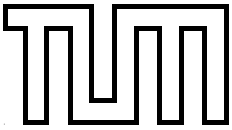
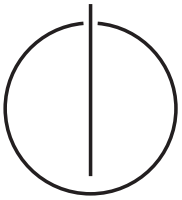


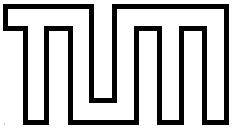
FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Visualization of 4D Breathing Motion from Medical Datasets

Markus Müller





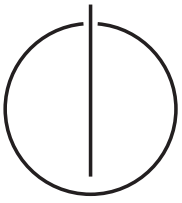
FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Visualization of 4D Breathing Motion from Medical Datasets

Visualisierung der 4D Atembewegung von medizinischen Datensätzen

Author:	Markus Müller
Supervisor:	Prof. Dr. Nassir Navab
Advisor:	Athanasios Karamalis
Date:	15. November 2010



Declaration

I assure the single handed composition of this Bachelor's Thesis only supported by declared resources.

Garching, November 15, 2010 Author

Abstract

Through new improvements in medical imaging, it is possible to record multiple three dimensional volumes over time, capturing the organ motion. To be useful for the medical staff these volumes need to be displayed in some way. This bachelor's thesis first introduces the basics of volume rendering, especially the principle of raycasting, to generate 3-dimensional images from medical data sets. The main focus lies on visualizations that enhance the viewer's perception of movements inside the volume.

Five different visualization techniques are presented which are all designed for an interactive usage and an optimal interplay with the volume renderer. In order to reach an interactive performance, all methods rely on hardware acceleration through a GPU.

Kurzfassung

Durch neuartige Entwicklungen in bildgebenden Verfahren der Medizin, ist es möglich mehrere drei dimensionale Volumen über die Zeit aufzunehmen und damit Organ Bewegungen einzufangen. Um für medizinisches Personal nützlich zu sein, müssen diese Volumen in irgendeiner Weise dargestellt werden. Diese Bachelorarbeit gibt zuerst eine Einführung in die Grundlagen des Volume Rendering, besonders das Prinzip des Raycastings, um damit 3-dimensionale Bilder von medizinischen Datensätzen zu erzeugen. Der Hauptfokus liegt allerdings auf Visualisierungen, die die Wahrnehmung der Volumen Bewegungen für den Betrachter verbessert.

Es werden fünf verschiedene Visualisierungen vorgestellt, die alle auf eine interaktive Benutzung und ein optimales Zusammenspiel mit dem Volume Renderer ausgelegt sind. Um eine interaktive Darbietung zu erreichen, sind alle Methoden auf eine Hardware Beschleunigung durch die GPU angewiesen.

Acknowledgments

My biggest gratitude goes to my advisor Athanasios for his support. He had always time for my questions and helped me out when I had some programming issues. I also want to thank professor Navab for his inspirations after my kick-off presentation. And finally thanks to my housemate Johanna for proofreading this thesis.

Contents

Chapter 1

General introduction and outline of the thesis

Through the rapid advancement of technology in the medical sector, clinicians are facing more and more data. Since the perception and cognition of the human brain is limited, the user of such dataset is easily overwhelmed with information [6]. So with increasing amount of data the medical staff may not take advantage of the information stored in the data. Solutions to this problem can be found in the area of information visualization. Information visualization can be defined as "the process of transforming data, information, and knowledge into visual form making use of humans' natural visual capabilities" [6]. With the rapidly growing performance and storage capacity, modern computer can be used for interactive visualization of enormous amounts of data. But the goal is not to let the computer do the interpretation of dataset – which is an active field of research by itself. Instead the data is just refined and reduced by a software to let the viewer explore the data more intuitive and thus get a deeper understanding of the details and relations which are not obvious to spot in the raw data.

Medical imaging splits up into the image acquisition, the processing and visualization of the acquired images [6]. Three of the main sources of medical images are the computed tomography (CT), magnetic resonance imaging (MRI) and ultrasound imaging. Each of these systems is capable of recording a stack of 2D slices of the human body, which can be interpreted as a 3D volume. In the case of a CT these images are generated by shooting x-rays through the body and ana-

lyzing their absorption. The basic idea is that different tissues have different absorption rates and therefore can be visualized as different gray values. A MRI first generates a powerful magnetic field around the person which causes water protons in the body to align with the field. An applied electromagnetic field then changes the alignment of the protons. When the electromagnetic field is turned off, the protons again align to the magnetic field which creates a signal that can be detected by a scanner to create the image. The strength of the signal depends on the strength of the magnetic field and the tissue the proton is part of. Ultrasound on the other hand uses sonic waves which are reflected by the tissue. The echoes with their amplitude and spatial location are recorded and imaged. Each procedure has its own area of application depending on the patient.

While 3D volume visualization is still an ongoing field of research [25][7][13], this thesis introduces different visualization techniques for 4D volume datasets. A 4D dataset is thereby only several 3D volumes taken over time. So additional to the images of the body's inside, also the motions of organs come clear. Unfortunately most of these movements are only marginal and therefore hard to spot especially over a greater period of time. A good visualization should counter this problem and help a viewer to understand and interpret the motions more easily. Since most of the motions in the body are caused by breathing, this thesis will concentrate on the breathing motion, but the presented methods should also work under different conditions.

Since all visualization should be usable at interactive frame rates, all procedures make use of the graphics processing unit (GPU). So the first part of the thesis will give an overview of different GPU programming techniques. By name these are OpenGL, an application programming interface (API) specialized on producing 3D graphics, and CUDA, an API for general programming on a (NVIDIA) GPU. The first part of the second chapter introduces a technique to visualize 3D and 4D volumes called raycasting. The second part describes a raycaster implementation in CUDA with emphasis on cutting planes and transfer functions. Previous work is also handled in the third chapter, which covers especially flow visualization. Finally the fifth chapter

presents different visualization approaches for 4D volumes. The visualizations start with some methods for 2D space but the focus lies on the following 3D techniques. At the end of the chapter the achieved application is presented and the performance of the visualizations estimated. The last chapter gives a short conclusion and proposals for future work.

Chapter 2

Introduction to OpenGL and CUDA

The following chapter describes two APIs for programming the graphics hardware. Though both utilize the same architecture, the first presented OpenGL is specialized in 2D and 3D graphics, while CUDA pursues a more general programming approach.

2.1 OpenGL - The Open Graphics Library

OpenGL is a 3D Graphics API currently developed by the Khronos Group, an industry consortium of companies like Apple, AMD, Intel, NVIDIA and more [3]. As the name suggests it is an open specification of a graphics library (GL) which is implemented by different (hardware) vendors (the implementation is often not open source). OpenGL may be hardware accelerated by a graphic card (GPU) but can also run in software – but properly with a heavy impact on performance. Additional functionality is provided with so called extensions. All members of the Khronos Group can add custom extension to the standard, however all methods and objects of the extension are marked by a vendor's suffix and are therefore not available in all implementations. When the extension prevailed and is implemented by different vendors, the members of the Khronos Group can decide to include the extension to the core specification.

There are basically two different versions and thus programming paradigm: the Fixed-Function Pipeline (version 1.0-1.5) and the Programmable Pipeline (since version 2.0). OpenGL 2.0 and 2.1 are a

bit special, because they support both pipelines [3]. Since the fixed functionality is deprecated, this thesis will concentrate on the programmable pipeline. However, many elements of the Fixed-Function Pipeline are still used in the programmable parts.

2.1.1 GLSL

The GL Shading Language is a programming language for so called shader [23]. A shader is a small program that is executed on a stream of elements - for example vertices or pixels. It is characteristic that every shader call is independent from previous or future calls, so that the processing order of the input stream does not have to be deterministic. Because of this the stream can be processed in parallel. I refer to "shader" in the following as the program that is executed on a single element of the stream, so for example "the shader modifies the vertex color" means that the shader changes the color of every single vertex of the stream. Modern graphic hardware persists of over 100 shader units thus having an extremely high through-put compared to a classic single core CPU (Fig. 2.5) – but of course with a more limited field of use. GLSL is based on the C programming language with some additions like build-in vector and matrix structures. Most of the C features like loops, branching, functions and preprocessor directives are supported. However, pointers and recursion are not part of GLSL.

Currently there are three different types of shader, which in order of execution are: vertex-, geometry- and fragment shader. All have a specialized function in the pipeline and therefore different in- and outputs. Each shader can send variables to the following stage but not the other way around – so a geometry shader can communicate with the fragment shader but not with the vertex shader. By the use of so called "uniform" variables the application can also set special variables in the shader. However, this has to be done before the shader is executed on a block of data. The three shader together build a so called "shader program" which defines more or less the core functionality of the pipeline. A shader program can be switched while rendering, so it is possible to use different shader programs on different scene objects.

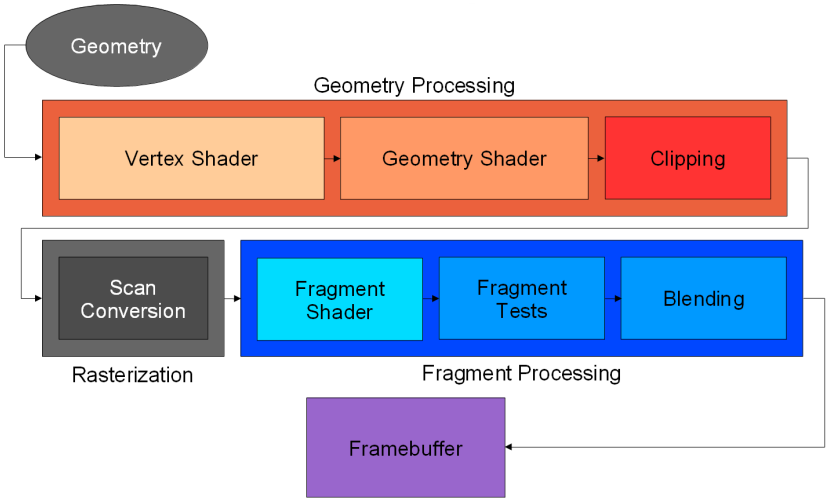


Figure 2.1: OpenGL Programmable Pipeline

GLSL is part of the OpenGL standard but it is also possible to use other shading languages like Cg (C for Graphics)[8]. These alternatives only differ in language features but the overall shader structure remains the same.

2.1.2 Vertex Buffer Objects

OpenGL is designed with a pipeline structure in mind, which can be seen in figure 2.1. At first the pipeline needs some raw data from the application describing the desired scene. OpenGL supports different primitives like points, lines, triangles, quadrilaterals or polygons which can be used to build more complex structures. All these primitives are defined by their corner points, called vertices (sg. vertex). This structures are uploaded in the form of Vertex Buffer Objects (VBO) [17]. A VBO consists of an array of vertices and a list of indices which describe the connection between vertices. Once uploaded to the GPU the VBO can be reused again and again without any interference of the CPU. However, if the data of the VBO is changed by the CPU it has to be uploaded again. Beside the position and index, a VBO can also contain information about the color, normal and texture coordinate for every vertex. A normal OpenGL scene consists of different objects which are all saved as VBOs and rendered consecutive.

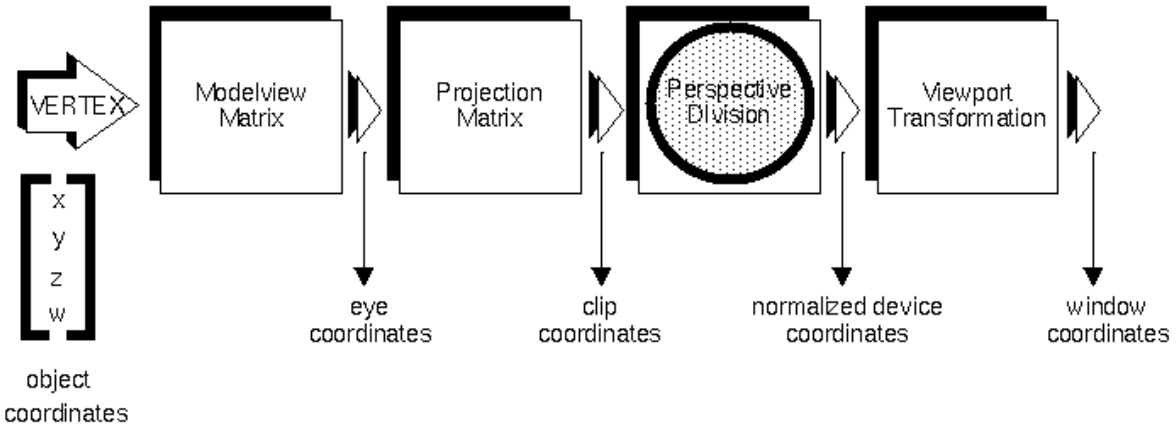


Figure 2.2: Stages of Vertex Transformation [3]

Until OpenGL 2.1 it was also possible to directly upload single primitives in each render pass. Since this is rather slow, primitives could be packed into so called display lists which are also just uploaded once. However display lists have a more restricted field of use than VBOs, so that since OpenGL 3.0 the direct mode and display lists are deprecated.

2.1.3 Vertex Shader

All VBOs entering the pipeline are first processed by the vertex shader. The shader gets a single vertex as input and also outputs a single vertex. It has access to all attributes of the vertex like the position, color or normal and can modify them. However, the shader has no information about the topology of the vertices and cannot remove or add a new vertex to the pipeline. Beside the vertex attributes, the shader has also read access to textures. A standard use case of the vertex shader is the ModelView- and Projective Transformation (Fig. 2.2). Before the vertex shader became programmable, it only applied both transformations and evaluated the local lighting.

ModelView Transformation

In order to avoid a constant change of the geometry to fit it in the right position of the scene, most OpenGL applications distinguish between

different spaces. A VBO starts in the object space where it is located in its own coordinate systems origin. For modeling a scene with more than one object, each object has to be transformed from its own object space to the global world space which can be done with a matrix multiplication. Each VBO uses an affine matrix containing rotations, translations and scalings which describes how to map its own coordinate system to the global system. This matrix used on every vertex of an object by the vertex shader transforms the complete object to the scene. While rotations and scalings can be described in one 3x3 matrix, the translation would be an extra addition with the translation vector. All three transformations can be put together into a 3x4 matrix. But since a multiplication of a 3x4 matrix with a 3x1 vector is not defined, a homogeneous coordinate is added to each vertex. Thus all vertices are extended from (x,y,z) to (x,y,z,1).

The same matrix can also be used to implement camera movements. Since OpenGL requires the camera in the origin, looking along the negative z-axis, the whole scene is moved around the camera instead of the camera around the scene – this is called the view- or eye space. So the object-to-scene matrix is simply multiplied by a scene-to-view matrix which contains the opposite transformations than the camera should have. Both matrices together are referred to as ModelView-Matrix.

Projective Transformation

Because the scene should later be displayed on the screen, it has to be projected from the 3D space to a 2D plane. The later rasterization will (more or less) just set the z coordinate to 0 – which results in an orthogonal projection. This, however, lacks any perspective impression since an object will retain its size regardless if it is near or far away from the camera. In order to get a real depth, far away objects should appear smaller than near ones. Since the vertices are already in homogeneous coordinates, a 4x4 projective matrix transforms them from (x,y,z,1) to (x,y,z,w). For simplicity let's assume to map a 2D scene to a 1D line with a distance of 1 from the origin like in figure 2.3. The Intercept

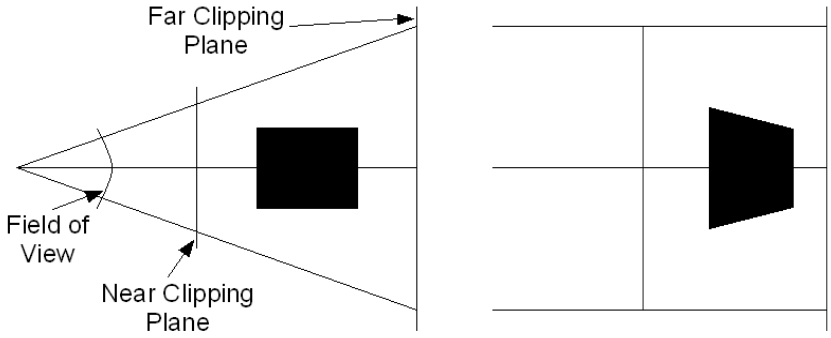


Figure 2.3: Image (a) before perspective transformation (b) after perspectiv transformation

theorem yields that (x,y,1) is mapped to (x/y,1,1) which is a multiple of (x,y,y). So transferred to the 3D case when (x,y,z,1) is mapped to (x,y,z,z) and then divided by z, all vertices will lie on the same hyperplane (x/z,y/z,1,1). Consequently a basic projection matrix would look like this:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

In order to support additional parameters like field of view, aspect ration and different clipping planes, the matrix can be extended with additional parameters.

2.1.4 Geometry Shader

A geometry shader gets a single primitive as input and can add or remove vertices from it or create new primitives [5]. The type of the in and out coming primitives must be known before runtime but can differ from each other. For example a shader gets points as input and outputs a triangle. The vertices of one primitive are accessible as an array but the shader emits the outgoing vertices one by one. The geometry shader stage is optional and can be skipped. Vertices or primitives which are added by the geometry shader are not processed by the vertex shader. Regardless of the input, the geometry shader can only emit

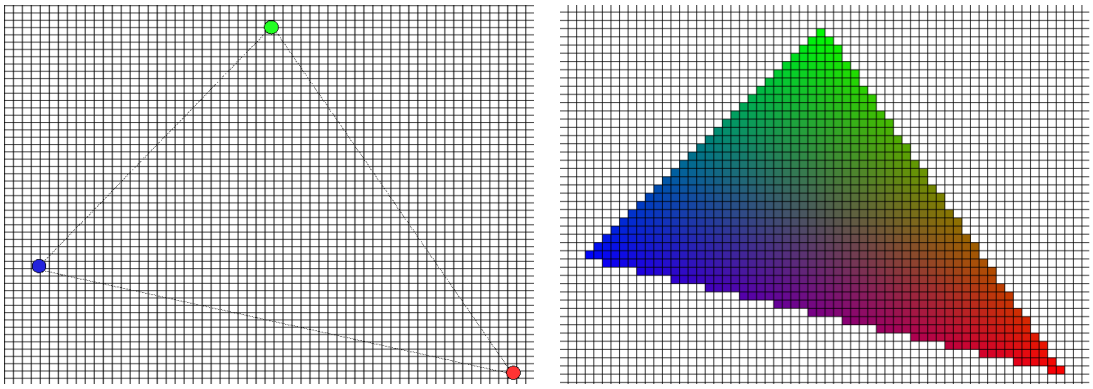


Figure 2.4: Image (a) triangle after projection (b) triangle after rasterization

points, lines or triangles. Even if this shader stage is unused, OpenGL will break quadrilaterals or polygons into triangles.

2.1.5 Clipping and Rasterization

Before the rasterization, OpenGL automatically clips all primitives which are outside the viewing volume. Primitives protruding into the view are split so that only the visible part remains.
Until now the scene consisted only of vertices and connections between them, the surface of a primitive was insignificant. OpenGL now projects all vertices onto a plane whose size matches the window size and rasterizes each primitive at a pixel grid. Since all polygons are broken up into triangles and triangles are always planar, the color and other attributes at all points inside the polygon can be linear interpolated from the three corner vertices. This concept is illustrated in figure 2.4. The resulting values for each pixel are called fragments.
Because all primitives are scanned independently and may also overlap, there may be more than one fragment per pixel. However, in the final image each pixel can have only one color, so the following steps provide methods to merge this values into one.

2.1.6 Fragment Shader

Fragments leaving the rasterzation are processed by the Fragment Shader. Similar to the Vertex Shader it can only edit one fragment, has no access to other fragments and cannot add new fragments to the pipeline, though the shader can remove fragments. Important built-in attributes are the fragment color, the corresponding 2D pixel position, a depth value and the texture coordinates. The depth is a floating-point value between 0.0 to 1.0 and describes the fragments' position in relation to the near (0.0) and far (1.0) clipping plane. However, this depth buffer is not linear so that values at the near plane are more accurate than distant ones.
An important concept of modern 3D graphics are textures. A texture is usually a 2D image which is mapped onto a polygon. For example instead of building a stone wall of single bricks, just a picture (the texture) of a wall is mapped onto a quad. The texture coordinates define the exact mapping of the 2D texture onto the 3D polygon. They are set per vertex by the application and are also interpolated in the rasterization conversion. OpenGL supports 1D, 2D and 3D textures with 1 to 4 color channels and precisions of 4 to 32-bit for each channel.
The Fragment Shader has to output a color and a depth value. Typical use cases are per-pixel lighting, bump mapping or bloom effects [23] [9].

2.1.7 Fragment Tests

After the Fragment Shader the remaining fragments can be tested regarding to various conditions. Since most of these tests are deprecated and can be replaced by the previous shader stage, only the depth test should be mentioned here. As noted before, there may be more than one fragment per pixel but basically only the foremost is needed. The depth test checks every incoming fragments' depth with the depth of the corresponding pixel in the frame buffer. If the fragments' depth is greater than the pixel depth, the fragment lies behind the pixel and is occluded. Else the fragment is assigned to the blending stage.

2.1.8 Blending

Fragments which passed the previous tests are merged with the current pixels in the frame buffer. It is generally distinguished between blending with or without alpha. The alpha channel of a color defines its transparency: 0.0 means complete translucent and 1.0 complete solid, values in between are transparent. If no alpha channel is present or always 1.0, an incoming fragment just replaces the pixel in the frame buffer. However, if it is transparent, it need to be blended with the pixel color.

$$color = alpha_{fragment} * color_{fragment} + (1 - alpha_{fragment}) * color_{pixel}$$

Of course correct result are only achieved if the scene primitives are rendered from back to front, otherwise fragments will not pass the depth test even if they are behind a transparent pixel.
When the complete scene has run through the pipeline, the resulting image can be displayed on the screen.

2.1.9 Frame Buffer Objects

It is sometimes useful to render the scene to an off screen image instead of the frame buffer, because for example the rendering is just an intermediate step for further rendering and should not displayed to the viewer yet. For this case OpenGL supports Frame Buffer Objects (FBO). A FBO encapsulates the normal frame buffer and is a collection of logical buffers [19]. The FBO for itself has no real use until a texture or a Render Buffer Object (RBO) is attached to a certain buffer. These attachment buffers are the depth, stencil and multiple color buffers. If a buffer should later be accessible by the application or a shader, a normal texture has to be used as attachment. For buffers which are only used in one render step but not later on, a RBO is more suitable because it is faster. When the attachments are complete, the Frame Buffer Object can be bound to the current context and sort of replaces the actual frame buffer. All values normally written to the frame buffer are now written to the associated attached buffer of the FBO.

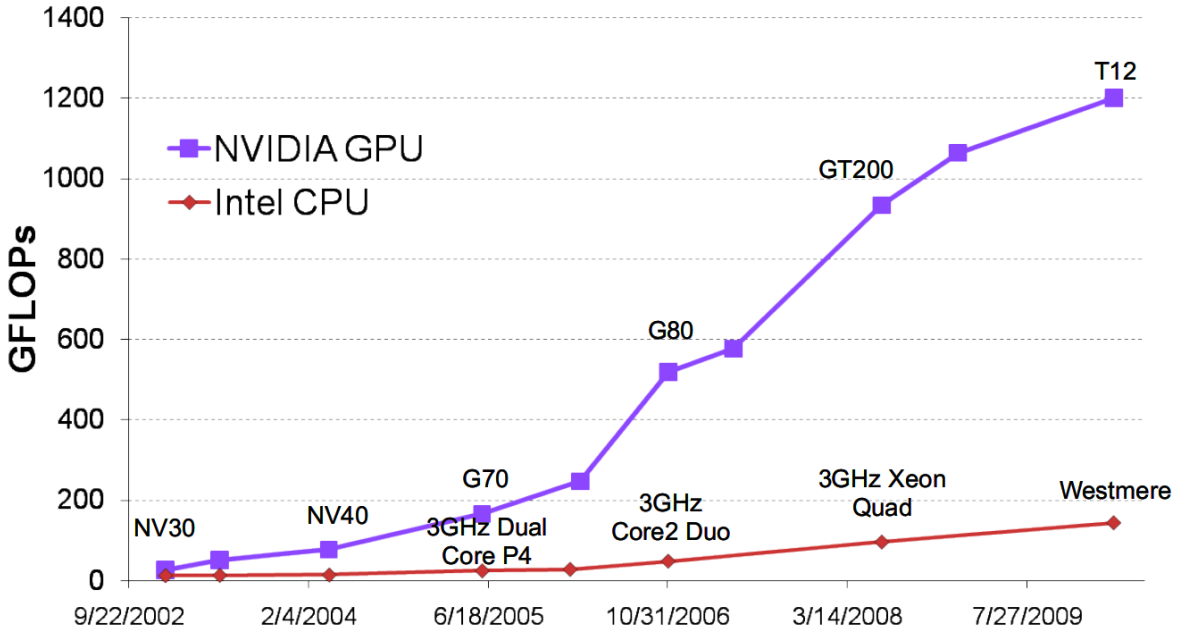


Figure 2.5: Comparison of development between CPUs and GPUs estimated in GFLOPs.[10]

For example it is possible to render only the depth buffer and use this buffer later on as a normal texture. At first a new FBO and a new 2D texture are generated. The texture is then attached to the depth buffer of the FBO. Before starting the first render pass, the FBO is bound to the current context. After the rendering, the Frame Buffer Object is unbound and the texture now contains the depth of the first render pass. Now a second scene can be rendered with the normal frame buffer and the shader can make use of the depth texture of the previous render step.

2.2 CUDA - Compute Unified Device Architecture

Realtime 3D graphics like we can see in modern video games or visualizations would not be possible without the acceleration of a separate hardware unit – the Graphics Processing Unit (GPU). This is achieved by a different architecture than regular Central Processing Units (CPU). Instead of just a few processing cores, a modern GPU consists of over 100 so-called shader units. These units work independently from each other on a stream of elements – in the case of OpenGL

vertices or fragments. This design enables an enormous data throughput compared to an iterative approach (Fig. 2.5). Of course these advantages are also be useful in non graphic related areas. For Nvidia GPUs this can be done with CUDA.

Programmers can use either the driver API or CUDA C. While the driver API is an assembly-like language, CUDA C is oriented on the C programming language and uses many of its features. A CUDA program consists of two parts in general: the host- and the device code. Since the GPU is usually a separate hardware device, it needs instructions from the CPU – the host code. It initializes certain parts, uploads resources and finally starts the device code. An overview of this process is displayed in figure 2.6. Since the host code is simply running on the CPU, it is written in C, C++ or any other language supported by CUDA and is often only executed once. The device code (kernel) on the other hand is written in CUDA C or with the driver API and is executed multiple times in parallel by the GPU. CUDA is design to use a hierarchical thread model. The kernel is executed by a thread, many threads together form a block and blocks are part of a grid. In order to execute the kernel, the host defines how many threads one block consists of and how many blocks should be used. Blocks are required to execute in parallel, threads in one block can synchronize their calculations. Each thread has its own local memory and all threads of one block can access the shared memory of the block which is slower but bigger than the thread memory. All threads independently of which block or grid can use the global memory, the biggest but slowest one.

Of course there are also other GPGPU (General-purpose computing on graphics processing units) architectures which are not limited to Nvidia cards: OpenCL [16] and DirectCompute. OpenCL is an open specification and shares many similarities with CUDA. Like OpenGL it is developed by the Khronos Group and can be implemented on every GPU by any vendor. DirectCompute is part of Microsoft's DirectX SDK since version 10.0. There is also AMD FireStream [1], but it has an analogous restriction like CUDA, namely that it only works on GPUs from AMD.

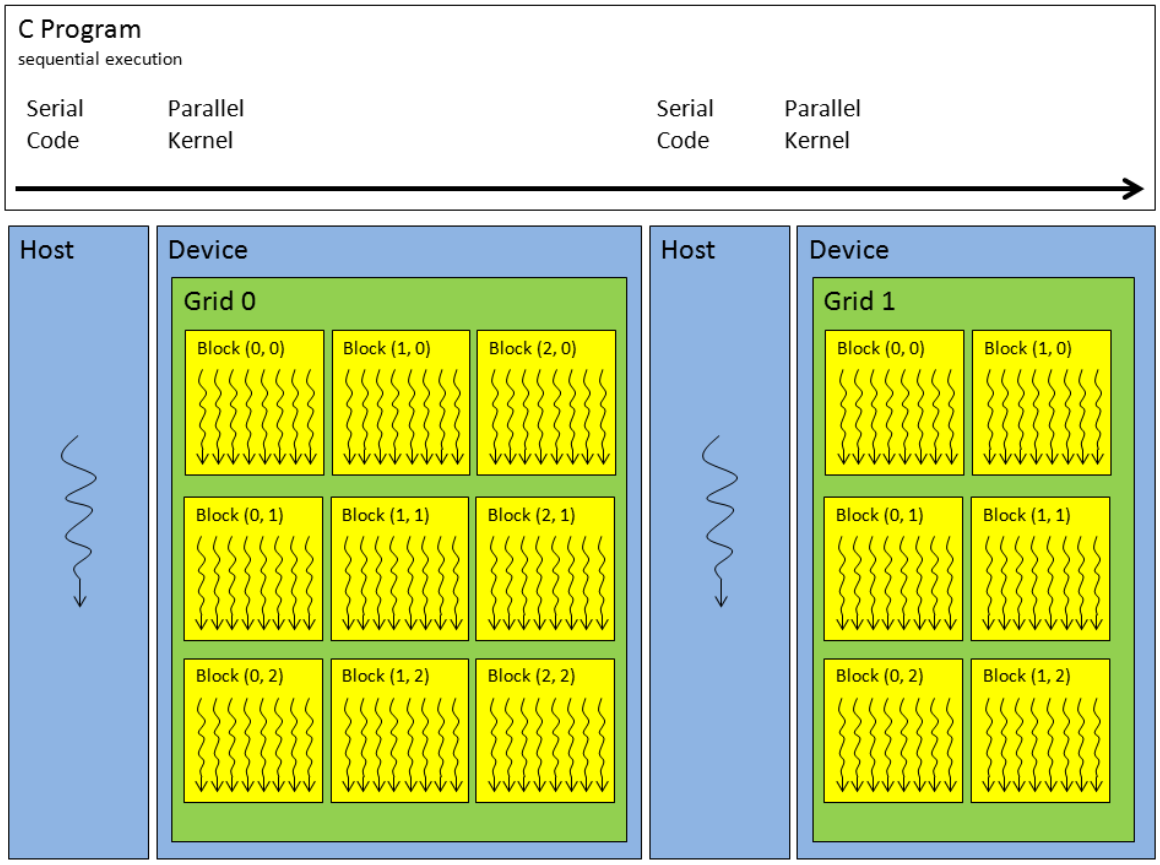


Figure 2.6: Serial code executes on the host while parallel code executes on the device.[18]

Chapter 3

3D - 4D Volume visualization

A fast and easy implemented technique to represent a volume in 3D is a GPU-based raycaster. It was first used in the 1980s but because of the high requirements on the graphics hardware the first interactive implementations were published around 2003 [7][13]. Basically a raycaster shoots rays from the viewpoint through the volume, takes samples along the way and finally sums these samples up to one value per ray. Since every ray can be calculated independently from all other rays, it is a perfect application to run parallel on the GPU. A raycaster should not be confused with a raytracer which has a complete different area of application. The following chapter will first describe the basic techniques of a raycaster and afterward an implementation with CUDA. Also an alternative volume rendering method is mentioned in the following. The end of the chapter gives an introduction to flow visualization and other previous work.

3.1 Raycasting

The first part of a raycaster is the generation of the rays. Since the result should be a two dimensional image of the volume, one ray is casted for each pixel. There are basically two approaches to setup the starting point and direction of a ray. The first one is straight forward by computing the intersection of the ray with the bounding box of the volume. The method described by Kay et al. [11] breaks the box into 3 pairs of parallel planes for the 6 sides of the cube, intersects them with

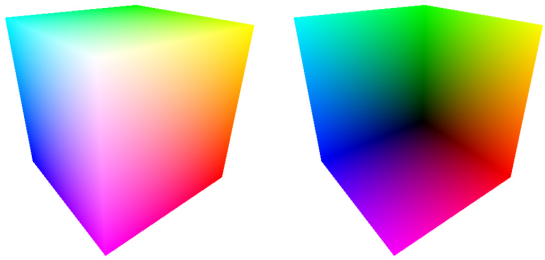


Figure 3.1: Rendering only the front or back faces of the color-coded bounding box retrieves starting and ending positions for the rays in volume coordinates for every single screen pixel [25]

the ray and returns the nearest and farest intersection distances. A ray is defined by its origin (R_o) and its direction (R_d). To use this method properly, the bounding box of the volume must be axis aligned. The algorithm first computes the intersection with all planes

$$T_1 = (\frac{box_{min} - \vec{R_o}}{\vec{R_d}}) \qquad T_2 = (\frac{box_{max} - \vec{R_o}}{\vec{R_d}}) \tag{3.1}$$

where box_{min} and box_{max} are three dimensional vectors containing the minimal and maximal extents of the box. The division is meant component-wise. Now for each dimension the smallest and largest values of T_1 and T_2 are stored in the 3d vectors T_{min} and T_{max}. The intersection points are now the largest value in T_{min} and the smallest value in T_{max}. If the largest value is smaller than the smallest, the box is not hit.

Another approach uses OpenGL features for the ray direction [25][13]. At first the volume bounding box is drawn as a solid cube. The colors of the corners encode their position ranging from (0.0,0.0,0.0) for the front, lower, left corner to (1.0,1.0,1.0) for the back, upper, right corner. This colored cube is now rendered into a texture from the desired camera position (Fig. 3.1 left). Remember that values between the corners are interpolated. The second steps renders the same cube but instead of the front faces now the back faces (Fig. 3.1 right). In order to receive the ray start and end position a fragment shader just needs to subtract the back face color from the front face color saved in the texture. The ray position are then already given in

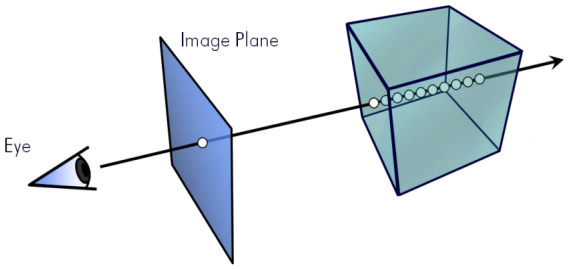


Figure 3.2: Schematic functionality of a raycaster [7]

volume coordinates and therefore always inside the volume.

With the definition of the start and end positions of the ray, the raycaster must now evaluate the volume-rendering integral [7] (see eq. 3.2), which is done by evaluating each ray at discrete sampling position and blending the samples.

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(t)dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} \kappa(t)dt} ds \tag{3.2}$$

The ray is initialized with the start position and a transparency (alpha) value of 0.0. A normal medical volume data set only contains values from 0.0 to 1.0 which can be interpreted as transparency. In the following steps each ray is sampled on discrete points, also displayed in Fig. 3.2:

- **volume access** the current position of the ray is used to look-up the data value from the volume
- **blending** the value of the previous data access is blended with the current color of the ray
- **advance ray position** the sampling position is advanced one step along the ray

If the ray leaves the volume or reaches a transparency value of 1.0, the sampling loop is aborted. When all rays are sampled, the image plane contains a picture of the volume from the desired camera position.

3.2 CUDA Implementation

As basis for the implementation of the raycaster, an example of the NVIDIA CUDA SDK was used. Since it is an example, the code is structured simply and provides a solid foundation for more complex extensions. So the first part will give a description of the basic implementation and then of two custom extensions.

At first some basic requirements have to be setup. In order to store the resulting image, an OpenGL pBuffer is used since later on the displaying is done by OpenGL and it saves two data copies from CUDA (device) to the application (host) and again from the application to OpenGL (device). The CUDA block and grid size depends on the pBuffer size, so that for every pixel in the buffer one CUDA thread is generated. The volume is uploaded as a 3D floating point texture with normalized access.

At first the direction of the ray and the intersection points with volume have to be determined. The direction is simply a vector from the origin at (0,0,0) to (x,y,z) where x and y are the coordinates of the currently processed pBuffer pixel and z the distance to the (virtual) image plane. To determine the intersection points with the volume, I decided to use the ray-box intersection algorithm, because it does not need any modifications if the camera is inside the volume. In order to ease the texture look-up, I assume that the bounding box of the volume begins at the coordinate systems origin and stretches along the positive axes.

Starting at the near intersection point the ray is now sampled step-by-step at equidistant points. Since every sampling point is inside the volume and the volume stretches from (0,0,0) to its borders, the sampling position only needs to be divided by the volume size to get the normalized look-up position. Values received from the volume are automatically linear interpolated by CUDA. To sum up all alpha values along the ray, a front-to-back blending is used. A ray starts with a magnitude of 0.0 and the sampled values are blended by the following formula

$$alpha_{dst} = alpha_{dst} + (1 - alpha_{dst}) * alpha_{src} \tag{3.3}$$

where $alpha_{dst}$ is the current transparency of the ray and $alpha_{src}$ the looked up value at the sampling point. If $alpha_{dst}$ reaches 1.0 the traversal of the ray can be stopped, because there will be no more change. Afterwards the position is advanced to the next sampling point until the far intersection is reached and the resulting ray color saved to the pBuffer.

3.2.1 Transfer Function

Because the value of the voxels represent some physical property dependent on the acquisition procedure, they can not be just interpreted as transparency values like above. The resulting image would not present a suitable visualization of the volume. A common solution for this problem is the use of a transfer function. Instead of using the values directly, a n-dimensional transfer function maps values acquired from to the volume to a color (composed of red, green and blue) and transparency (alpha) value.

$$f(\vec{x}) : R^n -> R^4 \tag{3.4}$$

One aspect of a medical volume datasets is that similar tissues have similar absorption values, for example the air surrounding the patient has a really low magnitude, bones instead have a high one. The transfer function can now assign zero values to all areas except the range of the bones, what results in a 3D visualization of the skeleton. It is also possible to assign a color value to a specific range and thus highlighting different parts of tissue. However, the blending function has do be adapted in order to handle the color values correctly.

$$color_{src} = color_{src} * alpha_{src} \tag{3.5}$$

$$color_{dst} = color_{dst} + (1 - alpha_{dst}) * color_{src} \tag{3.6}$$

where $color$ is a 3-dimensional vector representing the red, green and blue color channels (RGB). The calculation of the alpha channel stays the same as in equation 3.3.

Unfortunately the design of a meaningful transfer function is not trivial. The simplest implementation uses a 1D look-up texture which maps values from 0.0 to 1.0 to a RGBA color value. This 1D function can than be modified by the user, e.g. through a histogram. Engel et al. [7] also describes some more complex transfer functions using also the gradient of the volume to find transitions between tissues and making it therefore easier for the user to select specific features (organs) from the volume. However, designing a transfer function may not be intuitive for the user. The (semi-) automatic creation of a meaningful transfer function is still an ongoing field of research and Most techniques only try to simplify the user interaction like described in König et al. [12]

3.2.2 Cutting Planes

Often the viewer is not interested in the whole volume but rather in some specific area. Of course a transfer function could be used which removes all features except the desired area. On the other hand it is quite uncomfortable to constantly change the TF for every area that should be examined. Another approach is to just "cut away" parts of the volume with a cutting plane. These are easy to implement by using the Hesse normal form (HNF)

$$\vec{p} \cdot \vec{n}_0 - d = s \tag{3.7}$$

where $\vec{p}$ is the location vector of an arbitrary point P, $\vec{n}_0$ the unit normal vector of the plane, d the distance between the plane and the coordinate systems origin and s the distance of the point P to the plane. If s is greater than zero, P is on the side of the plane the normal points to and on the other side if s is smaller than zero respectively. For s=0 the point is on the plane. For each sampling point the HNF is evaluated and checked for s>0. If so, the sampling is continued normally. Otherwise this sampling point is skipped and the next point examined. This way even more than one cutting plane can be implemented effectively by just evaluating the HNFs for each plane at the sampling positions.

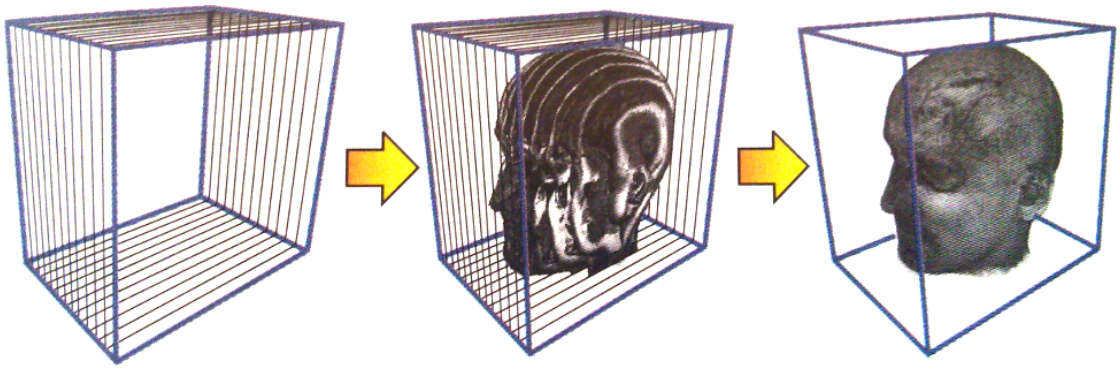


Figure 3.3: Schematic functionality of a object-based volume renderer [7]

3.3 Texture Slicing

For completeness another volume rendering technique should be mentioned. This method makes use of the fact that the 3D volume can be represented as a stack of 2D slices [7]. These 2D slices are rendered as quadrangles with a transparent 2D texture on it (Fig. 3.3). So the first step is to setup the proxy geometry with all the 2D slices. Afterwards the textures are mapped onto the slices. In the final step the slices are blended with a back-to-front blending for example.

The advantage of this technique compared to the raycaster is that it only needs standard Fixed-Function OpenGL commands and has therefore a high performance [7]. However, in a not optimized implementation the image quality is reduced because of heavy aliasing effects which are caused through the fixed distance (sampling rate) of the texture slices. Of course this object-based approach also supports advanced techniques like a transfer function, but in this thesis a raycaster will be used because it produces a better image quality and is easier to adapt to different situations.

3.4 Flow Visualization

One important subfield of visualization is the so called flow visualization. It deals with the dynamics of fluids like water and gas and is used in many different areas of interest including the automotive

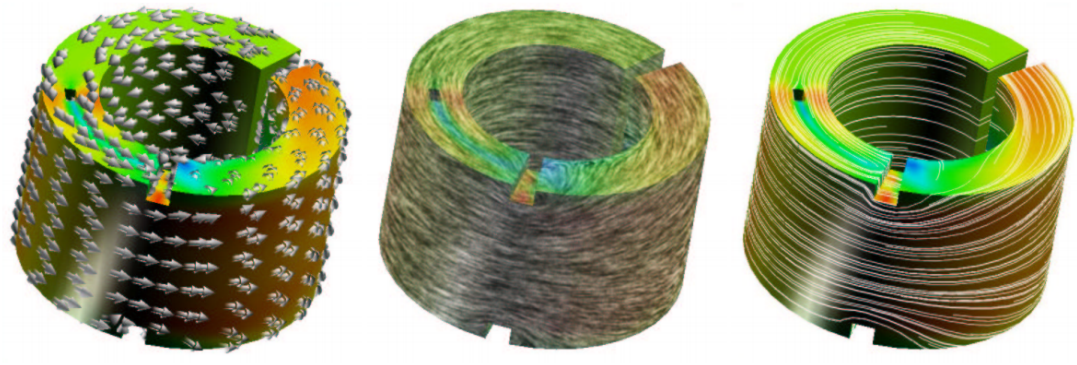


Figure 3.4: (left) direct visualization by the use of arrow glyphs, (middle) texture-based by the use of LIC, and (right) visualization based on geometric objects, here streamlines. Images from [14]

industry, aerodynamics, meteorology and medical visualization [14]. Since the motions in the human body are also a kind of flow, it makes sense to review the important aspects of flow visualization (FlowVis) for the visualization of breathing motion. FlowVis can be roughly split up into 4 categories:

1. Direct flow visualization
2. Geometric flow visualization
3. Dense, texture-based flow visualization
4. Feature-based flow visualization

FlowVis always works on a vector field either in a discrete or analytic representation. The field can be 2- or 3-dimensional, but some visualizations are only useful for 2D vector fields. In the following chapter a short introduction to each of these categories is given.

3.4.1 Direct flow visualization

The most straight forward approach of vector field visualization is the direct flow visualization. On the one hand arrows are used to visualize the direction and magnitude of the vector field on a certain position (Fig. 3.4 left). This arrows can be layed out on a regular grid or varied in density depending on local criteria of the field. The advantages of

this technique are that they are easy to understand, simple to draw and usable both in 2D and 3D. Another part of direct FlowVis is the color coding of the magnitude or velocity of the field. Thereby only scalar values are visualized by a certain color which is useful when the direction of the field is not of interest. Color coding is generally unsuitable in 3D space but can be used on surfaces of 3D objects. Laramee et al. [14] refers to this as 2.5D.

3.4.2 Geometric flow visualization

Unlike the direct approach, which only shows local features of a vector field, the geometric visualization first integrates the data set and displays the results with geometric objects. Namely there are three different types: stream, path and streak lines.

stream lines describe an instantaneous particle path in a steady flow [4]. A steady flow does not change over time, unlike an unsteady one. Stream lines are calculated by solving an initial value problem of an ordinary differential equation [4]. (Fig. 3.4 right)

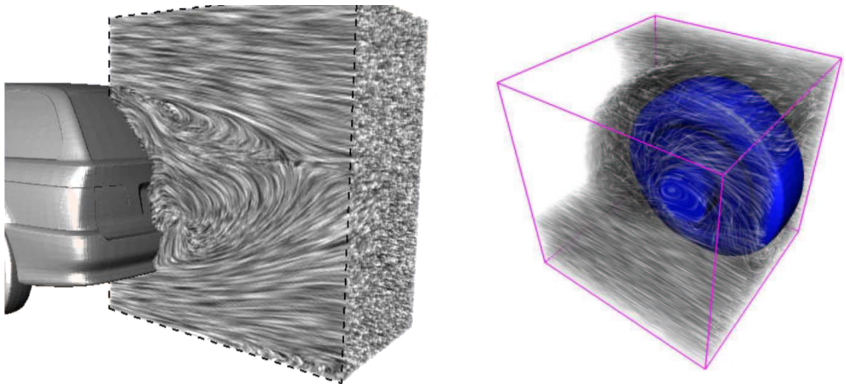
path lines are similar to stream lines, but are the trajectories of a particle in an unsteady flow. Path lines are also the solution of an initial value problem.

streak lines are often compared to dye injected into an unsteady flow [4]. Particles are released from a fixed position into the flow and then follow the current flow direction.

In a steady flow all three visualizations yield the same result. These geometric approaches are especially suited for fluid or gaseous flows.

3.4.3 Dense, texture-based flow visualization

Another important technique in this categories is the Line Integral Convolution (LIC). The basic primitive here is a noise texture: the texture is convolved using a kernel filter in the direction of the underlying vector field [14]. It is like smearing the noise spots along the flow.



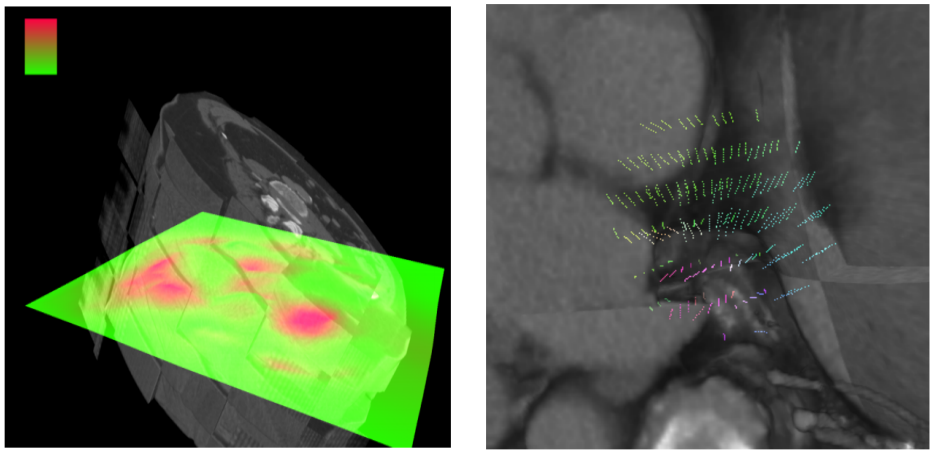
(a) (b)

Figure 3.5: Image (a) shows a 3D LIC of a aerodynamic flow. Image (b) shows a 3D LIC of a flow around a wheel. Both images are courtesy of Rezk-Salama et al. [22]

There are many different types of LICs but all are most suitable for 2D or 2.5D visualization. 3D is also possible but the perception may not be intuitive for the user. The visualization in Figure 3.5 (a) is three dimensional but only the flow on a two dimensional plan is really visible. The example in Figure 3.5 (b) uses a volume renderer with a transfer function to address the perceptional problems. More about LICs and other texture-based approaches like Spot Noise can be found in [14].

3.4.4 Feature-based flow visualization

Feature-based flow visualization is actually no real visualization by itself but an extension to the three previous ones. Before the visualization the dataset is scanned for specific flow features – like important phenomena or topological informations [14]. The visualization only uses this extracted features, thereby removing unnecessary information in the dataset which is often a desirable effect. Unfortunately the process of extraction is not trivial and dependent on the area of use, certain features might have different meanings. In general, different mathematical approaches are applied to find points of interests inside the data, for example by computing the eigenvalues and -vectors of the velocity gradient tensor. More on this topic can be found in [21]



(a) (b)

Figure 3.6:

3.5 Other previous work

Vill [28] describes different methods for the visualization of motion in a 3D medical volume. The thesis also uses a raycaster to generate a 3D image of the volume. However only single 3D volumes are discussed and not 4-dimensional ones. The visualization is split into four groups: grid-based-, particle-, plane-based- and 2D view effects. Furthermore some selective rendering techniques are discussed. While most of the visualizations convey intuitive understanding of the movements inside the body, they lack a direct integration of the volume, which negatively affects the perception (see Fig. 3.6). Thus, one of the goals in my thesis was to integrate the visualization more into the raycaster and thereby improve the perception of the relations between the volume and the occurring motions inside the volume.

Chapter 4

Visualization of 4D Breathing Motion

Before beginning with the actual implemented features, some basic setups are necessary. As dataset I am using the POPI-model (Point-validated Pixel-based Breathing Thorax Model) [26]. It contains a high resolution 4D CT scan of the thorax at 10 different time steps. Beside the original images, also preprocessed images are available, which have a significant smaller size but still contain all patient features. The ten 3D volumes represent one average breathing cycle, from the begin of the inspiration to the end of the expiration.

Additionally the POPI-model also provides a vector field estimating the breathing motion. All vectors represent the displacement of one voxel in regard to a reference volume – in this case the volume of the second time step. Since the vector field to the reference volume would be always zero, it is not part of the dataset. However, to avoid inconsistencies in the presented methods, a vector field with no displacements is used in this case. Again the vector fields were calculated with two different methods which are only slightly different. The precision of both fields was thoroughly evaluated using landmarks identified by medical experts in each of the 3D volumes [26].

The presented shader need at least GLSL in version 1.40 and therefore a OpenGL 3.1 capable graphics card. The CUDA raycaster should work with CUDA version 1.0 but was only tested with a 1.3 compatible GPU. Though it is not essential that the raycaster is implemented in CUDA. The rest of the application is written in C++.

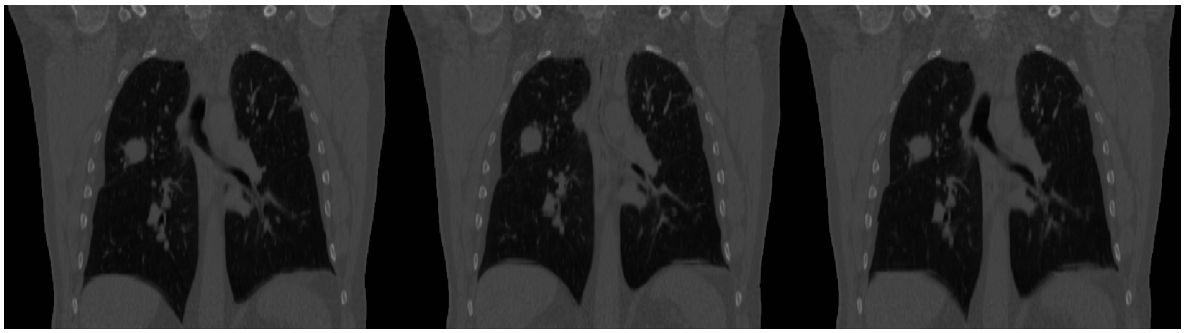


Figure 4.1: slices of the CT volume at different time steps

4.1 2D slice rendering

First I will present a rather simple motion visualization which is based on a 2D slice (Multi-Planar-Reformatting or MPR). In order to get a 2D plane from the 3D volume a fragment shader is used which renders into a texture. The shader gets the volume texture, the physical size (in millimeter) of the volume and a 4D transformation matrix as input. The later is an affine matrix describing the rotation and translation of the slice in physical coordinates. The basic setup is a quad filling exactly the whole screen and with the 3D texture coordinates (0,0,0), (1.0, 0.0, 0.0), (1.0, 1.0, 0.0) and (0.0, 1.0, 0.0) for the four corners in counter clockwise direction starting at the lower left.

The fragment shader now uses the interpolated texture coordinates and transforms them into physical coordinates by component-wise multiplication with the physical volume size. Now these coordinates match a slice from the left-lower-front to the right-upper-front of the physical volume. By multiplying the transformation matrix, the slice is translated and rotated to its desired position. In the following this will be referred to as volume coordinates or volume space (according to the OpenGL spaces). Positions in volume coordinates can be divided by the volume size to get normalized look-up coordinates for the 3D texture the volume is saved as. With these normalized volume coordinates the fragment shader can look up the volume value and return it as the fragment color.

In order to animate the slice over time, only the texture reference

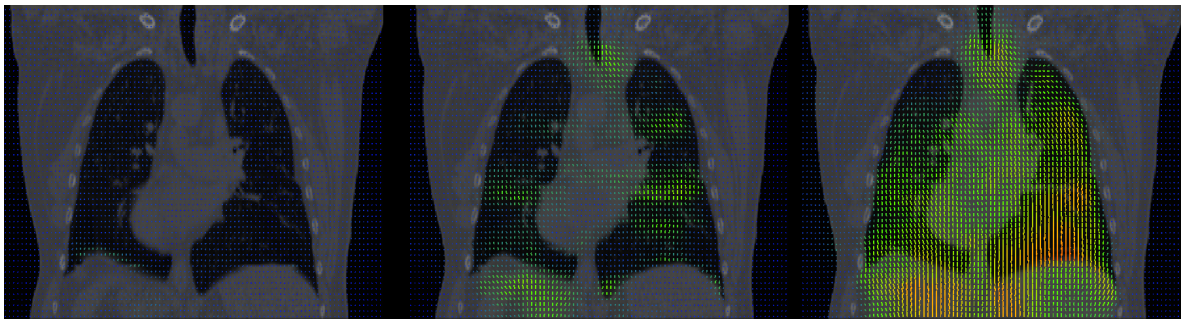


Figure 4.2: slices of the volume with color coded vector visualization at different time steps

has to be updated every time step. Just by the animation of a few time steps the viewer already gets an impression of the occurring movements (see Fig. 4.1). To enhance the perception of motion direction I decided to add a direct visualization based on vectors which are placed on a uniform grid. To implement this vector grid with a common VBO would rather be slowly because at every frame the CPU has to consult the vector field data, update the vectors accordingly and then upload it to the GPU.

To save CPU cycles I used a geometry shader to generate lines which represent the vectors. The shader gets a grid of points in physical coordinates uniformly distributed from (0,0,0) to ($PhysVolSize_x$,$PhysVolSize_y$,0) as input. It also needs the physical volume size, the transformation matrix of the slice and of course the vector field. Since the vector field exists only of 3 dimensional floating-point vectors, it can be uploaded as a 32bit floating-point RGB texture - encoding xyz in rgb. The geometry shader now works similar to the fragment shader generating the slice: At first every vertex is transformed to volume space by multiplying the transformation matrix. This position is then emitted as origin of the vector. On the second step the normalized volume coordinates of the vertex are used to get the displacement from the vector field. Since the volume and the vector field have the same physical size, the volume coordinates can also be used for a vector field look-up. The displacement added to the origin forms the second emitted vertex. Both vertices together now build a line from the original position of a voxel in the reference volume to the current one. Again to display another volume frame only the

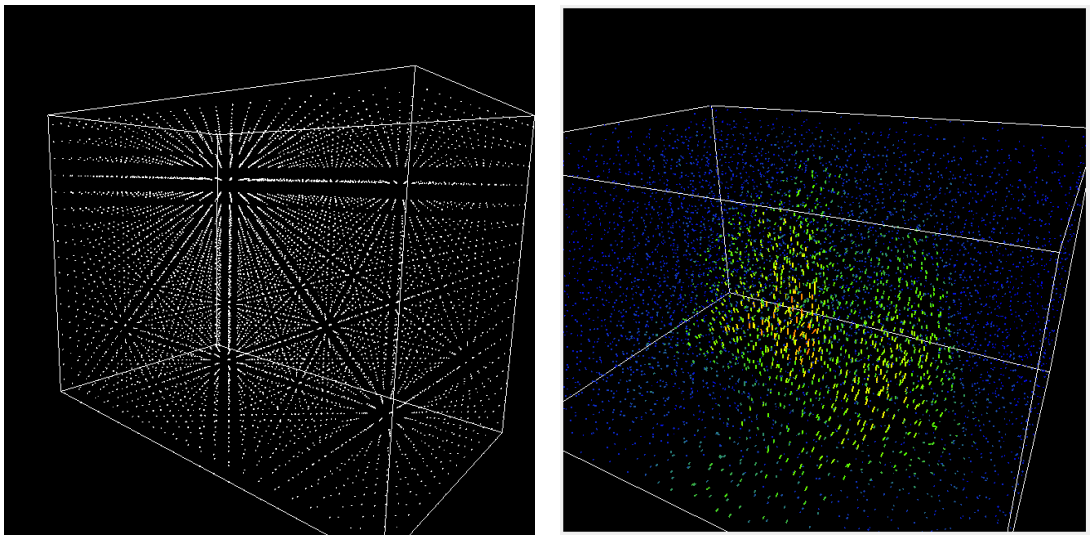
texture IDs of the volume and vector field have to be updated.

4.2 3D vector visualization

The previous 2D vector visualization can be easily extended to a 3D one. Since the shader works on all kind of points, the plane of dots is just replaced by a cloud of dots, e.g. a uniform grid. Unfortunately this does not look very good. If the grid is too dense, it generates aliasing effects which are visual not appealing like you can see in figure 4.3(a). The effect can be reduced by adding a jitter value to every dot instead of distribute them evenly (Fig. 4.3(b)).

But still the visualization is not really meaningful. There are too many dots and the magnitude of the vectors is not big enough to yield a good perception of the occurring motion in the field. By just increasing the spacing between the vectors, the information of the values between two vectors will be lost or if one vector represents the average of all surrounding vectors, high and therefore significant magnitudes are smoothed. In order to enhance the perception of important vectors, a color coding is more suitable than a change of the vector spacing. The color of a vector is therefore determined by its length. The shader only needs a look-up table for the color and the maximal and minimal magnitude occurring in the vector fields. The look-up table is a 1D texture which maps the vector length to a color. The look-up position for each vector is obtained by normalizing the length depending on the overall minimum and maximum length, meaning that all magnitudes are in the range of 0.0 to 1.0, with 0.0 representing the smallest value and 1.0 the biggest. As for the colors, all combinations can be used. A transition from blue over green to red performs empirically well, because the blue, unimportant values are dark so that the red, more important ones stand out. Figure 4.3(b) shows the color coding on a jittered grid.

With only the visualization of the vector field the viewer can identify areas of major movement but not to which parts of the human body these motions are connected. Of course the MPR of the 2D approach can also be shown as a plane in the 3D view. But then the user



(a) (b)

Figure 4.3: Image (a) a small grid results in aliasing effects (b) same grid with jittered dots and color coding

always has to move the plane to the area of interest and also would only get about the same amount of information he gets from the 2D view.

Consequently the best way to show the volume in 3D is with a raycaster. Since the presented raycaster in chapter 3.1 is implemented in CUDA but the rest of the application in OpenGL, it had to be ensured that the volume size and the camera angle are both render exactly the same with both APIs. The first problem that arises is that the OpenGL camera is always in the origin and the scene is rotated around the camera. In the raycaster on the other hand, the volume is at the center and the camera is rotating around it. Since one is just the inverse of the other, the raycaster adjusts its camera with the inverse of the OpenGL ModelView matrix. But even if the camera positions match, the resulting volume sizes may not. Of course both volumes have the same physical size but this size is distorted by the perspective transformation. In OpenGL the perspective transformation depends on the field of view and the aspect ration. In the CUDA raycaster this parameters may not seem obvious but are represented by the distance and size of the virtual image plane (see figure 3.2). Because the raycaster shoots

one ray per pixel in the pBuffer, the aspect ratio is just determined by the aspect ratio of the pBuffer. So as long as the pBuffer has always the same size as the OpenGL window (or at least the same aspect), these two parameters match. The field of view of the raycaster depends on the distance of the image plane to the camera. This distance is calculated in relation to the field of view's angle

$$d = \frac{w}{\tan(\frac{\alpha}{2})} \tag{4.1}$$

where d is the distance of the image plane, α the angle describing the field of view and w the width of the image plane. An empirical good value for the field of view is between 50 to 60 degree.

At the end of both render steps, two different images of the same space are acquired, which have to be combined into one image. One solution uses the image of the volume as background. After the raycaster generated the volume image, it is mapped onto a quadrilateral which fills the complete viewport. An orthogonal projection is the most suitable way to do this. Afterwards the depth buffer is cleared with 1.0. Now the OpenGL scene is rendered with the appropriate perspective projection. Since the depth buffer was cleared, all objects are always on top of the quadrilateral which displays the volume, but because both scenes were rendered from the exact same viewpoint it seems that both occupy the same space. If the OpenGL scene contains transparent objects, alpha blending needs to be enabled to produce the correct result.
Of course this also works the other way around: first the OpenGL scene is drawn into the framebuffer and afterward the depth buffer set to 1.0. Then the raycaster generates the texture with the volume which is then again mapped onto a screen size quadrilateral and blended with the scene.

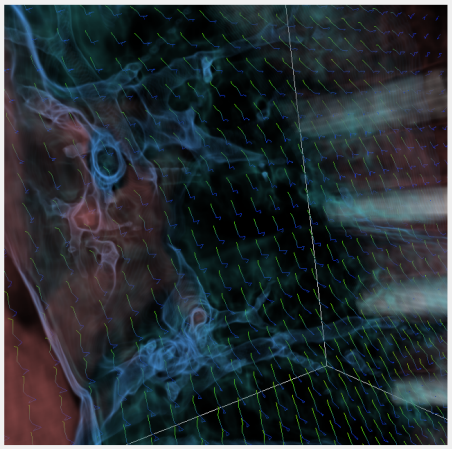


Figure 4.4: motion path visualization between the heart and the left lung

4.3 Motion Path Visualization

The organ and tissue motion during a breathing cycle is complex and shows great variety between regions. For example, the motions in the lungs are relative consistent, while the movements in the heart are more chaotic. Since the vector visualization only shows the displacement compared to the beginning frame, it is hard to tell on which way one voxel reached the current displacement. However this information would tell the viewer the overall movement of one voxel. Path or stream lines (see 3.4) are designed for this use case. Unfortunately these approaches are more suited for continues flows like liquids and not periodic flows like the breathing motion. Most of the existing flow visualizations inject some kind of particle into the flow which then follows the vector field. Used on the breathing motion vector field, the particles would not distinguish between different organs and would flow through the whole body. So instead of a global evaluation of the flow a local one would be more suitable.

Since the vector field already contains the information of the position of one voxel in all time steps, the path can easily be tracked. In order to draw a path from the starting position in the first frame to the position in the current frame, a line needs to pass through all position between those two frames. Therefore only the starting position of the path must be set by the application, the rest can be drawn in a geome-

try shader. So like before the vectors, the geometry shader works on a cloud of points and generates a path from each point. Now instead of just the vector field texture of the current frame, the shader needs the vector fields of all frames. Unfortunately GLSL in version 1.4 does not support arrays of textures so all textures need to be defined explicitly. The maximal emitted vertices need to be set in the shader layout to the amount of frames.

The shader works as follows: First the incoming vertex is transformed to volume coordinates, referred to as $origin$. Now a loop is started with one iteration for every frame. At every iteration $origin$ is used to look up the $displacement$ from the vector field of the iterated frame. The $origin$ and $displacement$ are added, transformed to clip coordinates and emitted. The resulting primitive is a line strip connecting all points one voxel passed through until the current frame.

Of course this rather crude result can be enhanced in different ways. In order to spot large displacement magnitudes more quickly, the path can again be color coded. This time there are two different approaches: On the one hand the color can be dependent on the overall length of the path by simply summing all displacement lengths up. However this can lead to delusive results because paths containing only medium displacements would have the same color as paths with one extreme displacement and several small ones. Also to calculate the complete length of the path, the shader first has to look up all displacements from the vector fields, compute the length and then emit the individual vertices. A simpler approach would be to color code each point of the line with the associated displacement, so that a vertex can be emitted right after the displacement look-up. The result can be seen in figure 4.4.

A problem is that the shader currently needs references to all vector fields at once, which may exceed the maximum supported textures in a shader. Also especially the breathing motion is a cyclic motion which is not represented very well by the visualization. The path is build up step by step with every new frame. When the frames start over, the path "plobs" back to the beginning length. Both problems can be countered with some modifications. The basic idea is that the line

strip representing the path only contains a fixed number of lines which is smaller than the amount of frames. Instead of adding every frame one line segment, at every frame only the last e.g. four line segments are drawn (the predecessors of the very first frame would be the last frame). Consequently the drawn path moves cyclic like the motion field and the shader does only need references to a part of the vector fields.

4.4 Scene Occlusion

Because the OpenGL scene rendering and the CUDA volume rendering are two separate procedures, there is no realistic occlusion of objects. This effects the depths perception of certain items. If for example geometry is always rendered on top of the volume, the viewer cannot tell if it is in front, behind or inside the volume. By moving the camera he can at least determine the rough position of an object. This can be enhanced by drawing a wire frame cube around the volume, so that the viewer can spot a more precise object position in relation to the volume. Unfortunately the occlusion problem remains. Even if the user can now see that the object is somewhere in the middle of the volume, he cannot tell for sure that it lies exactly e.g. in the heart.

In order to implement a real occlusion of geometry by the raycasted volume , rays which would hit an object must terminate earlier. So the only required information needed is the distance between the camera and the object. This information determines how long a ray can be at most. A method using the depth buffer of the scene is described by Engel et al. [7]. It first renders the scene into a depth texture and then uses the depth to terminate the ray - object intersection. Unfortunately it is currently not possible to use a depth texture directly in CUDA. OpenGL uses one of the GL_DEPTH_COMPONENT* internal formats which are all not supported in CUDA. It would be possible to copy the texture back to the CPU and upload it again with a supported CUDA format but this would heavily impact the performance. A solution to this problem needs a little modification of the vertex shader. The idea

is to save the distance of a vertex to the camera in the alpha channel of the vertex color. These values are then interpolated over all fragments and saved in the frame buffer. The distance calculation is done in clip space so that the result will be linear between the near and far clipping plane in contrary to the normal depth. The algorithm works as follows:

- the scene is rendered into the frame buffer object. After the projective transformation in the vertex shader, the z coordinate of the vertex already describes the distance to the near clipping plane. Since color values in OpenGL have to be normalized between 0.0 and 1.0, the distance also has to be clamped to this range before it can be saved in the alpha channel. All relevant distances are between the near and far planes, so the final result can be calculated with $distanceToCam = (gl\_Position.z + near)/(far + near)$
- after the OpenGL scene is rendered, the resulting texture of the frame buffer object is shared with the CUDA raycaster. Each ray now fetches its maximum length by multiplying the alpha value of the scene texture with $(near + far)$. During the ray traversal, this value is compared to the distance between the cameras' origin and the current sampling position. If the distance is greater than the maximum length, the traversal is stopped and the ray color written to the output buffer.
- at last the raycasted image and the scene texture are blended by OpenGL. Of course the raycasted image has to be on top of the scene, to get the desired result. It would also be possible to do the blending inside the raycaster, since the raycaster also has access to the color values of the scene. However only objects inside the volume would be visible and it is doubtable that this approach would be faster.

An example to this technique is shown in figure 4.5. Of course this does not work correctly with transparent objects. If two transparent objects overlap, they are blended to one value with one depth information. The raycaster on the other hand would need both color and depth values separately because the samples of a ray passing these

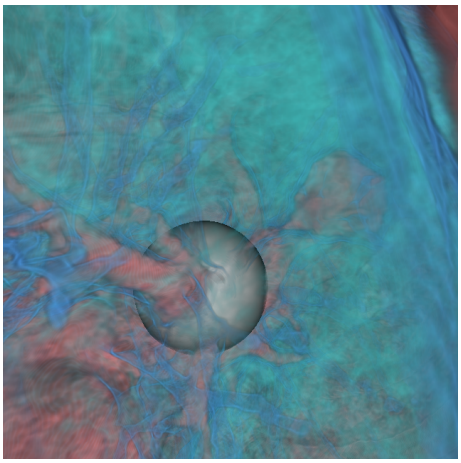


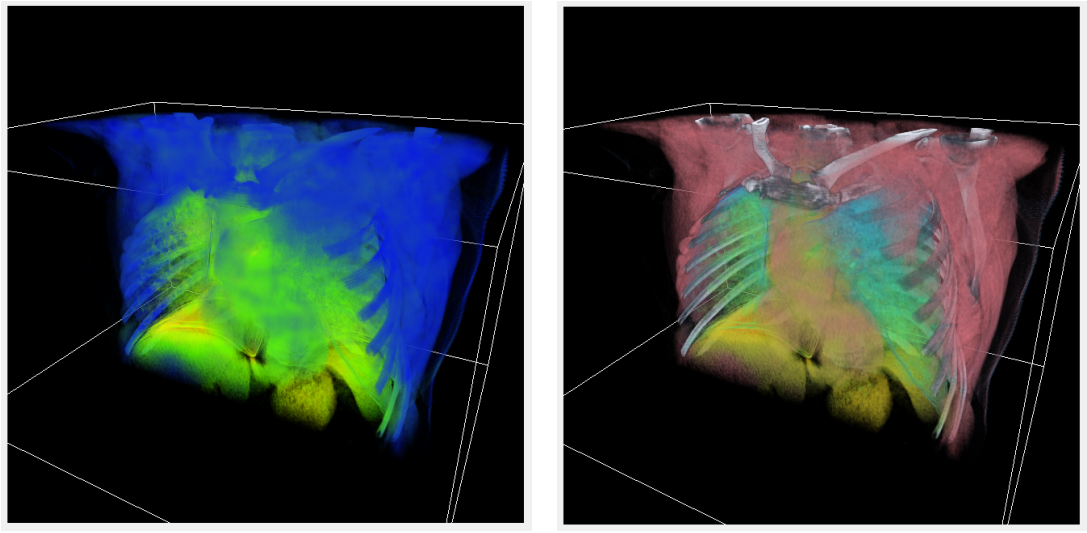
Figure 4.5: a sphere partially occluded by the volume

two objects can lie in between both objects. So the color information of the both objects has to be inserted at the current position between the samples, before all is blended together. A solution allowing transparency of objects can be found in the work of Bavoil et al. [2].

4.5 Volume coloring

The direct vector visualization has still a few drawbacks. Like mentioned above the grid needs to be very dense if no significant information should be lost. Even with lower density the single vectors are hard to spot when they are rendered together with the volume. So to get an overview of the all the motions in the volume the direct vector visualization is not really suitable. So an idea is to use the volume directly for the visualization of the movements. Until now I only used a 1-dimensional transfer function to transfer the magnitude of the CT volume to a color and a transparency value. But it is also possible to deploy a higher dimensional function. For example Engel et al.[7] joins the magnitude and the gradient of the volume to locate transitions between different tissues.

With a 2-dimensional transfer function both the transparency of the volume and a color coding of the vector field can be combined. Since the vector field is already accessible as texture for the CUDA



(a) (b)

Figure 4.6: (a) the color coding replacing the volume color (b) color coding and the volume color blended

volume renderer, the motion vector at all sampling points along a ray can be determined with an additional texture look-up. Instead of passing only the intensity of the volume, the intensity and the magnitude (length) of the vector are now passed to the transfer function, which determines a new color and transparency based on these both parameters.

Unfortunately it is even harder to design a good 2D transfer function than a 1D one. Then again the 2D TF could also be split up into two 1D TFs: The first one gets the CT intensity as input and calculates the transparency, the other one gets the vector length and returns the color. The second one can basically be the same 1D look-up texture used for the color coding of the vector and the first the 1D transfer function used previously. Figure 4.6 (a) shows the results on the CT scan of the thorax. The colors code the magnitude from blue (lowest) over green and yellow to red (highest). The TF for the transparency highlights the bones, the heart, the lungs and the liver partially. For example it is clearly visible that the biggest movement is above the liver in the left-lower corner of the volume. Apparently there is no motion in the center of the liver so it can be concluded that the liver is

compressed.

Unfortunately the former coloring of the volume is completely overridden by the vector color coding, which is essential to highlight different parts of the body. Especially in the beginning time frames where no deformation is occurring, the volume of the previous example is mainly blue and it is hard to distinguish between different features. To counter this the color of the volume and the applied color coding can be blended. Therefor the look-up texture for the color codes needs an additional alpha channel. It should describe how important a color is regarded. The blending is done by following formula

$$color_{sample} = color_{mag} * alpha_{mag} + color_{vol} * (1 - alpha_{mag}) \tag{4.2}$$

$$alpha_{sample} = alpha_{vol} \tag{4.3}$$

mag is the color and alpha of the magnitude transfer function and vol the color and alpha of the intensity TF respectively. So the final color is dominated by the magnitude's alpha. In contrast the sample alpha is only controlled by the intensity transfer function. 4.6 (b) shows the volume with a linear alpha distribution from 0.0 for blue and 1.0 for red which serves quite well.

Of course special care has to be taken with the color selection. A red magnitude coding will not be visible on a red colored organ. In figure 4.6 (b) this problem is avoided by using colors with a low saturation for the volume and colors with full saturation for the coding. Another way would be to use separated color spaces for the two transfer function. For example red and white for the volume TF and blue and green for the coding TF.

It should also be mentioned that the color coding does not really work on a vector field but rather on a scalar field. So in the above description the vector field is in fact implicitly transformed into a scalar field by calculated the scalar length for each vector. But there are also other features of the flow that could be used for visualization, like the divergence of the vector field.

$$div(\mathbf{F}) = \nabla \cdot \mathbf{F} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z} \tag{4.4}$$

The divergence describes the flow in or out of a region by a scalar value. Source regions, where the field is pointing outwards, have a positive value, sink regions, where the vector field is pointing inwards, are negative. The idea was, that tissue is stretched at source regions and compressed at sink regions. But the visualization did not verify this assumption and also did not yield a useful visualization at all.

4.6 Geometry Deformation

The above color coding method is good to quickly spot areas of motion but not to examine them closer. So the viewer can only tell that something moved but not exactly where to. Of course he can take a closer look at the area of interest in order to estimate the motion from the animation of the volume. Alternatively the viewer uses the former direct vector visualization to examine details. This solutions are satisfying if the area of interest is rather small and the attention is not paid on the relation of different motions, like the movement of adjacent organs. A concept for an extended local visualization is inspired by the deformable grid described by Vill [28]. Now instead of a grid, a deformable mesh is used. A polygon mesh is a collection of triangles or other primitives which together form some kind of object. In OpenGL a mesh is represented by vertices which form the triangles and is therefore stored as a vertex buffer object. For visualization the mesh is now placed by the user at the area of interest. At each time step of the animation, the mesh is deformed according to the underlying vector field, so if the field at the upper part of the mesh is facing to the top and the lower part to the bottom, the mesh would be stretched vertically.

The implementation uses only the vertex shader. Besides the ModelView- and the Projection-Matrix, the shader also needs a reference to the vector field (as a 3D texture), the transformation and

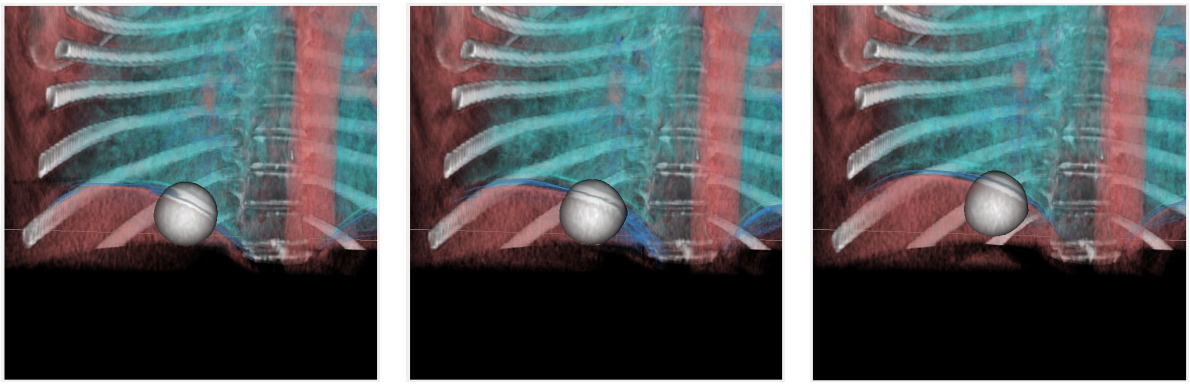


Figure 4.7: geometry deformation at different time steps

the physical volume size. At first the vertex is transformed into volume space and then used to look up the displacement from the vector field. The displacement is simply added to the vertex which, applied to the complete mesh, distorts every vertex along the vector field. After that the vertex shader can execute some standard calculations, like the ModelView and Projective transformation or the evaluation of a local illumination model.

Figure 4.7 shows the deformation on a sphere at three different time steps. The mesh consists of about 5000 triangles and is positioned at the transition between the right lung and the liver. Beside the deformation, the vertex shader in this example also evaluates the Phong illumination [20] with a light source that is always emitting from the camera origin. Beside a sphere all other kinds of meshes are possible, even with lower amounts of triangles. For test purposes I also tried the mesh of a monkey head which performed surprisingly well. I suppose that this effect is similar to Chernoff faces and the psychology of facial recognition [15]. An examination of this assumption would be interesting for future work.

4.7 Performance evaluation

The graphical user interface (GUI), designed with the Qt framework, is displayed in appendix **??**. It contains two OpenGL context which

shared resources at the left and right of the center and various interactive parameters for the visualization in the lower tab bar. The menu provides functions to load volumes and vector fields, changing the transfer function, the look-up table used for the color coding and an option to enable or disable animations. The context on the left displays a raycasted image of the volume. The raycaster supports a 1D transfer function, two clipping planes and the camera can be moved freely around and even inside the volume. The right context shows a MPR of the first clipping plane used in the raycaster. Both context are using the same volumes and vector fields and are synchronized so that they always display the same timestep. Through the lower tab bar the user can enable the different kinds of visualization and also mix them together.

The transfer function can be modified by the user in a simple GUI you can see in appendix **??**. The x-axis corresponds to the values of the volume and the y-axis to the alpha the values should be transferred to. By placing colored dots the user can now modify the output of the function. Values between the dots are interpolated. The effect on the volume renderer is displayed in real time so that the user can easily try different values to identify parts of the body. It would certainly be possible to preassign the transfer function to the volume data which would save one texture look up at each sampling point but would also mean to lose the interactivity and if colored would take up to 4 times more memory (4 color channels per voxel instead of 1).

The whole setup was tested on a Intel Core 2 Quad CPU with 4 cores running at 2.50 Ghz, 4.00 GB of RAM and a NVIDIA GeForce GTX260 with 896 MB VRAM. The first test case investigates the performance of the raycaster with three different resolutions for the pBuffer. The results are displayed in figure 4.8 (a). Since a ray is casted for each pixel, the resolution indicates the total number of rays. The axis of abscissas represents four different ray sampling distances while axis of the ordinate shows the average achieved frames per second. Smaller sampling distances improve the image quality of the raycaster and reduces aliasing effects. However the sampling rate has a significant impact on the performance since the texture look-ups for the volume

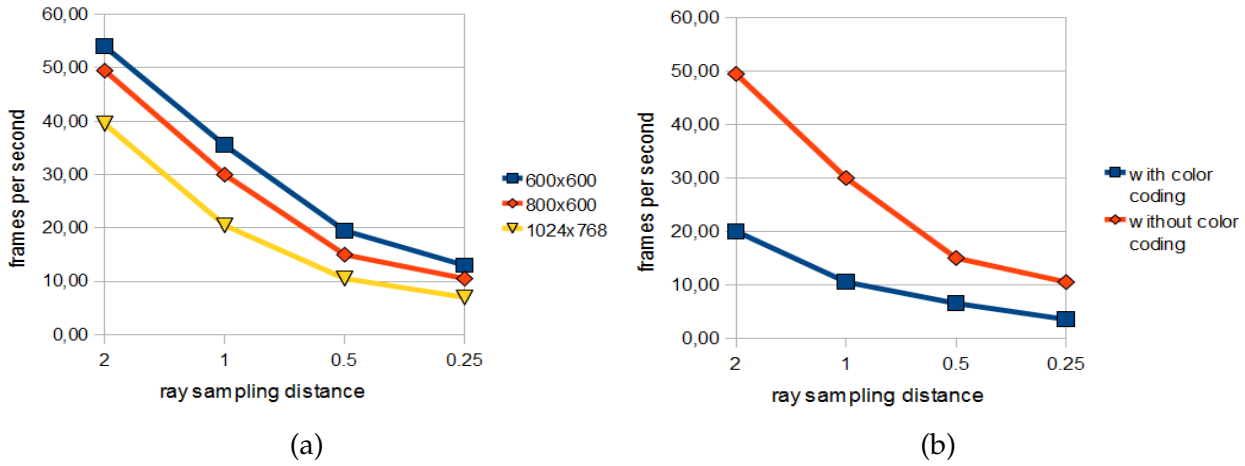


Figure 4.8: (a) test results for the raycaster (b) test results volume coloring visualization

and the transfer function performed at each sampling position are the most costly operations in the whole raycasting process.

When the raycaster also checks for scene occlusion the total frame rate drops around 1-2 fps depending on the resolution. Since the texture look-up to get the scene depth is only performed once per ray, the occlusion method is independent from the sampling rate. Big objects can even enhance the frames per second because rays may be terminated earlier.

The volume coloring visualization was tested with a fixed resolution of 800x600 (see Fig. 4.8 (b)). Since the vector field has to be consulted for every sample, the performance depends primarily on the sampling rate. There are totally two additional texture look-ups per sample, because beside the vector field also the color codes needs to be accessed at every sample. This explains why the performance is halved.

Rendered together with the raycaster, the vector- and path visualization on a grid with 49.000 points have no measurable impact on performance. The grid size relates to a spacing of 20 mm between two points, an empirical well performing size, where still enough information is displayed but the viewer is not overwhelmed by the sheer amount of vectors/paths. On a uniform gird of 482x360x242 mm (the physical size of the used volume) the performance drops rapidly un-

der 60 fps when the spacing is smaller than 8mm for the direct vector visualization respectively around 14mm for the motion path depending on the length of the path.

Chapter 5

Conclusion and future work

All presented visualizations by them self have some kind of limitation, be it a limited perception of details or the absence of certain informations. However combined, the different visualizations compensate their individual lacks. E.g. the volume coloring gives a good overview of all occurring motion but lacks informations about the direction of single movements. On the other hand the motion path visualization shows the exact movement in detail but can be confusing in a large scale especially when partially occluded by the volume. Together the viewer can first identify interesting regions through the volume coloring and examine them closely with the path motion visualization.

Especially the geometry deformation visualization is designed to be used in an animated framework. But also all other methods gain advantages through animations. The differences between two time steps are then much easier to spot than when the steps are displayed static like side by side in figure 4.4. Also the visualization should be interactive for the user. Like mentioned above the combination of different methods works best when the viewer can activate or deactivate them on demand because more than two active visualizations would be more confusing than helpful.

Of course these are only concepts of possible visualizations and there is still some future work before using them in a productive environment. On the one hand there is the raycaster performance. The presented raycaster uses nearly no optimization and therefore the image quality must be reduced to achieve interactive frame rates. However

realtime volume rendering is still an active field of research and there are already many different optimizations. For example the current implementation also samples rays in empty regions of the volume (e.g. regions of air), which produce no real information. A common solution to this and other problems is called bricking, which divides the volume into smaller chunks called bricks [24]. Empty bricks are removed and are not passed by rays.

Another current problem is the memory usage. Since beside the volumes also the vector fields must be uploaded to the GPUs' VRAM, large volumes are problematic. The POPI model for example consists of 10 volumes with a size of 482x360x141 voxels and a resolution of 16-bit. These alone already consume around 470 MB of the available VRAM. Because one volume is used as reference, there are only 9 vector fields with a size of 235x176x141 vectors. Each vector contains three 32-bit floating point numbers, which sums up to about 600 MB. So together the graphic hardware should at least contain 1280 MB of VRAM to also hold other, minor variables and textures. Of course the volumes and vector fields can be down sampled but only with a loss of information. Also the POPI model only contains the thorax and not the whole body. Complexer volumes would need a more sophisticated memory management. Basically only the currently displayed volume and vector field needs to be hold in VRAM. So while the current volume is displayed by the GPU, the data for the next frame can be uploaded to VRAM by the CPU, which is called asynchronous streaming [27]. This technique would reduce memory consumption but should not affect the overall performance of the application.

All visualizations are designed to be used interactively, however a special graphical user interface is missing. A GUI should be intuitive to use and help the user by his examination of the volume. For example it would be nice if the user just can click on an area of interest in the volume to display the geometry deformation visualization instead of moving the mesh to this position by entering coordinates. Also the visual presentation could be improved. The path generated in the path motion visualization is currently quite angled and would properly look more appealing when displayed as a curve. The depth percep-

tion could of course also be enhanced with stereoscopic 3D. Since the visualizations were not tested by any members of a medical staff, a usability study would also be interesting for future work.

Chapter 6

Appendix

Ich hab alles geschaft

Bibliography

[1] Ati stream computing. Tech. rep., ATI Technology.

[2] BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., COMBA, J. L. D., AND SILVA, C. T. Multi-fragment effects on the gpu using the k-buffer. *Symposium on Interactive 3D Graphics* (2007).

[3] BOARD, O. A. R., SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1 (6th Edition)*, 6 ed. Addison-Wesley Professional, 2007.

[4] BÜRGER, K., SCHNEIDER, J., KONDRATIEVA, P., KRÜGER, J., AND WESTERMANN, R. Interactive visual exploration of unsteady 3d flows. *Eurographics/ IEEE-VGTC Symposium on Visualization* (2007).

[5] BROWN, P., AND LICHTENBELT, B. Nv_geometry_shader4. Tech. rep., NVIDIA, 2009.

[6] CHITTARO, L. Information visualization and its application to medicine. *Artificial Intelligence in Medicine 22* (2000).

[7] ENGEL, K., HADWIGER, M., KNISS, J. M., REZK-SALAMA, C., AND WEISKOPF, D. *Real-Time Volume Graphics*. A K Peters, Ltd., 2006.

[8] FERNANDO, R. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, 2003.

[9] FERNANDO, R. *GPU Gems*. Addison-Wesley Professional, 2004.

[10] HOBEROCK, J., AND TARJAN, D. Introduction to massively parallel computing. Tech. rep., Stanford University, 2008.

[11] KAY, AND KAYJIA. Ray - box intersection, April 1998.

[12] KÖNIG, A. H., AND GRÖLLER, E. M. Mastering transfer function specification by using volumepro technology. In *Spring Conference on Computer Graphics* (2001), vol. 17, pp. 279–286.

[13] KRUGER, J., AND WESTERMANN, R. Acceleration techniques for gpu-based volume rendering. *Visualization* (2003), 287 – 292.

[14] LARAMEE, R. S., HAUSER, H., DOLEISCH, H., VROLIJK, B., POST, F. H., AND WEISKOPF, D. The state of the art in flow visualization: Dense and texture-based techniques. *COMPUTER GRAPHICS Forum 22*, 2 (March 2004).

[15] MORRIS, C. J., EBERT, D. S., AND RHEINGANS, P. An experimental analysis of the effectiveness of features in chernoff faces. Tech. rep., University of Maryland Baltimore County, 2000.

[16] MUNSHI, A. The opencl specification 1.1.

[17] NVIDIA. *Using Vertex Buffer Objects (VBOs)*.

[18] NVIDIA, C. Programming guide 2.0. *NVIDIA Cooperation* (2008).

[19] PERSSON, E. *Framebuffer Objects*. ATI Technologies, Inc.

[20] PHONG, B. T. Illumination for computer generated pictures. *Communications of the ACM 18 Issue 6* (June 1975).

[21] POST, F. H., VROLIJK, B., HAUSER, H., LARAMEE, R. S., AND DOLEISCH, H. Feature extraction and visualisation of flow fields. *EUROGRAPHICS* (2002).

[22] REZK-SALAMA, C., HASTREITER, P., TEITZEL, C., AND ERTL, T. Interactive exploration of volume line integral convolution based on 3d-texture mapping. *Proceedings IEEE Visualization* (1999), 233–240.

[23] ROST, R. J., LICEA-KANE, B., GINSBURG, D., KESSENICH, J. M., LICHTENBELT, B., MALAN, H., AND WEIBLEN, M. *OpenGL Shading Language*, 3 ed. Addison-Wesley Professional, 2009.

[24] RUIJTERS, D., AND VILANOVA, A. Optimizing gpu volume rendering. *Journal of WSCG 14* (2006), 9–16.

[25] SCHARSACH, H. Advanced raycasting for virtual endoscopy on consumer graphics hardware. Master's thesis, TU Wien, 2005.

[26] VANDEMEULEBROUCKE, J., SARRUT, D., AND CLARYSSE, P. The popi-model, a point-validated pixel-based breathing thorax model. *XVth International Conference on the Use of Computers in Radiation Therapy (ICCR)* (2007). obtained from: the Léon Bérard Cancer Center & CREATIS lab, Lyon, France.

[27] VENKATARAMAN, S. 4d volume rendering. In *GPU Technology Conference* (2009), NVIDIA.

[28] VILL, M. Advanced visualization techniques for non-rigid 3d 3d registration. Master's thesis, TU Munich, 2008.

List of Figures