Direct Pixel-Accurate Rendering of Smooth Surfaces

Jon Hjelmervik

Sintef ICT and University of Oslo
Jon.M.Hjelmervik@sintef.no

Abstract. High-quality rendering of B-spline surfaces is important for a range of applications. Providing interactive rendering with guaranteed quality gives the user not only visually pleasing images, but also trustworthy information about the model. In this paper we present a view-dependent error estimate for parametric surfaces. This estimate forms the basis of our surface rendering algorithm, which makes use of the hardware tessellator functionality of GPUs.

We use the screen space distance between the tessellated surface and the corresponding surface point as an error metric. This makes the algorithm particularly useful when visualizing additional attributes attached to the surface. An example of this is isogeometric analysis, in which simulation results are visualized along with the surface.

1 Introduction

Smooth surfaces are used in settings ranging from the entertainment industry to CAD applications. In the entertainment industry, the model's sole purpose is to create visually pleasing images. The CAD-related industry, on the other hand, use visualization both to get an overview of models and to investigate their geometric qualities, for example the smoothness of a car's body. Most CAD models are at some point also used in an analysis setting, e.g., stress analysis, to investigate the model's physical properties. It is therefore a great demand for fast rendering methods that provide accurate, and visually pleasing results that includes associated data such as textures and simulation results.

One way of computing a correct rendering of a surface is to find the first intersection between the surface and rays originating from a virtual camera. This is called ray-casting, which is computationally expensive. GPUs are designed to rasterize triangles and rendering performance is often dominated by the number of triangles in a scene. Therefore, the main challenge is to determine which triangles to draw, and how to invoke their rendering the main challenge.

Single-pass rendering is the most common way to render triangles, because each object is sent only once through the rendering pipeline. *Multi-pass* methods, on the other hand, write partial results to a framebuffer (usually an off-screen buffer) for use in a succeeding rendering pass. A GPU delivers its best performs when it can render a large set of triangles in parallel. Multi-pass algorithms therefore impose a performance penalty when a rendering pass must wait for the

M. Floater et al. (Eds.): MMCS 2012, LNCS 8177, pp. 238–247, 2014.
© Springer-Verlag Berlin Heidelberg 2014

previous the completion of a previous pass. A CAD model can be composed of a large set of parts, each represented by a number of boundary surfaces. In such settings, with hundreds or thousands of surfaces it may be difficult to implement multi-pass algorithms without adding a large overhead due to pipeline stalls.

We propose a single-pass rendering method driven by the guarantee of pixel accurate rendering. Obviously, this indicates a view-dependent tessellation, i.e., the surface tessellation depends on the location and orientation of the scenes virtual camera. Our focus is an algorithm suitable for isogeometric models.

1.1 Related Work

Traditionally, algorithms could either provide error guarantees *or* be interactive. Filip et al. [1] proposed to use bounds on the second derivatives to create a semi-uniform tessellation of a C^2 continuous surface. The main idea is to split the surface into a set of patches, and find the tessellation levels independently for each patch. In order to create a watertight tessellation without cracks, each patch boundary has a separate tessellation level matching abutting patch edges. Their implementation is CPU based, but their approach fits very well with the OpenGL tessellator, and will be the basis for our work.

Cook et al. [2] propose to recursively split a surface until each triangle is less than a pixel. Their tessellations will for most surfaces be much denser than required. A fast CPU based algorithm for generating these tessellations was presented by Fisher et al. [3]. Since the triangles is recomputed based on the view position, they must be transmitted through the PCI-express bus each frame, limiting the rendering speed.

If we scarify guaranteed accuracy, there are numerous methods for view-dependent tessellation of smooth surfaces. Guthe et al. [4] proposed to use the CPU for deciding the required tessellation level of a semi-uniform tessellation, and use first generation shader technology to evaluate the surface. Hjelmervik and Hagen [5] proposed a two-pass algorithm using the GPU both for determining tessellation level and the evaluation itself. It is based on early GPUs, and is therefore not optimized for graphics hardware of today.

Lutterkort [6] developed an algorithm for computing piecewise linear enclosure of polygonal surfaces, called slefes. Yeo et al. [7] developed a interactive rendering algorithm with guaranteed pixel accuracy based on these slefes for DX11 compatible hardware. Their algorithm use separate rendering passes for generation of slefe boxes (estimate of the surface's deviation from linear), determining tessellation level, and the tessellation itself.

1.2 Hardware Tessellator

Hardware tessellators as exposed by OpenGL 4 and DirectX 11 are used to create semi-uniform tessellations. It lets us concentrate on error estimates, tessellation levels and surface evaluation, without concern for generating the triangles and managing their connectivity.

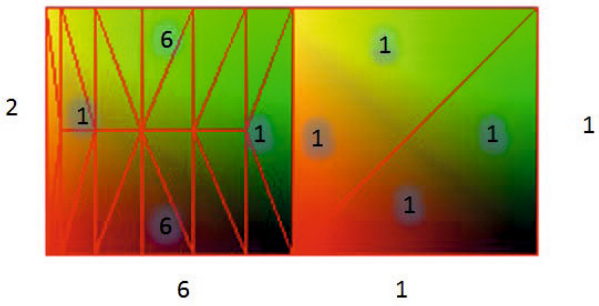


Fig. 1. Example tessellation with interior and boundary tessellation levels. Note that it generates triangulation with consistent connectivity.

The hardware tessellator allows for implementations in which the entire tessellation is performed at the GPU, and the triangles are directly rasterized without being transferred to off-chip memory. The tessellator acts on individual surface patches, which in our case will be the same as the Bézier patches. Each patch is tessellated individually, as illustrated in Figure 1.

The hardware tessellator adds two programmable shader stages to the existing OpenGL graphics pipeline that all triangles undergo. The first stage, the *tessellation control shader*, controls the tessellation process by specifying the parameters for the sampling density for each patch. A set of triangles complying with these criteria is then automatically created, and the *tessellation evaluation shader* has the responsibility to compute the position of each new vertex, based on its parameter value. To ensure that the triangles form a valid triangulation without holes, the the control shader specifies the sampling density along each boundary edge in addition to the parameter directions.

Note that all sampling parameters are set in the control shader, before any triangle is created. Traditional algorithms based on iteratively refining the triangles do therefore not fit with this setup. To fully take advantage of the hardware tessellator, the triangles should never leave the chip, meaning the tessellation will have to be redone each frame.

1.3 Contribution

Our research was performed independently of work by Yeo et al. [7] based on slefes, but contains many of the same features. Both methods use the same error metric to determine the required tessellation level, but estimate the error in different ways. Where Yeo et al. use slefes, where the theory is only fully developed for polynomial surfaces, our algorithm can be used for any C^2 continuous surface with bounds on the second order derivatives. This also allows our approach to be extended such that the lowest possible tessellation is less than one triangle per Bézier patch.

CAD-models often consist of a large number of objects, each described by their boundary surfaces. Special surfaces such as cylinders and swept surfaces, which are of different polynomial order in the parameter directions, play important roles in CAD. However, when used in a simulation it is common to use the same polynomial order in both parameter directions. Our algorithm allows for more dense sampling in the parameter direction with the highest second order derivative. Furthermore, in contrast to slefe based tessellation, all computations are performed after camera transformation, and thus, the tessellation is therefore ignorant to which space the object is modeled in.

As a starting point, we took the algorithm for semi-uniform tessellation described in Filip et al. [1], and analyzed how the error estimate is affected by projecting the surface to the screen. Their implementation, guarantees that the tessellation is within the error tolerance at the uniformly tessellated interior of each patch, and at the boundary curves separating the patches. However, the ring of triangles connecting the patch boundary with its interior may not. Some of our test cases included patches where one boundary curve was linear, and hence sampled only at its endpoints. This leads to visual artifacts, which is remedied in our implementation.

2 Algorithm

To use the tessellator we need a predicate which defines the tessellation level without the need of actually sampling the surface. One of the simplest predicates is to measure the size of the surface's bounding box when it is projected to the screen. Such a predicate would lead to overtessellation of flat surfaces or undertessellation of the more complex parts. We therefore need a view-dependent predicate taking into account the shape and parametrization of the surface.

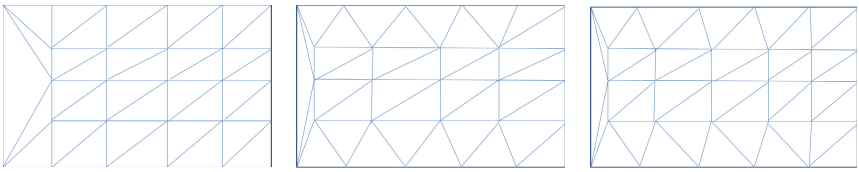
The rasterization solves the ray-casting problem of finding the intersection between a ray through each pixel center and the triangle. The texture coordinates (or parameter value) at a pixel is therefore associated to its center. Since we can use this parameter value to evaluate the surface and any additional data (textures of simulation data), it will be pixel accurate if the surface evaluated at the given parameter value belongs within the same pixel. We will use this property to define our error metric as follows:

$$e(u,v) = \|\mathtt{proj}(S(u,v) - T(u,v))\|_\infty. \tag{1}$$

Here S is the original surface, T is the tessellated version and `proj` is the projection from eye space to the screen. In this section we will ignore the projection and focus on the approximation error of a linear interpolation of a C^2 continuous surface.

A well known upper bound for linear interpolation of a C^2 continuous curve g is

$$|I_2 g - g|_\infty \leq \left(\frac{\Delta^2}{8}\right) \max |g''|, \tag{2}$$



(a) Original tessellation (b) Interior samples moved closer to the left boundary (c) Final tessellation

Fig. 2. Figure (a) shows a tessellation where the left boundary has only two sample points. Figure (b) shows how to move the interior sample points closer to the left boundary. Finally, in (c) an extra column of triangles is inserted.

in which $I_2 g$ is a linear interpolation of g with sampling distance Δ. Thus, we can choose the sampling distance to meet any given tolerance.

The interior of each patch is tessellated by triangles in which two of the edges follow the parameter directions of the surface. Therefore, we we can estimate the approximation error by

$$|I_2 f - f|_\infty \leq \frac{\Delta_u^2}{8} \max |f_{uu}| + \frac{\Delta_u \Delta_v}{4} \max |f_{uv}| + \frac{\Delta_v^2}{8} \max |f_{vv}|, \qquad (3)$$

where Δ_u and Δ_v are sampling distance in parameter direction u and v respectively. Again, if we can find upper bounds of the second order derivatives we can adjust the sampling distances to meet any given error tolerance. However, since there are two unknowns and only one requirement, the solution is not unique. The optimal solution is the one that has the lowest triangle count, whilst still fulfilling the error tolerance. Filip et al. chose different sampling densities in each parameter direction based on the estimates of the second order derivatives, while You et al. decided to use the same sampling densities. We use a greedy iterative process to determine the sampling distances.

In contrast to Filip et al., we experienced a breach of pixel accuracy for triangles adjacent to the patch boundaries. Any adjustment to the sampling density or position along the patch boundaries would create cracks in the tessellation, because patch boundaries are shared by the neighboring patch. However, in the interior of the patch, we may apply an transformation to the parameter values of each sample point before evaluating the surface. We chose to define an affine transformation that will narrow the boundary band as illustrated in Figure 2. Each patch is sampled uniformly in the inner, resulting in a uniform tessellation. The shapes of the boundary triangles are implementation specific, making it impossible to make a hard guarantee on the error in this area. We therefore assume that each triangle has one edge along the boundary curve and one edge parallel to the other parameter direction when computing the width of the boundary band.

Moving the interior samples closer to an edge will increase the size of the interior triangles, which may violate the pixel correctness. It may therefore be

necessary to increase the number of interior triangles. Remember that the tessellation levels and the affine transformation is computed based on information of the second derivative, without the need of actually generating the triangles to check their feasibility. Therefore, all decisions are made in the control shader. The control shader therefore performs the following steps:

1. compute the sampling distance for each edge
2. compute the height for each boundary band
3. compute the interior sampling distance
4. use the computed sampling distances to set the tessellation levels and affine transformation if necessary.

The hardware tessellator will create the required triangles and call the evaluation shader for each generated vertex, which applies the affine transformation to all interior points before evaluating the surface. Evaluation of B-spline surfaces in a shader is straight-forward and explained in Guthe et al. [4].

3 Pixel Accurate Tessellation

In Section 2, we described a tessellation algorithm that generates a tessellation with a guaranteed maximal distance from the original C^2 continuous surface. However, the error metric did not take into account the viewing distance or view direction, leading to a static tessellation. In this section, we will study the same error metric evaluated in screen space instead of model space, i.e., the error in terms of pixels on the screen.

3.1 Projected Error

As a first step, we study what a perturbation of a point in eye space leads to as a perturbation in screen space. Vertices in a 3D scene are represented by four dimensional homogeneous coordinates, and the x, y and z components are divided by the w component as a part of the fixed function perspective division in GPUs. The homogeneous coordinates allows perspective projection to be formulated as a matrix-vector multiplication. Here, we use we use the OpenGL projection matrix, which can be written as

$$\begin{bmatrix} A & 0 & B & 0 \\ 0 & C & D & 0 \\ 0 & 0 & E & F \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_{es} \\ y_{es} \\ z_{es} \\ w_{es} \end{bmatrix} = \begin{bmatrix} x_{cs} \\ y_{cs} \\ z_{cs} \\ w_{cs} \end{bmatrix}, \tag{4}$$

where $\mathbf{x_{es}}$ and $\mathbf{x_{cs}}$ are vectors in eye space and clip space respectively, see Shreiner et al. [8] for details. Let

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}, \text{ and, } \hat{\mathbf{x}} = \begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ \hat{y} \end{bmatrix} \tag{5}$$

be a point in eye space and its projection to the screen respectively. Then

$$\hat{y} = -\frac{Cy + Dz}{z}, \text{ and its inverse, } y = -\frac{\hat{y}z + Dz}{C} \tag{6}$$

describes the relationship between the y component in eye space and its projected counterpart. Let ϵ be the perturbation of $\mathbf{x}$ and $\hat{\epsilon}$ be the corresponding perturbation of of $\hat{\mathbf{x}}$. Then their relation can be written

$$\frac{y + \epsilon_y}{z + \epsilon_z} = -\frac{\hat{y} + \hat{\epsilon}_y + D}{C}, \tag{7}$$

which leads to

$$\hat{\epsilon}_y = C\frac{\epsilon_z y - \epsilon_y z}{z(\epsilon_z + z)}, \tag{8}$$

which describes the projected perturbation given by position and perturbation before projection. Assuming ϵ_z is neglectable in the term $z(\epsilon_z + z)$ we arrive at

$$|\hat{\epsilon}_y| \lessapprox C\left(\left|\frac{\epsilon_z y}{z^2}\right| + \left|\frac{\epsilon_y}{z}\right|\right). \tag{9}$$

What remains is to use the estimates from Section 2 to express the error projected error by the bounding box of the surface and its second order derivatives.

Inserting (2) into (9) and solving for Δ, we arrive at

$$\Delta \lessapprox \sqrt{\frac{8\epsilon^y}{C} \Bigg/ \left(\frac{\lceil g_z'' \rceil \lceil y \rceil}{\lfloor z^2 \rfloor} + \frac{\lceil g_y'' \rceil}{\lfloor z \rfloor}\right)} \tag{10}$$

as the expression for the maximal sample distance of a curve, given tolerances ϵ_x and ϵ_y. Here, $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ denotes the minimal and maximal absolute value respectively. To restrict the error to be less than one pixel, ϵ_y and ϵ_x are set to $0.5/window\_width$ and $0.5/window\_height$ respectively.

For the curve case, we were able to derive an explicit formula for the tessellation levels. For the surface case we must first choose a strategy for balancing the tessellation levels in the two parameter directions. Clearly, the tessellation level in the interior of the Bézier patch must be at least as dense as the sampling of the boundary curves. We propose to use the most dense boundary tessellation in each parameter direction as an initial guess, and iteratively refine the parameter direction that reduces the approximation error the most. The maximal error given the tessellation level for each parameter direction can be estimated by inserting (9) into (3),

$$\begin{aligned}\frac{8\epsilon^x}{A} \geq\ & \Delta_u^2\left(\frac{\lceil z_{uu} \rceil \lceil x \rceil}{\lfloor z^2 \rfloor} + \frac{\lceil x_{uu} \rceil}{\lfloor z \rfloor}\right) \\ & + \Delta_v^2\left(\frac{\lceil z_{vv} \rceil \lceil x \rceil}{\lfloor z^2 \rfloor} + \frac{\lceil x_{vv} \rceil}{\lfloor z \rfloor}\right) \\ & + 2\Delta_u\Delta_v\left(\frac{\lceil z_{uv} \rceil \lceil x \rceil}{\lfloor z^2 \rfloor} + \frac{\lceil x_{uv} \rceil}{\lfloor z \rfloor}\right).\end{aligned}$$

For most cases the iteration terminates quickly and does not represent a bottleneck in the control shader.

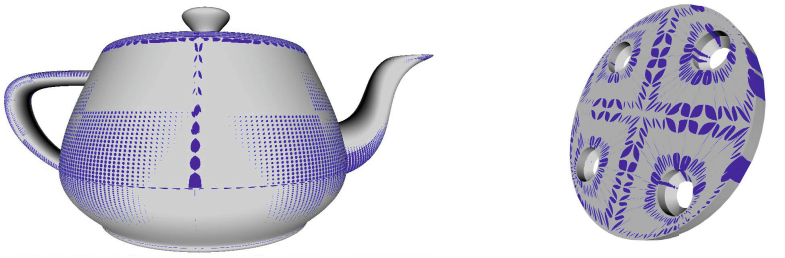


Fig. 3. Surface color illustrates how close the approximation error is to the given tolerance. Gray is used when the error is less than 10% of the tolerance, and blue indicates that the error larger. The error does not exceed the given tolerance.

4 Results

The tessellations generated by this algorithm are expected to be without any visual artifacts, as it provides pixel accurate rendering. What remains to be seen is whether the objects get excessively tessellated, meaning that more triangles than necessary are generated. It is also interesting to see which areas will have the highest triangle density.

In our experience, the tolerance is only violated when the calculated tessellation level is higher than the hardware limit of the GPU. To measure the quality of the tessellation we study the maximal error at each patch. If the maximal error is much less than the tolerance it indicates that the model is excessively tessellated. Figure 3 is colored based on the error relative to the given tolerance. Note that the error is largest along the boundary band, which is expected since these triangles may be larger than the interior triangles. Due to the shape of the boundary triangles the error estimate does not apply there, but experiments show good results here as well.

As expected, the result is pixel accurate and any tessellation algorithm with this property will produce indistinguishable images. The main objective of our work is to provide fast, high-quality, reliable rendering, which may also be achieved by relaxed the error tolerance beyond one pixel. The increased tolerance will improve the rendering speed, while keeping the assurance that the result will be within the given tolerance of the real model. Coarse tessellations near silhouette edges are easily detected. Several algorithms, including Dyken et al. [9] target this problem directly, and refines near silhouette triangles. As shown in Figure 4 no such special treatment is required here, as Bézier patches near silhouette edges generate smaller triangles compared to other areas. The triangles are also concentrated in high-curvature areas. Using a GeForce GTX 580 we are able to tessellate 7 million cubic Bézier patches per second, which is less than Yeo et al. [7]. This is both due to our algorithm being more compute intensive and due to lack of optimization of our code.

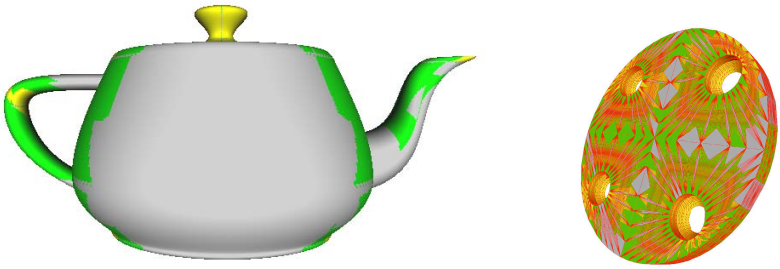


Fig. 4. Color encoding of triangle sizes. Gray triangles have maximal edge length of more than 5 pixels, yellow have 5-2.5, and green triangles less than 2.5 pixels.

5 Conclusion and Future Work

This work has been performed in parallel to the pixel accurate algorithm by Yeo et al. [7] with an almost identical goal. Their work was published at the time of writing this paper, making it natural to discuss the main differences which come from different strategies for estimating the rendering error. We require a one-pass algorithm to facilitate easy integration with large number of simple surfaces, and our focus is on CAD models.

Single-pass algorithms such as ours avoid latency introduced by multi-pass rendering, which is an advantage when rendering small surfaces. However, the control shader must recompute the tessellation levels based on surface coefficients each frame making it potentially slower for large models due to repeated computations. Also, patches sharing an edge will always choose compatible tessellation levels, removing the need to store adjacency information for the model to reach watertight tessellations.

The resulting tessellations from both approaches appear to be of similar quality and seems to have approximately the same triangle count for most examples, but we have not yet performed a head-to-head comparison. CAD surfaces such as swept surfaces and cylinder parts will take advantage of our approach where the parameter directions does not use the same tessellation levels.

Bézier patches fully outside the view frustum are efficiently detected and the tessellation level is set to zero in both algorithms. Small patches on the other hand, will result in a minimum of two triangles, even if they are much smaller than a pixel. For some applications it may therefore be better to treat more than one Bézier patch in each tessellation patch. Our error estimate can be used in this setting, since does not require a polynomial surface.

References

1. Filip, D., Magedson, R., Markot, R.: Surface algorithms using bounds on derivatives. Comput. Aided Geom. Des. 3(4), 295–311 (1987)
2. Cook, R.L., Carpenter, L., Catmull, E.: The reyes image rendering architecture. In: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, pp. 95–102. ACM, New York (1987)

3. Fisher, M., Fatahalian, K., Boulos, S., Akeley, K., Mark, W.R., Hanrahan, P.: Diagsplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. ACM Trans. Graph. 28(5), 150:1–150:10 (2009)
4. Guthe, M., Balázs, A., Klein, R.: Gpu-based trimming and tessellation of nurbs and t-spline surfaces. ACM Trans. Graph. 24(3), 1016–1023 (2005)
5. Hjelmervik, J., Hagen, T.: GPU-based screen space tessellation. In: Dæhlen, M., Mørken, K., Schumaker, L.L. (eds.) Mathematical Methods for Curves and Surfaces: Tromsø 2004, pp. 213–221. Nashboro Press (2005)
6. Lutterkort, D.C.: Envelopes of nonlinear geometry. PhD thesis, Purdue University, West Lafayette, IN, USA (2000); AAI3017831
7. Yeo, Y.I., Bin, L., Peters, J.: Efficient pixel-accurate rendering of curved surfaces. In: Garland, M., Wang, R., Spencer, S.N., Gopi, M., Yoon, S.E. (eds.) I3D, pp. 165–174. ACM (2012)
8. Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2. Addison-Wesley Longman Publishing Co., Inc., Boston (2005)
9. Dyken, C., Reimers, M., Seland, J.: Real-time GPU silhouette refinement using adaptively blended bézier patches. Computer Graphics Forum 27(1), 1–12 (2008)