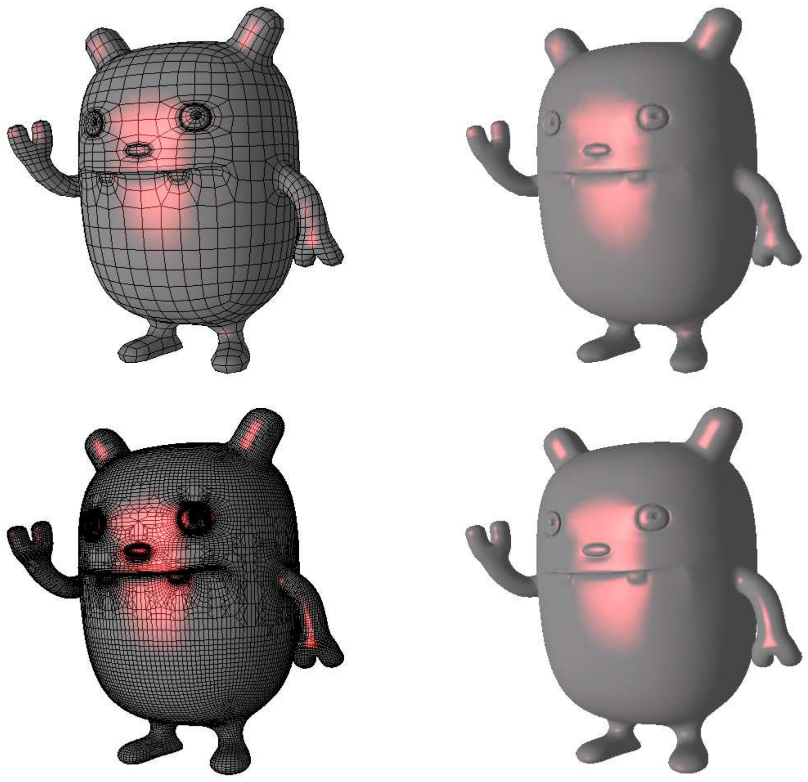


Figure 4.5: *Top:* The model at tessellation level $\ell = 0$. *Bottom:* The model at tessellation level $\ell = 3$. The improved visual quality of the left arm is especially noticeable.

Chapter 5

Conclusions and Future Work

5.1 Summary of research

In this thesis we have presented an adaptive, depth-first tessellation algorithm for smooth surfaces suitable for implementation on a GPU. In our particular implementation, we have chosen to process only cubic tensor-product Bézier patches, but other representations are possible. We chose Bézier patches since they allow us to derive a smooth object representation composed of many continuous patches.

Our tessellation algorithm avoids inconsistencies in the resulting mesh by basing the subdivision decision purely on boundary information. In this way, adjacent patches independently arrive at the same decision for the boundary they share. The adaptive algorithm will generate quadrilaterals as well as triangles, which is enabled by not explicitly generating and storing the control meshes for the individual parts of a subdivided patch.

One restriction of boundary-based subdivision is that it is possible to construct top-level patches with very short edge lengths that nonetheless have a large surface area, simply by locating the boundary control points close to each other, but pulling the center control points out by a certain distance. Such patches do not occur often in practice, and even if encountered, an artist can easily work around this restriction simply by subdividing the patch once. We therefore do not believe that this limitation poses a serious restriction in practice.

In addition to these advantages, a simple modification of the traversal order ensures that the polygons of the final tessellation are generated in an order corresponding to a space-filling curve. This ensures coherence in both image and texture space, and therefore helps to alleviate performance problems arising from memory accesses to the graphics RAM, which are often a bottleneck on modern GPUs. The adaptive tessellation in space-filling order is based on a thorough analysis of the different cases that can be encountered, a simple implementation using lookup tables has been described. This approach should be amenable to hardware implementations on the tessellator unit on future GPUs.

5.2 Satisfaction of Research Objectives

Our algorithm satisfies all our research objectives.

- It provides hole-free locally and globally adaptive surface tessellation without requiring any information on the neighbouring surfaces by using a boundary-based tessellation criteria.
- It has low GPU-to-graphics memory bandwidth requirements because it draws the tessellation polygons in an order that preserves the memory coherence properties of a space-filling curve.
- It uses little GPU memory because it stores patches in parametric form instead of calculating and storing new control meshes for every tessellation level-of-detail.

5.3 Future work

In principal we could apply the same subdivision method not only to smooth surfaces, but also to polygonal representations. This way it would be possible to create a scan-conversion engine that hierarchically subdivides polygons or patches into sub-pixel-sized micro polygons, and then renders these using point splatting. This would be similar in spirit to the original Reyes architecture [10], but due to the specific traversal order, the memory access patterns should be significantly improved.

Bibliography

[1] Salim S. Abi-Ezzi. *The graphical processing of B-splines in a highly dynamic environment.* PhD thesis, Rensselaer Polytechnic Institute, 1990.

[2] Kurt Akeley. Reality engine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 109–116, 1993.

[3] Stephan Bischoff, Leif P. Kobbelt, and Hans-Peter Seidel. Towards hardware implementation of Loop subdivision. In *Proceedings of the 2000 EUROGRAPHICS/SIGGRAPH workshop on Graphics Hardware*, pages 41–50. ACM, 2000.

[4] Jeffrey Bolz and Peter Schröder. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Proceeding of the 7th International Conference on 3D Web Technology*, pages 17–11. ACM, 2002.

[5] Montserrat Bóo, Margarita Amor, Michael Doggett, Johannes Hirche, and Wolfgang Strasser. Hardware support for adaptive subdivision surface rendering. In *SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware 2001*, pages 33–40, 2001.

[6] David Brickhill. Practical implementation techniques for multi-resolution subdivision surfaces. In *Proceedings of the 2001 Game Developers Conference*, 2001.

[7] Edwin Catmull and James Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10(6):350–355, 1978.

[8] James H. Clark. A fast scan-line algorithm for rendering parametric surfaces. *ACM SIGGRAPH Computer Graphics*, 13(2):174, 1979.

[9] James H. Clark. The geometry engine: A VLSI geometry system for graphics. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, pages 127–133, July 1982.

[10] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987.

[11] Tony DeRose, Mary L. Bailey, Bill Barnard, Robert Cypher, David Dobrikin, Carl Ebeling, Smaragda Konstantinidou, Larry McMurchie, Haim Mizrahi, and Bill Yost. Apex: two architectures for generating parametric curves and surfaces. *The Visual Computer*, 5(5):264–276, 1989.

[12] Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *Proceedings of SIGGRAPH '98*, pages 85–94. ACM, 1998.

[13] Michael Doggett and Johannes Hirche. Adaptive view dependent tessellation of displacement maps. In *Proceedings of SIGGRAPH 2000*, pages 59–66. ACM, 2000.

[14] Donald Doo. A subdivision algorithm for smoothing down irregularly shaped polyhedrons. In *Proc. Interactive Techniques in Computer Aided Design 1978*, pages 157–165, 1978.

[15] Daniel Filip, Robert Magedson, and Robert Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3(4):295–311, 1987.

[16] David R. Forsey and Robert Victor Klassen. An adaptive subdivision algorithm for crack prevention in the display of parametric surfaces. In *Proceedings on Graphics interface '90*, pages 1–8. Canadian Information Processing Society, 1990.

[17] Bin Han, Michael L. Overton, and Thomas P.-Y. Yu. Design of hermite subdivision schemes aided by spectral radius optimization. *Society for Industrial and Applied Mathematics Journal of Scientific Computing*, 25(2):643–656, 2003.

[18] Robert Victor Klassen. Integer forward differencing of cubic polynomials: Analysis and algorithms. *ACM Transactions on Graphics*, 10(2):152–181, 1991.

[19] Leif Kobbelt. $\sqrt{3}$ subdivision. In *Proceedings of SIGGRAPH 2000*, pages 103–112. ACM, 2000.

[20] Frans Kuijt and Ruud van Damme. Hermite-interpolatory subdivision schemes. Technical report, University of Twente, Faculty of Mathematical Sciences, 1998.

[21] Subodh Kumar, Dinesh Manocha, and Anselmo Lastra. Interactive display of large-scale nurbs models. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 51–58. ACM, 1995.

[22] Aaron Lee, Henry Moreton, and Hugues Hoppe. Displaced subdivision surfaces. In *Proceedings of SIGGRAPH 2000*, pages 85–94. ACM, 2000.

[23] Sheue-Ling Lien, Michael Shantz, and Vaughan Pratt. Adaptive forward differencing for rendering curves and surfaces. In *Proceedings of SIGGRAPH 87*, pages 111–118. ACM, 1987.

[24] Charles Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Department of Mathematics, 1987.

[25] Michael McCool, Chris Wales, and Kevin Moule. Incremental and hierarchical hilbert order edge equation polygon rasterization. In *Proc. of Graphics Hardware, 2001*, 2001.

[26] Henry Moreton. Watertight tessellation using forward differencing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 25–32. ACM, 2001.

[27] Heinrich Müller and Reinhard Jaeschke. Adaptive subdivision curves and surfaces. In *Proceedings of Computer Graphics International '98*, pages 48–58. ACM, 1998.

[28] Alyn P. Rockwood. A generalized scanning technique for display of parametrically defined surfaces. In *IEEE Computer Graphics and Applications*, pages 15–26. IEEE, 1987.

[29] Alyn P. Rockwood. Real-time rendering of trimmed surfaces. In *IEEE Computer Graphics and Applications*, pages 15–26. IEEE, 1987.

[30] Hans Sagan. *Space-Filling Curves.* Springer-Verlag, 1994.

[31] I. Schoenberg. On the Peano curve of Lebesgue. *Bulletin of the American Mathematical Society*, 44(519), 1938.

[32] Michael Shantz and Sheue-Ling Chang. Rendering trimmed NURBS with adaptive forward differencing. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 189–198. ACM, 1988.

[33] Le-Jeng Shiue, Ian Jones, and Jorg Peters. A realtime GPU subdivision kernel. In *Proceedings of SIGGRAPH '05*. ACM, 2005.

[34] Jos Stam and Charles Loop. Quad/triangle subdivision. *Computer Graphics Forum*, 22(1), 2002.

[35] Luiz Velho and Denis Zorin. 4-8 subdivision. *Computer-Aided Geometric Design: Special Issue on Subdivison Techniques*, 18(5):397–427, 2001.

[36] Alex Vlachos, Jorg Peters, Chas Boyd, and Jason L. Mitchell. Curved PN-triangles. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 159–166. ACM, 2001.

[37] Douglas Voorhies. *Graphics Gems II*, chapter Space-Filling Curves and a Measure of Coherence. Academic Press, 1991.

[38] Denis Zorin and Peter Schroder. A unified framework for primal/dual quadrilateral subdivision schemes. *Computer Aided Geometric Design*, 18(5):429–454, 2001.