Eurographics Symposium on Point-Based Graphics (2006)
M. Botsch, B. Chen (Editors)

GPU-Based Ray-Casting of Quadratic Surfaces

Christian Sigg Tim Weyrich Mario Botsch Markus Gross

Computer Graphics Laboratory, ETH Zurich

Abstract

Quadratic surfaces are frequently used primitives in geometric modeling and scientific visualization, such as rendering of tensor fields, particles, and molecular structures. While high visual quality can be achieved using sophisticated ray tracing techniques, interactive applications typically use either coarsely tessellated polygonal approximations or pre-rendered depth sprites, thereby trading off visual quality and perspective correctness for higher rendering performance. In contrast, we propose an efficient rendering technique for quadric primitives based on GPU-accelerated splatting. While providing similar performance as point-sprites, our methods provides perspective correctness and superior visual quality using per-pixel ray-casting.

1. Introduction

Due to their compact and simple definition, quadrics are a commonly used class of objects and often serve as basic building blocks for more complex models. They are widely used in geometric modeling applications, like for instance in CAD systems or constructive solid geometry, and are frequently employed for scientific visualization of tensor fields, particle simulations, or molecular structures in biological and medical applications.

Quadrics are known to be well suited for high quality visualizations using all variants of ray-tracing, because computing ray-quadric intersections involves solving quadratic equations only. However, their efficient visualization in interactive applications is still problematic, since current graphics hardware is optimized solely for triangle-based rasterization. Although rendering APIs support quadrics through a separate interface, they are tessellated for hardware accelerated forward mapping. As a consequence, high-quality visualizations require a sufficiently fine tessellation, which in turn causes disproportionate workload at the vertex shader and triangle setup stage.

The same kind of limitations triggered a lot of research in hardware-accelerated point-based rendering during the last years, and we basically follow the same track. The programmability of current GPUs enables the implementation of a direct hardware accelerated rendering of quadric primitives, which — in contrast to the traditional forward mapping — is a mixture between rasterization and ray-casting.

By representing and rendering each quadric primitive as just one vertex, a tight screen-space bounding box can be computed in a vertex shader. For each fragment generated during bounding box rasterization the ray-quadric intersection is computed in a pixel shader.

Both the bounding box and ray intersection can elegantly be stated as the root of a bilinear form corresponding to the implicit definition of the quadric in screen space. Using homogeneous coordinates, our approach naturally supports perspective transformations, which enables the pixel-precise, perspectively correct computation of bounding box and ray intersections. While perspective correctness was also considered in recent point-splatting approaches, existing bounding box computations are either heuristic or computationally very expensive. In contrast, our homogeneous formulation is mathematically elegant, numerically robust, and computationally efficient.

We developed a small library to support hardware accelerated quadratic surfaces within OpenGL. Spheres, ellipsoids, and cylinders are available as additional primitives, which can efficiently be rendered using just one vertex call. Based on this library we implemented a molecule renderer for the common balls-and-sticks and space-filling representations. The visualization features shadow maps and silhouette enhancement, which considerably improves the spatial perception. By this we achieve superior visual quality while still retaining high rendering performance thanks to an efficient deferred shading implementation (cf. Figure 1).

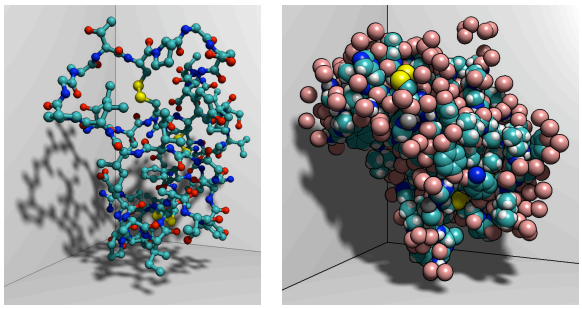

Figure 1: *Molecular representation of a plant seed protein and human insulin rendered with hardware accelerated sphere and cylinder primitives. Our rendering approach is fast enough to employ silhouette outlining and soft shadows for improved spatial perception at interactive rates.*

2. Previous Work

Although quadratic surfaces are featured by many graphics APIs, such as OpenGL utility library [OSW*99], there is currently no special hardware support for these primitives. The tessellation of quadrics into triangles requires to trade off rendering quality against rendering speed, as can be seen in the accompanying video.

Several attempts have been made to provide fast rendering of quadrics. As a special case of quadrics, spheres lend themselves to sprite rendering, using forward mapping of a precomputed image of a sphere. Depth sprites additionally read depth offsets from a texture for per-pixel depth corrections. While providing a better approximation than "flat" sprites, this approach is only valid for orthogonal projections and leads, e.g., to incorrect intersection curves between spheres. Moreover, these approaches do not easily generalize to other types of quadrics.

Gumhold uses programmable pixel shaders to compute ray-ellipsoid intersections for tensor field visualizations [Gum03]. This method is most similar to our technique, but has two important drawbacks: First, the bounding box computation is only correct for orthogonal projections, and thus might clip or cut off ellipsoids for perspective ones. Second, ellipsoids are rendered as object-space quads, which increases the vertex load and leads to rather inefficient rendering compared to our approach.

Similarly, early point-rendering approaches, where ellipses rather than ellipsoids are rendered, also use object-space polygons for splat rendering [RPZ02, PSG04]. However, more efficient methods [BK03, GP03, BHZK05] need just one vertex call per splat and employ programmable shaders for their rasterization: the vertex shader computes a screen-space bounding square, and during bounding box rasterization the pixel shader classifies fragments as belonging to the splat or not.

More recently, point-splatting approaches also focused on perspective correctness. The bounding box computation of Zwicker et al. [ZRB*04] is perspectively correct, but computationally expensive and numerically sensitive due to a required matrix inversion. Moreover, the splats' interiors are perspectively distorted in their method. In contrast, the per-pixel ray-casting of Botsch et al. [BSK04, BHZK05] is perspectively correct, but their bounding box computation is only heuristic and might erroneously clip splats. While slightly incorrect bounding boxes are not a problem for mutually overlapping splats representing a single surface, they cause clearly visible, hence unacceptable, artifacts for quadric-based molecule visualizations (cf. Figure 2).

Our approach adopts homogeneous coordinates for the implicit definition of the quadric. The resulting bilinear form can be projected into screen space by a simple linear transformation, which enables the robust, efficient, and perspectively correct computation of bounding box and ray intersection, since both can be stated as roots of the homogeneous bilinear form.

To demonstrate one possible application of our method, we show high-quality real-time visualization of molecules. Several standard atomic models are used for the study and dissemination of molecular structure and function. Space-filling representations and ball-and-stick models are among the most common ones. The research community uses a number of free and commercial programs to render these models, each having a specific trade-off between quality and rendering speed. Real-time rendering approaches that can cope with large models typically use hardware assisted triangle rasterization [HDS96], whereas high-quality images are produced with ray-tracing [DeL02].

Our method exploits programmable graphics hardware to produce real-time visualization of molecules at a quality comparable to off-line rendering methods. We implemented per-pixel evaluation and lighting of quadrics, and integrated soft shadow mapping and silhouette enhancement in order to emphasize the important aspects of the rendered image and to improve spatial perception.

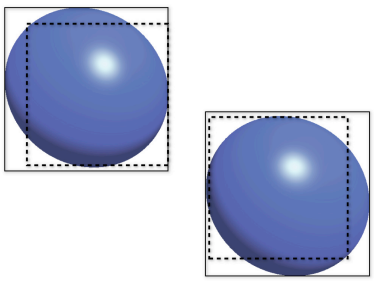

Figure 2: *Previous heuristic bounding box computations might clip quadrics (dashed box), whereas our homogeneous approach provides perspectively correct results (solid box).*

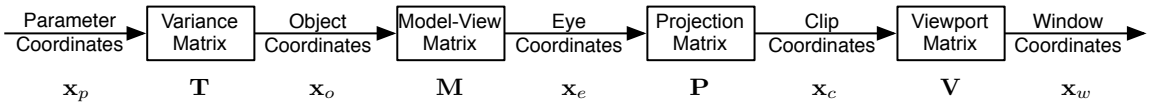

Figure 3: *The OpenGL vertex transformation sequence is preceded by an additional transformation from parameter coordinates to object coordinates. In parameter coordinates, the conic matrix defining the quadratic surface is a diagonal matrix.*

3. Splatting of Quadratic Surfaces

This section explains all steps necessary to render exact per-pixel shaded quadrics under perspective projections. We will first write the implicit definition of the quadratic surface as a bilinear form in homogeneous coordinates, which can be transformed to screen space and is closed even under perspective projections. This insight will allow the derivation of formulas to compute screen-space bounding boxes, solve ray intersections, and evaluate surface normals.

In general, quadratic surfaces are defined as the set of roots of a polynomial of degree two:

$$\begin{aligned} f(x,y,z) &= Ax^2+2Bxy+2Cxz+2Dx+Ey^2 \\ &\quad +2Fyz+Gy+Hz^2+2Iz+J = 0 \end{aligned}$$

The shape of the quadric is solely determined by the coefficients A through J. Using homogeneous coordinates $\mathbf{x} = (x,y,z,1)^T$ the quadric can compactly be written using the bilinear form $\mathbf{x}^T\mathbf{Q}\mathbf{x} = 0$ with the *conic matrix*

$$\mathbf{Q} = \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} .$$

This form is not only much shorter, it is also invariant under perspective projections, since those become linear transformations in homogeneous coordinates. Therefore, the quadric can still be defined in the same form when it is projected to screen space.

3.1. Homogeneous Transformations

In OpenGL terminology, the quadric is defined in object space and then subsequently transformed to window coordinates by the transformation sequence denoted in Figure 3. Each linear transformation in the sequence is determined by a matrix $\mathbf{M}$, expressing the basis of the previous coordinates in the new coordinate system. In order to transform the bilinear form to the new basis, its corresponding conic matrix $\mathbf{Q}$ needs to be multiplied by the inverse transformation matrix from both sides, i.e., $\mathbf{Q}' = \mathbf{M}^{-T}\mathbf{Q}\mathbf{M}^{-1}$. In fact, transforming the conic matrix to a new basis is equivalent to transforming its operands back to the old basis. This allows the bilinear form to be expressed in any coordinate system of the transformation sequence.

For each quadric, there is one distinct basis which can be used to simplify the formulas for bounding boxes and ray intersection. Due to the fact that the conic matrix $\mathbf{Q}$ is symmetric, it can be put into a normalized diagonal form by a basis transformation $\mathbf{T}$:

$$\mathbf{Q} = \mathbf{T}^{-T}\mathbf{D}\mathbf{T}^{-1} \quad \text{with} \quad \mathbf{D} \text{ diagonal}, d_{ii} \in \{0, \pm 1\} \quad (1)$$

The coordinate system where the bilinear form of a quadric has this diagonal, normalized form will be denoted *parameter space*. The transformation matrix $\mathbf{T}$, called *variance matrix*, expresses the basis of the parameter space in object coordinates. The columns contain the axes $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, and center $\mathbf{c}$ of the quadric

$$\mathbf{T} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{c} \\ 0 & 0 & 0 & 1 \end{bmatrix} . \quad (2)$$

In the transformation sequence of Figure 3, the parameter space is placed in front of the object coordinates. Therefore, every quadric can be defined as an affine transformation of one of the basic classes of quadrics in parameter space. The class of the quadratic surface is determined by the normalized diagonal form. Each quadric primitive supported by our library maps to one specific set of diagonal entries. The shape of the quadric in object space is then determined by the variance matrix, which is supplied with the vertex call.

3.2. Bounding Box Computation

The quadric rendering approach needs to comply with the graphics pipeline implemented on graphics cards, which currently support the rasterization of piecewise linear primitives only. Similar to recent point-based rendering techniques, we therefore compute a screen-space bounding box of the quadric. Pixels which are rasterized but are not covered by the quadric are culled during fragment shading by evaluating the correct quadratic function.

Obviously, the fewer pixels are culled in the fragment shader the better. On the other hand, complex bounding polygons should be avoided, because computations cannot be shared across the vertices of the polygon. The point sprite primitive provides the best trade-off between vertex count and pixel overdraw for most quadrics, since they are rendered with one single vertex call, which completely avoids re-computations.

Note that only ellipsoids are naturally bound by their implicit definition. Other quadrics, such as cylinders, additionally have to be clipped by the unit cube in parameter space.

Bounding box computations will first be explained for ellipsoids and later be generalized for other classes of quadrics. We will use the notation of Figure 3 to denote the coordinate systems of vectors as sub-indices and the transformation matrices between them.

To define the parameters of the point sprite, the vertex program needs to compute the center position in clip coordinates and the point sprite radius in window coordinates. A tight axis-aligned bounding box $[b_{x,1}, b_{x,2}] \times [b_{y,1}, b_{y,2}]$ of the projected quadric in clip coordinates is computed first, which is defined by four intersecting half-spaces. Each half-space is given by an equation of the following form:

$$\mathbf{n}_c^T \mathbf{x}_c \leq 0 \tag{3}$$

The half-space to the left of b_x, for instance, is given by $x_c \leq b_x$, corresponding to $\mathbf{n}_c = (1,0,0,-b_x)^T$. For the bounding box to be tight, the bounding plane of the half-space needs to touch the quadric. This condition can easily be enforced in parameter space, where the quadric is defined by the normalized diagonal matrix $\mathbf{D}$. In parameter space, the ellipsoid coincides with the S^2 sphere, and each point on the sphere also corresponds to a normal of a tangent plane. Therefore, the touching condition in parameter space becomes

$$\mathbf{n}_p^T \mathbf{D}\, \mathbf{n}_p = 0 \ . \tag{4}$$

Transforming this condition to clip space is slightly different than transforming the conic equation, since plane normals are transformed with the inverse-transposed matrix:

$$\mathbf{n}_p \ = \ (\mathbf{P} \cdot \mathbf{M} \cdot \mathbf{T})^T \, \mathbf{n}_c \ ,$$

with $\mathbf{T}$, $\mathbf{M}$, and $\mathbf{P}$ as in Figure 3. For the above example of $\mathbf{n}_c = (1,0,0,-b_x)^T$ this gives

$$\mathbf{n}_p \ = \ \mathbf{r}_1 - b_x \mathbf{r}_4 \ ,$$

with $\mathbf{r}_i$ being the i-th row of the compound transformation matrix $\mathbf{P} \cdot \mathbf{M} \cdot \mathbf{T}$. Substitution into the constraint (4) yields a quadratic equation for the horizontal bounding box coordinate b_x:

$$\left(\mathbf{r}_4^T \mathbf{D} \mathbf{r}_4\right) b_x^2 \ - \ 2 \left(\mathbf{r}_1^T \mathbf{D} \mathbf{r}_4\right) b_x \ + \ \left(\mathbf{r}_1^T \mathbf{D} \mathbf{r}_1\right) \ = \ 0 \ . \tag{5}$$

The two solutions of the quadratic equation correspond to the position of the left and right border of the bounding rectangle. For the vertical borders, we simply have to replace $\mathbf{r}_1$ with $\mathbf{r}_2$ in Equation (5). Finally, since the point sprite radius corresponds to the bounding box size in window coordinates, we apply the viewport transformation to the bounding box size and set half of the larger value (width or height) to the point size radius.

The vertex position of the point sprite coincides with the center of the bounding box in clip coordinates. In homogeneous coordinates, the center position can thus be stated as

$$\mathbf{v}_c \ = \ \left(\mathbf{r}_1^T \mathbf{D} \mathbf{r}_4 \, , \, \mathbf{r}_2^T \mathbf{D} \mathbf{r}_4 \, , \, 0 \, , \, \mathbf{r}_4^T \mathbf{D} \mathbf{r}_4\right)^T \ ,$$

where the z-coordinate can be set arbitrarily, since the depth value is overwritten in the fragment program anyway.

Point sprite parameters for cylinders can be computed with a similar approach. The quadrics are culled at the unit cube in parameter space and thus, they are cut off by ellipsoidal caps. A bounding box is computed per elliptic cap, and the point sprite is constructed to cover both bounding boxes. Overall, the vertex program for cylinders is only slightly longer than that for ellipsoids.

3.3. Ray-Quadric Intersection

The rasterization process initiates a fragment shader call for each pixel inside the point sprite. The task of the fragment shader is to kill fragments that are not covered by the quadric and to evaluate a lighting model for all others. In order to combine quadrics with standard primitives, the depth value of the surface needs to be computed per pixel as well. For the corresponding ray intersection problem, we again need to find the roots of a quadratic equation. If the equation has no real solution, the ray is not intersecting the quadric and the fragment can be killed.

We use the unknown depth value z_w of the intersection to parametrize the viewing ray corresponding to the pixel $(x_w, y_w)^T$ in window coordinates:

$$\mathbf{x}_w \ = \ \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \ = \ \mathbf{x}'_w + z_w \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \ ,$$

where $\mathbf{x}'_w$ denotes the front plane position $(x_w, y_w, 0, 1)^T$. Similar to the last section, we transform the ray equation to parameter space:

$$\mathbf{x}_p \ = \ (\mathbf{V} \cdot \mathbf{P} \cdot \mathbf{M} \cdot \mathbf{T})^{-1} \mathbf{x}_w = \mathbf{x}'_p + z_w \mathbf{c}_3 \ , \tag{6}$$

where $\mathbf{x}'_p$ is the front plane position in parameter coordinates and $\mathbf{c}_3$ is the third column of the inverse transformation matrix. Inserting the ray equation into the quadric definition reveals the formula for the intersection depth z_w in window coordinates:

$$\begin{aligned} 0 \ &= \ \left(\mathbf{x}'_p + z_w \mathbf{c}_3\right)^T \mathbf{D} \left(\mathbf{x}'_p + z_w \mathbf{c}_3\right) \\ &= \ \left(\mathbf{c}_3^T \mathbf{D} \mathbf{c}_3\right) z_w^2 + 2 \left(\mathbf{x}_p'^T \mathbf{D} \mathbf{c}_3\right) z_w + \mathbf{x}_p'^T \mathbf{D} \mathbf{x}'_p \ . \end{aligned} \tag{7}$$

If the discriminant of the quadratic equation is negative, there is no intersection and the fragment can be killed. If the polynomial has two real roots, the smaller one corresponds to the closer intersection and is written to the depth buffer.

The evaluation of the lighting model requires the position and normal of the surface in eye space. While the position is computed by transformation from window coordinates, the normal is transformed from parameter space:

$$\mathbf{p}_e \ = \ (\mathbf{V} \cdot \mathbf{P})^{-1} \mathbf{x}_w \ , \tag{8}$$

$$\mathbf{n}_e \ = \ (\mathbf{M} \cdot \mathbf{T})^{-T} \mathbf{n}_p \ . \tag{9}$$

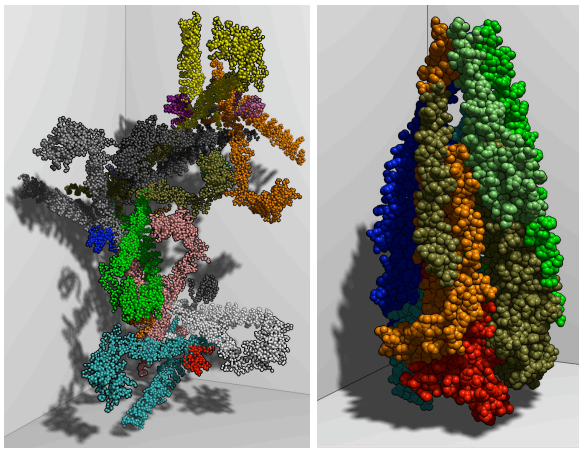

Figure 4: *Improved spatial perception with visual effects. In comparison to pure per-pixel Phong shading, silhouette and crease outlining and soft shadows make it much easier to conceive the structure of the molecule.*

Notice that in order to compute Equations (6), (8), (9), only the quadric-dependent matrix $(\mathbf{M}\cdot\mathbf{T})^{-1}$ needs to passed from the vertex shader to the fragment shader, since the matrix $\mathbf{V}\cdot\mathbf{P}$ is constant. Based on $\mathbf{p}_e$ and $\mathbf{n}_e$, any lighting model can be evaluated on a per-pixel basis.

4. Molecule Rendering

The previous section explained the approach of hardware accelerated quadratic surface rendering. In this section, we present molecule rendering as an exemplary application where this technique proves to be advantageous.

Balls-and-sticks models (Figure 1, left) are highly effective for displaying the covalent structure of a molecule. Each atom is represented by a sphere, and each pair of bonded atoms is connected by a cylinder. This metaphor is particularly useful for organic compounds, because the natural rules of covalent bonding are represented with consistent bond lengths, angles, and geometries. On the other hand, the electron distribution is best captured with space-filling representations (Figure 1, right), where a sphere is placed at each atom center with a radius corresponding to the contact distance between atoms.

Using our hardware-accelerated algorithm for quadric primitives, we can visualize both types of models in real-time even for large models consisting of hundreds of thousands of spheres and cylinders. This leaves enough performance capacities to integrate further techniques for improving depth perception, thereby providing a much better understanding of shape and function of molecules (cf. Figures 4, 7, and the accompanying video). Our multi-pass rendering algorithm is depicted in Figure 5 and explained below.

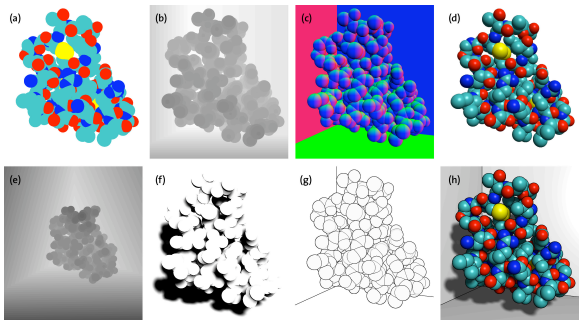

Figure 5: *Different buffers and effects generated for each frame of the deferred rendering pipeline. Three geometry buffers store the parameters of the shading equation: diffuse color (a), fragment depth (b), and surface normal (c). Per-pixel Phong lighting (d) is then evaluated for each pixel using the geometry buffers. A shadow map is rendered from light view (e) and filtered to generate smooth shadow edges (f). Silhouette and crease outlines (g) are extracted by an edge detection filter applied to (b) and (c). Shadows and outlines are composited with the Phong lighting to generate the final image (h).*

Deferred Shading. To avoid expensive evaluations of the shading equation for fragments which are subsequently overdrawn by other fragments closer to the camera, deferred shading [DWS*88] is employed, as also proposed in [BHZK05]. After rendering the scene from the light position to generate the *shadow map* (see below), the scene is rendered from the camera position in the second pass. However, instead of evaluating the shading equation for each fragment, we write fragment depth, diffuse color, and surface normal into a *geometry buffer*. In the final pass, the color of each image pixel is determined by evaluating the shading model using the parameters stored in the geometry buffer.

Soft Shadows. Shadows provide a valuable information about the spatial relationship of groups of atoms that form a molecule. We implemented high-quality soft shadows based on percentage-closer filtering of shadow maps [RSC87]. The small-scale molecule structures lead to highly complex shadow maps, thus requiring 64 shadow map samples to avoid visible noise inside the penumbra region. For better performance, the first 8 samples are used to adaptively determine whether the full 64 samples are required.

Silhouette and Crease Lines. Object and crease outlines are a non-photorealistic visual cue for distinguishing neighboring objects of similar color. Using the geometry buffer, silhouettes and creases can easily be detected by applying a Sobel edge detection filter to depth values and surface normals. The resulting gradient length can be considered as silhouette strength, which is finally used to blend the outline color with the Phong lighting.

Figure	#Spheres	#Cylinders	Direct Shading		Deferred Shading	
			Phong-SM	Phong+SM	Phong-SM	Phong+SM+SL
Fig. 8	99k	198k	37 fps	3.9 fps	34 fps	13 fps
Fig. 7	52k	–	31 fps	2.6 fps	26 fps	15 fps
Fig. 4, right	15k	–	79 fps	5.5 fps	51 fps	22 fps
Fig. 1, right	1712	–	94 fps	7.2 fps	67 fps	22 fps

Table 1: *Performance comparison for standard* direct shading *and* deferred shading *at a viewport resolution of* 1024×768 *on a P4, 2.4GHz, with GeForce 6800GT. While for simple Phong shading without shadows (*Phong-SM*) the direct shading is faster, the deferred shading is clearly superior when adding soft shadows (*+SM*) and silhouette and crease lines (*+SL*).*

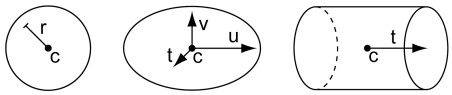

Figure 6: *Supported quadric primitives. Each primitive can be rendered with one single vertex call. The center is specified as the vertex position and the remaining parameters are stored in the texture coordinates. For cylinders, the length of its axis encodes its radius.*

5. Results

We implemented a simple library which extends the OpenGL API to render quadratic surfaces such as spheres, ellipsoids, and cylinders. Figure 6 lists the supported quadric types and the respective parameters defining their shape. Using vertex attributes to specify these parameters enables the use of efficient vertex arrays and vertex buffer objects.

Table 1 compares the rendering performance of different kinds of shading on several molecules of varying complexities. Using standard *direct shading*, the shading equation is evaluated multiple times for a single pixel when one surface is rendered on top of another one. In contrast, deferred shading evaluates the shading equation in a post-processing step exactly once per pixel at the additional cost of storing the shading parameters in offscreen buffers. While for simple shading models the direct approach is faster, the deferred technique clearly pays off for more complex shading including soft shadows and silhouette lines.

We also examined the relative workload of each individual rendering effect in our deferred shading pipeline: the initial shadow map generation is 12%, the per-pixel Phong shading 35%. Filtering the shadow map with 64 samples for smooth shadow edges is clearly the most time consuming element (45%). It is divided in an 8-sample-test to determine whether pixels lie in the penumbra region (19%) and the full 64-sample filtering for those pixels that do so (26%). Including the silhouette and crease lines then adds the missing 8%. Performing full shadow map filtering only in regions that are tested to lie inside a penumbra region results in an 31% rendering speed-up on average for our sample scenes.

6. Conclusions

We presented an algorithm for hardware accelerated rendering of quadratic surfaces. In typical CAD systems and scientific visualizations, high visual quality requires fine surface tessellations. In contrast, our method employs ray-casting on programmable graphics hardware and renders each quadric primitive with a single vertex call.

Transforming the implicit definition of the quadric to screen space is the key component for computing a tight bounding box and ray intersections. In contrast to existing techniques, our entire approach is perspectively correct. Our homogeneous formulation is shown to be robust as well as efficient, and can be generalized to point-based splatting in the future.

To demonstrate the usefulness of our approach, we implemented a molecule renderer which features smooth shadow maps and silhouette outlining to improve the crucial spatial perception of the molecular structures. Although these effects are usually considered too complex for interactive applications, we can achieve real-time frame rates even for large molecules and output resolutions.

References

[BHZK05] BOTSCH M., HORNUNG A., ZWICKER M., KOBBELT L.: High quality surface splatting on today's GPUs. In *Proc. of symposium on Point-Based Graphics 05* (2005), pp. 17–24.

[BK03] BOTSCH M., KOBBELT L.: High-quality point-based rendering on modern GPUs. In *Proc. of Pacific Graphics 03* (2003), pp. 335–343.

[BSK04] BOTSCH M., SPERNAT M., KOBBELT L.: Phong splatting. In *Proc. of symposium on Point-Based Graphics 04* (2004).

[DeL02] DELANO W.: The pymol molecular graphics system. http://www.pymol.org, 2002.

[DWS*88] DEERING M., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *Proc. of ACM SIGGRAPH 88* (1988), pp. 21–30.

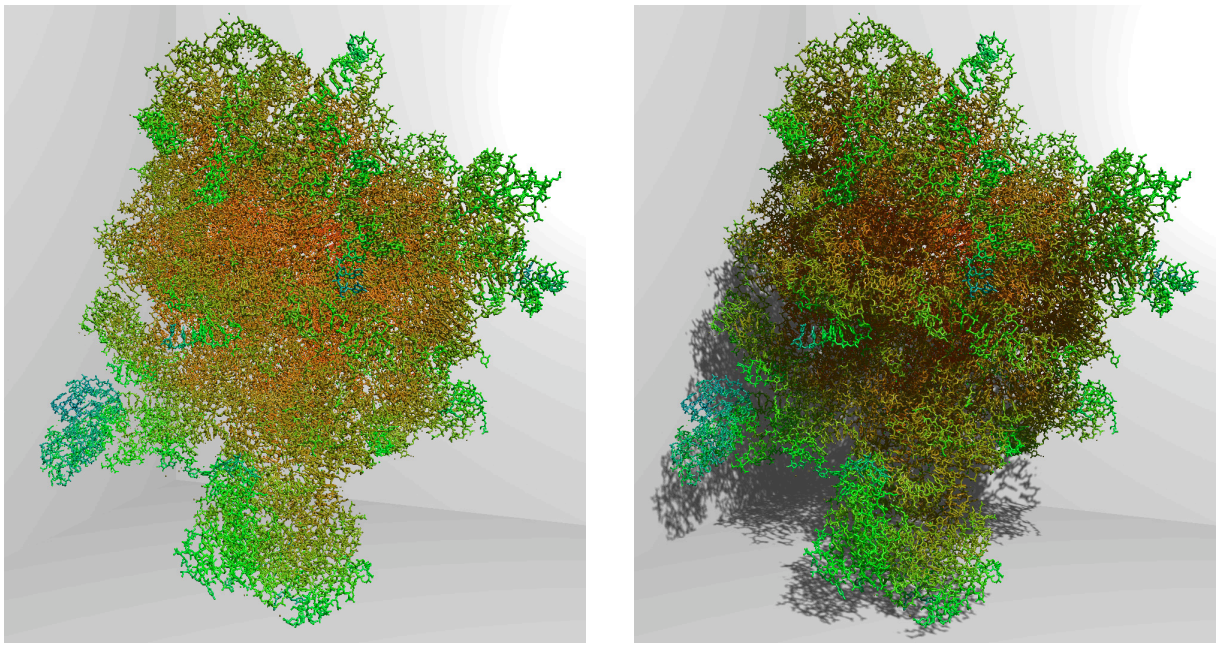

Figure 7: *A ball-and-stick model consisting of 99k spheres and 198k cylinders. Compared to simple shading (left), our superior quality visualization (right) greatly improves the spatial perception, and can still be rendered at 13 fps (1024×768 res.).*

[GP03] GUENNEBAUD G., PAULIN M.: Efficient screen space approach for hardware accelerated surfel rendering. In *Proc. of Vision, Modeling, and Visualization 03* (2003).

[Gum03] GUMHOLD S.: Splatting illuminated ellipsoids with depth correction. In *Proc. VMV 03* (2003), pp. 245–252.

[HDS96] HUMPHREY W., DALKE A., SCHULTEN K.: VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics 14* (1996), 33–38.

[OSW*99] OPENGL ARB, SHREINER D., WOO M., NEIDER J., DAVIS T.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 1999.

[PSG04] PAJAROLA R., SAINZ M., GUIDOTTI P.: Confetti: Object-space point blending and splatting. In *IEEE Transactions on Visualization and Computer Graphics* (2004), vol. 10, pp. 598–608.

[RPZ02] REN L., PFISTER H., ZWICKER M.: Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Proc. of Eurographics 02* (2002), pp. 461–470.

[RSC87] REEVES W. T., SALESIN D., COOK R. L.: Rendering antialiased shadows with depth maps. In *Proc. of ACM SIGGRAPH 87* (1987), pp. 283–291.

[ZRB*04] ZWICKER M., RÄSÄNEN J., BOTSCH M., DACHSBACHER C., PAULY M.: Perspective accurate splatting. In *Proc. of Graphics Interface 04* (2004).

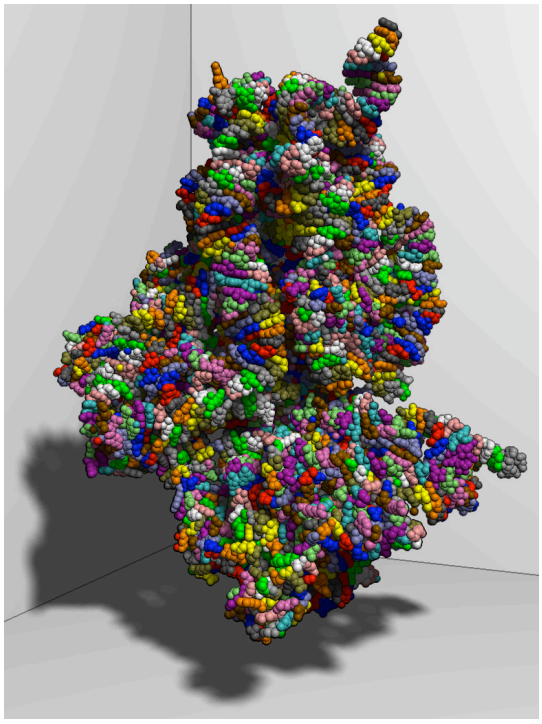

Figure 8: *This complex molecular structure consists of 52k atoms and can be rendered at 15 fps including soft shadow maps and silhouette contouring (1024×768 resolution).*