Watertight Tessellation using Forward Differencing

Henry Moreton[1]
NVIDIA Corporation

ABSTRACT

In this paper we describe an algorithm and hardware for the tessellation of polynomial surfaces. While conventional forward difference-based tessellation is subject to round off error and cracking, our algorithm produces a bit-for-bit consistent triangle mesh across multiple independently tessellated patches. We present tessellation patterns that exploit the efficiency of iterative evaluation techniques while delivering a defect free adaptive tessellation with continuous level-of-detail. We also report the rendering performance of the resulting physical hardware implementation.

CR Categories and Subject Descriptors: I.3.1[Computer Graphics]: Hardware Architecture – Graphics processors; I.3.5[Computer Graphics]: Computational Geometry – Curve, surface, solid, and object representations; splines

Additional Keywords: CAD, Curves & Surfaces, Geometric Modeling, Graphics Hardware, Hardware Systems, Rendering Hardware

1 INTRODUCTION

The desire or need to hardware-render a higher order representation such as a Bézier patch stems from three areas of strength: animation, level of detail, and bandwidth. A primitive with the right characteristics permits an application to easily animate a large amount of geometry/visual complexity. Manipulating a small number of high-level controls (e.g., control points) can cause broad-scale changes. During the rendering process, on most contemporary architectures, higher-level primitives are converted into triangles. Since there is freedom in choosing the sampling density, the tessellation may be easily scaled based on the current view or desired frame rate. Finally, current high performance PC graphics accelerators are capable of drawing roughly 40 million triangles per second, fully saturating the standard interface (AGP4x) if specified with perfect efficiency. A higher-order primitive can be regarded as a compressed representation for a collection of triangles, and thus consumes less bandwidth. In reality, realized AGP bandwidth is often 50-75% of peak, and triangle mesh specification is rarely close to 24 bytes per triangle. So, with higher order primitives there is a real opportunity to deliver increased performance, and a richer visual experience.

[1]270 San Tomas Expressway, Santa Clara, CA 95050, moreton@nvidia.com

HWWS '01 Los Angeles, CA USA

Surface rendering may be invoked using standard APIs, such as OpenGL® [22] and DX8™. While OpenGL has a history of rendering Bézier patches, only the recent DX8 version introduces higher order primitives to D3D™. In both cases, the primitives are converted to triangles at the top of the graphics pipeline; the remainder of the pipeline is oblivious to the source of triangles, the application or a surface tessellator. This has the advantage that it is possible to insulate the rest of the design from the tessellator. As an aside, the primary disadvantage of this placement is that a vertex program[1] cannot modify the control points of a patch. As a result, blending and morphing must be performed by the application.

The *evaluator* functionality provided by the standard OpenGL interface and its typical implementations have several shortcomings. In the following, we discuss four shortcomings, contrasting them with our interface and implementation.

1. The interface only supports tensor product patches; models exported from modern modeling packages freely mix triangular and tensor product patches. Our implementation supports both triangular and tensor product patches.

2. The current OpenGL interface provides two mechanisms to generate triangles from patches. A mesh operation produces a rectangular grid of triangles, and a point evaluation routine permits the generation of individual triangles. This combination supports rendering of trimmed surfaces, triangular patches, and some limited adaptive tessellation. To avoid cracks these two mechanisms must be numerically "equivalent". This requirement in turn limits opportunities for efficient evaluation. Our programming interface uses a single primitive to invoke the underlying algorithms and hardware. The interface permits the application to specify flexible tessellation with an *independent* tessellation factor for each of the patch's sides. This is in contrast to a uniform tessellation where a regular grid of triangles is produced. The details of the interface are available in an OpenGL extension [16].

3. When using the current OpenGL mesh primitive, the tessellation factors are specified as integers. While integers seem the logical choice, using a floating-point specification for these tessellation factors permits the underlying implementation to render a mesh varying continuously in density, which avoids visual popping.

4. Most hardware implementations of evaluators have shared floating-point resources in the graphics subsystem with vertex transform, lighting, and clipping. The result is an implementation with little additional cost beyond the program memory for the tessellation code. However, the performance achieved by these implementations could never approach the performance of the same platform directly rendering triangles. Our implementation uses a dedicated unit. By splitting the computation between the driver running on the host and an on-chip processor we are able to tessellate surfaces without degrading the downstream triangle performance.

[1] "Vertex shader" in DX8

The remainder of the paper is organized as follows. We survey the previous work on surface tessellation and rendering. We compare polynomial evaluation methods. We describe tessellation patterns that permit the efficient use of iterative evaluation techniques and support continuous LOD. We present simple methods for computing tangent and normal patches, necessary for lighting purposes. We describe our approach to the support of triangular patches. We present the details of algorithms necessary for the generation of consistent, defect-free surface tessellations. And close with preliminary performance results from working hardware.

2 PREVIOUS WORK

Previous work on surface rendering can be divided into two categories, those algorithms that render the surface directly and those that generate an intermediate representation, triangles.

The first category is made up of exhaustive subdivision schemes [3], scanline renderers [11], ray tracers [9] and the isoparametric scanning algorithms [12][20]. Obviously, these approaches are not able to take significant advantage of huge industry investment in efficient triangle rendering hardware. Pulleyblank [18] proposes hardware acceleration of exhaustive subdivision. Beyond inadequate performance for real-time rendering, the iso-scanning algorithms often draw the same pixel multiple times causing problems with frame buffer blending operations, and parity dependent schemes such as stencil-based shadow volumes [7].

The second category is made up of tessellation algorithms. These schemes adaptively tessellate a surface based on the characteristics of a surface, or an application supplied specification. The first of these are the scanline algorithms of Lane and Carpenter [11] and subsequently Clark [4]. Referring to Figure 1 we see a pair of adjoining patches (a), the red patch requires further subdivision, the gray is sufficiently tessellated. Lane and Carpenter divide the red patch while rendering the gray using a single quad. In (b) we see the crack that results from differing levels of tessellation; a solution to this problem is described in [4]. The red vertex forming the crack is forced to the green edge of the gray quad. In (c) we see the pixel dropouts that result from the "T" junction formed. Because the location of a vertex is represented using finite precision, the vertex does not actually lie on this segment. Although this problem is exacerbated by the low precision of some hardware rasterizers, it exists for ***any*** finite precision representation, including IEEE floating point. Note that the only way to guarantee a flawless rendering is through precise representation of relationships; vertices that are logically equal must be exactly equal. Our algorithms produce crack-free tessellations completely independent of the precision of the evaluation techniques.

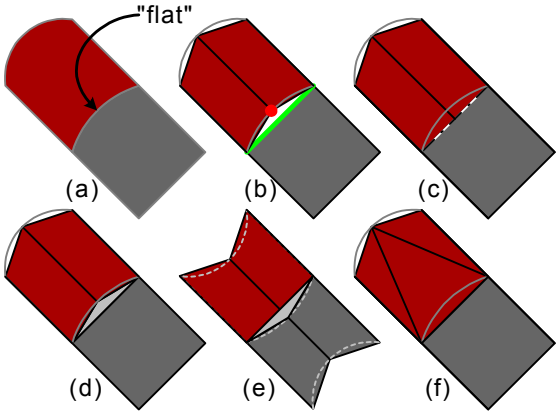

Figure 1: Adaptive tessellation solutions & problems.

In Figure 1(d) we see the introduction of an interstitial triangle filling the crack. In Figure 1(e) we see an example of the problem with this approach, a pair of patches concave relative to the viewer. Both patches incident to the flat edge require refinement, but also generate a triangle to fill the potential crack. The result is the light gray fin sticking up. A similar alternative is to generate an interstitial triangle to fill the "crack" caused by finite precision in Figure 1(c). The triangle fills the gap even though it is geometrically zero area. This is still inadequate, while we have filled any possible pixel dropouts, double hits may occur, and possible shading artifacts may result from the inconsistent sampling.

Figure 1(f) illustrates the general solution to these problems, explicit triangulation or meshing, free of degenerate triangles. In [21], Rockwood et al. uniformly tessellate the interior of each patch into a grid of rectangles whose density is selected to satisfy a user specified tolerance. These rectangles are connected by triangles to points on the patch boundary (coving). OpenGL evaluators were used to render the parameter space triangulation. In [10], Kumar et al. use a similar tessellation algorithm. Instead of using evaluators they directly render triangles. By using an incremental retessellation algorithm they are able to achieve performance superior to Rockwood's. Neither Rockwood et al nor Kumar et al support continuous level of detail, and Kumar's scheme consumes considerable bandwidth and CPU memory. Lastly, as noted in the introduction the current OpenGL evaluator interface has some inherent performance limitations.

Bruijns [2] uses forward differencing to adaptively tessellate quadratic Bézier patches. Regular interior meshes are heuristically stitched to an independently sampled perimeter. This scheme does not support continuous LOD, and does not take measures to avoid problems due to round-off error resulting from forward differencing. Bischoff et al. [1] propose a scheme for rendering Loop subdivision surfaces [15]. Regular (polynomial) sub-patches are evaluated using forward differencing. They make no provisions for flexible tessellation, continuous level of detail, or defects from round off. Pulli and Segal [19] also present a method for Loop subdivision surface triangulation using geometry engine processors. While efficient, their scheme has no provision for flexible tessellation or continuous level of detail. Finally, Vlachos et al [23] describe patches called *PN triangles*. These patches derive their shape from triangles with normals specified at the corners; they may be rendered without tessellation. When refined they are converted to Bézier triangles and uniformly tessellated. Their primary drawbacks are the inability to naturally express creases, and their lack of flexible tessellation and continuous LOD.

3 POLYNOMIAL EVALUATION

We compare two common polynomial evaluation schemes, forward differencing and De Casteljau [5]. De Casteljau uses nested linear interpolation to evaluate a univariate polynomial in $O(n^2)$ operations, where n is the degree of the polynomial. In Figure 2 we provide a geometric interpretation of De Casteljau for evaluating a cubic curve, and a bi-quadratic patch. This technique is very stable, precise, and supports evaluation at arbitrary parameter values.

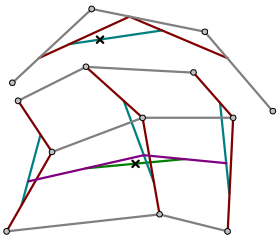

Figure 2: De Casteljau polynomial evaluation.

In contrast, forward differencing uses additions to evaluate a polynomial, and is $O(n)$. In Figure 3 we illustrate the evaluation of a cubic polynomial at a fixed parametric interval of 1/5. Once the first four values are computed, each further value may be computed using three additions.

This technique is very efficient, however it is subject to round-off error and is limited to sampling a polynomial at a fixed parametric interval:

$$\{t^3 \quad t^2 \quad t \quad 1\} \cdot \mathbf{B} \cdot \begin{Bmatrix} a \\ b \\ c \\ d \end{Bmatrix}$$

where $t = jt_{\text{step}}, j \in Z$, $\mathbf{B}$ is a basis matrix, and $\{a,b,c,d\}^T$ are polynomial coefficients relative to $\mathbf{B}$. We can compute the forward difference coefficients of Figure 3 using the following formula, where Δ^r is the iterated forward difference operator [5]:

$$\begin{Bmatrix} p \\ \Delta^1 p \\ \Delta^2 p \\ \Delta^3 p \end{Bmatrix} = \begin{Bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 6 & 2 & 0 & 0 \\ 6 & 0 & 0 & 0 \end{Bmatrix} \cdot \begin{Bmatrix} t_{\text{step}}^3 & 0 & 0 & 0 \\ 0 & t_{\text{step}}^2 & 0 & 0 \\ 0 & 0 & t_{\text{step}} & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix} \cdot \mathbf{B} \cdot \begin{Bmatrix} a \\ b \\ c \\ d \end{Bmatrix}$$

or $$\begin{Bmatrix} p \\ \Delta p \\ \Delta^2 p \\ \Delta^3 p \end{Bmatrix} = \mathbf{M}_3 \cdot \mathbf{P} \cdot \mathbf{B} \cdot \begin{Bmatrix} a \\ b \\ c \\ d \end{Bmatrix}$$

For general degree $\mathbf{M}_{i+1} = \begin{Bmatrix} 0 & \mathbf{M}_i \\ \mathbf{m}_{i+1} & \mathbf{0}^T \end{Bmatrix}$, $\mathbf{m}_3 = \{1 \quad 6 \quad 6\}^T$,

$\mathbf{m}_4 = \{1 \quad 14 \quad 36 \quad 24\}$, and $\mathbf{m}_5 = \{1 \quad 30 \quad 150 \quad 240 \quad 120\}$.

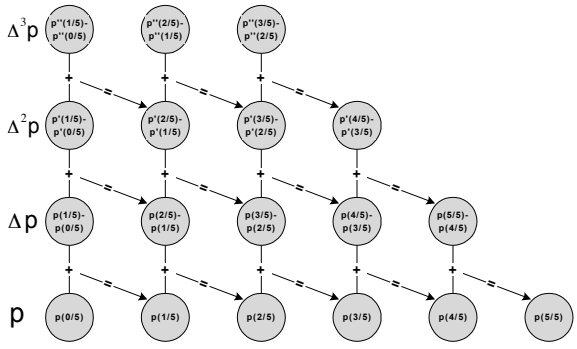

Figure 3: Forward difference evaluation.

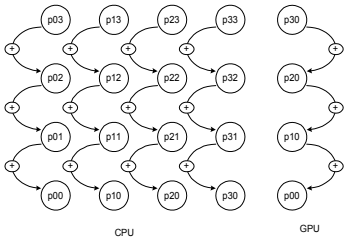

Figure 4: Tensor product forward differencing.

When operating on a tensor product surface there is a natural division of labor between the host CPU and the GPU (Figure 4). When forward differencing is applied in tensor product form, the CPU steps a matrix along, producing a series of evenly spaced *curves*. The bottom row of the matrix is extracted and handed off to the GPU, which in turn iterates it producing a series of evenly spaced points. Because our tessellation algorithms are impervious to round-off error, patches need not be broken into subpatches to avoid errors when heavily tessellated (see section 8).

4 HARDWARE

By dividing the work of tessellation between the host CPU and the graphics card, the amount of chip area consumed by surface tessellation may be kept quite small, making it feasible to have a dedicated tessellation engine running in parallel with other geometry operations, such as transform and lighting. In Figure 5 we provide a block diagram of the engine. It is made up of four banks of memory 32x128 bits each. Each memory can supply one operand to the 4x32-bit floating-point adder. The adder stores its results back into one of the two source operand memory banks. During normal operation one pair of banks is used for the calculation of a series of points on a curve while at the same time the other is loaded with the coefficients for the next curve.

In Figure 6 we illustrate how a mesh of triangles is calculated and drawn. First a row of vertices (0-4) is computed and stored in RAM. Then the second row of vertices (5-9) is calculated and stored in the same RAM overwriting the previous row, while generating two triangles per vertex calculated, in the interior of the mesh. For increased performance the surface tessellator relies on the ability to calculate a vertex, use it to draw a triangle, and then reuse the vertex without recalculation in a subsequent triangle. These vertices are stored in this memory after being transformed and lit. *Euler's Formula for Polyhedra,* [8] $V - E + F = 2$, tells us that, in a closed mesh of triangles, there are two triangles per vertex. However we cannot store an

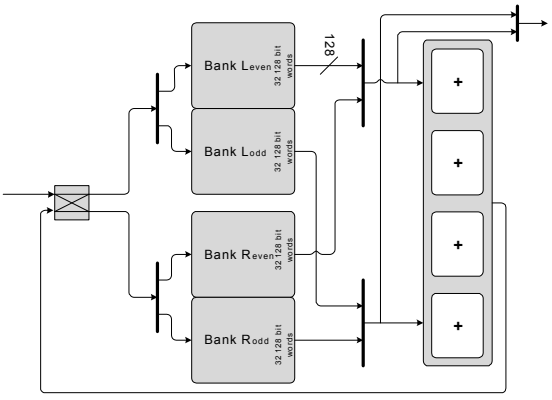

Figure 5: Diagram of forward differencing hardware.

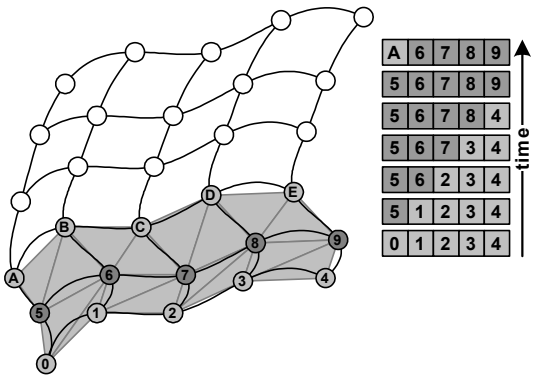

Figure 6: Mesh generation and vertex reuse.

unbounded number of vertices. We have chosen a memory size that represents a good cost/performance tradeoff, and are able to render approximately 1.8 triangles per vertex produced. A complication of this finite memory is that we must break heavily tessellated patches up into *swaths* of triangles that are sufficiently narrow that a row of vertices fit in this memory.

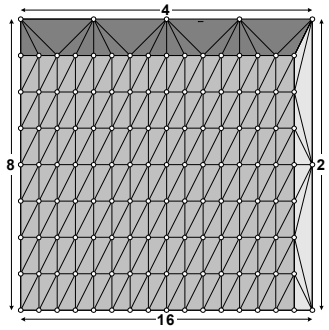

Figure 7: Integer Tessellation Pattern (2,4,8 &16 segments).

5 TESSELLATION PATTERNS

Given an engine that can very efficiently tessellate a regular grid, and generate a series of points on a curve, how do we get the engine to support the independent tessellation factors, and continuous LOD described in the introduction? Our engine supports two patterns of tessellation, *integer* and *fractional*. The integer pattern produces an optimal triangulation, generating a constrained Delaunay triangulation [17] of the irregular portions of the mesh. The fractional style supports continuous level of detail.

5.1 Integer Tessellation

In the integer case a regular mesh is generated covering the majority of the patch. The number of rows in the mesh is one less

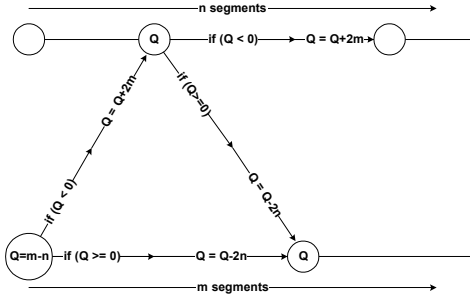

Figure 8: Integer transition stitching state machine.

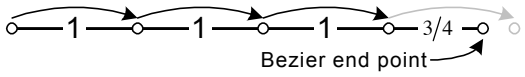

Figure 9: Evaluating a curve with tessellation factor 3.75.

than the greater of the two row tessellation factors; the number of columns is set similarly. In Figure 7 the resulting regular mesh has 7 rows and 15 columns.

Transitions are drawn to fill in the remaining row and column. A state machine in the hardware curve unit controls the stitching of each transition region. Similar to Bresenham's algorithm for drawing lines, a state variable Q is initialized to the difference of the tessellation factors (6 in the vertical transition of Figure 7.) Then depending on the sign of Q the engine generates a triangle by advancing along either of the two sides of the transition. The transition is filled with a set of triangles of optimal shape in the parameter space of the patch.

5.2 Fractional Tessellation

The so-called integer tessellation pattern breaks the mesh of triangles into a combination of a regular mesh and transition regions. The fractional tessellation scheme also uses a mixture of regular meshes and transition regions. The patterns we use are constrained by rules necessary to guarantee a continuous level of detail. Vertices must be introduced at the position of existing vertices. Edges may only be created or destroyed when one of their endpoints is introduced or removed. Vertices must move continuously. These rules result in unavoidable sliver triangles, which can make integer tessellation more desirable in some circumstances.

When performing integer tessellation the application specifies an integer number of segments n and the driver computes differencing coefficients with a step size $1/n$. Note that the n^{th} vertex evaluated corresponds to the end point of the Bézier curve. When doing fractional tessellation we proceed in exactly the same fashion, except that the number of segments m is real. We step $\lfloor m \rfloor$ times and use the Bézier end point as the final vertex, see Figure 9. This approach preserves the efficiency of forward differencing while using the Bézier end point to avoid calculating a vertex at an odd step size.

One problem with this approach is that adjoining patches may not define their shared edges with the same parametric traversal. Consider a Möbius strip, such consistent traversal is impossible, and in general not something with which to burden the application. In Figure 10(a) two such patches share a conflicting boundary. By switching to the symmetric pattern shown in (b) of the figure, we avoid any cracking problems while preserving continuous level of detail and the efficiency of forward

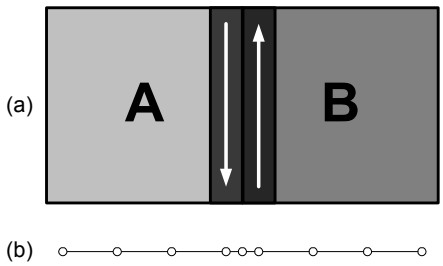

Figure 10: A symmetric tessellation pattern.

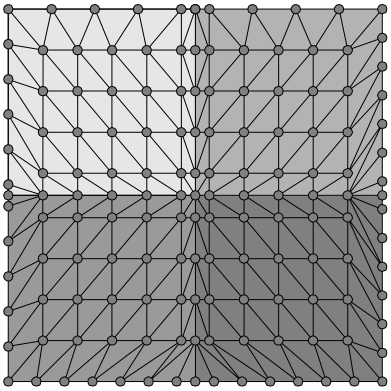

Figure 11: Fractional tensor product tessellation.

differencing.

Extending this pattern of tessellation from a curve to a tensor product patch we arrive at the pattern shown in Figure 11. The original patch is subdivided into four subpatches that are rendered individually performing evaluation from the original patch corners to the midpoints of the sides. With this arrangement, as LOD varies, vertices are introduced in pairs at the midpoints of the patch boundaries. Pairs of rows (or columns) of triangles are introduced forming a cross through the middle of the patch.

6 DERIVATIVES AND NORMALS

The calculation of tangent frames and normals is an integral part of surface tessellation. Many shading algorithms depend on a coordinate frame defined at the vertices of a triangle. To calculate these values the tessellation system simply evaluates additional polynomials representing the two partial derivatives and/or the normal. By subtracting adjacent control points as shown in Figure 13, a patch representing a scaled version of the derivative is created. Since most rendering algorithms are primarily concerned with the directions of the tangent frame axes, this is sufficient. The Bézier control points of a patch representing the normal direction may be computed by taking the cross product of the scaled derivative patches [6]. The CPU, either the application or driver, performs the calculation of derivatives and normals.

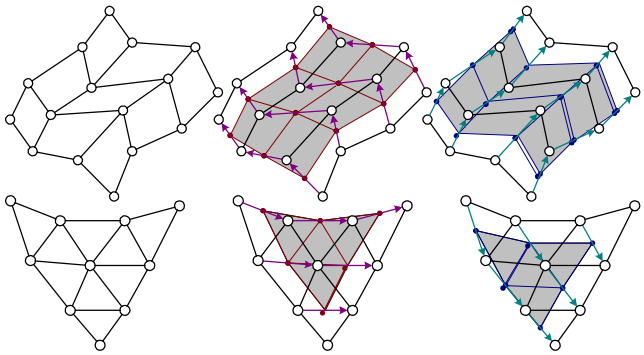

Figure 13: Calculation of tangent patches.

7 TRIANGULAR PATCHES

The algorithm for integer tessellation has an obvious analogous pattern in a triangular domain (Figure 14). However, we chose to handle triangular patches using an approach that had no hardware impact. Through a linear reparameterization a triangular patch of degree k can be converted to a tensor product patch of degree k. This is because triangular patches have total degree k while

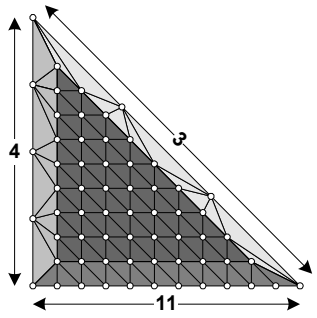

Figure 14: Integer tessellation of a triangular domain.

tensor product patches have total degree $2k$. While there is an increase in total degree, the isoparametric curves remain degree k; they are no more costly to evaluate.

We use two different reparameterizations, one for integer tessellation, and one for fractional, shown in Figure 12. In the integer case the triangular patch is rendered as a degenerate tensor product patch; further details of the reparameterization may be found in [13]. Note that the tangent patches (derivatives) must be computed before reparameterization in order to avoid lighting problems due to the singularity. In the fractional case the triangular patch is divided into three tensor product patches, and each is tessellated using one quadrant of the fractional tessellation pattern for tensor product patches. In this case the reparameterization can be accomplished using Bézier composition techniques [14].

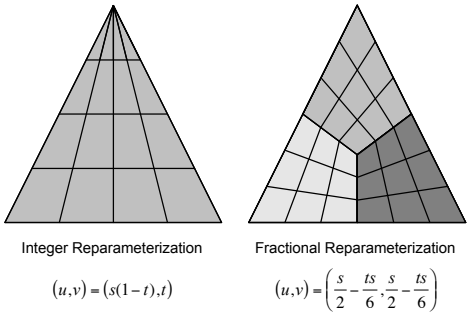

Figure 12: Triangle to tensor product reparameterizations.

8 CONSISTENT MESH GENERATION

In this section we discuss how to guarantee consistent mesh generation while independently rendering multiple patches. We assume that all patches are represented using the Bézier representation, and that any patches sharing a boundary have equivalent control points defining that boundary. The order of the curves sharing the boundary must match, and the positions of the control points defining boundary curve must be exactly equal. A consistent mesh is one where all relationships are expressed exactly, if two triangles in the mesh share an edge then the end points of the edge are equal bit-for-bit. This must hold even if the two triangles originate from different patches. Finally, we must obviously assume that the application has specified consistent tessellation factors.

There are several reasons that consistent mesh generation is not simple. Forward differencing, our evaluation technique, suffers from round-off error when evaluating a long sequence of vertices. Because the implementation divides the work between host and

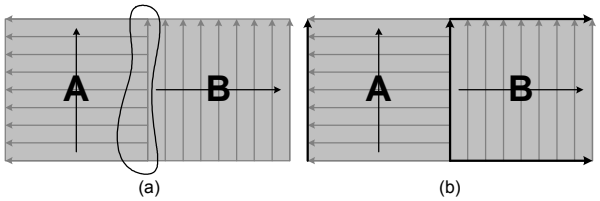

Figure 15: Guard curves for consistency.

graphics card, we are using two different floating-point engines, one in the CPU and one in the GPU. Even if the implementations *were* identical, the same inputs with differing rounding modes yield unequal results. Also, because patches are drawn independently it is not possible to clean up small discrepancies after all tessellation is completed. Finally, it is important to realize that in order to have a guarantee of perfect rendering there can be no errors or inconsistencies, not even a single bit. Calculations are always performed the same way and on the same floating-point unit to ensure consistency.

8.1 Guard Curves

Consider abutting patches **A** and **B** in Figure 15(a), the rows of patch **A** run perpendicular to the rows of patch **B**. There is no way for the patch tessellator to know that this has occurred, and the vertices circled will have inconsistent values, those computed for patch **B** by the GPU will have accumulated round off, and those computed for patch **A** are from the CPU. Our solution is to introduce *guard curves,* evaluated by the GPU and shown in bold in Figure 15(b). In our implementation these curves contain only position and normal attributes, other vertex attributes are permitted to have slight mismatches along the shared border. When the GPU is drawing a mesh, it maintains a pair of guard curves whose values it computes and uses for the beginning and ending of the strip curves. Because all vertices are computed using the same coefficients and the same processor, this mechanism guarantees that the first *strip curve* of patch **B** will match the starting points of each of the strip curves of patch **A**.

8.2 Reversed Transitions

Because forward differencing has an inherent direction of evaluation, the curves on the perimeter of every patch must be traversed in a consistent direction. The direction of traversal is chosen by the driver and is determined by sorting the control points defining the boundary. Because the decision is localized to boundary information the decision will be made consistently every time the curve is encountered. This can result in conflicts see Figure 16(a). The driver has determined the directions of traversal for the curves on the perimeter of the patch, and the top and bottoms curves conflict. Even if the top and bottom tessellation factors agree, a transition is created to resolve the differing directions, a *reversed transition.* The driver must pass the coefficients for traversing the curves in the correct direction. The GPU's tessellator evaluates the inner curve saving results in reverse order, the correct order relative to the outer curve. When the outer curve is evaluated, the transition is stitched up as usual. Reversed transitions may also resolve differences in tessellation factor, their usual function. Note multiple reversed transitions may be required, Figure 16(c).

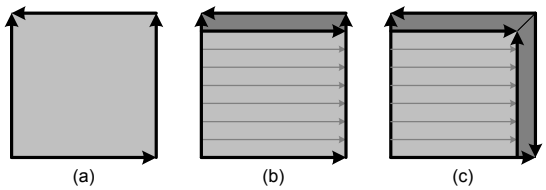

Figure 16: Reversed transitions.

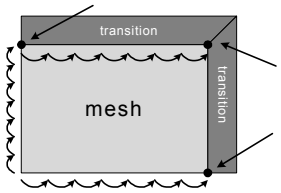

Figure 17: Special Vertices.

8.3 Special Vertices

The mixture of regular meshes and transition regions creates a subtle problem. Because the vertices shown in Figure 17 result from a long chain of calculations during the tessellation of the regular mesh portion of the patch, they are saved by the hardware state machine for subsequent use in the transitions. At most three such vertices must be saved.

8.4 Swatches

As mentioned in section 4, highly tessellated patches must be broken into swaths sufficiently narrow that a row of vertices fits in on chip memory for reuse. Recall the situation illustrated in Figure 15 where guard curves were added to ensure that patches A and B matched on their shared boundary. Because the rows of patch B may be broken into multiple swaths, the guard curves of patch A must be segmented to match. The result is that patches are rendered in *swatches* that have rows and columns sized to match the on-chip memory, see Figure 18.

9 TESTING

The algorithms described in this paper result in relatively complex state machines, for managing vertex reuse, transition stitching, and consistency. To test the implementation for correctness we analyzed the triangle meshes produced by a bit accurate simulation of the tessellation engine. The meshes were tested for consistency by verifying that their triangles had consistent chirality and that the meshes contained the correct number of unique triangles and vertices.

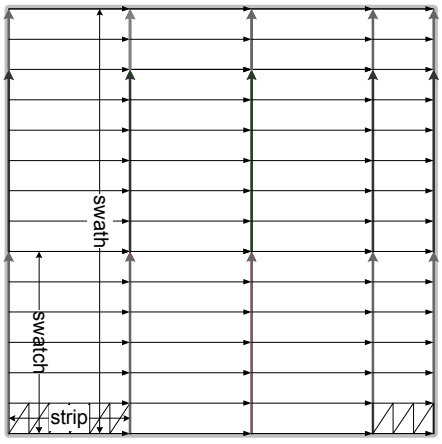

Figure 18: Breakup of a patch into swaths and swatches.

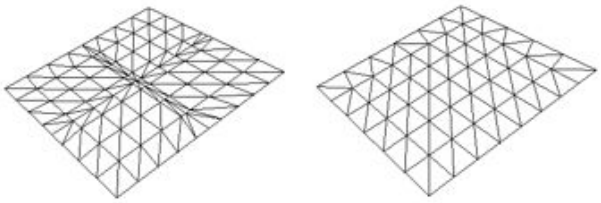

Figure 19: A single patch, fractional and integer tessellation.

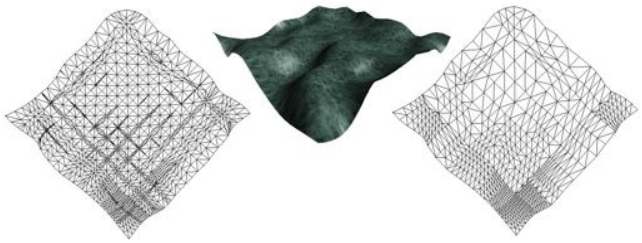

Figure 20: Multipatch adaptive tessellation.

10 EXAMPLES

In Figures 19 thru 22 we provide examples of the meshes generated by our system. In Figure 19 we provide an example of the fractional and integer tessellation styles applied to the same input patch and tessellation factors. In Figure 20 we illustrate the character of the tessellations resulting from LOD based on screen space edge length. Twenty-five patches are rendered with varying detail, the result of perspective foreshortening.

Figures 21 and 22 illustrate the meshes generated for three levels of detail based on screen space edge length. Figure 21 illustrates the integer tessellation scheme where the lowest level of detail consists of 72 triangles. The lowest level of detail in Figure 22 consists of 288 triangles.

11 PERFORMANCE

In this section we present preliminary performance results run on a 933mHz PIII with a 200mHz GeForce3 graphics card that implements the algorithms described in this paper. These statistics are based on rendering a version of the Utah teapot from Microsoft, containing 36 patches. The teapot is rendered once per frame, using a simple lighting model requiring a 24-byte vertex. The performance numbers include per frame overhead such as screen/Z clear. These results also do not include the overhead of host computations, a significant factor. Thus the results only reflect the efficiency of the graphics card as fed data across the AGP bus.

In Figure 23 we plot the triangle rate versus tessellation factor. As expected, the setup cost is amortized improving performance as the level of tessellation is increased. The trend continues until the tessellation factor requires multiple swatches, at which point we see a dip and return to the trend. We have measured a peak rate of 30 million triangles per second. In Figure 24 we measure bandwidth savings as the tessellation factor increases. Here we see the bytes per triangle dropping to roughly half of what could be achieved with perfect vertex reuse and triangle strips for the same tessellation; a rate of 12 bytes per triangle when two triangles are generated per vertex transferred.

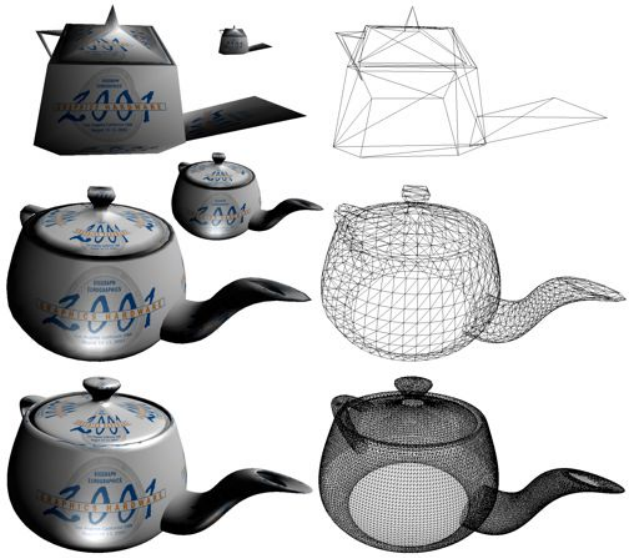

Figure 21: Integer levels of detail.

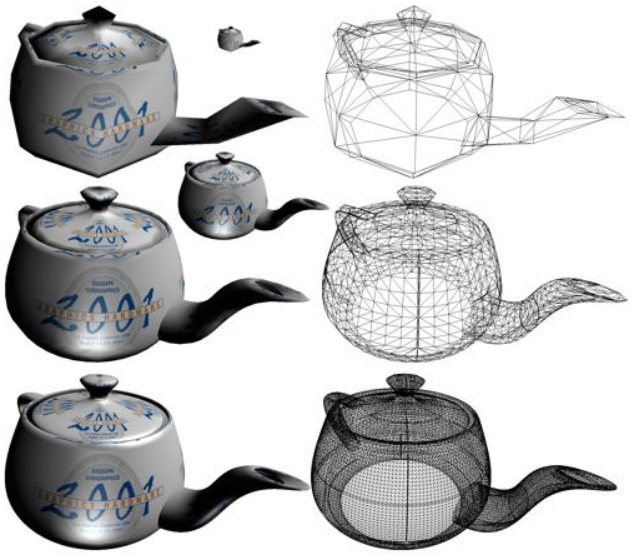

Figure 22: Fractional levels of detail.

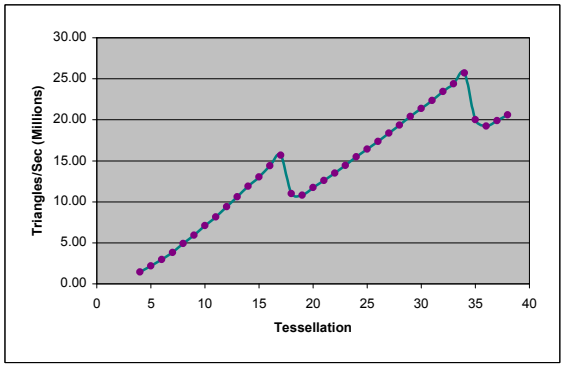

Figure 23: Triangles/sec vs. tessellation factor.

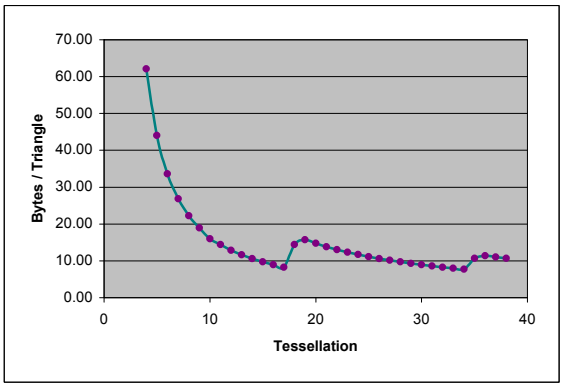

Figure 24: Bytes/triangle vs. tessellation factor.

The driver for our current implementation is still under development. The APIs are designed to permit caching of calculations when the tessellation is unchanging. However, it is apparent that the tessellation system is host limited when the application dynamically varies the input geometry and level of detail.

12 CONCLUSIONS

In this paper we have described a system for the defect free tessellation of multiple independent polynomial patches. The implementation splits the tessellation task between the host CPU and the graphics card achieving high performance with little incremental chip cost. The surface tessellator has been implemented in the NVIDIA GeForce3. The implementation performs as designed, achieving high triangle rate, continuous LOD, and defect free tessellation and rasterization of polynomial patches. Future work will explore quantifying driver overhead and moving more of the tessellation calculations to the GPU to better balance the system.

13 ACKNOWLEDGEMENTS

We would like to thank Doug Rogers, Al Zimmerman, Daniel Rohrer, Fred Fisher, Matt Craighead, Mark Kilgard, and David Kirk for their engineering contributions and support.

REFERENCES

[1] Stephan Bischoff, Leif P. Kobbelt and Hans-Peter Seidel. Towards Hardware Implementation Of Loop Subdivision, 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware, pages 41-50 (August 2000). ACM SIGGRAPH / Eurographics / ACM Press

[2] J. Bruijns; Quadratic Bezier triangles as drawing primitives; Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH workshop on Graphics hardware, 1998, Page 15

[3] Edwin E. Catmull. Computer Display of Curved Surfaces. Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure, pages 11-17 (May 1975).

[4] J. H. Clark. A Fast Scan-Line Algorithm for Rendering Parametric Surfaces, Computer Graphics, *13()*, pages 7-11 (August 1979).

[5] Gerald Farin. Curves and Surfaces for Computer Aided Geometric Design, pages 464 (1990). Academic Press. 2nd edition, ISBN: 0-12-249051

[6] R. T. Farouki and V. T. Rajan. Algorithms for Polynomials in Berstein Form, Computer Aided Geometric Design 5, pages 1-26, 1988.

[7] Tim Heidmann. Real Shadows, Real Time, Iris Universe, No. 18, pp 23-31, Silicon Graphics Inc., November 1991.

[8] D. Hilbert and S. Cohn-Vossen. Geometry and the Imagination. Chelsea Publishing Company, New York, 1952.

[9] James T. Kajiya. Ray Tracing Parametric Patches, Computer Graphics (Proceedings of SIGGRAPH 82), *16(3)*, pages 245-254 (July 1982, Boston, Massachusetts).

[10] Subodh Kumar, Dinesh Manocha and Anselmo Lastra. Interactive Display of Large-Scale NURBS Models, 1995 Symposium on Interactive 3D Graphics, pages 51-58 (April 1995). ACM SIGGRAPH. Edited by Pat Hanrahan and Jim Winget. ISBN 0-89791-736-7.

[11] J. Lane, L. Carpenter and T. Whitted and J. Blinn. Scan line methods for displaying parametrically defined surfaces, Communications of the ACM, *23 (1)*, pages 23-34 (1980).

[12] Sheue-Ling Lien, Michael Shantz and Vaughan Pratt. Adaptive Forward Differencing for Rendering Curves and Surfaces, Computer Graphics (Proceedings of SIGGRAPH 87), *21 (4)*, pages 111-118 (July 1987, Anaheim, California). Edited by Maureen C. Stone.

[13] Dani Lischinski. Converting Bézier Triangles Into Rectangular Patches, Graphics Gems III, pages 256-261, 536-537 (1992, Boston). Academic Press. Edited by David Kirk. ISBN 0-12-409673-5.

[14] Wayne Liu and Stephen Mann. An Optimal Algorithm for Expanding the Composition of Polynomials, ACM Transactions on Graphics, *16(2)*, pages 155-178 (April 1997). ISSN 0730-0301.

[15] Charles T. Loop. Smooth Subdivision Surfaces Based on Triangles. Master's thesis, University of Utah, Department of Mathematics, 1987.

[16] NVIDIA OpenGL Extension Specifications, NVIDIA Corporation, March 1, 2001. http://www.nvidia.com/marketing/developer/devrel.nsf/oglFrame?OpenPage

[17] Franco P. Perparata and Michael Ina Shamos. Computational Geometry, an Introduction. Springer-Verlag. New York 1985.

[18] Ron Pulleyblank and John Kapenga. The Feasibility of a VLSI Chip for Ray Tracing Bicubic Patches, IEEE Computer Graphics & Applications, *7 (3)*, pages 33-44 (March 1987).

[19] Kari Pulli and Mark Segal. Fast Rendering of Subdivision Surfaces. 7th Eurographics Rendering Workshop, Porto, Portugal, pages 61-70 and 282, June 1996.

[20] Alyn P. Rockwood. A generalized scanning technique for display of parametrically defined surfaces, IEEE Computer Graphics & Applications, *7 (8)*, pages 15-26 (August 1987).

[21] Alyn Rockwood, Kurt Heaton and Tom Davis. Real-time rendering of trimmed surfaces; Conference proceedings on Computer graphics, 1989, pages 107 - 116

[22] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.2.1). www.opengl.org.

[23] Alex Vlachos, Jörg Peters, Chas Boyd and Jason L. Mitchell. Curved PN Triangles. 2001 Symposium on Interactive 3D Graphics, March 19-21, 2001.