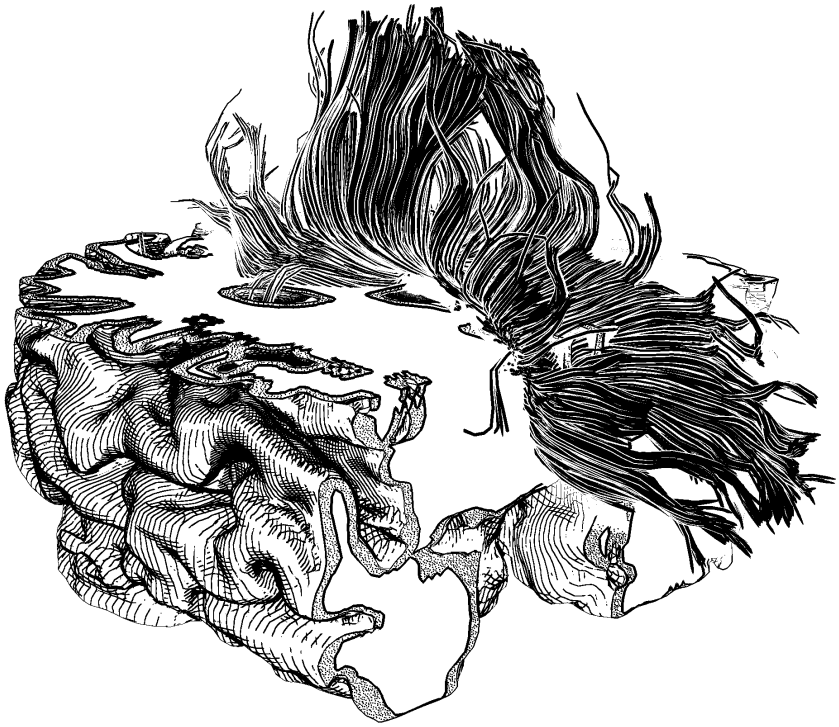


Master thesis

Illustrative Rendering and Multi-Touch Exploration of DTI data and Its Context

Pjotr Svetachov

University of Groningen

March 2010

Abstract

I present an interactive illustrative visualization method that is inspired by traditional pen-and-ink illustrations styles as can be found in medical textbooks. The main goal of this technique is to render DTI fiber tracts and the context around them. The context can consist of the brain surface, the skull, or object such as tumors. These surfaces are extracted from segmentation data or are generated with a fast iso-surface extraction method. Fiber tracts are rendered using the depth-dependent halos method while the context is rendered with a hatching style which is visually similar to that of the fibers. The hatching uses a real-time slice-based rendering method which is guided by ambient occlusion. I also provide a way to explore the context around the fiber tracts through a set of cutting planes where gray matter is indicated using stippling. All these methods are implemented using GPU techniques and, thus, work in real-time. However, care was taken to also be able to produce high quality images for print reproduction. I also investigate the possibilities of multi-touch interaction to explore the fiber tracts and the context. Using a intuitive frame-based interaction technique, the screen is stripped of clutter (such as control widgets and toolbars). An informal evaluation with domain experts assert the success of the methods developed.

Parts of this thesis have been published or are under review as follows:

- Pjotr Svetachov, Maarten H. Everts, and Tobias Isenberg (2010) DTI in Context: Illustrating Brain Fiber Tracts In Situ. Computer Graphics Forum, 29(3), June 2010.
- Lingyun Yu, Pjotr Svetachov, Petra Isenberg, Maarten H. Everts, Tobias Isenberg (2010) FI3D: Direct-Touch Interaction for the Exploration of 3D Scientific Visualization Spaces. IEEE Transactions on Visualization and Computer Graphics, 16(6), November/December 2010. (to appear)

Acknowledgments

I woud like to thank my supervisors Tobias Isenberg and Maarten Everts for guiding me through my thesis. The way they guided me really motivated me to try out new things and because of this I have learned a lot and I also have a feel that I accomplished a lot with this thesis.

Pjotr Svetachov
Groningen
November 22, 2010

Contents

Chapter 1

Introduction

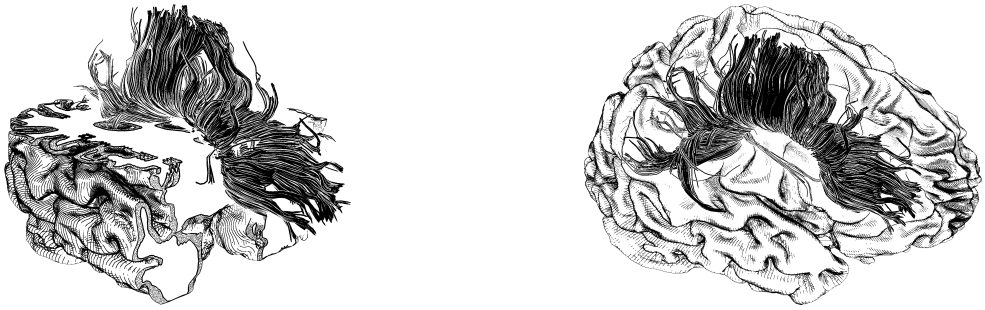
For a long time, people used illustrations in books to create scientific visualizations. Examples of these illustrations can be found in medical books or scientific papers (e.g., Dauber 2005, Catani and Thiebaut de Schotten 2008, Schmahmann et al. 2007, House and Pansky 1960). Traditionally, these illustrations are hand-drawn. Researchers have been trying to bridge the gap between hand drawn illustrations and computer-aided scientific visualization (Viola et al. 2005, Ebert and Sousa 2006) and illustration (Winkenbach and Salesin 1994, Ostromoukhov 1999). The reason for wanting to bridge this gap is that with hand-drawn illustrations authors can easily show important aspects of the data, usually done through abstraction and emphasis. An example of this type of rendering is shown in Figure 1.2. The authors were able to show parts of the brain's anatomy by using black and white colors only. The authors where also able to do this in a very clear way. I take those illustration as an inspiration for my rendering technique.

1.1 Illustrative rendering

Let us first analyze the images in Figure 1.2. The authors used different techniques to achieve their goal. One of the techniques is the use of hatching. Hatching plays an important role not just in these but in many illustrative renderings. By using different hatching patterns, an artist can make an distinction of the different objects. For instance, in the left picture one can clearly see a specific hatching pattern that is used for the outside of the brain, while in the right picture the artists use cross hatching to show the structure of the brain. Also, more occluded areas tend to have thicker strokes and also have more layers of hatching. This kind of effect is similar to ambient occlusion as used in rendering, so this technique could be used to guide the hatching.

Another important technique is stippling. The authors take a slice out of the data to show the inside of the brain, here they use stippling to show the gray matter regions. Because the drawing style of stippling is distinctive from hatching the viewer can easily see where the outer brain will end and the inner slice will begin.

The middle picture also shows fibers that connect different parts of the brain. These fibers are drawn as thin lines and have (where needed) a halo around them. The use of the halos is to emphasize the lines and to not let them melt together with the background. The goal of my research is to try to, similar to (Tietjen et al. 2005), combine different techniques to achieve the same rendering style as in the figure above. The rendering must also happen on interactive frame rates. I also try to produce only black and white images of high resolution as they are a better candidate to be used in print because this lets the printer control the half-toning (Hodges 2003). Results can as seen in Figure 1.1.

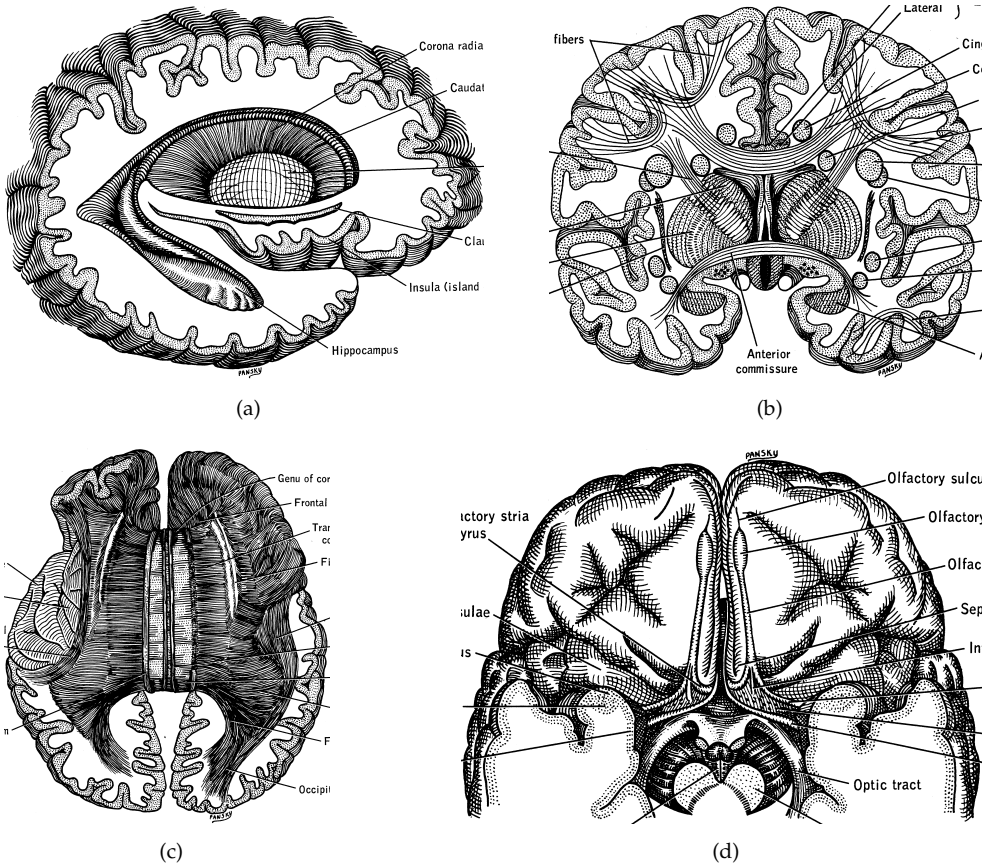


(a) (b)

Figure 1.1: *Results of my method.*

1.2 Multi-touch interaction

Another important aspect of my research is exploration of DTI data using multi-touch interfaces. For this I use a SMART Board multi-touch display that supports two touches. It is important to be able to do most aspects of DTI exploration on a touch screen. These aspects do not only include translation, scaling and rotation but also exploring into the data by slicing and selecting different fibers. The interaction must be as intuitive as possible and the interface must not contain much clutter. To accomplish all this I as inspiration the drawing canvas (Nijboer et al. 2010) as seen in Figure 1.3. In this Figure the user can manipulate the canvas by clicking and dragging on the frame borders around the canvas.



(a) (b)

(c) (d)

Figure 1.2: *Examples of hand-drawn pen-and-ink illustrations of the brain's anatomy, from (House and Pansky 1960).*

1.3 Organization

In Chapter 2 I discuss the related work on both illustrative rendering and multi-touch interaction. Based on the observation from Figure 1.2, several techniques are presented that are used in the Figure. These techniques include rendering of fiber tracts, stippling, ambient occlusion, hatching and the use of halos.

Chapter 3 explains my rendering method, first I describe how I extract surface information from MRI data. Then, I show how the techniques that are described in Chapter 2 are used in my program.

Chapter 4 describes my decisions for multi-touch exploration of the data. Inspired by the canvas from Nijboer et al., I also use a frame border. I explain how

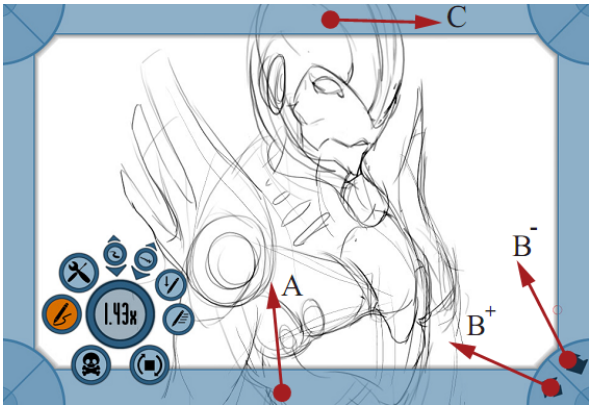


Figure 1.3: *Taken from (Nijboer et al. 2010): manipulating the canvas using a frame. The arrows show the different click and drag motions, A for is translation, B is for for scaling and C for rotation.*

I map the different aspects of DTI exploration to click and drag gestures using this border.

Chapter 5 includes implementation details. In this Chapter I explain on how all the techniques described earlier are put together. I will also describe some pitfalls, like memory management, that can occur when implementing the algorithms.

Chapter 6 has the results together with performance measurements. I show how my algorithms scale with different drawing styles and resolution.

Finally, Chapter 7 includes the conclusion. Also some suggestions for future work are also provided in this Chapter.

Chapter 2

Related work

In the previous Chapter I made the observation that the use of hatching, an effect similar to ambient occlusion, and stippling are often used in in medical illustration. Figure 1.2, for instance, shows the use of all these techniques in the same image. In this Chapter I will review these NPR techniques in Section 2.1. I also stated in the previous Chapter that I will make use of a multi-touch screen to support the intuitive exploration of DRI data. Multi-touch exploration is not a new concept and in the last few years a lot of research have been done on this concept. I will cover multi-touch exploration in Section 2.2.

2.1 Rendering techniques

For my research I try to combine many rendering techniques to produce certain style pen-and-ink illustrations. In this Section I will explain the techniques and provide examples of previous work.

2.1.1 Fiber rendering

Fibers tacks are bundles of lines extracted from DTI data (e.g., Mori and van Zijl 2002, Wakana et al. 2004). These give an indication of neural pathways in the brain. Rendering bundles of lines is a well researched area and there exists a number of different techniques. For example, tubes are often used, here color plays an important part to indicate certain properties like direction or magnitude (Post et al. 2002), an example is shown in Figure 2.1. I use Depth-Dependent Halos (Everts et al. 2009) for rendering the fibers. As seen in Figure 2.2 this method produces good looking and sharp visuals are inspired by the visuals in Figure 1.2. This method also uses halos to show depth relations. This method can also put emphasis on bundles of lines and can be used for on-screen rendering of rendering high resolution images for print. Apart from this, the method is also flexible, the width of the lines and the halos can be adjusted in real-time. Also filtering can be used to show only important lines.

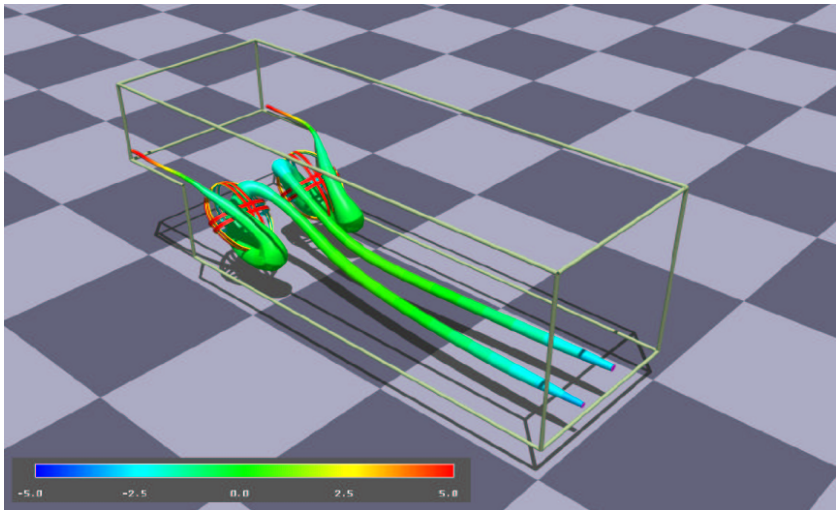


Figure 2.1: *From (Post et al. 2002). Example of using steamtubes. The radius of the tubes is inversely proportional to the square root of the local velocity magnitude, and the color of the tubes corresponds to the pressure.*

Figure 2.2: *From (Everts et al. 2009). Example of using depth dependent halos.*

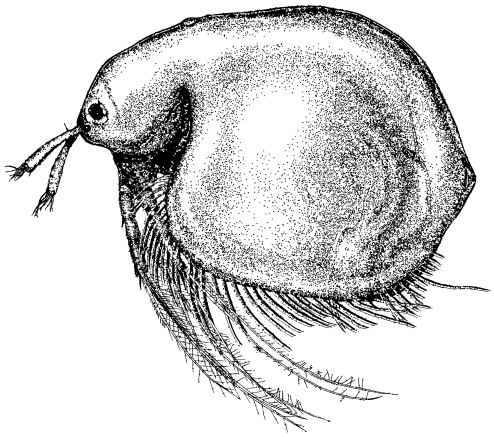


Figure 2.3: *From (Kim et al. 2009). Example of using stipples.*

2.1.2 Stipple rendering

Most work that has been done on stipple rendering has been mainly in generating stipple renderings from gray scale drawings (Kim et al. 2009, Kopf et al. 2006) as seen in Figure 2.4. Some have also tried to use stippling for volume visualization (Lu et al. 2003). There are also ways to stipple 3D objects (Meruvia Pastor et al. 2003) but this solution requires preprocessing the object, this gives a problem when object in real-time from data. A difficulty in rendering stipples is that the stipples need to be distributed evenly for the final rendering to be appealing to the user. To achieve a good distribution the object can be is tessellated (Lu et al. 2003), some statistical error measure are used (Kim et al. 2009), a dart throwing with gradually decreasing dart radius (Kopf et al. 2006) or Voronoi diagrams are used (Deussen et al. 2000). Most methods require computational time and are not suited for interactive applications thus for real-time stippling the stipples are often precomputed (Secord et al. 2002, Kopf et al. 2006). The method of Kopf et al. (2006) can convert gray drawings to stipple renderings, most of the computation time of this method is in its preprocessing step. This method can be adapted to 3D by parameterizing the mesh, in case of cutting planes parameterizing is very easy which makes this method a suitable for rendering stipples on planes.

Figure 2.4: *From (Secord et al. 2002). Example of using stipples.*

2.1.3 Ambient occlusion

Ambient occlusion (AO) is a shading method that takes into account the amount of occlusion of a rendered pixel when performing lighting calculations, occluded areas are colored black like seen in Figure 2.5. One of the first implementations of ambient occlusion is by Miller (Miller 1994) where he calculated how well an area could be reached by spheres and how well spheres could be fitted locally. This information was then used to shade the scene. Over the years, ambient occlusion got a status as an easy and a cheap way to simulate global illumination (Zhukov et al. 1998, Evans 2006). While global illumination has some other advantages like color bleeding, ambient occlusion can be easier implemented and is faster. The easiest way to calculate AO is to cast rays in random directions from a point and calculate how far the rays will travel before hitting an object in the scene, the more rays used the better the end results are. In the last couple of years, a new method called SSAO (screen space ambient occlusion) has been heavily used in games (Shanmugam and Arikan 2007, Landis 2002). The basic idea behind SSAO is

Figure 2.5: *A monkey head rendered in Blender using only ambient occlusion.*

the same as behind AO but SSAO is done screen space, SSAO is actually an approximation to AO. First, for each pixel sample the neighborhood pixels in the depth buffer then use this information to approximate the occlusion of the given pixel. The occlusion can be calculated in different ways but it is usually inspired by the normal AO where rays are cast. While this method is an approximation to AO, a lot of research has been put into developing algorithms to make SSAO indistinguishable from ray traced AO (Bavoil et al. 2008, Bavoil and Sainz 2009, McGuire 2010) at the cost of performance, one of the main drawbacks of SSAO is that object that are drawn in front of other objects will get an outline, even if the objects are far apart from each other and do not necessary occlude each other. To overcome this drawback, multi-layered depth buffers can be used. Because my data usually has only one object or in case of more objects like tumors, the objects are relatively close together, ordinary SSAO gives me good enough results.

2.1.4 Hatching and halos

Hatching is a well research topic and there exists a lot of different 2D (Winkenbach and Salesin 1994, Ostromoukhov 1999, Hertzmann and Zorin 2000) and 3D (Deussen et al. 1999, Praun et al. 2001, Zander et al. 2004, Ritter et al. 2006) techniques. For example, Praun et al. and Ritter et al. both presented a very good real-time technique that makes use of textures (see Figure 2.6), by blending them over each other one can generate different hatching densities. Deussen et al. presented a technique that is not real-time but has a very good line control, each line is treated separately. For my research I looked for a method that is both scale-dependent (Freudenberg et al. 2001, Salisbury et al. 1996), real-time and has a good line control so I can adapt

Figure 2.6: *From (Praun et al. 2001): Real-time hatchnig using textures.*

the rendering style to that of the drawn fibers if necessary.

I found the method by Deussen et al. to produce hatching that is similar to that of Figure 1.2. It also has very good line control because each line is treated separately. The method can also be adapted to make use of the GPU to become real-time.

Like hatching, halos are also a well researched topic (Appel et al. 1979, Elber 1995). Usually, halos are used to enhance depth perception or to set focus on special objects (Tarini et al. 2006, Bruckner and Gröller 2007) or to separate objects (Everts et al. 2009). A new method called depth-aware halos or depth-dependent halos has become a success; the properties of the halos change depending of depth to the objects behind them (see Figure2.2). Everts et al. use depth aware halos to separate halos by changing the side of the halos according to depth. Tarini et al. use halos to separate molecules by changing the color according to depth.

2.2 Multi-touch exploration

Multi-touch surfaces are a good way to interact with virtual 2D objects. Touch interaction, however, consist of a 2 degree of freedom (DOF) input. Interaction aspects as translation, rotation, and scaling in 2D also have 2 DOF. The combination of these interaction aspects will add extra DOF to the input. To bypass the limiting 2 DOF touch interaction gives researches use time-dependent techniques like gestures (Hancock et al. 2006, Kruger et al. 2005), for example Kruger et al. presented a method called rotation-and-translation where the rotation of an object is calculated based on movement over time. Widgets on objects are often used. The idea is that

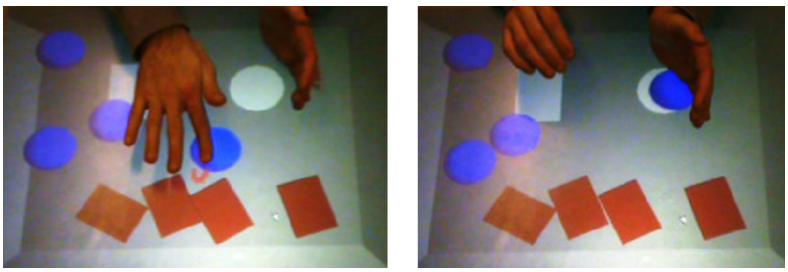


Figure 2.7: *From (Wilson et al. 2008): Throwing a ball from one hand and catching it with the palm of the other hand.*

little widget are drawn over objects, touching them will initiate an action. For example, small widgets can be drawn at the corners of an object, touching them will initiate a rotation. An example of a physics-based system would be where the user can throw a ball from one side of the screen to the other as seen in Figure 2.7 (Wilson et al. 2008).

Using multi-touch, more DOF can be acquired in the input. With two touches you can have 4 DOF. By using the techniques mentioned before with multi-touch, even more DOF can be acquired. For example, with the RTS (Hancock et al. 2007) method one can rotate, translate, and scale at the same time using two fingers.

A new technique, frame border interaction, has been used in the last year (Yu and Isenberg 2009, Nijboer et al. 2010). This technique adds more DOF by placing an interactive frame border around the scene as seen in Figure 1.3. The user can now choose to start his touch at the border, doing so opens new functionality. The idea is that it does not matter where on the border the user starts and the border is large enough so the user does not need to focus much on touching the border. This way the user will not get distracted by the border. In combination with intuitive gestures this technique can be very powerful. This is why I will investigate the possibilities of frame border interaction with exploration of DTI data and context.

Chapter 3

Rendering context

In the previous Chapter I made a review of techniques used in illustrative visualization. In this Chapter I explain how I adapt these techniques to achieve my goal. For example, I need to be able to extract iso-surfaces from volume data in real-time based on a given threshold, this is explained in Section 3.1. Then I need an zoom independent hatching method that is both flexible (e.g., I need to be able to control line width and density), works in real-time, and I must be able to apply this method do that iso-surfaces I just extracted without any preprocessing, this method is covered in Section 3.2. I also observed before that the hatching in some illustrative visualizations is guided by ambient occlusion. For this I explain in Section 3.3 how to calculate the ambient occlusion term that is used to modify the hatching style. Finally, I also observed how stippling is used to indicate gray matter. For this I need a stippling method that works in real time and can be applied to cutting planes. Section 3.4 explains how the stippling algorithm works.

3.1 Surface extraction from volume data

The first step in rendering context is to extract a surface from a given volume data. The easiest way of doing this is with the marching cubes algorithm (Wyvill et al. 1986, Lorensen and Cline 1987). The algorithm works by laying a grid over the volume. For each point on this grid, one takes the neighborhood 8 points which form a (imaginary) cube. Now, for each vertex, by looking if it is inside or outside of the iso-surface one can determine the polygons needed to represent the surface that passes through the cube. Lorensen and Cline already showed than only 15 different configurations are needed. These can be stored in an lookup table and because there are only 8 points the table has 256 entries.

I use the GPU to accelerate the algorithm (Johansson and Carr 2006, Tatarchuk et al. 2007). One can use geometry shaders to port marching cubes to the GPU. This is because each point of the grid can be evaluated independent from other points. The algorithm starts with a grid of points and these points are passed to the GPU. In the geometry shader, each point is evaluated: a imaginary cube is calculated and

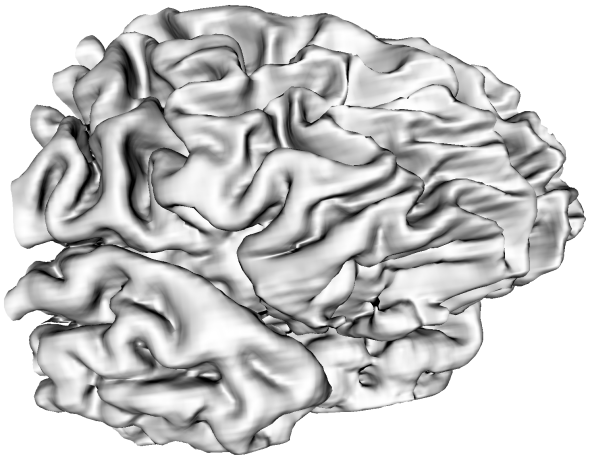


Figure 3.1: *Example rendering of only the surface.*

the polygons that represent the surface that passes through the cube is calculated by looking at each vertex of the cube and computing if this vertex is inside or outside the iso-surface. The maximum number of points needed for each cube is 16 so the whole lookup table can be saved in a 256 by 16 texture and passed to the geometry shader. The disadvantage of this method is that the output consist of a lot of triangles and connectivity information is lost. So lighting calculations that rely on vertex normals cannot be done without first connecting the polygons. For the remainder part of the algorithm, however, connectivity information is not needed so this will not be a problem.

After implementing this algorithm, I achieved frame rates of 5–6 fps on a Geforce 8800 GTX when recalculating the iso-surface each frame. I found that these were too low for any good interaction so I implemented a caching mechanism in my program. OpenGL has a special mode called transform feedback, in this mode you can fetch the output from a shader. I cache the output in a separate buffer and only change the buffer when the user changes the iso-value. I assume the user does not change the iso-value after choosing a correct one. Figure 3.1 has an example of a rendering of an iso-surface.

3.2 Zoom independent slice-based hatching on the GPU

I use a modified version of the hatching method by Deussen et al. (1999) to render the context. The algorithm works by slicing a model with a certain number of

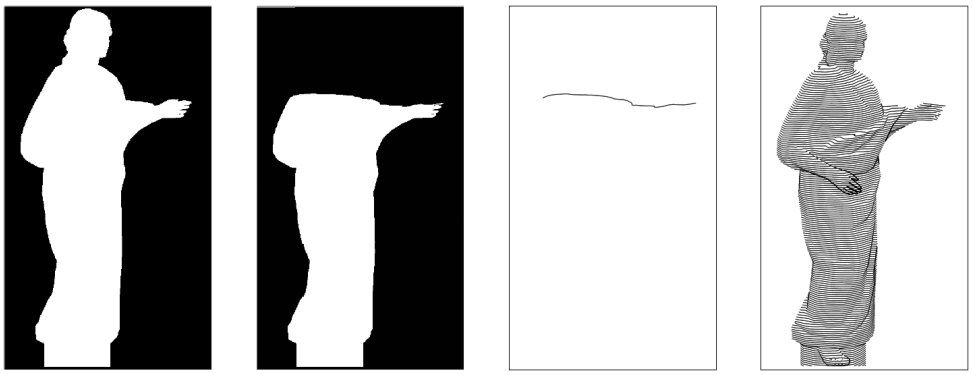


Figure 3.2: *Taken from (Deussen et al. 1999). OpenGL-based generation of intersecting lines: (a) The whole model rendered to a stencil buffer. (b) Clipped image rendered to a stencil buffer (c) Curve extracted by comparing the two stencil buffers. (d) The whole set of intersection curves.*

planes. The place where the planes intersect the model is painted black, giving an idea of a stroke. The planes are positioned on a curve that follows the skeletal axis of the model, this gives the end result a less artificial appearance.

To get the intersection between the model and the planes, Deussen et al. take advantage of graphics hardware. First, the model is drawn to a stencil buffer (see Figure (a) of 3.2), then the model is drawn again to a second stencil buffer and, this time, one of the planes is used as a clipping plane (see Figure (b)). From both buffers the pixels along the boundaries are calculated. These two boundaries, when subtracted from each other (Figure (c)), form a curve that indicates where a stroke needs to be placed. This process is repeated for all the planes (Figure (d)).

This method produces hatching strokes that can mimic those seen in Figure 1.2 if stroke shading is applied correctly. However, there are two disadvantages to using this method directly in my program. First, drawing the model for every stroke is expensive and thus the method does not work in real time. Second, the method relies of knowing the skeletal axis of a model which is found using an edge collapse algorithm on the original model. The iso-surfaces extracted from the volume using the algorithm from Section 3.1 do not have any connectivity information and extracting the skeletal axis would require some computational time.

Therefore, I modified the algorithm to remove these two disadvantages. The new algorithm uses deferred shading (Deering et al. 1988, Hargreaves and Harris 2004) to remove the need to re-render the scene and bend planes to remove the artificial look. I will first explain how the algorithm works when using regular planes and,

then, I will show how to modify the algorithm to take into account the bending of planes.

Imaging a model that is sliced using a set of planes. These planes are all parallel to each other (thus have the same normal N) and two neighboring planes have the same distance d to each other. Now, I want to know if a certain fragment F_i at position P_i belongs to the intersection of the planes with the model. To check this, first the plane that should cut the fragment is found and then one looks if one of the fragments in the neighborhood of F_i is on the other side of this plane. Thus in the case we have an intersection. To check for these intersections, we perform the following. We calculate the distance $d_p = P_i \cdot N$ of F_i to the defining plane located at $(0, 0, 0)$. Then, we calculate the distance d_l from F_i to the closest plane below the fragment as $d_l = d_p \mod d$. Now, we can compute the distance of this plane to the defining plane as $d_c = d_p - d_l$. We check in the 8-neighborhood of F_i to see if one of the fragments is below that plane. If this is the case then the fragment is part of a stroke. This process is shown in Figure 3.3, here the black box represends the fragment tested, the circles represent the 8 neighborhood. Circles above the plane are colored black and circles below are colored white. More layers of hatching lines can be added by added more sets of planes that point into other directions. This will simulate cross-hatching.

As said before I use deferred shading. First the scene is rendered to a G-buffer (Saito and Takahashi 1990), the result can be seen in Figure 3.4. Then, this buffer is bound as a texture and mapped on a screen-filling quad. Then, a pixel shader is used to draw the lines. The positions are stored as 32 bit floats, experiments show that storing the positions with less precision will produce jagged lines due to round off errors.

To remove the artificial look, bend planes are used instead of normal ones. To enable the use of bend planes, a modification to the distance function is needed. I tried several different functions and the results are displayed in Figure 3.5. The fact that the brain itself has a lot of bends also helps removing the artificial look. Experiments show that the function to use depends on the dataset and it is up to the user to decide which one is the best looking. When designing a distance function, there is a point where care must be taken. The distance formula usually has one fluctuating variable, in my case this is F_i. The formula must be remain "steady" when this variable changes. What I mean with steady in this case is that when the distance increases the formula must also increase and do not make sudden jumps. Also, the increase must be near linear, otherwise the distance between lines will fluctuate and this will show in the result as one part of the surface will have more lines than another as seen in the last example of Figure 3.5.

Using this approach, all the lines will have the same thickness regardless of the

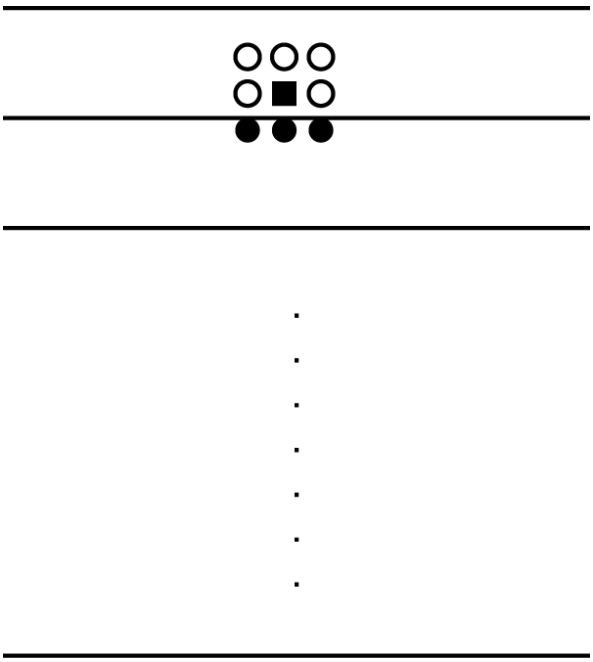


Figure 3.3: *The process of determining which fragments to color.*

angle of the surface with respect to the planes which would be the case if textures were used or if only the distance of F_i to the closest cutting plane was used without taking into account the neighbor fragments (Leister 1994), this is good because I have complete control of the thickness in this way. This thickness is controlled by the size of the neighborhood in which we sample the 8 neighbor fragments. When expanding the neighbor size, we can still use only 8 neighbor fragments, thereby not effecting the performance. Theoretically, we can get artifacts because of our low sampling rate but these are often not visible or distracting. One could even argue that they make the images less artificially looking.

When zooming in and out, there are two options. The first is changing the size of the lines, this gives the idea of actually zooming in and out. Another possibility is to change the distance between the lines, this preserves the drawing style of the image at different scales (Freudenberg et al. 2001, Salisbury et al. 1996). After experimenting with both approaches I found that adding more lines produced better results. The way I implemented this is by setting d as linear function of the zoom level. Theoretically, this will produce swimming lines and a better solution would be to blend in new lines as zooming takes place. After experimenting, however,

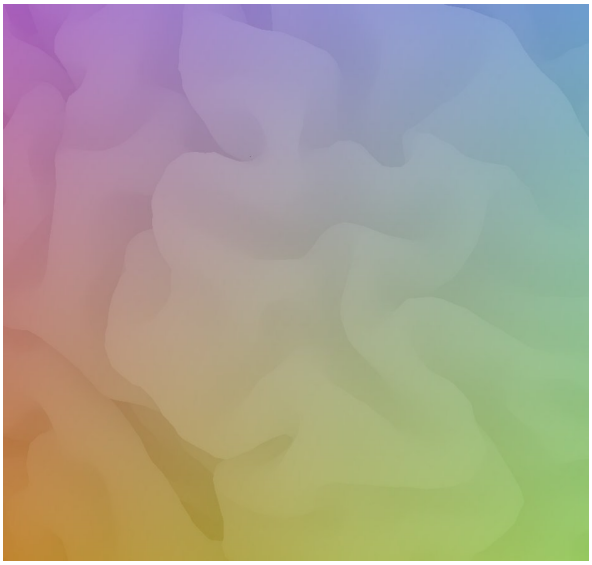


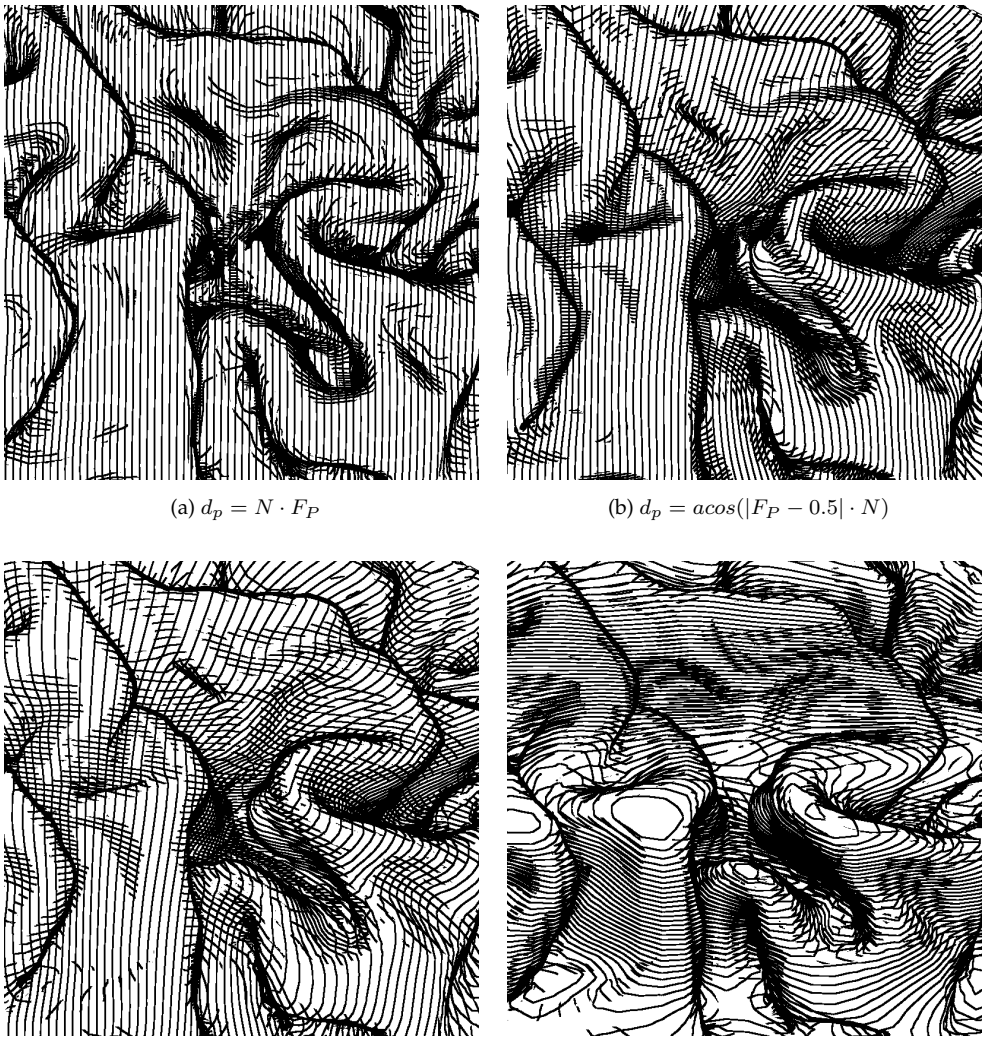
Figure 3.4: *Using a G-Buffer to store the position of the samples.*

I found that swimming of the lines is not noticeable under normal circumstances. Only when dealing with thick lines that are far apart one can notice the effect of the swimming lines. Figure 3.6 has examples of different zoom levels.

The above method has some limitations with respect to the algorithm of Deussen et al. First, the planes are aligned in the same direction. Second, some suitable distance functions need to be made. But the advantage of the method are that it is very flexible. Lines can be made of any width and the width can be controlled for every fragment. All this control is done in real-time as there are no texture used of any kind and no pre-calculation is done. Also there are no requirements for the objects themselves. There is no need to parametrize a surface or to calculate any skeleton or curvature information. The algorithm works on a set of triangles without any connectivity information (polygon soup), the speed of the algorithm also does not depend on the object itself but on the screen resolution.

3.3 Screen space ambient occlusion

As explained in Section 2.1.3, Ambient Occlusion (AO) is a shading model that takes into account how much a point is occluded. AO can be used to approximate global illumination. AO is used as the only lighting model in my program. After studying



(a) $d_p = N \cdot F_P$

(b) $d_p = acos(|F_P - 0.5| \cdot N)$

(c) $d_p = acos(cos(|F_P|) * (|F_P - 0.5| \cdot N))$

(d) $d_p = w_1(N \cdot F_P) + w_2(sin(F_P.x) + w_3(F_P.y) + w_4(F_P.z)$ where $w_1, \ldots w_4$ are weights that sum up to 1

Figure 3.5: *Examples of distance functions. Here F_p is a vector showing the position which is normalized between 0 and 1 for x, y, z. I substract 0.5 from these components to bend planes around the center of the volume.*

several versions of SSAO, an approximation to AO, I choose the go with the fast implementation in favor of a slower more accurate one. The reason for this is that

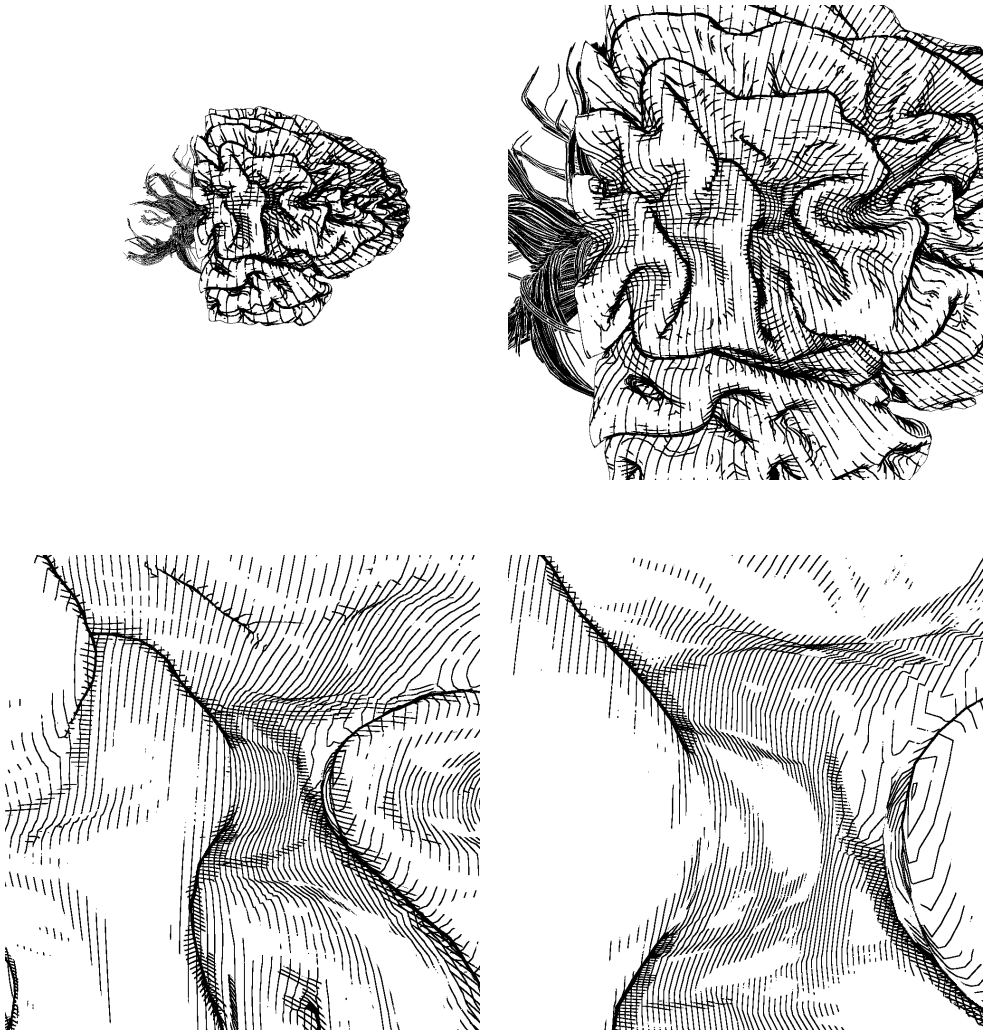


Figure 3.6: *Example of different zoom levels.*

ambient occlusion is used for guiding my hatching algorithm. Hatching itself will not benefit from the small details better SSAO implementation will give. The main idea for using AO was to achieve the same look as in the right image of Figure 1.2, in this image cross hatching is used on the brain to indicate where the sulci are. To achieve the same effect a simple SSAO method will suffice.

My method is based on the basic principles of SSAO that are described in Section 2.1.3. I use a G-Buffer to store the depth and normal of each pixel. The algorithm works with one pass over this buffer using a fragment shader. This shader samples 16 random nearby fragments and uses the differences in the depth D_d and the angle of the normal D_a (the difference of the normals is calculated using the dot product) compared to the original fragment to calculate an AO term. The theory is that the greater the difference the more the point must be occluded. Only using the depth will produce an effect that can be best described as ghosting. It will look like the same image is duplicated and offset a few times and blended over each other. Also, taking into account the normal removes this effect. The formula in my implementation is:

$$AOterm = \sum_{i \in LT} (1 - D_a(i)) * D_d(i),$$

where LT is a lookup table with 16 random offset vectors.

Most SSAO implementations use a Gaussian blur in a second pass to remove any artifacts and to produce a smoother image. I found out that this pass is not necessary, hatching will not benefit from this extra pass.

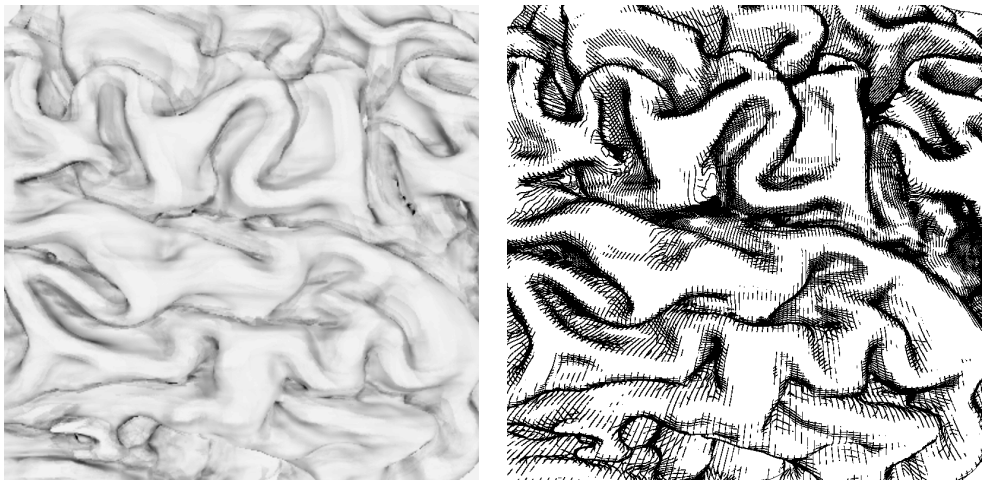
Like explained earlier, AO is used to guide the hathing. In practice, more layers of hatching directions are added in dark areas. The line thickness also changes depending on the AO term; darker areas are drawn with thick lines while less darker with thin lines.Figure 3.7 shows examples results for this process.

3.4 Slices and zoom-independent stippling

In Figure 1.2 stippling is used to indicate gray matter. In the figure the stipples are all of the same size and have the same distance with respect to each other. I try to achieve the same effect.

Slices are used to explore the inner parts of the brain. The slices are drawn as quads. When drawing the slices a fragment shader uses the volume texture to see if a given fragment is inside or outside the iso-surface. Fragments on the inside are colored white and outside are discarded. Fragments on the boundary are colored black to simulate gray matter. This method is only used when no information about gray matter is available.

I also have access to segmented data showing gray matter. To render gray matter, I take my inspiration from Figure 1.2 and use stippling. Stippling is distinguishable from the hatching drawing style and, therefore, very suitable for showing areas of interest. Just as for the hatching my goal for the stippling was to have a method that is flexible, fast, and zoom-independent.



Ambient occlusion. Using AO to guide hatching.

Figure 3.7: *Example of using AO to guide hatching.*

For the stippling I make use of the method by Kopf et al. (2006). The algorithm by Kopf et al. uses of recursive Wang tiles. Wang tiles are a special set of tiles that can tile a plane in a non-periodic way. Kopf et al. use a recursive set of tiles which makes infinite zoom possible. Each tile has (apart from a number of subtiles) a set of points associated with it. The distribution of the points is chosen in a special way. Points from all the possible neighbor, parent, and child tiles were considered when distributing the points. This is so the points will look uniformly distributed and no gaps are seen when adding two or more tiles together. Each point in a tile has a rank. When rendering, the Wang tile hierarchy is is walked from top to bottom and, depending on the zoom level, the walking stops at a certain level. To assert if a point of a Wang tile should be drawn or not, the zoomlevel, point rank, and intensity of the image at the spot of the point are taken into account. Because all points in a tile have a different rank the points are added one by one when zooming in. The generation a tile dataset itself takes some amount of time and Kopf provides a dataset on his website that I used for my program.

To render the points, I first render the slices only to a texture (see (a) in Figure 3.8. On this texture the areas where the gray matter is shown are colored gray, the gray matter belonging to different slices has a different intensity of gray, this way I can store all the slices in one image. For each slice I map a set of Wang tiles to the plane of that slice. The CPU is used to walk along the Wang hierarchy and compare the Wang tiles against the view frustum. The stipples are drawn as OpenGL point

(a) Slice texture. (b) End result.

Figure 3.8: *Example of stippling. First a slice texture is generated, this texture is then used to apply the stippling.*

sprites and a fragment shader uses the texture to determine if a point is inside the gray matter area of the appropriate slice. If it is then the stipple is drawn. The stipples are always turned toward the camera and remain round at any angle, just like in Figure 1.2. The end result can be seen in Figure 3.8 and the effect of zooming in Figure 3.9.

3.5 Summary

In this Chapter I explained the three different techniques, hatching, stippling and AO, I use for visualizing context for DTI fiber tracts. The important goals for my method is that it is applicable to extracted polygon meshes from volume data, have a style that is similar to that seen in illustrative textbooks and on top of that the method needs to be real-time. The techniques discussed in this chapter perform in real-time, it is also possible to apply the techniques to iso-surfaces extracted from volume data. It is therefore now possible to extract a mesh from MRI data and draw it using hatching lines. It is also possible to cut the mesh using slices that show gray matter areas using stippling. The next step is to combine these techniques. My method uses AO to guide the hatching process and everything is implemented as a shader that works in screen space on the G-Buffer. How I do this will be discussed in Chapter 5.

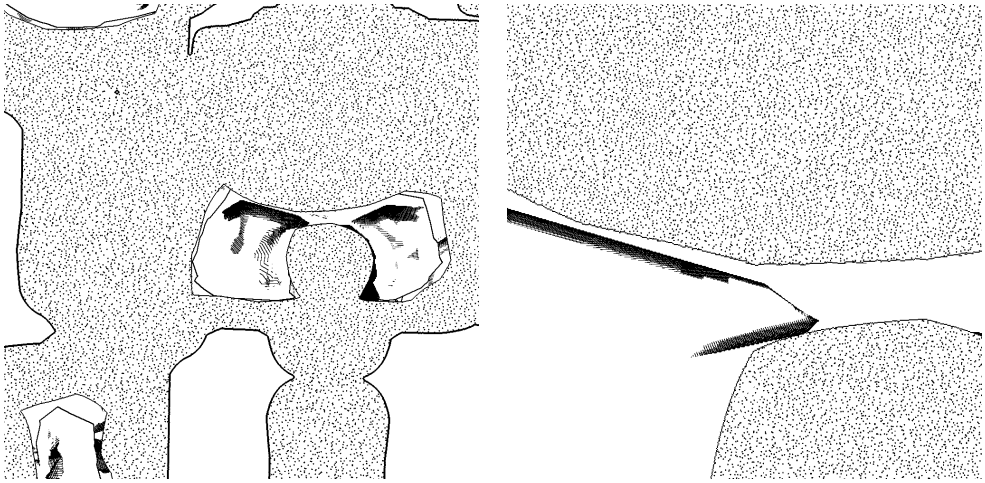


Figure 3.9: *Example of zooming and stippling.*

Chapter 4

Exploration of DTI data

In Chapter 1 I explained the need to be able to explore that data using multi-touch displays then in Chapter 2 I looked at the current research in multi-touch exploration. One method that looked promising was the use of a frame border. By using a frame border it is possible to add extra DOF to the application thus removing the need to extra widgets like toolbars. In Section 4.1 I explain what a frame border is and show an example.

After this I will show how to map common interaction aspects like translation, scaling and rotation in Sections 4.2 and 4.3. For this I will take inspiration of how Nijboer et al. (2009) have used a frame border in their application seen in Figure 1.3. There are some interaction aspects that are very specific to my application. One of these is managing cutting planes. How I map this aspect to the frame border is discussed in Section 4.4. Another aspect that is specific to this application is fiber selection, the user needs to be able to only select a few fibers. How I map this is explained in Section 4.5.

4.1 The frame border

In my program I use a frame border to implement interaction. A frame border was inspired by the method of (Nijboer et al. 2010, Yu and Isenberg 2009). The concept of a frame border is to have an additional border around the scene that is used to initiate gestures from. The idea is that the use of this border reduces interface elements like widgets. It must be noted that a border does not act like a toolbar and does not consist of a collection of buttons. With a toolbar the user needs to concentrate and move the attention to the bar itself when selection an option. The frame border acts like a big button and it does not matter where the user touches the frame border. The idea is that once the border is touched, the user can make a gesture that will indicate what he/she wants to do. The border in my program consist of 8 areas only. The four corners and the four planes of the screen, Figure 4.1 shows and example.

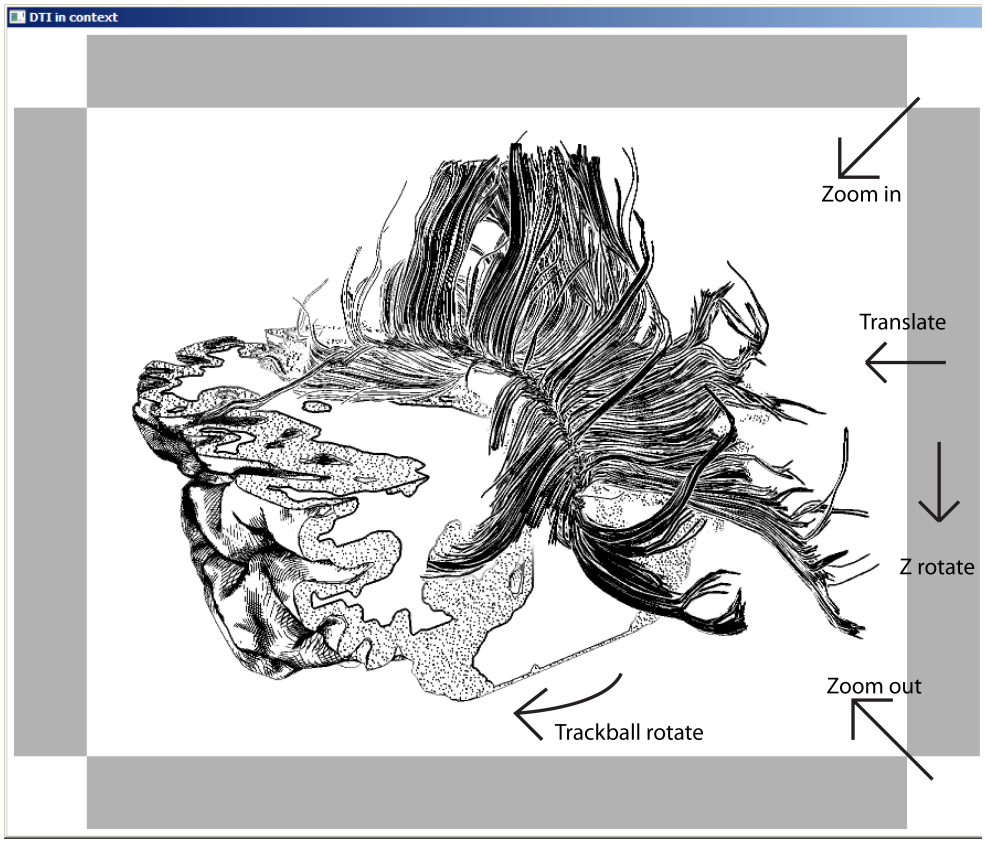


Figure 4.1: *Example of an interface with a frame border.*

4.2 Translation and zooming

Translation is initiated at the frame border. The user has to touch one of the four borders and then move his/hers finger perpendicular to the frame border to initiate the translation. Once initiated, the user can continue to translate the scene as long as he/she still touches the wall display with only one touch. The amount of x-/y-translation is calculated by taking the x and y distance from the point at which the hand is at to the point the translation is initiated at. We use all the four borders because, when initiating a translation at the frame border, there is only room to move your hand into 3 different directions as the screen border blocks the last direction.

Zooming is initiated at the corners of the frame. The user touches one of the corners of the frame and starts moving his/hers hand. Using the two bottom corners

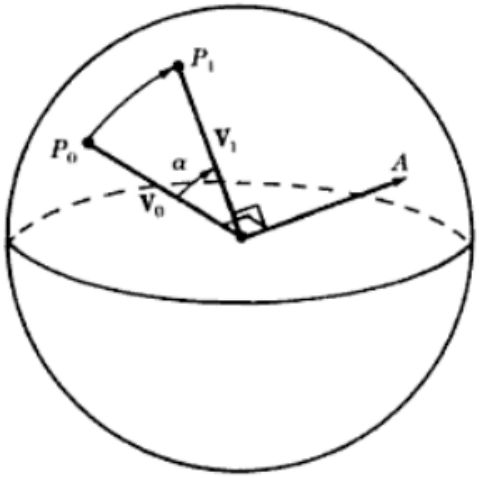


Figure 4.2: *From (Hultquist 1990). An illustration of the workings of a trackball.*

and moving the hand to the center of the screen will zoom out while the top corners zoom into the scene. This is because like with the translation the physical screen border will allow our hand to only be moved to the center of the screen. At any time the hand is still touching the screen the amount of zooming to be carried out is calculated by taking the Manhattan distance from the starting position of the touch to the position the hand is at at the time of the calculation. This is important because using the Euclidean distance will present different results under different angles between the starting position and the position of the hand.

4.3 Rotation

I use two types of rotation. The first is trackball rotation (Hultquist 1990) and second is z-rotation. The concept of a virtual trackball is that there is an invisible sphere that encapsulates the scene. When the user touches the scene the point P_0 on the sphere that the user touches is calculated. Then when moving his or hers finger, the point below his finger P_1 is also calculated, then the two angles at which to rotate the sphere so P_0 moves to P_1 are also calculated (see Figure 4.2). These are the angles the whole scene is then rotated by. To initiate the trackball, the user has to touch somewhere inside the frame border.

The initiation of z-rotation is performed the same way as that of the x-/y-translation except that the user now needs to move his hand parallel to the frame border in-

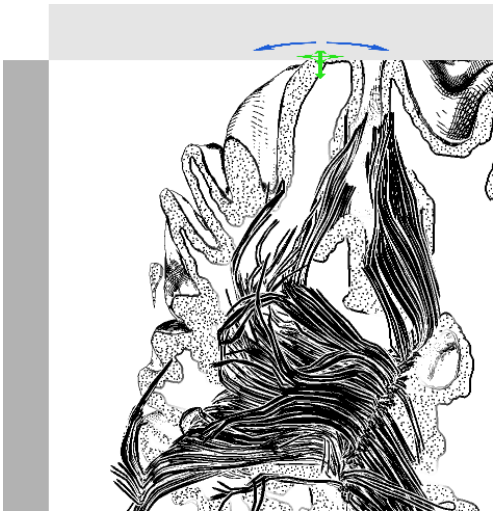


Figure 4.3: *An icon indicating to the user of how to move his hand to initiate x-/y-translation or z-rotation.*

stead of orthogonal. To help the user out a little icon is displayed as soon as the user touches the border as seen in Figure 4.3.

4.4 Managing cutting planes

Meyer and Globus presented a method (Meyer and Globus 1993) to manipulate cutting planes with single touch using widgets assigned to the cutting planes. With this method, cutting planes could be positioned, rotated, and scaled at the cost of adding widgets to the screen. For our purpose, the cutting planes only need to be dragged in three different directions and I have multi-touch capabilities. I also try to avoid clutter on the screen. My method, therefore, is gesture-based. The user starts with the first touch at the frame border, then the user makes a second touch inside the frame. Now, the gesture for moving the cutting planes is initiated. The second task the user needs to do is to select which plane to move. This is done by dragging the second finger in the direction of the plane the user wants to move. These directions are view dependent. To assist the user, three lines are drawn on the place of the second touch to indicate to the user in which directions the user needs to move his or her finger to select a plane (see Figure 4.4). When a plane is selected, it can be controlled by further motion of the finger in the same direction as was used to select the plane. Sometimes it can happen that the view is aligned in such a way that one of the planes is almost parallel to the view (that plane is almost parallel to

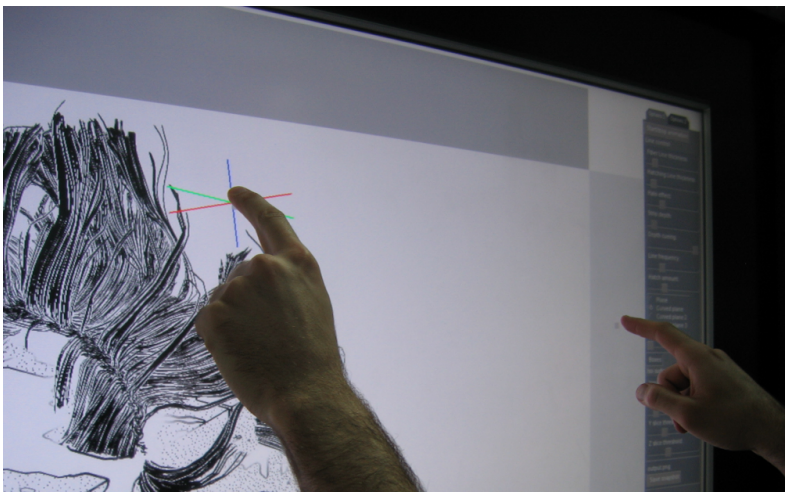


Figure 4.4: *An example of managing slices.*

the near or far cutting plane of the camera). In this case, slightly rotating the camera can make a huge difference on the direction of that plane and it also becomes hard for the user to guess which direction the plane is pointed to. Because of this I disable the manipulation of that particular plane in this view and draw only two lines corresponding to the other two planes. It can also happen that two planes point in almost the same direction on the screen in this case the manipulation of the plane that, of those two planes, is most parallel to the our view, is disabled.

4.5 Fiber selection

By double tapping the screen the user can enter a mode where fiber selection can be done. All the other gestures still work in this mode but this mode has several extra functions. Here, the user is presented with two boxes and only the fibers that go through both of these boxes are shown. By touching a box, the side of the box that is touched will light up and the box can be moved in the direction of the side that is touched, this enables the user to move the boxes in any direction and independent of the current view point. While this approach works and is highly accurate it takes too long to position both boxes with this approach. Therefore, I introduce another approach to quickly position the boxes. In this approach a gesture is initiated by touching the screen at two different places (see Figure 4.5). When this happens, a ray is cast for both touches and intersection points are calculated for those rays with

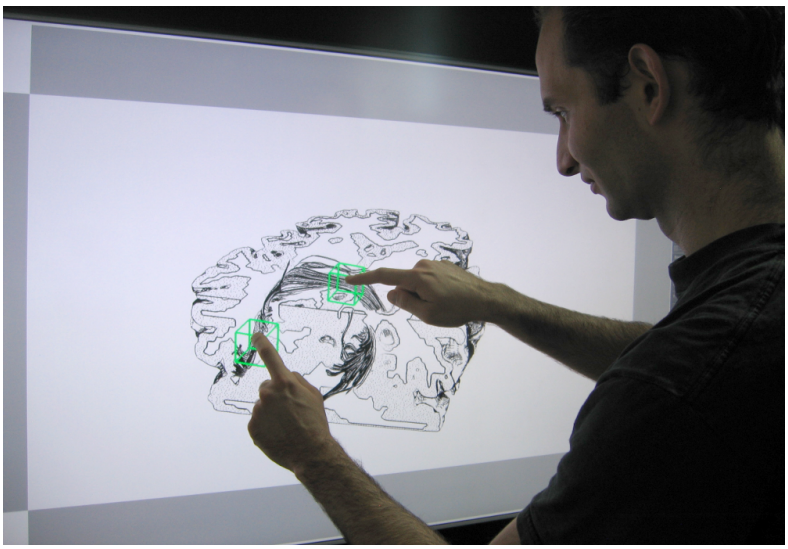


Figure 4.5: *An example of box selection.*

the cutting planes. For each ray the closest intersection point is picked and the boxes are positioned to those intersection points. The idea behind this second approach is that the user can roughly position the boxes by pointing out two points of interest on the cutting planes and then use the first approach to position the boxes more accurately.

4.6 Summary

In this Chapter I explained how a multi-touch screen can be used to explore DTI data. By using a frame border is it possible to map the important aspects like translation, rotation, scaling, managing cutting planes and filtering of fiber tracts to a multi-touch screen that supports two touches. To see if this frame border is effective in practice a small evaluation is done with an expert in neural tractography. The findings of this evaluation are presented in Chapter 6.

Chapter 5

Implementation

In this chapter I talk about the technical aspects of the program. First, I explain how the box filtering is implemented in Section 5.1. Then Section 5.2 explains how to put everything together and how to (de-)emphasize context. Then, Section 5.3 includes the subtle optimizations like memory management and the structure of the shader code. Finally, Section 5.4 explains the important aspects on programming for the multi-touch interface and the problems I came across and the solutions to these problems.

5.1 Box filtering

As explained before I use boxes to select certain fibers. For this I have implemented the method by Blass et al. (2005). This method uses a special version of an octree to quickly search the list of fibers that pass though a box. I have implemented this method straight without any modifications to it. The result is a method that can filter 11000 lines in real-time.

5.2 Combining the different methods

In Chapter 3 I described different stages of the rendering pipeline. How all these stages can be put together can be seen in Figure 5.1. I use separate shaders to construct the isosurface, the different G-buffers, the stipple mask, and the fiber rendering. Then, I use one fragment shader that combines all the buffers into a final rendering. I do not have a separate buffer for the hatching pass because of performance considerations outlined in Section 5.3. Also this allows me to use the position of the fibers to add extra elements to the final composition that will (de-)emphasize context. To support anti-aliasing, all buffers are rendered two times as large as the screen resolution and are sampled down when the final image is rendered.

When combining the fibers with the context, a halo is drawn around the fibers. For each pixel that is part of the context, the halo is calculated by taking a 5×5 uniformly sampled grid around the pixels neighborhood and counting how many sam-

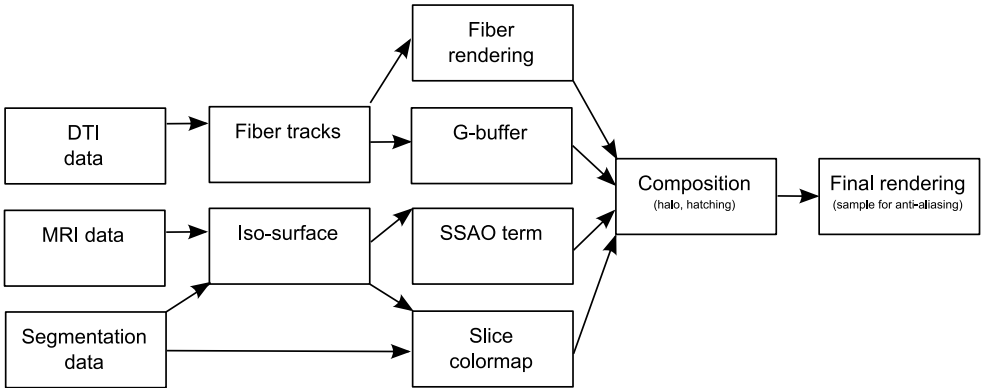


Figure 5.1: *This diagram shows how the different stages of the process relate to each other.*

(a) Halos and line thinning.

(b) Making the conext gray.

Figure 5.2: *Different ways to emphasize fibers and context.*

ples are part of the fibers. The typical use of halos is to have a white or black outline around the object. Instead, I use the halos to modify the width of the hatching lines that pass though the halo. The further the line is inside the halo the thinner it becomes till the line is not visible at all. This gives a very good-looking effect because instead of just turning gray and then while the lines actually become dashed and then disappear. This effect is demonstrated in Figure 5.2(a). Another way to emphasize the lines to color the context gray, this is demonstrated in Figure 5.2(b).

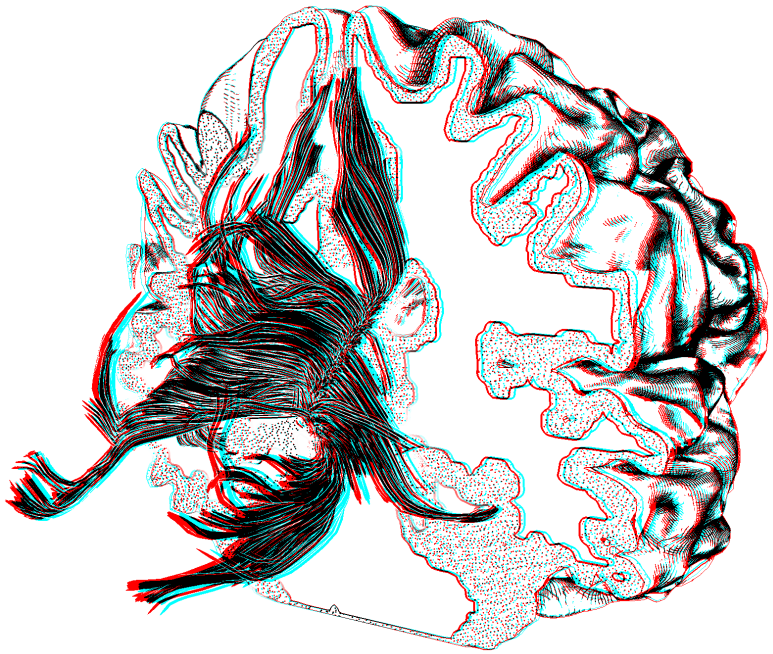


Figure 5.3: *An example of anaglyph rendering.*

Following the example by Everts et al. (2009), I also added anaglyph rendering to my program. Anaglyph rendering does not preserve color very well. However, because all the renderings are in black and white my illustrations work nicely with anaglyph rendering. I implemented anaglyph rendering by taking two snapshots of the scene, one for each eye and blend them together in a fragment shader. Figure 5.3 shows an example of anaglyph rendering.

5.3 Optimization and efficiency

The method presented in this thesis heavily relies on the GPU and under large resolutions, can show fill rate problems. Because of this I optimized the shader code to take as little instructions per fragment as needed and exit the shader as soon as possible.

Most of the process happens in the final shader where everything is put together. This shader uses a lot of instructions because both fibers, iso-surface and slices are combined, halos are added and hatching lines are drawn. Both the halos and hatching lines take up a lot of GPU processing power; it is therefore important to calculate things in a certain order. First, I look if a pixel belongs to the fibers, iso-surface or slices. If not, I exit the shader immediately. If the pixel belongs to the fibers I draw the fibers and exit the shader. If the pixel belongs to the slice I calculate the halo term for the pixel and draw the slice texture for that fragment. Finally if, the pixel belongs to the iso-surface I calculate the halo and the hatching. Doing it this way makes sure no extra calculations are done (e.g. calculating halo terms for pixels that do not need halos).

5.3.1 Memory management

Memory management is a very important aspect. While new graphics cards have 1GB of ram there are some pitfalls that will lead to using up all of that memory. In this section I outline possible pitfalls and solutions.

In earlier versions of my program a lot of memory was used for the buffers that are created for the iso-surface. The buffers need to be made before the transform feedback operation is carried out. It is very easy to assign too much memory to the buffers. The marching cubes algorithm uses a grid of points and for every point polygons belonging to the iso-surface are outputted. The maximum number of polygons per point on a grid is 16. Allocating the space for all the polygons can cost a lot of memory. For instance, in a grid of size 128 one would expect that the maximum number points one can get back is $128^3 * 16$. Every point consist of four (x, y, z and w) 32 bit values so the total memory need is $128^3 * 16 * 4 * 4 = 536,870,912$ bytes. Having two iso-surfaces (like a brain and a skull) will already not fit into the memory of most graphics cards. I found out that with all datasets I could safely assign 4 times less memory because most of the imaginary cubes lie completely inside or outside the iso-surface, thus producing no output.

Textures also tend to take up a lot of memory. As an example, having anti-aliasing enabled on a display with a resolution of 1920×1080 will have a viewport with a resolution of 1755×1068 and will require textures with a resolution of 3510×2136. A texture can have four channels each holding a 32 bit value. Storing such a texture requires $3510 * 2136 * 4 * 4 = 120$ megabytes of RAM. So one should take care when assigning textures. I did some experimenting with texture sizes and the result can be seen in Table 5.1. Notice that, to render the scene with z-buffering enabled, OpenGL requires a special texture format for the depth buffer and I was not able to read this texture inside a shader so I had to store my own depth in a separate

Table 5.1: *Memory used by different buffers. The resolution of the non anti-aliasing buffers is 1755 by 1068 pixels.*

	Bytes per pixel	Memory used in bytes
Isosurface G-Buffer		
Position buffer	12	112460400
Normal + depth buffer	8	74973600
Depth buffer	4	37486800
Slice G-Buffer		
Color+ depth	4	37486800
AO buffer		
Color	1	9371700
Line renderer G-Buffer		
Color	8	74973600
Depth buffer	4	37486800
Final result		
Color	3	28115100
Total		412 megabytes

channel to use later. These memory values are for both the anti-aliasing and non anti-aliasing buffers added together. This scenario is for the full-screen resolution of the wall display.

5.4 Interaction

The touch screen used is a SMART Board overlay on top of a large flatscreen LC display. The touch screen senses touch with four cameras in the corners. It can sense up to 2 touches simultaneously and calculate the size of the touches, it can also sense whether a finger is actually touching the display or hovering just a few millimeters above it. The hovering detection is not very stable so I did not use it. The touch screen has the advantage that is is very fast and that the LCD screen behind it can be any display. The disadvantage is that the use of four cameras have some limitation on the functionality. First the size of the touch is not calculated correctly but approximated, this is because calculating this is based on the presumed convex polygon that is seen by the four cameras. The calculated size can change for the same hand position where it is in the center of the screen or at the border. Because of this I did not use the size of the touches in my interface. I also encountered a bug where the positions of the touch were not calculated correctly. It was assured to me

that this bug would be eliminated in the next firmware release.

One issue noticed directly when experimenting with the touch screen is that some heuristics are needed to produce a robust interface. For instance, it is common to accentually release the screen for a split second. The interaction system needs to take this into account.

The design of the system consist of a state machine. The states of a machine can be one of the final states like z-rotation or moving the x-cutting plane. A state can also be an intermediary state like a state where the user has to choose where to do x-/y-translation, x-rotation or cutting plane manipulation (called S_{trc} from now on). As an example the user begins in a state where nothing has happened. When the user touches a frame with the first hand, the state changes to the S_{trc}. If the user makes a second touch on the center then then the state changes to the state where the user selects which cutting plane to manipulate. The system also has an invalid state (for instance when touching two frame boards at once). The user can not only progress forward but also backwards, for instance if the user is selecting which cutting plane to manipulate and suddenly releases the hand that is in center of the screen, then the system checks if the other touch is still on the frame board, if it is the system goes back to S_{trc} It is also possible to return from an invalid state to another state. Using this approach, the user does not need to start over when he/she does something wrong.

5.5 Summary

In the previous Chapters I described different methods I am using in my program. Combining these methods and adjusting them to each other required carefull consideration of sequence, interaction, and performance aspects of these individual techniques. In this Chapter I have provided with a description on how to combine the different methods. I also described the different pitfalls one can have when combining the methods. Also some solutions were given to these pitfalls.

Chapter 6

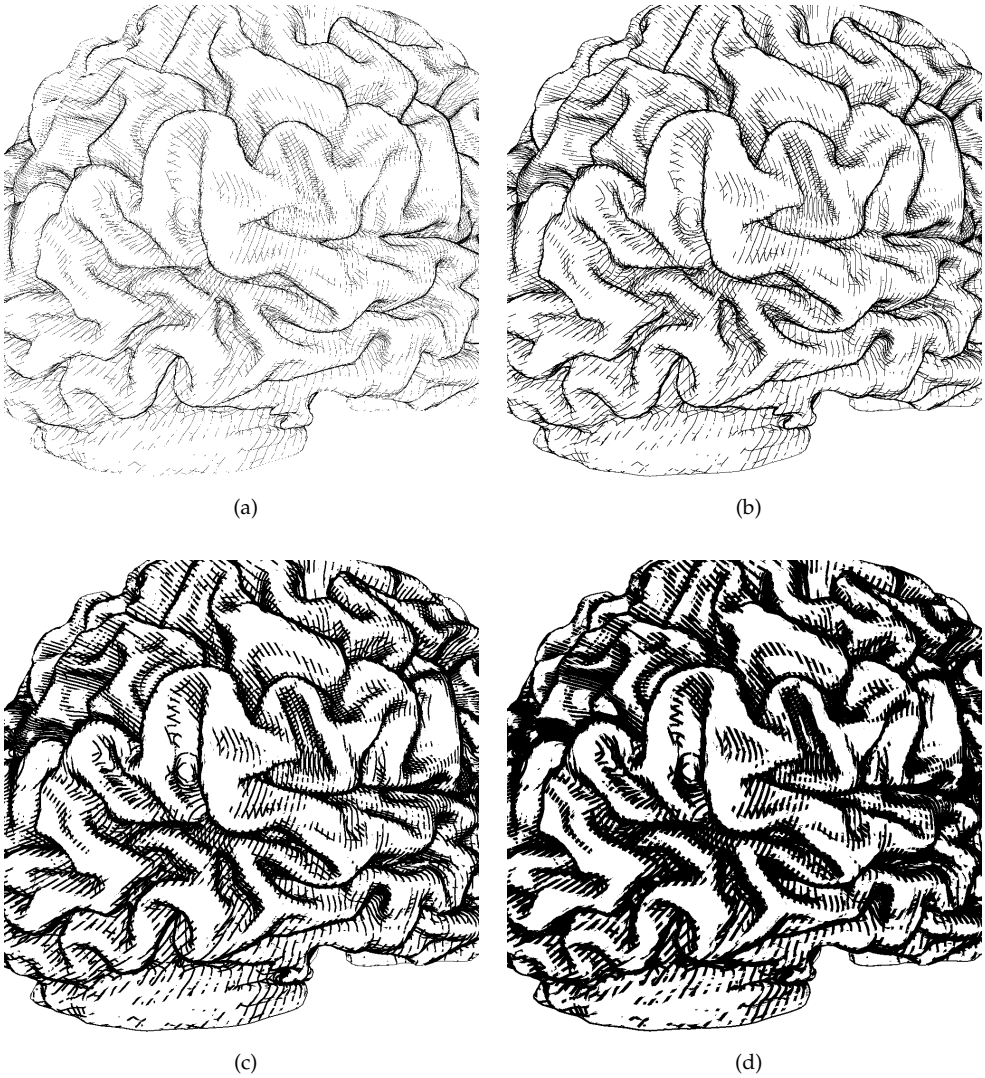
Results

In this Chapter I will show the results of my method. Section 6.1 contains examples of renderings that my method can produce and it also looks at the performance of my application at different resolutions and at different settings. Finally, Section 6.2 shows the results of an informal evaluation with users who are experienced in the field of neural tractography.

6.1 Results

This section contains renderings that show the different parameters settings of the program together with performance data. All renderings are made on a window with a resolution of 880×720 on a machine with 2 GB of ram, an AMD Athlon 64 X2 Dual Core 5200+ processor running at 2.61 GHz and a Geforce 8800 GTX card. Figure 6.1 shows the effect of different line widths al lthe configurations give me a stable frame rate of 24. Notice how very thin lines become dashed. Figure 6.2 shows stipples at different density and sizes, notice again how the frame rate stays constant at 28–29 frames per second. Figure 6.3 shows the effect of halos to bring out the content, it also shows the effect of anaglyph rendering. Notice how the frame rate it cut in half by anaglypth rendering, this is because the scene is rendered twice now, one for each eye.

From these results it is clearly visible that the performance is not effected by any of the setting too much. Only Figure 6.1 has a lower frame rate, this is due to the algorithm performing a halo check for all the pixels not part of the fibers and because no fibers are visible a check will have to be performed for all of them. In fact, the performance is only effected by the size of the render buffer. In Table 6.1 frame rates for different resolutions are shown for examples in Figure 6.2, every time I tried to fill the whole screen with the model. Because of this users can choose any configuration of lines and stipples possible withouth effecting performance too much. The configuration to choose depends on the flavor of the user but normally thinner lines and stipples tend to look better on screen and also make the fiber tracts come out.

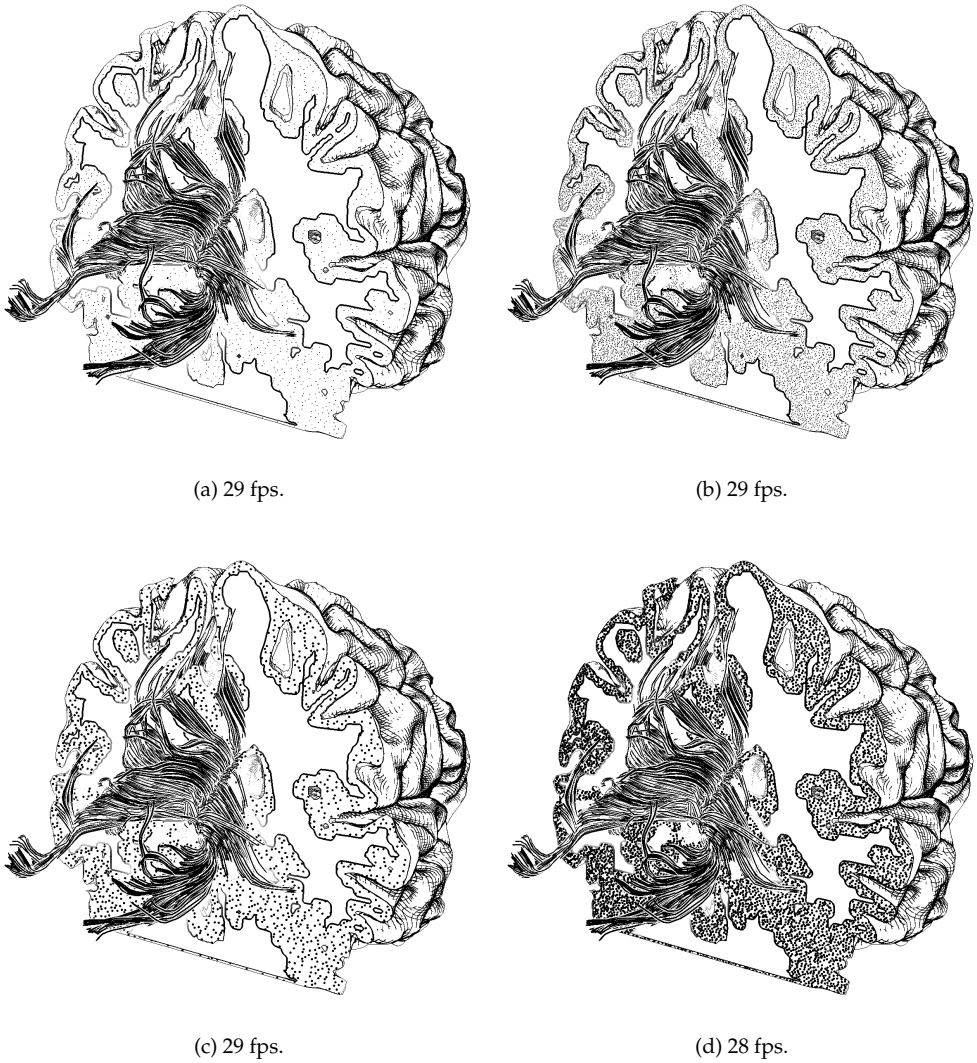


(a) (b)

(c) (d)

Figure 6.1: *Examples of renderings with different line width. For all the pictures the frame rate was stable at 24 fps.*

It is also possible to display datasets with presegmented objects like a skull or a tumor in them. It is also possible to add different hatching styles to these objects. For this I use masks just like with the stippling to tag the pixels belonging to the presegmented objects. Some examples of these are seen in Figure 6.5



(a) 29 fps.

(b) 29 fps.

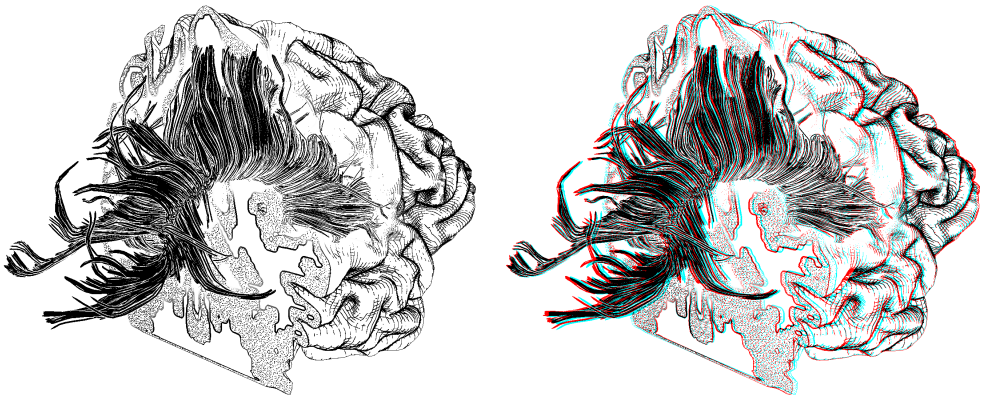
(c) 29 fps.

(d) 28 fps.

Figure 6.2: *Examples of renderings with different stipple width and density. Also giving the frame rates when rendered to a 880×720 buffer.*

6.2 Evaluation

Two informal evaluations were carried out. One for the illustrative rendering method and one for the frame interaction method. Both evaluations provided a lot of valuable feedback that will be discussed next.



28 fps 14 fps

Figure 6.3: *Example of transparency and halos and anaglyph rendering*

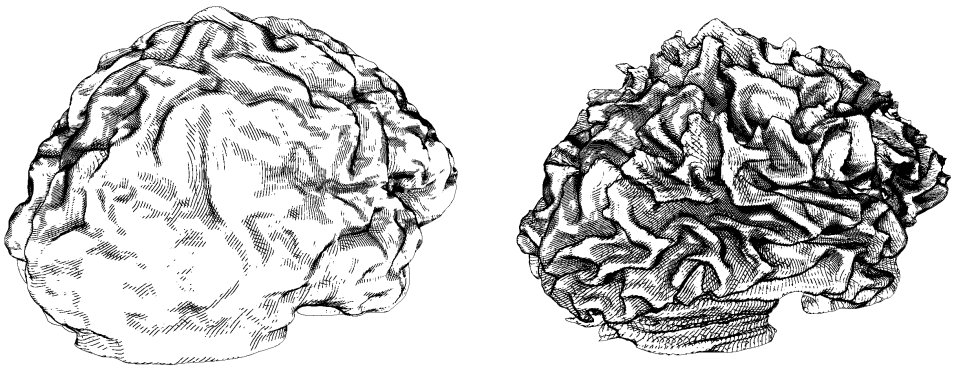
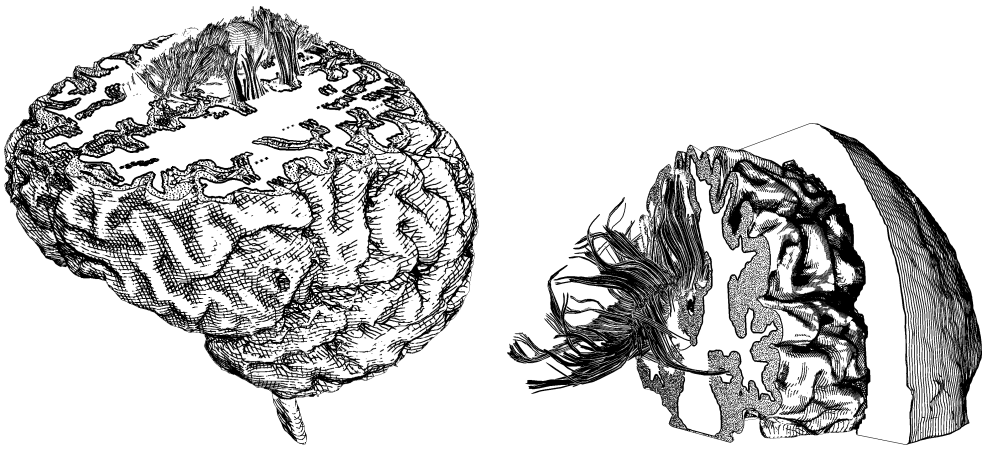


Figure 6.4: *Effect of extracting surface at different iso-values. When altering iso-values the performance of the program stays at 5-6 fps.*

Table 6.1: *Frame rates at different resolutions.*

Resolution	Avarage frame rate
880×720	29
935×888	20
1280×920	15



A dataset with a presegmented tumor. A datset with a presegmented skull

Figure 6.5: *Rendering data with different presegmented object, each with it's own style.*

6.2.1 Rendering evaluation

We held an informal evaluation with two neuroscientists. We invited them to our lab and showed them our software on an 52" screen. We also showed them examples of printed images to show them how this method looks on paper.

They were impressed with the images and saying that they looked like drawings from Leonardo da Vinci. The scientist also mentioned that this technique can be used in several cases. One is printed material, the scientist said that to have color pages in publications increases the publication costs by a lot. But they also said that this software can be used in lectures. They also liked the anaglyphic rendering because this made it easier to set apart the fibers from the context. They also mentioned that there is no other software that will produce these kind of pictures and that adding this method to already existing packages like FSL or SPM (Friston 2003) would benefit the medical community a lot.

6.2.2 Interaction evaluation

An informal evaluation of the interaction part was carried out. The main goal was to evaluate the frame interaction method compared to software packages that are normally used for brain and fiber exploration (e.g., TrackVis). For this I invited a neuroscientist who already had experience working with those packages to our lab. I first explained to him how the frame interaction method works and then I let the scientist try the software out for himself. While he was working with the software I

asked for comments on the interaction and I also observed how fast he was able to learn to use the software.

After the explanation of the mappings, the scientist noticed immediately that the two-point RST interaction was missing and asked why. This was accompanied by a two hand gesture towards the screen as if trying to rotate or to zoom in. This indicates the kind of interaction people expect from and multi-touch interface. After explaining to him that he could not use RST and only the mappings we presented to him he started trying out the software. The scientist was able to figure out how to use the frame border immediately. He commented that he liked the idea and that it was very easy to use. He especially liked how the manipulation of the cutting planes was implemented and compared it to TrackVis saying that TrackVis has 3 little view ports with extra sliders to control the slices and that he liked that this method is free of those extra windows.

What he did not like was the way the box selection was implemented. He liked being able to point with two fingers to position the boxes but he did not like the way you could move the boxes by touching the sides of them. He compared this to the software packages he is used to. These packages often have spheres that can be dragged parallel to the view plane and that is easier and more intuitive to use than dragging the boxes one axis at a time.

Chapter 7

Conclusion

7.1 Conclusion

In this thesis I presented a method to create illustrative visualization of fiber tracks and context. The presented method combines a number of previously developed techniques and makes them available for interactive illustrative visualization of DTI data of the brain. The main contributions of this thesis lies in adapting the techniques to the GPU to achieve real-time rendering and in combing the techniques to produce an images and illustrative style that stays consistent. The illustration style is inspired by illustrations in old medical books as shown in Figure 1.2 and by images in scientific papers. One of the additional goals of this method was also to produce clear high-resolution illustrations that could be used in print.

Another contribution of this thesis is to demonstrate the use of how the frame border interaction technique to explore DTI datasets. Using the frame border, I presented a number of simple gestures that can be used to do rotation, translation and scaling of the data. It is also possible to look inside object by slicing them open and to select fibers using boxes.

From evaluation with experts in tractography it also became clear that this software can be used in teaching. The software could also be used for making pictures for papers or illustrations for biology books. The multi-touch aspect can also be used in, for instance, collaboration; having a big screen is more accessible to a crowd of scientists than one computer.

7.2 Future work

From the two informal evaluations held a number of ideas came up for using this technique. From the illustration point of view, the scientist provided me with more examples of illustrations in medical papers. An interesting one was about animal (Mesulam and Mufson 1982) tractography where only the end points of the fibers where drawn on the brain surface (see Figure 7.1). Those endpoints where represented as dots. Another point was integrating this method in already existing soft-

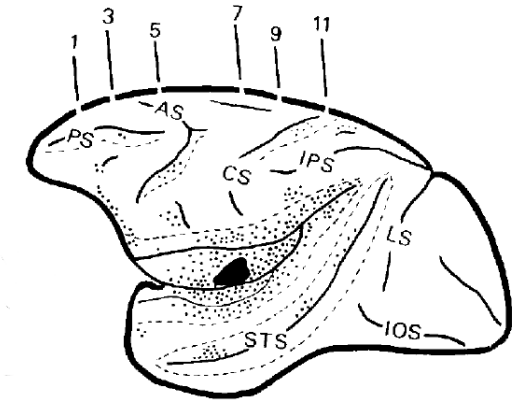


Figure 7.1: *From (Mesulam and Mufson 1982): Animal tractography.*

ware packages. FSL and PSM were mentioned as examples. PSM is a collection of scrips for matlab and has the ability to be expanded by the user so this is a likely candidate for integrating this method. Another possible extension of my work is to be able to select only part of fibers, right now if a part of a fiber is inside a selection box the whole fiber is shown. The existing method can be adapted to do this but the filter algorithm is not designed to take parts of fibers into account and this can lead to unforeseen performance issues.

From an interactive point of view, one scientist mentioned that this software can be used to let scientist collaborate. Collaboration is very common in tractography and a big touch screen makes it easy for more that one person to provide input into a program.

Bibliography

Appel, A., Rohlf, F. J. and Stein, A. J.: 1979, The Haloed Line Effect for Hidden Line Elimination, *ACM SIGGRAPH Computer Graphics* **13**(3), 151–157.

Bavoil, L. and Sainz, M.: 2009, Multi-layer dual-resolution screen-space Ambient Occlusion, *SIGGRAPH '09: SIGGRAPH 2009: Talks*, ACM, New York, NY, USA.

Bavoil, L., Sainz, M. and Dimitrov, R.: 2008, Image-space horizon-based Ambient Occlusion, *SIGGRAPH '08: ACM SIGGRAPH 2008 talks*, ACM, New York, NY, USA.

Bruckner, S. and Gröller, E.: 2007, Enhancing Depth-Perception with Flexible Volumetric Halos, *IEEE Transactions on Visualization and Computer Graphics* **13**(6), 1344–1351.

Catani, M. and Thiebaut de Schotten, M.: 2008, A Diffusion Tensor Imaging Tractography Atlas for Virtual in Vivo Dissections, *Cortex* **44**(8), 1105–1132.

Dauber, W.: 2005, *Feneis' Bild-Lexikon der Anatomie*, 9th edn, Georg Thieme Verlag, Stuttgart.

Deering, M., Winner, S., Schediwy, B., Duffy, C. and Hunt, N.: 1988, The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics, *ACM SIGGRAPH Computer Graphics* **22**(4), 21–30.

Deussen, O., Hamel, J., Raab, A., Schlechtweg, S. and Strothotte, T.: 1999, An Illustration Technique Using Intersections and Skeletons, *Proc. Graphics Interface*, Morgan Kaufmann, San Francisco, pp. 175–182.

Deussen, O., Hiller, S., van Overveld, C. and Strothotte, T.: 2000, Floating points: A method for computing stipple drawings, *Computer Graphics Forum* **19**, 40–51.

Ebert, D. S. and Sousa, M. C.: 2006, Illustrative Visualization for Medicine and Science, *ACM SIGGRAPH 2006 Courses*.

Elber, G.: 1995, Line Illustrations $\in$ Computer Graphics, *The Visual Computer* **11**(6), 290–296.

Evans, A.: 2006, Fast Approximations for Global Illumination on Dynamic Scenes, *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, ACM, New York, NY, USA, pp. 153–171.

Everts, M. H., Bekker, H., Roerdink, J. B. and Isenberg, T.: 2009, Depth-Dependent Halos: Illustrative Rendering of Dense Line Data, *IEEE Transactions on Visualization and Computer Graphics* **15**(6), 1299–1306.

Freudenberg, B., Masuch, M. and Strothotte, T.: 2001, Walk-Through Illustrations: Frame-Coherent Pen-and-Ink Style in a Game Engine, *Computer Graphics Forum* **20**(3), 184–191.

Friston, K.: 2003, Introduction: Experimental design and statistical parametric mapping, *in* R. Frackowiak, K. Friston, C. Frith, R. Dolan, K. Friston, C. Price, S. Zeki, J. Ashburner and W. Penny (eds), *Human Brain Function*, 2nd edn, Academic Press.

Hancock, M., Carpendale, S. and Cockburn, A.: 2007, Shallow-Depth 3D Interaction: Design and Evaluation of One-, Two- and Three-Touch Techniques, *Proc. CHI*, ACM, New York, pp. 1147–1156.

Hancock, M. S., Carpendale, S., Vernier, F. D., Wigdor, D. and Shen, C.: 2006, Rotation and Translation Mechanisms for Tabletop Interaction, *Proc. Tabletop*, IEEE Computer Society, Los Alamitos, pp. 79–88.

Hargreaves, S. and Harris, M.: 2004, Deferred Shading, *Whitepaper and presentation*, NVIDIA.

Hertzmann, A. and Zorin, D.: 2000, Illustrating Smooth Surfaces, *Proc. SIGGRAPH*, ACM, New York, pp. 517–526.

Hodges, E. R. S. (ed.): 2003, *The Guild Handbook of Scientific Illustration*, 2nd edn, John Wiley & Sons, Hoboken, NJ.

House, E. L. and Pansky, B.: 1960, *A Functional Approach to Neuroanatomy*, McGraw-Hill Book Company, New York.

Hultquist, J.: 1990, A Virtual Trackball, *Graphics gems*, Academic Press Professional, Inc., San Diego, CA, USA, pp. 462–463.

Johansson, G. and Carr, H.: 2006, Accelerating Marching Cubes with graphics hardware, *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, ACM, New York, NY, USA, p. 39.

Kim, S., Maciejewski, R., Isenberg, T., Andrews, W. M., Chen, W., Sousa, M. C. and Ebert, D. S.: 2009, Stippling By Example, *Proc. NPAR*, ACM, New York, pp. 41–50.

Kopf, J., Cohen-Or, D., Deussen, O. and Lischinski, D.: 2006, Recursive Wang Tiles for Real-Time Blue Noise, *ACM Transactions on Graphics* **25**(3), 509–518.

Kruger, R., Carpendale, S., Scott, S. D. and Tang, A.: 2005, Fluid Integration of Rotation and Translation, *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, New York, NY, USA, pp. 601–610.

Landis, H.: 2002, Production-Ready Global Illumination, *Course Notes of SIGGRAPH 2002*, number 16, ACM, New York, chapter 5, pp. 331–338.

Leister, W.: 1994, Computer Generated Copper Plates, *Computer Graphics Forum* **13**(1), 69–77.

Lorensen, W. E. and Cline, H. E.: 1987, Marching Cubes: A high resolution 3d surface construction algorithm, *SIGGRAPH Comput. Graph.* **21**(4), 163–169.

Lu, A., Morris, C. J., Taylor, J., Ebert, D. S., Hansen, C., Rheingans, P. and Hartner, M.: 2003, Illustrative Interactive Stipple Rendering, *IEEE Transactions on Visualization and Computer Graphics* **9**(2), 127–138.

McGuire, M.: 2010, Ambient Occlusion Volumes, *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA.

Meruvia Pastor, O., Freudenberg, B. and Strothotte, T.: 2003, Real-Time, Animated Stippling, *IEEE Computer Graphics and Applications* **23**(4), 62–68.

Mesulam and Mufson, E. J.: 1982, Insula of the old world monkey. iii: Efferent cortical output and comments on function., Vol. 212, pp. 38–52.

Meyer, T. and Globus, A.: 1993, Direct Manipulation of Isosurfaces and Cutting Planes in Virtual Environments, *Technical Report CS-93-54*, Brown University, Providence, RI, USA.

Miller, G.: 1994, Efficient Algorithms for Local and Global Accessibility Shading, *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, pp. 319–326.

Mori, S. and van Zijl, P. C.: 2002, Fiber Tracking: Principles and Strategies – A Technical Review., *NMR Biomed* **15**(7–8), 468–480.

Nijboer, M., Gerl, M. and Isenberg, T.: 2010, Exploring Frame Gestures for Fluid freehand Sketching, *Proceedings of the Sketch Based Interfaces and Modeling Symposium*.

Ostromoukhov, V.: 1999, Digital Facial Engraving, *Proc. SIGGRAPH*, ACM, New York, pp. 417–424.

Post, F. H., Vrolijk, B., Hauser, H., Laramee, R. S. and Doleisch, H.: 2002, Feature Extraction and Visualisation of flow fields.

Praun, E., Hoppe, H., Webb, M. and Finkelstein, A.: 2001, Real-Time Hatching, *Proc. SIGGRAPH*, ACM, New York, pp. 581–586.

Ritter, F., Hansen, C., Dicken, V., Konrad, O., Preim, B. and Peitgen, H.-O.: 2006, Real-Time Illustration of Vascular Structures, *IEEE Transactions on Visualization and Computer Graphics* **12**(5), 877–884.

Saito, T. and Takahashi, T.: 1990, Comprehensible Rendering of 3-D Shapes, *ACM SIGGRAPH Computer Graphics* **24**(3), 197–206.

Salisbury, M. P., Anderson, C., Lischinski, D. and Salesin, D. H.: 1996, Scale-Dependent Reproduction of Pen-and-Ink Illustration, *Proc. SIGGRAPH*, ACM, New York, pp. 461–468.

Schmahmann, J. D., Pandya, D. N., Wang, R., Dai, G., D'Arceuil, H. E., de Crespigny, A. J. and Wedeen, V. J.: 2007, Association Fibre Pathways of the Brain: Parallel Observations from Diffusion Spectrum Imaging and Autoradiography, *Brain* **130**(3), 630–653.

Secord, A., Heidrich, W. and Streit, L.: 2002, Fast primitive distribution for illustration.

Shanmugam, P. and Arikan, O.: 2007, Hardware Accelerated Ambient Occlusion Techniques on GPUs, *Proc. I3D*, ACM, New York, pp. 73–80.

Tarini, M., Cignoni, P. and Montani, C.: 2006, Ambient Occlusion and Edge Cueing for Enhancing Real Time Molecular Visualization, *IEEE Transactions on Visualization and Computer Graphics* **12**(5), 1237–1244.

Tatarchuk, N., Shopf, J. and DeCoro, C.: 2007, Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline, *ACM SIGGRAPH 2007 Courses*, number 28, ACM, New York, chapter 9, pp. 122–137.

Tietjen, C., Isenberg, T. and Preim, B.: 2005, Combining Silhouettes, Shading, and Volume Rendering for Surgery Education and Planning, *Proc. EuroVis*, Eurographics Association, Aire-la-Ville, Switzerland, pp. 303–310.

Viola, I., Gröller, M. E., Hadwiger, M., Buhler, K., Preim, B., Costa Sousa, M., Ebert, D. and Stredney, D.: 2005, Illustrative Visualization, *Course Notes of IEEE VIS 2005*, IEEE Computer Society, Los Alamitos.

Wakana, S., Jiang, H., Nagae-Poetscher, L. M., van Zijl, P. C. M. and Mori, S.: 2004, Fiber Tract-based Atlas of Human White Matter Anatomy, *Radiology* **230**(1), 77–87.

Wilson, A. D., Izadi, S., Hilliges, O., Garcia-Mendoza, A. and Kirk, D.: 2008, Bringing Physics to the Surface, *Proc. UIST*, ACM, pp. 67–76.

Winkenbach, G. A. and Salesin, D. H.: 1994, Computer-Generated Pen-and-Ink Illustration, *Proc. SIGGRAPH*, ACM, New York, pp. 91–100.

Wyvill, G., McPheeters, C. and Wyvill, B.: 1986, Data Structure For Soft Objects.

Yu, L. and Isenberg, T.: 2009, Exploring One- and Two-Touch Interaction for 3D Scientific Visualization Spaces, *in* M. Ashdown and M. Hancock (eds), *Posters of Interactive Tabletops and Surfaces (ITS 2009, November 2325, 2009, Banff, Alberta, Canada)*. Extended abstract and poster, to appear.

Zander, J., Isenberg, T., Schlechtweg, S. and Strothotte, T.: 2004, High Quality Hatching, *Computer Graphics Forum* **23**(3), 421–430.

Zhukov, S., Inoes, A. and Kronin, G.: 1998, An Ambient Light Illumination Model, *Rendering Techniques*, Springer-Verlag, Wien, New York, pp. 45–56.