# Improving GPU based Evaluation and Rendering of NURBS

Course project report – Sushrut Pavanaskar – CS284 Fall 2009

## Contents

# Introduction

Non uniform B-Splines or NURBS is a popular representation of geometry (particularly curves and surfaces) in today's modeling world. Due to the inherent accuracy of rational B-spline curves, NURBS represent the geometry extremely accurately and yet are quite compact to represent. Not surprisingly, many solid modeling software such as Solidworks, use NURBS to internally represnt surfaces of models.

NURBS owe their accuracy to the elegant mathematical formulation of B-splines that they base on. Thus to display a NURBS based surface, one has to necessarily evaluate it first at certain points on the surface and then use those points to render it. This is a repetitive mathematical task, and owing to the complexity of models (models can have hundreds of surfaces) can become computationally intensive. Also, there is no hardware support for direct NURBS rendering offered by any of the leading vendors.

Saving grace to this scenario is a fact that alike B-spline surfaces, evaluation of a point on a NURBS surface involves computation of basis functions and then multiplication by values of control points etc. It does not, in any way, depend on the adjacent points on the surface. Further, the 3 coordinates X, Y and Z of a point on the surface are evaluated in exactly the same manner using the same basis function values. Necessarily, these conditions allow parallelism in evaluation.

Researchers in our lab have successfully demonstrated that NURBS surfaces can be efficiently evaluated and directly rendered using parallel processing algorithms on GPUs [1]. GPUs, which, inherently are tuned for parallel processing, can also perform non-graphics computational tasks as they can be programmed. This allows use of GPUs to implement parallel algorithms for NURBS evaluation and direct rendering.

However, parallel processing at various levels results in some issues. The model, which may have many surfaces, is built surface by surface and is displayed as a collection of those. There is no knowledge or consideration given to the neighboring surfaces when surface is displayed. Further, since surfaces can be smaller or larger, the parameterization of each may not be of the same size when seen in screen space. This causes artifacts at the boundaries where two or more such unknown neighbors meet. These are termed as "cracks", as highlighted by yellow circles in Figure 1.

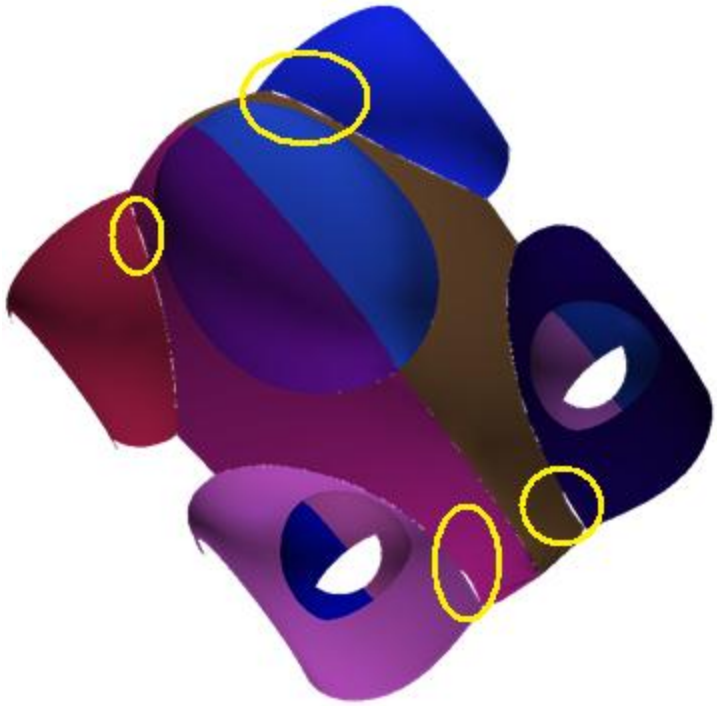

Figure 1 GPU rendered NURBS model with 12 surfaces and some "cracks" highlighted with yellow circles

In this project, I tried to work on removing these cracks to obtain a clean, artifact free display of a given model by using the direct evaluation and rendering code. The emphasis of this report is to enumerate the approaches I used, followed by the most plausible approach that was implemented conclusing with thoughts about the current state and known issues.

## Problem Statement and Scope

After judging the span of the project for this course, I decided to formally state the problem as "to remove cracks from a GPU-evaluated NURBS model to improve its visualization and rendering." I also decided to limit myself to mathematically correct geometric models supplied by the user (no holes in the model, all face normal pointing outwards etc.) in SAT file format and using all other features as developed by the researchers in [1].

Clearly, this is a clean-up type of task and thus as such does not add any "value". Therefore, I decided to limit myself to a simpler approach which will be essentially invisible (in terms of resources) and will work in nearly no time. I also decided to restrict myself to C++ environment and use existing data structure in the code to store geometric entities.

# Relevant literature

The problem of filling the cracks, somewhat similar to repairing a mesh with holes, has been tackled in various forms by a number of researchers in the literature. Following figures, show some of the popular ways. Generally, as shown in Figure 2, a triangle strip can be built in the cracked region or the cracked

region itself can be eliminated by artificially “welding” or merging the two vertices, which ideally should have been coincident.

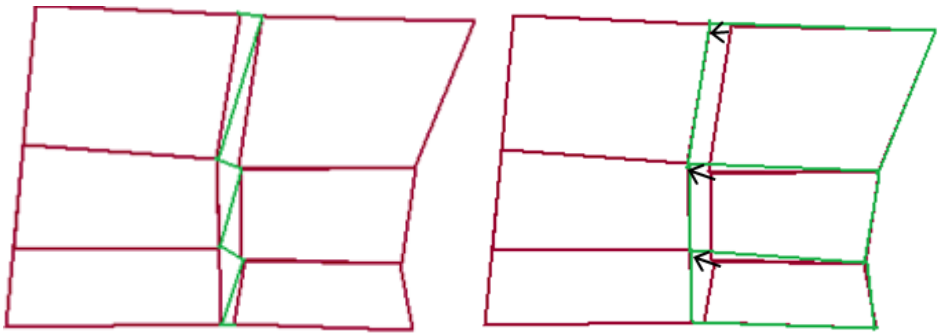

Figure 2 Triangle strip and welding of vertices

Both these approaches work better with one to one vertex correspondence. The triangle stip (or fan) or the merging operation needs to be computed with every new frame since cracks change every time the viewport changes (e.g. by zooming or panning). Further, triangle fans would result in tiny skinny triangles which could also hinder the view. Thus I decided not to use these techniques in this project.

## Original proposal and Initial Work

Initially, to ensure simplicity and reduced computational intensity, I thought of using an entirely 2D image processing based approach to this problem. I proposed to identify cracks by checking for color discontinuity in a n x n pixel neighborhood and then fill the crack pixel with the adjacent pixel color. Essentially, this would act like a n x n pixel mask, run over an image of the size of the viewport. However, I quickly realized that this was not sufficient. The coloring information was much more than just RGB (involved shading) and it changed with every zoom/pan/rotate operation. Thus the image processing approach was computationally intensive (the 9 pixel filter would have to operate each time the viewport image changed) and not good enough too since the pixels would look like a flat shaded surface defect.

Therefore I decided to work in the 3D model space and fill the cracks in model space so that when they went through later transformations, they would automatically generate a visually perfect, shaded surface. Also, model space operations would be needed just once unlike the image processing filter needed to operate every time the viewport changed.

### Outward growing of surfaces

After realizing that the cracks always appeared when two surfaces met, i.e. at the boundaries, I decided to follow a simplistic approach of growing the surfaces outwards beyond the boundaries. This meant that the surfaces would cover up the cracks and go even beyond. Since the models were 2 manifold, these additionally grown surfaces would not be seen (just like when one stitches of two pieces of fabric with an overlap, looking from outside the overlap is hidden inside). This is shown in Figure 3. But I observed difficulties with this approach as it created artifacts of additional floating regions when the two meeting surfaces had an acute angle between them. So I discontinued this approach and focused on filling the cracks only where they existed.

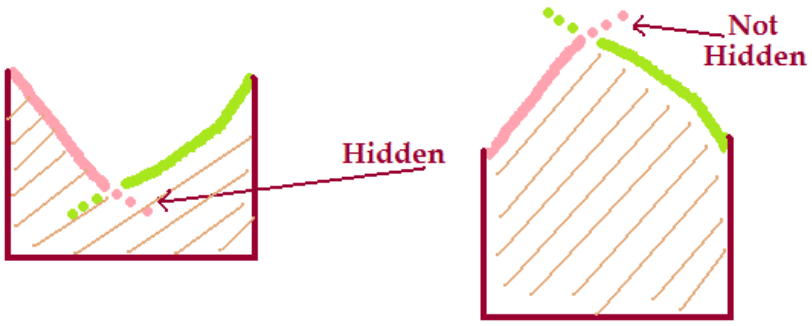


Figure 3 Problems with surface "growing"

## Cracks' surroundings

Cracks hinder the visualization of an originally water-tight model. Sometimes, through a crack, the background would show up or sometimes, another surface of the model would be seen.These 2 cases are shown symbolically in Figure 4. In rare cases, one of the two neighboring surfaces of a crack, if highly curved, would be seen from the crack. Sometimes there was a crack at the boundary on the silhouette and in that case it would just merge into the background.

Figure 4 Cracks with background seen in the left case and another (yellow) surface seen in the right case

Thus, merely looking at the neighborhood it was difficult to say whether a color-mismatching pixel was indeed a crack or not. An important observation still prevailed that the cracks always appeared around the boundaries (edges). This led me to work on edges for crack prevention. Inspired from the literature [2], I decided to use a similar approach of thickened edges as used in [2], to cover up the cracks. I decided to do this "thickening" in model space as reasoned in the earlier section.

## Implemented Solution of thick edges

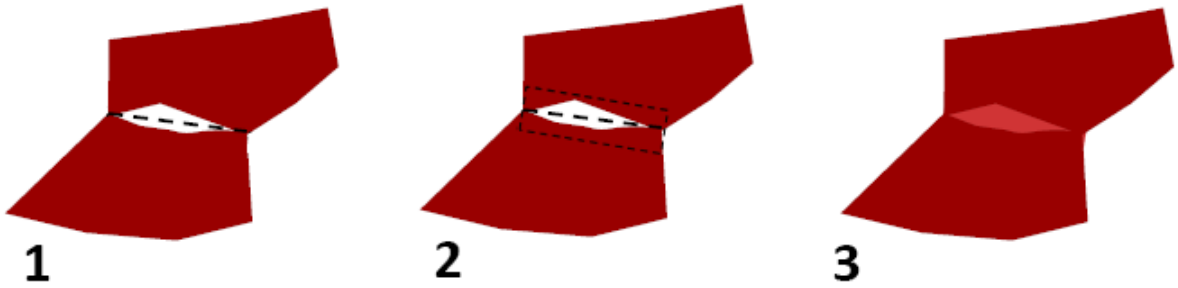


Figure 5 Thematic represenation of the approach

Algorithmically, I attempted to eliminate cracks in 3 steps as follows. In the first step, I read all the edges of the model and subdivided each to a 10 segment polyline. Then I developed 2 pixel wide quadrilaterals (edge-quads) around each segment of this polyline. Finally, during rendering, I implemented a GPU fragment shader which rendered only those parts of the edge-quad which were a "crack". Thematically, this is shown in Figure 5.

### Reading the Edges

I used ACIS APIs to read edges from a SAT file and stored it in my own data structure. The topmost entity is a vector called Edges. First I stored 10 parametrically equidistant points on the edge (edge-points). ACIS provides direct handles to these and computes them directly from the NURBS curve of the edge (hence they are accurate). Then I also stored IDs of the two neighboring surfaces of that edge. This step is important as I used these IDs later in the shader to determine a real crack. Due to particular format restrictions, even using APIs, this reading step was a huge learning step of the project.

### Developing the Quad

This is perhaps the most important step. From the screen space coordinates of the edge segment, I computed a 2 pixel wide quadrilateral around each edge segment. This operation was done is screen space and essentially was a 2D operation only in X, Y. Thus the quads were computed parallel to the viewport. Then I transformed the quad corners to model space coordinates so that I could draw the quads in model space. This step ensured consistency in coloring and was extremely important to have the cracks merge smoothly into the actual surface. The back and forth transformations of the coordinates was done using openGL view matrix and projection matrix (and their inverses). Thus after computing the quads, I actually drew the quads while drawing the whole model in the model space. I used triangle strips to draw the quads as it is more efficient to do so.

### Renderging using GPU shader

As targeted, I wrote the GPU shader to perform a mere comparison job and hence it worked extremely fast. The shader, implemented in CG, is supplied with a quad to draw and with each quad are supplied two IDs of the neibhboring surfaces to that edge. It compares each ID of the screen pixel to the two supplied IDs and renders a pixel in the quad only if the ID on the viewport does not match any of the two supplied neighbor IDs. Therefore a quad is rendered only when there is a crack and something other than the two neighboring surface is seen.

It may be noted, that in order to have the fragment shader do this comparison, it should originally have the IDs of all the pixels in the current viewport. Those are computed early in the code with a separate pass and stored as a 2 dimensional array on the GPU. Therefore no new CPU-GPUdata transfer was introduced in the original code.

## Early Results

Following result (Figure 6) shows cracks filled first with a different color just for illustration purpose (in the middle figure) and in the right most figure, with original edge colors. The model was purposefully evaluated at lower resolution to have clearly visible cracks. A few new issues came up after testing.

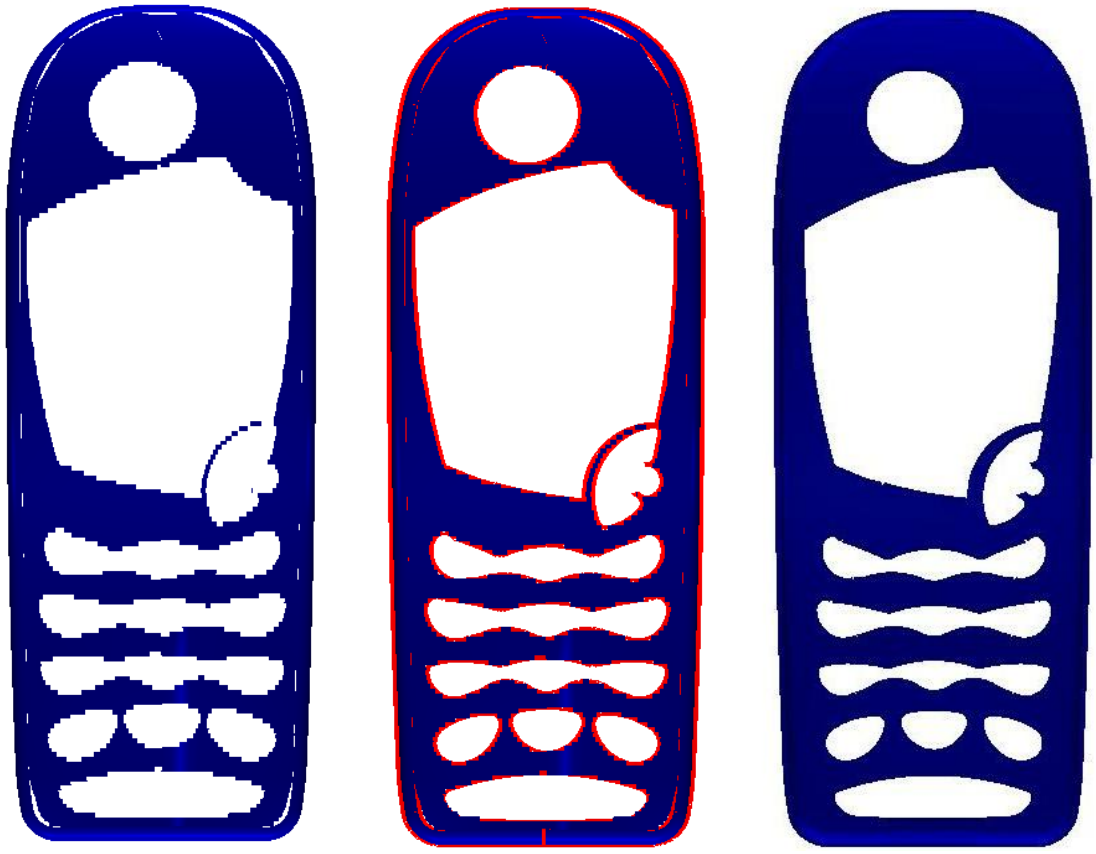

Figure 6 Early results of filling the cracks

Most importantly, there was an issue of quads being rendered at some unexpected points as marked by orange circles in Figure 7 below. Upon further study, I could claim that these were due to edge semgments on the silhouette. These additional quads were not exactly required and thus I implemented another fix to stop the shader from rendering them automatically.

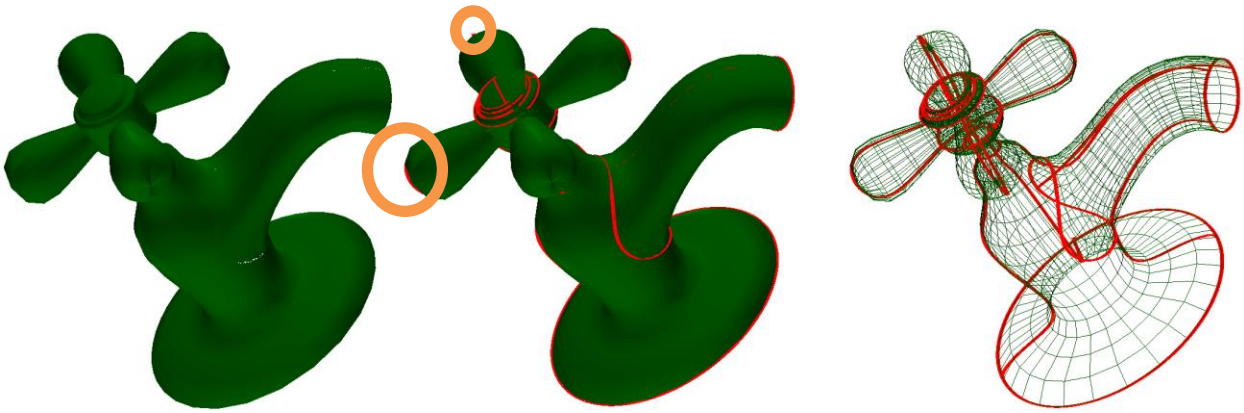

Figure 7 A Faucet model rendered after filling the cracks, with an unexpected region encircled in orange

## Silhouette Edges

As clearly seen in the figure, the edge on the silhouette need not be patched for cracks. Even if they had cracks, either the cracks were behind the actual object and so were not seen or the cracks merged with the background and hence un-noticed. Therefore, I attempted to find out such silhouette edge segments. To do this, I computed the normals of the two triangles of the edge quad around each segment and then compared these normals to the view vector derived from openGL. Their dot product, if less than 0 meant that the edge was on a silhouette and thus need not be patched. In such cases, I made the quad around that edge segment to be of zero width thereby avoiding artifacts.

## Improved Results

Figure 8 shows a model where a part of the circular edge is rendered with edge quads but rest of it is not. The edge near the horizon, i.e. the silhouette, is not rendered.

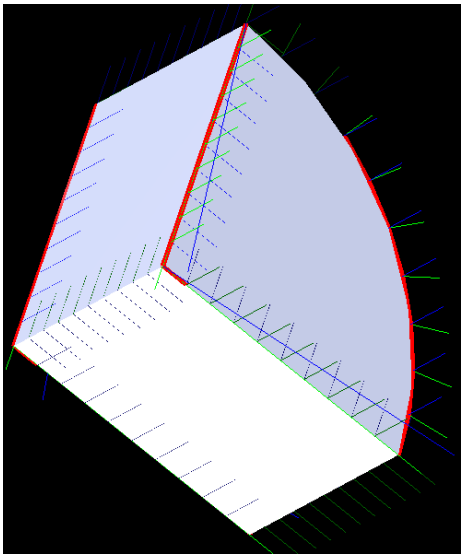

Figure 8 Result of a silhouette edge test

## Conclusion

From the current results and tests, it can be seen that this work of patching the cracks with edge quads has promise. It works with little computational time, and as such there is no effect seen while in operation. Though, I have not yet tested or profiled the code for actual use of the resources. The models before patching are significantly improved when the algorithm is implemented and show no visual cracks. Also special attention to coloring ensures that crack filling is “invisible”.

There still are a few unaddressed issues as there are cases where attention is needed. These include some areas where silhouette test fails. Also the implementation needs more attention in terms of memory leaks.

### Further thoughts

There are some areas where more work is required to make this code robust. First, the strategy to build zero width quads around the silhouette edge to avoid artifacts needs more work. Also, sometimes, particularly in areas of high curvature, the normals might be largly different even for two consective edge segments. This can lead to an edge being detected as silhouette (and so no quad may be rendered) but yet may have some portion visible thereby causing a crack.

Another thought is to develop the quads in the direction of the neighboring surface rather than parallel to the viewport. This will take care of all the silhouette issues automatically and also may not need a shader as artifacts would not appear. But to develop such quads in model space will need some more computations and/or data extraction from the model.

## User manual to test or use 3rd party SAT files

This code is developed in visual C++ and runs under Microsoft Visual Studio 2005 environment. Currently we have implemented the shaders in CG and we also use GLUT libraries for improved openGL visualization. To run the code, along with Visual Studio, openGL, GLUT and CG must be installed. Also the computer should have a programmable GPU (any dedicated video card of the current times is OK). The onboard graphics chip like Intel Media Graphics Accelerator, however, is not supported since it does not have a programmable GPU. NVIDIA GeForce GPUs (or equivalent ATI cards) are programmable (after version 6XXX) and so are supported. Also, for Nvidia chips, the computer should use a graphics driver version higher than 185.68.

To run the code, just open the solution file in MS Visual Studio environment and RUN. To change the model being evaluated and rendered, change the command line parameter in Project Options by right clicking the project name and selecting Debug properties.

To run the executable from command line, just input “GPView” and make sure that the SAT file model is in the same folder.

Once running, the GUI is developed to have normal mouse controls, zooming based on mouse wheel and panning is done by holding down the mouse wheel button. To rotate the model, use left click and drag. Actions caused by right click and other keyboard controls are beyond the scope of this project.

## References

[1] KRISHNAMURTHY, A., KHARDEKAR, R., AND MCMAINS, S. 2007. Direct evaluation of nurbs curves and surfaces on the GPU. In SPM ’07: Proceedings of the 2007 ACM symposium on Solid and physical modeling, ACM, New York, NY, USA, 329–334.

[2] MCGUIRE, M., AND HUGHES, J. F. 2004. Hardware-determined feature edges. In NPAR ’04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering, ACM, New York, NY, USA, 35–47.