FAST AND MEMORY EFFICIENT GPU-BASED RENDERING OF TENSOR DATA

Mario Hlawitschka*, Sebastian Eichelbaum#, and Gerik Scheuermann*
University of Leipzig
PF 100920, 04009 Leipzig
**{hlawitschka,scheuermann}@informatik.uni-leipzig.de*
#mai03jnw@studserv.uni-leipzig.de

ABSTRACT

Graphics hardware is advancing very fast and offers new possibilities to programmers. The new features can be used in scientific visualization to move calculations from the CPU to the graphics processing unit (GPU). This is useful especially when mixing CPU intense calculations with on the fly visualization of intermediate results. We present a method to display a large amount of superquadric glyphs and demonstrate its use for visualization of measured second-order tensor data in diffusion tensor imaging (DTI) and to stress and strain tensors of computational fluid dynamic and material simulations.

KEYWORDS

Tensor visualization, hardware acceleration, glyph rendering, GPU

1. INTRODUCTION

During the last years modern graphics hardware advanced very fast and with the introduction of shader programs and more recently high level shading languages enabled a new way to interfere into the normal rendering pipeline. Programmable shading can not only be used for enhancing surface features but to move parts of the calculation to the GPU, nowadays a fast and highly parallelized computer on its own. This influences not only traditional computer graphics applications but is used to develop advanced visualization techniques or improve existing ones in speed and visual quality.

Glyphs are a very basic but useful tool to visualize local information. For symmetric second-order tensors, i.e., symmetric matrices, tensor glyphs provide a useful tool to represent the full tensor information. In contrast to volume rendering of tensor data or tensor splats (Benger and Hege, 2004), their well-defined boundaries make them useful for quantitative visualization.

We use the features of modern graphic cards to implement a general and flexible version of tensor glyphs called superquadric tensor glyphs introduced by Kindlmann (2004). Our approach allows the interactive manipulation of glyph parameters, as the data only needs to be prepared for the GPU once. In addition to that, it frees the CPU from doing additional preprocessing of graphic data and therefore is useful when combined with animation as the sole responsibility of the CPU is to provide the raw tensor data. The GPU then takes all the action required for rendering the glyph described by an implicit surface.

2. BACKGROUND

According to Kindlmann (2004) a superquadric tensor glyph is defined by the surface given by the implicit function

$$q(x,y,z) := \left(x^{2/\alpha} + y^{2/\beta} \right)^{\alpha/\beta} + z^{2/\beta} - 1 = 0. \qquad (1)$$

The parameters α and β define the shape of the tensor glyph and are calculated using the eigenvalues of the tensor using the linear and planar anisotropy c_l and c_p defined by Westin et al (1997), and

$$\alpha = (1- c_p)^\gamma, \ \beta = (1- c_l)^\gamma \qquad \text{if } c_l \geq c_p \qquad (2)$$
$$\alpha = (1- c_l)^\gamma, \ \beta = (1- c_p)^\gamma \qquad \text{else.}$$

The remaining parameter γ defines the sharpness of the glyph, i.e., a visual parameter defining how close the glyph is to an ellipsoidal tensor glyph ($\gamma = 0$) or to a box glyph ($\gamma = \infty$). A detailed description of the parameter can be found in the original paper by Kindlmann (2004). The resulting parameterized glyph is then transformed by the tensor to represent its eigenvectors and eigenvalues.

In literature, there exist methods that can be used to render tensor glyphs, expecially the special cases of superquadric glyphs with γ=0, i.e., spheres and ellipsoids (Sigg et al., 2006 and Ranta et al., 2007). Schirsky et al. (2006) provide methods to render tubes. While rendering ellipsoids can be reduced to the implicit case of rendering spheres, tubes visually provide a high degree of symmetry, as the main axis is constant and the other two axes are rotationally symmetric. While there still is a point and plane symmetry in the superquadric glyph, there is no trivial reduction to the spherical case as it is not a linear transformation of a sphere. Nor does it provide the symmetry along two axes that is used when rendering tubes.

Another related field is the rendering of implicit surfaces. Here, most optimizations cannot be used because of the in general non-algebraic character of the superquadric function. While methods like those of Singh et al. (2007) are capable of rendering single glyphs at reasonable frame rates, they are not intended for a large amount of independent surfaces.

The basic idea behind our solution is writing a highly specialized ray caster for solving Eq 1. Therefore, a ray is parameterized at each pixel of the viewing plane, where to reduce the cost of shooting rays that miss the object and therefore improve the overall speed, the viewing plane is reduced to a bounding box of the projected glyph. As there is no analytic formula for the intersection of rays and superquadrics, we use a combination of iterative approaches for finding the roots of the implicit function. Thus, good starting parameters along the viewing ray need to be estimated in advance, as finding the iterative steps are the crucial and cost intensive part of the algorithm and therefore the number of steps should be as small as possible.

3. METHOD

The next section describes the transformation into the glyph's parameter space required to speed up most future calculations. Then the estimation of two kinds of bounding boxes is described in Sections 3.2 and 3.3. The main approach is described in Section 3.4 followed by a definition of stop criteria in Section 3.5. In Section 3.6, we split the algorithm in two parts that will be mapped to the vertex and the fragment shader of the GPU.

3.1 Transformation to Glyph’s Parameter Space

While all computations can be performed in a global coordinate system, it turns out, that most calculations presented in succeeding sections become quite simple when performed in the glyph's parameter space. This space is given uniquely if the tensor is not degenerate, i.e., at least two eigenvalues are the same, or that one or more eigenvalues becomes zero. The second case can be either ignored, as in most cases a tensor with a zero eigenvalue is meaningless, e.g., an eigenvalue smaller or equal to zero can not occur in a diffusion tensor magnetic resonance scan — or a threshold for the eigenvector size can be provided to ensure a certain minimal thickness of the glyph in all directions. The first case, in which eigenvalues are equal, does not matter as long as the transformation is chosen uniquely throughout the calculation, which is guaranteed, as in our implementation the coordinate system is set up once per glyph. The transformation from tensor space to world space is given by the eigenvector representation of the tensor, i.e, a matrix whose columns are represented by the scaled eigenvectors

$$M = (\lambda_1 e_1 \quad \lambda_2 e_2 \quad \lambda_3 e_3). \qquad (3)$$

The inverse transformation obviously is defined the inverse scaled eigenvectors as row vectors:

$$M^{-1} = (1/\lambda_1 e_1, \quad 1/\lambda_2 e_2, \quad 1/\lambda_3 e_3). \qquad (4)$$

The calculation of the coordinate system is performed in the vertex shader of the graphics board using the analytic approach by Hasan et al. (2001) that has been implemented using one square root and three trigonometric operations.

3.2 Estimating a Sprite Bounding Box

We estimate the glyph's bounding box on-the-fly in the vertex shader to avoid too many rays to be started per glyph. As including the sharpness parameter γ we do a worst-case approximation of the bounding box, by setting $\gamma=\infty$ where the glyph becomes a box. When transforming the eigenvectors into viewing space, they represent offsets to the center vertex that are used to compute a 2D bounding box in the viewing plane. Finally the plane is parameterized by transformed coordinates of the eye ray. This lets us use the internal interpolation of the graphic hardware that computes the final viewing ray parameters.

3.3 Volumetric Bounding Box

While the sprite only provides a raw estimate of a bounding box, we use a volumetric clipping of the rays for two purposes: First, the bounding box check enables us to discard most rays that will not hit the glyph, and second, it provides clipping parameters along the ray that provide good seeding values for the iteration step described later.
Since we transformed the ray into the glyph coordinate system we can do a fast test if the ray hits the glyph bounding box. This is now done by a simple intersection test with the unit cube. Therefore we have to intersect the ray with, in the worst case, 6 planes. The ray will most probably hit all non parallel planes. But only one hit is of interest: the hit where the ray enters the unit cube. A quick test, whether a side of the cube is a front or a back plane can be performed by testing whether the ray faces the cube's surface normal, i.e., $\langle \mathbf{n},\mathbf{v}\rangle < 0$. As the cube is axis aligned, the normal is parallel to the coordinate axes, and therefore this results in testing the sign of the corresponding direction vector component. This test will be successful for at most three planes. Those planes have to be tested if the ray hits it in the interval of [-1, 1] in both directions on plane. This is done by getting the remaining two coordinates of the point at which the ray hits the plane and test this point to be in the mentioned interval, in the both plane directions. If this is the case we found the plane where the ray enters the cube and no further comparisons are required. The already computed parameter of this point is used as seed for the iterative root search described in the next section and ensures that we always start iteration close to the surface of the glyph. Fig. 1 illustrates the alignment of the sprite and the volumetric bounding box in relation to an example glyph. In addition to the bounding box check, we know that the maximal value of t is reached at the point, where the ray would intersect the ellipsoid spanned by the scaled eigenvectors, which is in the glyph's coordinate system the unit sphere. This is always true as the glyph is a convex shape bounded by the ellipsoid and the box. Therefore, a simple ray-sphere intersection leads to a maximal value of t for those cases, where the sphere is hit. This may be kept in mind for further optimizations but is not used in our current implementation.

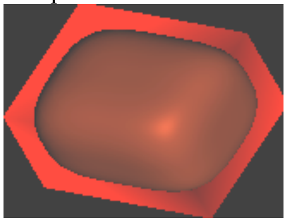

Figure 1: Shader output that visualizes both types of bounding box: The sprite bounding box is shown in black while the volumetric bounding box is shown around the glyph in red (light gray). Only rays in the red area are computed, all other points are discarded immediately.

3.4 Raycasting

We start by defining the parameterized viewing ray $\mathbf{r}(t):=\mathbf{p} + t\,\mathbf{v}$, where $\mathbf{v}$ is the viewing direction vector and $\mathbf{p}$ the point on the projection plane in the glyph coordinate system. Second, we need the surface function evaluated along the ray, which leads us to a one-dimensional function

$$q(\mathbf{r}(t)) := (r_x(t)^{2/\alpha} + \mathbf{r}_y(t)^{2/\alpha})^{\alpha/\beta} + r_z(t)^{\,2/\beta} - 1, \qquad (5)$$

whereof we have to compute the roots, especially find the smallest t with $q(\mathbf{r}(t)) = 0$. As there is no analytical way of solving this equation, numerical methods have to be used for solving it. There are many methods in literature for finding roots of one dimensional functions but as we are limited by the flexibility of the GPU, we use a Newton approach because sphere tracing has proven to be suitable for similar applications here. The main advantage is, that it does not require the calculation of higher derivatives, leading to additional evaluations of functions known to be slow on the GPU. In contrast to other papers, an implementation of sphere casting is not suitable here, as the estimates turn out to be worse than those of Newton's approach.

It definitely is a trade-off between the number of iterations and the complexity and computational efforts taken in the main loop but it can be seen from the results in Section 5 that the number of iterations required is low and therefore the algorithm performs good. The following pseudo-code fragment shows the main loop of the fragment shader, the method is illustrated in Fig. 2, too. We test the hit criterion first because it increases the overall performance:

```
float t = getBBoxIntersection();
for (i=0, i < maxNumSteps, i++)
{
  vec3 r=t*v + p;          // evaluate ray at position t
  float sq=q(r);           // superquadric function at current position
  vec3 gradient = grad(q(r));  // gradient at current position
  float sqd = dot(gradient, v);// first derivative
  float t2=t-(sq / sqd);       // the newton step

  if (hitCriteria(...))        // hit criteria fulfilled?
  {
    // lighting and depth calculations
    // the surface normal is stored in the gradient vector
    break;                     // stop iteration
  }

  if (breakCriteria(...))     // there will be no hit for sure
    discard;                  // do not paint anything
  t=t2;
}
```

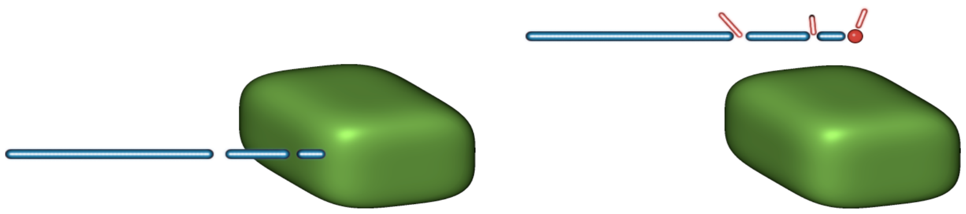

Figure 2: An Example of the newton approach. Left: We stop the iteration when the Newton step size reaches a pre-defined epsilon threshold. Right: If the gradient direction indicated by red bars flips, the glyph is missed and the iteration is stopped.

Because of our setup, the Newton iteration always approaches the glyph from outside and we descent the gradient until we reach a root of the function. As the gradient can only flip when the function's gradient is orthogonal to the ray, it becomes obvious that, if the current position is not completely tangent to the surface, we already missed the glyph. Therefore, the first stop criterion is the step size. If it becomes negative, we already missed the glyph for sure. This frees us from doing a lot of unneeded iteration steps as, e.g., approaches like sphere tracing usually do when passing close to a surface but fail to hit the surface.
While a test for decreasing step size usually terminates the Newton iteration, we test for $|q(\mathbf{r}(t))| - \epsilon < 0$, we already have to evaluate the function which means, it has no additional cost. As this test may fails in rare cases, where the viewing ray is almost orthogonal to the surface, we add a second hit criterion. This is the

abortion criterion of the Newton iteration: If the step size along the ray becomes smaller than a threshold, we assume to have hit the glyph.

3.4 Splitting the Calculation

As mentioned previously, a large part of the calculation has to be done once per glyph only. Therefore, almost every calculation can be done in the vertex shader. The following code falls into the responsibility of the vertex shader to

- compute the tensor decomposition (if required and not done by the CPU),
- compute 1-c_l, 1-c_p,
- compute the sprite coordinates
- compute the transformation matrix
- compute the (transformed) ray that will interpolated by the GPU,
- and compute the base color of the glyph.

The remaining responsibility of the fragment shader is to

- compute the bounding box intersection,
- compute the Newton iteration,
- and finally, if the glyph is hit, compute a neat surface shading and the depth component.

4. IMPLEMENTATION

We implemented the presented method into an existing visualization tool for medical and computational fluid dynamics data using C++, OpenGL and OpenGL Shading Language (GLSL). Due to limitations of our graphics hardware relating the size of point sprites, we use a GL_QUAD per glyph to create the graphic primitives, which increases the memory use slightly but still performs well. The depth buffer is used to render intersecting glyphs correctly.

While a set of different shading models are implemented in our system, we use the Blinn-Phong illumination model (Blinn, 1977) throughout this paper to keep the results comparable.

5. RESULTS

We tested the algorithm on multiple real-life data sets wherefrom we present two in this paper to illustrate the capabilities. The first data set is the well-known push-pull tensor data set shown in Fig. 3. We use this quite simple data set to test the scaling and color coding in the case of negative eigenvalues.

The second data set has been provided by the Max Planck Institute for Human Cognitive and Brain Science, Leipzig, Germany and shows a diffusion weighted magnetic resonance scan of a human brain. The raw data has been converted to a second-order tensor representation using a least-squares fit Basser et al. (1994) and contains about one million sample points on a regular grid. The tensors are masked and a total of about 100.000 valid glyphs inside the brain are sent to the graphics board where a filtering by fractional anisotropy takes place to avoid visualization of outliers and remaining data outside the brain.

Each change of parameters, like FA thresholds or changing the parameter γ takes typically about 50 μs of CPU time. While taking a closer look at all different types of glyphs, it has proven that a maximum of ten iterations for the Newton approach is enough to render all glyphs of the data set correctly. Most iterations after only three steps. This proves that seeding the glyphs at the bounding box is a good estimate for the start parameter t.

A test with a delta-wing data set containing about four million tensors has shown, that this amount of glyphs can be visualized using our approach at interactive frame rates. Obviously, visual clutter because of the limitation of number of pixels on screens of desktop workstations and PCs renders the visualization of this amount of glyphs non-practical but it could be useful on larger display walls or when implemented in a cave where larger screen surfaces are available.

6. CONCLUSION AND FUTURE WORK

We provided a way of implementing the superquadric tensor glyph on the GPU to make it a usable tool in combination with other tools that are heavily using the CPU (e.g., interactive glyph placement by Hlawitschka et al. (2007)) to provide visual feedback at interactive frame rates. While it already performs well on present data sets, there are still some improvements for the near future. First, we want to evaluate how far we can limit the parameter space for α, β and γ to perform most of the time-consuming evaluations by using one- and two-dimensional texture lookups. For standard applications, clipping α and β values to [0.2,1.0] limits the area of powers calculated to [1, 10] (terms 2/α and 2/β and [0, 5] (terms α/β and α/β-1 in the derived superquadric function of Eq. 1).

Although our new method reduces the amount of memory tremendously, it still is limited to the GPU's main memory and we want to couple the visualization with the underlying acceleration structure provided computed for our data set to be able to perform fast clipping of the data outside the viewport and implement level-of-detail approaches for glyphs further away from the viewer.

A further way to speed up the calculation would be the use of deferred shading as used in Sigg et al. (2006) Although computing the actual shading is the least part of the calculation, it may provide ways to perform early ray termination to discard fragments that are occluded by other glyphs early.

ACKNOWLEDGEMENT

We thank the Max Planck Institute for Human Cognitive and Brain Science, Leipzig, Germany for providing the data set used in this paper. We thank the image and signal processing group at the University of Leipzig for providing the *FAnToM* visualization system and Gordon Kindlmann for the valuable discussion.

REFERENCES

Basser, P., Mattiello, J. & LeBihan, D. (1994), 'Estimation of the effective self–diffusion tensor from the NMR spin echo', *Journal of Magnetic Resonance* 3(103), 247–254.

Benger, W, Hege, H.-C, (2004) ,Tensor Splats', *In Proceedings of the Conference on Visualization and Data Analysis*

Blinn, J. F. (1977), 'Models of light reflection for computer synthesized pictures', *Proc. 4th annual conference on computer graphics and interactive techniques*. pp. 192–198.

Hasan, K. M., Basser, P. J., Parker, D. L. & Alexander, A. L. (2001), 'Analytical computation of the eigenvalues and eigenvectors in DT–MRI', *Journal of Magnetic Resonance* 152, 41–47.

Hlawitschka, M., Scheuermann, G., Hamann, B. (2007) ,Interactive Glyph Placement for Tensor Fields', *Advances in Visual Computing: Third International Symposium, ISVC, Lake Tahoe, Nevada/California (Lecture Notes in Computer Science, LNCS 4841 and LNCS 4842, Springer),* George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Nikos Paragios, Syeda--Mahmood Tanveer , Tao Ju, Zicheng Liu, Sabine Coquillart, Carolina Cruz-Neira, Torsten Möller, and Tom Malzbender eds.

Kindlmann, G. (2004), 'Superquadric tensor glyph', *Joint EUROGRAPHICS – IEEE TCVG Symposium on Visualization*

Ranta, S. M., Singh, J. M. & Narayanan, P. J. (2006/2007), 'GPU ob jects', *Lecture Notes in Computer Science* 4338/2006.

Schirski, M., Bischof, C. & Kuhlen, T. (2006), Interactive Particle Tracing on Tetrahedral Grids Using the GPU, *In 'Proceedings of Vision, Modeling, and Visualization (VMV) 2006'*, pp. 153–160.

Sigg, C., Weyrich, T., Botsch, M. & Gross, M. (2006), 'GPU–based ray–casting of quadric surfaces', *Eurographics Symposium on Point–Based Graphics* .

Singh, J. M. & Narayanan, P. J. (n.d.), Real-time ray-tracing of implicit surfaces on the GPU, *Technical Report IIIT/TR/2007/72, Centre for Visual Information Technology, International Institute of Information Technology*, Hyderabad 500032. India.

Westin, C.-F., Peled, S., Gudbjartsson, H., Kikinis, R. & Jolesz, F. A. (1997), Geometrical diffusion measures for MRI from tensor basis analysis, *'ISMRM '97'*, Vancouver Canada, p. 1742.

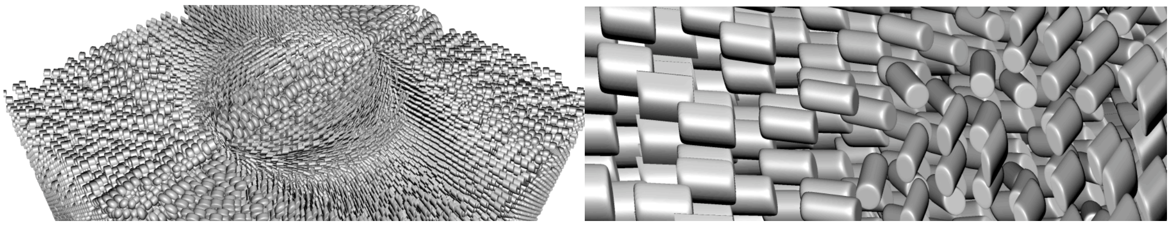
Figure 3: Glyphs in a double point data set deviator field. Overview(left) and close-up (right).

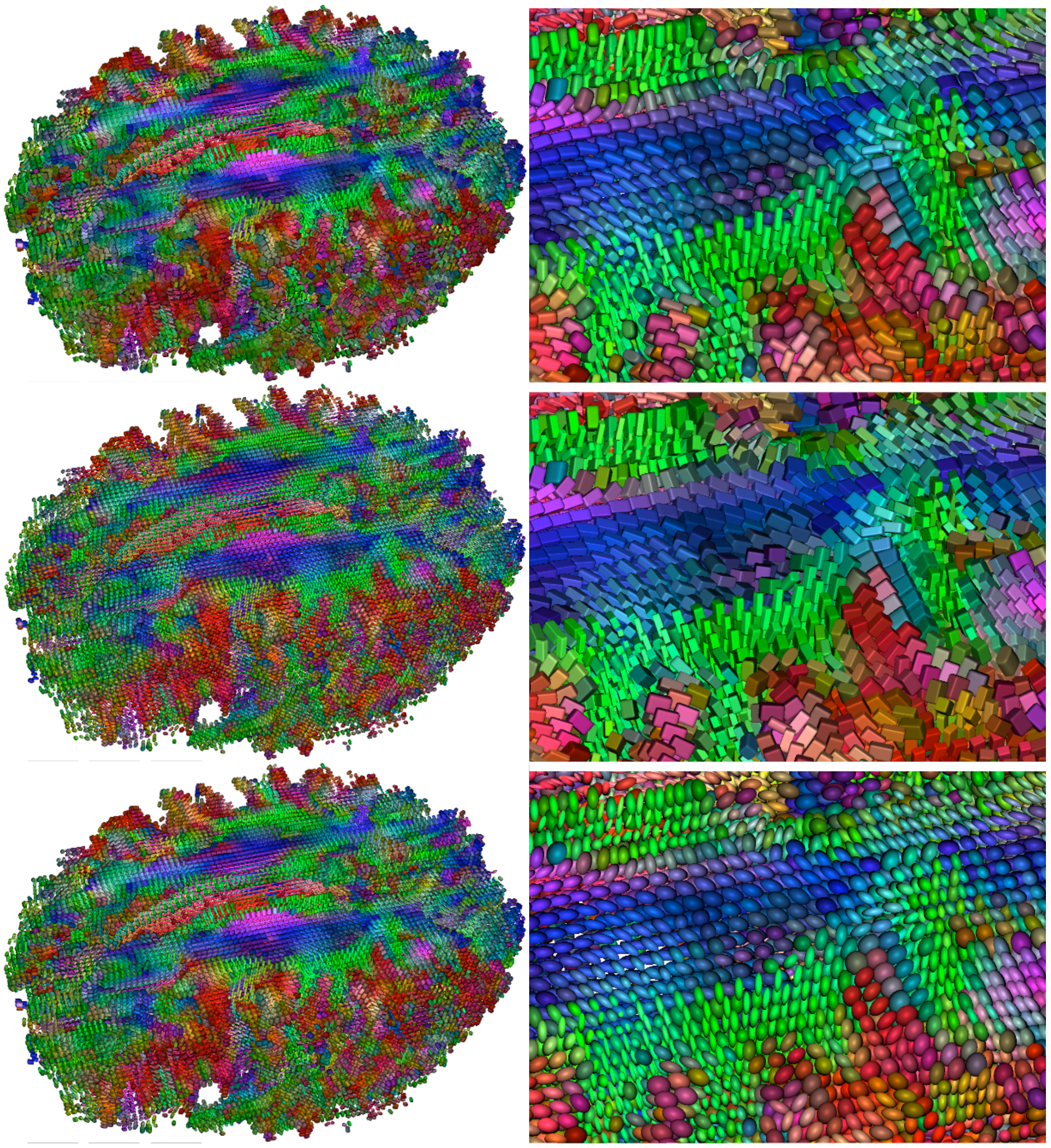
Figure 4: A full brain scan of diffusion tensors visualized using our approach for different parameters of γ. Left column: full brain. Right column: a zoomed view of the left column. γ is set to 0.0, 3.0 and 20.0 from top to bottom.