Real-Time GPU Silhouette Refinement using adaptively blended Bézier Patches

Christopher Dyken[1,2] and Martin Reimers[1] and Johan Seland[1]
[1]Centre of Mathematics for Applications, University of Oslo, Norway
[2] Department of Informatics, University of Oslo, Norway

EARLY DRAFT
Final version to appear in Computer Graphics Forum

Draft

Abstract

We present an algorithm for detecting and extracting the silhouette edges of a triangle mesh in real time using GPUs (Graphical Processing Units). We also propose a tessellation strategy for visualizing the mesh with smooth silhouettes through a continuous blend between Bézier patches with varying level of detail. Furthermore, we show how our techniques can be integrated with displacement and normal mapping. We give details on our GPU implementation and provide a performance analysis with respect to mesh size.

1 Introduction

Coarse triangular meshes are used extensively in real-time rendering applications such as games and virtual reality systems. Recent advances in graphics hardware have made it possible to use techniques such as normal mapping and per pixel lighting to increase the visual realism of such meshes. These techniques work well in many cases, adding a high level of detail to the final rendered scene. However, they can not hide the piecewise linear silhouette of a coarse triangular mesh. We propose an effective GPU implementation of a technique similar to the one proposed by two of the authors in [7], to adaptively refine triangular meshes along the silhouette, in order to improve its visual appearance. Since our technique dynamically refines geometry at the vertex level, it integrates well with pixel based techniques such as those mentioned above.

We start by reviewing previous and related work in the following section, before we introduce our notation and recall the *silhouetteness* classification method that was introduced in [7]. In Section 4 we discuss the construction of a cubic Bézier patch for each triangle in the mesh, based on the mesh geometry and shading normals. These patches are in the subsequent section tessellated adaptively using the silhouetteness to determine the local level of detail. The result is a "watertight" mesh with good geometric quality along the silhouettes, which can be rendered efficiently. We continue by discussing details of our GPU implementation in Section 6, and show how to integrate our approach with normal and displacement mapping. Thereafter, in Section 7, we compare the performance of our GPU implementation with several CPU based methods, before we conclude.

2 Previous and related work

Silhouette extraction. Silhouette extraction has been studied extensively, both in the framework for rendering soft shadows and for use in non-photorealistic-rendering. Isenberg et.al. [12] provides an excellent overview of the trade-offs involved in choosing among the various CPU-based silhouette extraction techniques. Hartner et.al. [10] benchmark and compare various algorithms in terms of runtime performance and code complexity. For comparison, we present runtime performance for our method within this framework in Section 7. Card and Mitchell [5] propose a single pass GPU assisted algorithm for render-

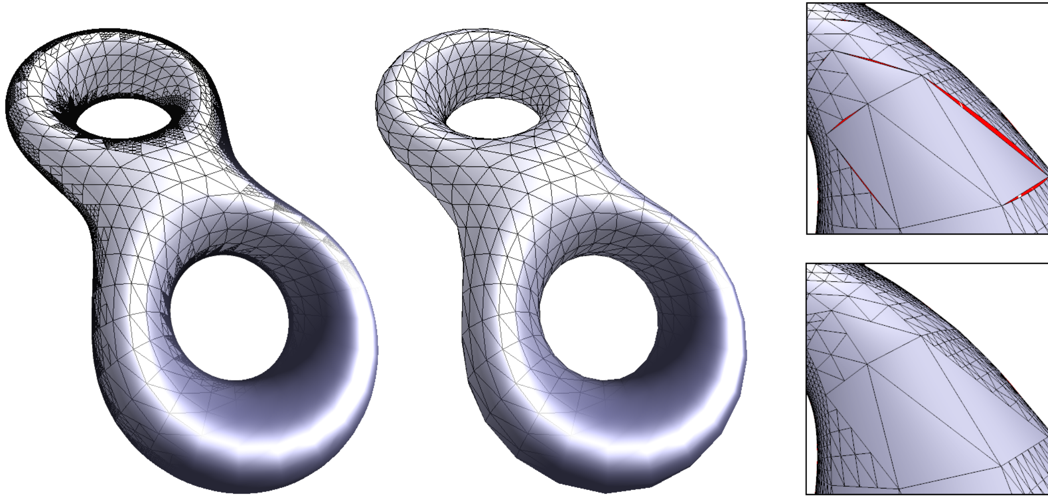


Figure 1: A dynamic refinement (left) of a coarse geometry (center). Cracking between patches of different refinement levels (top right) is eliminated using the technique described in Section 5 (bottom right).

ing silhouette edges, by degenerating all non silhouette edges in a vertex shader.

Curved geometry. Curved point-normal triangle patches (PN-triangles), introduced by Vlachos et.al. [22], do not need triangle connectivity between patches, and are therefore well suited for tessellation in hardware. An extension allowing for finer control of the resulting patches was presented by Boubekeur et.al. [2] and dubbed scalar tagged PN-triangles. A similar approach is taken by van Overveld and Wyvill [21], where subdivision was used instead of Bézier patches. Alliez et.al. describe a local refinement technique for subdivision surfaces [1].

Adaptivity and multi resolution meshes. Multi resolution methods for adaptive rendering have a long history, a survey is given by Luebke et.al. [14]. Some examples are progressive meshes, where refinement is done by repeated triangle splitting and deletion by Hoppe [11], or triangle subdivision as demonstrated by Pulli and Segal [16] and Kobbelt [13].

GPU techniques. Global subdivision using a GPU kernel is described by Shiue et.al. [19] and an adaptive subdivision technique using GPUs is given by Bunnel [4]. A GPU friendly technique for global mesh refinement on GPUs was presented by Boubekeur and Schlick [3], using pre-tessellated triangle strips stored on the GPU. Our rendering method is similar, but we extend their method by adding adaptivity to the rendered mesh.

A recent trend is to utilize the performance of GPUs for non-rendering computations, often called GPGPU (General-Purpose Computing on GPUs). We employ such techniques extensively in our algorithm, but forward the description of GPGPU programming to the introduction by Harris [9]. An overview of various applications in which GPGPU techniques have successfully been used is presented in Owens et.al. [15]. For information about OpenGL and the OpenGL Shading Language see the reference material by Shreiner et.al. [20] and Rost [17].

3 Silhouettes of triangle meshes

We consider a closed triangle mesh Ω with consistently oriented triangles $T_1, \ldots, T_N$ and vertices $\mathbf{v}_1, \ldots, \mathbf{v}_n$ in $\mathbb{R}^3$. The extension to meshes with boundaries is straightforward and is omitted for brevity. An edge of Ω is defined as $e_{ij} = [\mathbf{v}_i, \mathbf{v}_j]$ where $[\cdot]$ denotes the con-

Draft

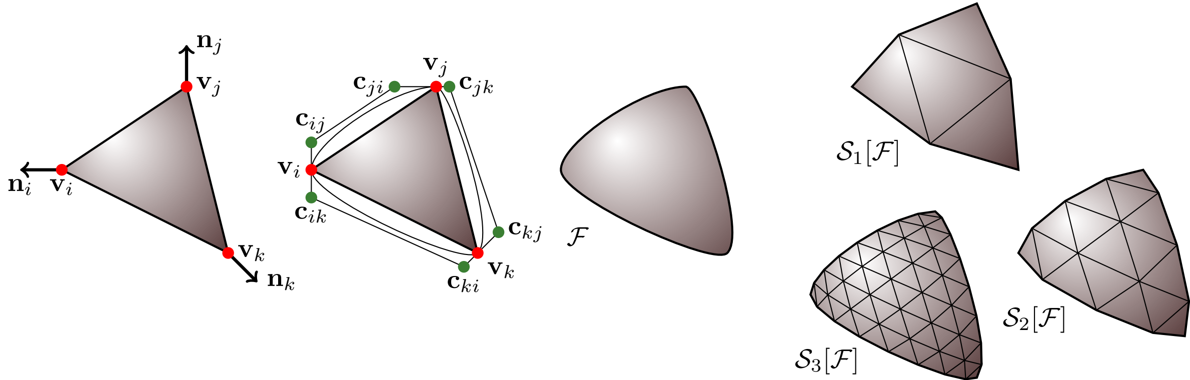


Figure 2: From left to right: A triangle $[\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k]$ and the associated shading normals $\mathbf{n}_i$, $\mathbf{n}_j$, and $\mathbf{n}_k$ is used to define three cubic Bézier curves and a corresponding cubic triangular Bézier patch $\mathcal{F}$. The sampling operator $\mathcal{S}_i$ yields tessellations of the patch at refinement level i.

vex hull of a set. The *triangle normal* $\mathbf{n}_t$ of a triangle $T_t = [\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k]$ is defined as the normalization of the vector $(\mathbf{v}_j - \mathbf{v}_i) \times (\mathbf{v}_k - \mathbf{v}_i)$. Since our interest is in rendering Ω, we also assume that we are given shading normals, $\mathbf{n}_{ti}, \mathbf{n}_{tj}, \mathbf{n}_{tk}$ associated with the vertices of T_t. The *viewpoint* $\mathbf{x} \in \mathbb{R}^3$ is the position of the observer and for a point $\mathbf{v}$ on Ω, the *view direction vector* is $\mathbf{v} - \mathbf{x}$. If $\mathbf{n}$ is the surface normal in $\mathbf{v}$, we say that T is *front facing* in $\mathbf{v}$ if $(\mathbf{v} - \mathbf{x}) \cdot \mathbf{n} \leq 0$, otherwise it is *back facing*.

The silhouette of a triangle mesh is the set of edges where one of the adjacent triangles is front facing while the other is back facing. Let $\mathbf{v}_{ij}$ be the midpoint of an edge e_{ij} shared by two triangles T_s and T_t in Ω. Defining $f_{ij} : \mathbb{R}^3 \to \mathbb{R}$ by

$$f_{ij}(\mathbf{x}) = \left(\frac{\mathbf{v}_{ij} - \mathbf{x}}{\| \mathbf{v}_{ij} - \mathbf{x} \|} \cdot \mathbf{n}_s \right) \left(\frac{\mathbf{v}_{ij} - \mathbf{x}}{\| \mathbf{v}_{ij} - \mathbf{x} \|} \cdot \mathbf{n}_t \right), \tag{1}$$

we see that e_{ij} is a silhouette edge when observed from $\mathbf{x}$ in the case $f_{ij}(\mathbf{x}) \leq 0$.

Our objective is to render Ω so that it appears to have smooth silhouettes, by adaptively refining the mesh along the silhouettes. Since the resulting silhouette curves in general do not lie in Ω, and since silhouette membership for edges is a binary function of the viewpoint, a naive implementation leads to transitional artifacts: The rendered geometry depends discontinuously on the viewpoint. In [7], a continuous silhouette test was proposed to avoid such artifacts. The *silhouetteness* of e_{ij} as seen from $\mathbf{x} \in \mathbb{R}^3$ was defined as

$$\alpha_{ij}(\mathbf{x}) = \begin{cases} 1 & \text{if } f_{ij}(\mathbf{x}) \leq 0; \\ 1 - \frac{f_{ij}(\mathbf{x})}{\beta_{ij}} & \text{if } 0 < f_{ij}(\mathbf{x}) \leq \beta_{ij}; \\ 0 & \text{if } f_{ij}(\mathbf{x}) > \beta_{ij}, \end{cases} \tag{2}$$

where $\beta_{ij} > 0$ is a constant. We let β_{ij} depend on the local geometry, so that the transitional region define a "wedge" with angle ϕ with the adjacent triangles, see Figure 3. This amounts to setting $\beta_{ij} = \sin\phi\cos\phi\sin\theta + \sin^2\phi\cos\theta$, where θ is the angle between the normals of the two adjacent triangles. We also found that the heuristic choice of $\beta_{ij} = \frac{1}{4}$ works well in practice, but this choice gives unnecessary refinement over flatter areas.

The classification (2) extends the standard binary classification by adding a transitional region. A silhouetteness $\alpha_{ij} \in (0, 1)$ implies that e_{ij} is nearly a silhouette edge. We use silhouetteness to control the view dependent interpolation between silhouette geometry and non-silhouette geometry.

4 Curved geometry

We assume that the mesh Ω and its shading normals are sampled from a piecewise smooth surface (it can however have sharp edges) at a sampling rate that yields non-smooth silhouettes. In this section we use the vertices

Draft

and shading normals of each triangle in Ω to construct a corresponding cubic Bézier patch. The end result of our construction is a set of triangular patches that constitutes a piecewise smooth surface. Our construction is a variant of the one in [7], see also [22].

For each edge e_{ij} in Ω, we determine a cubic Bézier curve based on the edge endpoints $\mathbf{v}_i, \mathbf{v}_j$ and their associated shading normals:

$$\mathbf{C}_{ij}(t) = \mathbf{v}_i B_0^3(t) + \mathbf{c}_{ij} B_1^3(t) + \mathbf{c}_{ji} B_2^3(t) + \mathbf{v}_j B_3^3(t),$$

where $B_i^3(t) = \binom{3}{i} t^i (1-t)^{3-i}$ are the Bernstein polynomials, see e.g. [8]. The inner control point $\mathbf{c}_{ij}$ is determined as follows. Let T_s and T_t be the two triangles adjacent to e_{ij} and let $\mathbf{n}_{si}$ and $\mathbf{n}_{ti}$ be the shading normals at $\mathbf{v}_i$ belonging to triangle T_s and T_t respectively. If $\mathbf{n}_{si} = \mathbf{n}_{ti}$, we say that $\mathbf{v}_i$ is a *smooth edge end* and we determine its inner control point as $\mathbf{c}_{ij} = \frac{2\mathbf{v}_i + \mathbf{v}_j}{3} - \frac{(\mathbf{v}_j - \mathbf{v}_i)\cdot \mathbf{n}_{si}}{3} \mathbf{n}_{si}$. On the other hand, if $\mathbf{n}_{si} \neq \mathbf{n}_{ti}$, we say that $\mathbf{v}_i$ is a *sharp edge end* and let $\mathbf{c}_{ij} = \mathbf{v}_i + \frac{(\mathbf{v}_j - \mathbf{v}_i)\cdot \mathbf{t}_{ij}}{3} \mathbf{t}_{ij}$, where $\mathbf{t}_{ij}$ is the normalized cross product of the shading normals. We refer to [7] for the rationale behind this construction.

Next, we use the control points of the three edge curves belonging to a triangle $[\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k]$ to define nine of the ten control points of a cubic triangular Bézier patch of the form

$$\mathcal{F} = \sum_{l+m+n=3} \mathbf{b}_{lmn} B_{lmn}^3. \tag{3}$$

Here $B_{lmn}^3 = \frac{6}{l!m!n!} u^l v^m w^n$ are the Bernstein-Bézier polynomials and u, v, w are barycentric coordinates, see e.g. [8]. We determine the coefficients such that $\mathbf{b}_{300} = \mathbf{v}_i$, $\mathbf{b}_{210} = \mathbf{c}_{ij}$, $\mathbf{b}_{120} = \mathbf{c}_{ji}$ and so forth. In [22] and [7] the center control point $\mathbf{b}_{111}$ was determined as

$$\mathbf{b}_{111} = \tfrac{3}{12}\left(\mathbf{c}_{ij} + \mathbf{c}_{ji} + \mathbf{c}_{jk} + \mathbf{c}_{kj} + \mathbf{c}_{ki} + \mathbf{c}_{ik}\right) - \tfrac{1}{6}\left(\mathbf{v}_i + \mathbf{v}_j + \mathbf{v}_k\right). \tag{4}$$

We propose instead to optionally use the average of the inner control points of the three edge curves,

$$\mathbf{b}_{111} = \tfrac{1}{6}\left(\mathbf{c}_{ij} + \mathbf{c}_{ji} + \mathbf{c}_{jk} + \mathbf{c}_{kj} + \mathbf{c}_{ki} + \mathbf{c}_{ik}\right). \tag{5}$$

This choice allows for a significantly more efficient implementation at the cost of a slightly "flatter" patch, see Section 6.4 and Figure 4. This example is typical in that the patches resulting from the two formulations are visually almost indistinguishable.

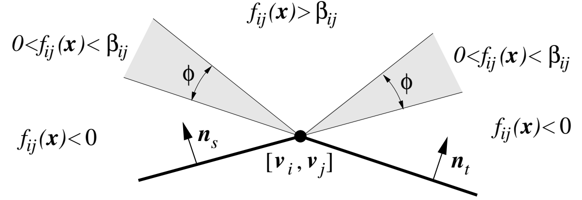


Figure 3: Values of f_{ij} in (1) looking along the edge e_{ij} with the transitional region ϕ marked gray.

5 Adaptive tessellation

In the previous section we defined a collection of cubic Bézier patches based on the mesh Ω and its shading normals. We next propose a strategy for tessellating these patches adaptively for rendering. We let the tessellation level (which controls the number of triangles produced) depend on the local silhouetteness so that silhouettes appear to be smooth, while retaining the coarse geometry away from the silhouettes. We avoid "cracking" by ensuring that the tessellations of neighboring patches meet continuously, see Figure 1.

The parameter domain of a triangular Bézier patch $\mathcal{F}$ is a triangle $\mathcal{P}_0 \subset \mathbb{R}^2$. We can refine $\mathcal{P}_0$ to form a triangle mesh $\mathcal{P}_1$ by splitting each edge in half and forming four new triangles. A further refinement $\mathcal{P}_2$ of $\mathcal{P}_1$ is formed similarly, by splitting each triangle in $\mathcal{P}_1$ in four new triangles, and so forth. The m'th refinement $\mathcal{P}_m$ is a triangle mesh with vertices at the dyadic barycentric coordinates

$$\mathbb{I}_m = \left\{ \frac{(i,j,k)}{2^m} : \quad i,j,k \in \mathbb{Z}^+, \quad i+j+k = 2^m \right\}. \tag{6}$$

A tessellation $\mathcal{P}_m$ of the parameter domain $\mathcal{P}_0$ and a map $f : \mathcal{P}_0 \to \mathbb{R}^d$, gives rise to a continuous approximation $\mathcal{S}_m[f] : \mathcal{P}_0 \to \mathbb{R}^d$ of f that is linear on each triangle of $\mathcal{P}_m$ and agrees with f at the vertices of $\mathcal{P}_m$. For example, $\mathcal{S}_m[\mathcal{F}]$ maps a triangle $[\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k]$ in $\mathcal{P}_m$ linearly to a triangle $[\mathcal{F}(\mathbf{p}_i), \mathcal{F}(\mathbf{p}_j), \mathcal{F}(\mathbf{p}_k)]$ in $\mathbb{R}^3$. It is clear that the collection of all such triangles forms a tessellation of $\mathcal{F}$. We will in the following call both the map $\mathcal{S}_m[\mathcal{F}]$ and its image $\mathcal{S}_m[\mathcal{F}](\mathcal{P}_0)$ a tessellation. A piecewise linear map $\mathcal{S}_m[f]$ can be evaluated at a point $\mathbf{p} \in \mathcal{P}_0$ as follows: Let $T = [\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k]$ be a triangle in $\mathcal{P}_m$ containing $\mathbf{p}$ and let (u_i, u_j, u_k) be the barycentric coordinates of $\mathbf{p}$ with

Draft

respect to T. Then $\mathbf{p} = u_i\mathbf{p}_i + u_j\mathbf{p}_j + u_k\mathbf{p}_k$ and

$$\mathcal{S}_m[f](\mathbf{p}) = u_i f(\mathbf{p}_i) + u_j f(\mathbf{p}_j) + u_k f(\mathbf{p}_k). \quad (7)$$

Given two tessellations $\mathcal{P}_s$ and $\mathcal{P}_m$ and two integers $s \leq m$, the set of vertices of $\mathcal{P}_s$ is contained in the set of vertices of $\mathcal{P}_m$ and a triangle of $\mathcal{P}_m$ is contained in a triangle of $\mathcal{P}_s$. Since both maps are linear on each triangle of $\mathcal{S}_m\big[\mathcal{S}_s[f]\big]$ and agrees at the corners, the two maps must be equal in the whole of $\mathcal{P}_0$. This implies that a tessellation can be refined to a finer level without changing its geometry: Given a map $f : \mathcal{P}_0 \to \mathbb{R}^d$, we have a corresponding tessellation

$$\mathcal{S}_m\big[\mathcal{S}_s[f]\big] = \mathcal{S}_s[f]. \quad (8)$$

We say that $\mathcal{S}_m\big[\mathcal{S}_s[f]\big]$ has *topological refinement level* m and *geometric refinement level* s. From the previous result we can define tessellations for a non-integer refinement level $s = m + \alpha$ where m is an integer and $\alpha \in [0, 1)$. We refine $\mathcal{S}_m[f]$ to refinement level $m+1$ and let α control the blend between the two refinement levels,

$$\mathcal{S}_{m+\alpha}[f] = (1-\alpha)\mathcal{S}_{m+1}\left[\mathcal{S}_m[f]\right] + \alpha\mathcal{S}_{m+1}[f]. \quad (9)$$

See Figure 5 for an illustration of non-integer level tessellation. The sampling operator $\mathcal{S}_m$ is linear, i.e. $\mathcal{S}_m[\alpha_1 f_1 + \alpha_2 f_2] = \alpha_1\mathcal{S}_m[f_1] + \alpha_2\mathcal{S}_m[f_2]$ for all real α_1, α_2 and maps f_1, f_2. As a consequence, (8) holds for non-integer geometric refinement level s.

Our objective is to define for each triangle $T = [\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k]$ a tessellation $\mathcal{T}$ of the corresponding patch $\mathcal{F}$ adaptively with respect to the silhouetteness of the edges e_{ij}, e_{jk}, e_{ki}. To that end we assign a geometric refinement level $s_{ij} \in \mathbb{R}^+$ to each edge, based on its silhouetteness as computed in Section 3. More precisely, we use $s_{ij} = M\alpha_{ij}$ where M is a user defined maximal refinement level, typically $M = 3$. We set the topological refinement level for a triangle to be $m = \lceil \max\{s_{ij}, s_{jk}, s_{ki}\} \rceil$, i.e. our tessellation $\mathcal{T}$ has the same topology as $\mathcal{P}_m$. Now, it only remains to determine the position of the vertices of $\mathcal{T}$. We use the sampling operator $\mathcal{S}_s$ with geometric refinement level varying over the patch and define the vertex positions as follows. For a vertex $\mathbf{p}$ of $\mathcal{P}_m$ we let the geometric refinement level be

$$s(\mathbf{p}) = \begin{cases} s_{qr} & \text{if } \mathbf{p} \in (\mathbf{p}_q, \mathbf{p}_r); \\ \max\{s_{ij}, s_{jk}, s_{ki}\} & \text{otherwise,} \end{cases} \quad (10)$$

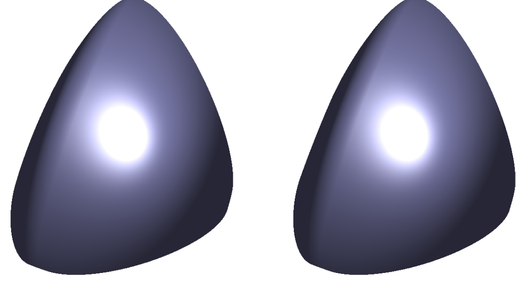


Figure 4: The surfaces resulting from the center control point rule (4) (left) and (5) (right), applied to a tetrahedron with one normal vector per vertex. The difference is marginal, although the surface to the right can be seen to be slightly flatter.

where $(\mathbf{p}_q, \mathbf{p}_r)$ is the interior of the edge of $\mathcal{P}_0$ corresponding to e_{qr}. Note that the patch is interpolated at the corners $\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k$. The vertex $\mathbf{v}$ of $\mathcal{T}$ that corresponds to $\mathbf{p}$ is then defined as

$$\mathbf{v} = \mathcal{S}_{s(\mathbf{p})}[\mathcal{F}](\mathbf{p}) = \mathcal{S}_m\left[\mathcal{S}_{s(\mathbf{p})}[\mathcal{F}]\right](\mathbf{p}). \quad (11)$$

Note that $s(\mathbf{p})$ is in general a real value and so (9) is used in the above calculation. The final tessellation is illustrated in Figure 6.

The topological refinement level of two neighboring patches will in general not be equal. However, our choice of constant geometric refinement level along an edge ensures that neighboring tessellations match along the common boundary. Although one could let the geometric refinement level $s(\mathbf{p})$ vary over the interior of the patch, we found that taking it to be constant as in (10) gives good results.

6 Implementation

We next describe our implementation of the algorithm. We need to distinguish between *static meshes* for which the vertices are only subject to affine transformations, and *dynamic meshes* with more complex vertex transformations. Examples of the latter are animated meshes and meshes resulting from physical simulations. We assume that the geometry of a dynamic mesh is retained in a texture on the GPU that is updated between frames. This

Draft

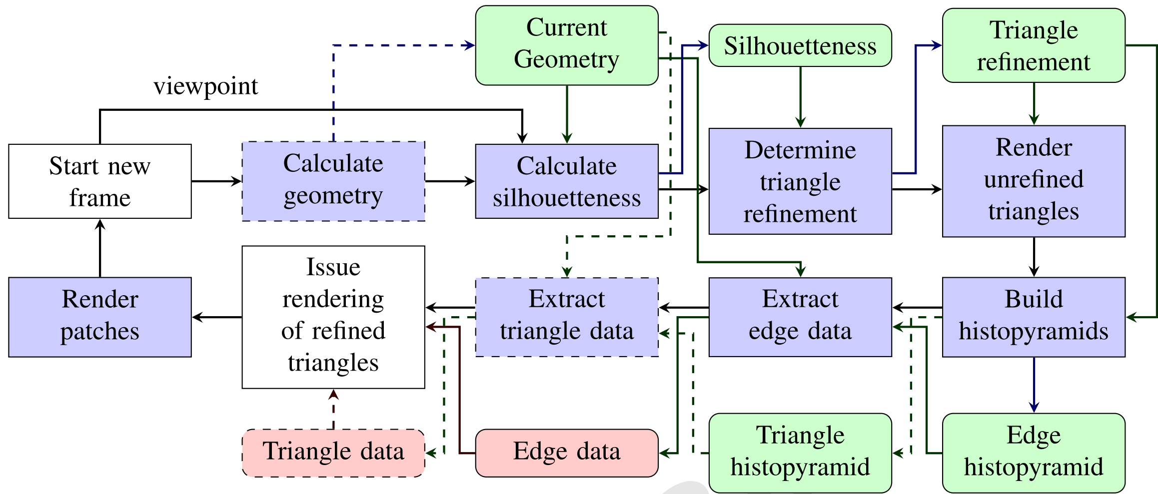


Figure 7: Flowchart of the silhouette refinement algorithm. The white boxes are executed on the CPU, the blue boxes on the GPU, the green boxes are textures, and the red boxes are pixel buffer objects. The dashed lines and boxes are only necessary for dynamic geometry.

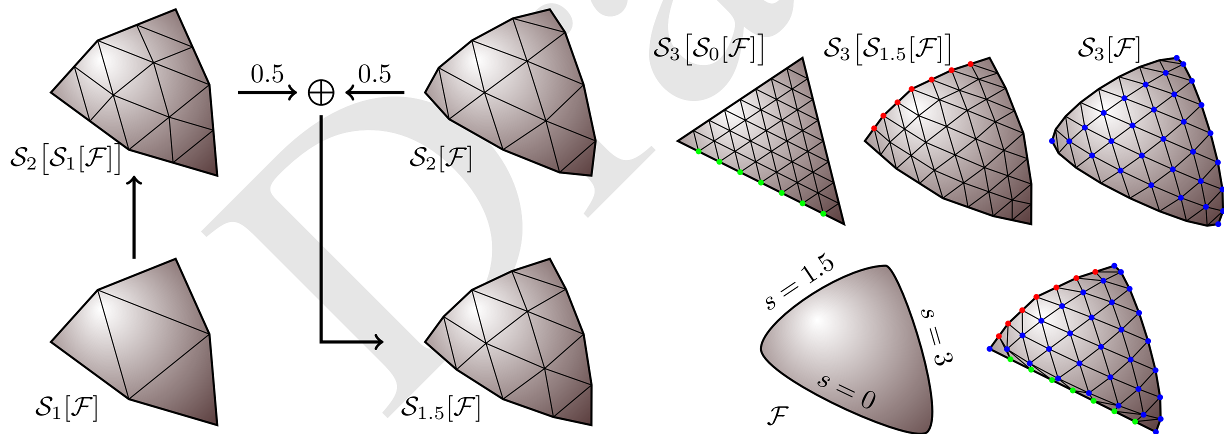


Figure 5: To tessellate a patch at the non-integer refinement level $s = 1.5$, we create the tessellations $\mathcal{S}_1[\mathcal{F}]$ and $\mathcal{S}_2[\mathcal{F}]$, and refine $\mathcal{S}_1[\mathcal{F}]$ to $\mathcal{S}_2[\mathcal{S}_1[\mathcal{F}]]$ such that the topological refinement levels match. Then, the two surfaces are weighted and combined to form $\mathcal{S}_{1.5}[\mathcal{F}]$.

Figure 6: Composing multiple refinement levels for adaptive tessellation. Each edge have a geometric refinement level, and the topological refinement level is dictated by the edge with the largest refinement level.

Draft

implies that our algorithm must recompute the Bézier coefficients accordingly. Our implementation is described sequentially, although some steps do not require the previous step to finish. A flowchart of the implementation can be found in Figure 7.

6.1 Silhouetteness calculation

Silhouetteness is well suited for computation on the GPU since it is the same transform applied to every edge and since there are no data dependencies. The only changing parameter between frames is the viewpoint.

If the mesh is static we can pre-compute the edge-midpoints and neighboring triangle normals for every edge as a preprocessing step and store these values in a texture on the GPU. For a dynamic mesh we store the indices of the vertices of the two adjacent triangles instead and calculate the midpoint as part of the silhouetteness computation.

The silhouetteness of the edges is calculated by first sending the current viewpoint to the GPU as a shader uniform, and then by issuing the rendering of a textured rectangle into an off-screen buffer with the same size as our edge-midpoint texture.

We could alternatively store the edges and normals of several meshes in one texture and calculate the silhouetteness of all in one pass. If the models have different model space bases, such as in a scene graph, we reserve a texel in a viewpoint-texture for each model. In the preprocessing step, we additionally create a texture associating the edges with the model's viewpoint texel. During rendering we traverse the scene graph, find the viewpoint in the model space of the model and store this in the viewpoint texture. We then upload this texture instead of setting the viewpoint explicitly.

6.2 Histogram pyramid construction and extraction

The next step is to determine which triangles should be refined, based on the silhouetteness of the edges. The straightforward approach is to read back the silhouetteness texture to host memory and run sequentially through the triangles to determine the refinement level for each of them. This direct approach rapidly congests the graphics bus and thus reduces performance. To minimize transfers over the bus we use a technique called *histogram pyramid extraction* [23] to find and compact only the data that we need to extract for triangle refinement. As an added benefit the process is performed in parallel on the GPU.

The first step in the histogram pyramid extraction is to select the elements that we will extract. We first create a binary base texture with one texel per triangle in the mesh. A texel is set to 1 if the corresponding triangle is selected for extraction, i.e. has at least one edge with non-zero silhouetteness, and 0 otherwise. We create a similar base texture for the edges, setting a texel to 1 if the corresponding edge has at least one adjacent triangle that is selected and 0 otherwise.

For each of these textures we build a histopyramid, which is a stack of textures similar to a mipmap pyramid. The texture at one level is a quarter of the size of the previous level. Instead of storing the average of the four corresponding texels in the layer below like for a mipmap, we store the sum of these texels. Thus each texel in the histopyramid contains the number of selected elements in the sub-pyramid below and the single top element contains the total number of elements selected for extraction. Moreover, the histopyramid induces an ordering of the selected elements that can be obtained by traversal of the pyramid. If the base texture is of size $2^n \times 2^n$, the histopyramid is built bottom up in n passes. Note that for a static mesh we only need a histopyramid for edge extraction and can thus skip the triangle histopyramid.

The next step is to compact the selected elements. We create a 2D texture with at least m texels where m is the number of selected elements and each texel equals its index in a 1D ordering. A shader program is then used to find for each texel i the corresponding element in the base texture as follows. If $i > m$ there is no corresponding selected element and the texel is discarded. Otherwise, the pyramid is traversed top-down using partial sums at the intermediate levels to determine the position of the i'th selected element in the base texture. Its position in the base texture is then recorded in a pixel buffer object.

The result of the histopyramid extraction is a compact representation of the texture positions of the elements for which we need to extract data. The final step is to load associated data into pixel buffer objects and read them back to host memory over the graphics bus. For static meshes we output for each selected edge its index and silhouetteness. We can thus fit the data of two edges in the RGBA

Draft

values of one render target.

For dynamic meshes we extract data for both the selected triangles and edges. The data for a triangle contains the vertex positions and shading normals of the corners of the triangle. Using polar coordinates for normal vectors, this fit into four RGBA render targets. The edge data is the edge index, its silhouetteness and the two inner Bézier control points of that edge, all of which fits into two RGBA render targets.

6.3 Rendering unrefined triangles

While the histopyramid construction step finishes, we issue the rendering of the unrefined geometry using a VBO (vertex buffer object). We encode the triangle index into the geometry stream, for example as the w-coordinates of the vertices. In the vertex shader, we use the triangle index to do a texture lookup in the triangle refinement texture in order to check if the triangle is tagged for refinement. If so, we collapse the vertex to $[0, 0, 0]$, such that triangles tagged for refinement are degenerate and hence produce no fragments. This is the same approach as [5] uses to discard geometry.

For static meshes, we pass the geometry directly from the VBO to vertex transform, where triangles tagged for refinement are culled. For dynamic meshes, we replace the geometry in the VBO with indices and retrieve the geometry for the current frame using texture lookups, before culling and applying the vertex transforms.

The net effect of this pass is the rendering of the coarse geometry, with holes where triangles are tagged for refinement. Since this pass is vertex-shader only, it can be combined with any regular fragment shader for lightning and shading.

6.4 Rendering refined triangles

While the unrefined triangles are rendered, we wait for the triangle data read back to the host memory to finish. We can then issue the rendering of the refined triangles. The geometry of the refined triangles are tessellations of triangular cubic Bézier patches as described in Section 4 and 5.

To allow for high frame-rates, the transfer of geometry to the GPU, as well as the evaluation of the surface, must be handled carefully. Transfer of vertex data over the graphics bus is a major bottleneck when rendering geometry. Boubekeur et.al. [3] have an efficient strategy for rendering uniformly sampled patches. The idea is that the parameter positions and triangle strip set-up are the same for any patch with the same topological refinement level. Thus, it is enough to store a small number of pre-tessellated patches $\mathcal{P}_i$ with parameter positions $\mathbb{I}_i$ as VBOs on the GPU. The coefficients of each patch are uploaded and the vertex shader is used to evaluate the surface at the given parameter positions. We use a similar set-up, extended to our adaptive watertight tessellations.

The algorithm is similar for static and dynamic meshes, the only difference is the place from which we read the coefficients. For static meshes, the coefficients are pregenerated and read directly from host memory. The coefficients for a dynamic mesh are obtained from the edge and triangle read-back pixel pack buffers. Note that this pass is vertex-shader only and we can therefore use the same fragment shader as for the rendering of the unrefined triangles.

The tessellation strategy described in Section 5 requires the evaluation of (11) at the vertices of the tessellation $\mathcal{P}_m$ of the parameter domain of $\mathcal{F}$, i.e. at the dyadic parameter points (6) at refinement level m. Since the maximal refinement level M over all patches is usually quite small, we can save computations by pre-evaluating the basis functions at these points and store these values in a texture.

A texture lookup gives four channels of data, and since a cubic Bézier patch has ten basis functions, we need three texture lookups to get the values of all of them at a point. If we define the center control point $\mathbf{b}_{111}$ to be the average of six other control points, as in (5), we can eliminate it by distributing the associated basis function $B_{111} = uvw/6 = \mu/6$ among the six corresponding basis functions,

$$
\begin{aligned}
\hat{B}^3_{300} &= u^3, & \hat{B}^3_{201} &= 3wu^2{+}\mu, & \hat{B}^3_{102} &= 3uw^2{+}\mu,\\
\hat{B}^3_{030} &= v^3, & \hat{B}^3_{120} &= 3uv^2{+}\mu, & \hat{B}^3_{210} &= 3vu^2{+}\mu,\\
\hat{B}^3_{003} &= w^3, & \hat{B}^3_{012} &= 3vw^2{+}\mu, & \hat{B}^3_{021} &= 3wv^2{+}\mu.
\end{aligned} \qquad (12)
$$

We thus obtain a simplified expression

$$
\mathcal{F} = \sum b_{ijk} B_{ijk} = \sum_{(i,j,k)\neq(1,1,1)} b_{ijk} \hat{B}_{ijk} \qquad (13)
$$

Draft

involving only nine basis functions. Since they form a partition of unity, we can obtain one of them from the remaining eight. Therefore, it suffices to store the values of eight basis functions, and we need only two texture lookups for evaluation per point. Note that if we choose the center coefficient as in (4) we need three texture lookups for retrieving the basis functions, but the remainder of the shader is essentially the same.

Due to the linearity of the sampling operator, we may express (11) for a vertex $\mathbf{p}$ of $\mathcal{P}_M$ with $s(\mathbf{p}) = m + \alpha$ as

$$\mathbf{v} = \mathcal{S}_{s(\mathbf{p})}[\mathcal{F}](\mathbf{p}) = \sum_{i,j,k} \mathbf{b}_{ijk} \mathcal{S}_{s(\mathbf{p})}[\hat{B}_{ijk}](\mathbf{p}) \tag{14}$$

$$= \sum_{i,j,k} \mathbf{b}_{ijk} \Big((1-\alpha)\mathcal{S}_m[\hat{B}_{ijk}](\mathbf{p}) + \alpha \mathcal{S}_{m+1}[\hat{B}_{ijk}](\mathbf{p}) \Big) .$$

Thus, for every vertex $\mathbf{p}$ of $\mathcal{P}_M$, we pre-evaluate $\mathcal{S}_m[\hat{B}^3_{300}](\mathbf{p}), \ldots, \mathcal{S}_m[\hat{B}^3_{021}](\mathbf{p})$ for every refinement level $m = 1, \ldots, M$ and store this in a $M \times 2$ block in the texture. We organize the texture such that four basis functions are located next to the four corresponding basis functions of the adjacent refinement level. This layout optimizes spatial coherency of texture accesses since two adjacent refinement levels are always accessed when a vertex is calculated. Also, if vertex shaders on future graphics hardware will support filtered texture lookups, we could increase performance by carrying out the linear interpolation between refinement levels by sampling between texel centers.

Since the values of our basis function are always in in the interval $[0, 1]$, we can trade precision for performance and pack two basis functions into one channel of data, letting one basis function have the integer part while the other has the fractional part of a channel. This reduces the precision to about 12 bits, but increases the speed of the algorithm by 20% without adding visual artifacts.

6.5 Normal and displacement mapping

Our algorithm can be adjusted to accommodate most regular rendering techniques. Pure fragment level techniques can be applied directly, but vertex-level techniques may need some adjustment.

An example of a pure fragment-level technique is normal mapping. The idea is to store a dense sampling of the object's normal field in a texture, and in the fragment shader use the normal from this texture instead of the interpolated normal for lighting calculations. The result of using normal mapping on a coarse mesh is depicted in the left of Figure 8.

Normal mapping only modulates the lighting calculations, it does not alter the geometry. Thus, silhouettes are still piecewise linear. In addition, the flat geometry is distinctively visible at gracing angles, which is the case for the sea surface in Figure 8.

The displacement mapping technique attacks this problem by perturbing the vertices of a mesh. The drawback is that displacement mapping requires the geometry in problem areas to be densely tessellated. The brute force strategy of tessellating the whole mesh increase the complexity significantly and is best suited for off-line rendering. However, a ray-tracing like approach using GPUs has been demonstrated by Donnelly [6].

We can use our strategy to establish the problem areas of the current frame and use our variable-level of detail refinement strategy to tessellate these areas. First, we augment the silhouetteness test, tagging edges that are large in the current projection and part of planar regions at gracing angles for refinement. Then we incorporate displacement mapping in the vertex shader of Section 6.4. However, care must be taken to avoid cracks and maintain a watertight surface.

For a point $\mathbf{p}$ at integer refinement level s, we find the triangle $T = [\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k]$ of $\mathcal{P}_s$ that contains $\mathbf{p}$. We then find the displacement vectors at $\mathbf{p}_i$, $\mathbf{p}_j$, and $\mathbf{p}_k$. The displacement vector at $\mathbf{p}_i$ is found by first doing a texture lookup in the displacement map using the texture coordinates at $\mathbf{p}_i$ and then multiplying this displacement with the interpolated shading normal at $\mathbf{p}_i$. In the same fashion we find the displacement vectors at $\mathbf{p}_j$ and $\mathbf{p}_k$. The three displacement vectors are then combined using the barycentric weights of $\mathbf{p}$ with respect to T, resulting in a displacement vector at $\mathbf{p}$. If s is not an integer, we interpolate the displacement vectors of two adjacent levels similarly to (9).

The result of this approach is depicted to the right in Figure 8, where the cliff ridges are appropriately jagged and the water surface is displaced according to the waves.

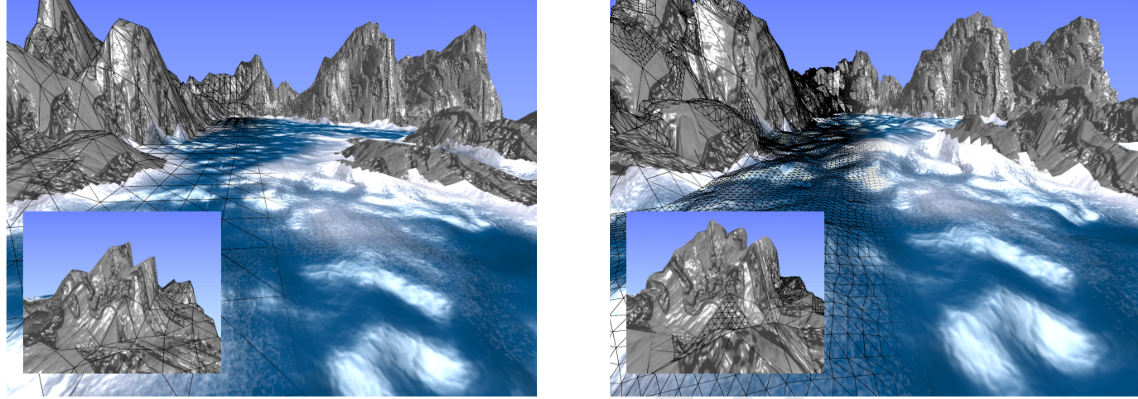
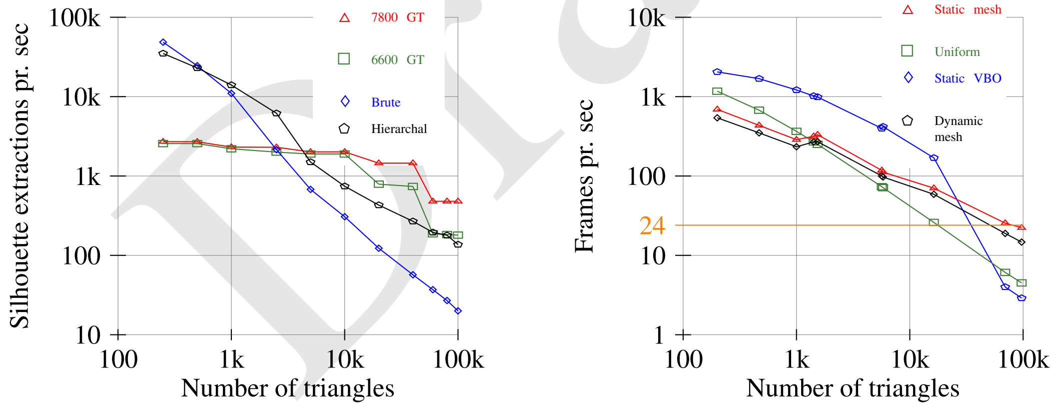


Figure 8: The left image depicts a coarse mesh using normal mapping to increase the perceived detail, and the right image depicts the same scene using the displacement mapping technique described in Section 6.5.

Draft



(a) The measured performance for brute force CPU silhouette extraction, hierarchical CPU silhouette extraction, and the GPU silhouette extraction on the Nvidia GeForce 6600GT and 7800GT GPUs.

(b) The measured performance on an Nvidia GeForce 7800GT for rendering refined meshes using one single static VBO, the uniform refinement method of [3], and our algorithm for static and dynamic meshes.

Figure 9: Performance measurements of our algorithm.

7 Performance analysis

In this section we compare our algorithms to alternative methods. We have measured both the speedup gained by moving the silhouetteness test calculation to the GPU, as well as the performance of the full algorithm (silhouette extraction + adaptive tessellation) with a rapidly changing viewpoint. We have executed our benchmarks on two different GPUs to get an impression of how the algorithm scales with advances in GPU technology.

For all tests, the CPU is an AMD Athlon 64 running at 2.2GHz with PCI-E graphics bus, running Linux 2.6.16 and using GCC 4.0.3. The Nvidia graphics driver is version 87.74. All programs have been compiled with full optimization settings. We have used two different Nvidia GPUs, a 6600 GT running at 500MHz with 8 fragment and 5 vertex pipelines and a memory clockspeed of 900MHz, and a 7800 GT running at 400MHz with 20 fragment and 7 vertex pipelines and a memory clockspeed of 1000Mhz. Both GPUs use the PCI-E interface for communication with the host CPU.

Our adaptive refinement relies heavily on texture lookups in the vertex shader. Hence, we have not been able to perform tests on ATI GPUs, since these just recently got this ability. However, we expect similar performance on ATI hardware.

We benchmarked using various meshes ranging from 200 to $100k$ triangles. In general, we have found that the size of a mesh is more important than its complexity and topology, an observation shared by Hartner et.al. [10]. However, for adaptive refinement it is clear that a mesh with many internal silhouettes will lead to more refinement, and hence lower frame-rates.

7.1 Silhouette Extraction on the GPU

To compare the performance of silhouette extraction on the GPU versus traditional CPU approaches, we implemented our method in the framework of Hartner et.al. [10]. This allowed us to benchmark our method against the hierarchical method described by Sander et.al. [18], as well as against standard brute force silhouette extraction. Figure 9(a) shows the average performance over a large number of frames with random viewpoints for an asteroid model of [10] at various levels of detail. The GPU measurements include time spent reading back the data to host memory.

Our observations for the CPU based methods (hierarchical and brute force) agree with [10]. For small meshes that fit into the CPU's cache, the brute force method is the fastest. However, as the mesh size increases, we see the expected linear decrease in performance. The hierarchical approach scales much better with regard to mesh size, but at around 5k triangles the GPU based method begins to outperform this method as well.

The GPU based method has a different performance characteristic than the CPU based methods. There is virtually no decline in performance for meshes up to about 10k triangles. This is probably due to the dominance of set-up and tear-down operations for data transfer across the bus. At around 10k triangles we suddenly see a difference in performance between the 8-pipeline 6600GT GPU and the 20-pipeline 7800GT GPU, indicating that the increased floating point performance of the 7800GT GPU starts to pay off. We also see clear performance plateaus, which is probably due to the fact that geometry textures for several consecutive mesh sizes are padded to the same size during histopyramid construction.

It could be argued that coarse meshes in need of refinement along the silhouette typically contains less than 5000 triangles and thus silhouetteness should be computed on the CPU. However, since the test can be carried out for any number of objects at the same time, the above result applies to the total number of triangles in the scene, and not in a single mesh.

For the hierarchical approach, there is a significant preprocessing step that is not reflected in Figure 9(a), which makes it unsuitable for dynamic meshes. Also, in realtime rendering applications, the CPU will typically be used for other calculations such as physics and AI, and can not be fully utilized to perform silhouetteness calculations. It should also be emphasized that it is possible to do other per-edge calculations, such as visibility testing and culling, in the same render pass as the silhouette extraction, at little performance overhead.

7.2 Benchmarks of the complete algorithm

Using variable level of detail instead of uniform refinement should increase rendering performance since less geometry needs to be sent through the pipeline. However,

Draft

the added complexity balances out the performance of the two approaches to some extent.

We have tested against two methods of uniform refinement. The first method is to render the entire *refined* mesh as a static VBO stored in graphics memory. The rendering of such a mesh is fast, as there is no transfer of geometry across the graphics bus. However, the mesh is static and the VBO consumes a significant amount of graphics memory. The second approach is the method of Boubekeur and Schlick [3], where each triangle triggers the rendering of a pre-tessellated patch stored as triangle strips in a static VBO in graphics memory.

Figure 9(b) shows these two methods against our adaptive method. It is clear from the graph that using static VBOs is extremely fast and outperforms the other methods for meshes up to 20k triangles. At around 80k triangles, the VBO grows too big for graphics memory, and is stored in host memory, with a dramatic drop in performance. The method of [3] has a linear performance degradation, but the added cost of triggering the rendering of many small VBOs is outperformed by our adaptive method at around 1k triangles. The performance of our method also degrades linearly, but at a slower rate than uniform refinement. Using our method, we are at 24 FPS able to adaptively refine meshes up to 60k for dynamic meshes, and 100k triangles for static meshes, which is significantly better than the other methods. The other GPUs show the same performance profile as the 7800 in Figure 9(b), just shifted downward as expected by the number of pipelines and lower clock speed.

Finally, to get an idea of the performance impact of various parts of our algorithm, we ran the same tests with various features enabled or disabled. We found that using uniformly distributed random refinement level for each edge (to avoid the silhouetteness test), the performance is 30–50% faster than uniform refinement. This is as expected since the vertex shader is only marginally more complex, and the total number of vertices processed is reduced. In a real world scenario, where there is often a high degree of frame coherency, this can be utilized by not calculating the silhouetteness for every frame. Further, if we disable blending of consecutive refinement levels (which can lead to some popping, but no cracking), we remove half of the texture lookups in the vertex shader for refined geometry and gain a 10% performance increase.

8 Conclusion and future work

We have proposed a technique for performing adaptive refinement of triangle meshes using graphics hardware, requiring just a small amount of preprocessing, and with no changes to the way the underlying geometry is stored. Our criterion for adaptive refinement is based on improving the visual appearance of the silhouettes of the mesh. However, our method is general in the sense that it can easily be adapted to other refinement criteria, as shown in Section 6.5.

We execute the silhouetteness computations on a GPU. Our performance analysis shows that our implementation using histogram pyramid extraction outperforms other silhouette extraction algorithms as the mesh size increases.

Our technique for adaptive level of detail automatically avoids cracking between adjacent patches with arbitrary refinement levels. Thus, there is no need to "grow" refinement levels from patch to patch, making sure two adjacent patches differ only by one level of detail. Our rendering technique is applicable to dynamic and static meshes and creates continuous level of detail for both uniform and adaptive refinement algorithms. It is transparent for fragment-level techniques such as texturing, advanced lighting calculations, and normal mapping, and the technique can be augmented with vertex-level techniques such as displacement mapping.

Our performance analysis shows that our technique gives interactive frame-rates for meshes with up to 100k triangles. We believe this makes the method attractive since it allows complex scenes with a high number of coarse meshes to be rendered with smooth silhouettes. The analysis also indicates that the performance of the technique is limited by the bandwidth between host and graphics memory. Since the CPU is available for other computations while waiting for results from the GPU, the technique is particularly suited for CPU-bound applications. This also shows that if one could somehow eliminate the read-back of silhouetteness and trigger the refinement directly on the graphics hardware, the performance is likely to increase significantly. To our knowledge there are no such methods using current versions of the OpenGL and Direct3D APIs. However, considering the recent evolution of both APIs, we expect such functionality in the near future.

A major contribution of this work is an extension of

Draft

the technique described in [3]. We address three issues: evaluation of PN-triangle type patches on vertex shaders, adaptive level of detail refinement and elimination of popping artifacts. We have proposed a simplified PN-triangle type patch which allows the use of pre-evaluated basis-functions requiring only one single texture lookup (if we pack the pre-evaluated basis functions into the fractional and rational parts of a texel). Further, the use of a geometric refinement level different from the topological refinement level comes at no cost since this is achieved simply by adjusting a texture coordinate. Thus, adaptive level of detail comes at a very low price.

We have shown that our method is efficient and we expect it to be even faster when texture lookups in the vertex shader become more mainstream and the hardware manufacturers answer with increased efficiency for this operation. Future GPUs use a unified shader approach, which could also boost the performance of our algorithm since it is primarily vertex bound and current GPUs perform the best for fragment processing.

Acknowledgments

We would like to thank Gernot Ziegler for introducing us to the histogram pyramid algorithm. Furthermore, we are grateful to Mark Hartner for giving us access to the source code of the various silhouette extraction algorithms. Finally, Marie Rognes has provided many helpful comments after reading an early draft of this manuscript. This work was funded, in part, by contract number 158911/I30 of The Research Council of Norway.

References

[1] P. Alliez, N. Laurent, and H. S. F. Schmitt. Efficient view-dependent refinement of 3D meshes using $\sqrt{3}$-subdivision. *The Visual Computer*, 19:205–221, 2003.

[2] T. Boubekeur, P. Reuter, and C. Schlick. Scalar tagged PN triangles. In *Eurographics 2005 (Short Papers)*, 2005.

[3] T. Boubekeur and C. Schlick. Generic mesh refinement on GPU. In *ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware*, pages 99–104, 2005.

[4] M. Bunnell. *GPU Gems 2*, chapter 7 Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. Addison-Wesley Professional, 2005.

[5] D. Card and J. L.Mitchell. *ShaderX*, chapter Non-Photorealistic Rendering with Pixel and Vertex Shaders. Wordware, 2002.

[6] W. Donnelly. *GPU Gems 2*, chapter 8 Per-Pixel Displacement Mapping with Distance Functions. Addison Wesley Professional, 2005.

[7] C. Dyken and M. Reimers. Real-time linear silhouette enhancement. In *Mathematical Methods for Curves and Surfaces: Tromsø 2004*, pages 135–144. Nashboro Press, 2004.

[8] G. Farin. *Curves and surfaces for CAGD*. Morgan Kaufmann Publishers Inc., 2002.

[9] M. Harris. *GPU Gems 2*, chapter 31 Mapping Computational Concepts to GPUs. Addison Wesley Professional, 2005.

[10] A. Hartner, M. Hartner, E. Cohen, and B. Gooch. Object space silhouette algorithims. In *Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production System SIGGRAPH 2003 Course Notes*, 2003.

[11] H. Hoppe. Progressive meshes. In *ACM SIGGRAPH 1996*, pages 99–108, 1996.

[12] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte. A developer's guide to silhouette algorithms for polygonal models. *IEEE Computer Graphics and Applications*, 23(4):28–37, July-Aug 2003.

[13] L. Kobbelt. $\sqrt{3}$-subdivision. In *ACM SIGGRAPH 2000*, pages 103–112, 2000.

[14] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., 2002.

Draft

[15] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.

[16] K. Pulli and M. Segal. Fast rendering of subdivision surfaces. In *ACM SIGGRAPH 1996 Visual Proceedings: The art and interdisciplinary programs*, 1996.

[17] R. J. Rost. *OpenGL(R) Shading Language*. Addison Wesley Longman Publishing Co., Inc., 2006.

[18] P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe, and J. Snyder. Silhouette clipping. In *SIGGRAPH 2000*, pages 327–334, 2000.

[19] L.-J. Shiue, I. Jones, and J. Peters. A realtime GPU subdivision kernel. *ACM Trans. Graph.*, 24(3):1010–1015, 2005.

[20] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide*. Addison Wesley Longman Publishing Co., Inc., 2006.

[21] K. van Overveld and B. Wyvill. An algorithm for polygon subdivision based on vertex normals. In *Computer Graphics International, 1997. Proceedings*, pages 3–12, 1997.

[22] A. Vlachos, J. Peters, C. Boyd, and J. L. Mitchell. Curved PN triangles. In *ACM Symposium on Interactive 3D 2001 graphics*, 2001.

[23] G. Ziegler, A. Tevs, C. Tehobalt, and H.-P. Seidel. Gpu point list generation through histogram pyramids. Technical Report MPI-I-2006-4-002, Max-Planck-Institut für Informatik, 2006.

Draft