Splatting Illuminated Ellipsoids with Depth Correction

Stefan Gumhold

University of Tübingen,
WSI/GRIS
Sand 14, 72076 Tübingen, Germany
Email: gumhold@gris.uni-tuebingen.de

Abstract

Ellipsoids are important primitives used in visualization and modeling, where often a larger number of ellipsoids have to be displayed in real-time. The standard approach of tessellating each ellipsoid into a smooth polygonal mesh leads to unacceptable polygon counts that dramatically increase the rendering time. In this paper a method is proposed to splat ellipsoids perspectively correct. The splatted ellipsoids are illuminated with the accuracy of floating point precision by exploiting the fragment shader facility of current graphics accelerators. It is also shown how to correct the depth value of the fragment position such that overlapping ellipsoids are displayed correctly.

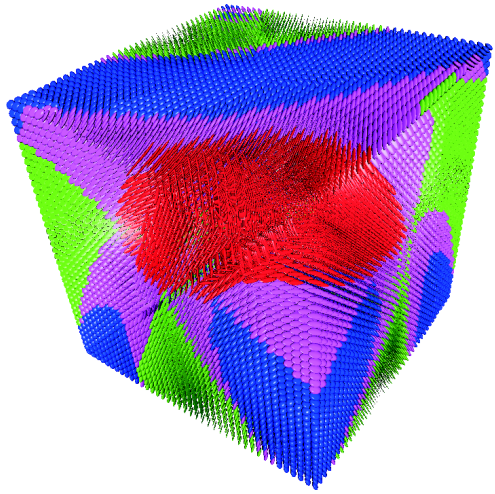


Figure 1: Test scene of a large number of ellipsoids that visualize a tensor field.

1 Introduction

Ellipsoids have been used as primitives in different applications for visualization and modeling. In the visualization domain, ellipsoids have been used successively for the splatting of volumetric data sets [15, 8, 9, 13, 10, 17], 3D clouds [3]. and point sampled surfaces [11, 18, 12]. In all three types of approaches ellipsoids are the basis functions into which the volumetric or surface data set is decomposed. The ellipsoids represent the density of the data set, heavily overlap and have to be superposed in order to reconstruct the data set.

In our approach we are not interested in the splatting of densities but in splatting the illuminated surface of ellipsoids. The application for which we developed our approach is the visualization of symmetric tensor fields [14, 2, 7, 6, 5, 16]. The most intuitive approach to visualize a symmetric tensor is to render an ellipsoid. The mathematical relation between ellipsoids and symmetric tensors is derived in section 2. In the simplest tensor field visualization approach one tessellates the three dimensional domain of the tensor field with regularly spaced ellipsoids as shown in Figure 1. The coloring illustrates the shape of the ellipsoids. Red ellipsoids look like cigars, green ones like pan cakes and blue ones like balls. The magenta ones are not classified in one of the three groups. At the moment we are working on a better placement of the ellipsoids that illustrate the underlying topology of the tensor field. But for both approaches an interactive visualization tool demands for fast rendering of a large number of illuminated ellipsoids.

Guthe et al. [4] have proposed a method to splat simple geometric shapes such as ellipsoids, arrows, etc, that exhibit a rotational symmetry. The primitives are pre-rendered from a large number of view points and during visualization the best pre-

rendered version is splat on the screen. All the primitives were pre-lighted and no depth correction could be performed.

Another approach that would benefit from our method was described by Bischoff and Kobbelt [1]. They decompose 3D-models represented as triangle meshes into a set of overlapping ellipsoids, such that the surface of the union of ellipsoids re-samples the surface of the 3d-model. They rearrange the ellipsoids into an order suitable for progressive transmission. The representation is very robust to the loss of single ellipsoids. For the reconstruct of the 3D model they use the marching cubes algorithm producing again a triangular mesh. With our ellipsoid rendering approach the ellipsoidal representation can directly be rendered efficiently. The combination of both approaches has the potential for a new view dependent rendering system.

There are several approaches to render ellipsoids. One can ray-trace ellipsoids, tessellate them into triangles and render the triangles with a graphics accelerator or one can splat the ellipsoid. The by far fastest approach is splatting with a graphics accelerator, i.e. one simply renders a triangle or as we do a quadrilateral that covers the primitive (ellipsoid) and one supplies a texture that represents shape and illumination of the primitive. The shape is typically encoded in the α-channel with a 0 for texels that do not belong to the shape and a 1 for texels that do. The graphics accelerators allow to discard fragments (rastered pixels) in dependence of the α value, such that pixels where the fragments α is zero are not touched at all.

Most of the newer graphics accelerators come with an extended texturing facility that can be accessed through a vertex and a fragment shader API, such as CG from NVIDIA or the corresponding OpenGL ARB extensions. The vertex shader API allows to perform view dependent calculations for each vertex without the need to re-specify the vertex data again if the view point changes. The output of the vertex shader is a collection of scalar and vector valued data that is passed to the fragment shader. This data is dealt with in the same way as texture coordinates, which are perspectively correct interpolated over the triangles or quadrilaterals that are rasterized. In world coordinates this means that the data is linearly interpolated over the triangles and quadrilaterals. For each pixel encountered during rasterization the interpolated data is passed to the fragment shader that computes the final color and depth coordinate of the fragment. The latter information is used for the α- and z-buffer tests and if these succeed for the combination of the fragment color with the current pixel color.

In this paper we first elaborate on some basic characteristics of ellipsoids in section 2. Then we derive the necessary formula to compute the corner vertices of a quadrilateral splat that contains the silhouette of an ellipsoid seen from the current view point in section 3. In section 4 we solve the ray-ellipsoid intersection problem for a given ray from the view location to a pixel. The result is used to compute the surface normal and surface location at the pixel, what is necessary for the illumination computation and the depth correction. Section 5 solves some problems arising with less flexible fragment shaders. We close the paper with a comparison of the proposed rendering approach with other approaches to render symmetric tensors.

The main contribution of this paper is the derivation of a simple solution of the ray-ellipsoid intersection problem, which can be used incrementally and implemented in vertex and fragment shaders of currently available graphics accelerators.

2 Background on Ellipsoids

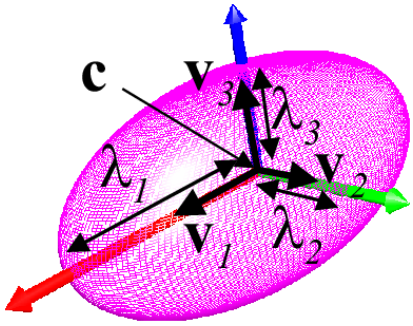


Figure 2: characteristic quantities of an ellipsoid

An ellipsoid in 3D is given by a center location $\mathbf{c}$, an orthonormal basis $\mathbf{v}_1, \mathbf{v}_2$ and $\mathbf{v}_3$ and for each basis direction a radius λ_i as illustrated in Figure 2. We adopt the convention that $\lambda_1 \geq \lambda_2 \geq \lambda_3$, what can always be ensured by a permutation of the basis vectors $\mathbf{v}_i$. Any ellipsoid can be generated from a sphere by stretching the sphere by the λ_i in the

major directions $\mathbf{v}_i$. If we enter the radii in the diagonal matrix Λ and the $\mathbf{v}_i$ as columns in the rotation matrix $O = (\mathbf{v}_1\mathbf{v}_2\mathbf{v}_3)$, we can define the set of points on an ellipsoid from the points on the unit sphere

$$E = \{\mathbf{q} = O\Lambda\mathbf{p} + \mathbf{c} | \, \|\mathbf{p}\| = 1\}.$$

As a rotation of the points $\mathbf{p}$ on the unit sphere reproduces the unit sphere, we can as well replace $O\Lambda$ in the definition of the ellipsoid by the symmetric positive definite matrix

$$T \stackrel{\text{def}}{=} O\Lambda O^T.$$

We can interpret $T\mathbf{v} + \mathbf{c}$ as a transformation from a parameter space of points on the unit sphere into world space. We denote points $\tilde{\mathbf{p}}$ in the parameter space with a tilde on top and the definition of an ellipsoid becomes

$$E = \{\mathbf{p} = T\tilde{\mathbf{p}} + \mathbf{c} | \, \|\tilde{\mathbf{p}}\| = 1\}, \qquad (1)$$

which is valid for an arbitrary symmetric, positive definite matrix T. We can conclude that a non degenerate ellipsoid is defined by a symmetric, strictly positive definite matrix T and a center location $\mathbf{c}$.

From equation 1 we can directly derive the implicite representation of an ellipsoid in world space by inverting $\mathbf{p} = T\tilde{\mathbf{p}} + \mathbf{c}$

$$1 = \|\tilde{\mathbf{p}}\|^2 = \left\|T^{-1}(\mathbf{p} - \mathbf{c})\right\|^2.$$

The surface normal can be computed from the implicite representation via the gradient operator, resulting in the not normalized normal vector $\mathbf{n}$ in world coordinates:

$$\begin{aligned} \mathbf{n} &= \nabla\left[\left\|T^{-1}(\mathbf{p} - \mathbf{c})\right\|^2\right] \\ &= T^{-2}(\mathbf{p} - \mathbf{c}) \qquad (2) \\ &= T^{-1}\tilde{\mathbf{p}} = T^{-1}\tilde{\mathbf{n}}. \qquad (3) \end{aligned}$$

In equation 3 we transformed $\mathbf{p}$ back to parameter space, i.e. on the sphere, where the normalized normal $\tilde{\mathbf{n}}$ is the same as the location vector $\tilde{\mathbf{p}}$. Although the equations for the normal are defined all over space they only make sense on the surface of the ellipsoid. Equation 3 tells us that the normal is transformed from parameter space to world space with the inverse $T^{-1} = O\Lambda^{-1}O^T = \left(T^{-1}\right)^T$.

3 Splatting the Silhouette

For the rendering of ellipsoids we assume a pinhole camera as used in most applications with an eye point $\mathbf{e}$, a view look at point and a view up direction. In order to be able to splat an ellipsoid with a planar quadrilateral, the silhouette of the ellipsoid is determined. Here only the eye point is of interest. The silhouette of the ellipsoid seen from the eye point is given by all points on the ellipsoids where the surface normal is orthogonal to the vector to the eye point

$$S = \left\{\mathbf{p} \in E | \mathbf{n}^T(\mathbf{p} - \mathbf{e}) = 0\right\}.$$

If we transform the definition of the silhouette into parameter space by transforming $\mathbf{n}$ via equation 3 and $\mathbf{p}$ and $\mathbf{e}$ via 1 we get the silhouette $\tilde{S}$ in parameter space

$$\tilde{S} = \left\{\tilde{\mathbf{p}} | \, \|\tilde{\mathbf{p}}\| = 1 \wedge \tilde{\mathbf{n}}^T(\tilde{\mathbf{p}} - \tilde{\mathbf{e}}) = 0\right\}, \qquad (4)$$

as the T and T^{-1} cancel each other out. Thus we can compute the silhouette in parameter space and transform it back.

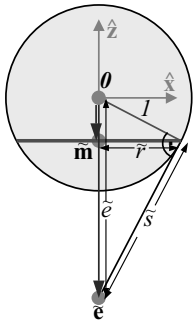


Figure 3: Computation of the silhoutte in parameter space. The silhoutte is a circle.

Figure 3 shows the 2D version of the silhouette reconstruction problem in parameter space, where the ellipsoid is a unit sphere with center in the origin. It is obvious that the silhouette is a circle on a plane orthogonal to $\tilde{\mathbf{e}}$. Let $\tilde{\mathbf{m}}$ denote the center of the circle and $\tilde{r}$ its radius. By applying twice Pythagoras one can derive that $\tilde{\mathbf{m}}$ is $1/\|\tilde{\mathbf{e}}\|$ away from the center in direction of the eye point, i.e. with $\tilde{e} = \|\tilde{\mathbf{e}}\|$

$$\tilde{\mathbf{m}} = \frac{1}{\tilde{e}^2}\tilde{\mathbf{e}}.$$

The radius of the circle can be computed from

$$1 = \tilde{r}^2 + \frac{1}{\tilde{e}^2} \quad \Longrightarrow \quad \tilde{r}^2 = 1 - \frac{1}{\tilde{e}^2}. \qquad (5)$$

If we build an orthonormal basis $\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$ in parameter space with the z-direction in the opposite direction of $\tilde{\mathbf{e}}$ as illustrated in Figure 3, the silhouette is the circle parameterized through ϕ

$$\tilde{S} = \{\tilde{\mathbf{m}} + \tilde{r}\,(\cos\phi\hat{\mathbf{x}} + \sin\phi\hat{\mathbf{y}})\,|\phi \in [0, 2\pi]\}\,.$$

To splat the ellipsoids we simply transform the location $\tilde{\mathbf{m}}$ and the vectors $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ back to world space

$$\mathbf{m} = T\tilde{\mathbf{m}} + \mathbf{c}, \mathbf{x} = T\hat{\mathbf{x}}, \mathbf{y} = T\hat{\mathbf{y}}$$

and splat a quadrilateral with the four corners

$$V_{\pm\pm} \stackrel{\text{def}}{=} \mathbf{m} + \tilde{r}\,(\pm\mathbf{x} \pm \mathbf{y}) \qquad (6)$$

resulting from choosing the four possible sign combinations $++$, $+-$, $-+$ and $--$. If we only wanted to fill the ellipsoid with a uniform color we could simply texture the quadrilateral with a texture containing a filled unit circle. All the computations necessary to compute the four corners of a quadrilateral can be easily performed in the vertex shader units. We will come back to that later on when it will be clear what further parameters are needed by the fragment shader.

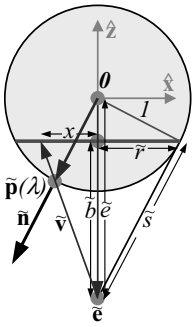
4 Incremental Ray Tracing of Ellipsoids

4.1 Equation of the Intersection

After we have derived the formulas to compute the corners of a quadrilateral splat that covers the silhouette completely, we need to compute the intersection of the ray

$$\mathbf{p}(\lambda) = \mathbf{e} + \lambda\mathbf{v} \qquad (7)$$

from the eye location $\mathbf{e}$ in the direction $\mathbf{v}$ of the current pixel, where $\mathbf{v}$ is the vector from the eye location to the pixel location as illustrated in parameter space in Figure 4. This computation has to be done for each pixel covered by the splat in the fragment shader. Therefore, we want to derive a formula as simple as possible, which allows to share as many computational results as possible between the pixels covered by one ellipsoid.



a)

$$\tilde{\mathbf{v}} = \begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{b} \end{pmatrix}$$

$$\tilde{\mathbf{e}} = \begin{pmatrix} 0 \\ 0 \\ -\tilde{e} \end{pmatrix}$$

$$\Rightarrow \tilde{\mathbf{e}}^T\tilde{\mathbf{v}} = -eb$$

b)

Figure 4: Illustration of the intersection computation between the ray from $\tilde{\mathbf{e}}$ in direction of $\tilde{\mathbf{v}}$ with the spherical ellipsoid in parameter space. The intersection point $\tilde{\mathbf{p}}(\lambda)$ is equal to the normalized normal $\tilde{\mathbf{n}}$ in parameter space.

The intersection of the ray and the ellipsoid has to be between the plane of the silhouette and the eye location. From this follows that λ has to be in the interval $[0, 1]$. The transformation to parameter space is an affine transformation and preserves straight lines. Thus, we can as well compute λ in parameter space from the much simpler ray-sphere intersection

$$1 = \|\tilde{\mathbf{p}}(\lambda)\|^2\,. \qquad (8)$$

Figure 4 illustrates the ray intersection problem with all the necessary lengths. On the right the coordinates of the vectors $\tilde{\mathbf{e}}$ and $\tilde{\mathbf{v}}$ are shown. We see that the scalar product $\tilde{\mathbf{e}}^T\tilde{\mathbf{v}}$ is given by $-\tilde{e}\tilde{b}$. For the silhouette length $\tilde{s}$ we get from Pythagoras and with the equation for the radius $\tilde{r}$

$$\tilde{s}^2 = \tilde{e}^2 - 1 = \tilde{e}^2\tilde{r}^2 = \tilde{e}^2\left(\tilde{s}^2 - \tilde{b}^2\right). \qquad (9)$$

And by solving for $\tilde{e}\tilde{b}$ and applying $\tilde{s}^2 = \tilde{e}^2 - 1$ once more, we get $\tilde{\mathbf{e}}^T\tilde{\mathbf{v}} = \tilde{s}^2$. Plugging this into 8 yields

$$0 = \tilde{s}^2 - 2\tilde{s}^2\lambda + \tilde{v}^2\lambda^2.$$

Next we plug in the coordinates of $\tilde{\mathbf{v}}$ and devide the equation by $\tilde{s} = \tilde{e}^2\tilde{r}^2$:

$$0 = 1 - 2\lambda + \frac{\tilde{b}^2 + \tilde{x}^2 + \tilde{y}^2}{\tilde{e}^2\tilde{r}^2}\lambda^2.$$

The ray only has an intersection with the sphere if the 2D vector formed by the x- and y-coordinates of $\tilde{\mathbf{v}}$ has a length smaller than $\tilde{r}$. By dividing the coordinates through $\tilde{r}$, the in this way normalized

x- and y-coordinates need to be inside the easier to handle unit circle. We therefore define

$$\tilde{\mathbf{q}} \stackrel{\text{def}}{=} \frac{1}{\tilde{r}} \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} \qquad \tilde{q}^2 = \frac{1}{\tilde{r}} \left(\tilde{x}^2 + \tilde{y}^2 \right) \in [0,1].$$

With the easily derivable equality $\tilde{b}^2 = \tilde{e}^2 \tilde{r}^4$ the equation for λ simplifies to

$$\begin{aligned} 0 &= 1 - 2\lambda + \left(\tilde{r}^2 + \frac{\tilde{q}^2}{\tilde{e}^2} \right) \lambda^2 \\ &\stackrel{(5)}{=} 1 - 2\lambda + \left(1 - \frac{1 - \tilde{q}^2}{\tilde{e}^2} \right) \lambda^2. \quad (10) \end{aligned}$$

We finally define the reciprocal $\tilde{u}$ of $\tilde{e}$, which has to be between zero and one for any view location outside of the ellipsoid, and the two quantities α and β as follows

$$\begin{aligned} \tilde{u} &\stackrel{\text{def}}{=} \frac{1}{\tilde{e}} \in [0,1] \\ \alpha &\stackrel{\text{def}}{=} \sqrt{1 - \tilde{q}^2} \\ \beta &\stackrel{\text{def}}{=} \frac{1}{\tilde{e}} \alpha = \tilde{u}\alpha. \end{aligned}$$

Comparing the definition of β with equation 10 gives

$$0 = 1 - 2\lambda + (1 - \beta^2)\lambda^2,$$

which has only one solution, which is smaller than one

$$\lambda = \frac{1}{1 \mp \beta} \stackrel{\lambda \leq 1}{=} \frac{1}{1 + \beta} = \frac{1}{1 + \tilde{u}\sqrt{1 - \tilde{q}^2}}. \quad (11)$$

4.2 Incremental Implementation

This surprisingly simple formula enables a very fast incremental computation of the ray-ellipsoid intersection, which is perfectly suited for the implementation in a fragment shader. For this we define corresponding to the four corners (6) of the splat two linearly interpolated vertex attributes $A[0]$ and $A[1]$ and one constant $C[0]$

$$\begin{aligned} A[0]_{\pm\pm} &\stackrel{\text{def}}{=} V_{\pm\pm} - \mathbf{e}, \\ A[1]_{\pm\pm} &\stackrel{\text{def}}{=} (\pm 1, \pm 1, \tilde{u})^T, \\ C[0] &\stackrel{\text{def}}{=} \mathbf{e}, \end{aligned}$$

which encapsulate $\mathbf{v}$, $\tilde{\mathbf{q}}$ together with the per ellipsoid constant term $\tilde{u}$ and $\mathbf{e}$. For each pixel we first compute $\tilde{q}^2$ and check if it is ≤ 1. If not, the fragment is discarded. Otherwise λ is computed and via equation 7, $\mathbf{v} = A[0]$ and $\mathbf{e} = C[0]$ the intersection in world space.

4.3 Per Fragment Lighting

For the lighting computations we need the surface normal of the ellipsoid in world coordinates. As the surface normal in parameter space is identical to the ray-sphere intersection $\tilde{\mathbf{p}}$, the normal can be computed via equation 3 to

$$\mathbf{n}(\lambda) = T^{-1} \tilde{\mathbf{p}}(\lambda) = T^{-1} \left(\tilde{\mathbf{e}} + \lambda \tilde{\mathbf{v}} \right).$$

If we express the constitutes of $\tilde{\mathbf{p}}$ in the corresponding quantities in world coordinates we get

$$\mathbf{n}(\lambda) = T^{-2} \left(\mathbf{e} - \mathbf{c} + \lambda \mathbf{v} \right).$$

For the computation of the not normalized surface normal we introduce one per ellipsoid constant vertex attribute $A[2]$ and one linearly interpolated attribute $A[3]$

$$\begin{aligned} A[2] &\stackrel{\text{def}}{=} T^{-2} \left(\mathbf{c} - \mathbf{e} \right), \\ A[3]_{\pm\pm} &\stackrel{\text{def}}{=} T^{-2} A[0]_{\pm\pm}, \end{aligned}$$

which allow to compute the surface normal incrementally.

4.4 Per Fragment Depth Correction

As the x- and y-coordinates in screen space are known from the rasterization process, only the z-coordinate needs to be corrected. Let M be the transformation matrix from world space to screen space, i.e. perspective transformation and model view transformation and M_i its i-th row. For the depth correction we have to interpolate the screen space z- and w-coordinate, compute z and w for the intersection and finally divide the resulting z by the resulting w. For this we compute the two 2D vertex attributes in the vertex shader

$$\begin{aligned} A[4]_{\pm\pm} &\stackrel{\text{def}}{=} \left(M_2^T A[0]_{\pm\pm}, M_3^T A[0]_{\pm\pm} \right)^T, \\ A[5]_{\pm\pm} &\stackrel{\text{def}}{=} \left(M_2^T A[1]_{\pm\pm}, M_3^T A[1]_{\pm\pm} \right)^T. \end{aligned}$$

Figure 5 illustrates the depth correction at the example of two overlapping ellipsoids. In b) we zoomed onto the intersection curve which is nicely sampled on screen resolution independent of the viewing distance.

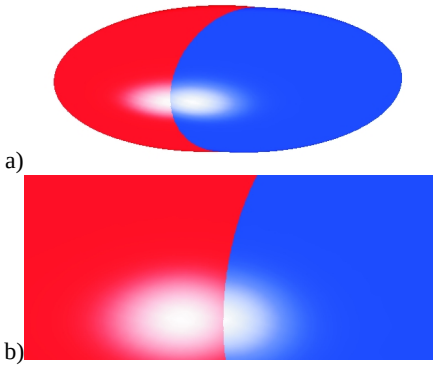


Figure 5: Illustration of depth correction for overlapping ellipsoids.

5 Low Precision Implementation

One can even implement the shading of splatted ellipsoids without depth correction on graphics accelerators that do not support square root operations but at least one dependent texture lookup in the fragment shader, as for example the Radeon 9000. On these graphics accelerators one can implement the computation of λ by a 3D texture lookup. For this one simply scales the coordinates of $\tilde{\mathbf{q}}$ to the range $[0,1]$ and uses $\tilde{u}$ as the third texture coordinate.

Two problems arise with this approach. Firstly, does the computation of $\mathbf{n}$ consume the only available dependent texture lookup, which would be necessary to allow for Phong shading. And secondly is λ for most distances $\tilde{e}$ very close to 1 all over the splat, what leads to severe numerical problems.

The first problem can be solved by implementing Phong shading with a 2D texture lookup for the diffuse component and a 3D texture lookup for the specular component. The arguments to the diffuse texture map are $s = \mathbf{n}^T\mathbf{n}$ and $t = \mathbf{n}^T\mathbf{l}$, where $\mathbf{l}$ is the direction vector to a directional light source. The map simply implements the function $t/\sqrt{s}$ clamped to $[0,1]$. Similarly, does the specular map take the three coordinates $s = \mathbf{n}^T\mathbf{n} \cdot \mathbf{h}^T\mathbf{h}$, $t = \mathbf{n}^T\mathbf{h}$ and $u = shininess/128$, where $\mathbf{h}$ is the interpolated half-vector. The specular map implements the function $(s/\sqrt{t})^{128u}$.

The second problem can be solved by transforming λ into the parameter μ that varies for any distance $\tilde{e}$ between zero and one. For this we examine the range of λ, which only depends on $\tilde{q}^2$. Substituting zero and one for $\tilde{q}^2$ results in the range

$$\lambda \in \left[\frac{\tilde{e}}{\tilde{e}+1}, 1\right].$$

Thus we define μ as

$$\mu \stackrel{\text{def}}{=} (\tilde{e}+1)\lambda - \tilde{e} = \frac{1-\alpha}{1+\tilde{u}\alpha} \in [0,1],$$

where the second equation can be derived with simple algebra. The ray-ellipsoid intersection computes to

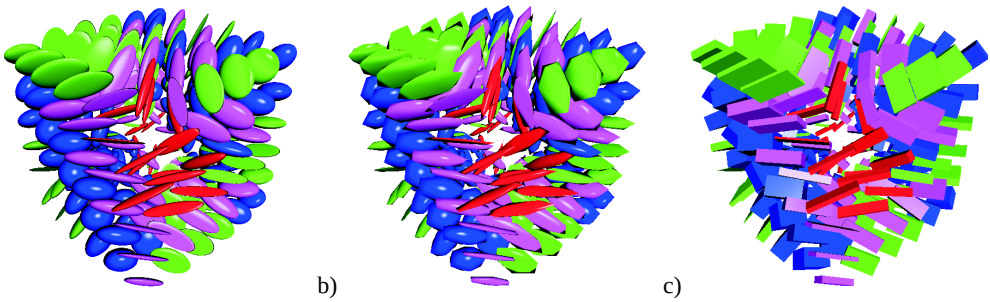
$$\mathbf{p} = \mathbf{e} + \lambda\mathbf{v} = \mathbf{e} + \frac{\tilde{e}}{\tilde{e}+1}\mathbf{v} + \mu\frac{1}{\tilde{e}+1}\mathbf{v}.$$

In a similar way we change the vertex attributes $A[2]$ and $A[3]$ necessary for the illumination calculations. One final problem arises as the inverse transformation T^{-1} can scale the world space surface normal to a length exceeding one, which leads to a problem for the texture lookup in the diffuse and specular maps. As the normal in parameter space is normalized, the transformation back to world coordinates can scale it no more than $\left\|T^{-1}\right\|$, which is equal to the largest eigenvalue of T^{-1} or the reciprocal of the smallest eigenvalue of T. The world space normal can therefore be kept shorter or equal to length one, if one divides the modified definitions of $A[2]$ and $A[3]$ by $\left\|T^{-1}\right\|$.

6 Results

We implemented vertex and fragment shaders with the OpenGL ARB vertex and fragment program extensions and the approach of section 5 with the ATI fragment shader extension. Our API consists of five functions:

1. `enableEllipsoidShader(const ViewDescr& vd)`... sets up the vertex and fragment programs and creates textures in the first call and binds textures and programs in successive calls and sets per frame constants.
2. `renderEllipsoid(const Pnt& center, const SymMat& T, const SymMat& I)`... renders an ellipsoid at the given center location, which is given by a symmetric matrix. Also the inverse of the symmetric matrix has to be provided.



a) b) c)

Figure 6: Visual comparision of three different rendering approaches for symmetric positive tensors: ellipsoids, icosahedra and boxes; all Phong shaded.

3. `setMaterial(float ambient, float diffuse, float specular, float shininess)...` sets the different material coefficients used for Phong shading.
4. `setSpecularColor(float r, float g, float b)...` sets the specular color.
5. `disableEllipsoidShader()...` turns off the vertex and fragment programs.

The color of the rendered ellipsoids can be specified via `glColor` commands.

Figure 1 shows our test scene of a symmetric tensor field visualized on a grid of $40^3 = 64000$ ellipsoids. In Figure 6 we compare a coarser sampling of the test scene tensor field for three different rendering approaches, that are typically used for the visualization of tensor fields. In a) the presented approach of ellipsoid visualization is used, in b) tessellated icosahedra are used and in c) rectangular boxes are shown. We optimized the rendering of icosahedra and boxes by stripification.

To analyze and compare the performance of our approach we rendered the test scene with different sized ellipsoids. We computed the number of ellipsoids rendered per second and the number of rastered fragments per second. In the diagrams of Figure 7 we plotted the number of rastered fragments per second over the number of ellipsoids per second, i.e. the fill rate over the setup speed. We compared the different rendering approaches for two graphics accelerators: a GeForceFX 5800 and a Radion 9000. For both cards the box rendering is always faster than the icosahedron rendering and the ellipsoid rendering is fast than the icosahedron rendering for small ellipsoids, i.e. a low fill rate, and slower for larger fill rates. In the case of the GeForce card, the ellipsoid rendering is even faster than box rendering in case of low fill rates. We can conclude that the setup for ellipsoid splatting is very fast but the splatting is fill rate limited already for small splat sizes (about 10^2 splats). But splatting is never more than twice slower as icosahedron rendering and achieves a much higher image quality. An appropriate tessellation of the sphere consumes surely more than twice as many triangles than an icosahedron and will in most cases be slower than our ellipsoid splatting approach.

In future work we want to apply our ellipsoid rendering strategy to the ellipsoidal representation proposed by Bischoff and Kobbelt [1]. We want to investigate how to render ellipsoid decompositions of 3D models view dependently. Furthermore, we want to investigate different texturing approaches for the ellipsoids such that also the anti-symmetric part of a tensor field can be visualized by for example a spiral pattern or a bump map.

References

[1] S. Bischoff and L. Kobbelt. Ellipsoid decomposition of 3d-models. In *Proceedings of 3DPVT Conference*, pages 480–488, 2002.

[2] T. Delmarcelle and L. Hesselink. Visualization of second order tensor fields and matrix data. In *Proceedings of IEEE Visualization Conference 1992*, pages 316–323, 1992.

[3] P. Elinas and W. Stürzlinger. Real-time rendering of 3d clouds. *Journal of Graphics Tools*, 5(4):33–45, 2000.

[4] S. Guthe, S. Gumhold, and W. Straßer. Interactive visualization of volumetric vector fields

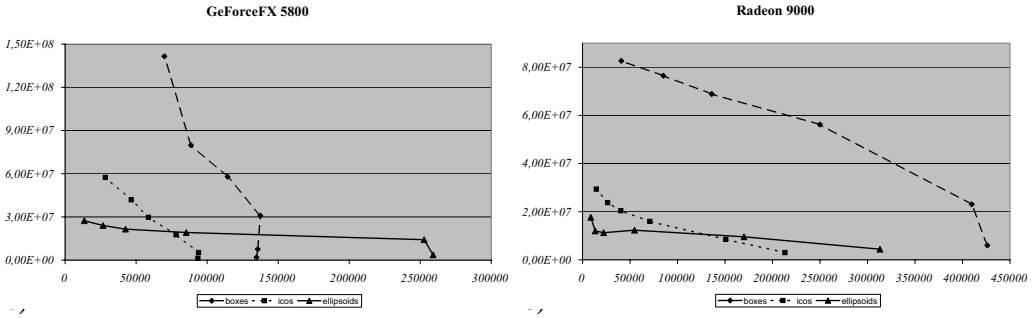


Figure 7: For two different graphics accelerators the fill rate in fragments per second plotted over the setup up speed in rendered tensors per second.

using texture based particles. In *Proceedings of WSCG Conference 2002*, 2002.

[5] G. Kindlmann, D. Weinstein, and D. Hart. Strategies for direct volume rendering of diffusion tensor fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):124–138, 2000.

[6] G. L. Kindlmann and D. M. Weinstein. Hueballs and lit-tensors for direct volume rendering of diffusion tensor fields. In *Proceedings of IEEE Visualization Conference 1999*, pages 183–189, 1999.

[7] D. H. Laidlaw, E. T. Ahrens, D. Kremers, M. J. Avalos, R. E. Jacobs, and Carol Readhead. Visualizing diffusion tensor images of the mouse spinal cord. In *Proceedings of IEEE Visualization Conference 1998*, pages 127–134, 1998.

[8] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Proceedings of ACM SIGGRAPH Conference 1991*, pages 285–288, 1991.

[9] X. Mao. Splatting of non rectilinear volumes through stochastic resampling. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):156–170, 1996.

[10] K. Mueller, T. Moeller, and R. Crawfis. Splatting without the blur. In *Proceedings of IEEE Visualization Conference 1999*, pages 363–370, 1999.

[11] H.-P. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH Conference 2000*, pages 335–342, 2000.

[12] L. Ren, H. Pfister, and M. Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings of Eurographics Conference 2002.*, 2002.

[13] J. E. Swan, K. Mueller, T. Moeller, N. Shareef, R. Crawfis, and R. Yagel. An anti-aliasing technique for splatting. In *Proceedings of IEEE Visualization Conference 1997*, pages 197–204, 1997.

[14] J. J. van Wijk. Spot noise: Texture synthesis for data visualization. In *Proceedings of ACM SIGGRAPH Conference 1991*, pages 309–318, 1991.

[15] L. Westover. Footprint evaluation for volume rendering. In *Proceedings of ACM SIGGRAPH Conference 1990*, pages 367–376, 1990.

[16] S. Zhang, C. Demiralp, D.F.Keefe, M. J. da Silva, D. H. Laidlaw, B. D. Greenberg, P.J. Basser, and E.A. Chiocca andC. Pierpaoli T.S. Deisboeck. An immersive virtual environment for dt-mri volume visualization applications: A case study. In *Proceedings of IEEE Visualization Conference 2001*, 2001.

[17] M. Zwicker, H. Pfister, J. VanBaar, and M. Gross. Ewa volume splatting. In *Proceedings of IEEE Visualization Conference 2001.*, 2001.

[18] M. Zwicker, H.-P. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proceedings of ACM SIGGRAPH Conference 2001*, pages 371–378, 2001.