DISPLAYING TRIMMED NURBS SURFACES USING THE GPU

ADARSH KRISHNAMURTHY

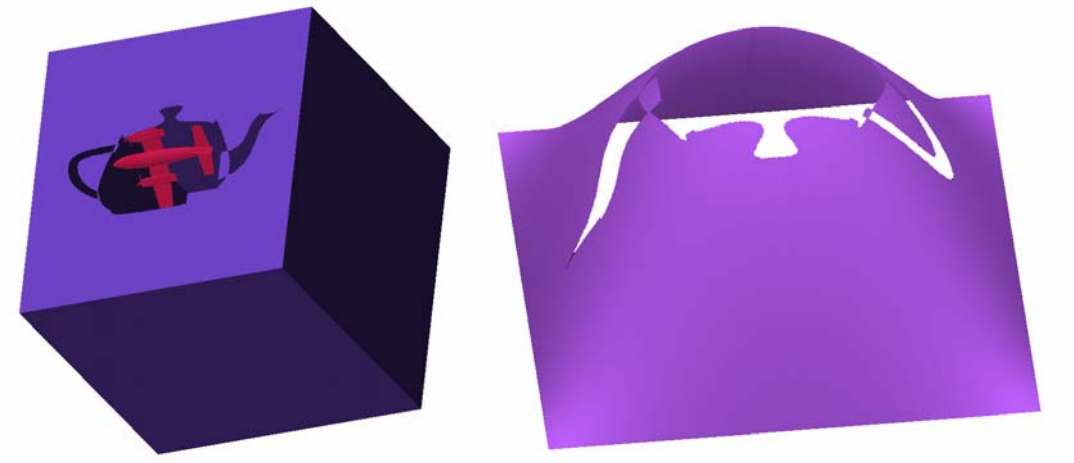

Figure 1: Rendering of Trimmed NURBS Surfaces

ABSTRACT

Non-Uniform Rational BSplines or NURBS, even though being industry standard in representing solid models, do not have inbuilt hardware support for displaying them. The existing software implementation in OpenGL is good for Non-Trimmed NURBS surfaces but has issues when displaying Trimmed NURBS surfaces. This project tries to display such Trimmed NURBS surfaces using programmable graphics hardware. The non-trimmed NURBS surfaces are evaluated first on the CPU. Then using texture mapping and a special texture called the Trim-Texture they are then trimmed on the graphics hardware. Such a method is faster than trimming on the CPU and produces good-looking results. Various examples of trimming using the OpenGL implementation and using the proposed algorithm are presented in this project. An outline is also presented on how to implement it in current solid modeling packages.

INTRODUCTION

NURBS or Non-Uniform Rational B-Splines are the industry standard in commercial Computer Aided Design systems for the representation and design of geometry. The low-level primitives like triangles and lines can be used to define arbitrarily shaped objects, but at the cost of mathematical properties; for example, a circle that is approximated by a sequence of line segments will change its shape when rotated. One of the advantages of NURBS curves is that they offer a way to represent arbitrary shapes while maintaining mathematical exactness and resolution independence. Among their useful properties are the following

1. They can represent virtually any desired shape, which include points, straight lines, polylines, conic sections (circles, ellipses, parabolas and hyperbolas) and free-form curves with arbitrary shapes.
2. They give you great control over the shape of a curve. A set of control points and knots, which guide the curve's shape, can be directly manipulated to control its smoothness and curvature.
3. They can represent very complex shapes with remarkably little data.
4. They are invariant under affine as well as perspective transformations.
5. They can be evaluated reasonably fast by numerically stable and accurate algorithms.
6. They are generalizations of non-rational B-Splines and non-rational and rational Bezier curves and surfaces.

Even though NURBS are ubiquitous in the Design Industry, there is no inbuilt hardware support for displaying NURBS surfaces in current graphic cards. Therefore, all current CAD systems evaluate and tessellate the NURBS into triangles and then use the existing graphics pipeline to display them. This is not an ideal way to deal with rendering NURBS surfaces as the tessellation is done only for visualization purposes and it is usually a time consuming process. Thus, this project based on [Guthe et al. 2005] makes use of programmable graphics hardware to display trimmed NURBS surfaces without actual tessellation.

NURBS REPRESENTATION

As mentioned before, NURBS are the standard surface representation in all major CAD software. Mathematically NURBS curves are defined by the set of points called the control points. The amount by which a control point influences the curve is defined by the BSpline basis function and the extent to which it influences the curve is defined by a vector called the Knot vector. Moreover, the control points can also have weights attached to them that make the BSpline rational.

Equation (1) gives the definition of a NURBS curve C as a function of the parameter u. The $N_{i,p}$ is the BSpline basis function given by Equation (2). The u_i in the definition of the basis functions represent the i^{th} component of the knot vector. This definition of the basis function is recursive and the basis function of order k depends on the basis functions of order k-1.

$$C(u) = \sum_{i=0}^{n} \left(\frac{N_{i,p}(u) w_i}{\sum_{j=0}^{n} N_{j,p}(u) w_j} \right) \mathrm{P}_i \tag{1}$$

$$\begin{aligned} N_{i,p}(u) &= \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \\ N_{i,0}(u) &= \begin{cases} 1 & \text{if } u_i \le u \le u_{i+1} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{2}$$

NURBS surfaces are a direct extension of the NURBS curves. They are tensor product surfaces of NURBS curves in two parametric directions. Equation (3) gives the mathematical definition of such a surface.

$$S(u,v) = \sum_{j=0}^{m} \sum_{i=0}^{n} \left(\frac{N_{i,p}(u) w_i}{\sum_{k=0}^{n} N_{k,p}(u) w_k} \right) \left(\frac{N_{j,q}(v) w_j}{\sum_{k=0}^{m} N_{k,q}(v) w_k} \right) \mathrm{P}_{i,j} \tag{3}$$

TRIMMED NURBS SURFACES

CAD programs usually use Boundary Representation or B-Rep for representing the solid models. Moreover, such a boundary of the CAD model is usually represented by NURBS surfaces. While it is possible to stitch a complex model together by placing one surface adjacent to another, it is tedious and can make model modification difficult. In addition, these surfaces being tensor product surfaces are rectangular sheets. Therefore they are not very flexible, especially when it comes to representing a surfaces which are not rectangular or those with holes or complex local geometries. In addition, as shown in Figure 2, complex NURBS surfaces can result due to certain Boolean operations performed by CAD programs. The solution to this problem of representing such complex surfaces is to trim the NURBS surface. Trimming can be visualized as cutting a portion of the NURBS surface and then stretching it by changing the control points.

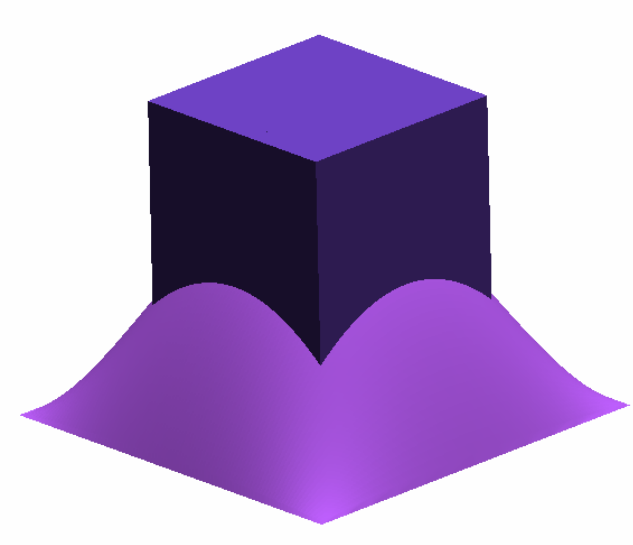

Figure 2: Source of Trimmed NURBS surfaces in CAD programs

The trim information is usually defined in the parametric domain of the surface as shown in Figure 3. This is important to note, since the parametric domain is 2D, which simplifies many computations over those that would result if the trims were defined directly on the 3D surface. Usually all trims form directed closed loops and the direction of the loop determines which side of the trim to cut away. In addition, all surfaces have at least one trim that bounds the valid surface region. There can also be multiple loops per surface; one defining the boundary and others defining interior holes.

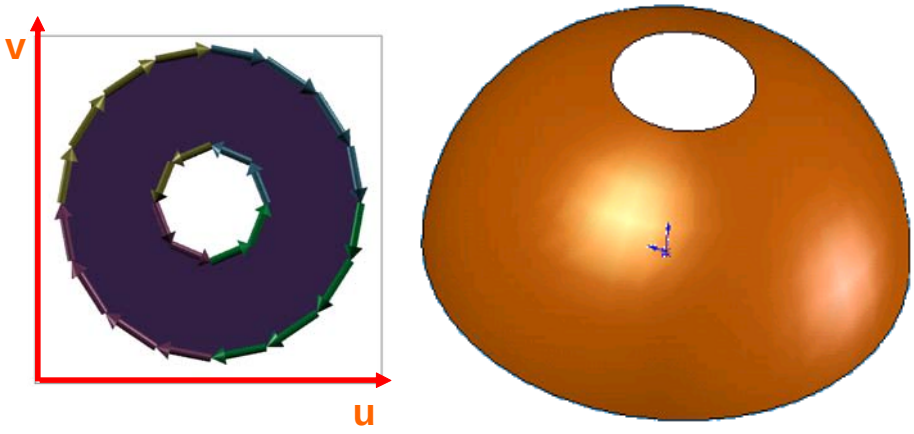

Figure 3: Trim curve in the parametric domain and the Trimmed NURBS surface patch

RENDERING OF TRIMMED NURBS SURFACES

Theoretically, Trimmed NURBS surfaces can be rendered by evaluating the surface points, which is essentially a mapping of parametric points from $\Re^2$ to $\Re^3$. Then the parts of the NURBS surface that lie outside the trim curves are cut away. However, current implementations of rendering trimmed curves usually approximate the NURBS surface to a Bezier surface patch as a first step. Then these patches are trimmed and tessellated into triangles. This triangle mesh is then sent to the graphics card for display.

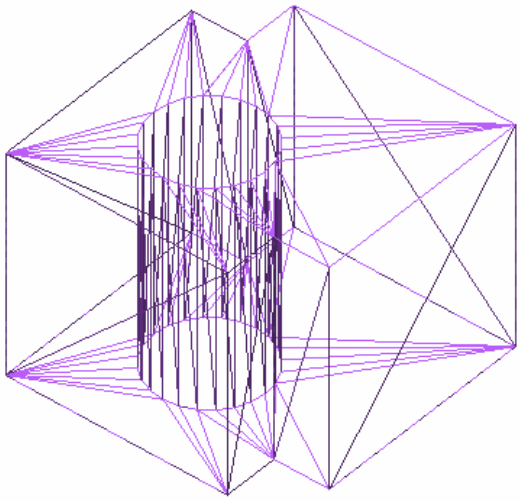

Figure 4: Object tessellated in ACIS

This method, apart from being time consuming, sometimes leads to long skinny triangles. These types of triangles are not good for display purposes as the color and normal information when interpolated over such a long dimension will lead to display artifacts. In addition, since the curves are approximated with piecewise line segments, it leads to certain tessellation artifacts like jagged edges. Thus, a method of displaying NURBS without actual tessellation will be an ideal solution.

OPENGL IMPLEMENTATION

OpenGL has its own software implementation for displaying trimmed NURBS surfaces. It is based on the implementation given in [Rockwood et al. 1989] that deals with trimmed parametric surfaces. The syntax of the OpenGL implementation of Trimmed NURBS is given below [Redbook].

```
void gluNurbsSurface(GLUnurbsObj *nobj, GLint uknot_count,
            GLfloat *uknot, GLint vknot_count, GLfloat *vknot,
            GLint u_stride, GLint v_stride, GLfloat *ctlarray,
            GLint uorder, GLint vorder, GLenum type);
```

Describes the vertices (or surface normals or texture coordinates) of a NURBS surface, nobj. Several of the values must be specified for both u and v parametric directions, such as the knot sequences (uknot and vknot), knot counts (uknot_count and vknot_count), and order of the polynomial (uorder and vorder) for the NURBS surface. Note that the number of control points isn't specified. Instead, it's derived by determining the number of control points along each parameter as the number of knots minus the order. Then, the number of control points for the surface is equal to the number of control points in each parametric direction, multiplied by one another. The ctlarray argument points to an array of control points. The last parameter, type, is one of the two-dimensional evaluator types. Commonly, you might use GL_MAP2_VERTEX_3 for nonrational or GL_MAP2_VERTEX_4 for rational control points, respectively. You might also use other types, such as GL_MAP2_TEXTURE_COORD_* or GL_MAP2_NORMAL to calculate and assign texture coordinates or surface normals.

```
void gluBeginTrim (GLUnurbsObj *nobj);
void gluEndTrim (GLUnurbsObj *nobj);
```

Marks the beginning and end of the definition of a trimming loop. A trimming loop is a set of oriented, trimming curve segments (forming a closed curve) that defines the boundaries of a NURBS surface.

```
void gluPwlCurve (GLUnurbsObj *nobj, GLint count, GLfloat *array,
        GLint stride, GLenum type);
```

Describes a piecewise linear trimming curve for the NURBS object nobj. There are count points on the curve, and they're given by array. The type can be either GLU_MAP1_TRIM_2 (the most common) or GLU_MAP1_TRIM_3 ((u, v, w) homogeneous parameter space). The type affects whether stride, the number of floating-point values to the next vertex, is 2 or 3.

The OpenGL implementation of untrimmed NURBS evaluation is reasonably sophisticated. The level of detail is changed dynamically based on the distance of the surface from the eye point. There is also an option to perform view frustum based culling automatically.

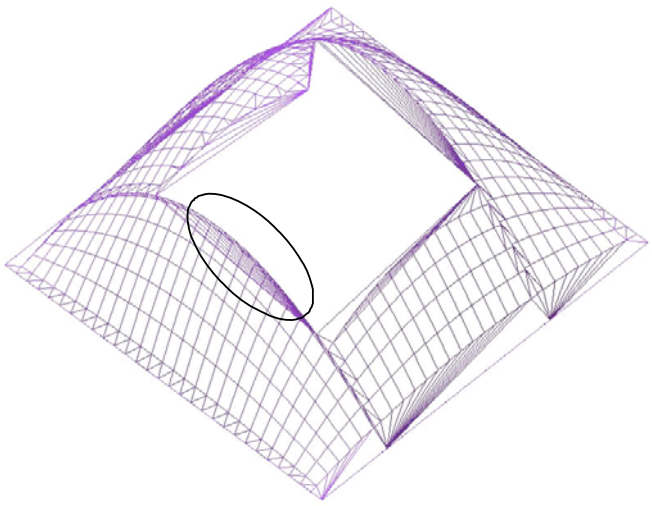

Figure 5: Trimmed NURBS surface tessellated using OpenGL

However, the implementation of Trimmed NURBS surfaces is not that clean and has some tessellation errors. For example, Figure 5 shows a trimmed NURBS surface tessellated using the OpenGL implementation. Clearly, there are some tessellation artifacts as marked out in the figure. These artifacts are because the NURBS surface is first trimmed in the 2D parametric space and then mapped into 3D. While the edges along the trim curves' may overlap each other and lie in the same plane in the parameter space, they might be mapped to different z positions. As a result, the lines connecting the points along the edge of the trim curve do not map exactly to those on the NURBS surface and this leads to some errors.

ALGORITHM

Figure 6 shows the proposed algorithm to display trimmed NURBS surfaces. It is a 2-pass algorithm. In the first pass, the trimming curves are evaluated partly on the CPU as well as on the GPU and the Trim-Texture is generated. The program makes use of the OpenGL evaluation of the NURBS surface, which is done in the CPU. Then in the second pass, the actual trimming of this NURBS surface is performed on GPU. The different sections of the algorithm are discussed in detail in the following sections.

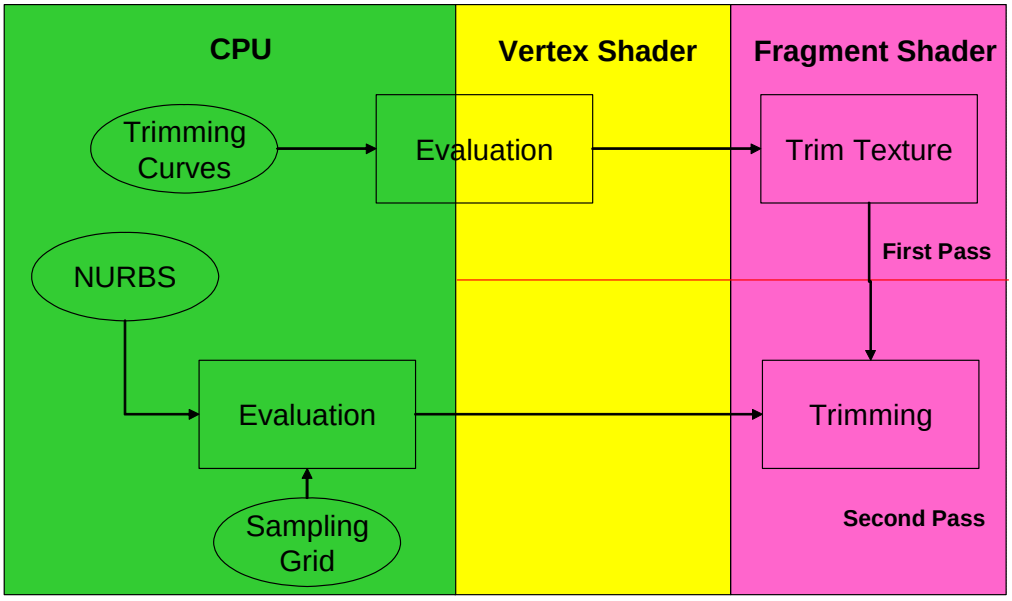

Figure 6: Algorithm

TRIM TEXTURE GENERATION

The first step in displaying the trimmed NURBS surfaces is to generate the trim texture. This is done in the first pass and the program makes use of the Frame-buffer object to render directly to a texture. The basic principle is to render the trim curves on to a 2D texture with binary values. Then this texture is mapped on to the NURBS surface and is used for determining which part of the surface is to be rendered.

Also as mentioned in [Guthe et al. 2005], the trim texture is generated by rendering different parts using triangles and then use those regions which are rendered an odd number of times. This is shown in Figure 7, where only parts of the domain that are rendered once or thrice are considered to be inside the trim curve. The one advantage of using such an algorithm is that the orientation of the trim curves need not be explicitly considered.

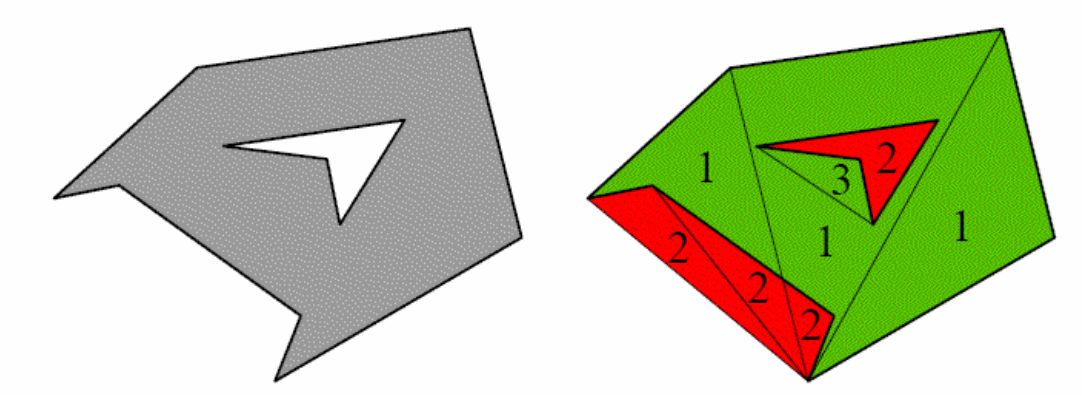

Figure 7: Trim Texture Generation

The above algorithm, even though conceptually very simple was a little difficult to implement directly as there is no access to the frame buffer from the fragment program and also the same texture cannot be read and written at the same time. Hence, an implementation based on the alpha blending functionality of graphics cards is used to generate the trim texture.

First the trim curves are evaluated and converted to piecewise linear segments. The viewport is set up to match the size of the trim texture, which is determined based on the distance of the farthest corner the NURBS surface. This size is then rounded to the next highest power of 2. The error introduced in sampling only the corner is small as the

value is anyway rounded to the next power of 2. Then the Model View matrix is set to 2D mode with view box going from [0 1] in the length and height dimension corresponding to the u and v directions. Furthermore, the background color is cleared to (0, 0, 0, 0). In OpenGL the color values of the incoming fragment (the source) are combined with the color values of the corresponding currently stored pixel (the destination) in a two-stage process during alpha blending. First, the user specifies how to compute source and destination factors. These factors are RGBA quadruplets that are multiplied by each component of the R, G, B and A values in the source and destination, respectively. Then the corresponding components in the two sets of RGBA quadruplets are added. To show this mathematically, let the source and destination blending factors be (S_r, S_g, S_b, S_a) and (D_r, D_g, D_b, D_a), respectively, and the RGBA values of the source and destination be indicated with a subscript of s or d. Then the final, blended RGBA values are given by Equation (4). Each component of this quadruplet is eventually clamped to [0, 1].

$$(R_s S_r + R_d D_r,\ G_s S_g + G_d D_g,\ B_s S_b + B_d D_b,\ A_s S_a + A_d D_a) \tag{4}$$

The blending factors are chosen carefully and the final blending function is given in Equation (5). This function effectively toggles the value from 0 to 1 or 1 to 0 whenever a new fragment is drawn over an existing one.

$$(R_s(1-R_d)+R_d \times 0,\ G_s(1-G_d)+G_d \times 0,\ B_s(1-B_d)+B_d \times 0,\ A_s(1-A_d)+A_d \times 0) \tag{5}$$

Once all the parameters are set up, triangles made up of the 1st, nth and the n+1th vertex is drawn with color (1, 1, 1, 1). The pseudo code for the above algorithm is given below.

```
gluOrtho2D(0.0, 1.0, 0.0, 1.0);
// Set The Blending Function For Translucency
glBlendFunc(GL_ONE_MINUS_DST_COLOR,GL_ZERO);

glClearColor(0,0,0,0);
glColor4f(1,1,1,1);
glBegin(GL_TRIANGLES);
      Triangle Fan
glEnd();
```

TRIMMING

Then next part of the algorithm is to use the trim texture to trim off parts of the NURBS surface. This is done in the fragment program during the rendering pass. The trim texture, even though has alpha values that can be mapped directly to the surface by using alpha blending, this may lead to wrong results. One such example is shown in Figure 8, where alpha blending is used. Since the curve is not usually rendered from front to back order, the blending may not be correct and leads to certain parts of the surface not being rendered.

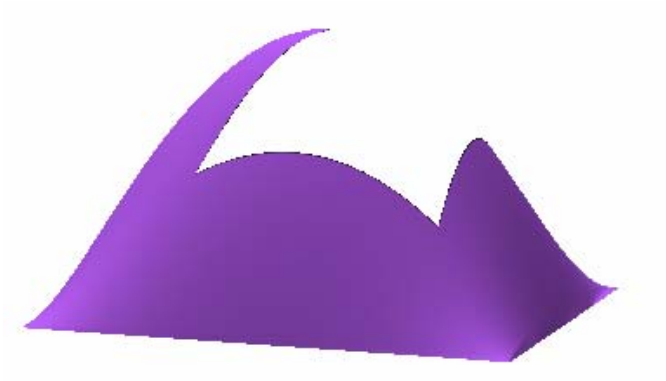

Figure 8: Incorrect rendering using alpha blending

To overcome this problem, the parts of the NURBS surface that do not lie inside the trim curves are totally discarded. This has both advantages and disadvantages. The advantage is that the lighting calculations need not be done to those fragments that are discarded. However, this implementation uses branching and can lead to performance drop in certain graphic cards.

Figure 9: Trimmed NURBS surface rendered using the GPU

The fragment program, written in Cg, used for the trimming operation is given below. It makes use of the “discard” command that kills the fragment when its color value

is 0. This implementation uses the red channel for the calculations. To save memory we can store different trim textures in different channels of the same texture. Figure 10 shows the NURBS surface in the same orientation as in Figure 5. The resulting surface is much cleaner than the OpenGL implementation and correctly represents the trimmed surface.

```
half4 main( half2 coords : TEXCOORD0,
            half4 color : COLOR,
            uniform sampler2D texture) : COLOR
{
      half4 c  = tex2D(texture, coords);
      if (c.x == 0)
            discard;
      return color;
}
```

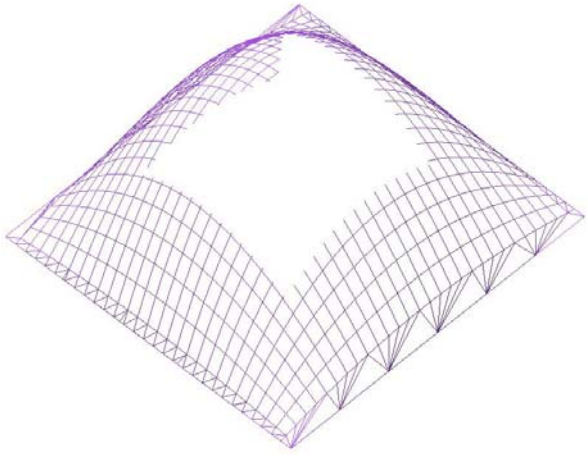

Figure 10: Trimmed NURBS surface

TRIM CURVE EVALUATION

Trim curves that are themselves defined by NURBS curves are evaluated separately and converted to piecewise line segments. This conversion process itself can be done in the GPU using the vertex program, but doing so requires some simplifications. For example if the curve is known to be of a low order uniform spline like cubic Bezier, it can be evaluated. However, for this project the trim curves are evaluated on the CPU. This is not optimal in the sense that the evaluation being numerically intensive is much more suited for evaluation on the GPU. Figure 11 shows such a trimmed NURBS surface where the trim curve itself is a NURBS curve.

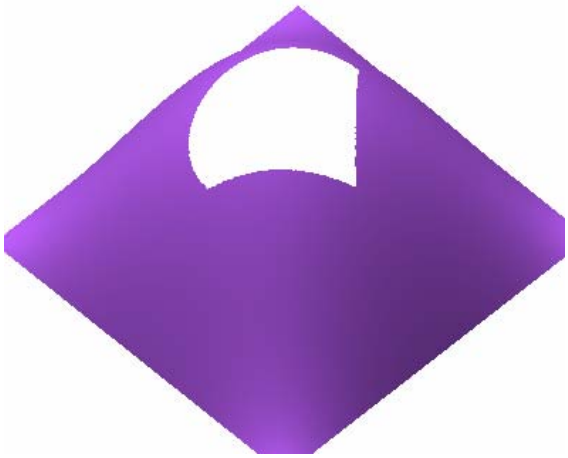

Figure 11: NURBS surface with NURBS trim curve

RESULTS

The GPU algorithm developed gives better and cleaner results than the OpenGL implementation. This is evident from all the figures given in this report. Moreover, to evaluate the effectiveness of the GPU algorithm, some timing calculations were made. Figure 12 gives the time it takes for the GPU algorithm and the CPU OpenGL implementation. The GPU algorithm is faster than the CPU implementation, but the difference in timing is not that drastic. This is because the CPU is still used for the surface evaluation. If this part of the algorithm is also ported to the GPU, it might lead to a better performance.

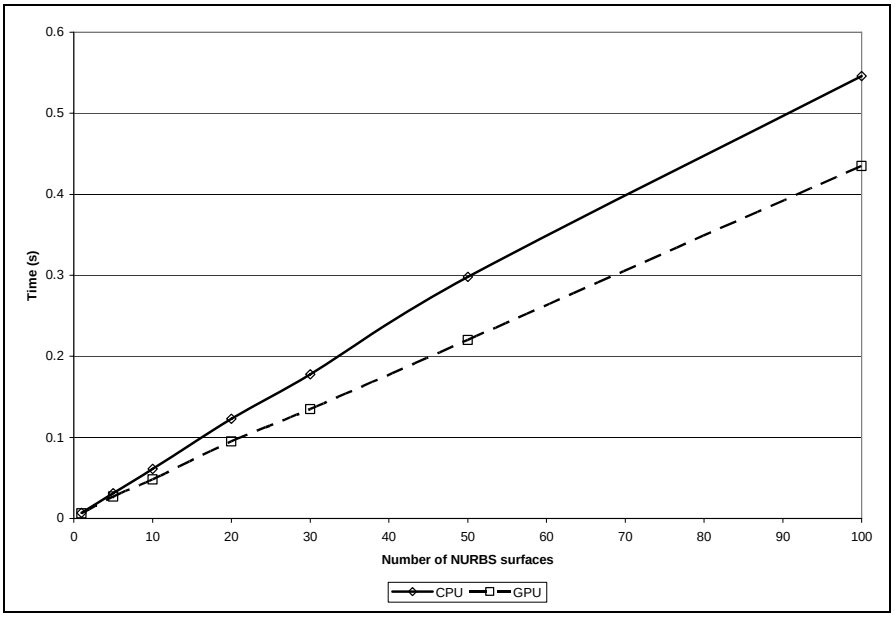

Figure 12: Timing comparison between GPU algorithm and OpenGL implementation

SOLID MODELING INTEGRATION

One of the advantages of the proposed method is that it can be easily integrated into existing solid modeling packages. For example, ACIS, a solid modeling kernel, splits the solid model into various levels of hierarchies. One of the top levels in the hierarchy is the "FACE" that stores information about the surface. The information that is required for generating the trim texture, which essentially consists only of the trim curves, is already present in the data structure. This information can be used for generating and mapping the trim texture while the model is being displayed. Thus, only the rendering portion of the solid modeling package needs to be changed to use the texture-mapping algorithm instead of the tessellation algorithm.

CONCLUSIONS

A better algorithm for displaying trimmed NURBS surfaces is proposed, which obviates the need to tessellate the surface for display purposes. This method, even though it uses GPU to some extent, does not shift all the calculations away from the CPU. The CPU is still required to evaluate the NURBS surface. All current implementation of NURBS evaluation on the GPU use some kind of approximations like converting the NURBS surface to bi-cubic Bezier surfaces. This is because a general NURBS surface can have a large number of control points and knots. Hence, an efficient method of passing this information to the GPU has to be found.

One possible method to evaluate the NURBS curves or surfaces can make use a property of the NURBS called "Local support." This property states that for any NURBS curve of order k, at any given parameter value, the NURBS curve is utmost blended with k control points. Hence, due to this property, only k control points and the corresponding k knot vectors need to be sent to the GPU for evaluating a point. Future work on this project will focus on using this property to evaluate the NURBS surface using the GPU.

REFERENCES

1. Michael Guthe, Akos Balazs and Reinhard Klein, “GPU-based trimming and tessellation of NURBS and T-Spline surfaces,” ACM Transactions on Graphics, Vol. 24, No. 3, August 2005.
2. Alyn Rockwood, Kurt Heaton and Tom Davis, “Real-Time Rendering of Trimmed Surfaces,” Computer Graphics, Volume 23, Number 3, July 1989.
3. The OpenGL Programming Guide - The Redbook