GPU-Based Screen Space Tessellation

Jon M. Hjelmervik and Trond R. Hagen

Abstract. We propose a method for providing high quality view-dependent tessellations of certain types of parametric surfaces including B-spline surfaces. The objective is to ensure that, regadless of the camera positions, the rendered triangles will cover approximately the same number of pixels. In order to do this efficiently we use graphics hardware to sample and evaluate the surfaces. In contrast to previous work we use triangles as primitives in the rendering of the samples. This paper also describes certain artifacts that can occur and how to avoid these.

§1. Introduction

Due to the demand for advanced real time visualization in computer games, graphics cards have gradually evolved both with respect to performance and flexibility. The introduction in 2003 of programmable 32-bit floating point fragment processors into commodity hardware was a major breakthrough, because it led to both more advanced visualization and to the use of such processors for non-graphical purposes.

Visual inspection of surfaces is important when validating the quality of CAD surfaces. In this process the goal is not to produce a visually pleasing image, but rather to produce a correct rendering of the surface. Focusing too much on performance has a tendency to lead to the loss of important details.

The standard method for visualizing a parametric surface is to tessellate it (uniformly or non-uniformly) and render the triangulation. The rendering system performs the lighting calculation using vertex positions and normals, and linearly interpolates the color values across the triangles. A more advanced method is to use programmable fragment processors, in order to linearly interpolate the surface normals (or other relevant quantities) instead of the colors, and do the lighting calculation per pixel. This

xxx and xxx (eds.), pp. 1–4.

ISBN 0-9728482-x-x

strategy produces higher visual quality, but the correct color is still only obtained at the vertices.

Yet another method is to attach to each vertex in the tessellation the associated pair of surface parameter values and utilize a fragment shader in order to (correctly) evaluate the surface positions and normals. This is possible for various types of surfaces, including B-spline and subdivision surfaces [1]. While this improves both the visual quality and the correctness, we still have the problem that only the vertices are rendered correctly. Therefore it is necessary to use a tessellation where each rendered triangle only covers a few pixels on the screen. This paper presents a method that utilizes the graphics processing unit (GPU) to create a view-dependent tessellation, where each triangle covers approximately the same number of pixels.

§2. Related Work

2.1. View-Dependent Tessellation

There are numerous methods for view-dependent tessellation of parametric surfaces. These include hierarchical methods like quadtree-based triangulations [4] and progressive meshes [3]. Common to all methods is that the tessellation is computed on the CPU. A copy of the tessellation is kept in graphics memory, and must be updated continuously. Such methods may cause high CPU use, and the bandwidth to the GPU may limit the frame rate.

2.2. GPU-Based Surface Evaluation

Pre-evaluated basis functions are commonly used in efficient implementations of B-spline surface evaluation. Evaluation of sub-division surfaces with pre-evaluated basis functions utilizing a GPU is described by Bolz and Schröder [1]. The basis functions and the control points are stored in textures, and fetched by the fragment shader, where the surfaces are evaluated. The method is generalized to all surface types which can be written as linear combinations of basis functions. Our implementation of surface evaluation is based on their work.

2.3. Geometry Images

Geometry images were introduced by Gu et al. [2] as a method for representing triangulations as images. To create a geometry image the triangulation is cut such that it becomes topologically equivalent to a disk. This cut version is then parametrized on a rectangle before it is sampled, one sample per pixel in the image. This representation allows the triangulation to be stored without any topological information.

A similar technique is that of Multi-chart geometry images by Sander et al. [5]. The triangulation is cut into several charts, such that each chart is topologically equivalent to a disk. Each chart is then parametrized and sampled before the charts are rasterized and stored in a texture. When reconstructing such an image G^0 continuity only holds within each chart. Therefore zippering steps are necessary to ensure that the reconstructed triangulation is continuous. We create similar charts in our method. These charts are sampled on the GPU and the resulting image is used directly as vertices for rendering, without being processed by the CPU. It is therefore undesirable to do zippering. Instead, we create images that lead to a continuous triangulation.

2.4. Per Pixel Correct Rendering

Yasui and Kanai [6] proposed a method for using the surface evaluation capabilities [1] of modern GPUs for correct rendering of surfaces. Loosely speaking their method renders a coarse tessellation of the surface using a fragment shader which evaluates the surface using the linearly interpolated parameter values, and “moves” the fragment into the correct position of the frame buffer. Since the fragment processor does not have the freedom to choose which pixel it is writing to, i.e. moving the fragment, the algorithm is split into the following steps.

First the tessellation is rendered using a fragment shader that returns the parameter values (linearly interpolated across each triangle). The positions and surface color (calculated based on position and normal) are then calculated and stored in separate off-screen buffers, which in turn are converted into vertex arrays. The vertex arrays are then rendered as points. Since one of the vertex arrays contains the evaluated positions, the points are drawn into the correct positions in the frame buffer.

The result is an image where each pixel is colored based on the “moved” position and the surface normal evaluated accordingly. As the final image is generated by rendering points, it may contain holes. The solution proposed is to enlarge the off screen buffers used, which in turn will increase the number of points drawn. This will reduce the probability of holes, but the amount of enlargement needed is not clear. Only points originating from parameter values of visible geometry is rendered. This may in some cases cause incorrect rendering or holes in the geometry.

We propose a method inspired by Yasui and Kanai [6], which aims to remove these artifacts. In addition our method ensures that the resulting image does not contain holes or other defects.

§3. GPU-Based Screen Space Tessellation

Our method creates a new tessellation based on an initial coarse tessellation $T(u, v)$ (a piecewise linear interpolation) of a parametric surface $S(u, v)$. The goal is that each new triangle covers a predefined number of pixels independent of the distance to the camera. The new vertices

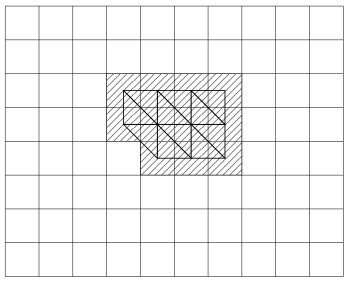


Fig. 1. Rasterization of a surface, which leads to 11 triangles.

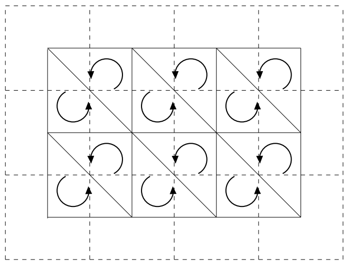


Fig. 2. Dashed lines mark boundary between the six pixels, solid lines mark edges of the resulting triangles and the arrows show the ordering of the vertices in the triangles.

are created using a similar method to that of Yasui and Kanai [6], but instead of treating the vertices as points, we treat them as vertices of a triangulation, as illustrated in Figure 1. Therefore our method does not leave gaps in the surface when the neighboring vertices “moves away” from each other.

Since each vertex is lit correctly and each triangle is small the resulting image has acceptable quality. Further the small triangles ensures that there will not be noticeable artifacts like popping and flickering.

3.1. Tessellation

In order to re-tessellate, T is rendered into two off-screen buffers. In the first buffer we store the parameter values, and in the second we store S evaluated at these parameter values. Both these buffers are then converted into vertex arrays, i.e. the pixels of each buffer are converted into a one-dimensional array of vertex attributes.

Since the resulting vertices originate from pixels, they are arranged in a regular grid. It is trivial to create quads between all sets of four neighboring vertices. In order to split each quad into triangles, we choose the diagonal from the lower right to the upper left vertex. The vertices are then rendered in counter clockwise ordering, as illustrated in Figure 2.

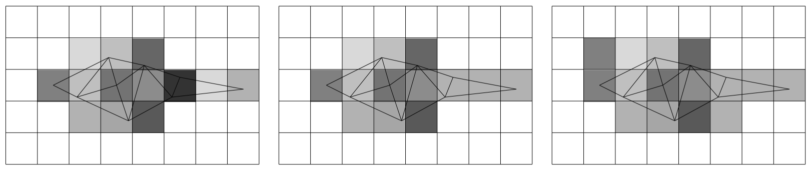
We use the alpha value in the position buffer to mark whether a pixel is modified or not by the rendering of T. When rendering the new tessellation we use a vertex shader that uses this information to detect if the vertex is valid (originates from a modified pixel). If the vertex is valid the position is transformed by affine transformation defined by the fixed function matrices, otherwise the position is set to $(0, 0, -\infty)$. This ensures that triangles containing invalid vertices are degenerated and not rendered. Observe that this also holds for triangles containing only one invalid vertex because the triangle becomes parallel to the viewing direction.

The main steps of the algorithm are:

1. Set up the graphics system for rendering to multiple render targets (rendering to more than one off-screen buffer). Enable a fragment shader that evaluates S. The fragment shader outputs the evaluated positions into one buffer and the parameter values into the other buffer.
2. Clear the position buffer such that the alpha value is set to zero.
3. Render the initial coarse tessellation T.
4. Convert the off-screen buffers into vertex arrays.
5. Set up the graphics system for rendering to the frame buffer. Enable the vertex shader that detects if the vertex is valid, and sets the position accordingly. Enable a fragment shader which evaluates the surface normal of S, and does the lighting calculation based on this normal. Set up the graphics system to use the vertex arrays from Step 4 such that the vertex arrays containing positions and parameter values are used as vertex position array and texture coordinate array respectively.
6. Use a predefined index array in order to render the new tessellation.

3.2. Objects with boundaries

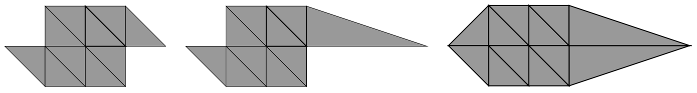
When dealing with objects that have boundaries various issues arise. The midpoint of each fragment is used in the linear interpolation process of the parameter values. Therefore the parameter values at the fragments nearest the boundary will generally not belong to the boundary of the object. This would result in a tessellation which is shrunk related to S and have a jagged boundary. This is solved by rendering the boundary edges as one-pixel wide lines. Often there are special points along the surface boundary which the new tessellation should interpolate, e.g. the corners of the parameter domain. These points should also be rendered.



(a) The initial tessellation is shown together with the rasterization of its triangles and boundary edges.

(b) The pixels drawn into by more than one of the edges connected to the leftmost vertex, are replaced by the vertex.

(c) Boundary repairing finished.



(d)

(e)

(f)

Fig. 3. Figure (a), (b) and (c) shows the rasterization of the initial tessellation T, and how the off-screen buffers are modified near the boundary. Figure (d), (e) and (f) shows the topology of the resulting tessellation.

Near a vertex where the angle between the (projected) boundary edges is less than 45° the pixels may form unfortunate patterns such that corresponding vertices are discarded during triangulation, because all triangles they are part of are degenerate. The three pixels furthest to the right of Figure 3(a) form such a pattern. The pattern occurs when the edges on both sides of a vertex are drawn into the same pixels. We detect the vertex and the connected edges on the CPU. By utilizing the stencil buffer we detect the pixels that are drawn into by the edges on both sides of the vertex and replace them with the parameter value of the vertex, as illustrated in Figure 3(b). Notice that a consequence of this is that the triangles containing the vertex are somewhat enlarged.

As mentioned we tessellate by connecting neighboring pixels as illustrated in Figures 1 and 2. In the cases where three of four neighboring pixels are drawn into by the rendering of T they may not form a triangle, which cause the boundary to be jagged, see Figure 3(e). To solve this we use a fragment shader that detects when this happens, and copies one of the neighboring pixels into the otherwise empty pixel. Figure 3(f) shows the result after the steps to correct the problems that have been made.

§4. Hidden Geometry

The resulting tessellation only contains vertices that correspond to parameter values that are associated with visible triangles of T. This may cause parts of the image to remain blank.

4.1. Outside the Viewport

A point $S(u_0, v_0)$ may be within the viewport while $T(u_0, v_0)$ is outside the viewport of the off-screen buffer. Then the screen space tessellation will not cover this part of the surface, and the boundary of the viewport will remain blank. This can be avoided by expanding the viewport in the off-screen buffer until the viewport covers the missing parts of T. By calculating a upper bound of $\|S(u,v) - T(u,v)\|$ it is possible to determine the necessary expansion.

4.2. Silhouette Issues

Silhouette curves are important features of a surface, and defects in the visualization of these curves give a false impression of the surface. A feature of our method is that it provides high quality silhouette curves.

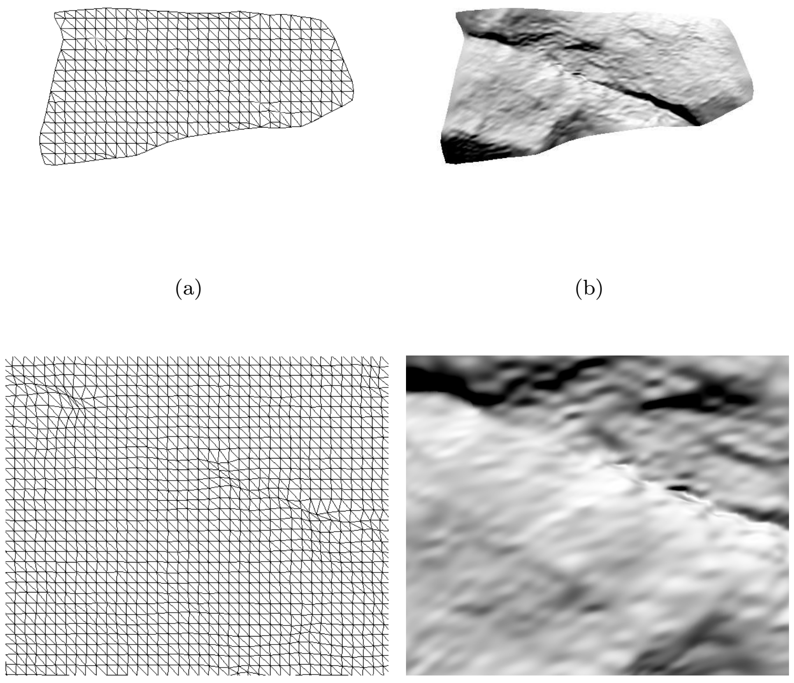
Special care must be taken to ensure that a silhouette curve c_T on T does not hide parts of its corresponding silhouette curve c_S on S, i.e. a point $S(u_0, v_0)$ on c_S has its corresponding point $T(u_0, v_0)$ on a back facing triangle. Otherwise there may be missing geometry near silhouette edges in T where the front facing triangles occlude the back facing triangles. In order to avoid this we extend the surface across such edges, thus creating a band of triangles containing the parameter values of the back facing triangles near the silhouette edges.

4.3. Occluded Geometry

From a given view direction the surface may consist of several layers of geometry, such that parts of the surface are occluded by layers closer to the camera. It is important to ensure that points from different layers are not falsely connected in the new tessellation. Therefore we split T into charts such that each chart only consists of one layer of geometry.

Each chart may contain points from different layers of T therefore care must be taken so that these points are not connected falsely. This can happen when points that are not neighbors in the chart are rendered into neighboring pixels. We avoid this by requiring that each chart projected onto the viewport is not self-intersecting within an intersection tolerance of one pixel. The algorithm should aim at avoiding angles less than 45° along the boundaries of the charts, because this may lead to larger triangles in this area.

To improve the performance it is desirable to render all charts into the same off-screen buffers. We do this by estimating the size of each chart and then pack the charts into non-overlapping domains of the off-screen buffers.



(a) (b)

(c) (d)

Fig. 4. Cubic spline surface, (a) and (b) shows the entire surface, (c) and (d) are close-ups.

§5. Conclusion and Future Work

Our method provides high quality view-dependent tessellations which are suitable for surface inspection of parametric surfaces, as illustrated in Figure 4. We also ensure that the final rendering of the surfaces is without gaps. A coarse tessellation of a free form surface usually captures the main characteristics of the surface. The geometry of such a tessellation is often simple. Therefore real-time performance is obtained using simple methods for detecting silhouette curves and splitting the initial tessellation into charts.

View-dependent tessellation is an important topic in visualization and is useful in numerous applications. By adopting state of the art methods for creating and packing the charts, our method can be used when the initial tessellation is of complex geometry. We believe the method can be of special interest in the topic of terrain visualization, where view-dependent tessellation is commonly used.

§6. References

1. Bolz. J., and P. Schröder, Evaluation of subdivision surfaces on programmable graphics hardware, submitted for publication, 2003.
2. Gu, X., S. J. Grothler, and H. Hoppe, Geometry images, ACM Trans. on Graphics **21** (2002), 355–361.
3. Hoppe, H., View-dependent refinement of progressive meshes, in *Proceedings of SIGGRAPH 97*, ACM Press, 1997, 189–198.
4. Lindström, P., D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner, Real-time, continuous level of detail rendering of height fields, in *Proceedings of SIGGRAPH 96*, ACM Press, 1996, 109–118.
5. Sander, P. V., Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe, Multi-chart geometry images, in *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing*, Eurographics Association, 2003, 146–155.
6. Yasui, Y., and T. Kanai, Surface quality assessment of subdivision surfaces on programmable graphics hardware, in *Proc. International Conference on Shape Modeling and Applications 2004*, IEEE CS Press, Los Alamitos, CA, 2004, 129–136.

Jon M. Hjelmervik
Sintef ICT, Applied Mathematics
Oslo, Norway
`jon.a.mikkelsen@sintef.no`

Trond R. Hagen
Sintef ICT, Applied Mathematics
Oslo, Norway
`Trond.R.Hagen@sintef.no`