Incremental Raycasting of Piecewise Quadratic Surfaces on the GPU

Carsten Stoll* Stefan Gumhold† Hans-Peter Seidel‡
MPI Informatik TU Dresden MPI Informatik

Abstract

To overcome the limitations of triangle and point based surfaces several authors have recently investigated surface representations that are based on higher order primitives. Among these are MPU, SLIM surfaces, dynamic skin surfaces and higher order iso-surfaces. Up to now these representations were not suitable for interactive applications because of the lack of an efficient rendering algorithm.

In this paper we close this gap for implicit surface representations of degree two by developing highly optimized GPU implementations of the raycasting algorithm. We investigate techniques for fast incremental raycasting and cover per fragment and per quadric backface culling. We apply the approaches to the rendering of SLIM surfaces, quadratic iso-surfaces over tetrahedral meshes and bilinear quadrilaterals. Compared to triangle based surface approximations of similar geometric error we achieve only slightly lower frame rates but with much higher visual quality due to the quadratic approximation power of the underlying surfaces.

Keywords: raytracing, graphics hardware.

Index Terms: I.3.7 [Computer Graphics]: Raytracing—GPU based raycasting;

1 Introduction

Second order primitives offer large advantages over traditional flat rendering primitives like triangles or points. Not only are the first order surface derivatives smooth (which is especially noticeable when using environment maps and phong shading), but it is also possible to visualize more complex geometry with a lower number of primitives.

Natural applications for such kind of primitives are thus surface reconstruction algorithms and methods which deal with modeling and deformation of such, where an interactive visualization is crucial.

To determine the shape and shading of a quadratic surface it is necessary to either sample it and render it using other primitives available in graphics hardware or to perform a per pixel raycasting to determine the appearance. We will show that a raycasting approach is not only viable but also fast enough for interactive visualization of models consisting of several 100k quadratic surface elements.

In this paper we will show how to exploit the processing capabilities of modern graphics hardware for efficient visualization of quadric surface primitives. Our contributions are

- An incremental raycasting approach that minimizes the computational load of the fragment shader and in this way significantly increases the fillrate.
- Conservative per quadric backface culling tests that increase performance by a factor of two.
- A reformulation of the expressions for the solution of the quadratic ray-quadric intersection that avoids an expensive test in the fragment shader.
- A conversion formula from bilinear quadrilaterals to quadrics that facilitates the application of our method also to bilinear quadrilaterals.

*e-mail: stoll@mpi-inf.mpg.de
†e-mail: sg30@mail.inf.tu-dresden.de
‡e-mail: hpseidel@mpi-inf.mpg.de

IEEE Symposium on Interactive Ray Tracing 2006
18 - 20 September, Salt Lake City, UT, USA
1-4244-0693-5/06/$20.00 ©2006 IEEE

2 Related Work

Graphics hardware development in the last years has made it more interesting to use other primitives for rendering than triangles. Often models have a density so high that a single triangle would cover not much more than a single pixel on the screen. This and a more general interest in points as primitives for other applications such as modeling led to the development of a large number of point splatting techniques such as from Rusinkiewicz and Levoy [17]. These techniques have grown more advanced in recent years, adding more information to the single splats to be able to visualize sharp corners like Pauly et al in [13] and generate smoother shading and better looking images like Botsch et al presented in [3] and [2] by using additional surface curvature information. All current point splatting approaches have in common that they are actually flat planes bounded by an ellipsoid and do not vary the actual geometry in each splat, meaning that often more splatting primitives have to be used to approximate geometry than would be necessary in a higher order surface approach.

Quadratic surfaces have been a basic primitive in computer graphics for a long time and their properties and techniques for visualization are very well researched (see for example Bloomenthal and Wyivills *Introduction to Implicit Surfaces* [1]).

Examples for recent techniques, that use piecewise quadratic or cubic surface patches to approximate or interpolate highly detailed surfaces, are the multi-level partition of unity and the SLIM surfaces by Ohtake et al ([11], [12]). In both techniques piecewise polynomial surfaces are blended together in order to generate a smooth surface representation. For SLIM surfaces a software based raycasting algorithm was proposed, that simplifies the blending technique to a screen space blending approach (see [12]).

Edelsbrunner use parts of spheres and hyperboloids to represent a C^1 smooth skin surface over a union of spheres in [5]. This surface representation is used mainly for visualizing molecular structures.

Roessl et al use piecewise quadratic splines over a tetrahedral subdivision of a regular grid to approximate a volumetric data set as close to interpolating as possible ([16]). The approach is applied to the visualization of iso surfaces and a software implementation of a raycasting algorithm is proposed.

Raytracing on graphics hardware has been a focus for many researchers in the past few years, from general implementations like Carr et al [4] or Purcell et al [14], who both implemented complete raytracing pipelines for triangular objects, to more specialized algorithms for volumes such as Krueger and Westermanns work in [10] and implementations of acceleration structures like Foley and Sugermans KD-tree implementations [6].

A few approaches have been developed to use modern graphics hardware to efficiently visualize quadric surfaces or subsets of them. One of the first approaches was presented by Gumhold in [7], where ellipsoids are splatted as on screen quads and a ray-ellipsoid

Figure 1: Dragon SLIM model rendered with our system with different shaders applied at roughly 40 fps. From left to right : NPR shading, Reflection mapping, Phong Shading.

intersection test is evaluated for each fragment of the splat. An important contribution was the minimization of the load for the fragment shader by pre-computing all parts of the ray-ellipsoid intersection and normal computation that vary linearly over the splatted quad in the vertex shader. We refer to this optimization as *incremental raycasting*. Klein and Ertl describe an ellipsoid-splatting approach that works with point splats in [9]. This reduces data traffic between CPU and GPU but approximates the ellipsoids not as close as a quad leading to a higher fragment count. In [15], Reina and Ertl present a hardware accelerated raycasting approach that can visualize glyphs, which are composed of spheres and cylinders. Stoll et al use a splatting technique to visualize generalized cylinders by rendering a quad on an approximating silhouette of a segment of the generalized cylinder and interpolating normals and depth values on those in [19]. Wooed et al and Toledo and Levy raycast general quadratic implicit surfaces as we do, but only use an unoptimized non incremental raycasting approach and do not discuss backface culling nor do they study modern quadric based surface representations such as SLIM surfaces ([21], [20]).

Sigg et al just published their work on quadric raycasting in [18], but unlike our work focus on Spheres, Ellipsoids, and Cylinders and visualization of molecular structures, and do not deal with culling and blending of primitives.

3 Overview

Modern GPUs allow us to load custom programs to be executed into two stages of the graphics pipeline, the vertex shader and the fragment shader. The vertex shader is responsible for transforming the incoming vertices into screen space coordinates and setting up further values to be interpolated across the primitive. The fragment shader is executed for each pixel generated when rasterizing the primitive and outputs color and depth value. A current trend in polygonal mesh rendering is to overtesselate surfaces such that several triangles fall into one pixel on the screen. This is not only unnecessary but also leads to aliasing in the shading if the surface is rough. We therefore assume that in a typical rendering application a multiscale technique is used that ensures that in average several pixels are covered by one primitive. Especially for higher order surfaces this makes a lot of sense. Therefore, there are many more fragments which need to be processed than vertices and it is important to move as many calculations as possible to the vertex shader and keep the fragment program short.

Raycasting maps naturally to this separation into per vertex and per fragment calculations. To raycast a single quadric surface we need to define the camera position $\mathbf{e}$, the camera parameters (view direction, fov, resolution, aspect ratio) and the active light sources. For each of the pixels of the camera we need to calculate an intersection of the ray emanating through it and the quadrics.

Unlike traditional raycasting approaches, we are not able to test for an intersection with all quadrics for a given ray at once and so have to fall back to a splatting approach, where each quadric is rendered on its own and the front most intersection is determined with the help of the depth buffer of the graphics cards.

As we are only interested in a small part of the quadric surface, it is possible to calculate a screen space bounding area which defines an area of interest to which we will limit all ray/quadric intersection tests. For SLIM surfaces the area of interest is a screen space quadrilateral and for tetrahedral meshes and bilinear quadrilaterals the front facing triangles of a tetrahedron.

The GPUs vertex shader is used to precalculate per quadric and frame constant values based on the current viewpoint and determine if the quadric can be culled completely. In case of SLIM surfaces we also determine the position of the vertices for the area of interest to tightly encapsulate the screen space projection of the bounding sphere.

The fragment shader calculates the actual intersection point of the current ray with the quadric surface and its depth value, as well as the surface normal at this position. This information will then be used to either calculate the shading with a phong lighting algorithm or be stored in render buffers for further processing with deferred shading.

In Section 4 we will explain the mathematical background of our algorithms, followed by a more detailed explanation of the implementation in Section 5.

4 Theoretical Considerations

There are different quadratic surface representations. The simplest one is a *graph function* $f_G(x,y)$ that specifies the height over a planar domain, where typically some estimate of the tangent space is used. A quadratic graph function is given in the monome basis by a sum $k_{00} + k_{10}x + k_{01}y + k_{11}xy + k_{20}x^2 + k_{02}y^2$.

An *implicit surface* is defined via a function $f_I(x,y,z)$ over the three dimensional space. The surface of the function is defined as the set of all points for which $f_I(x,y,z) = 0$, the interior as all points $f_I(x,y,z) < 0$ and the exterior as all points with $f_I(x,y,z) > 0$. Again

a quadratic implicit surface, which is also called a *quadric*, is given by a sum over all mixed terms in x, y and z of maximum degree two.

The third representation that we will consider is a *bilinear quadrilateral*, which is defined in parametric form with four 3d points $\mathbf{p}_1, \ldots \mathbf{p}_4$:

$$\mathbf{s}(u,v) = (1-v)\left((1-u)\mathbf{p}_1 + u\mathbf{p}_2\right) + v\left((1-u)\mathbf{p}_4 + u\mathbf{p}_3\right). \quad (1)$$

The most general of these three representations, that also subsumes the other two, is the implicit quadratic surface. Therefore we develop our rendering approaches for the implicit representation and map the other representations to the implicit one. The graph function can trivially be mapped to an implicit representation via the equation $z = f_G(x,y)$, i.e. the corresponding implicit representation is simply $f_I(x,y,z) = z - f_G(x,y)$. Section 4.5 describes an algorithm that maps a bilinear quadrilateral to a quadric.

A single quadratic surface is quite limited in its descriptive power. All the applications discussed later on combine several quadratic patches into a complex surface. For this, each patch is clipped with a bounding volume. Two types of bounding volumes are discussed in this work: bounding spheres and bounding tetrahedra.

In the next subsection the basic notation is introduced. Subsection 4.2 derives the expressions necessary to compute ray-quadric intersections. In subsection 4.3 the expressions are simplified by choosing the coordinate origin in the eye point. Finally, we discuss backface culling on a per fragment and per quadric basis in subsection 4.4.4.

4.1 Basic Notation

Three equivalent notations are commonly used for the implicit function f_I of a quadric:

1. with 10 coefficients k_{ijk} of the mixed monomes in $\{x,y,z\}$, where $0 \le i,j,k \le 2$ and $i+j+k \le 2$:

$$\begin{aligned} f_I(x,y,z) &= k_{000} + k_{100}x + k_{010}y + k_{001}z + \\ &\quad k_{110}xy + k_{011}yz + k_{101}zx + \\ &\quad k_{200}x^2 + k_{020}y^2 + k_{002}z^2, \end{aligned}$$

2. with a symmetric 3x3 matrix $\mathbf{A}$, a 3d vector $\vec{b}$ and a constant c:

$$f_I(\mathbf{x}) = \left(\mathbf{x}^t\mathbf{A} + 2\vec{b}^t\right)\mathbf{x} + c, \text{ with } \mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad (2)$$

$$c = k_{000}, \quad \vec{b} = \begin{pmatrix} \frac{1}{2}k_{100} \\ \frac{1}{2}k_{010} \\ \frac{1}{2}k_{001} \end{pmatrix},$$

$$\mathbf{A} = \begin{pmatrix} k_{200}x & \frac{1}{2}k_{110} & \frac{1}{2}k_{101} \\ \frac{1}{2}k_{110} & k_{020} & \frac{1}{2}k_{011} \\ \frac{1}{2}k_{101} & \frac{1}{2}k_{011} & k_{002} \end{pmatrix}, \text{ or} \quad (3)$$

3. with a symmetric 4x4 homogeneous matrix $\tilde{\mathbf{Q}}$:

$$0 = \tilde{\mathbf{x}}^t\tilde{\mathbf{Q}}\tilde{\mathbf{x}}, \quad \text{with} \quad \tilde{\mathbf{Q}} = \begin{pmatrix} \mathbf{A} & \vec{b} \\ \vec{b} & c \end{pmatrix}, \tilde{\mathbf{x}} = \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix}, \quad (4)$$

where the tilde over the symbols denotes the homogeneous representation. Please note that the transpose of a point or vector times another point or vector corresponds to a scalar product. Thus $\tilde{\mathbf{x}}^t\tilde{\mathbf{Q}}\tilde{\mathbf{x}}$ can be interpreted as the scalar product of the vector $\tilde{\mathbf{x}}$ and the vector $\tilde{\mathbf{Q}}\tilde{\mathbf{x}}$. In the following we use the matrix and homogeneous notations for the quadric because of their compactness.

4.2 Ray-Quadric Intersection

For rasterization we have to intersect the ray emanating from the eye point $\mathbf{e}$ in direction of the view vector $\vec{v}$ with the quadric $\tilde{\mathbf{Q}}$. This can be simply done by substituting the parameter form of the ray

$$\mathbf{x}(\lambda) = \mathbf{e} + \lambda\vec{v}$$

into the equation defining the points on the quadric. For the matrix representation and the homogeneous matrix representation this results in

$$\begin{aligned} 0 &= \left((\mathbf{e}+\lambda\vec{v})^t\mathbf{A} + 2\vec{b}^t\right)(\mathbf{e}+\lambda\vec{v}) + c \\ &\quad 0 = (\tilde{\mathbf{e}}+\lambda\tilde{v})^t\tilde{\mathbf{Q}}(\tilde{\mathbf{e}}+\lambda\tilde{v}) \qquad (5) \\ 0 &= \vec{v}^t\mathbf{A}\vec{v}\lambda^2 + 2\left(\mathbf{e}^t\mathbf{A}+\vec{b}^t\right)\vec{v}\lambda + \left(\mathbf{e}^t\mathbf{A}+2\vec{b}^t\right)\mathbf{e} + c \\ &\quad 0 = \tilde{v}^t\tilde{\mathbf{Q}}\tilde{v}\lambda^2 + 2\tilde{\mathbf{e}}^t\tilde{\mathbf{Q}}\tilde{v}\lambda + \tilde{\mathbf{e}}^t\tilde{\mathbf{Q}}\tilde{\mathbf{e}}, \end{aligned}$$

where we exploited the symmetry of the matrices. The second pair of lines is in the standard form for solving a quadratic equation

$$0 = \alpha\lambda^2 + 2\beta\lambda + \gamma, \quad (6)$$

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} \vec{v}^t\mathbf{A}\vec{v} \\ \left(\mathbf{e}^t\mathbf{A}+\vec{b}^t\right)\vec{v} \\ \left(\mathbf{e}^t\mathbf{A}+2\vec{b}^t\right)\mathbf{e}+c \end{pmatrix} \text{ i.e. } \begin{pmatrix} \tilde{v}^t\tilde{\mathbf{Q}}\tilde{v} \\ \tilde{\mathbf{e}}^t\tilde{\mathbf{Q}}\tilde{v} \\ \tilde{\mathbf{e}}^t\tilde{\mathbf{Q}}\tilde{\mathbf{e}} \end{pmatrix},$$

with the solution for λ

$$\lambda_\pm = \frac{-\beta \pm \sqrt{\beta^2 - \alpha\gamma}}{\alpha}. \quad (7)$$

Equation 7 is not the only way of writing the solutions for λ, but it has the nice property that λ_- is exactly the solution where the quadric surface faces the viewer. We will discuss this in section 4.4.2 in further detail. There, we will also derive a different expression for $\lambda_\pm$ resulting in a more efficient implementation in the case where backface culling is turned off.

4.3 Choosing the Origin

To simplify equation 6 we transform the world coordinate system such that the eye point becomes the coordinate origin. As a result we also have to transform the representation of the quadric. Let $\mathbf{y}$ be a point in the transformed coordinate system and $\mathbf{x}$ one in the world coordinate system. Then the transformation can be written as

$$\mathbf{y} = \mathbf{x} - \mathbf{e} \qquad \tilde{\mathbf{y}} = \tilde{\mathbf{x}} - (\tilde{\mathbf{e}} - \tilde{\mathbf{0}}), \quad \text{with} \quad \tilde{\mathbf{0}} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

To find out, what the transformed quadric $\tilde{\mathbf{Q}}'$ looks like, we substitute the inverse transformation in equations 2 and equivalently 4:

$$\begin{aligned} 0 &= \left((\mathbf{y}+\mathbf{e})^t\mathbf{A} + 2\vec{b}^t\right)(\mathbf{y}+\mathbf{e}) + c \\ &= \mathbf{y}^t\left(\mathbf{A}\mathbf{y} + 2(\vec{b}+\mathbf{A}\mathbf{e})\right) + c + \mathbf{e}^t\left(\mathbf{A}\mathbf{e} + 2\vec{b}\right) \end{aligned} \quad (8)$$

Comparing the result in equation 8 with equation 2 we can read off the transformed quadric as

$$\tilde{\mathbf{Q}}' = \begin{pmatrix} \mathbf{A}' & \vec{b}' \\ \vec{b}' & c' \end{pmatrix} = \begin{pmatrix} \mathbf{A} & \vec{b}+\mathbf{A}\mathbf{e} \\ \vec{b}+\mathbf{A}\mathbf{e} & c+\mathbf{e}^t\left(\mathbf{A}\mathbf{e}+2\vec{b}\right) \end{pmatrix}. \quad (9)$$

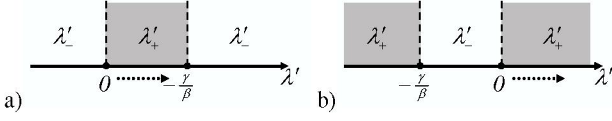

Figure 2: Ranges for the two solutions of $\lambda'_{\mp}$ in the case that a) $\frac{\gamma}{\beta} > 0$, and b) $\frac{\gamma}{\beta} < 0$.

In the transformed coordinates the expressions from equation 6 for the parameters $\{\alpha, \beta, \gamma\}$ simplify to

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} \vec{v}'^t \mathbf{A}' \vec{v}' \\ \vec{b}'^t \vec{v}' \\ c' \end{pmatrix} \text{ i.e. } \begin{pmatrix} \tilde{v}'^t \tilde{\mathbf{Q}}' \tilde{v}' \\ \tilde{\mathbf{0}}^t \tilde{\mathbf{Q}}' \tilde{v}' \\ \tilde{\mathbf{0}}^t \tilde{\mathbf{Q}}' \tilde{\mathbf{0}} \end{pmatrix} \quad (10)$$

Please note that the homogeneous component of the view vector $\tilde{v}$ or $\tilde{v}'$ is zero and not one.

4.4 Backface Culling

4.4.1 Normal Test

To perform backface culling on implicits we first need access to the surface normal. As we defined the exterior of an implicit surface from $f_I(\mathbf{x}) > 0$, the surface normal is proportional to the gradient $\nabla_{\mathbf{x}}$ of $f_I(\mathbf{x})$ with respect to the coordinates of $\mathbf{x}$. Substituting equations 2 and 4 yields

$$\hat{n}(\mathbf{x}) \propto \frac{1}{2}\nabla_{\mathbf{x}}\left(\mathbf{x}^t\mathbf{A} + 2\vec{b}^t\right)\mathbf{x} + c = \mathbf{A}\mathbf{x} + \vec{b} = \tilde{\mathbf{Q}}\tilde{\mathbf{x}}|_{x,y,z}, \quad (11)$$

where the last expression uses the coordinate selection operator "|" to select the first three components of the homogeneous vector $\tilde{\mathbf{Q}}\tilde{\mathbf{x}}$.

We are now in the position to check for a given point $\mathbf{x}$ on the surface, whether the normal points in direction of the viewer or away from it. With the view vector $\vec{v} = \mathbf{x} - \mathbf{e}$ the backface culling test becomes

$$0 < \vec{v}^t \hat{n} \quad \Leftrightarrow \quad 0 < \vec{v}^t \mathbf{A}\mathbf{x} + \vec{v}^t \vec{b} = \tilde{v}^t \tilde{\mathbf{Q}}\tilde{\mathbf{x}}. \quad (12)$$

4.4.2 Per Fragment Backface Culling

In order to restrict our considerations to points on the implicit surface, we use the result from the ray-quadric intersection into the backfacing test of equation 12

$$\begin{aligned} 0 &< \vec{v}^t\mathbf{A}(\mathbf{e} + \lambda\vec{v}) + \vec{v}^t\vec{b} = \vec{v}^t\mathbf{A}\vec{v}\lambda + (\mathbf{e}^T\mathbf{A} + \vec{b}^t)\vec{v} \Leftrightarrow \\ 0 &< \alpha\lambda + \beta. \end{aligned} \quad (13)$$

A very interesting fact can be derived if we plug in for λ the solutions in the form of equation 7:

$$0 < \alpha\lambda + \beta = \pm\sqrt{\beta^2 - \alpha\gamma}. \quad (14)$$

This means that in case of two solutions, there is always a backfacing and a frontfacing solution and the solution with the minus before the square root does always select the front facing solution. If backface culling is turned on, we can simply use the solution with the minus and can neglect the other solution completely. This significantly speeds up rendering.

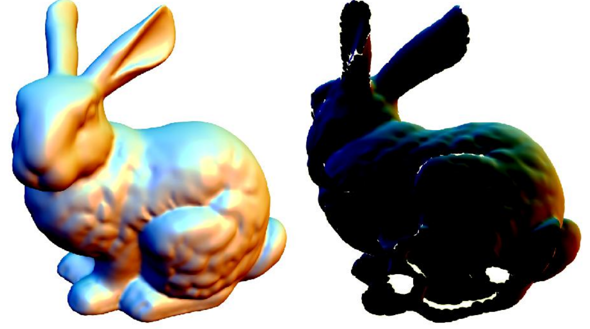

Figure 3: Stanford bunny. Left: Complete SLIM model. Right: Surfaces culled by the per quadric backface culling test.

4.4.3 Without Per Fragment Backface Culling

If backface culling is not wanted, for example in case of visualization of a cylinder, where we want to see also the inside at the ends, both solutions need to be considered. This complicates the intersection test, as we not only have to check for both solutions if they are inside the bounding volume, but also need to determine the valid solution closer to the observer.

Interestingly, one can rewrite the solutions for λ from equation 7 in the following manner:

$$\begin{aligned} \lambda'_{\mp} &= \frac{-\beta \pm \sqrt{\beta^2 - \alpha\gamma}}{\alpha} \cdot \frac{-\beta \mp \sqrt{\beta^2 - \alpha\gamma}}{-\beta \mp \sqrt{\beta^2 - \alpha\gamma}} \\ &= \frac{\beta^2 - (\beta^2 - \alpha\gamma)}{\alpha \cdot (-\beta \mp \sqrt{\beta^2 - \alpha\gamma})} = \frac{\gamma}{-\beta \mp \sqrt{\beta^2 - \alpha\gamma}} \\ &= \frac{1}{-\frac{\beta}{\gamma} \mp \sqrt{\left(\frac{\beta}{\gamma}\right)^2 - \frac{\alpha}{\gamma}}} \end{aligned} \quad (15)$$

For the expressions $\lambda_{\pm}$ in equation 7 the two solutions are symmetric around the location $\lambda = -\frac{\beta}{\alpha}$. On the other hand, the two solutions $\lambda'_{\mp}$ are not placed symmetrically around $-\frac{\gamma}{\beta}$ and Figure 2 shows the ranges of λ'_- and λ'_+ for the two cases $-\frac{\gamma}{\beta} < 0$ and $-\frac{\gamma}{\beta} > 0$. For both cases, λ'_+ is the first solution which is larger than zero. In raycasting terms, λ'_+ always specifies the ray-quadric intersection closer to the eye point. This simplifies the calculations, as one can first check the solution λ'_+, and only needs to check the solution λ'_- if the first solution is outside the bounding volume. A similar simplification of the ray intersection test is not possible with the expression $\lambda_{\pm}$ of equation 7.

4.4.4 Per Quadric Backface Culling

To reduce the load of the fragment shader it is more efficient to also clip quadrics that are backfacing all over the bounding volume, if the user enables backface culling. For a conservative test we have to ensure that no ray from $\mathbf{e}$ in direction of $\vec{v}$ passing through the bounding volume also intersects the quadric in a front facing manner. In the coordinate system where $\mathbf{e} = \mathbf{0}$, we can parameterize the backface test over $\vec{v}'$. A quadric can then be culled if the minimal value of $\vec{v}'^t\hat{n}$ over all possible $\vec{v}'$ is still larger than zero

$$0 < \min_{\vec{v}'}\left(\vec{v}'^t\hat{n}(\vec{v}')\right) = \min_{\vec{v}'}\left(\vec{v}'^t(\mathbf{A}\vec{v}'\lambda + \vec{b}')\right). \quad (16)$$

We are only interested in those locations in view direction for which $\lambda > 0$. Therefore the <-sign is preserved when we multiply with λ

$$0 < \min_{\vec{v}'}\left(\vec{v}'^t\mathbf{A}\vec{v}'\lambda^2 + \vec{v}'^t\vec{b}'\lambda\right).$$

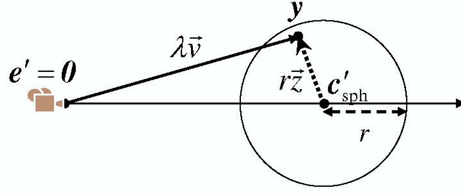

Figure 4: An arbitrary point $\mathbf{y}$ in the bounding sphere can be split into $\mathbf{c}'_{\text{sph}}$ and a vector $r\vec{z}$.

Furthermore, we can restrict ourselves to points on the implicit surface. For these, $\vec{v}'^t \mathbf{A} \vec{v}' \lambda^2 + 2\vec{v}'^t \vec{b}' \lambda + c' = 0$ holds and we can eliminate the quadratic term in λ and $\vec{v}'$

$$0 < \min_{\vec{v}'} \left(-\vec{v}'^t \vec{b}' \lambda - c' \right) = -c' - \max_{\vec{v}'} \left(\vec{v}'^t \vec{b}' \lambda \right). \tag{17}$$

This much simpler test is specialized for the two bounding volumes of consideration in the next two paragraphs.

Backface Culling with Bounding Spheres Firstly, we discuss the case that the bounding volume is a sphere with radius r centered at $\mathbf{c}_{\text{sph}}$, or $\mathbf{c}'_{\text{sph}}$ in the coordinate system with the eye point in the origin.

Independent of λ we consider all points $\mathbf{y}$ inside the bounding sphere that are also on the surface of the quadric. We denote the set of points in the bounding sphere with S. In the local coordinate system, $\mathbf{y}$ coincides with $\vec{v}'\lambda$ and the test in equation 17 is reduced to

$$c' < -\max_{\mathbf{y} \in S} \left(\mathbf{y}^t \vec{b}' \right). \tag{18}$$

We can split the position vector $\mathbf{y}$ into the constant position vector $\mathbf{c}'_{\text{sph}}$ to the sphere center plus a vector $r\vec{z}$ pointing to an arbitrary point inside a sphere of radius r. Let U be all points in a unit sphere. Then we can rewrite the test as

$$c' < -\max_{\vec{z} \in U} \left((\mathbf{c}'_{\text{sph}} + r\vec{z})^t \vec{b}' \right) = -\mathbf{c}'^t_{\text{sph}} \vec{b}' - r \max_{\vec{z} \in U} \left(\vec{z}^t \vec{b}' \right).$$

The maximum is achieved if $\vec{z}$ has maximum length 1 and at the same time is parallel to $\vec{b}'$, what results in the simple per quadric backface culling test for spherical bounding volumes

$$c' < -\mathbf{c}'^t_{\text{sph}} \vec{b}' - r ||\vec{b}'||.$$

Backface Culling with Bounding Tetrahedron Let $(\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3, \mathbf{p}'_4)$ be the four corner points of the tetrahedron in the coordinate system with $\mathbf{e} = \mathbf{0}$. A point in or on the tetrahedron is given by

$$\mathbf{y} = \sum_{i=1}^{4} \sigma_i \mathbf{p}'_i, \quad \text{with} \quad \sum_{i=1}^{4} \sigma_i = 1 \wedge 0 \le \sigma_i \le 1.$$

This can simply be plugged into equation 18:

$$c' < -\max_{\sigma_i} \left(\left(\sum_{i=1}^{4} \sigma_i \mathbf{p}'_i \right)^t \vec{b}' \right).$$

The maximum over the σ_i can only be achieved at the corners of the tetrahedron itself, resulting in the simple to evaluate backface culling test for tetrahedral bounding volumes

$$c' < -\max_{i=1..4} (\mathbf{p}'^t_i \vec{b}'). \tag{19}$$

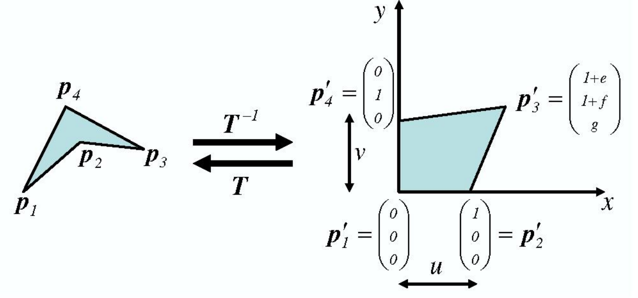

Figure 5: Illustration of affine transformation of an arbitrary bilinear quadrilateral into a coordinate system suitable for conversion to a quadric.

4.5 Conversion of Bilinear Quadrilaterals to Quadrics

The geometry of a bilinear quadrilateral is defined by four cyclically arranged corner points $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4)$. The quadrilateral is completely contained in the convex hull of the four points and therefore in the tetrahedron over the four points $\mathbf{p}_i$. The idea is to render a bilinear quad with a quadric that is clipped by the tetrahedron $(\mathbf{p}_{i=1..4})$. The only question that has to be solved is whether a quad can be represented as a quadric and how to compute the quadric from the four points.

In the following we assume that the quad is *not degenerated* in the sense that the four points do not reduce to a triangle, segment or even a point, since in any of these cases, one can render the corresponding primitive instead. The special case of a planar quad can be rendered with two triangles.

Under the non degeneracy assumption we can define an affine transformation $\mathbf{T}^{-1}$ that brings the quad to the very specific configuration shown in figure 5. The transformed points have the coordinates

$$\begin{pmatrix} \mathbf{p}'_1 & \mathbf{p}'_2 & \mathbf{p}'_3 & \mathbf{p}'_4 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1+e & 0 \\ 0 & 0 & 1+f & 1 \\ 0 & 0 & g & 0 \end{pmatrix}$$

If we plug the transformed points into equation 1, we can describe the surface points $\mathbf{y}$ in transformed space by the simple expression

$$\mathbf{y} = \begin{pmatrix} u(1+ve) \\ v(1+uf) \\ uvg \end{pmatrix}.$$

The next task is to set this equal to the vector $\begin{pmatrix} x' & y' & z' \end{pmatrix}^t$, and to eliminate u and v.

$$\begin{aligned} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} &= \begin{pmatrix} u(1+ve) \\ v(1+uf) \\ uvg \end{pmatrix} \\ \Longrightarrow \quad \begin{pmatrix} x'gv \\ y'gu \end{pmatrix} &= \begin{pmatrix} z'(1+ve) \\ z'(1+uf) \end{pmatrix} \\ \Longrightarrow \quad \begin{pmatrix} (x'g - z'e)v \\ (y'g - z'f)u \end{pmatrix} &= \begin{pmatrix} z' \\ z' \end{pmatrix} \\ \Longrightarrow \quad (x'g - z'e)(y'g - z'f) &= z'g \end{aligned}$$

This yields the coefficients for the quadric and proves the ability of quadrics to represent bilinear quadrilaterals.

$$2g^2 x'y' - 2fgx'z' - 2egy'z' + 2efz'^2 - 2gz' = 0. \tag{20}$$

The quadric in the affinely transformed coordinate system is therefore

$$\mathbf{A}' = \begin{pmatrix} 0 & g^2 & -fg \\ g^2 & 0 & -eg \\ -fg & -eg & 0 \end{pmatrix}, \quad \vec{b}' = \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix}, \quad c' = 0. \tag{21}$$

Finally, we have to transform the quadric back to world coordinates. The corresponding transformation $\mathbf{T}$ is composed of a 3x3-matrix $\mathbf{M}$ and a 3d translation vector $\vec{t}$, i.e. $\mathbf{p} = \mathbf{M}\mathbf{p}' + \vec{t}$, with

$$\begin{aligned} \mathbf{M} &= \left(\mathbf{p}_2 - \mathbf{p}_1 \quad \mathbf{p}_4 - \mathbf{p}_1 \quad (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_4 - \mathbf{p}_1) \right), \\ \vec{t} &= \mathbf{p}_1. \end{aligned} \tag{22}$$

To find out how the quadric transforms under an affine transformation, we plug $\mathbf{p}' = \mathbf{M}^{-1}(\mathbf{p} - \vec{t})$ into the quadric:

$$(\mathbf{p} - \vec{t})^t (\mathbf{M}^{-1})^t \mathbf{A}' \mathbf{M}^{-1} (\mathbf{p} - \vec{t}) + 2\vec{b}'^t \mathbf{M}^{-1} (\mathbf{p} - \vec{t}) + c',$$

and we can read off

$$\mathbf{A} = (\mathbf{M}^{-1})^t \mathbf{A}' \mathbf{M}^{-1} \tag{23}$$

$$\vec{b} = (\mathbf{M}^{-1})^t \vec{b}' - \mathbf{A}\vec{t} \tag{24}$$

$$c = c' + \vec{t}^t (\mathbf{A}\vec{t} - 2(\mathbf{M}^{-1})^t \vec{b}'). \tag{25}$$

The very last thing is to compute e, f and g:

$$\mathbf{p}'_3 = \mathbf{M}^{-1}(\mathbf{p}_3 - \vec{t}) \tag{26}$$

$$e = \mathbf{p}'_3|_x - 1 \tag{27}$$

$$f = \mathbf{p}'_3|_y - 1 \tag{28}$$

$$g = \mathbf{p}'_3|_z \tag{29}$$

Equations 22 to 29 completely define the quadric that represents the bilinear quadrilateral $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4)$.

5 Implementation

All our shaders were implemented in the OpenGL Shading Language. The rendering of a quadric inside a bounding sphere or bounding tetrahedron works as follows: the bounding volume is splatted on the screen by a quad in case of the bounding sphere or the frontfacing triangles of the tetrahedron. In this way all fragments corresponding to rays from the eye point which can intersect the quadric inside the bounding volume are covered. The vertex and fragment shaders are optimized according to the paradigm of incremental raycasting, which means that all computations that lead to temporary results, which vary linearly over the splatted quad or triangles, are precomputed in the vertex shader and only the final evaluation of the temporary results is done in the fragment shader. This is very important as in nearly all applications the rendering is fillrate limited and speeding up the fragment shader significantly increases the frame rate.

From the formulas derived in the previous section we can extract an incremental raycasting approach with the following distribution of the work onto vertex and fragment shader:

Vertex shader: The user passes the homogeneous representation $\tilde{\mathbf{Q}}$ of the quadric in form of a 4x4 attribute matrix to the vertex shader. The bounding sphere is specified in a 4d vector with the radius in the fourth component. A bounding tetrahedron on the other hand is specified by a second 4x4 attribute matrix with the four corner points. Once per frame, a vector valued constant is set to mark the current eye point $\mathbf{e}$. The shader uses equation 9 to transform the quadric to the coordinate system with $\mathbf{e} = \mathbf{0}$. Next, the per quadric backface culling test from equation 18 or 19, respectively, is evaluated. In case of quadric culling the output position is set to a point outside of the current clipping view pyramid, which removes the quadric from further processing. After this, the constant term and the terms linear in the view vector c', $\vec{b}'^t\vec{v}'$, $\vec{v}'$ and $\vec{v}'^t\mathbf{A}'$, which are necessary for the ray-quadric intersection of equation 10 and the computation of the surface normal, are pre-computed and passed as varying attributes to the fragment shader.

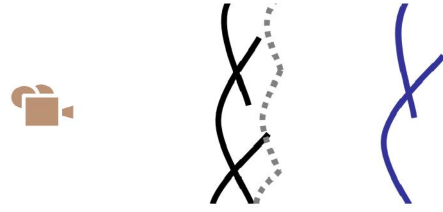

Figure 6: State of the depth buffer for the shading pass. The dotted gray line depicts the depth buffer after the first pass. The blue surfaces are culled, the black surfaces are rendered and blended at the overlap.

Fragment shader: The coefficients γ and β from equation 7 are direct inputs to the fragment shader. α results from a single scalar product of the two interpolated vectors $\mathbf{A}\vec{v}'$ and $\vec{v}'$. Special care has to be taken if $|\alpha|$ becomes small and equation 7 reduces to a linear equation in λ. This case is caught by a separate test and λ is computed to $-\gamma/2\beta$. In both cases, the resulting λ is used to calculate the intersection point $\mathbf{x}$, which is clipped against the bounding volume. The fragment's normal vector is calculated from equation 13, again with the help of the interpolated vectors, and used to calculate the fragments shading or stored in the render buffer in case of the SLIM surface application, where the normals of adjacent quadrics are blended.

In the following we describe the application of our quadric shaders to SLIM surfaces and quadratic implicits over tetrahedralized regular volumes in more detail. The data needed for the visualization of both methods is stored in a display list to eliminate the CPU/GPU bottleneck. Note that vertex buffer objects do not present a noticeable increase in rendering performance in our test situations as we are only handling static geometry. If dynamic modifications of the quadrics need to be performed, vertex buffer objects are the more appropriate choice for storage. This will be addressed in future work.

5.1 Applications

5.1.1 Sparse low degree implicits

The so called SLIM surfaces have been introduced last year in [12] and present an efficient combination of surface reconstruction with implicit surfaces similar to previous work like MPU surfaces [11] and splatting algorithms in the line of [2].

A SLIM surface consists of a set of spheres, each containing a polynomial implicit surface approximating points of a point cloud model to a given error threshold. The surface is visualized using a raytracing approach which generates a weighted sum of the first few intersection points and normals of each quadric along each ray and shading the point using the normalized result. While this "per ray"-blending is only an approximation to an actual weighted blending of the implicit surfaces, [12] show that the method is very efficient and the error negligible. This per ray blending can be implemented in a straightforward way on the graphics hardware using a three pass deferred shading algorithm similar to [2].

A SLIM quadric is splatted using a single quad primitive which is extruded on the vertex shader to tightly encapsulate the quadrics

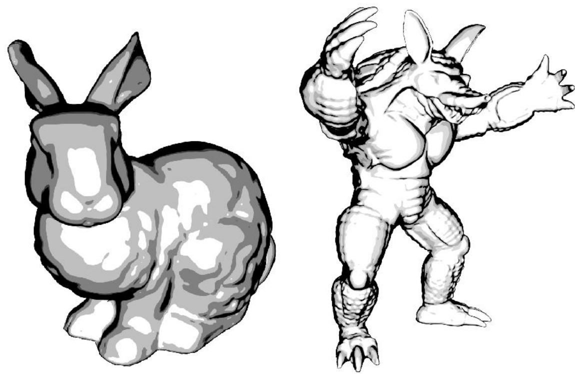

Figure 7: Stanford bunny and armadilo models rendered with a simple non-photorealistic shader in the deferred shading pass.

bounding sphere in screen space coordinates. A texture containing a circular mask can be used to quickly discard fragments in the corner of the quad which lie outside of the bounding spheres screen projection to avoid unnecessary intersection tests.

Note that it is possible to use point sprites instead of quads as base primitive like [2] and thereby reducing load on the vertex processor, but points are limited in their maximum screen space size and will not allow for a close up inspection of the models as holes will appear. With the next generation of the graphics hardware the new programmable geometry shader in the graphics pipeline will make it possible to select the primitives on the fly, allowing for optimizations in this stage.

After calculating the intersection position $\mathbf{x}$ we clip against the bounding sphere by testing if $\|\mathbf{x} - \mathbf{c}_{\text{sph}}\|^2/r^2 < 1$. In accordance to [12] we calculate a weight w for the generated fragment which is 1 at the center of the sphere and 0 on its boundary and smooth in between. [12] chose a piecewise smooth function consisting of a Gaussian in the first half and a quadratic polynomial in the second. This weight can be precalculated into a 1D-texture and used with a lookup table in the fragment shader.

As it is essential that only the quadrics belonging to the surface sheet are blended, the depth buffer needs to be initialized in a *visibility pass* where all quadrics are rendered only to the depth buffer with a small offset away from the eye point as sketched in figure 6.

In the following *rendering pass*, the quadrics are rendered with disabled depth buffer write and enabled additive blending using several floating point render targets, where position and normal values of each fragment are cumulatively multiplied by their weight w in the red, green and blue channels and the weights themselves are added up in the alpha channel. The use of 8 bit render targets introduces quantization artifacts and results in noisy images. Current graphic cards support blending functions on 16 bit floating point buffers, which proves to be sufficient and does not introduce any noticeable visual artifacts.

In the final *shading pass*, one quadrilateral is drawn over the complete viewport. A special fragment shader uses the position and normal values of each of the previous render targets. For each pixel on the screen, the position and normal vectors are divided by their total weight in the alpha channel and the shading is computed. Decoupling the rendering from the shading pass allows an easy implementation of various shading techniques (see Figures 7, 11 and 12) and increases efficiency in scenes with high overdraw and a higher number of light sources.

A simple extension to the SLIM rendering is to include an additional clipping plane in each quadric. This allows the representation of sharp features by cutting parts of the quadric similar to the technique proposed in [13] for point splats (see figure 8).

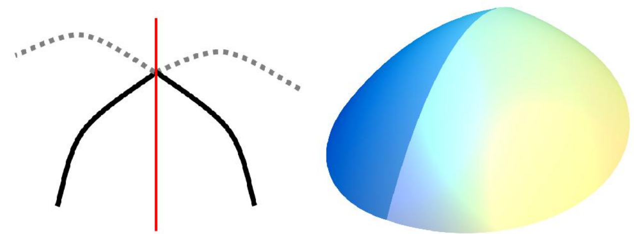

Figure 8: By including an additional clipping plane in the SLIM representation as illustrated on the left (clipping plane in red, clipped quadrics in dotted gray), sharp features can be modeled. The right image shows two paraboloids meeting at a sharp angle.

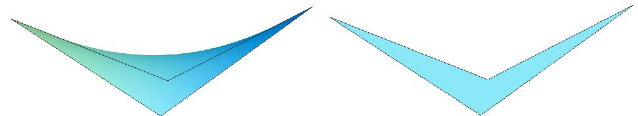

Figure 9: Rendering of bilinear quadrilaterals. Left: with our tetrahedral implementation. Right: with standard opengl quads. Overlaid black lines show the outline of the quad.

5.1.2 Tetrahedral surfaces

Quadratic surfaces contained in volumes are generated by a variety of algorithms. Skin surfaces, introduced in [5], consist of spherical and hyperbolical patches contained in polyhedral clipping volumes. Skin surfaces are mainly used for modeling molecular structures and a large variety of works on triangulation of these surfaces exits, but no efficient raycasting method.

To be able to efficiently clip the quadric surfaces to the containing volume it is necessary to limit ourselves to tetrahedrons, as we cannot pass an arbitrary number of clipping planes to the shader. In a preprocessing step all polyhedrons are split into a set of tetrahedrons, increasing model complexity but allowing for an efficient rendering of each element.

Another algorithm generating similar datasets was introduced in [16], where quadratic patches contained in tetrahedral mesh volumes are used to approximate iso surfaces in volume data. Their method generates huge amounts of data, making the use of an efficient rendering technique crucial.

A third application is the correct rendering of bilinear quadrilaterals. A hardware accelerated interpolation scheme for quadrilaterals is presented in [8], While their method correctly interpolates values across the quadrilateral it does not approximate the geometry correctly in all cases. Using equation 23, it is possible to convert a bilinear quadrilateral to a quadric surface and render it using a tetrahedral volume defined by the 4 corner points (see figure 9).

For the visualization of the isosurfaces, only one pass is necessary since there is no need for any blending. Instead of splatting screen space quads like in the SLIM surfaces, we render the faces of the tetrahedron with enabled backface culling as area of interest. This way each fragment is only rendered once for each quadric.

The fragment clipping of the tetrahedron is achieved by transferring the 4 bounding planes of the tetrahedron to the shader in a 4x4 matrix M_{clip} and checking if $M_{clip} * \mathbf{x} < 0$ for each fragment and discarding it early if this is the case.

Afterwards the fragments normal, position and depth are written to a set of floating point rendering buffers and visualized using a deferred shading pass as in the third pass of the SLIM rendering to reduce the overhead for the shading computations.

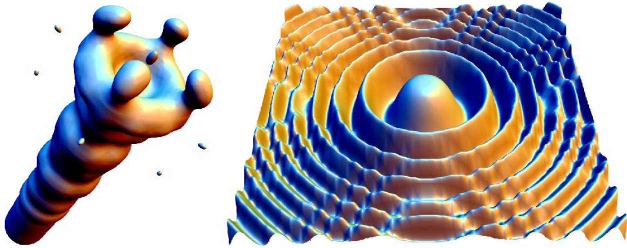

Figure 10: Visualization of the iso-surfaces extracted from the Fuel and Marschner-Lobb volume datasets with [16].

Type	Fill rate	Quadrics/s
Brute force	342.8 MF/s	
No clipping	600.0 MF/s	
SLIM	457.1 MF/s	6.27 M
Tetrahedral	400.0 MF/s	2.16 M

Table 1: Throughput measurements. Fill rate in million fragments per second measured with NVidias nvshaderperf software for a GeForce 7800 GT. Quadric throughput measured in a 2x2 viewport in million quadrics per second.

Figure 11: SLIM renderings of the Stanford Lucy and a scanned statue with 140k respectively 180k quadric primitives.

Model	Tetrahedrals	1024x1280	512x512
Lobster	1700148	0.5	0.5
Sphere	137232	5.8	5.9
Marschner-Lobb	65844	26.3	40.8
Fuel	36082	56.4	71.2

Table 3: Rendering speeds in frames per second for tetrahedral quadric examples. Performance degrades fast with a higher amount of primitives due to memory and bandwidth constraints.

6 Results

In this section we discuss the quality and performance of the approaches we implemented. The use of deferred shading allowed us to easily switch between different shaders for visualization, such as phong shading, environment mapping or non-photo realistic rendering by simply exchanging the shader for the final pass. All measurements were taken on an Opteron 2.4GHz with a GeForce 7800 GT GPU.

Compared to a brute force raycasting implementation as was used in [20], our optimized fragment shader implementation is nearly twice as fast before taking clipping into consideration. Even with activated clipping subroutines our shader performs better (see table 1). Measurements show that our implementation is capable of rendering up to 6.27 M SLIM quadrics and 2.16 M tetrahedral clipped quadrics per second (as long as all data is stored in GPU memory).

Our implementation of the shaders is fast enough to render complex models like in figure 11, which consist of 140k respectively 180k quadrics, in an interactive way with close to 25 FPS when rendering in a full screen 1280x1024 resolution (see table 2). The original software implementation of the algorithm is one order of magnitude slower than our implementation, needing nearly one second to render the scene with a single light source.

Also notable is the fact that, depending on model complexity and screen size, our per quadric culling test can increase the frame rate by a factor up to 2 by reducing the overdraw significantly. The conservative backface culling tests remove quadrics efficiently (see figure 3).

The quality of the resulting rendering is clearly superior to visualization using triangles, as can be seen especially well with the environment shader on the Stanford Buddha in figure 12. The triangle model was simplified to the same error threshold as the SLIM representation (0.01% of the bounding box) and, while being rendered faster, the SLIM surface is of clearly higher quality.

The tetrahedral surface approach is able to render iso surfaces of a size of 65k tetrahedrals interactively with still over 25 FPS in a 1280x1024 resolution. The lower performance compared to the SLIM surfaces results from a higher data rate per primitive. Still even huge iso surfaces like the Lobster CT scan (which consists of nearly 2 million tetrahedrons) can be visualized in half a second when streaming the data over the PCI Express bus to the graphics card.

7 Conclusions

We presented a method to efficiently visualize implicit quadratic surfaces in an interactive way using graphics hardware. We improved rendering performance by pre-computing as much as possible in the vertex shader. Furthermore, we derived formulas for conservative per quadric culling that improve rendering performance by up to a factor of two. We also showed how to reduce the number of tests in the fragment shader by reformulating the expressions for the solution of the quadratic equation. Finally, we derived the formulas necessary to map bilinear quadrilaterals to quadrics.

The results of our experiments strengthen our belief that higher order surfaces can be a very viable alternative to classical triangle rendering or point splatting approaches. On current hardware we can achieve interactive visualization rates for reasonable large models. The next generation of GPUs will bring new possibilities to increase the performance of the algorithms even further, for example by using the geometry shader to select levels of detail.

In future we plan to examine cubic implicit surfaces. We believe that the examination of the solution set of the cubic equation allows for similar reduction of tests in the fragment shader as in the

Model	Quadrics	GPU 1024x1280 culling	GPU 1024x1280 no culling	GPU 512x512 culling	GPU 512x512 no culling	Software 1024x1280	Software 512x512
Statue	183774	24.9	17.6	32.5	24.0	1.02	5.72
Statue	74285	35.5	23.4	68.4	46.8	1.33	6.02
Lucy	140724	27.8	17.4	41.7	29.5	1.08	5.05
Buddha	49203	27.4	15.8	73.8	46.8	1.06	3.70
Bunny	5064	35.6	21.5	>120.0	68.0	1.13	3.22

Table 2: Rendering speeds in frames per second for our SLIM implementation and the original software in fullscreen 1024x1280 resolution and a 512x512 window, with and without culling. The per quadric culling nearly doubles performance in some situations.

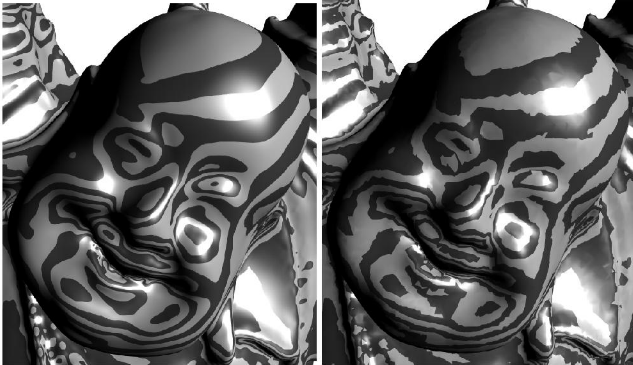

Figure 12: Close up on the head of the Stanford Buddha rendered with a line environment map and phong shading. Left: SLIM reconstruction (11 fps). Right: Simplified mesh with similar geometric error (24 fps). Note the higher quality of the reflection lines and specular highlights on the quadric representation.

quadratic case. We also want to optimize our system for dynamic updates by minimizing the data that has to be transferred from CPU to GPU.

Acknowledgements

This work was supported in part by AIM@SHAPE, a Network of Excellence project (506766) within EU's Sixth Framework Programme. The software SLIM raytracer is courtesy of Yutaka Ohtake. Models are courtesy of the Stanford 3D Scanning Repository and the Volvis repository of GRIS, University of Tübingen.

References

[1] J. Bloomenthal and B. Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[2] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Point-Based Graphics*, pages 17–24, 2005.

[3] M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. In *Proceedings Symposium on Point Based Graphics*, pages 25–32, 2004.

[4] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *Graphics hardware*, pages 37–46, 2002.

[5] H. Edelsbrunner. Deformable smooth surface design. *Discrete & Computational Geometry*, 21(1):87–115, 1999.

[6] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Graphics hardware*, pages 15–22, 2005.

[7] S. Gumhold. Splatting illuminated ellipsoids with depth correction. In *Workshop on Vision, Modeling and Visualization*, pages 245–252, 2003.

[8] K. Hormann and M. Tarini. A quadrilateral rendering primitive. In T. Akenine-Möller and M. McCool, editors, *Graphics Hardware*, pages 7–14, Grenoble, France, Aug. 2004.

[9] T. Klein and T. Ertl. Illustrating Magnetic Field Lines using a Discrete Particle Model. In *Workshop on Vision, Modelling, and Visualization*, pages 387–394, 2004.

[10] J. Krueger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization*, page 38, 2003.

[11] Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, and H.-P. Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

[12] Y. Ohtake, A. G. Belyaev, and M. Alexa. Sparse low-degree implicits. In *Symposium on Geometry Processing*, pages 149–158, 2005.

[13] M. Pauly, R. Keiser, L. P. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3):641–650, 2003.

[14] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics 21*, pages 703–712, 2002.

[15] G. Reina and T. Ertl. Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In K. W. Brodlie and D. J. Duke and K. I. Joy, editor, *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization*, pages 177–182, 2005.

[16] C. Rössl, F. Zeilfelder, G. Nürnberger, and H.-P. Seidel. Reconstruction of volume data with quadratic super splines. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):397–409, 2004.

[17] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343–352, July 2000.

[18] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. Gpu-based ray-casting of quadratic surfaces. In *Eurographics Symposium on Point-Based Graphics*, 2006.

[19] C. Stoll, S. Gumhold, and H.-P. Seidel. Visualization with stylized line primitives. In *IEEE Visualization*, pages 695–702, 2005.

[20] R. Toledo and B. Levy. Extending the graphic pipeline with new gpu-accelerated primitives. Technical report, INRIA, 2004.

[21] A. Wood, B. McCane, and S. King. Ray tracing arbitrary objects on the gpu. In *Image and Vision Computing*, pages 327–332, 2004.

Figure 1: Dragon SLIM model rendered with our system with different shaders applied at roughly 40 fps. From left to right : NPR shading, Reflection mapping, Phong Shading.

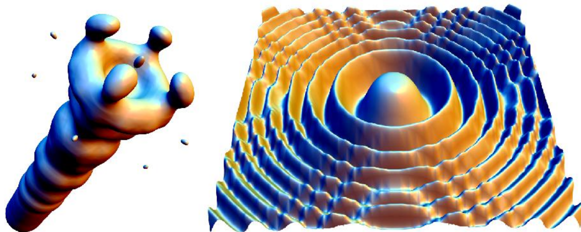

Figure 10: Visualization of the iso-surfaces extracted from the Fuel and Marschner-Lobb volume datasets with (16).

Figure 11: SLIM renderings of the Stanford Lucy and a scanned statue with 140k respectively 180k quadric primitives.