GPU Based Ray-Casting of Quadratic Surfaces

Nico de Poel (1277219)
Jaap Bresser (1567160)

February 6, 2009

1 Introduction

Quadratic surfaces are a simple, useful representation for certain types of curved objects, that can be used as building blocks for more complex models. Examples of quadratic surfaces are spheres, cylinders and paraboloids. These shapes are difficult to render accurately using traditional rendering methods, because of limitations inherent to those methods.

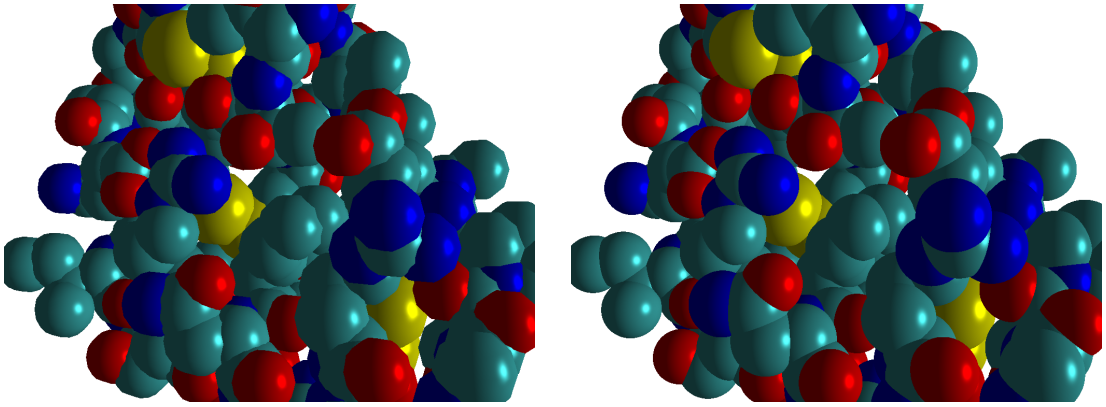
In order to produce high-quality real-time renderings of quadratic surfaces, Christian Sigg et al. have developed a method for ray-casting quadrics using programmable shaders on a GPU. We have studied the paper they have written about this technique [1], modified parts of it and created a custom implementation.

2 Problem definition

The problem is that traditional rendering methods do not excel at visualizing curved objects, such as quadratic surfaces. Three-dimensional objects are commonly represented as a triangle mesh, but for quadrics this means that a tesselated approximation has to be generated. Any triangle mesh consists of hard-angled edges, thus a mesh representation of a quadric can never be completely smooth. Finer tesselation will result in a smoother appearance, but at the cost of a higher vertex processing workload. Despite this, closely zooming in on a triangle mesh will always reveal its angular edges, as Figure 1a illustrates.

Another method for representing spheres is to pre-render them to a 2D depth sprite, which are rendered as a flat billboard in a 3D scene. This method works for spheres because they look the same from all angles, so using a flat texture that stays faced at the camera is enough to convey the idea that the object is a spherical volume. The problem with this method is that textures have limited resolution and so depth sprites will become blurry when closely zoomed in. Also, rendering depth sprites does not take into account deformations of objects caused by perspective projection.

What we need is a method for rendering quadratic surfaces that does not require a coarsely approximated representation of a surface and which produces perspective-correct results, while being efficient enough for rendering at real-time speeds.



(a) Polygonal tesselation (b) Ray-casting

Figure 1: *Different methods for visualizing spheres. Note that the polygonal representation displays angular deformations of the sphere outlines and their specular highlights, as well as jagged edges where spheres intersect.*

3 Solution method

In order to accurately render a quadratic surface, a more mathematical approach is required. In general, quadratic surfaces can be represented by the following polynomial equation:

$$\begin{aligned} f(x,y,z) &= Ax^2 + 2Bxy + 2Cxz + 2Dx + Ey^2 \\ &\quad +2Fyz + 2Gy + Hz^2 + 2Iz + J \\ &= 0 \end{aligned} \tag{1}$$

The coefficients A through J determine the shape, size and orientation of a quadric.

It is possible to define a ray which travels from the viewer's eye through a pixel on the image plane into the 3D scene. By calculating at which point this ray satisfies Equation 1 (i.e. where the ray intersects the quadric), we can determine the point on the surface that corresponds to the pixel on the screen. This ray-casting approach results in a per-pixel accurate representation of the quadratic surface (Fig. 1b).

The challenges in implemented such a method on the GPU are calculating an accurate screen-space bounding box of the quadric, and efficiently computing the ray-quadric intersection for each fragment. Sigg et al. have come up with solutions for these challenges, which they have described in their paper.

Sigg approaches the problem by first representing Equation 1 in a matrix form, which can be separated into a normalized diagonal matrix and a so-called *variance matrix*. The normalized diagonal matrix ($\mathbf{D}$) represents the quadratic surface in its most basic form, without any scaling, rotation or translation. The coordinate system where the quadric has this normalized form is dubbed *parameter space*. With this matrix $\mathbf{D}$, the quadratic surface can be defined as the set of points $\vec{x}$ that satisfy the following equation:

$$\vec{x}^T \mathbf{D} \vec{x} = 0 \tag{2}$$

The variance matrix ($\mathbf{T}$) defines the transformation of the quadric from parameter space to

its specialized form in object space. This essentially expands the rendering pipeline with an additional transformation, preceding the object transformations.

The advantage of this method is that coordinates from any coordinate space can be transformed into parameter space, where computations with the quadratic surface can be done using the simpler normalized diagonal form. For computation of a sphere's bounding box, advantage is taken of the fact that a point on the surface of a normalized sphere equals the normal of the surface at that point. Concerning the ray-quadric intersection, once the ray has been transformed into parameter space, a simple quadratic equation can be constructed that, when solved, yields the intersection point.

For our own implementation, we had to slightly alter Sigg's method.

Ray-quadric intersection

Sigg defines the ray-quadric intersection point as a window-space vector $\vec{x_w} = (x_w, y_w, z_w, 1)$, where x_w and y_w are the known fragment window coordinates, and z_w is the unknown depth value that we need to solve for. By transforming this window-space vector to parameter space, a quadratic equation can be derived with z_w as the unknown variable. Once z_w has been determined, its value can be further used to determine the position and normal of the intersection point.

In practice however, implementing Sigg's method turned out to be problematic and more importantly, difficult to reason about. Sigg's paper leaves some crucial details unspoken and from his implementation we concluded that some very unintuitive steps had to be taken to find a correct solution.

To counter this problem, we employed an alternative, more intuitive way to represent the viewing ray and derived a quadratic solution from this, similar to Sigg's method. This made it easier to implement the ray casting and to properly compute all the desired result values. The only downside to our method is that it allows less precomputation to be done in the vertex shader than Sigg's method, so it might be slightly less efficient.

Instead of defining a viewing ray as Sigg does, we define the ray as linear interpolation from the eye through the image plane:

$$\begin{aligned} \vec{p} &= (1-\mu)\vec{e} + \mu\vec{f} \\ &= \vec{e} + \mu(\vec{f} - \vec{e}) \end{aligned}$$

Where $\vec{e}$ is the position of the viewer's eye, $\vec{f}$ is the position of the fragment on the image plane, and $\vec{p}$ is the intersection point of the ray with the quadratic surface. μ in an interpolation factor that dictates how far along the viewing ray the intersection point lies. It is this value that we will want to find.

We can simplify this equation by introducing a ray vector $\vec{r} = \vec{f} - \vec{e}$, which is the direction from the eye to the fragment:

$$\vec{p} = \vec{e} + \mu\vec{r} \tag{3}$$

Note that this equation holds in all the different vector spaces, and that the value of μ has the same meaning in every vector space.

The position of the fragment in eye-space ($\vec{f_e}$) can be determined by performing an inverse projection on the fragment's windows coordinates. Its corresponding z-coordinate is the known distance from the image plane to the camera center. The eye position in eye-space ($\vec{e_e}$) is constant and known; $\vec{e_e} = (0, 0, 0)^T$ by definition. This makes the ray equation even simpler in eye-space:

$$\begin{aligned} \vec{p_e} &= \vec{e_e} + \mu(\vec{f_e} - \vec{e_e}) \\ &= \mu\vec{f_e} \end{aligned} \tag{4}$$

Both the eye position and the fragment position can be transformed to parameter-space by multiplying them with the inverse variance-modelview transformation matrix $(\mathbf{M} \cdot \mathbf{T})^{-1}$. From these positions $\vec{e_p}$ and $\vec{f_p}$ a parameter-space ray $\vec{r_p}$ can be composed, which conforms to equation 3.

We also know that the intersection point in parameter-space ($\vec{p_p}$) lies on the quadratic surface, and thus has to satisfy the following equation (according to Equation 2):

$$\vec{p_p}^T \mathbf{D} \vec{p_p} = 0$$

Substituting p_p with its ray equivalent from Equation 3, we get:

$$\begin{aligned} 0 &= (\vec{e_p} + \mu\vec{r_p})^T \, \mathbf{D} \, (\vec{e_p} + \mu\vec{r_p}) \\ &= (\vec{r_p}^T \mathbf{D} \vec{r_p})\mu^2 + 2(\vec{e_p}^T \mathbf{D} \vec{r_p})\mu + (\vec{e_p}^T \mathbf{D} \vec{e_p}) \end{aligned}$$

This is a straightforward quadratic equation with μ as the only unknown variable. It can be solved easily with the standard quadratic formula.

As with any quadratic equation, there can 0, 1 or 2 solutions for μ. In case of zero solutions, the ray does not hit the quadratic equation, so it can be discarded. When there is only one solution, the ray grazes an edge of the quadratic surface. In most cases, there will be two solutions for the equation; one intersection with the front of the surface and one with the back. The smaller of the two values for μ corresponds to the front intersection, the larger one to the back. It is only the smallest value that we will want to use, because typically the front surface occludes the back. If the front surface is behind the image plane (e.g. the camera is inside the sphere), we could be able to see the back surface, but then we can also assume that back-face culling should be applied, and the ray can be discarded altogether. This reduces the choice between the two solutions to simply computing the smaller solution only.

The resulting value of μ can be inserted directly into equation 4 to obtain the surface point in eye-space.

Since spheres are normalized in parameter-space, a surface point's position equals the surface normal at that position in parameter-space. With μ, the parameter-space surface point and thus the surface normal can be determined. Transforming this normal using the inverse transpose of the variance-modelview matrix $(\mathbf{M} \cdot \mathbf{T})^{-T}$ yields the surface normal in eye-space.

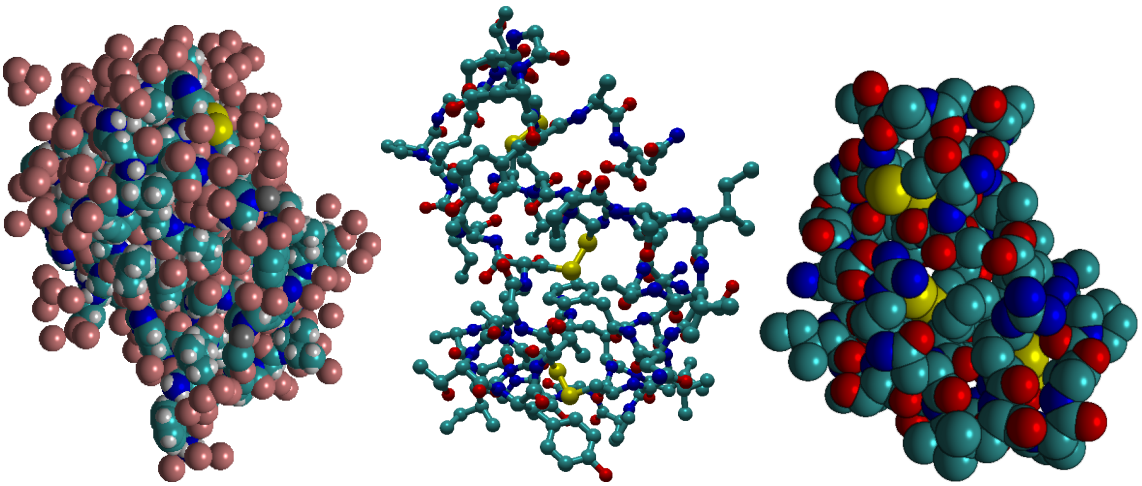
With both the surface position and normal known in eye-space, it is possible to evaluate a per-pixel shading model, such as the Phong shading model.

4 Implementation

Together with his paper, Sigg has written a small library implementing ray-casting of spheres and cylinders, which is available from his web site [2]. The shader files included with this implementation have been written using the arbvp1/fp1 assembly-like instruction sets and are heavily optimized. This makes them fairly difficult to decipher, limiting their use as reference for our own implementation.

Our implementation was done using Nvidia's high-level GPU programming language Cg. Aside from the deviations we described in the previous section, we have largely followed the methods from Sigg's paper. Occasionally, Sigg's implementation has been used as reference and support.

We have implemented nearly all the features mentioned in Sigg's paper. This includes the bounding box computation and ray-casting of both spheres and cylinders (Figs. 2a, 2b). Using positions and normals obtained by ray-casting, a Phong shading term is evaluated for each fragment. Deferred shading has been implemented, as well as rendering of silhouette and crease outlines (Fig. 2c).



(a) Spheres only (b) Spheres and cylinders (c) Silhouette outlines

Figure 2: *Ray-casted renderings of several data sets with varying effects*

In his implementation, Sigg also applies soft shadow mapping to the models. Implementing shadow mapping takes a considerable amount of time and is not the purpose of this exercise, so this step has been omitted in our implementation.

Deferred shading

In addition to standard direct shading, a rendering path that uses deferred shading has also been implemented. It works pretty much like any implementation of deferred shading, with one exception: instead of storing a fragment's depth value in the geometry buffer, its value of the interpolation factor μ is stored instead. This allows the eye-space surface position to be computed in the same way as it is for direct shading.

Silhouette lines are added as an option to improve the distinction of each quadric, using the geometry buffers stored for deferred shading. It is implemented by applying a Sobel edge detection filter to both the normals and the depth values. Although we only store the value of μ in the geometry buffers, it can be used as an approximation of the depth value and provides a good enough gradient for generating silhouette lines. Applying edge detection to the normals causes intersections between quadrics, or creases, to also be marked with an outline. The strength of these two types of outlines can be tuned through several parameters in the deferred shading fragment shader.

Optimization

Both the vertex shaders and the fragment shaders have been optimized to decrease the number of instructions per shader program, in order to increase their performance.

One interesting optimization for both shaders, inspired by Sigg's implementation, involves some modifications to the quadratic formula, to essentially perform several multiplications and divisions in a single step.

The most common notation of the quadratic formula is as follows:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We can define the following helper variables:

$$\begin{aligned} b' &= \frac{b}{2a} \\ c' &= \frac{c}{a} \end{aligned}$$

These values can be computed on the GPU with a single instruction by dividing a temporary vector $(\frac{b}{2}, c)^T$ by a.

Then:

$$\begin{aligned} b'^2 - c' &= \frac{b^2}{4a^2} - \frac{c}{a} \\ &= \frac{b^2}{4a^2} - \frac{4ac}{4a^2} \\ &= \frac{b^2 - 4ac}{4a^2} \end{aligned}$$

This is enough to determine if the discriminant is positive, negative or zero and so whether a fragment should be discarded or not.

The rest of the quadratic formula can be solved as follows:

$$\begin{aligned} \sqrt{b'^2 - c'} &= \sqrt{\frac{b^2 - 4ac}{4a^2}} \\ &= \frac{\sqrt{b^2 - 4ac}}{2a} \end{aligned}$$

The entire quadratic formula can thus be written as:

$$\begin{aligned} x &= -b' \pm \sqrt{b'^2 - c'} \\ &= -\frac{b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a} \end{aligned}$$

Simplifying the quadratic formula to this form saves a significant number of multiplications, thus reducing the number of instructions necessary to compute its solutions.

Cylinders

Cylinders differ from spheres insomuch that they do not look the same from all angles and they do not have a bounded volume. These differences are also reflected in their respective shaders.

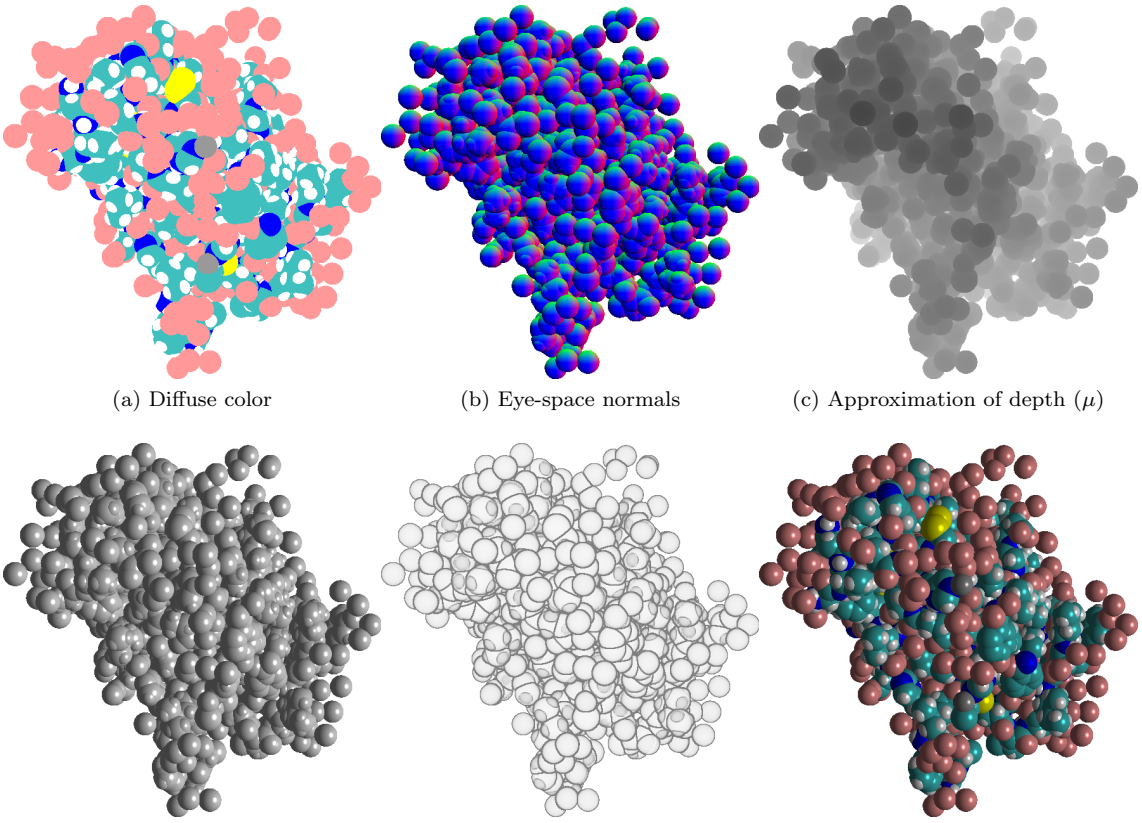
The ray-casting portion done in the fragment shader is mostly similar for cylinders. The only differences are that the surface normal does not equal the surface position in parameter space, but instead the normal equals the parameter-space position with its z-coordinate set to 0. Also, a cylinder needs to be capped, so an additional test is done that discards fragments outside the parameter-space unit cube.

The variance matrix generated in the vertex shader is very simple for spheres, because it involves only scaling and translation; rotation is ignored because spheres appear the same from all angles. For cylinders on the other hand, rotation is an important factor, making the variance matrix and its inverse that much more complex to compute.

The bounding box computation for cylinders is also very different. Because a cylinder is an infinite volume, it needs to be capped by the unit cube in parameter space to be useful for rendering. A cylinder's bounding box is computed by taking the union of the bounding boxes of the two end caps.

Currently, the vertex shader for cylinders is not optimized at all. At the time of writing, the vertex shader compiles to 166 instructions, compared to 35 instructions for spheres. Also, the bounding box computation has not been thoroughly tested and might not be completely correct.

Because of the limitations of the cylinder implementation, its inclusion is considered to only be a proof of concept. Cylinders have not been taken into account in the benchmarks.



(a) Diffuse color (b) Eye-space normals (c) Approximation of depth (μ)

(d) Phong shading (e) Silhouette outlines (f) Final composite image

Figure 3: *Buffers and effects generated during rendering of a deferred shaded image with silhouette outlines*

5 Results

The quality of the images rendered by our custom quadric ray-caster closely match those of Sigg's paper (Fig. 3). On first impression, the performance is very good; on a mid-range GeForce 8600 GT is fluid enough in most cases to be considered real-time.

Performance

Since Sigg uses a different testing system, and our implementation differs in several ways (i.e. no shadow maps, differences in efficiency, different scene setup), our performance numbers can not be directly compared. However, we might be able to see similar patterns in the figures.

All benchmarks are performed at a resolution of 1024×768. Our default testing system is an Intel Core 2 Duo E6550 (2.33 GHz) with 2 GB RAM and a GeForce 8600 GT graphics card.

The data sets used for the tests are the same as those used by Sigg. Some sets represent the same molecule structures, but with different sized spheres. A list of all these data sets, along with the abbreviations used in this section, is available in Appendix A.

Our benchmarking method is as follows. The camera is positioned at one of three different distances from the model. The model rotates at a constant speed, while the rendered frames are counted. After ten seconds, the benchmarks stops and the average number of frames per second is reported.

Cylinders are not taken along in the benchmarks, because they are not properly optimized and would therefore skew the results.

Sphere count

First, we see how the number of spheres in a model affects the performance. This benchmark was run on the default system, with the camera at 5 units distance from the center of each data set. The three different shading types (direct shading, normal deferred shading, deferred shading with silhouette lines) have all been tested. The results of the benchmarks are shown in Figure 4.

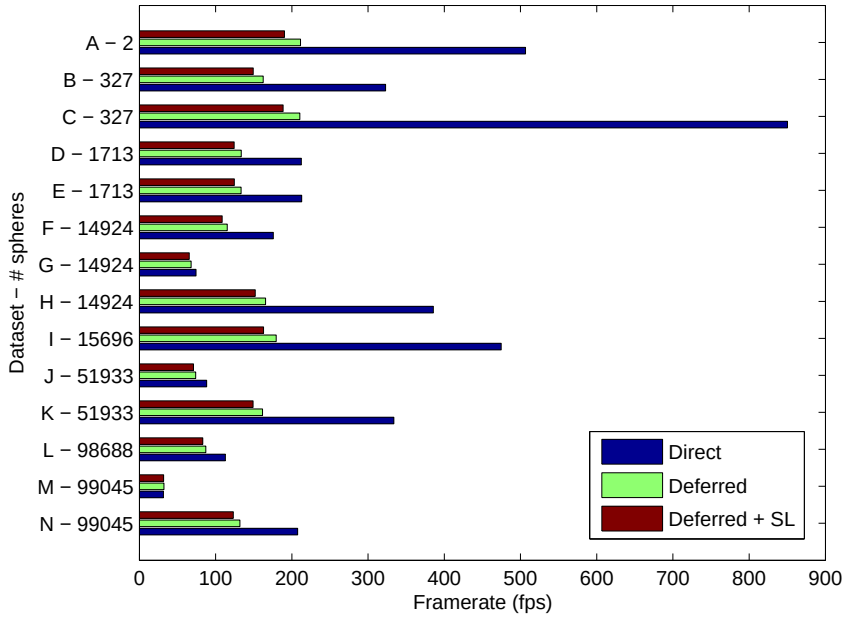


Figure 4: *Performance impact from the number of spheres*

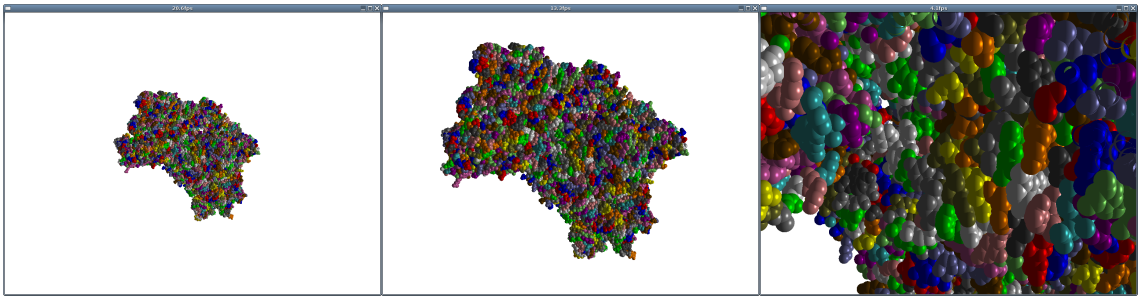
Although a larger number of spheres generally does decrease the performance, there are also cases where a small number of spheres has worse performance than a larger number of spheres (e.g. data set G vs. data set K). It seems there are factors other than the raw number of spheres that strongly affect the performance. This was also notable in Sigg's performance results.

What's also directly visible is that deferred shading is generally slower than direct shading.

Apparantly, the memory bandwidth requirements for deferred shading outweigh the lower number of shading calculations that it yields. In Sigg's paper, deferred shading was only faster than direct shading when shadow mapping was applied. This is something we can not test with our implementation, but we might be able to create a situation where the balance between shader performance and memory bandwidth shifts.

Camera distance

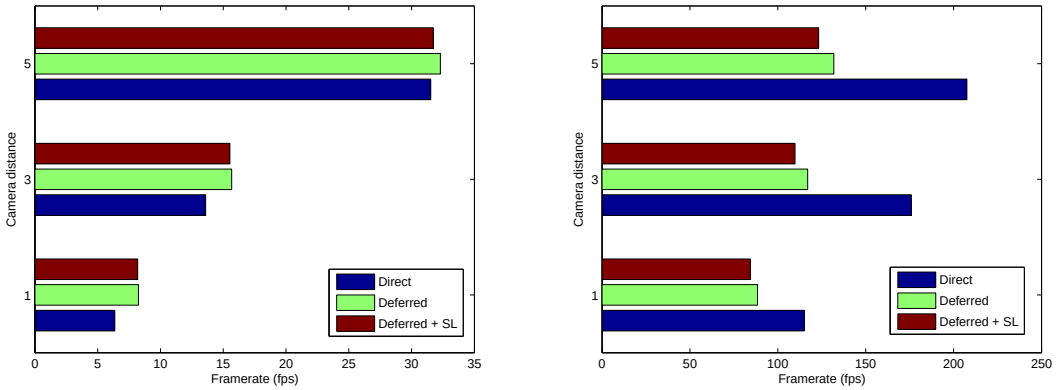
During basic testing, it became clear that the proximity of the spheres to the camera was crucial in the performance of the renderer. This is visible in the screenshots from Figure 5.



(a) Distance = 5, fps = 20.6 (b) Distance = 3, fps = 13.3 (c) Distance = 1, fps = 4.1

Figure 5: *Camera position affecting performance on a GeForce 8400 GS*

To verify this observation, we have performed several tests on two dense sets of spheres, with varying camera distances. The results of these tests are shown in Figure 6.



(a) Dataset M (b) Dataset N

Figure 6: *Framerates at different distances of the camera*

Clearly, having the camera closer to the spheres dramatically decreases performance. This indicates that the sphere count is not the main factor in determining the rendering speed. Rather,

the number of fragments covered by each sphere, as well as the number of discarded (i.e. wasted) fragments affects performance far stronger.

Also, with data set M at a distance of 1 unit (many large spheres fill up the entire screen), deferred shading manages to overtake direct shading in performance. This suggests that indeed deferred shading can be faster when its memory bandwidth requirements are outweighed by the workload of the shaders. Nevertheless, deferred shading is only faster when framerates are very low, and then the performance advantage is small.

Graphics card comparison

Finally, we have tested rendering performance among a selection of graphics cards. This was done using the large and heavy data set N, with the camera distance set to 5 units. The test results are shown in Figure 7.

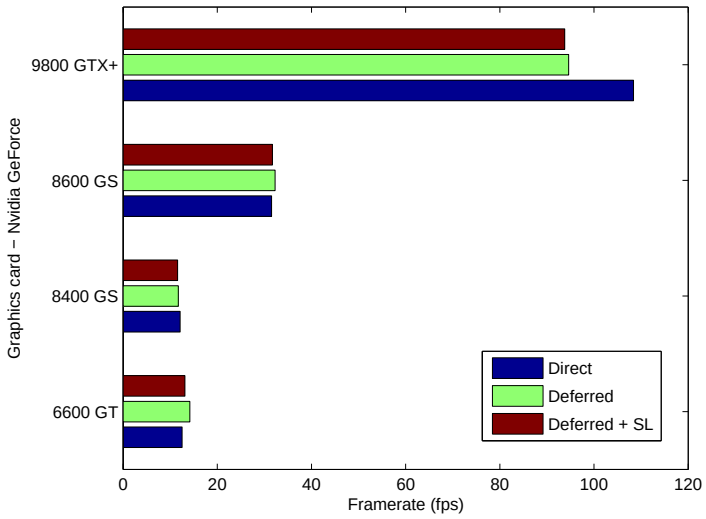


Figure 7: *Performance among different graphics cards*

There is a clear performance improvement from one generation of graphics card to the next, so the ray-casting method scales well on faster hardware. Direct shading is still the winner on the GeForce 9800 GTX+, while deferred shading is a much closer match on the other cards.

Perspective correctness

According to Sigg's paper, both the bounding box computation and the ray casting methods should produce perspective correct results. We have tested this by rendering a set of spheres using an extreme perspective (field of vision set to 120 °), as shown in Figure 8. It is clearly visible that the spheres farthest from the image center are strongly deformed by this perspective and that each sphere's bounding box has been properly scaled to deal with this deformation (i.e. none of the spheres are cropped). This confirms Sigg's claims.

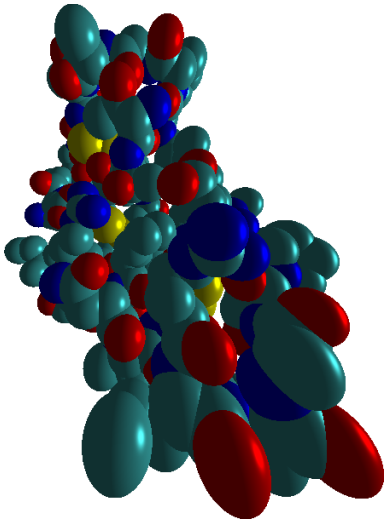


Figure 8: *Spheres deformed by extreme perspective*

Depth perception

The main reason Sigg added shadow maps to his quadric rendering method was to increase depth perception. We can not verify this with our own implementation, but it is clear from Sigg's screenshots that self-shadowing greatly improves the ability to see the structure of a model.

Our own implementation shows that indeed the depth perception on still images with only Phong shading is not outstanding. However, Sigg failed to mention that seeing a model in motion also allows one to get a better grasp on its internal structure, which is confirmed by our own observations.

6 Conclusion

This is the conclusion. This either did or did not go well.

References

[1] Christian Sigg, Tim Weyrich, Mario Botsch, Markus Gross: *GPU-Based Ray-Casting of Quadratic Surfaces*; Proceedings of Eurographics Symposium on Point-Based Graphics 2006;

[2] Christian Sigg's Web Page (http://graphics.ethz.ch/people/archive/siggc)

A Dataset labels

Label	Filename	# Spheres
A	handle.sql	2
C	1crn_vdw.sql	327
B	1crn_cpk.sql	327
D	1ben_vdw.sql	1713
E	1ben_vdw_white.sql	1713
F	2bg9_cpk.sql	14924
G	2bg9_vdw.sql	14924
H	2bg9_vdw_index.sql	14924
I	1xi5_vwd.sql	15696
J	1j5e_cpk.sql	51933
K	1j5e_vdw.sql	51933
L	1kqs_cpk.sql	98688
N	1vqm_vdw.sql	99045
M	1vqm_lic.sql	99045

B Benchmark results

Nvidia GeForce 6600 GT with camera distance 1

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	118.78	70.23	48.48
1crn_cpk.sql	327	124.75	79.95	54.04
1crn_vdw.sql	327	17.17	18.60	16.95
1ben_vdw.sql	1713	7.13	8.65	8.15
1ben_vdw_white.sql	1713	7.34	8.62	8.17
2bg9_cpk.sql	14924	36.87	35.72	29.57
2bg9_vdw.sql	14924	3.79	4.28	4.25
2bg9_vdw_index.sql	14924	6.95	8.25	7.83
1xi5_vwd.sql	15696	50.24	45.20	36.25
1j5e_cpk.sql	51933	40.14	38.62	31.77
1j5e_vdw.sql	51933	3.29	3.95	3.92
1kqs_cpk.sql	98688	10.27	12.04	11.35
1vqm_lic.sql	99045	41.85	38.14	32.55
1vqm_vdw.sql	99045	2.04	2.46	2.44

Nvidia GeForce 6600 GT with camera distance 3

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	114.25	77.17	52.66
1crn_cpk.sql	327	294.27	114.33	67.42
1crn_vdw.sql	327	47.42	43.87	35.58
1ben_vdw.sql	1713	29.90	30.64	26.38
1ben_vdw_white.sql	1713	29.93	30.71	26.24
2bg9_cpk.sql	14924	84.05	63.70	45.88
2bg9_vdw.sql	14924	10.25	11.78	11.14
2bg9_vdw_index.sql	14924	28.65	29.66	25.46
1xi5_vwd.sql	15696	122.08	79.56	53.49
1j5e_cpk.sql	51933	118.75	80.40	54.23
1j5e_vdw.sql	51933	14.83	16.67	15.28
1kqs_cpk.sql	98688	23.20	24.53	21.53
1vqm_lic.sql	99045	69.92	58.77	43.32
1vqm_vdw.sql	99045	4.98	5.99	5.79

Nvidia GeForce 6600 GT with camera distance 5

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	187.62	97.88	62.49
1crn_cpk.sql	327	460.40	128.25	72.26
1crn_vdw.sql	327	118.83	78.39	53.26
1ben_vdw.sql	1713	74.48	59.69	44.11
1ben_vdw_white.sql	1713	74.72	60.04	43.96
2bg9_cpk.sql	14924	177.71	93.20	60.25
2bg9_vdw.sql	14924	26.55	27.63	24.15
2bg9_vdw_index.sql	14924	67.55	55.52	41.87
1xi5_vwd.sql	15696	237.03	106.27	64.77
1j5e_cpk.sql	51933	143.67	89.73	58.39
1j5e_vdw.sql	51933	35.86	34.76	29.26
1kqs_cpk.sql	98688	47.09	43.56	34.99
1vqm_lic.sql	99045	84.51	67.47	48.14
1vqm_vdw.sql	99045	12.52	14.18	13.14

Nvidia GeForce 8400 GS with camera distance 1

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	173.55	54.24	51.09
1crn_cpk.sql	327	114.85	46.97	44.84
1crn_vdw.sql	327	21.03	20.07	19.73
1ben_vdw.sql	1713	8.58	10.00	9.92
1ben_vdw_white.sql	1713	8.57	10.00	9.92
2bg9_cpk.sql	14924	41.62	29.93	29.14
2bg9_vdw.sql	14924	5.10	6.36	6.30
2bg9_vdw_index.sql	14924	7.70	8.92	8.84
1xi5_vwd.sql	15696	49.78	32.62	31.51
1j5e_cpk.sql	51933	38.28	27.36	26.67
1j5e_vdw.sql	51933	3.72	4.55	4.52
1kqs_cpk.sql	98688	13.62	13.75	13.60
1vqm_lic.sql	99045	40.00	27.83	27.01
1vqm_vdw.sql	99045	2.48	3.14	3.13

Nvidia GeForce 8400 GS with camera distance 3

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	109.45	48.16	46.63
1crn_cpk.sql	327	199.28	54.06	51.17
1crn_vdw.sql	327	52.15	34.92	33.65
1ben_vdw.sql	1713	32.00	26.28	25.61
1ben_vdw_white.sql	1713	32.02	26.28	25.61
2bg9_cpk.sql	14924	72.89	38.35	36.87
2bg9_vdw.sql	14924	10.80	11.52	11.37
2bg9_vdw_index.sql	14924	28.43	23.21	22.65
1xi5_vwd.sql	15696	98.73	43.24	41.32
1j5e_cpk.sql	51933	96.11	43.33	41.45
1j5e_vdw.sql	51933	14.56	13.88	13.66
1kqs_cpk.sql	98688	20.85	17.82	17.54
1vqm_lic.sql	99045	61.80	35.67	34.39
1vqm_vdw.sql	99045	5.28	5.87	5.83

Nvidia GeForce 8400 GS with camera distance 5

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	155.52	56.68	53.57
1crn_cpk.sql	327	246.85	58.01	54.56
1crn_vdw.sql	327	106.69	46.14	44.29
1ben_vdw.sql	1713	72.16	39.89	38.31
1ben_vdw_white.sql	1713	72.11	39.90	38.35
2bg9_cpk.sql	14924	124.28	46.51	44.41
2bg9_vdw.sql	14924	26.35	22.02	21.62
2bg9_vdw_index.sql	14924	60.87	35.18	33.89
1xi5_vwd.sql	15696	152.05	50.00	47.68
1j5e_cpk.sql	51933	117.23	47.05	44.84
1j5e_vdw.sql	51933	33.33	25.15	24.48
1kqs_cpk.sql	98688	40.24	27.56	26.77
1vqm_lic.sql	99045	74.45	40.55	39.01
1vqm_vdw.sql	99045	12.10	11.73	11.58

Nvidia GeForce 8600 GT with camera distance 1

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	636.40	201.64	179.60
1crn_cpk.sql	327	349.70	166.42	152.07
1crn_vdw.sql	327	55.96	60.20	58.22
1ben_vdw.sql	1713	22.62	28.17	27.72
1ben_vdw_white.sql	1713	22.62	28.18	27.74
2bg9_cpk.sql	14924	121.20	97.22	92.15
2bg9_vdw.sql	14924	13.53	17.45	17.24
2bg9_vdw_index.sql	14924	20.58	24.97	24.63
1xi5_vwd.sql	15696	145.81	107.30	101.33
1j5e_cpk.sql	51933	109.80	86.46	82.84
1j5e_vdw.sql	51933	9.64	12.09	12.01
1kqs_cpk.sql	98688	37.58	40.42	39.60
1vqm_lic.sql	99045	114.99	88.45	84.37
1vqm_vdw.sql	99045	6.36	8.25	8.19

Nvidia GeForce 8600 GT with camera distance 3

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	333.90	180.88	164.30
1crn_cpk.sql	327	651.00	198.00	178.28
1crn_vdw.sql	327	146.17	112.03	106.12
1ben_vdw.sql	1713	89.51	81.81	78.48
1ben_vdw_white.sql	1713	89.66	81.98	78.51
2bg9_cpk.sql	14924	216.01	129.40	120.88
2bg9_vdw.sql	14924	29.54	33.11	32.52
2bg9_vdw_index.sql	14924	80.27	71.64	68.92
1xi5_vwd.sql	15696	290.00	148.17	137.13
1j5e_cpk.sql	51933	278.62	147.04	135.62
1j5e_vdw.sql	51933	39.32	39.85	39.00
1kqs_cpk.sql	98688	58.39	53.96	52.38
1vqm_lic.sql	99045	175.85	117.00	109.78
1vqm_vdw.sql	99045	13.60	15.67	15.53

Nvidia GeForce 8600 GT with camera distance 5

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	506.55	211.28	190.28
1crn_cpk.sql	327	850.20	210.58	188.68
1crn_vdw.sql	327	322.94	162.45	149.36
1ben_vdw.sql	1713	212.91	133.37	124.65
1ben_vdw_white.sql	1713	212.56	133.59	124.35
2bg9_cpk.sql	14924	385.60	165.50	152.05
2bg9_vdw.sql	14924	74.35	67.87	65.43
2bg9_vdw_index.sql	14924	175.68	115.24	108.62
1xi5_vwd.sql	15696	474.61	179.38	162.92
1j5e_cpk.sql	51933	333.93	161.54	149.13
1j5e_vdw.sql	51933	88.16	73.75	71.04
1kqs_cpk.sql	98688	112.78	87.11	83.13
1vqm_lic.sql	99045	207.52	131.89	123.25
1vqm_vdw.sql	99045	31.53	32.29	31.72

Nvidia GeForce 9800 GTX+ with camera distance 1

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	3293.40	1235.70	1142.30
1crn_cpk.sql	327	1790.70	1031.40	954.40
1crn_vdw.sql	327	340.07	372.40	362.33
1ben_vdw.sql	1713	138.63	172.93	169.80
1ben_vdw_white.sql	1713	138.20	171.98	170.41
2bg9_cpk.sql	14924	648.14	572.30	546.20
2bg9_vdw.sql	14924	82.49	105.12	104.81
2bg9_vdw_index.sql	14924	118.69	145.24	143.39
1xi5_vwd.sql	15696	749.40	622.84	591.74
1j5e_cpk.sql	51933	561.80	493.65	474.25
1j5e_vdw.sql	51933	53.00	65.18	64.89
1kqs_cpk.sql	98688	204.80	220.73	217.36
1vqm_lic.sql	99045	551.94	479.40	460.05
1vqm_vdw.sql	99045	36.49	46.44	46.27

Nvidia GeForce 9800 GTX+ with camera distance 3

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	1786.50	1026.60	943.20
1crn_cpk.sql	327	2814.20	1217.80	1108.30
1crn_vdw.sql	327	816.84	692.40	655.83
1ben_vdw.sql	1713	484.90	476.20	460.75
1ben_vdw_white.sql	1713	490.65	482.55	463.85
2bg9_cpk.sql	14924	1071.30	755.20	711.83
2bg9_vdw.sql	14924	149.57	166.58	164.48
2bg9_vdw_index.sql	14924	352.36	309.74	301.04
1xi5_vwd.sql	15696	1399.10	869.30	812.80
1j5e_cpk.sql	51933	1134.40	847.90	790.12
1j5e_vdw.sql	51933	147.99	132.92	131.73
1kqs_cpk.sql	98688	298.74	289.81	283.20
1vqm_lic.sql	99045	717.73	626.04	598.14
1vqm_vdw.sql	99045	58.75	59.26	58.77

Nvidia GeForce 9800 GTX+ with camera distance 5

Filename	# Spheres	Direct shading	Deferred shading	
		Phong (fps)	Phong (fps)	Phong + SL (fps)
handle.sql	2	2535.30	1184.30	1073.10
1crn_cpk.sql	327	3432.50	1302.60	1184.50
1crn_vdw.sql	327	1642.40	996.30	919.40
1ben_vdw.sql	1713	992.40	742.90	702.33
1ben_vdw_white.sql	1713	990.00	743.30	702.20
2bg9_cpk.sql	14924	1801.30	1022.90	947.60
2bg9_vdw.sql	14924	318.94	287.34	280.87
2bg9_vdw_index.sql	14924	639.14	453.55	436.76
1xi5_vwd.sql	15696	1799.30	1002.00	929.60
1j5e_cpk.sql	51933	1212.20	859.40	801.50
1j5e_vdw.sql	51933	261.80	210.66	207.12
1kqs_cpk.sql	98688	483.55	448.46	431.60
1vqm_lic.sql	99045	835.90	659.30	636.87
1vqm_vdw.sql	99045	108.38	94.61	93.78