Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

Non-Uniform Fractional Tessellation

Jacob Munkberg Jon Hasselgren Tomas Akenine-Möller†

Lund University, Sweden

Abstract

We present a technique that modifies the tessellator in current graphics hardware so that the result is a more uniformly distributed tessellation in screen space. For increased flexibility, vertex tessellation weights are introduced. Our results show that the tessellation quality is improved at a moderate cost.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture-Graphic Processors I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Curve, surface, solid, and object representations

1. Introduction

Recent graphics processors [Dog05, TOM*07] include a tessellation unit, allowing data amplification by tessellating *base triangles* to many smaller triangles in the graphics hardware. This helps lowering the bus traffic from the host computer to the graphics processor, by sending higher level surface representations instead of finely tessellated geometry.

Surface tessellation is a vast area of research, and we will limit the discussion here to work directly related to our approach. The REYES architecture [CCC87] splits the input primitives in eye-space iteratively until they have a size smaller than a certain threshold. Then, these smaller primitives are diced into pixel-sized bilinear patches called *micropolygons*. The tessellation rate is determined by the projected screen-space size of each primitive. This results in an approximately uniform tessellation in screen space. Notice that dicing is performed prior to displacement shading, so there is no guarantee for fully uniform screen-space tessellation, which is similar to the approach we will present.

On current graphics hardware, an input primitive (line, triangle or quad) is tessellated in parameter space and the vertex positions in the generated mesh are determined by a vertex/evaluation shader. This allows approximations of higher order surfaces, such as Beziér patches and subdivision surfaces [LS07]. It is hard to adapt the tessellation to the final projection on-screen *before* the evaluation shader, as the shader may move the vertex position arbitrarily. However, micro-triangles closer to the camera should generally be smaller than micro-triangles far away.

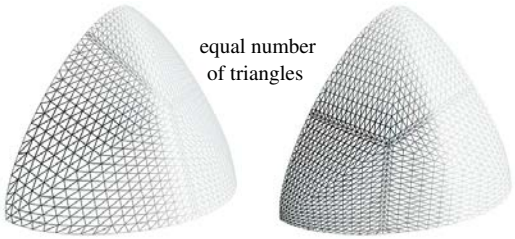


Figure 1: *Comparison between tessellation on a PN-displaced triangle [VPBM01]. Our algorithm places more vertices (non-uniformly) closer to the camera, which results in more uniform screen-space triangle areas.*

To allow for continuous level-of-detail (LOD) without visual "popping" and T-junctions, a tessellation scheme, hereafter called *regular fractional tessellation* [Mor01], can be used, where a floating point tessellation factor per edge is provided. An overview of this approach is provided in Section 2.

In this paper, we present a modification of regular fractional tessellation. By using perspective-correct interpolation [Bli91, HM91] and complementary *vertex weights*, we obtain an almost uniform tessellation in screen space. We warp the parametric coordinates of each tessellated mesh vertex *before* the evaluation shader so that the screen-space projection of the tessellation pattern has triangles with as uniform areas as possible. The only assumption is that the evaluation/vertex shader contains a perspective projec-

† {jacobljonltam}@cs.lth.se

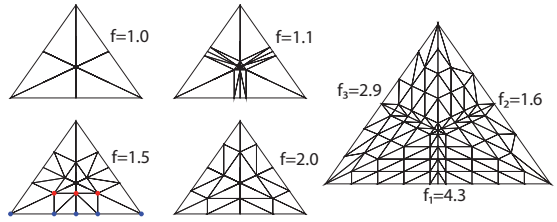


Figure 2: *Left: Four regular fractional tessellation examples are shown with a common tessellation factor (f) on all edges, from f=1.0 up to f=2.0. As can be seen in the lower left triangle, for each inner triangle, the number of vertices decreases with two per edge. Right: Each edge of a triangle can also have a unique tessellation factor*

.

tion transform. There are many published adaptive tessellation techniques that use information of the tessellated surface *after* surface evaluation [BAD*01,CK01,CK03], which achieve higher quality, but with a substantially higher computational cost. Given a graphics card with regular fractional tessellation, our algorithm can be implemented directly as a first step in a vertex/evaluation shader.

For a base triangle with an edge along the view vector, regular fractional tessellation adapts poorly. We adjust the scheme to better distribute the vertices over the triangle, while retaining many its strong advantages. Figure 1 shows an example of our technique.

2. Regular Fractional Tessellation

Regular fractional tessellation is a continuous tessellation scheme where floating point weights are assigned to each edge of a triangle. The description in this section is heavily influenced by the original presentation by Moreton [Mor01], but is included here for clarity. To allow for a continuous level of detail, new vertices emerge symmetrically from the center of each edge. Furthermore, vertices must move continuously with respect to the tessellation factors. The scheme consists of one inner, regular part, and a transitional part (the outermost edges). An example of the continuous introduction of new vertices is shown in Figure 2. Each outer edge is divided in two for symmetry, and each half-edge can be treated independently. Given an edge with tessellation factor f, first compute the integer part of f: $n = \lfloor f \rfloor$. Then step n times with a step size $1/f$ (assuming an half-edge length of one), and finally, connect the current vertex with the mid-point of the edge. This allows for efficient surface evaluation schemes, such as forward differencing, which need uniform step sizes. The other half-edge is tessellated symmetrically, resulting in two smaller distances close to the mid-point.

2.1. Edge tessellation factors

In the uniform case, the edges of a sub-triangle have two vertices fewer than the triangle edges one level further out (see Figure 2). In the case of equal tessellation weights f on all three edges, we obtain a regular inner triangle with tessellation factors $f-1$.

In the general setting, each edge has a unique tessellation factor. With different tessellation factors, the symmetric interior and the outermost edges can be connected by a stitching state-machine based on Bresenham's line drawing algorithm (see Moreton's paper [Mor01] for details). Figure 2 illustrates an example triangle with three different edge tessellation factors, f_i. The tessellation factor for the interior part can, for example, be chosen as $\max_i(f_i) - 1$.

The edge tessellation factors can be computed by, for example, projecting each triangle edge on the image plane and computing their screen-space lengths, giving larger weights to edges closer to the camera. This is reasonable, as one strives for having equal area of each generated triangle. For displacement-mapped surfaces, local characteristics of the displacement map, as heights and normal variations, can also be exploited to determine the tessellation rate [DH00].

2.2. Fractional Tessellation on Current GPUs

Recent graphics hardware from AMD/ATI supports regular fractional tessellation. In their implementation, the tessellation unit takes vertices and edge tessellation factors of a base triangle as inputs, and generates a set of new vertices. The tessellation unit computes the barycentric coordinates for every created vertex, and executes a vertex or *evaluation shader*. The task of this shader is to compute the position of a vertex as a function of its barycentric coordinates and the three vertices of the base triangle.

The edge tessellation factors can be computed either on the CPU, or by adding an additional pass on the GPU and using "render to vertex buffer" capabilities to execute a shader program that computes the factor for each edge.

3. Non-Uniform Fractional Tessellation

A disadvantage of the regular fractional tessellation algorithm is that vertices along an edge are distributed uniformly (except locally around the center, where new vertices are introduced). If an edge is parallel to the view direction, a uniform tessellation along this edge is far from optimal. Our goal is to create a tessellation pattern that preserves the qualities of regular fractional tessellation, such as continuous level of detail and introduction of new vertices at an existing vertex. In addition, we strive for uniform micro-triangle sizes in screen space before the evaluation shader is executed, similar to what is done in the REYES architecture [CCC87].

Given a base triangle, we first tessellate using the regular fractional tessellation algorithm as described in Section 2. We then modify the barycentric coordinate of each vertex in the generated tessellation so that its projection in screen

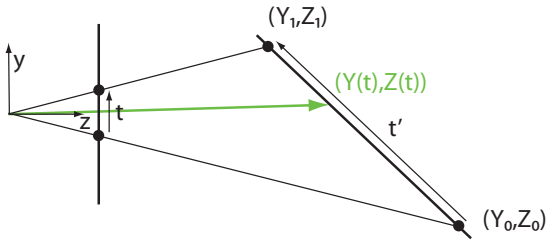


Figure 3: *Perspective-correct interpolation.*

space results in uniform micro-triangle sizes. This achieved by using reverse projection, as described in the following section.

3.1. Reverse Projection

We start with a simple example in two dimensions. Figure 3 shows a line $l = (1-t')(Y_0,Z_0)+t'(Y_1,Z_1)$ in perspective. Let t' denote a parameter along the line in camera space and t a parameter along the projection of the line in screen space. Using similar triangles and linear interpolation in t and t', we can derive a relationship between them as:

$$t' = \frac{t/Z_1}{t/Z_1 + (1-t)/Z_0}. \tag{1}$$

Now, assume we have a uniform distribution of points in t. Figure 4 shows the corresponding distributions in t' for various depth values Z_0 and Z_1. The bigger the depth difference, the more non-uniform distribution in t'. All the distributions t' from Figure 4 will project back to a uniform distribution in screen-space, by construction.

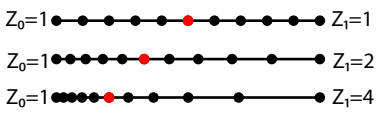


Figure 4: *Perspective remapping of a uniform edge for three different combinations of vertex depths.*

Next, this is generalized to two dimensions. Denote the barycentric coordinates of the triangle (u',v'), and the projected barycentric coordinates in screen space as: (u,v). Regular fractional tessellation will create a uniform pattern in the plane of the triangle, but when projected on-screen, this will no longer be uniform. However, assume we have a regular fractional tessellation in screen space, and reverse-project the pattern out on the triangle in camera space. If we know the vertex depths in camera space of our base triangle, we can generalize the derivation from the two-dimensional example above to form the standard perspective-correct barycentric coordinates [Bli91, HM91] for triangles:

$$\begin{aligned} u' &= \frac{u/Z_1}{(1-u-v)/Z_0 + u/Z_1 + v/Z_2}, \\ v' &= \frac{v/Z_2}{(1-u-v)/Z_0 + u/Z_1 + v/Z_2}. \end{aligned} \tag{2}$$

These are the barycentric coordinates in camera space that project to a uniform tessellation in screen space. This can also be seen an a function that adjust the barycentric coordinates of the triangle (u',v') before projection so that they create a uniform distribution of (u,v) in screen space, using three *vertex weights*, $\{Z_i\}$.

In the GPU-pipeline, the evaluation shader receives barycentric coordinates *before* projection as input, and by simply applying Equation 2 to these barycentric coordinates as a first step in the evaluation shader, the pattern will be roughly uniform in screen-space after projection. Note that we need the depth values (in camera space) for each vertex of the base triangle. One approach is to compute these vertex weights in a shader in a preceding pass, similar to how edge tessellation factors are handled in current hardware solutions (see Section 2.2). Another possibility is to compute the depth values in the evaluation shader (a dot product), just before we perform the reverse projection. This solution avoids sending data between different passes, but performs redundant work.

The same correction technique works for quad primitives by using generalized barycentric coordinates. For example, *mean value coordinates* works as generalized barycentric coordinates for quads. Please refer to Hormann and Tarini's work on quad rendering [HT04] for details.

Discussion Note that reverse projection gives a (roughly) constant triangle area tessellation in screen space only if the base triangle is not undergoing any transformations other than the projection. In practice, this is not true as subdivision surfaces and displacement mapping are the most common applications of tessellation. However, the resulting tessellation quality is more likely to be better if we start with a uniform tessellation in screen space, even when an arbitrary vertex shader follows.

3.2. Clipping

Our reverse projection is based on perspective-correct interpolation, which means that problems occur when part of a triangle is behind the camera (straddling triangles). The mathematics of the perspective-correct interpolation breaks down as the projected triangle "wraps around" infinity. In most settings, this problem is avoided, as triangles are clipped to the near-plane of the view-frustum. Our algorithm is executed prior to clipping, and must handle this case.

A further complication is that triangles with one or two vertices in front of the near plane, but outside the view frustum will get an unnecessary concentration of vertices outside the view frustum, as shown in the left part of Figure 5.

A proposed solution is to clip the base triangles against the view frustum (we use Cohen-Sutherland clipping [NS79]), and split the straddling triangles in smaller triangles entirely on either side of the clip volume. For triangles outside the frustum, we compute new weights so that the interpolation

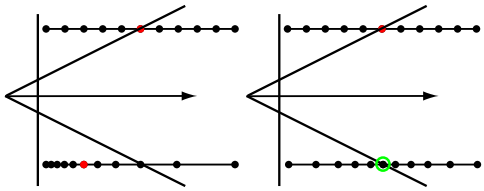


Figure 5: *Left: for triangles that straddle the view frustum, uniform tessellation can be better than our corrected version. Right: By clipping the base primitives to the frustum, we alleviate this situation.*

distributes triangles closer to the frustum edge. The right part of Figure 5 shows this. This approach simply updates the vertex weights for each base primitive in the clipping pass, and no detection is needed in the evaluation shader. Although the clipping is costly, it is only performed on the coarser base geometry in a preceding shader pass.

We want to stress that the evaluation (vertex) shader is not known, and that it may displace the tessellated vertices arbitrarily. For instance, it may move a vertex over the near clipping plane, thereby making it visible. Our mirrored projection is well motivated in that it distributes many vertices around the intersection with the view frustum. Under the assumption that the vertex displacement is local, it is more likely that a vertex close to a frustum border is moved in front of it, than a vertex further away.

4. Implementation

Our algorithm can be implemented in hardware, as well as in shader code. Current fractional tessellation hardware already feeds barycentric coordinates to the evaluation shader. We can essentially just insert code for our reverse projection algorithm in the beginning of the evaluation shader to compute new barycentric coordinates. These coordinates can then be fed to the remainder of the evaluation shader, which may differ depending on the application.

In our implementation, we perform regular fractional tessellation on the CPU. This could have been done by recent GPUs, but currently there are no public APIs for using the tessellation unit. Our reverse projection is implemented in a vertex shader, inspired by the evaluation shader approach by AMD [Dog05, TOM*07]. The inputs to the vertex shader are the positions of all three vertices of the base triangle, as well as the barycentric coordinates of the current tessellated vertex. Given this setup, our reverse projection code compiles to 11 vertex shader assembly instructions. By comparison, an extremely simple evaluation shader that interpolates a single position attribute and transforms it to clip space, compiles to 10 instructions. Thus, our overhead here is very small. Our frustum clipping is considerably more expensive, but must only be performed on the original base triangles in a preceding pass.

5. Results and Conclusions

Figure 6 shows regular fractional tessellation and the proposed technique for a displaced brick road. For our technique, the triangle density is more uniformly spread out in screen space, and the close-up detail is better preserved. This can especially be seen in the center of the images where the bricks have a smoother look with our tessellation, and in the far back where more triangles are gathered for regular fractional tessellation. Similar effects are shown in Figure 1, where a Bézier-triangle has been generated in an evaluation shader. These images use exactly the same number of micro-triangles. However, as can be seen, the micro-triangles are more uniform in terms of projected micro-triangle area with our technique, which was our goal.

One potential problem is vertex "swimming" during animation, as we warp the parametric space. However, this is true for *any* scheme using fractional tessellation with tessellation weights computed per frame. In practice, we found that our scheme shows about the same or less swimming artifacts compared to regular adaptive fractional tessellation. The accompanying video compares these artifacts during animation.

The warping technique presented here must not be limited to perspective-correction, but could be seen as a more general approach to achieve better control over surface tessellation. As future work, it would be interesting to test other warping functions, allowing each edge to have an independent warping function and investigating more elaborate LOD measures for the vertex weights. We hope that this paper will stimulate further research in the field.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research, Intel Corporation and an AMD Fellowship.

References

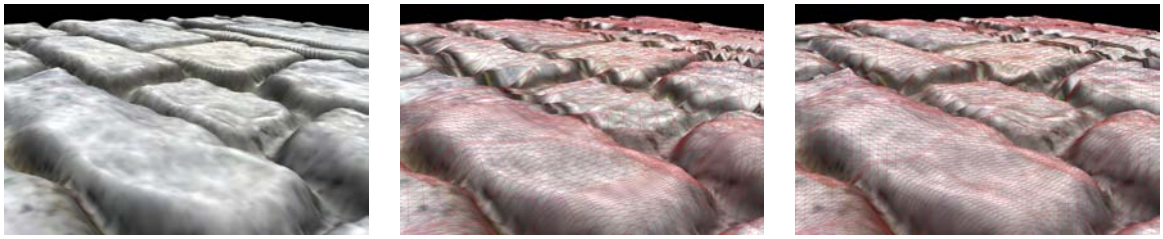
[BAD*01] BÓO M., AMOR M., DOGGETT M., HIRCHE J., STRASSER W.: Hardware support for adaptive subdivision surface rendering. In *Graphics Hardware* (2001), pp. 33–40.

[Bli91] BLINN J.: Hyperbolic Interpolation. *IEEE Computer Graphics and Applications, 11*, 1 (1991), 89–94.

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)* (1987), pp. 96–102.

[CK01] CHHUGANI J., KUMAR S.: View-dependent adaptive tessellation of spline surfaces. In *Symposium on Interactive 3D graphics* (2001), pp. 59–62.

[CK03] CHUNG K., KIM L.-S.: Adaptive Tessellation of PN Triangle with Modified Bresenham Algorithm. In *SOC Design Conference* (2003), pp. 102–113.



Reference Regular Our technique

Figure 6: *Brick road test scene. We use low tessellation to stress the algorithms.*

[DH00] DOGGETT M., HIRCHE J.: Adaptive View Dependent Tessellation of Displacement Maps. In *Graphics Hardware* (2000), pp. 59–66.

[Dog05] DOGGETT M.: Xenos: XBOX 360 GPU. Eurographics presentation, September 2005.

[HM91] HECKBERT P. S., MORETON H.: Interpolation for Polygon Texture Mapping and Shading. In *State of the Art in Computer Graphics: Visualization and Modeling* (1991), pp. 101–111.

[HT04] HORMANN K., TARINI M.: A Quadrilateral Rendering Primitive. In *Graphics Hardware* (2004), pp. 7–14.

[LS07] LOOP C., SCHAEFER S.: *Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches*. Tech. rep., MSR-TR-2007-44, Microsoft Research, 2007.

[Mor01] MORETON H.: Watertight Tessellation using Forward Differencing. In *Graphics Hardware* (2001), pp. 25–32.

[NS79] NEWMAN W., SPROULL R.: *Principles of Interactive Computer Graphics*, 2nd ed. New York: McGraw-Hill, 1979.

[TOM*07] TATARCHUK N., OAT C., MITCHELL J. L., GREEN C., ANDERSSON J., MITTRING M., DRONE S., GALOPPO N.: Advanced Real-Time Rendering in 3D Graphics and Games. SIGGRAPH course, 2007.

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J. L.: Curved PN triangles. In *Symposium on Interactive 3D graphics* (2001), pp. 159–166.