GPU-Based Rendering of PN Triangle Meshes with Adaptive Tessellation

Michael Schwarz, Marco Staginski, Marc Stamminger

Computer Graphics Group, University of Erlangen-Nuremberg
Am Weichselgarten 9, 91058 Erlangen, Germany
Email: {schwarz, stamminger}@cs.fau.de

Abstract

Numerous approaches to describe curved surfaces have been proposed in computer graphics. Basically all of them require the generation of an appropriately refined triangle mesh for rasterization-based rendering characteristic of current-generation GPUs.

In this paper we deal with PN triangles as surface primitives and render them with GPU-resident refinement patterns obtained by successive 1-to-4 splits of a generic triangle. The actual pattern and hence the subdivision level is chosen on a per-primitive basis by means of a novel screen-space error metric, yielding an adaptive tessellation according to the viewing situation with vertex position computations being performed solely on the GPU. To avoid visible cracks in the resulting mesh, stitching is performed by rendering appropriate connection patterns. These also help to close holes inherent to the PN triangulation of coarse base meshes with corner-like features.

1 Introduction

Often a key to a realistic and visually pleasing appearance of rendered objects are curved surfaces. One possibility to model them in a resolution-independent manner are PN triangles [16]. A mesh composed of such primitives is uniquely described by a coarse triangular base mesh with per-vertex normal information. While ray tracing techniques can directly operate on the resulting parametric representation [13, 15], in a rasterization-based approach each PN triangle is first subtriangulated such that its surface is approximated reasonable well and then the resulting triangle mesh is rendered.

Selecting the appropriate tessellation level is crucial since a too coarse refinement causes the surfaces and especially the model's silhouettes to no longer appear curved while an overly fine subdivision results in additional dispensable vertices and triangles to be processed by the GPU, unnecessarily slowing down rendering. More precisely, the subtriangulation level has to be chosen on a per-patch or per-edge basis and not be fixed for all PN triangles composing a model since the PN triangles' screen-space sizes often vary considerably and they are usually curved to different degrees. For instance, flat PN triangles need not to be subtriangulated at all if per-fragment shading is performed.

In this paper we propose a method to obtain such an adaptive tessellation for real-time rendering of PN triangle meshes. It is driven by a novel screen-space error metric that takes the actual degree to which a PN triangle is curved into account. For each PN triangle an appropriately selected refinement pattern is rendered with merely the triangle's control points being provided as parameters and the actual calculation of vertex positions being carried out on the GPU. These patterns are shared among all PN triangles and stored in fast graphics memory. They are created by successively performing 1-to-4 splits of a generic triangle.

Because the subdivision level is chosen independently for each PN triangle, cracks between the tessellation of adjacent PN triangles may occur. To this end, a stitching process is performed which involves rendering connection patterns similar to the refinement patterns. As a side-effect, holes due to corner-like features in the model's coarse base mesh get closed, too.

2 Related work

A PN triangle [16] is a cubic triangular Bézier patch

$$\mathbf{b}(u,v) = \sum_{i+j+k=3} \frac{3!}{i!\,j!\,k!}\, w^i u^j v^k\, \mathbf{b}_{ijk} \qquad (1)$$

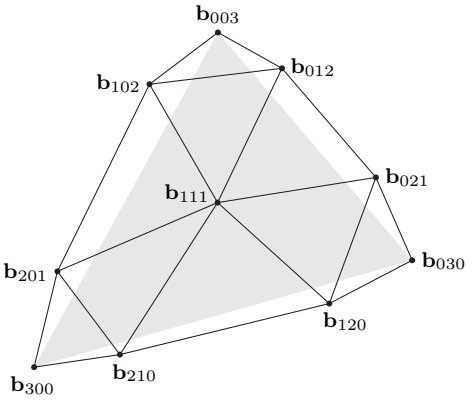


Figure 1: Control points $\mathbf{b}_{ijk}$ of a PN triangle's cubic Bézier patch.

with $w = 1-u-v$, whose control points are merely depending on the vertex positions and normals of a defining base triangle. Since each PN triangle in a mesh is thus independent of neighboring triangles, they can be treated in isolation which makes this primitive amenable to hardware tessellation.

Given a base triangle with vertices at positions $\mathbf{P}_m$ and accompanying normals $\mathbf{N}_m$, the corresponding PN triangle's vertex control points (cf. Fig. 1) are chosen to coincide with the base triangle's vertices, i.e.

$$\mathbf{b}_{300} = \mathbf{P}_1, \quad \mathbf{b}_{030} = \mathbf{P}_2, \quad \mathbf{b}_{003} = \mathbf{P}_3.$$

The tangent control points for the edge $\mathbf{P}_1\mathbf{P}_2$ are given by

$$\begin{aligned}\mathbf{b}_{210} &= \tfrac{1}{3}\big(2\mathbf{P}_1 + \mathbf{P}_2 - w_{12}\mathbf{N}_1\big)\\ \mathbf{b}_{120} &= \tfrac{1}{3}\big(2\mathbf{P}_2 + \mathbf{P}_1 - w_{21}\mathbf{N}_2\big)\end{aligned}$$

where $w_{ij} = \langle \mathbf{P}_j - \mathbf{P}_i, \mathbf{N}_i \rangle$. The ones for the remaining edges are computed analogously. Finally, the center control point is derived via

$$\begin{aligned}\mathbf{b}_{111} = \tfrac{1}{4}\big(&\mathbf{b}_{210} + \mathbf{b}_{120} + \mathbf{b}_{021} + \mathbf{b}_{012}\\ &+ \mathbf{b}_{102} + \mathbf{b}_{201}\big) - \tfrac{1}{6}\big(\mathbf{b}_{300} + \mathbf{b}_{030} + \mathbf{b}_{003}\big).\end{aligned}$$

The normal field of the PN triangle is given by a separate quadratic Bézier patch whose control points $\mathbf{n}_{ijk}$ are a function of the input positions $\mathbf{P}_m$ and normals $\mathbf{N}_m$, too.

PN triangles were devised to easily enable control point computation and tessellation by consumer graphics hardware so that just the base triangles' vertex positions and normals need to be specified by the application. While some ATI chips provide such a hardware support, they require the user to choose a fixed and uniform subtriangulation level for all PN triangles.

More recent work by Chung and Kim [6] suggests a hardware implementation which adaptively tessellates PN triangles. To this end, a subdivision level for each edge is determined by the edge's screen-space length and a corresponding triangular mesh is generated on-the-fly. Note that this approach yields a non-uniform tessellation pattern in case the subdivision levels for a triangle's edges differ [12]. Furthermore, because subdivision is only guided by the edge length, each PN triangle can still be treated in isolation by the proposed hardware implementation without causing cracks in the model.

A similar goal as in the adaptive tessellation of curved surfaces is pursued in the view-dependent refinement and coarsening of triangular meshes [9, 10], where the selection of an appropriate level-of-detail is governed by some error metric [11]. While most of the early approaches operated on single edges or triangles, more recent ones changed the granularity to patches of up to several thousand triangles [2, 7, 14] to account for the vastly increased processing power of current-generation GPUs and avoid rendering the CPU-based LOD adaptation a main bottleneck.

Taking the recent developments of graphics hardware into account, Boubekeur and Schlick [3] proposed a mesh refinement technique that exploits the GPU's programmability, renders primitives in larger batches and keeps the amount of geometry data transferred from CPU to GPU low. More precisely, refinement patterns are generated by subdividing a generic triangle to various levels with each vertex being assigned its barycentric coordinates as position. When rendering a model, for each coarse-level triangle a refinement pattern is drawn and additional parameters are provided depending on the kind of refinement like procedural displacement mapping or PN tessellation. These parameters are then used in a vertex shader to translate the pattern's vertices to the correct positions according to the employed refinement method. Since there is no need to create and store refined triangulations of a whole model but merely an appropriate refinement pattern shared by the model's triangles must

be held as vertex array in the graphics card's RAM, the memory footprint is rather low.

While this technique originally only allows for uniform refinement of triangular meshes, several more complex GPU-based adaptive tessellation schemes were suggested for other surface description methods like Catmull-Clark subdivision surfaces [1, 5] and trimmed NURBS and T-spline surfaces (approximated by rational bi-cubic Bézier patches) [8].

3 Adaptive tessellation

In our approach we adopt rendering refinement patterns for the subtriangulation of PN triangles, i.e. a pattern is rendered for each PN triangle with the control points $\mathbf{b}_{ijk}$ and $\mathbf{n}_{ijk}$ being provided as parameters. While the vertex shader performs the translation of the pattern's vertices onto the PN triangle's surface by evaluating Eq. (1), the fragment shader evaluates the normal field for shading.

However, in contrast to Boubekeur and Schlick [3], we don't use the same pattern for all triangles of a model but select the pattern on a per-triangle basis such that both the refinement level is as low as possible and a given screen-space error bound is satisfied, resulting in an adaptive tessellation of the model's PN triangle mesh. Moreover, we restrict ourselves to refinement patterns created by successive 1-to-4 splits of a triangle (cf. Fig. 2), i.e. to subdivision levels of $2^\ell - 1$. When switching from one pattern to the next one, this choice enforces that either the mesh gets refined by adding new vertices or it is coarsened by removing some vertices while keeping the remaining vertices' positions unchanged. Hence the borders of a PN triangle's tessellation are not suffering from "swimming" artifacts.

The hierarchical structure imposed by the 1-to-4 splits can also be exploited to keep the memory requirements low. Since the vertices of patterns for coarser refinement levels are real subsets of the ones for the finest level, it suffices to store the vertices of the most detailed pattern in a vertex array and provide index arrays for all refinement patterns. Also note that the restriction to uniform subdivision via 1-to-4 splits keeps the number of different patterns comparatively low which would be especially hard in case different subdivision levels were supported for each edge.

To select the coarsest refinement pattern for each PN triangle which respects a given screen-space error bound, we employ a novel error metric that takes a PN triangle's actual shape and size into account. In particular, it seems mandatory to include the degree to which a PN triangle is curved and hence its surface deviates from the corresponding base triangle in such a metric to avoid overly fine subtriangulation. For instance, in the extreme case of a flat PN triangle where all per-vertex normals coincide with the triangle's face normal, no subdivision is needed at all. Note that because the normal field is evaluated on a per-fragment basis, the smoothness of the surface's shading and the appearance of highlights don't depend on the actual degree of geometric subdivision.

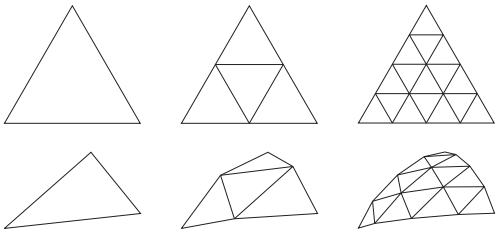


Figure 2: Refinement patterns for subdivision levels 0, 1 and 3 (top, from left to right) and their usage for tessellating a PN triangle (bottom).

A crucial quantity in the derivation of our error metric is the distance of the tangent control points from their corresponding edges. While it can easily be shown that this distance is bound by $L/6$ [16], where L denotes the base triangle's longest edge, we determine the actual distances d_{ijk} to get a tighter error bound for flatter PN triangles.

Suppose a PN triangle is tessellated with a subdivision level of $2^\ell - 1$. Then for those vertices that would be newly inserted along an edge by the next 1-to-4 refinement step, it can be shown that their maximum distance from the polyline currently approximating the Bézier boundary curve in any direction perpendicular to the corresponding edge is bound by

$$d_{\max}(d_1, d_2, \ell) \leq 4^{-\ell} \cdot \max_{t \in [0,1]} \left| \left(\tfrac{3}{2} - \tfrac{9}{4}\, t\right) d_1 + \left(\tfrac{3}{4} - \tfrac{9}{4}\, t\right) d_2 \right|$$

where d_1 and d_2 denote the quantities d_{ijk} of the involved tangent control points. While this metric

only takes certain vertices into account, it provides a reasonable estimate for the maximum deviation of the Bézier boundary curve from its polyline approximation.

Hence, after ℓ 1-to-4 splits of the base triangle, the maximum deviation of the resulting tessellation from the actual PN triangle can be estimated by

$$\begin{aligned} d_{\max}(\ell) &= \max\{d_{\max}(d_{210}, d_{120}, \ell), \\ &\quad d_{\max}(d_{021}, d_{012}, \ell), d_{\max}(d_{102}, d_{201}, \ell)\} \\ &\leq 4^{-\ell} \cdot \tfrac{9}{4} \max\{d_{210}, d_{120}, d_{021}, \\ &\qquad d_{012}, d_{102}, d_{201}\} := 4^{-\ell} \cdot \tfrac{9}{4}\, d'_{\max} \end{aligned}$$

as far as the PN triangle's boundary is concerned. However, we note without further proof that this approximation also accounts for the maximum deviation of the PN triangle's surface with respect to the flat base triangle. Finally, since

$$\sum_{i+j+k=3} \frac{3!}{i!\,j!\,k!}\, w^i u^j v^k\, d_{ijk} \leq d'_{\max}$$

holds, where $d_{300} = d_{030} = d_{003} = 0$ and $d_{111} \leq \frac{3}{2} d'_{\max}$, the overall deviation is bound by $d'_{\max}$.

For each PN triangle, $d'_{\max}$ is calculated in a preprocess. During runtime, it is projected into screen space, yielding the error bound $e_{\max}$. This is then used to determine the appropriate refinement pattern for rendering the PN triangle. By comparing $e_{\max}$ to the user-specified screen-space error bound ε, we first check whether subtriangulation is required at all. If this is the case, ℓ is determined such that $4^{-\ell}\, e_{\max} \leq \varepsilon$ holds.

Note that with our metric a single 1-to-4 refinement causes the screen-space error to be reduced by a factor of four while employing the edge length for guiding the subdivision level would only indicate a halving of the error. Consequently, for larger values of $e_{\max}/\varepsilon$ our method requires the processing of far fewer vertices and triangles without degrading visual quality compared to approaches based on edge length as coarser refinement patterns can be rendered.

4 Stitching

Usually more than one distinct refinement pattern is used for rendering the PN triangles of a model because differently fine subdivision levels are required to satisfy a given screen-space error bound. Hence there are some adjacent PN triangles which are tessellated to different degrees and whose common boundary curve is thus approximated by different polylines. As a consequence, cracks are introduced in the resulting refined triangular mesh of a model.

Referring to Fig. 3, basically two situations can be distinguished at such places of discontinuity in subdivision. Either tiny holes appear due to the cracks (a) or the refinement patterns rendered for two adjacent PN triangles overlap slightly (b). While the first case can lead to visible artifacts as single pixels along the common boundary curve in question might be omitted and reveal the background, the second setting is usually less problematic. In particular, if the screen-space error bound is chosen small enough, no visible discontinuities appear at all.

To avoid visible holes due to cracks, we stitch the mesh resulting from approximating the PN triangles with refinement patterns. This is achieved by rendering for each pair of adjacent PN triangles with different refinement levels a general triangle strip that connects the two polylines used for the common boundary curve (the gray areas in Fig. 3 (a) and (b) depict such strips). Similar to the refinement patterns, connection patterns for all possible stitching strips are generated and kept in fast graphics memory. These are then used for rendering the strips, keeping the amount of data to be transferred from CPU to GPU to a minimum as merely the control points for the common boundary curve have to be provided as parameters.

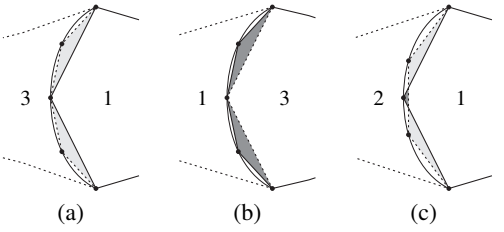


Figure 3: Adjacent triangles of different subdivision levels.

In situations where the refinement patterns for adjacent PN triangles overlap, stitching results in mesh fold-overs which could lead to visual artifacts if a large screen-space error bound is chosen. To alleviate such disturbances, we check for these cases

and skip rendering those triangles of the connecting strips which would cause a fold-over. For reasons of simplicity, consistent numerical precision and speed we don't perform the necessary tests on the CPU but render the connection patterns with back-face culling enabled, which works fine in practice for closed models.

Note that in case the subdivision levels of two adjacent PN triangles differ by more than one 1-to-4 split and the shared boundary curve has an S-like shape, minor imprecisions can occur. However, thanks to our restriction to refinement patterns resulting from successive 1-to-4 splits these situations are very rare. In contrast, if subdivisions level other than $2^\ell - 1$ were allowed, such settings would be encountered quite often (cf. Fig. 3 (c)). Moreover, a much larger number of connecting strips would have to be generated, significantly increasing the memory load.

We further note that in a certain sense, our approach of rendering refinement and connection patterns comes down to decomposing non-uniform tessellation patterns into a uniformly tessellated central part of highest subdivision and into transition regions to boundary curves of lower subdivisions.

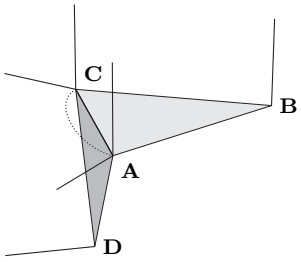


Figure 4: Adjacent triangles with different per-vertex normals at the common edge $\mathbf{AC}$.

In case the coarse triangle mesh has not explicitly been modeled for being rendered with PN triangles, it might happen that triangles sharing a single vertex provide different normals for the common vertex to model features like corners. Hence adjacent base triangles can lead to PN triangles that don't share a common boundary curve. For instance, referring to Fig. 4, the common edge $\mathbf{AC}$ becomes a boundary curve of PN triangle $\mathbf{ABC}$ whereas the counterpart at PN triangle $\mathbf{DAC}$ is the dotted curve, i.e. a hole appears in the PN triangle mesh not present in the coarse base mesh.

However, as a side-effect of performing stitching by rendering connecting strips, such holes inherent to the PN triangulation get closed (cf. Fig. 5). Therefore, if neighboring PN triangles have different per-vertex normals specified, we render connection patterns even in case the subdivision levels are equal. For the pattern's normal field we adopt the heuristic of using the normals along the boundary curve of that affected PN triangle whose base triangle's face normal deviates less from the average face normal of the connection strip for subdivision level 1.

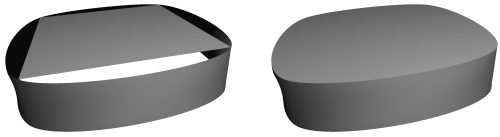


Figure 5: Holes in models (left) caused by specifying different normals for a vertex shared by multiple faces can be closed with connecting strips (right).

We reckon that an enriched PN triangle description like ST meshes [4] might be a better option for modeling special features like creases. Yet, they require the user to specify additional scalar values for the involved vertices and support only two different normals per vertex. On the other hand, without careful modeling, providing different normals for a vertex shared by two base triangles cannot only lead to holes in the PN triangulation of the base mesh but to more severe artifacts like neighboring PN triangles which intersect.

5 Geomorphing

Since our adaptive tessellation method is based on successive 1-to-4 splits, geomorphing support can be incorporated in a natural way. But adopting such a smooth blending scheme for switching between different refinement patterns has some shortcomings. First, geomorphing incurs higher rendering costs as the computation of the vertex positions becomes more complex. Even worse, connection patterns have to be rendered not only for adjacent PN triangles whose subdivision levels differs but also for those employing the same refinement pattern but different geomorphing transition values. This could be alleviated to some degree, however, by determining the transition parameter per edge instead

of per triangle and allowing for non-uniform transition values for the vertices of a rendered refinement pattern [2, 14].

Second, geomorphing is only useful if the screen-space error bound is sufficiently large (e.g. $\varepsilon > 0.5$ pixels) as otherwise no popping artifacts can be noticed anyway. Since an increased number of connection patterns need to be rendered if geomorphing is performed it is not even ensured that the rendering time is reduced if the screen-space error bound is increased and geomorphing is enabled. Note that in supra-threshold settings where the error bound allows for deviations of several pixels, overlapping refinement patterns might lead to visible artifacts. Moreover, the connection patterns might become clearly visible as their normal fields stay constant in the surface direction perpendicular to the approximated boundary curves.

Summing up, since a small screen-space error bound is essential for avoiding artifacts, geomorphing is not really necessary but incurs additional overhead. However, note that it is well-suited to avoid popping in case of uniform tesselation in supra-threshold settings.

6 Results

Some example scenes rendered with our method are depicted in Fig. 6. Note that the employed models were not explicitly modeled with PN triangulation in mind and that geomorphing was disabled.

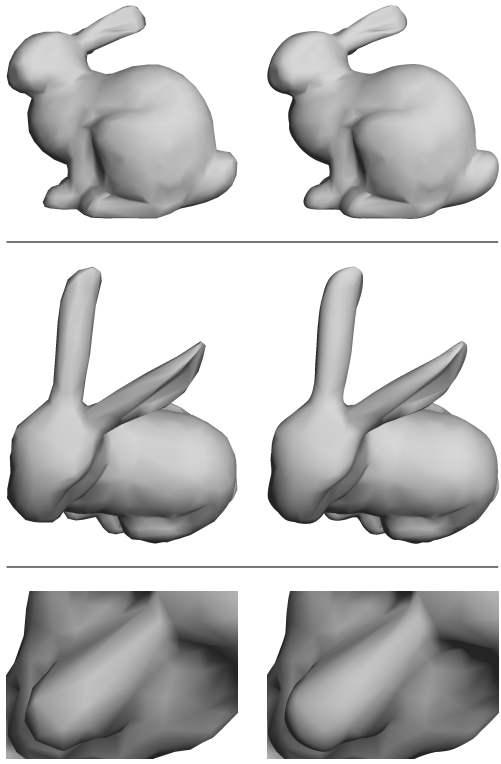


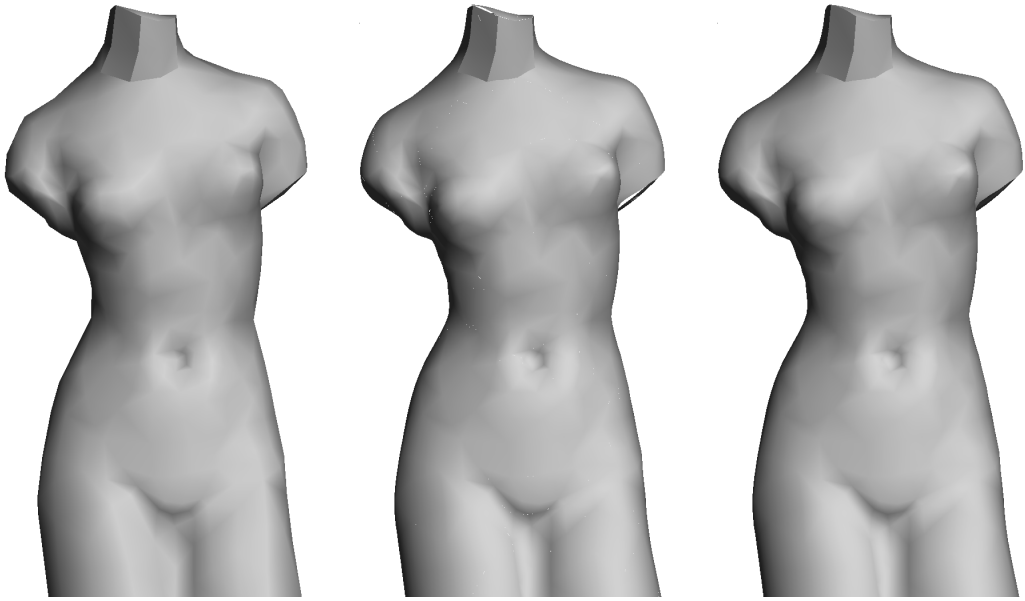
Figure 6: Side (top row) and top view (middle) of bunny model and close-up of Caesar mask model (bottom) rendered both as triangle mesh (left) and as adaptively tessellated PN triangle mesh (right).

	Triangles			Frame rate		
Scene	b	a	u	b	a	u
Bunny (side)	2000	46537	129152	341	218	140
Bunny (top)	2000	45026	129152	397	244	145
Caesar mask	2000	88054	606720	173	110	44
Venus statue	1418	26894	91744	462	319	188

Table 1: Number of drawn triangles and rendered frames per second in the example scenes (viewport 1600×1144) in case of no subtriangulation of the base mesh (b), adaptive (a) and uniform (u) tessellation with a screen-space error bound of 0.5 pixels.

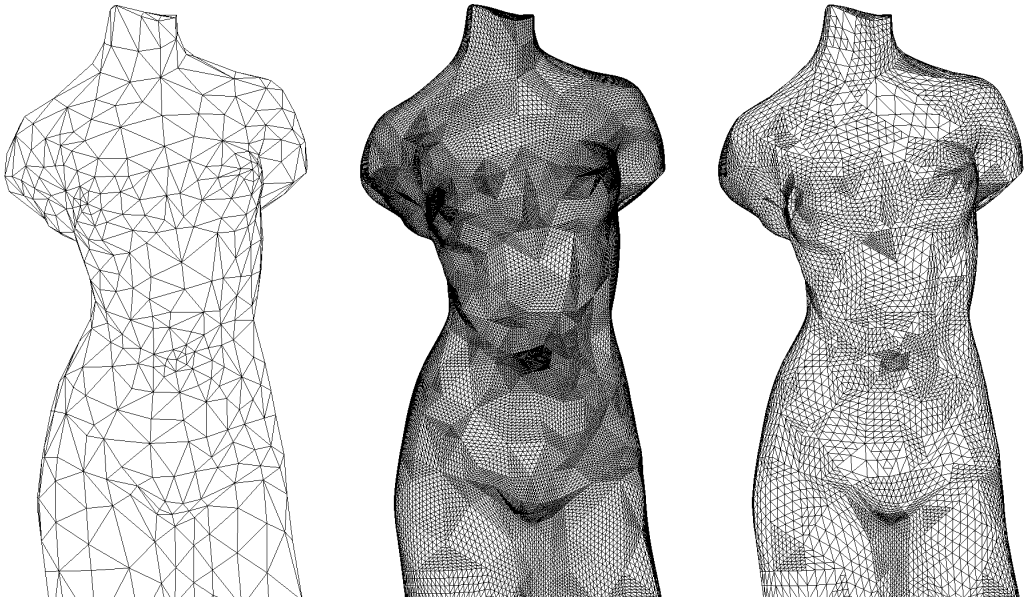
By means of the model of a Venus statue, Fig. 7 demonstrates the effectiveness of using connection patterns for closing holes caused by either cracks or by specifying multiple normals at certain vertices (e.g. at the left arm stump). As the triangle counts show, only an overhead of roughly 15% is introduced by our stitching approach based on connection patterns. Finally, in Fig. 8 the advantage of using adaptive tessellation becomes apparent. In case of uniform tessellation, the finest subdivision level required by any of the PN triangles to respect a given screen-space error bound determines the refinement pattern rendered for all PN triangles.

Quantitative results for these scenes obtained with an NVIDIA GeForce 7800 GT are listed in Table 1. All reported frame rates, including the ones for the case of no subtriangulation, comprise the evaluation of the PN triangles' normal fields on a per-fragment basis as well as Phong shading with a single light source. Adaptive tessellation results in 2.7 to 6.9 times fewer triangles being rendered in our example scenes and an increase in frame rate by a factor of 1.6 to 2.5 compared to uniform tessellation.



1418 triangles 23357 triangles 26894 triangles

Figure 7: Venus statue's base mesh (left), adaptive tessellation without (center) and with stitching (right).



1418 triangles 91744 triangles 26894 triangles

Figure 8: Venus statue's coarse base mesh (left), corresponding uniform (center) and adaptive tessellation (right) respecting the same screen-space error bound.

7 Conclusion

We presented an approach to render meshes composed of PN triangles with current GPUs. An adaptive tessellation guided by a novel screen-space error metric is performed by rendering appropriate refinement patterns for the PN triangles. Furthermore, connection patterns are employed to avoid visible cracks and to close holes inherent to a model's PN triangulation.

Acknowledgements

The Julius Caesar mask model is provided courtesy of INRIA by the AIM@SHAPE Shape Repository. The bunny model is courtesy of the Stanford 3D Scanning Repository. Both data sets were simplified with Michael Garland's QSlim.

References

[1] Jeff Bolz and Peter Schröder. Evaluation of subdivision surfaces on programmable graphics hardware, 2003. `http://www.multires.caltech.edu/pubs/GPUSubD.pdf`.

[2] Louis Borgeat, Guy Godin, François Blais, Philippe Massicotte, and Christian Lahanier. GoLD: Interactive display of huge colored and textured models. *ACM Transactions on Graphics*, 24(3):869–877, July 2005.

[3] Tamy Boubekeur and Christophe Schlick. Generic mesh refinement on GPU. In *Proceedings of Graphics Hardware 2005*, pages 99–104, July 2005.

[4] Tamy Boubekeur, Patrick Reuter, and Christophe Schlick. Scalar tagged PN triangles. In *Eurographics 2005 Short Presentations*, pages 17–20, 2005.

[5] Michael Bunnell. Adaptive tessellation of subdivision surfaces with displacement mapping. In Matt Pharr, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 7, pages 109–122. Addison Wesley Professional, March 2005.

[6] Kyusik Chung and Lee-Sup Kim. Adaptive tessellation of PN triangle with modified Bresenham algorithm. In *Proceedings of SOC Design Conference 2003*, pages 448–452, November 2003.

[7] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Batched multi triangulation. In *Proceedings of IEEE Visualization 2005*, pages 207–214, October 2005.

[8] Michael Guthe, Ákos Balázs, and Reinhard Klein. GPU-based trimming and tessellation of NURBS and T-spline surfaces. *ACM Transactions on Graphics*, 24(3):1016–1023, July 2005.

[9] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of ACM SIGGRAPH 97*, pages 189–198, August 1997.

[10] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of ACM SIGGRAPH 97*, pages 199–208, August 1997.

[11] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2002.

[12] Henry Moreton. Watertight tessellation using forward differencing. In *Proceedings of Workshop on Graphics Hardware 1999*, pages 25–32, August 1999.

[13] S. H. Martin Roth, Patrick Diezi, and Markus H. Gross. Ray tracing triangular Bézier patches. *Computer Graphics Forum*, 20(3):422–432, September 2001.

[14] Petro V. Sander and Jason L. Mitchell. Progressive buffers: View-dependent geometry and texture LOD rendering. In *Proceedings of Eurographics Symposium on Geometry Processing 2005*, pages 129–138, July 2005.

[15] Wolfgang Stürzlinger. Ray tracing triangular trimmed free form surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):202–214, July–September 1998.

[16] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved PN triangles. In *Proceedings of ACM Symposium on Interactive 3D Graphics 2001*, pages 159–166, April 2001.