Extensions for 3D Graphics Rendering Engine Used for Direct Tessellation of Spline Surfaces

Dr. Adrian Sfarti, Prof. Brian A. Barsky, Todd J. Kosloff, Egon Pasztor, Alex Kozlowski, Eric Roman, and Alex Perelman

University of California, Berkeley

Abstract. In current 3D graphics architectures, the bus between the triangle server and the rendering engine GPU is clogged with triangle vertices and their many attributes (normal vectors, colors, texture coordinates).

We have developed a new 3D graphics architecture using data compression to unclog the bus between the triangle server and the rendering engine. This new architecture has been described in [1]. In the present paper we describe further developments of the newly proposed architecture.

The current paper shows several interesting extensions of our architecture such as backsurface rejection, NURBS real time tesselation and a description of a surface based API. We also show how the implementation of our architecture operates on top of the pixel shaders.

1 Introduction

In [1] we described a new graphics architecture that exploits faster arithmetic such that the CPU will serve parametric patches and the rendering engine (GPU) will triangulate those patches in real time. Thus, the bus sends control points of the surface patches, instead of the many triangle vertices forming the surface, to the rendering engine. The tesselation of the surface into triangles is distance-dependent, it is done in real time inside the rendering engine. The implementation of the architecture is through reprogramming one or more vertex processors inside the GPU.

2 Previous Work

There have been very few implementations of real time tesselation in hardware. In the mid-1980's, Sun developed an architecture for this that was described in [2] and in a series of associated patents. The implementation was not a significant technical or commercial success because it did not exploit triangle based rendering; instead it attempted to render the surfaces in a pixel-by-pixel manner [3]. The idea was to use adaptive forward differencing to interpolate infinitesimally close parallel cubic curves imbedded into the bicubic surface.

[4] describe hardware support for adaptive subdivision surface rendering.

Recently, NVIDIA has resurrected the real time tesselation unit [5]. The NVIDIA tesselation unit is located in front of the transformation unit and outputs triangle databases to be rendered by the existent components of the 3D graphics hardware.

More recently Bolz and Schroder have attempted to evaluate subdivision surfaces via programmable GPUs [6].
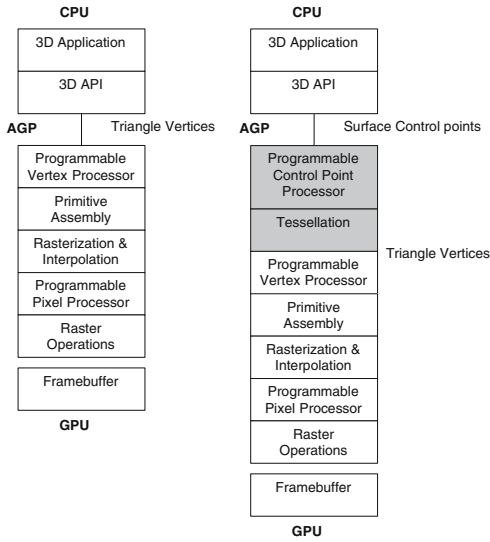
```
        CPU                         CPU
┌─────────────────┐        ┌─────────────────┐
│  3D Application │        │  3D Application │
├─────────────────┤        ├─────────────────┤
│     3D API      │        │     3D API      │
└─────────────────┘        └─────────────────┘
AGP      │ Triangle Vertices  AGP    │ Surface Control points
┌─────────────────┐        ┌─────────────────┐
│  Programmable   │        │  Programmable   │
│ Vertex Processor│        │ Control Point   │
├─────────────────┤        │   Processor     │
│    Primitive    │        ├─────────────────┤
│    Assembly     │        │  Tessellation   │
├─────────────────┤        ├─────────────────┤
│ Rasterization & │        │  Programmable   │ ─ Triangle Vertices
│  Interpolation  │        │ Vertex Processor│
├─────────────────┤        ├─────────────────┤
│  Programmable   │        │    Primitive    │
│ Pixel Processor │        │    Assembly     │
├─────────────────┤        ├─────────────────┤
│     Raster      │        │ Rasterization & │
│   Operations    │        │  Interpolation  │
└─────────────────┘        ├─────────────────┤
┌─────────────────┐        │  Programmable   │
│   Framebuffer   │        │ Pixel Processor │
└─────────────────┘        ├─────────────────┤
        GPU                │     Raster      │
                           │   Operations    │
                           └─────────────────┘
                           ┌─────────────────┐
                           │   Framebuffer   │
                           └─────────────────┘
                                   GPU
```

Fig. 1. Conventional programmable architecture (left), new architecture (right). The new architecture adds two stages to the GPU pipeline, which are shown in grey.

3 Tesselator Unit Back-Facing Surface Removal and Trivial Clipping

Since surfaces that are entirely back-facing should be discarded before attempting tesselation, we first test to determine if any portion of the patch is facing towards the viewer. Although back-facing triangles are discarded in the triangle rendering portion of the pipeline, the discarding of entire surfaces as early as possible in the pipeline reduces time spent on tesselation. To determine whether or not a patch is back-facing, we use the convex hull of the patch. Specifically, we test the polyhedral faces of the convex hull to determine if they are forward-facing. If any face is forward-facing, then the patch is not discarded. Note that not all faces need to be tested; as soon as a forward-facing polyhedral face is found, then we know that the patch cannot be discarded.

If the 16 control points of a surface all lie outside the same clipping plane, the surface can be trivially clipped out of the database.

We produced five different animations in order to prove our backface removal and surface trivial clipping algorithm.

4 Tesselator Unit Extension to NURBS

The algorithm described above has a straightforward extension for rational surfaces such as non-uniform rational B-spline (called "NURBS") surfaces.

A nonuniform rational B-spline surface of degree (p,q) is defined by

$$S(s,t) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} N_{i,p}(s)N_{j,q}(t)w_{i,j}P_{i,j}}{\sum_{i=1}^{m}\sum_{j=1}^{n} N_{i,p}(s)N_{j,q}(t)w_{i,j}}$$

where $N_{i,p}()$ denotes the B-spline basis of degree p, w denotes the matrix of weights, and P denotes the matrix of control points.

Each patch of such a surface lies within the convex hull formed by its control points. To illustrate the concept, consider $p = q = 4$ and $m = n = 4$. There are 16 control points, P_{11} through P_{44} (similar to the surfaces described in the previous paragraph). The surface lies within the convex hull formed by P_{11} thru P_{44}.

Now consider any one of the curves:

$$C(s) = \frac{\sum_{i=1}^{m} N_{i,p}(s)w_i P_i}{\sum_{i=1}^{m} N_{i,p}(s)w_i}$$

where the parameter p denotes the order, $N_{i,p}(s)$ the B-spline basis functions, P_i dentoes the control points, and w_i is the weight of control point P_i and is represented as the last coordinate of the homogeneous point. The curve lies within the hull formed by the control points. Such a curve can be obtained by fixing one of the two parameters s or t in the surface description. For example holding $t = 0$ while letting s vary produces such a curve. As in the case of Bézier surfaces, there are eight such curves; that is, four boundary curves and four internal curves. The tesselation of the surface reduces to the subdivision of the hull of the boundary curves or of the internal curves as described in the case of the Bézier surfaces. By subdividing the B-spline surface we reduced the problem to a problem that we have already solved. Trimmed NURBS are treated as the intersection of the triangle meshes resulting from the tesselation of the untrimmed NURBS used in the intersection. This simple method outputs the trimming loops rather than requiring them as an input in a drastic departure from the current approaches.

5 Tesselator Unit and Programmable Shaders

While we can readily build GPUs according to the proposed architecture from existing designs, there are some issues surrounding the transition that we will identify and resolve. One concern for vertex shader programming under this new architecture, is how to pass custom per-vertex information to the vertex shader. Since the 3D

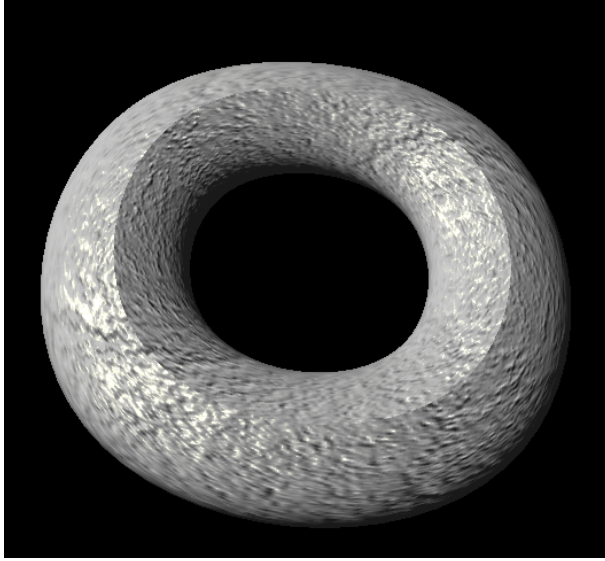**Fig. 2.** Per pixel lighting using programmable shaders

Fig. 3. Bump mapped ridged torus using programmable shaders

application is sending a stream of control points to the GPU, all of the vertices generated by the tesselator and subsequently fed to the programmable vertex processor, have properties calculated by the Tesselation Processor as described in [1], not by the application. This complication can be resolved quite naturally using a technique already familiar to shader programmers. Property maps for custom attributes can be encoded as 2D-textures, and vertex shaders could sample the texture to extract interpolated values. This allows each vertex generated within the surface to assume a meaningful custom attribute value (could use s/t values, position, texture coordinates etc. for sampling location). Many existing vertex shaders rely on heavily (and uniformly) tessellated models. The greater number of vertices can increase the quality of per-vertex lighting calculations, and facilitate displacement mapping of vertices. We support this class of shader despite our distance-dependent subdivision, to the degree required by the vertex shader program.

- Firstly, we might increase the overall fineness of the resulting tessellation by controlling the termination criterion.
- Secondly, we could invoke an adaptive mesh refinement as part of our tesselation algorithm This could be sufficient for displacement mapping, where the vertex shader merely needs more points to play with. More sophisticated mesh refinements would benefit lighting and texturing without having to evaluate more points using the surface description.

Given the introduction of the control point shader, for maximum effectiveness the vertex shader should only permute geometry in subtle ways. Following this convention will ensure that vertex shaders do not invalidate the perceptual accuracy of our

distance-dependent tessellation. Lastly, one should remember that the Tesselation Processor already incorporates a minimum level of tessellation . The use of vertex/pixel shaders was a natural extension in our software demonstration, and required no special enhancements. Figures 2 and 3 show the combination of real-time tessellation and programmable shaders to bump map bicubic surfaces.

6   A Prototype for a Graphics Library Utility

To facilitate the design of drivers for the proposed architecture, we must develop a Graphics Library Utility (GLU). The primitives of the GLU are strips, fans, meshes and indexed meshes. Current rendering methods are based on triangle databases (strips, fans, meshes) resulting from offline tesselation via specialized tools. These tools tessellate the patch databases and ensure that there are no cracks between the resulting triangle databases. The tools use some form of zippering. The triangle databases are then streamed over the AGP bus into the GPU. There is no need for any coherency between the strips, fans, etc., since they are, by definition, coherent (there are no T-joints between them). The net result is that the GPU does not need any information about the entire database of triangles, which can be quite huge. Thus, the GPUs can process virtually infinite triangle databases. Referring to Figure 4., in a strip, the first patch will contribute sixteen vertices, and each successive patch will contribute only twelve vertices because four vertices are shared with the previous patch. Of the sixteen vertices of
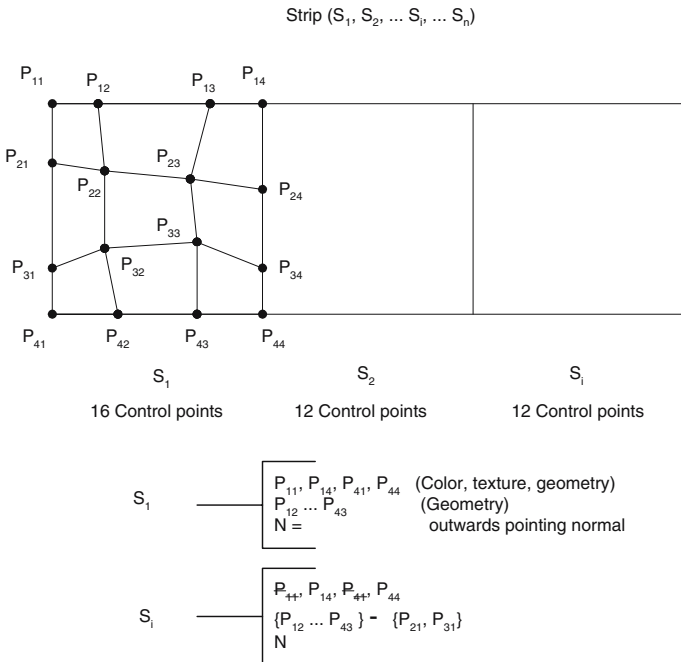

Fig. 4. Strip

Mesh (S_{11}, S_{12}, ... S_{1N}, ... S_{21}, ... S_{2N}, ... S_{M1}, ... S_{MN})

S_{M1} 12 Control Points	S_{M2} 9		S_{Mi} 9		S_{MN} 9
S_{21} 12 Control Points	S_{22} 9 Control Points		S_{2i} 9		S_{2N} 9
S_{11} 16 Control Points	S_{12} 12 Control Points		S_{1i} 12		S_{1N} 12

Fig. 5. Mesh

S_1

11 Control Points

S_i

8 Control Points

S_1 ———— $\overline{P_{11}, P_{14}, P_{41}, P_{44}}$ (Color, texture, geometry)

{ P_{12} ... P_{43} } ▬ {P_{24}, P_{34}, P_{33}}   (Geometry)

N

S_i ———— $\overline{P_{11}, P_{14}, P_{41}, P_{44}}$

{P_{12} ... P_{43} } ▬ {P_{24}, P_{34}, P_{33}} ▬ {P_{12}, P_{13}}
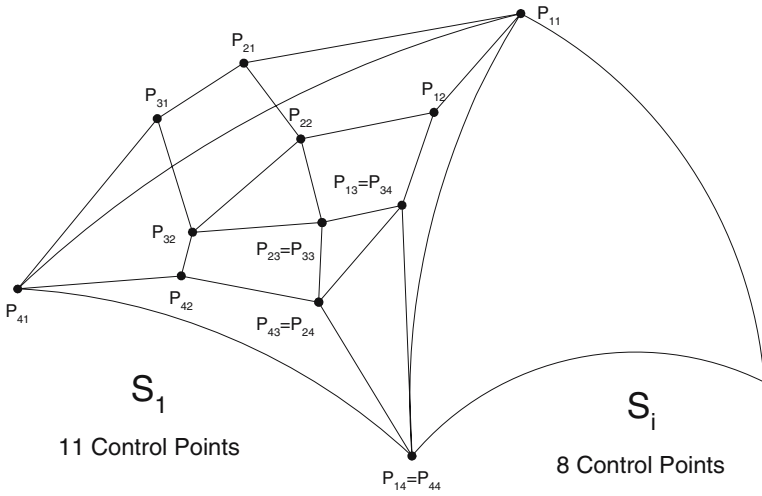
N
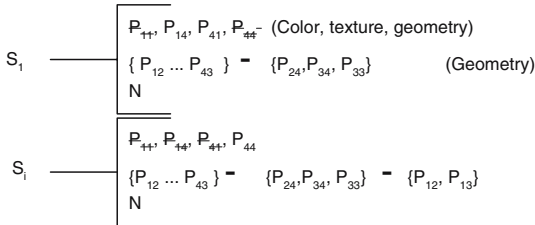
Fig. 6. Fan

the first patch, S_1, there will only be four vertices (namely, the corners P_{11}, P_{14}, P_{41}, P_{44}) that will have color and texture attributes; the remaining twelve vertices will have only geometry attributes. Of the twelve vertices of each successive patch, S_i, in the strip,

there will only be two vertices (namely, P_{14} and P_{44}) that will have color and texture attributes. It is this reduction in the number of vertices that will have color and texture attributes that accounts for the reduction of the memory footprint and for the reduction of the bus bandwidth necessary for transmitting the primitive from the CPU to the rendering engine (GPU) over the AGP bus. Further compression is achieved because a patch is expanded into potentially many triangles by the Tessellator Unit inside the GPU. Referring to Figure 5, in a mesh, the anchor patch, S_{11} has sixteen vertices, all the patches in the horizontal and vertical strips attached to S_{11} have twelve vertices and all the other patches have nine vertices. Each patch has an outward pointing normal. Referring to Figure 6, each patch has only three boundary curves, the fourth boundary having collapsed to the center of the fan. The first patch in the fan enumeration has eleven vertices; each successive patch has eight vertices. The vertex P_{11}, which is listed first in the fan definition, is the center of the fan and has color and texture attributes in addition to geometric attributes. The first patch, S_1, has two vertices with color and texture attributes, namely P_{41} and P_{14}; the remaining nine vertices have only geometric attributes. Each successive patch, Si, has only one vertex with all the attributes.

The meshed curved patch data structures introduced above are designed to replace the triangle data structures used in the conventional architectures. Within one patch strip, the edge database must be retained for zippering reasons but no information needs to be stored between two abutting patches. If two patches bounding two separate surfaces share an edge curve, they share the same control points and they will share the same tesselation. By doing so we ensure the absence of cracks between patches that belong to data structures that have been dispatched independently and thus our method scales the exactly the same way the traditional triangle based method does.

7   Conclusion

We developed a new 3D graphics architecture that replaces the conventional idea of a 3D engine (GPU) that renders triangles with a GPU that will tesselate surface patches into triangles. Thus, the bus sends control points of the surface patches, instead of the many triangle vertices forming the surface, to the rendering engine. The tesselation of the surface into triangles is distance-dependent, it is done in real time inside the rendering engine.

Acknowledgements

References

1. Sfarti, A., Barsky, B., Kosloff, T., Pasztor, E., Kozlowski, A., Roman, E., Perelman, A.: New 3d graphics rendering engine architecture for direct tessellation of spline surfaces. In: International Conference on Computational Science (2). (2005) 224–231

2. Lien, S.L., Shantz, M., Pratt, V.R.: Adaptive forward differencing for rendering curves and surfaces. In: SIGGRAPH '87 Proceedings, ACM (1987) 111–118
3. Lien, S.L., Shantz, M.: Shading bicubic patches. In: SIGGRAPH '87 Proceedings, ACM (1987) 189–196
4. Boo, M., Amor, M., Dogget, M., Hirche, J., Stasser, W.: Hardware support for adaptive subdivision surface rendering. In: ACM SIGGRAPH/Eurographics workshop on Graphics Hardware. (2001) 30–40
5. Moreton, H.P.: Integrated tesselator in a graphics processing unit. U.S. patent (2003) #6,597,356.
6. Bolz, J., Schroder, P.: Evaluation of subdivision surfaces on programmable graphics hardware. http://www.multires.caltech.edu/pubs/GPUSubD.pdf (2003)
7. Clark, J.H.: A fast algorithm for rendering parametric surfaces. In: Computer Graphics (SIGGRAPH '79 Proceedings). Volume 13(2) Special Issue., ACM (1979) 7–12
8. Moreton, H.P.: Watertight tesellation using forward differencing. In: Proceedings of the ACM SIGGRAPH/Eurographcs workshop on graphics hardware. (2001)
9. Chung, A.J., Field, A.: A simple recursive tesselator for adaptive surface triangulation. JGT **5(3)** (2000)
10. Moule, K., McCool, M.: Efficient bounded adaptive tesselation of displacement maps. In: Graphics Interface 2002. (2002)
11. Boo, M., Amor, M., Dogget, M., Hirche, J., Strasser, W.: Hardware support for adaptive subdivision surface rendering. In: Proceedings of the ACM SIGGRAPH/Eurographics workshop on Graphics Hardware. (2001) 33–40
12. Hoppe, H.: View-dependent refinement of progressive meshes. In: Proceedings of the 24th annual conference on computer graphics and interactive techniques. (1997)
13. Sfarti, A.: Bicubic surface rendering. U.S. patent (2003) #6,563,501.
14. Sfarti, A.: System and method for adjusting pixel parameters by subpixel positioning. U.S. patent (2001) #6,219,070.
15. Barsky, B.A., DeRose, T.D., Dippe, M.D.: An adaptive subdivision method with crack prevention for rendering beta-spline objects. Technical Report, UCB/CSD 87/384, Computer Science Division, Electrical Engineering and Computer Sciences Department, University of California, Berkeley, California, USA (1987)
16. Lane, J.F., Carpenter, L.C., Whitted, J.T., Blinn, J.F.: Scan line methods for displaying parametrically defined surfaces. In: Communications of the ACM. Volume 23(1)., ACM (1980) 23–24
17. Forsey, D.R., Klassen, R.V.: An adaptive subdivision algorithm for crack prevention in the display of parametric surfaces. In: Proceedings of Graphics Interface. (1990) 1–8
18. Velho, L., de Figueiredo, L.H., Gomes, J.: A unified approach for hierarchical adaptive tesselation of surfaces. In: ACM Transactions on Graphics. Volume 18(4)., ACM (1999) 329–360
19. Kahlesz, F., Balazs, A., Klein, R.: Nurbs rendering in opensg plus. In: OpenSG 2002 Papers. (2002)