Akademia Górniczo-Hutnicza

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA INFORMATYKI

REVIEW OF REAL-TIME VOLUME RENDERING TECHNIQUES

Author:
Paweł Szczepanek

album: 47 901

Supervisor:
Dr inż. Witold Alda

Kraków 2007

Abstract

This thesis is a review of techniques commonly used today to achieve real-time rendering of volume data. It describes volume data and volume graphics and their basic concepts and qualities along with a short historical introduction. It goes on to describe the key algorithms used to render volume graphics and their implementations, including the hardware accelerated approach using shaders and 3D textures. In closing chapters it shows practical techniques used to optimize performance and increase quality of rendering.

Keywords

Volume Rendering, Voxel, Real-time rendering, 3D texture

Table of Contents

1. Introduction

Volume rendering is a visualisation of three-dimensional data without explicit definition of surface geometry. Real-time rendering is a process which can be iterated many times a second, fast enough to allow smooth interaction with the data.

1.1. Contents of this thesis

This is a review of rendering techniques and algorithms used to visualise volumetric data which exhibit high enough performance to achieve interactive framerates[1]. All popular techniques are described and the ones currently employed and actively developed are reviewed in detail. Techniques which achieve great visual quality but are unable to do so at an interactive framerate are a complex and interesting field in themselves but will only be mentioned here as a reference as they constitute a subject for a separate thesis, being in concept very different from real-time algorithms. What is “Interactive framerate” can be defined differently according to context but arbitrarily choosing a minimum of 10 fps[2] may serve as a good rule of thumb[3]. More important than the framerate is whether the concept behind the technique aims at achieving a required level of quality or performance. This thesis will concern itself with the latter type of techniques.

---

[1] Framerate – benchmark of rendering speed expressed in number of frames rendered per second.

[2] Fps – frames per second.

[3] Paul S. Calhoun et al. gives 5-10 fps as a characteristic of real-time rendering [1].

This thesis starts with an introduction to the core concepts of voxels and with a basic context for them. It follows on with a more detailed explanation of general issues behind voxels in chapter 2 and builds on this with subsequent subchapters of chapter 3 which describe various rendering algorithms. The algorithms are explained and later their commercial application and development potential in relation to hardware acceleration evaluated. In chapters 4 and 5 techniques are shown which are used to improve – respectively to precedence – performance and quality. In the concluding chapter I try to assess the feasibility of commercial use of these concepts outside of their socialistic domains and the direction into which the development of new hardware forces them.

Attached in Appendix A is the documentation of the application used for producing some of the example images contained in this thesis.

1.2. Core concepts of the voxel

Voxel is an amalgam of words volume and pixel[4]. It denotes a single indivisible element of volumetric data. It is similar to a pixel in a 2D[5] image – extrapolated into the third dimension. Voxels, arranged uniformly in a 3D grid make up a volumetric dataset which can be used to model any three-dimensional object. You can imagine them as cubes stacked side by side and one on top of each other with no free space between them making up the volume (see figure 1) – keeping in mind it's just a visualisation of a mathematical construct.

Voxels themselves, just like pixels, don't contain explicit information about their location. It is inferred from their relative position in the grid. It is unlike other modelling methods such as polygons which are based on vertices that need their location in space to be given explicitly.

---

[4] The term pixel itself is an amalgam of words picture or pix and element.

[5] 2D – two-dimensional, analogically 3D stands for three-dimensional and 4D for three-dimensional encompassing changes in time.

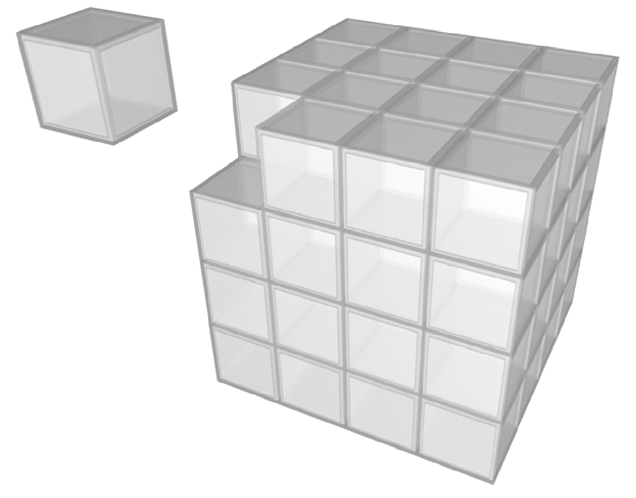


Figure 1: Visualising a set of voxels

The second most important feature in comparison with polygonal modelling (or any other surface modelling method for that matter) is the fact that volumetric datasets carry information irrespective of their position relative to the surface of the object. A cube modelled using polygons is an approximation which only stores information about the surface of the cube. A volumetric dataset can store information about the surface and interior of an object.

One more interesting comparison can be made by examining the impact of complexity of data on the two distinguishable approaches. Performance of polygon modelling suffers when subjected to complex data, organic or otherwise bereft of flat surfaces. Whereas data complexity has almost no impact on voxels which record the same amount of information irrespective of the complexity of data.

The data that the voxel actually carries is any number of scalar values which can represent any property of a modelled object sampled in the location corresponding to that voxel. This vector data in order to be rendered needs to be transformed into the visual domain using colour and opacity, usually contained in RGBA[6] data. Alternatively the native data can be used to construct surfaces which show voxels with values within a given threshold or project the maximum intensity onto a plane.

---

[6] RGBA – three channels, one for each primary colour – red, green, blue – and one channel for opacity – the alpha channel.

1.3. History and modern usage of voxels

It may be useful to start with a small digression at the beginning of this chapter concerning computer graphics in general and mention the fact that from its early days its progress has been fuelled by its potential commercial use in media and entertainment. This follows the rule that research needs funding and the business of entertainment has never lacked the money[7]. Other areas of human endeavour have benefited from it and conducted original research and development but the main commercial pull had always been coming from the media. That stated it should not be surprising that the concept of volume rendering has been born at LucasFilm[8] [1].

Ed Catamull and Alvey Ray Smith at the company's home in San Rafael created a computer graphics group which developed computer graphic techniques with volume rendering among others to be used in LucasFilm movies[9]. Later Steve Jobs bought the computer graphics division[10] of LucasFilm for 5 million dollars and founded Pixar in 1986 [5]. Pixar developed volume rendering and used it in its movies. The technology had proved to have many merits and soon research began in other fields of computer visualisation, especially in medical radiology[11] for which the technology was perfectly suited.

Computer Tomography was born in 1972 in England, invented by Godfred Hounsfield; the research has been founded by the record company EMI. At that early stage the visualisation comprised images derived from single slices. The first scanner (see figure 2) took hours to perform a scan and the image required a whole day to compute [1].

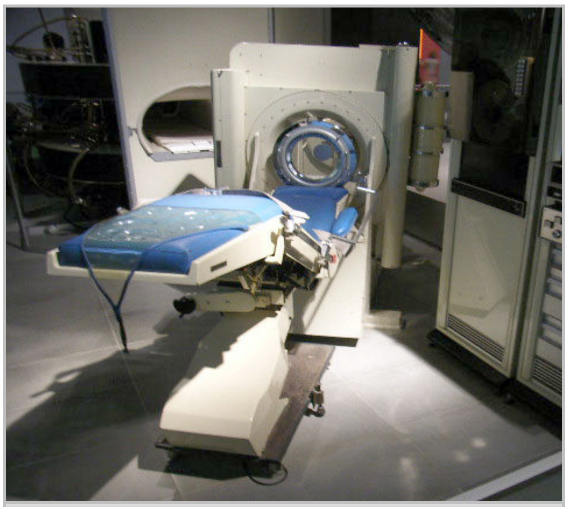


Figure 2: First commercial CT scanner

---

[7] A similar case can be made for the military and its influence on research.

[8] Film production company founded in 1971 by George Lucas.

[9] Examples can be seen in early Star Wars and Star Trek movies.

[10] He actually bought the technology from LucasFilm and gave another 5 million dollars as capital for the new company.

[11] Radiology – medical specialty dealing with radiation devices used for obtaining diagnostic data.

Volume rendering in medical applications was developed in mid 1980s [1] from previous research conducted in the preceding decade at the Mayo Clinic. It was seriously limited by its computational requirements exceeding the then available computing power[12]. Rendering was far from real-time and required hours of computations before delivering an image. It had been primarily developed for and used in medicine greatly facilitating tomography. Data acquired with the help of a scanner rotating around the subject needed complex transformation to compute a volumetric dataset of the scan. Then with the help of volume rendering techniques of that day – SSD[13] and MIP[14] [2] (discussed in detail in later chapters) – the clinician could examine thoroughly the whole dataset from any perspective and at any point.

With all-around progress in hardware and techniques volume rendering could employ more sophisticated algorithms which truly brought out the strengths of volume data. Unlike in SSD and MIP techniques volume rendering now could render all layers of data at once in a meaningful way (see figure 3).

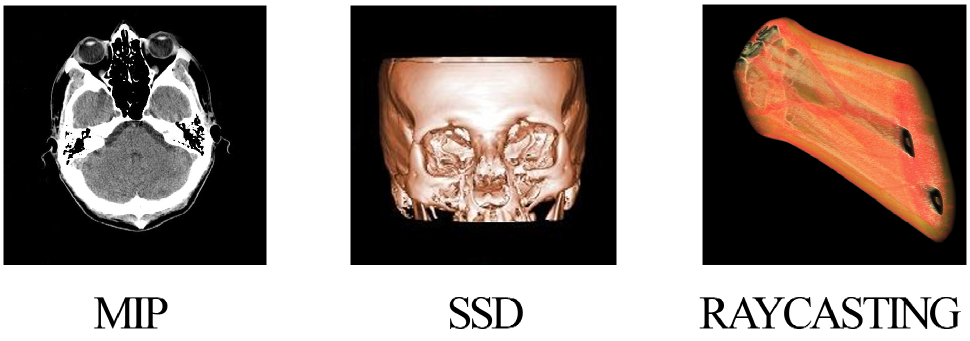

MIP SSD RAYCASTING

Figure 3: Different types of medical visualisations

Soon the technology began being implemented in other fields where the data exhibits certain characteristics suited for volume rendering – e.g. multilayer nature, inhomogeneous regions. Many fields have benefited from volume rendering, such as materials technology, fluid mechanics, non-invasive structure testing and others.

---

[12] Personal computers of the day carried a ca 8 MHz CPU and 1 MB of RAM while the supercomputers peaked at 4 GFLOPS.

[13] SSD – shaded surface display.

[14] MIP – maximum intensity projection.

Today, there are numerous, both commercial and free, packages available which are either libraries facilitating volume rendering or complete environments for acquiring, editing and rendering volume data. The technology is amply present and actively developed but can't stand up to polygon modelling in some areas. Its competitiveness is handicapped by hardware only supporting polygon representation of visual data.

In the late 90s of the last century hardware acceleration of graphics for consumer PCs have been introduced. This acceleration had been at first very restricted and specialised. Over the years the process has been standardised and the hardware became programmable; yet even now it still only supports polygonal modelling but this is about to change. In the last 10 years we've seen a staggering progress in consumer computer graphics being fuelled by the gaming industry. Due to this vast hardware acceleration polygon rendering is now the dominant, and for all practical purposes sole, rendering method in computer games and many professional applications. The GPU[15] can perform many more specialised instructions per second than the general CPU of the PC[16].

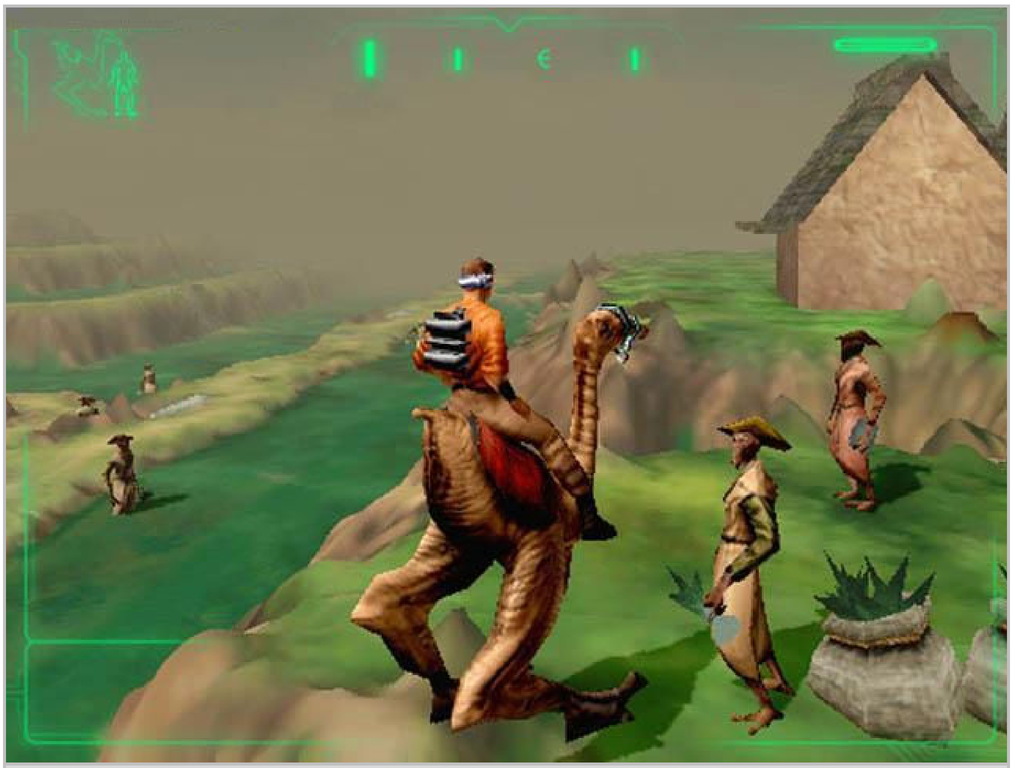


Figure 4: Voxel terrain in Outcast

---

[15] GPU - Graphics processing unit.

[16] The currently top nVidia video card GeForce 8800 GTX has the fillrate of 36800 mega-texels per second.

With such a powerful hardware supporting only polygons volume rendering has been neglected in the last decade. It has previously been an attractive alternative for developers. Most notably it has been used in many games as a technique for rendering terrain as seen in many NovaLogic games like *Comanche* and *Delta Force*. Later a more advanced approach had been used by Appeal in *Outcast* (see figure 4) which allowed for rendering of realistic terrain otherwise impossible to achieve on consumer PCs. Voxels have been used often at one point of its history by Westwood Studios in its *Command & Conquer: Tiberian Sun* (see figure 5), *Red Alert 2* and *Blade Runner* (see figure 6) which used them for rendering units and characters. Other notable titles include *Sid Meier's Alpha Centauri*, *Master of Orion 3*, *Shadow Warrior*, *Vangers* and *Perimeter*. Up until 1998 it's been a free game for both technologies. More titles were planning on using voxels but when 3D accelerators became too widespread not to use their computational power it became clear that polygon graphics was the way to go and their engines were scrapped or rewritten accordingly.

Voxel technology is however too powerful in principle to be buried completely. The new DirectX 10 and the new shaders 4.0 bring back the possibility for both approaches to coexist. It's safe to assume that there is a high probability we will see things which have happened to polygons in recent years happening to volume rendering. This can only be a good thing as it will bring more developers into voxel technologies, increasing the pace of their advance yielding benefits in all fields of their usage.

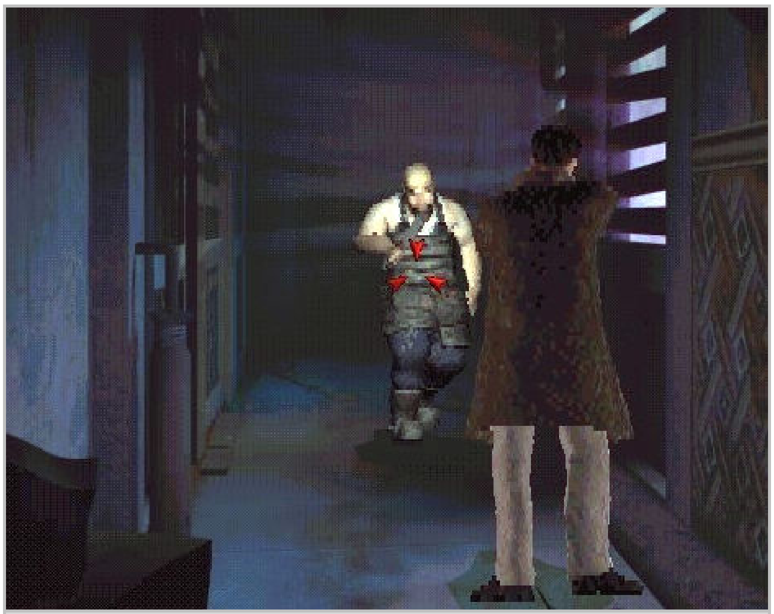


Figure 5: Voxel characters in Blade Runner

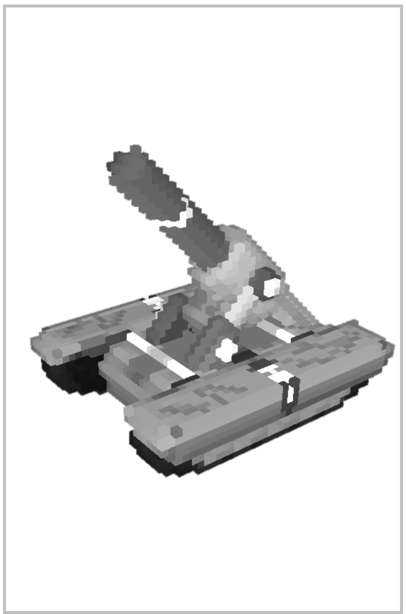


Figure 6: A voxel unit in C&C: Tiberian Sun

2. Volumetric Datasets

When modelling 3D objects we are used to representing them as mathematical models defined by sets of vertices, lines and polygons. The visual properties of any given point on the surface of the model is produced by a shading algorithm which takes into account the data acquired from the environment and vertices defining the given polygon. Over the years the data used to produce the final effect has grown enormously along with complexity of the shading algorithms. To produce a single pixel of the final image it may be necessary to take into account the environment's conditions and post-processing effects[17], the vertices' positions, normals and one or more set of texture coordinates and to load and evaluate for the interpolated colour at all locations on polygons as many textures as the programmer managed to write into the shader. It is common to use a 32-bit texture[18] to describe the diffuse colour and opacity of the surface, another 32-bit texture[19] to describe the normal vectors and the specular intensity of the texels[20] and another 24-bit texture[21] with different texture coordinates to multiply by the output colour to achieve additional lighting effects. This is a massive amount of data describing however only the characteristics of the surface of the object. There is no way to store the object's characteristics of its inside parts. For that, you need volumetric data. Given the trend and rate of computer graphics getting closer to photorealism (ATI has recently released a demo demonstrating real-time sub-surface scattering) it may soon be necessary to store such data.

---

[17] Effects to which the rasterised image can be subjected to before being rendered on the screen such as various colour shifts, distortion, blur and other 2D effects.

[18] The diffuse texture which alpha channel is used for opacity. This is arbitrary and depends on the implementation – this is given as an example of common usage. That applies to all following textures.

[19] Commonly known as the bump-map or normal map. The RGB channels are used to store the vector of the surface at each texel. Usually the alpha channel is used to map the intensity of the specular reflection to differentiate between matte and shiny areas within the surface.

[20] Texel – the pixel element of the texture mapped onto a surface of a polygon.

[21] Commonly known as the lightmap. This texture is usually mapped less densely than the diffuse texture and is used to give the effect of self shadowing of the object.

Volumetric data can store information about both the surface and inside of the object. Volumetric data due to, among other properties, its 'multi-layer' nature brings many interesting factors to take into consideration when working with. Its correlation between magnitudes of data and memory requirements are less intuitive than that of linear data. It is a common practice to develop and visualise algorithms in 2D for ease of design and then bring them into 3D. Every algorithm that needs to be adapted into the third dimension in volume rendering needs special care to work properly and retain if possible its performance. This chapter describes in detail the properties of volumetric data, ways of acquiring such data and how to deal with such vast amounts of it and translate it into visual properties.

2.1. Properties of volumetric datasets

Volume data is physically a three-dimensional matrix of scalar or vector data. The data is discretised at uniform distances; it represents the values sampled at corresponding points in the object. This process is called voxelisation [7] and creates a discretised model of an object consisting of voxels – cubic uniform units of volume tightly fit together with a corresponding numeric value. This process is not unlike the process of rasterisation.

Objects (over the atomic scale) in real space are continuous in nature and after voxelisation become a model in a quantised space. This has profound consequences which can be both beneficial and disadvantageous. One obvious shortcoming of such a model is its finite resolution. This means a certain loss in accuracy of the representation and any calculation made on it. Measurements of area and distance made on voxel approximations are usually inferior in accuracy to ones made on surface models though they are simpler. Transformations made inside voxel space[22] due to space quantisation can result in data loss[23]. This, again, is very similar to problems with 2D raster graphics. These can't be completely overcome but can be minimised by anti-aliasing the data.

---

[22] Voxel space is simply the matrix onto which the numeric values of the volume are written; it can also be used to describe the technology of volume data rendering/manipulation.

[23] Practically any transformation other then rotation by 90° will result in loss of data and/or image distortion.

These are serious problems since unlike in 2D raster graphics increasing the resolution isn't as easy. Volume data brings the memory consumption, literally, to the next exponential step. Doubling the resolution of a 512x512 32-bit image file will result in an 4 times increase in image size and thus memory usage up to about 1 MB. Doubling the resolution of a moderate volume dataset of 256x256x256@16bit[24] will increase the size 8 times up to 256 MB. This not only creates a problem of storage but more so of data transfer. Even having enough memory and computational power you will very quickly find yourself in a bottleneck situation with the transfer rates of data to and from the processing unit prohibiting real-time operations. There is no visible solution to this problem other than proposing a different hardware architecture.

Admitting these substantial problems with volume data representation it nevertheless has enough redeeming qualities to have made itself irreplaceable in certain applications. The most obvious is the native ability to record data about the inner structure of the object. Another prominent quality derives from the very same, most basic property described in the previous paragraphs. An object modelled in volume graphics is sampled at all points - disregarding its complexity. Inversely when modelling an object with polygons, you need to add more vertices at curved areas. This means that the complexity of a scene modelled with polygons depends on the complexity of the object modelled. Volumetric data is insensitive of scene complexity and its computational requirement is predictable and depends only on the volume's resolution[25].

Other, not so obvious an advantage is easy application of Boolean operations. This is useful in Constructive Solid Geometry[26] modelling [9]. When modelling solid objects with polygons special care must be taken when applying Boolean operations between meshes. Complex calculations are needed to create correct meshes which do not have holes and inverted faces[27]. Voxels on the other hand can be freely removed and added, without invalidating the object's integrity thus making Boolean operations trivial.

---

[24] A dataset of 256^3 voxels sampled at 16 bits.

[25] Not taking into account any content sensitive optimizations.

[26] Constructive Solid Geometry or CSG - technique of modelling that uses mainly Boolean operations to produce solid geometry.

[27] This involves tessellating the mesh, finding points of intersection, creating new vertices and faces. In a space of finite precision this can yield unexpected results which can produce an illegal mesh. This problem has eternally plagued 3D editing software and has as yet found no foolproof solution.

2.2. Acquiring volumetric data

Volumetric data can in theory be modelled the same way surface models are made – with the use of dedicated software designed for artists to use. In practice however there's very little need for voxel modelling software as most of volume data is acquired by scanning real objects or is generated procedurally by a programmer. In special cases when there is a need for an artist to create a model it is usually created in a polygon modelling software and then voxelised. There are few applications designed specifically for voxel modelling and they are usually either simple freeware programs with limited functionality or in-house[28] applications never released commercially.

2.2.1. Scanning

The primary source for volume data is 3D scanning done in various ways. Volume data is in most cases a direct result of scanning and needs no or little conversion; it is the native result of scanning[29]. The term scanning is used here in a very broad sense denoting any technique that allows for automatic conversion of real objects into digital data. This includes medical x-ray scanners, PET[30] scanners, DOT[31] scanners, acoustic scanners, optical scanners and even multiple photographs [10].

In some cases the data collected in raw form needs to be computed to result in volume data but this computation however complicated numerically is not analytical in nature and follows simple mathematical transformations. Such is the case with medical scans which use a scanner that rotates along an axis; the data needs to be transformed from 2D slices aligned along the axis into a 3D grid with Cartesian coordinates.

---

[28] Application made by a developer for its own internal use, usually for a single project; its development is rarely continued and it's usually crude and not very user friendly as it is made to complete a certain task and then be discarded.

[29] Conversely creating polygon or NURBS models from a 3d scanner requires complex reverse engineering of data.

[30] PET – Positron emission tomography.

[31] DOT – diffuse optical tomography.

As any physical device scanners are limited by the technology and working conditions. Data acquired with their use is susceptible to technical problems, most notably, noise. The characteristic of a scan which describes the quality of signal acquisition is called SNR[32]. It is a measure of how much the data is corrupted by random values the source of which is not the object being scanned but either the scanning apparatus itself or the environment. In some cases this can be, at least partially, overcome by multiple scans[33].

Nevertheless scanned data can suffer from many problems. The scanned object may loose fine detail due to insufficient resolution of the scan. Also despite multiple scans there may still be visible noise which can later cause problems when rendering and manipulating data (see figure 7). Different techniques are better or worse suited for presenting noisy data. Iso-surface representations suffer the most from such data as it creates needless small 'particles' increasing the polycount[34] and occluding the view. MIP rendering which is inherently tolerant of this by its high contrast results gives the clearest (though arguably not the most informative) results and raycasting along with rendering via proxy geometry lays in the middle-ground.

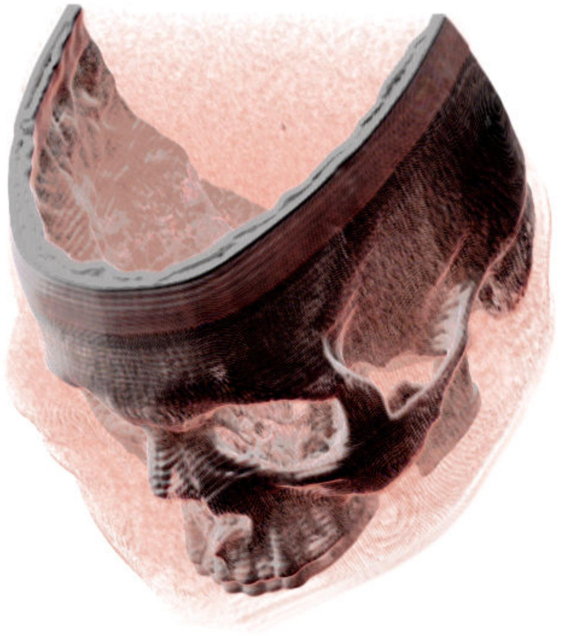


Figure 7: Skull rendered with proxy geometry with visible noise

---

[32] SNR – Signal-to-noise ratio.

[33] Since the signal remains the same and the noise values are random, taking the mean out of multiple scans will decrease the SNR.

[34] Polycount - number on polygons on the scene; a measure of scene complexity; a portmanteau of "polygon count".

2.2.2. Procedural generation

For some volume models there is no need to remember the exact data. This usually applies to non-scientific data such as for models used in movies and games. This may include fog, dust, fire, and various gaseous phenomena and even terrain. In such cases, the data can be either pre-computed with simple mathematical recipes or more often computed on-the-fly to save on storage space and memory access time. This usually involves the computed set being mapped onto a simple voxel model of usually lower resolution which defines the boundaries of the model. This way one can achieve a high resolution model without its high memory requirement for the cost o computing it on-the-fly.

Otherwise, procedural generation can be used to create volume sets which are either fictional, or would be otherwise difficult to scan but have a known algorithm they follow – like for example earth's layers, or trees' foliage. This is an interesting branch in itself, however the on-the-fly procedural generation will be more relevant to the subject of real-time rendering since only then it has an impact on performance.

2.2.3. Conversion from polygons

Like procedural generation, conversion from polygons is only important for non-scientific data. When it can't be scanned or generated procedurally there is no other way than to create it manually. Since there are virtually no professional voxel editors the best way to get around it is to model the object in conventional modelling software[35] and convert the meshes into voxels.

The mesh is virtually scanned like a real-life object and undergoes the same process of voxelisation. The only difference is the exporting program must provide algorithms for generating the inside of the shell. There's no commercial software for this, but the process is straightforward enough and such an exporter is usually programmed as an in-house tool for a specific task[36]. Until voxels regain popularity there is little hope of seeing commercial products including such a feature.

---

[35] Such as Maya, 3DSMAX, XSI.

[36] There are however some simple some freeware exporters that can be found distributed freely on the Internet.

2.3. Classification - the transfer function

Volume data can be used to describe any type of data. A dataset can hold many types of information. In order to access this information visually you need to devise a way of rendering it. Classification is the process of converting arbitrary scalar data into visual properties. The simplest way would be to treat the data as a 3D texture and its values as RGBA values. The data however is usually incompatible for such a simple conversion. The next step would be to change the values physically in the dataset to normalize them to RGBA value range. This may work but will probably produce poor visual results. Instead of changing the dataset itself it is far more useful to provide a way to interpret the data for rendering. This way you don't need to change the dataset every time you change the way you convert data values into RGBA values and you won't loose information due to limited float precision when normalising to the RGBA range. Such interpretation is done by means of a transfer function.

The transfer function uses the data explicitly stored in the dataset and derived from it to find a RGBA value for the voxel. A common approach is to use lookup tables. This way, a scalar value can be used as the address of the optical value in the lookup table[37] [11]. In simple cases it can be a 1D lookup table, but this gives little flexibility in visualising the data. Theoretically you can use as many dimensions as you want by means of using multiple lookup tables in conjunction[38], the only limit being performance.

Data classification can be viewed as a form of pattern recognition [6]. Its goal is to provide means of improving interpretation of the data visually by bringing important features of the data into view. The process can become very complex depending on the complexity of the data. This is why the transfer function creation is usually an interactive process helping the user find the most convenient way for him of viewing the data.

---

[37] An n-dimensional table which is then addressed by n values to read the result. The table is usually stored as a texture to take advantage of the fast video memory.

[38] You can use a 2D lookup table which takes two data values from the volumetric dataset and provides the value for another 2D lookup table to use with conjunction with a third value from the dataset itself.

3. Volume rendering

"Volume visualization is a method of extracting meaningful information from volumetric datasets through the use of interactive graphics and imaging, and is concerned with the representation, manipulation, and rendering of volumetric datasets" [8]

Kaufman, A.; Volume Visualization; IEEE Computer Society Press Tutorial; 1990

One subdivision of volume rendering, or be it not very useful, is into direct and indirect rendering. Indirect volume rendering sometimes isn't even considered volume rendering at all [1] as it only extracts data from the volume in order to render it by means of polygons as in the iso-surface technique. Direct volume rendering on the other hand renders the volume by means of classification which with the help of the transfer function which translates data values into optical parameters[39] (these are then all integrated by means of one of many algorithms into a flat output image). These categories aren't any good any longer as there are now techniques which can't be unambiguously assigned to either. Some use many iso-surfaces to integrate data from inside the volume by alpha-blending[40] them, some render iso-surfaces without reverting to polygons, some use polygons and textures or 3D textures to add hardware acceleration to what can be considered, in terms of integration, raycasting. The fault lies within the fact that this categorisation mixes implementation issues with abstract ideas.

A more useful approach might be categorising rendering techniques by optical models abstracted from implementation [12]. Each subsequent model described below is more computationally demanding but produces better results in terms of physical accuracy.

---

[39] This usually means RGBA values.

[40] Alpha-blend – to composite multiple layers by weighing the colours by the corresponding alpha values

The simplest models are absorption only and emission only models. The first one works on the assumption that the volume consists of matte objects which have a negligible emission of light; the voxels only occlude light and don't reflect incidental light or shine with light themselves[41]. The latter model takes the opposite approach; in it the objects are completely transparent; they let light through without interacting with it in any way. This works well for modelling hot gasses in which case the occlusion of incidental light is negligible to the light emitted by the object[42]. Both models produce pretty unrealistic but still usable renderings. The usability of these models can be increased greatly by shading the result to provide depth cues and show surface curvature.

Absorption only and emission only models are only good approximation in extreme cases of objects. A more sophisticated model of emission and absorption combined provides an approximation for more diverse range of objects which both emit and occlude light. This proves to be a sufficient approximation for most implementations of volume rendering. Its relative computational simplicity makes it by far the most popular model for real-time implementations. It lacks sophisticated realism such as light scattering and shadowing but with the help of a number of tricks it can achieve very similar, or at least acceptable, results.

The next step to achieving realism of rendering is the single scattering of external illumination model. This is a far more advanced model and would normally entail a lengthy explanation of it; however, as this model is already overkill for real-time rendering a detailed description of this and following models isn't necessary for the purposes of this thesis. For completeness of this categorization, the single scattering model takes into account the light from the scene scattered into the volume; inside the volume light travels unimpeded. Further models improve on this by adding shadowing and finally multiple scattering of light inside the volume. The complexity of this model makes it near impossible to implement without imposing certain limitations as theoretically each voxel contributes to the lighting of all other voxels.

---

[41] Smoke may be approximated quite well with this model

[42] Such as for example fire.

3.1. Raycasting

The most naive way of rendering is to cast a ray for each pixel of the output image into the scene and determine the resulting colour by finding all the objects the ray hits along the way. Surprisingly, when applying this approach to volumes this technique can be used to achieve real-time framerates.

3.1.1. The integral of raycasting in volumes

Raycasting rendering integral describes the mathematical operations needed to evaluate each pixel of the output image; while all rendering techniques use a similar evaluation method, raycasting is its most direct implementation.

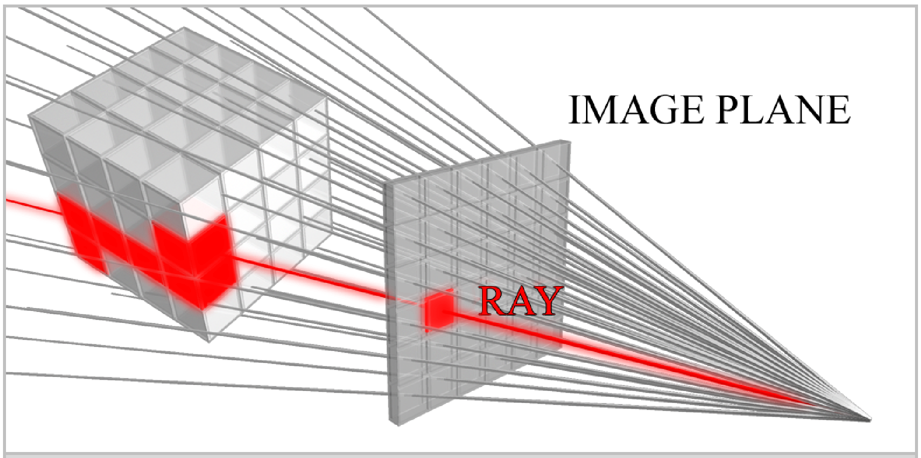


Figure 8: Raycasting

In raycasting for each pixel of the rendering image plane a line (or ray) is virtually drawn defined by two points – the pixel it cuts through and a point where all the lines meet which can be thought of as the eye, or the observer[43]. This can be thought of as a ray of light coming through the volume and hitting the eye (see figure 8). To determine the amount and colour of the light we need to evaluate the optical properties of the volume that contribute or obstruct light along this ray.

---

[43] Unless the projection is orthogonal in which case all the lines are parallel to each other and perpendicular to the image plane.

We can define the ray x, as a function x(t), where t is the distance from the image plane. Along the ray the volume absorbs and emits light. The amount of light emitted and absorbed can be described as a functions parameterised by the same distance – c(t) and κ(t) respectively. When using the absorption and emission only model this is all the information needed to evaluate the output image. The value of the pixel is the result of the integral of the emission and absorption along the ray.

$$C = \int_{o}^{\infty} c(t) \cdot e^{-\int_{0}^{d} \kappa(t) dt} dt \qquad [6]$$

Even though the absorption and emission is a continuous function in practice this integral is solved by a Riemann sum on discrete data sampled from the volume at uniformly distributed locations by trilinear interpolation[44]; Δt is the distance between resampling points. The resampled values are converted into RGBA values by the transfer function.

$$\int_{0}^{d} \kappa(t) dt \approx \sum_{i=0}^{t/\Delta t} \kappa(i \cdot \Delta t) \Delta t$$

We can substitute the Riemann sum in the exponent by a multiplication of the whole argument.

$$e^{-\sum_{i=0}^{t/\Delta t} \kappa(i \cdot \Delta t) \Delta t} = \prod_{i=0}^{t/\Delta t} e^{-\kappa(i \cdot \Delta t) \Delta t}$$

In order to make use of the RGBA values we have obtained from classification of the data we need to introduce the opacity A into the equation.

$$A_i = 1 - e^{-\kappa(i \cdot \Delta t) \Delta t}$$

---

[44] It takes the values of 8 closest voxels and calculates the resampled value by weighing the values according to their distances from the sampling point.

With this we can further transform the equation.

$$\prod_{i=0}^{t/\Delta t} e^{-\kappa(i\cdot\Delta t)\Delta t} = \prod_{i=0}^{t/d}(1-A_i)$$

This final result can be used for numerically calculating the approximation of absorption along the ray.

The emission function can be approximated likewise.

$$C_i = c(i\cdot\Delta t)\Delta t$$

Combining the approximation we can produce a usable approximation of the volume rendering integral.

$$C = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1}(1-A_j)$$

The resulting equation can be iteratively evaluated by alpha blending by stepping in a back-to-front[45] order from n-1 to 0, where n is the number of samples.

$$C_i' = C_i + (1-A_i)C_{i+1}'$$

The resulting colour C′ is calculated by compositing it against the last colour[46] using the alpha value from the current location. In practice opacity-weighted colours[47] are used to avoid artifacts[48] that would otherwise arise in the interpolation stage.

---

[45] Voxels farthest along the ray are evaluated first.

[46] If it's the first sample then the background colour is used.

[47] Colours pre-multiplied by their corresponding opacity values.

[48] Artifacts are flaws in rendering introduced by the process itself.

Alternatively the equation can be evaluated in a front-to-back order. This proves very useful in software implementations since it can benefit from an early ray termination[49] optimisation. Hardware implementations that don't use actual rays tend to use back-to-front compositing since then you needn't track the alpha values.

3.1.2. MIP

A special case of raycasting is a simplification called MIP[50] rendering. It is most commonly used in radiology as the data there is very often noisy to a degree that it's difficult to create any reasonable iso-surfaces or even perform normal raycasting. MIP results in much less informative but clearer renderings. The results are simply silhouettes of dense areas with all the depth data lost which can be advantageous in some cases. MIP rendering instead of integrating all voxels along the ray simply stores the highest value found and writes it onto the output image.

Given the nature of medical datasets the results prove usable. The data sets are usually narrow slices rather than full scans of body parts so as to be more suitable for such rendering. This type of rendering is virtually limited to medical radiology.

3.2. The shear-warp algorithm

The shear-warp algorithm proved to be a very effective technique for software volume rendering[51] which principles are even used in some hardware implementations. It is partly analogous to raycasting in the rendering integral but in opposition to raycasting it doesn't cast actual rays. Instead the volume itself is projected onto one of its sides [13]. This allows for very vast evaluation and easy bilinear filtering[52].

---

[49] The ray is evaluated until the output value of opacity reaches a certain level.

[50] MIP – maximum intensity projection

[51] Unfortunately it's been patented in 1995 by Pixar.

[52] The resampling points are always on the same slice.

The shear-warp algorithm relies on the natural storage structure of the volume to simplify and speedup rendering. It behaves as if the rendering had been done by raycasting with the image plane parallel to one of the sides of the volume and with no perspective distortion. In order to allow a full freedom of the camera, along with the illusion of perspective, the shear-warp technique manipulates the data itself while the virtual rays all remain perpendicular to the base plane[53] (see figure 9).

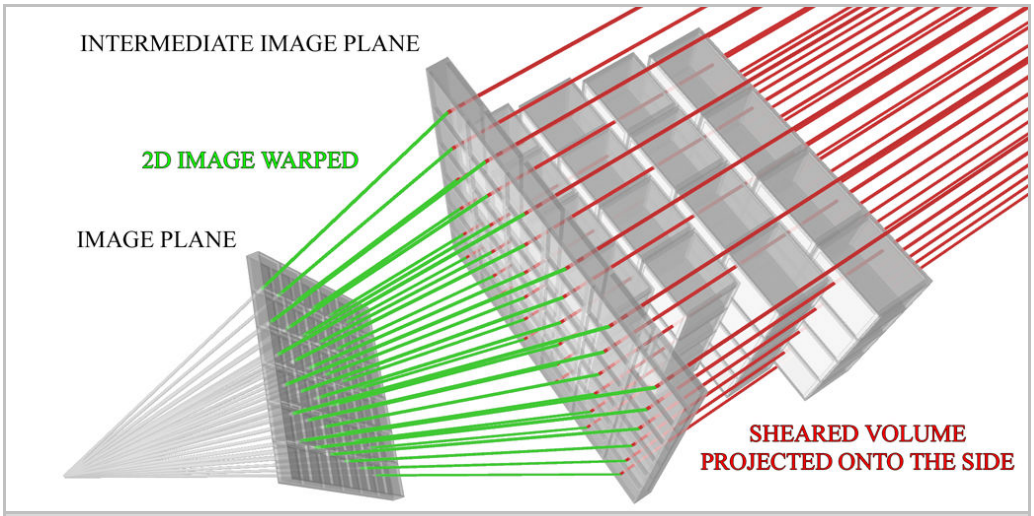


Figure 9: Shear-warp

The volume is sheared parallel to the side onto which the image is to be projected. The slices that make up the volume are then shifted so as to create the same direction of viewing through the volume as normal raycasting would achieve. If perspective is also to be taken into account the slices are then scaled accordingly. Such manipulated volume is then projected onto the base plane.

The intermediate image achieved immediately after the projection is then warped by a simple 2D transformation to fit the actual image plane. This transformation either looses some information calculated previously especially if the image plane was at an unfortunate angle – close to 45°, or conversely the missing information needs to be interpolated (depending on the implementation).

---

[53] The intermediate image plane that covers one of the sides of the volume and is later warped to fit the actual image plane.

If the volume is stored in a set of 2D textures this technique requires three sets – one for each possible orientation. This, apart from requiring more memory allocated, creates a problem when switching the image plane to the one closest to the viewing direction; it may cause a visible "jump" caused by the interpolation of the 2D warp.

3.3. Hardware raycasting

When wanting to move the calculations to the GPU[54] on the graphics card it may seam natural to convert the volume into polygons and take advantage of all the built in capabilities and heavily supported functions for polygon rendering of the 3D accelerators. The technique is very fast due to hardware support but lacks the possibility of benefiting from many optimisations that raycasting offers. It is however possible to retain all the advantages of raycasting while still taking advantage of the GPU's computing power.

All the principles of software raycasting remain unchanged, along with the front-to-back traversal of the ray; this is a "bringing what's best from the two worlds" approach. The technique is however very implementation specific. But as the graphics hardware tends to retain the set path of development – i.e. more and more parallelism and programmability – the technique will probably not only remain viable but actually prove more and more competitive to others. Raycasting is in its very nature parallel and lends itself perfectly to the GPU pipeline, always taking full advantage of the GPU parallel capabilities.

With the current state of GPUs there are a few problems which make this approach difficult. The GPUs lack certain functions such as loops along with conditions to exit them prematurely. Thus, parts of the ray traversal still require CPUs participation. This can soon be solved by the advancement of the hardware itself.

---

[54] GPU – graphics processing unit

3.3.1. GPU pipeline

The graphics pipeline defines the display traversal process. It is a set of fixed stages performed iteratively in order to render objects on the screen. The process takes as input a stream of data describing the scene, such as vertices, surfaces, textures, the camera, lights, etc., and outputs the pixel colours ready for display [16]. Over the years the pipeline has become more and more programmable allowing for complex arbitrary calculations to be performed on the GPU.

The pipeline is becoming more flexible but the key features remain – the pipeline to be effective uses multiple streams and the results of each stage of the pipeline are forwarded on to the next. For a convenient overview the pipeline can be divided into three tiers [11]. The geometry processing tier prepares the vertices in the scene[55] by setting them up by applying transformation matrices to them and joining them into geometric primitives[56] (see figure 10). In the rasterisation tier the primitives are decomposed into fragments[57] (see figure 12) and mapped by textures[58]. Lastly in the fragment operations tier additional shader programs[59] can be used to manipulate the pixel's colour which eventually is either output for display or discarded.

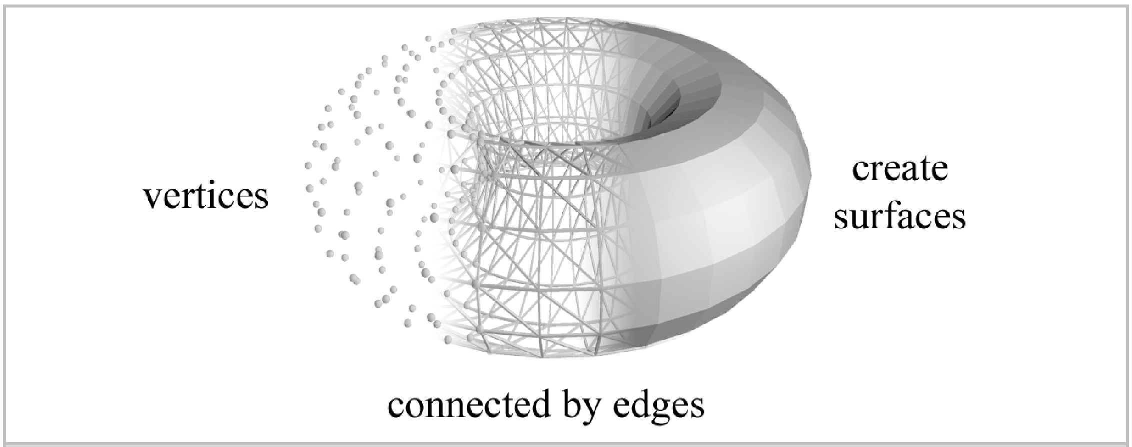


Figure 10: First tier of the GPU pipeline

---

[55] The operations at this tier are applied to vertices only.

[56] Geometric primitive - an atomic graphical object that can be drawn on screen by scan conversion, usually a triangle or a quad.

[57] Elements of the picture which correspond to each pixel of the rendered image.

[58] Operations at this tier are applied to fragments.

[59] Small programs written for the GPU pipeline performed for each vertex or each fragment.

More accurately the rendering pipeline can be divided into seven stages [16]. The first stage is creating primitives and applying modelling transformations to them. All the objects are described within their local coordinate system. Applying the transformation matrices we can translate the coordinates into a common world coordinate system. This way the primitives' positions can be compared in a meaningful way.

In a similar fashion during the next stage the objects are transformed by multiplying them by the view transformation matrix. This specifies their positions within the camera's coordinate system – accounting for the camera's position inside the scene, its direction and the resulting perspective distortion.

The next stage is the lighting calculations. The primitives are shaded by calculating the light values on each vertex and taking into account the normal value of the surface in relation to the light sources on the scene. This is done by a shading model which can be written into a vertex shader to achieve custom results.

In the following stage the clipping is performed. The parts of the objects that are outside the viewing volume are removed (see figure 11). This ensures that the fragments which won't be displayed on the screen won't be processed. This not only pertains to objects that are out of the screen boundaries but also the parts that are either to near to the camera or too far from it. The distance between the two clipping planes can't be infinite. It should be adjusted according to the size of the scene as the depth buffer has a limited precision.

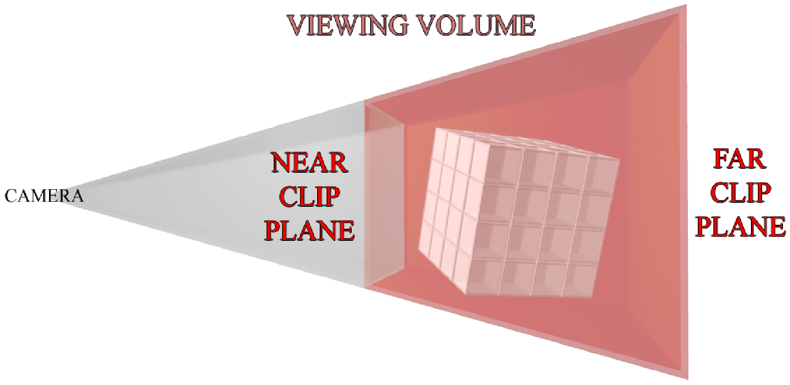


Figure 11: The viewing volume

In the final stage of the geometry tier, the objects are projected flat onto the viewing plane. Further operations are done by using screen coordinates.

During the scan conversion stage the polygons are divided into fragments; each fragment corresponds to a pixel on the display (see figure 12). During rasterisation the areas of polygons are filled with pixels whose colours are interpolated from the data stored in the vertices. This includes colour, light and texture coordinates. The appropriate texture is fetched and by interpolating the coordinates it's sampled and combined with the colour of the fragment. This can be done in a shader program which can additionally manipulate the colour of the fragment in a customised way. The final product is the RGBA quadruplet.

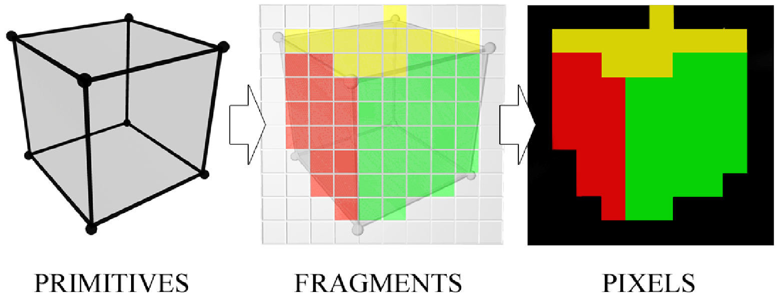


Figure 12: Rasterisation

The final stage is the visibility evaluation. All the fragments are written into the frame buffer[60]. Every time a fragment is written into the frame buffer several calculations are performed to composite it with the previous value found the frame buffer. First the new fragment's alpha value is compared with a set value and discarded accordingly. Next the stencil buffer[61] is looked up and if the fragment's corresponding pixel is set than this fragment is discarded. Analogously the depth buffer[62] is checked to see if the fragment is occluded by the previous value. Finally the fragment is blended with the current value stored in the frame buffer by weighing their respective alpha values.

---

[60] The part of video card RAM where the image to be displayed is stored.

[61] An additional buffer of the same resolution as the frame buffer carrying 1-bit values.

[62] An additional buffer of the same resolution as the frame buffer containing the depth information of the pixels in the frame buffer.

3.3.2. Programming shaders

In order to take full advantage of the features offered by today's graphics hardware and achieve non-standard rendering one needs to customise the processing of the vertices and fragments with the use of shaders. With these specialised programs run on the GPU almost any effect can be achieved. Shaders fall into two categories: vertex shaders and fragment (or pixel) shaders. The former manipulate the vertices of the objects and the latter the colour of the pixel. Recently the geometry shader has also been introduced which can create new vertices and surfaces. The shaders can be programmed in a special assembler like language with architecture specific instructions; high level languages exist to facilitate the programming[63].

The vertex shader can manipulate the attributes of the incoming vertex (see figure 13). Its location, normals and light can be changed according to the program. It can't add or removes vertices and can only operate on one vertex at a time. The vertex shader can access temporary registers which keep intermediate results to perform multiple instructions.

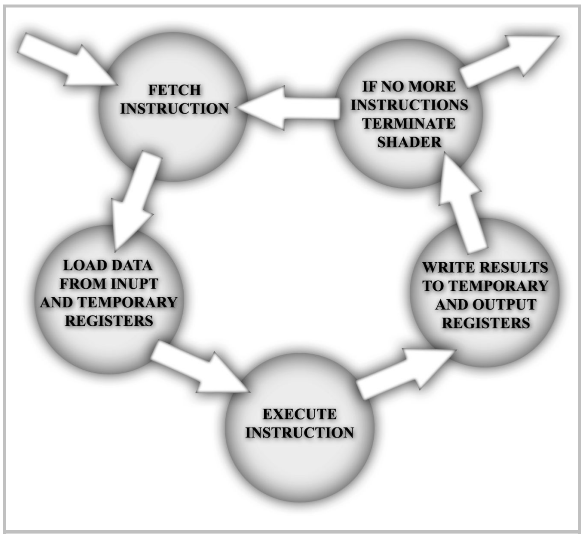

FETCH INSTRUCTION
IF NO MORE INSTRUCTIONS TERMINATE SHADER
LOAD DATA FROM INUPT AND TEMPORARY REGISTERS
WRITE RESULTS TO TEMPORARY AND OUTPUT REGISTERS
EXECUTE INSTRUCTION

Figure 13: Vertex shader loop

---

[63] Languages like Cg provided by nVidia, HLSL provided for DirectX by Microsoft or GLSL bundled with OpenGL

Fragment shaders work in a similar fashion (see figure 14). For each fragment the program is executed once. The fragment program can read all the data associated with the fragment such as texture coordinates, colour, normals and stores the final result in the output registers. The fragment program may also need to access the texels from the textures mapped onto the fragment. For this the appropriate memory address must be computed, the appropriate texels are fetched and filtered to get the colour for the sampled location.

Both vertex and fragment shaders allow for arbitrary computations to be performed on the data which can produce custom effects. Shaders are used to achieve an infinite number of effects, limited more by the programmer's imagination and less and less by hardware's architecture.

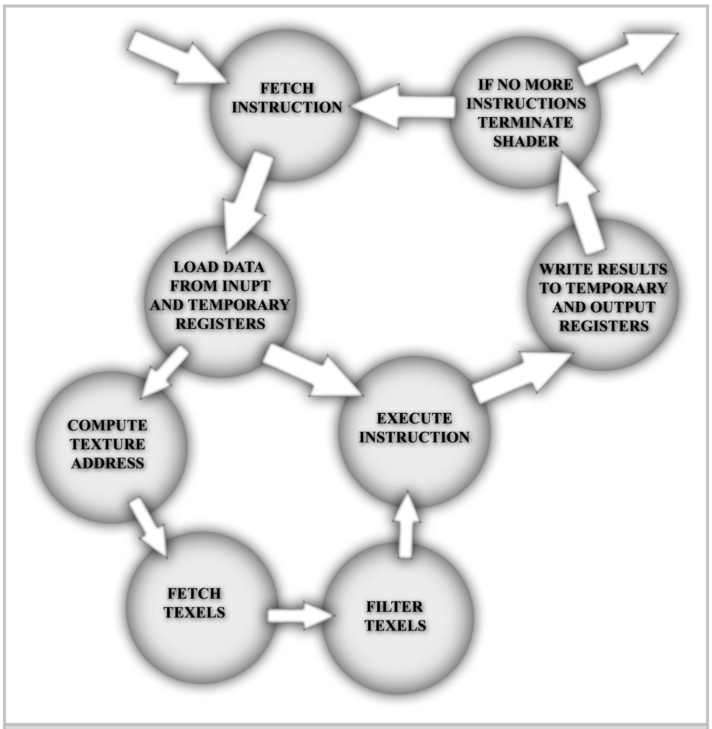

FETCH INSTRUCTION
IF NO MORE INSTRUCTIONS TERMINATE SHADER
LOAD DATA FROM INUPT AND TEMPORARY REGISTERS
WRITE RESULTS TO TEMPORARY AND OUTPUT REGISTERS
COMPUTE TEXTURE ADDRESS
EXECUTE INSTRUCTION
FETCH TEXELS
FILTER TEXELS

Figure 14: Fragment shader loop

3.3.3. Using shaders for raycasting

A volume of data in order to take advantage of the capabilities of the graphics card is stored as a 3D texture. Unlike however other hardware rendering techniques which don't use explicit rays, the data in the texture doesn't need to represent the optical values. Instead the texture can be used to store raw vector data. A texture is used for storage only to take advantage of the extremely fast bus used to access the video memory of the card[64]. The raw data can be translated into optical values on-the-fly by using the RGBA values as coordinates for the look-up tables (stored as textures as well) of the transfer function.

Additional textures are used to store intermediate results of raycasting – where each ray corresponds to a texel on the texture and eventually a pixel on the final image plane. Supplementary textures can be used to store the progress of the ray traversal for each ray and a pre-computed direction for the rays[65]. Depending on implementation additional textures can be used to store other intermediate parameters; alternatively this information can be computed on-the-fly.

Neither OpenGL nor DirectX supports simultaneous reading and writing to texture. To circumvent this two textures are used for each intermediate set of values. In every step of the traversal one is used to read from and the second to write to; after each step the roles are reversed[66].

After setting up the ray the rendering is done in many passes. Each time the ray traversal is moved forward, the resulting colour evaluated and the process checked for termination. The initiation of each pass must be done by the CPU, which manages skipping the terminated rays. The ray traversal and evaluation is done by fragment shader[67] programs. Another shader launched after every step terminates the ray if required.

---

[64] The 384-bit bus on the nVidia GeForce 8800 has a bandwidth of 86 GB/s.

[65] It is computed by normalizing the difference vector between the entry points and camera.

[66] This is called a ping-pong approach.

[67] Fragment shader or pixel shader depending on implementation (OpenGL or DirectX respectively).

Such an algorithm has been presented by Krüger and Westermann [14] who stress out the importance of devising such an implementation which maximises the usage of parallel capabilities of the GPU while minimizing the number of computations per ray. Their algorithm uses multiple-pass[68] rendering with early ray termination for rays which accumulated enough opacity or left the volume.

In the first pass the front faces of the volume are rendered to a 2D texture with the colours corresponding to the coordinates in 3D space. This is done by attaching extremes of coordinates to the vertices of the volume bounding box[69] and interpolating the area in-between. The texture is called TMP and has the resolution of the image plane; it stores the first intersection points of the rays with the volume bounding box.

In the second pass further pre-computations take place. A similar procedure as in the first pass is performed to find the direction of the rays. The back faces of the bounding box are rendered analogously. The values from the TMP texture are now subtracted from them and the normalized result is written to the colour component of a texture called DIR, while the length is written to its alpha channel. The RGB values of the DIR texture store the directions of the rays and the A (alpha channel) values store the lengths of the rays.

Now begins the actual ray traversal. This pass is repeated indefinitely until the loop is broken by a separate fragment program. The result is rendered to another texture called RES. The textures DIR and RES are mapped onto the screen with the same pair of coordinates. At each step the previous colour is blended with the new sample and written to the RES texture. At each step the sample’s coordinates in the 3d texture containing the volume data are found by moving the current position of the ray read from the POS texture along the direction read from the DIR texture by a given amount[70]. If the ray moves out of the volume the opacity in the RES texture is set to the maximum value. This is done by comparing the alpha values of POS and DIR textures.

---

[68] Rather than evaluating each ray completely, the ray is evaluated step by step, using the intermediate results as basis for each next step.

[69] The smallest possible box containing the volume

[70] The sampling distance can either by constant or for example read from a 3D texture used to vary the sampling rate.

After each pass a fragment program is launched to determine if the ray has left the volume or accumulated enough opacity to be terminated early. This is done by rendering the frontfaces and checking the opacity values. If the alpha value is higher than a given threshold than this ray is terminated by writing the maximum value into the z-buffer[71].

To finish rendering the volume we must know how many render passes there must be performed. This is done by computing the worst case scenario every time. The overhead however is minimised by the early z-test which skips the lengthy shader programs for terminated rays.

An alternative way of terminating the rendering process after terminating all rays is proposed by Stefan Röttger et al. [15] who propose an asynchronous occlusion query which returns the number of fragments that passed the z-test; when none have passed this means that no new samples were integrated and thus the result may be rendered and the loop terminated.

3.4. Rendering via proxy geometry

Instead of devising our own schemes for integrating rays using hardware acceleration we can use the existing scheme and let the hardware do all the rendering work. Today's consumer graphics hardware is specialised in rendering polygons; if we find a way of converting the volume data into polygons we can take advantage of the specialised but very efficient pipeline. While this approach still requires special care when rendering volume data there's no need to perform any form of ray traversal since this is substituted by display traversal and is done by the hardware.

The difficult part is efficiently converting the volume data into polygonal objects. This is done by creating proxy geometry for the pipeline to render data sampled from the volume on (see figure 15). The actual rendering is done by rendering these polygons and alpha-blending them in a back-to-front order. There are many schemes for creating such geometry and sampling the data which is then mapped onto these polygons as textures.

---

[71] It's a 2D texture that stores the fragment's distance from the camera. In this case the z-buffer will prevent the pixel shader from evaluating the pixels where the maximum value has been written – this method is called z-culling.

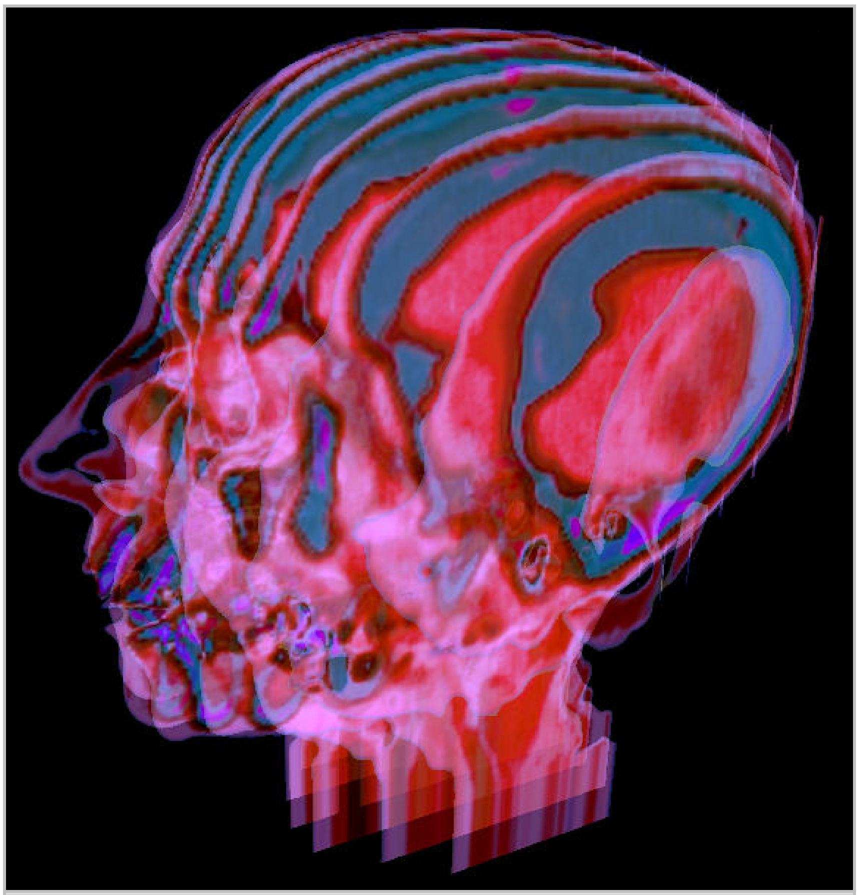


Figure 15: Planes mapped with a texture representing volume data

With this approach we loose some optimisations only available for raycasting algorithms but have the benefit of full hardware acceleration. In general this method can boast highest framerates among the techniques presented so far. This is counterbalanced by some global effects that need to be approximated since they can't be explicitly calculated.

3.4.1. Proxy geometry

The graphics hardware needs primitives to render anything. Even when implementing raycasting, to display the result it needs to be rendered on a polygon and the results can't physically leave its surface[72]. When compositing the volume via display traversal, several polygons stacked close to each other, called slices, are used to display the volume data.

---

[72] Even though the image seams to have depth, in actuality is a flat texture mapped onto a polygon.

Proxy geometry is a set of polygonal primitives used to represent the data in the volume [6]. They can be in the shape of planar quads, triangles or sphere fragments depending on the technique and implementation. After creating the geometry the data must be sampled and textures created and mapped onto this geometry. The data itself can be either kept in a 3D texture or in a stack of 2D textures. This depends on the orientation of the slices and the memory availability.

3.4.2. Object aligned slices

Object aligned slices are quads created at uniform distances from each other, parallel to one of the sides of the volume (see figure 16). Three stacks of slices are created – one for each possible viewing direction. The number of slices is usually equal to the number of voxels along one edge of the volume. The quads are mapped with a texture that can either by stored in a single 3D texture or a stack of 2D textures. Using the 3D texture we can address it with three coordinates and use trilinear interpolation. Since caches for textures are optimised for 2D textures it may be better to use them instead. Then the third coordinate becomes the index of the texture in the stack. The texture is then sampled with simpler and hardware-native bilinear interpolation.

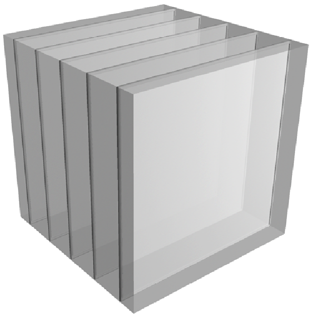


Figure 16: Object aligned slices

Using a stack of 2D textures creates a problem when the slices are not aligned with the stack of textures. If the viewing stack is perpendicular to the texture stack then each slice would require sampling texels from several textures. That's why along with three stacks of quads, three stacks of 2D textures are stored – each parallel to one of the sides of the volume. This however means that the volume is stored three times in memory which can be a serious disadvantage with large datasets. Additionally when the volume is rotated and the slices are switched from one stack to another which is more perpendicular to the new viewing direction there are visible artifacts when the sampling points change (see figure 17).

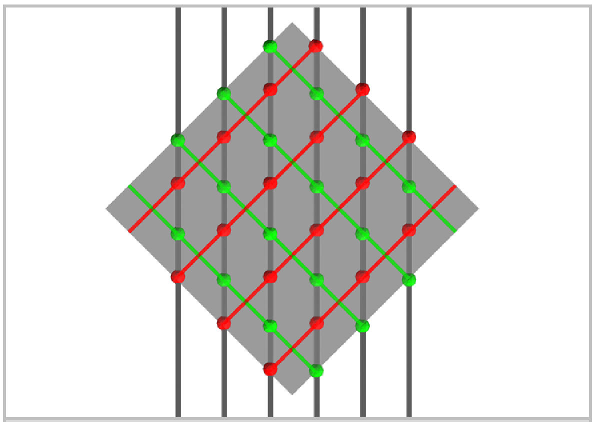


Figure 17: Switching stacks of object aligned slices causes a change of sampling points

Having created the slices, and chosen the appropriate stack for the viewing direction the stack is rendered from back to front. The slices are composited by alpha-blending them. Since the slices are at different angles to the viewing direction depending on the volume's rotation the actual distance between sampling points varies. The worst case is when the viewing direction is close to the point of changing stacks.

The changing distance between sampling points of the slices causes additional problems. Since the number of samples changes the opacity values must be distributed differently. Every time the viewing angle changes the transfer function is updated. The new opacity is calculated by multiplying the original values by the reciprocal of the cosine of the angle between the viewing direction and the stack direction.

If colour values are stored as opacity weighted colours they need to be calculated analogously.

Despite having large memory requirements this method proves very efficient; however the quality of the rendering suffers greatly from undersampling and stack switching. When using stacks of 2D texture the biggest problem with quality is the fixed number of slices which equals the number of textures in a stack. Adding more 2D textures could require too much additional memory. Since there's no filtering between textures in the stack we can't simply add new slices without creating textures for them. This can be circumvented by multitexturing[73]. By blending two 2D textures we can achieve the same effect as trilinear filtering. All this can be done on-the-fly by a fragment shader program. This way we can create arbitrary number of slices depending on the viewing conditions.

One thing the technique lacks is a way of compensating for the changes in sampling due to perspective projection. Besides opacity errors the off-centre areas suffer from undersampling.

3.4.3. View aligned slices and shells

The answer to problems of object aligned slices are view aligned slices or shells (see figure 18). Instead of keeping a 3D texture or 3 stacks of 2D textures the data is sampled on-the-fly. With this approach it is irrelevant how the proxy geometry is spanning the volume; thus we can use the best geometry for the current view to provide best sampling points. For this the card must support 3D textures and be able to address them directly with three coordinates. This means fast hardware trilinear interpolation.

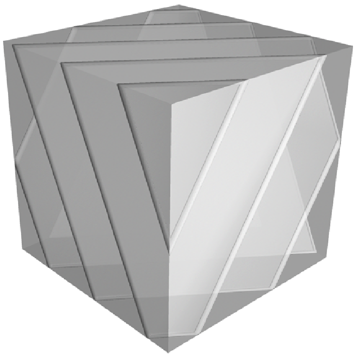


Figure 18: View aligned slices

---

[73] Compositing many textures on one primitive by alpha blending them.

Since the slices are freed from any restrictions new slices can be added on-the-fly to correspond to the viewing requirements. To further approximate equidistant sampling of raycasting instead of using slices made out of quads we can use shells; shells are fragments of spheres the centres of which are at the same point as the camera. The more complex the sphere, i.e. the more triangles it is made of, the better approximation of equidistant sampling is achieved. This approach however can become inefficient if the spheres are too complex and thus produce too many triangles. However such measures are only needed in extreme cases of perspective projection since at normal viewing angles the artifacts produced by not equidistant sampling are negligible.

3.5. Iso-surfaces

When exploring a volumetric dataset it is sometimes not needed to integrate all the data but it suffices to show the boundary of an object or its part. This can be done by selecting a value that can produce an iso-surface by rendering the voxels which have values within the selected threshold.

Iso-surfaces can be rendered with polygons. Rendering polygonal iso-surfaces is native to the hardware of current graphics cards. The only problem is converting the volume into a polygonal mesh. This can be done in several ways; a very good algorithm for such conversion is the marching cubes algorithm described in the following subchapter. Having the polygonal mesh it can be rendered extremely efficiently by the hardware. There are many algorithms provided by graphical API[74]s for shading and illumination of polygonal surfaces along with the possibility to write custom shaders.

Alternatively the volume may be rendered by raycasting or sampling via proxy geometry by discretising the opacity values into 1-bit values according to the selected threshold. This doesn't necessitate any conversion but requires algorithms for shading and illumination. Shading can be achieved by calculating the gradient for each voxel by analysing its neighbouring voxels [11]. The gradient can be either computed on-the-fly or stored as a 3D texture. The RGB values store the coordinates of the normalised vector and the A value stores its intensity. Now the fragment can be shaded during rasterisation by any shading model.

---

[74] API - application programming interface, such as DirectX or OpenGL.

3.5.1. The marching cubes algorithm

Probably the best algorithm for creating polygonal iso-surfaces is the marching cubes algorithm. It is the most optimal approach since it uses a minimal set of possible solutions to all possible combinations of 8 neighbouring voxels grouped in a cube. Despite seeming an obvious solution the algorithm had been patented in 1985 [25] but now that the patent has expired it is free to be used by everyone.

The algorithm evaluates the whole volume by grouping 8 voxels at a time into cubes. The values at each of the 8 locations are compared to the value of the iso-surface and are either evaluated as inside or outside the iso-surface. The resulting combination of these 8 voxels can be treated as an 8-bit integer that can be used as an index for a lookup table that stores all the 256 possible combinations (see figure 19). An array of vertices is retrieved from this lookup table. The vertices are described on the cube's edges. The exact location along the edge can be interpolated linearly from the original values of the voxels at the ends of each edge.

Two cases, one that has all the voxels outside and one that has all of them inside, don't contribute to the iso-surface mesh. All other combinations can be grouped into 14 families. A family is a group of combination that can all be derived from each other by rotating, mirroring or reversing one of them.

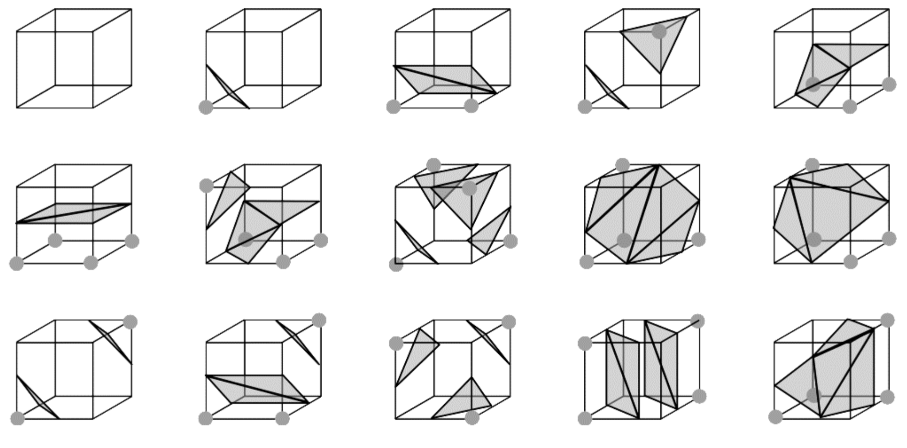


Figure 19: Marching cubes cases

4. Performance optimisations

In real-time rendering algorithms are required to achieve the best possible results with as little calculations as possible. Compromises must be often made between quality and performance. The rendering can't be made by brute force[75] with straightforward algorithms but in an intelligent manner by algorithms which can adapt to best suit the data and take advantage of its characteristics.

There are many techniques which can enhance the rendering algorithms presented in chapter 3. They can be grouped into two categories: techniques which optimise the rendering process and techniques minimising the memory usage.

Rendering oriented optimisations can minimise the number of calculations and memory access time, since less texture fetch operations are needed. Data oriented optimisations minimise the size of data in memory and some allow for substituting expensive texture fetch operations with calculations that can produce the required data on-the-fly.

4.1. Rendering oriented optimisations

Volumetric datasets are often very large. In order to render them in an acceptable quality they have to be sampled quite densely. With today's standard resolutions of up to and above of 1024x768 the number of sampling points would be far too large to evaluate them if one was to try and include all of them. Techniques are needed to decrease the number of samples to be integrated into the final image. Since not all data contributes visibly to the final image it can be safely ignored.

---

[75] The term brute force is usually used in cryptography and refers to trying to guess the password or key by trying every possible combination. In this context this means simply using the most straightforward mathematical equation approximating the object's appearance and evaluating it as many times as it is needed without taking advantage of data reliant optimisations.

Volumetric datasets often contain varying amounts of data arranged non-uniformly in the volume. Especially after applying the transfer function the data becomes mostly transparent. The techniques presented use this characteristic to decrease the amount of calculations in regions that contribute less to the final image.

4.1.1. Early ray termination

Early-ray termination is the most powerful optimisation technique for raycasting. It relies on the fact that once enough opacity along the ray is acquired by a pixel any further samples will have little or no impact on it. The level of opacity at which the ray is terminated can be set arbitrarily to set the compromise between quality and speed.

This optimisation can also be applied to hardware accelerated rendering by using an additional fragment shader after each pass to break the loop if the opacity has reached set levels. This of course requires front-to-back compositing and employs early z-test performed before every pass to determine if the fragment shader for this pixel needs to be launched. While this doesn't stop evaluating the fragment like it does with software raycasting, where the ray is actually terminated, it does skip the time consuming fragment shader.

The technique creates an immense speed-up for dense volumes. If the volume is sparse it should be replaced by empty space skipping as the opacity check and early z-test create an unnecessary overhead.

4.1.2. Adaptive sampling

An important raycasting optimisation is adaptive sampling. Volume data rarely contains data spread evenly; it often consists of large regions of empty space with an object in the centre. Also the object itself may contain homogenous areas. Raycasting however samples the data at the same distance irrespective of this. Some samples however don't contribute anything to the result thus finding a way to eliminate them would greatly speed up the rendering.

Adaptive sampling adjusts the sampling distance according to the homogeneity of the volume's partition. This can be done by pre-computing a 3D texture which will contain the value of the sampling step size for parts of the volume. This texture is called the importance volume [6] and has usually a smaller resolution than the actual data volume. It is computed by analysing the difference between scalar values in each partition corresponding to a texel of the importance volume.

Once the importance volume is created it can be accessed at each step to determine the size of the next step.

4.1.3. Empty space leaping

Volumetric data often contains large partitions of empty space. Even if the data itself is distributed across the whole volume the transfer function may still remove most of it and leave large partitions which do not contribute to the rendered image. To avoid evaluating these areas of the image and thus not waste time casting rays into them we need to know which rays actually never intersect any partition with visual data. This optimisation is called empty space leaping [14].

There are many ways of creating and maintaining data structures which can be quickly evaluated for empty spaces. Most commonly used structure is an octree[76] which can store the maximum and minimum scalar values [17] which after evaluation by the transfer function can be set to visible or invisible. This must be done only once every time the transfer function changes. If the raytracing is done by the GPU the octree can be stored as a 3D texture [14] with the minimum and maximum values stored as colour components and visibility in the alpha channel. The octree depth is limited by the resolution of the 3D texture which is usually significantly smaller then the resolution of the rendered dataset.

When the ray is traversed an additional condition is checked for every step. In hardware raytracing this is incorporated into the fragment shader. If the sample is taken in an empty region then the depth buffer is set to maximum and the fragment program for this sample is skipped; when the sample is taken in a region containing visual data the depth buffer is zeroed and the data is integrated.

---

[76] Octree is a hierarchical data structure which iteratively partitions the space into eight nodes which in turn can be partitioned into eight nodes and so on.

Alternatively instead of sampling along the ray at all points and skipping the time-consuming integration the we can store the distance to the nearest non-empty node which works in a similar fashion as a importance volume determining the next step size [18]. This is best stored in a non-hierarchical structure for faster access and stores the proximity-clouds of the data; these are the values which correspond to the step size needed to be taken to get to the next non-empty sampling point. The values create iso-surfaces around the visible voxels and virtually 'slow down' the incoming ray when there is data to be integrated ahead and 'speed up' the ray in empty regions.

4.1.4. Deferred shading

During normal rendering the shading of pixels is computed for all, even the invisible ones. Since shading can become complex limiting it to pixels that actually contribute significantly to the image can significantly boost performance.

Deferred shading is a technique that moves shading to the end of the pipeline. Instead of shading all the original samples the end result is shaded with help of additional images that store the necessary data to compute it. This reduces the complexity of shading to the final image's resolution. The reduced complexity allows for more sophisticated shading techniques which can produce significantly better results.

The most common application of this is shading of the iso-surfaces. The gradient for the final image is stored in a 2D texture and the shading can use this to reconstruct the normals at all the pixels of the final image.

4.2. Data oriented optimisations

The size of volumetric data is by far the largest problem to overcome when trying to achieve real-time rendering. The biggest bottleneck of rendering isn't the amount of calculations but the memory fetches they require; it so even on new graphic cards which offer high bandwidth communication with fast video memory.

Using the characteristics of volume data and common compression methods data oriented optimisations lower the memory requirements for storing the data. Techniques like procedural texturing allow for calculating the data on the GPU rather than fetching it from memory increasing the rendering speed dramatically. Wavelet decomposition improves the adaptive sampling process by providing blocks of the volume at more appropriate resolutions for sampling.

4.2.1. RLE

The letters RLE stand for run-length encoding. It is a common and very simple compression method which takes advantage of repeating values and stores them as a single value with attached information of how many times it is repeated. The simplicity of the algorithm limits its applicability to binary datasets[77] or indexed datasets[78]. The simplicity of the algorithm however means it doesn't add to the computational load. The only problem is restructuring the data if it changes.

The technique is only useful for software rendering as it relies on custom data structures which can't be stored in textures. They can be stored in a 2D array which contains pointers to lists where each element stores the start of the section and its height and points to the next element. This is extremely effective for software renderers since it can store large datasets using up little memory; rendering computes the necessary voxels on-the-fly without ever decompressing the volume. The ray traversal of such a volume can be very quick if the rays are cast parallel to the RLE direction. For engines with 4 degrees of freedom[79] this is natural but with additional tweaking this method of rendering can even be implemented in an engine with full freedom[80].

When used with conjunction with tileable 3D textures mapped onto the binary volume this technique may be very useful for rendering terrain with multiple layers.

---

[77] Dataset that only stores the opacity values of the object restricted to 0 and 1.

[78] Rather than storing linear values it stores the index of the value which is found in a lookup table (the palette).

[79] It is an engine which allows the camera to move along the surface, up and down and rotate around the axis perpendicular to the bottom of the volume.

[80] It is an engine which allows free movement of the camera.

4.2.2. Hardware texture compression

In hardware accelerated volume rendering techniques the large number of textures may create too big a demand for memory. Textures are used for source data, intermediate results, look up tables and various other data. An efficient way of storing them has been developed which allows for fast hardware based compression which allows for better utilisation of fast video memory.

Hardware compression had been first introduced by S3 Graphics [19] as the S3TC[81] algorithm. Presently there are five variations of this algorithm named DXT1 through to DXT5. They are specialised versions of the algorithm designed for particular texture types.

The S3TC is a lossy[82] compression algorithm. It divides the texture into 4x4 blocks and compresses them individually. The compression ratio is static[83]. Each block is stored as a 2-bit paletted image with only two colours explicitly stored and the intermediate two interpolated (see figure 20). The artifacts created by this compression technique depend on the image; some images create more visible artifacts than others depending on their nature.

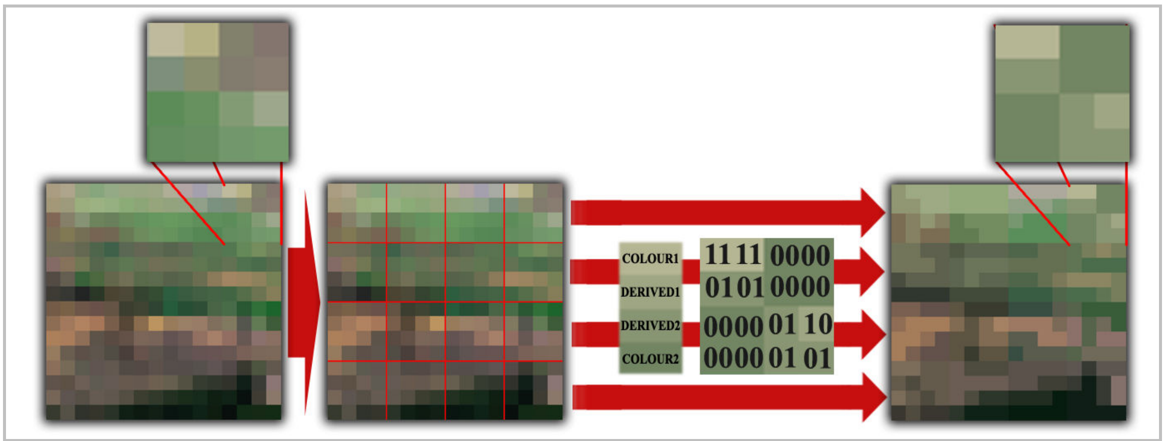


Figure 20: S3 texture compression

---

[81] S3TC – S3 Texture Compression

[82] Creating degradation in quality.

[83] 8:1 or 6:1 in DXT1 and 4:1 in others.

The S3TC standard is only really useful for 2D textures. Applying the same algorithm to 3D data would produce unacceptable artifacts. Also, the compression is biased towards human perception giving more weight to colours better perceived by humans so it's not suited for compressing arbitrary scalar data.

4.2.3. Wavelet decomposition

Volume datasets commonly comprise of regions of varying homogeneity. While some regions are rich in fine detail there are often vast partitions of data which is either homogenous or could be approximated accurately with a smooth gradient. Wavelet decomposition apart from serving as a possible method of compression which distinguishes such regions can be used to provide the rendering pipeline with hierarchical multi-resolution data. Since not all regions are equally important some partitions of the volume may suffice to be sampled from a smaller resolution volume.

Wavelet decomposition is derived from wavelet transformations which use a set of basic functions called wavelets to reconstruct the original signal. These wavelets are linked into a hierarchy which allows accessing the volume or part of it at different resolutions. Such wavelet decomposition algorithm is described by K. G. Nguyen and D. Saupe [20].

The volume is divided into smaller partitions 2n voxels per side. Each partition is low-pass filtered by the wavelets which results in a smooth sub-volume stored in n^3 voxels. The original partition is also high-pass filtered to produce $(2n)^3$-n^3 wavelet coefficients which store the detail for each voxel in the low-pass filtered sub-volume. Eight neighbouring low-pass filtered n^3 voxel sub-volumes are grouped into a $(2n)^3$ partitions and the process is repeated until there remains a single n^3 sub-volume. This process constructs the wavelet tree from leaves to a common root.

The result tree stores the volume at increasing resolutions which can be loaded on demand. This way when rendering, only the needed amount of detail is loaded from memory. By comparing the screen resolution with the partitions resolution and its size on screen we can decide if another descent into the tree of wavelets is needed.

The advantage of using wavelet decomposition is that it doesn't increase the size of the volume like pyramidal decomposition[84] and can be used to implement a lossy compression by setting a threshold for the high-pass filter which will discard too fine a detail.

4.2.4. Volume packing

Volume packing is a method of efficiently using GPU to store volumetric datasets. The basic idea of volume packing is to divide the original volume into blocks and store them scaled down by varying degrees and packed tightly together in a smaller volume [21].

The original volume is divided into blocks with overlapping border voxels. This is done to achieve the same results with bilinear interpolation when sampling from the packed volume. Depending on the data density in the block it is scaled down and packed with others into a smaller volume; less dense blocks are scaled down more than dense blocks. The original volume is replaced with a volume with indexes of the blocks in the packed volume and the scale to which they have been scaled down.

During rendering the data is unpacked on-the-fly by means of a fragment shader. The renderer samples the index volume and uses the value to create the transformation needed to find the coordinates of the voxel in the packed volume.

4.2.5. Procedural[85] and tileable[86] textures

Memory access time can be a large bottleneck in rendering. In some cases the data can be instead generated on-the-fly. It is faster than fetching it from memory and leaves more memory free for other uses. Especially in non-scientific applications the visualisation doesn't need to be an accurate representation of real scanned data. Clouds, smoke and even solid objects such as terrain need very dense data to look realistic but the data isn't concerned with where each particle of the object is but with the general layout of the object.

---

[84] A simple method of storing data at several resolutions, each being half the next one, up until the original resolution.

[85] Created by mathematical equations iterated times over.

[86] Tileable (seamless) textures if repeated multiple times do not exhibit visible borders at the edges.

To render such objects realistically we only need to store a small volume which defines the boundaries of the object and rules by which the detail is added to the volume. This can be done by storing a small volume which is repeated multiple times across the boundary volume. For best results this volume must be tileable – i.e. each side of the volume must be a natural continuation of the opposite side. Such detail volume can also be generated procedurally on-the-fly. D. Ebert et al. propose such an approach and use an additional volume for storing vectors to disturb the mapping of the procedural texture to create an even more realistic effect.

5. Quality optimisations

In real-time rendering there are often compromises in quality needed to be made in order to increase performance. In many places in the rendering pipeline these compromises may lead to visible artifacts. It is useful to know the origin of these artifacts in order to be able to try and minimise their effect on the rendered image.

Practically in each stage of the rendering pipeline there is room for quality optimisation. Moving along the pipeline we can categorise the artifacts by the stages in which they originate: sampling, filtering, classification, shading and blending artifacts.

Thanks to new functions of the GPU some of the quality problems can be solved by higher precision textures and fragment shaders. Still, moderation is needed when applying these quality optimisations as not to lose too much performance.

5.1. Sampling artifacts

The first step in the rendering pipeline is sampling the volume. Irrespective of whether we sample the volume by raycasting or use proxy geometry the resulting distances between samples have a direct impact on how accurately we can represent the original data; undersampling[87] will cause severe artifacts. The number of samples has a direct effect on the rendering speed; roughly approximating, the number of samples is proportionate to the rendering speed.

Volume data itself if zoomed into too much will show undersampling artifacts regardless of the rendering technique used. This is due to finite resolution of the scanning equipment. It is sometimes difficult to discern between artifacts due to such undersampling and the ones produced by the rendering process (see figure 21).

---

[87] Too large a distance between samples in respect to the resolution of the final image.

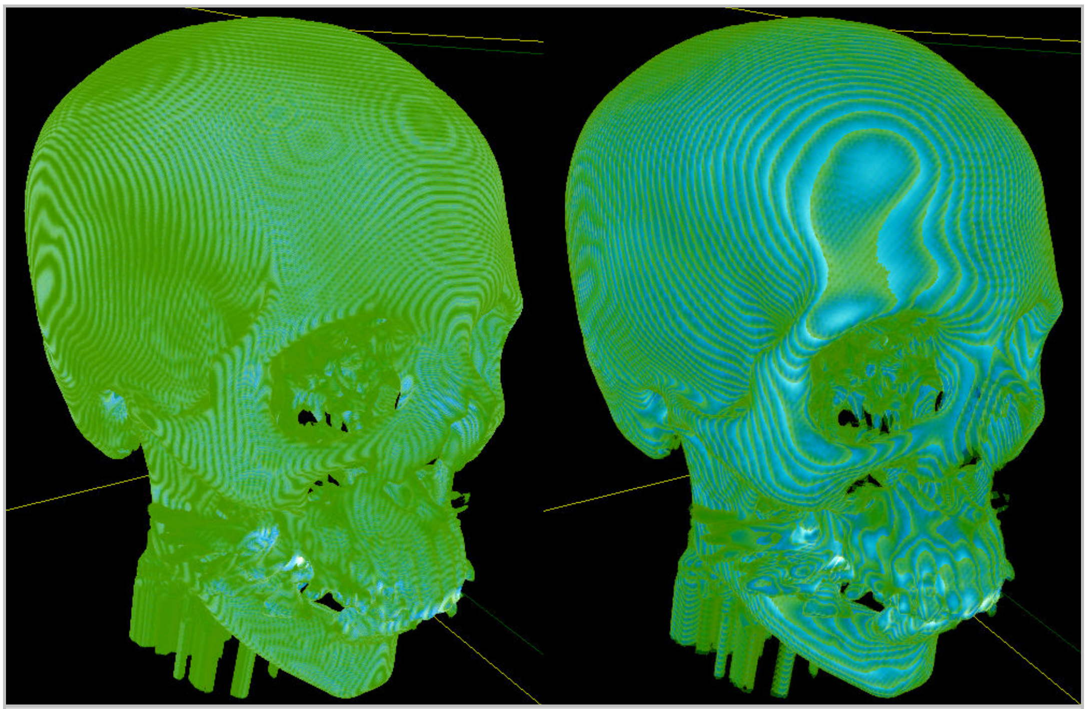


Figure 21: Undersampling artifacts inherent in data (left) and during rendering (right) (note that the inherent artifacts are present in both images both)

It may prove useful to know the minimal number of samples needed to be taken to achieve the exact data representation. The Nyquist-Shannon sampling theorem can be applied here, even though the sampled data is already in discrete form. The theorem states that when converting analogue signal to digital the original signal must be sampled at twice its frequency to allow for its perfect reconstruction. In volume rendering we actually resample the already sampled volume which represents the actual contiguous signal of the original object. But we can treat the volume as a contiguous signal of a frequency equal to its length divided by the number of voxels [6]. Thus to reproduce it correctly in the rendered image we need to sample the volume at twice this frequency; this results in sampling every voxel at least twice. It is rarely possible to do this due to the impact on performance made by so many sampling points.

In order to get around it we can increase the number of sampling points in certain areas by trading it off for reducing the number in other, less important areas. This can be achieved using a technique called adaptive sampling (described in detail in chapter 4.1.2) which uses an importance volume to distinguish between more and less important areas inside the volume. This results in smaller step sizes in the traversal of the rays in the important areas of the volume. When using proxy geometry we can't increase the number of slices since the slice spans the whole volume but instead we can use fragment shaders to sample certain areas of the slice multiple times by moving the mapping coordinates to sample the space in between slices.

5.2. Filtering artifacts

In order to be able to sample the data at arbitrary points we need to reconstruct the contiguous signal from discrete data. We do this by means of filtering. There are many techniques varying in complexity. The best results are achieved by the sync filter which is infinite in extent and therefore unfeasible to implement. In most cases we are limited to linear filtering which is the only filtering the hardware of the GPU supports natively.

In order to increase the quality of filtering we can implement our own filtering schemes by means of a fragment shader. One such technique is proposed by M. Hadwiger and T. Theußl et al. [24]. The algorithm they propose works diametrically different than traditional filtering. Determining the end result at each point is not done by sampling the neighbouring points but each point contributes to the neighbouring points instead. This is done in several passes and all the intermediate results are blended into the final result. This is a better approximation of the sync filter but is very performance hindering.

Klaus Engel et al. [6] propose that since the filtering artifacts are not visible during manipulation of the volume such complex filtering technique should only be used when the volume is static on the screen. To achieve smooth movement hardware native filtering provides acceptable quality.

5.3. Classification artifacts

Classification converts the scalar values in the dataset into visual attributes that can be rendered. The classification can be done either before or after filtering. The difference is very visible especially for high frequency transfer functions.

In pre-classification[88] high frequency transfer function are very badly reproduced (see figure 22). In post-classification[89] schemes the transfer function is reproduced properly but the frequency of the resulting volume becomes higher then that of the original dataset and would require a higher sampling rate to reproduce correctly. If the sampling rate is not increased post-processing creates sharp transitions. In order to reproduce the volume correctly the Nyquist-Shannon theorem would have to be applied to the frequency of the transfer function which could result in a too high sampling rate for the hardware to render interactively.

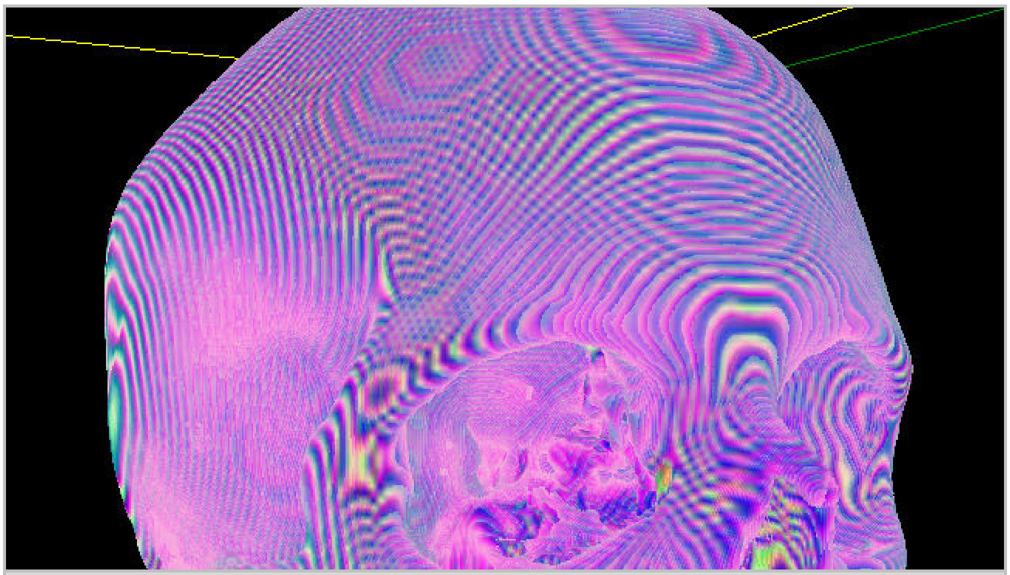


Figure 22: Filtering artifacts in pre-classification

---

[88] The classification is done on the original not-filtered data.

[89] The classification is done on the filtered data which better reproduced the transfer function.

Pre-classification reproduces the transfer function better but requires a higher sampling rate to produce smooth results. This can be circumvented by pre-integration of the transfer function [6]. This works by creating a lookup table for the transfer function which can produce the integration of the transfer function between any two scalar values. The lookup table is then addressed by data sampled from two points in the volume. This can produce a good approximation of pre-classification result at a higher sampling rate without actually increasing it. Integration of the lookup table can be very time-consuming but needs to be done only when the transfer function changes.

5.4. Shading artifacts

Shading is an optional step in rendering used when external lights are to be taken into account when rendering the volume. Shading can produces additional depth cues to better visualise the volume. It can be costly as it requires evaluating the gradient for each voxel in relation to the neighbouring voxels. Although it can be done on-the-fly it is better, performance-wise, to pre-compute the gradient and store it for easy access as a single fetch operation. The stored values of normals are normalised to an 8-bit range and filtered when fetched. This can produce visible artifacts especially in comparison with the results of rendering with normals computed on-the-fly. The only way to increase the quality is either to use the on-the-fly computed normals or store the pre-computed normals in higher precision textures.

The increase in quality is available at either a cost in multiple memory reads for on-the-fly gradient computation or additional memory space for storing the 16-bit texture for pre-computed normals.

5.5. Blending artifacts

Blending is done in the final step of the rendering pipeline. It integrates all the intermediate values accumulated during the sampling of the volume. Since the intermediate results are stored as quantised 8-bit values this can produces serious artifacts as the errors all accumulate in the 8-bit final texture.

To circumvent this problem the actual blending is done in a floating point 16-bit or 32-bit texture. The screen buffer is limited to an 8-bits fixed point accuracy therefore the floating point texture is rendered to by the ‘render to texture’ function and then the final result is copied into the frame buffer. The additional precision is costly in calculations and memory access time so a compromise needs to be made.

6. Voxel technology – a summary

Volume rendering is a powerful visualisation model. Developed decades ago it's a mature technology which offers unique properties which make it irreplaceable in many applications. Somehow overlooked by consumer hardware manufacturers it hasn't yet become widespread outside of technical applications. There, its services are irreplaceable at visualising 3D data in a meaningful way that can't be achieved to the same extent by polygonal models.

Real-time capabilities of volume rendering add easy and intuitive manipulation and enhance the ease of viewing of volume data. Many techniques exist and differ from field to field and from implementation to implementation. Recently commercial applications have adopted the GPU available on consumer hardware to increase their performance. This applies to all techniques – raycasting, rendering by proxy geometry and iso-surfaces. Due to the performance boost of using shaders techniques which can't benefit from it are being abandoned though some, like the shear-warp algorithm, still can prove competitive.

Benefits of real-time visualisation are recognised to be important enough to warrant compromises in quality of the rendering. Many optimisations have been developed to help minimize the effect of these compromises and increase overall performance of rendering. Today, using the hardware of modern graphic cards we can achieve high quality and real-time volume rendering on cheap consumer PCs.

6.1. Prospects of voxel based technology

The development in graphic cards' hardware and APIs promises better support for these rendering techniques. The new shaders 4.0 will support true branching and independent execution of all pipelines, significantly improving performance of shader-heavy rendering techniques. Additionally there is a trend to replace specialised shader hardware with unified shaders which can carry out any shader program thus allowing for more flexible load balancing of computations. This allows for an even fuller use of the available computing power. The new geometry shader will allow creating proxy geometry on-the-fly by the GPU rather than storing it in memory or creating it on the CPU. This and many other improvements expand the programmability of the GPU negating the disadvantage of using voxel based rendering techniques and placing the two techniques at more of an equal footing.

Given the rate at which the graphics in modern games and movies get closer to photorealism the voxel model may soon be more widespread as it offers certain features polygonal models are lacking in. Commercial hybrid engines[90] have been created before and may still make a comeback. Should voxels become more widespread again then the hardware will be forced to support them even more.

Hardware capabilities of consumer hardware are allowing for more and more complicated models of approximating reality. Soon the requirements of rendering may exceed the polygonal model's capabilities and the voxel model might be the one to fill the function.

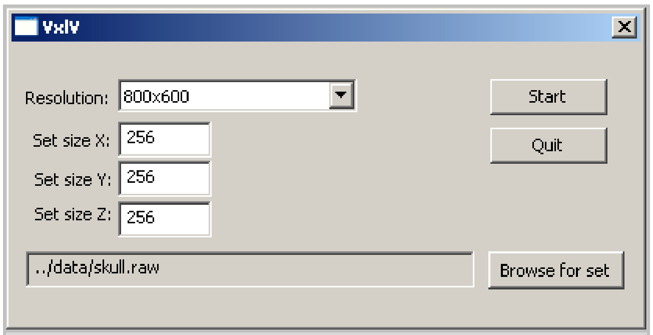
---

[90] Hybrid engines render both polygons and voxels.

A. Accompanying application's documentation

The accompanying application named VxlV (Voxel Viewer) has been developed in order to illustrate and evaluate some of the algorithms described in this thesis. It is written in C++ for the windows environment and it uses OpenGL as its rendering API. It also takes advantage of the glut package and uses shaders written with the aid of the CG language.

The application is capable of displaying volume datasets read from raw[91] format data files of resolution up to 256 on a side. It allows for free exploration of the data set and changing of the parameters of the visualisation interactively.



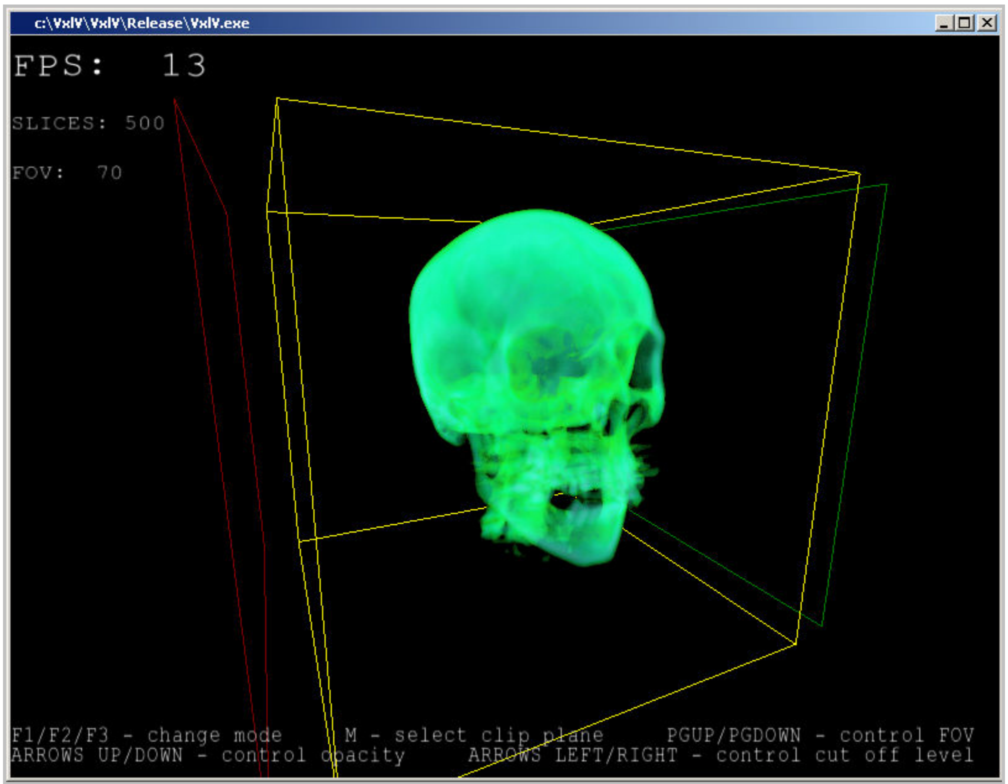
Resolution: 800x600
Set size X: 256
Set size Y: 256
Set size Z: 256
../data/skull.raw

Figure 23: Application launcher

The application presents the user with a launcher (see figure 23) which allows for setting up starting parameters. Any raw file containing volumetric data can be chosen for viewing, provided the dimensions of the data fit into a 256x256x256 texture. For proper interpretation of the raw file its dimensions must be manually entered. The user can now choose the starting resolution for the rendering window and start the program.

---

[91] Raw format files remember the volume data as 1D array of 2 byte values without any header information - the program must know how to interpret the data in order to reconstruct a 3D volume data



```
c:\VxlV\VxlV\Release\VxlV.exe
FPS:   13
SLICES: 500
FOV:   70
F1/F2/F3 - change mode      M - select clip plane      PGUP/PGDOWN - control FOV
ARROWS UP/DOWN - control opacity      ARROWS LEFT/RIGHT - control cut off level
```

Figure 24: VxlV

The main program window (see figure 24) displays volume data and basic debug information along with basic help. The dataset can be freely rotated and zoomed in with the use of the mouse by clicking and dragging. Other controls are displayed at the bottom of the screen.

After launching the program the program first converts the raw data into 3D textures. One of them contains the original values in a greyscale texture. A colour texture is created by applying a transfer function to the greyscale texture. The program can display either of the textures and the user can switch between them by pressing ‘t’.

The transfer function used to create the colour presentation of the dataset can be customised. By pressing ‘e’ the user can bring up the transfer function editing window (see figure 25). The colour channels and the alpha channel can be edited individually or together (you can activate any channel by pressing keys ‘1’ through ‘4’ or by pushing an appropriate button) simply by ‘painting’ them with the mouse cursor. Pressing ‘e’ again will turn off the editing window and recalculate the colour texture.

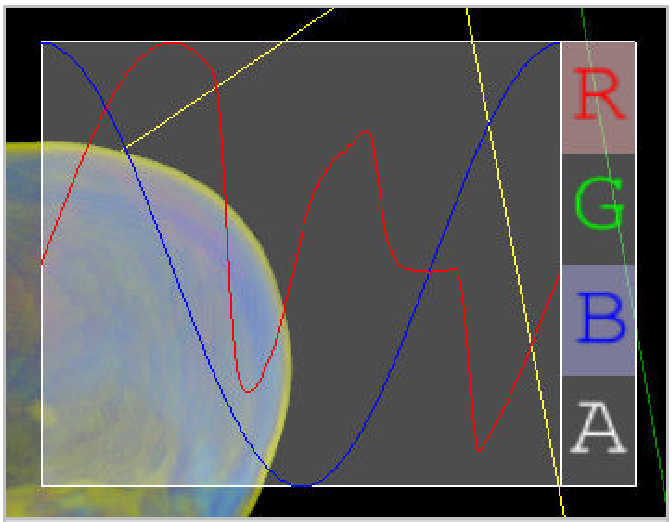


Figure 25: Transfer function editing window

For better exploration of data VxlV supplies the user with two clipping planes which can be moved independently to cut through the data set and let the user see a cross-section of it. You can switch between moving either of the clipping planes and the dataset itself by pressing the 'm' key. The currently chosen object remains highlighted.

VxlV provides the user with three modes of rendering the data. The mode can be changed freely at any moment by pressing 'F1', 'F2' or 'F3'. Each of the modes implements a different algorithm of volume rendering (see figure 26).

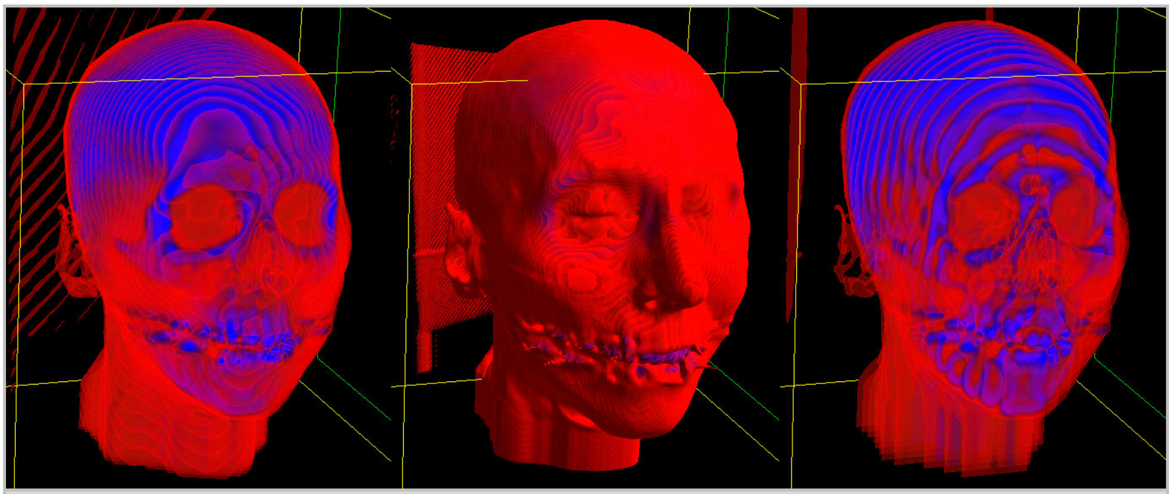


Figure 26: Modes of rendering (left to right): view aligned shells, view aligned slices on the GPU, object aligned slices

In the first mode the rendering is done by sampling a 3D texture with view aligned shells (see chapter 3.4.3). This mode is the slowest as the shells are very complex compared to slices used by the other two algorithms. The upside of this technique is that you can freely manipulate the FOV angle (by pressing the ‘page up’ and ‘page down’ keys) without any opacity errors created due to perspective distortion.

The second mode of rendering implements simple slices in the form of quads instead of shells. The rendering is done on the GPU by means custom shaders. This makes this the slowest technique in VxlV since the custom shaders are longer than those in the fixed pipeline used by the other two modes. The custom shaders simulate lighting on the volume which greatly enhances the visualisation’s readability.

The third mode of rendering uses three stacks of object aligned slices for sampling the texture (see chapter 3.4.2). By simple virtue of simpler geometry it proves faster than the first mode but at the cost of opacity errors at wider angles of camera’s FOV (see figure 27) and noticeable jumps when stacks are being switched.

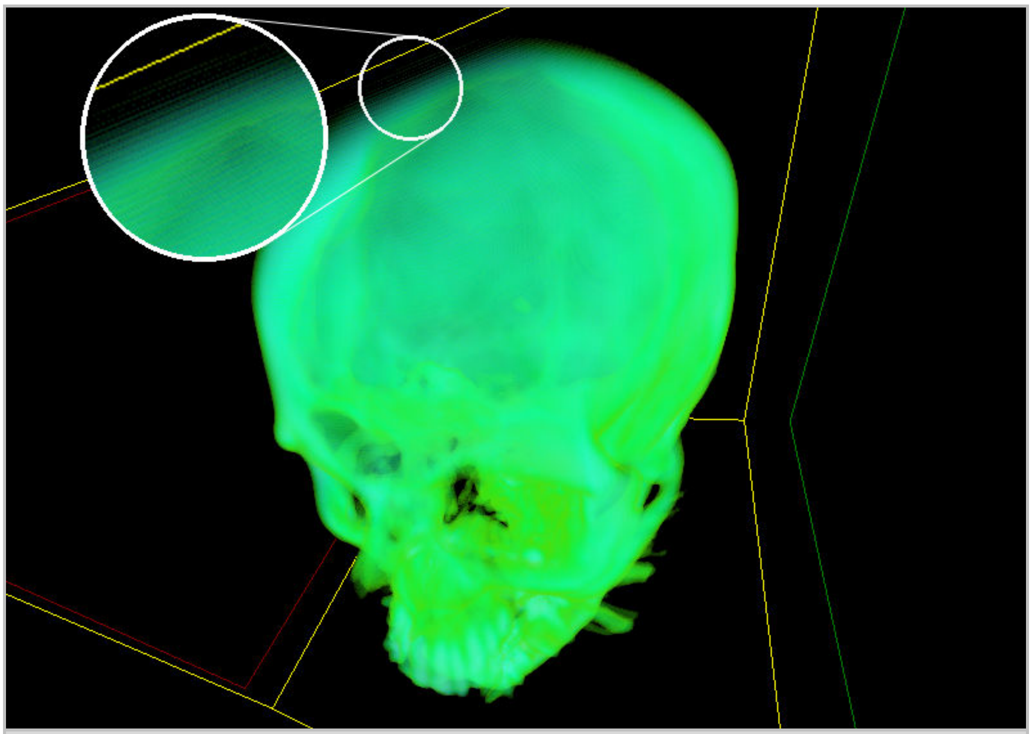


Figure 27: The darkening at off-centre areas of the volume is due to opacity errors caused by a large FOV angle

B. Index of figures

C. Bibliography

[1] Paul S. Calhoun, BFA, Brian S. Kuszyk, MD, David G. Heath, PhD, Jennifer C. Carley, BS and Elliot K. Fishman, MD; *Three-dimensional Volume Rendering of Spiral CT Data: Theory and Method*; presented as an infoRAD exhibit at the 1997 RSNA scientific assembly; accepted October 30

[2] Heath DG, Soyer PA, Kuszyk BS, et al.; *Three-dimensional spiral CT during arterial portography: comparison of three rendering techniques*; RadioGraphics 1995

[3] William E. Lorensen, Harvey E. Cline; *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*; SIGGRAPH '87 Conference Proceedings; ACM Siggraph 1987

[4] Brian S. Kuszyk and Elliot K. Fishman; *Technical Aspects of CT Angiography*; retrieved from http://www.ctisus.org/angioatlas/syllabus/techniques_CT_ANGIO.html on the 2nd of March 2007

[5] Alvy Ray's smith homepage; *History Pixar Founding Documents*; retrieved from http://alvyray.com/Pixar/default.htm on the 2nd of March 2007

[6] K. Engel, M. Hadwiger. J. M. Kniss, A. E. Lefohn, C. Rezk Salama, D. Weiskopf; *Real-Time Volume Graphics*; Course Notes 28, SIGGRAPH 2007 Conference Proceedings; ACM Siggraph 2007

[7] Arie Kaufman, Daniel Cohen, Roni Yagel; *Volume Graphics*; IEEE Computer, Vol. 26, No. 7; July 1993

[8] Kaufman, A.; *Volume Visualization*; IEEE Computer Society Press Tutorial; 1990

[9] D. H. Laidlaw, W. B. Trumbore, J. F. Hughes; *Constructive solid geometry for polyhedral objects*; Proceedings of SIGGRAPH '86; ACM Siggraph 1986

[10] Steve Seitz; *From Images to Voxels*; SIGGRAPH 2000 Course on 3D Photography; ACM Siggraph 2000

[11] Joe Kniss, Gordon Kindlmann, Markus Hadwiger, Christof, Rezk-Salama, Rüdiger Westermann; *High-Quality Volume Graphics on Consumer PC Hardware*; VIS2002 presentation; 2002

[12] Nelson Max; *Optical models for direct volume rendering*; IEEE Transactions on Visualization and Computer Graphics; June 1995

[13] P. Lacroute, M. Levoy; *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*; Proceedings of SIGGRAPH ‘94; ACM Siggraph 1994

[14] J. Krüger and R. Westermann; *Acceleration Techniques for GPU-based Volume Rendering*; Computer Graphics and Visualization Group; Technical University Munich 2003

[15] S. Röttger, S. Guthe, D. Weiskopf, T. Ertl; *Smart Hardware-Accelerated Volume Rendering*; Proceedings of EG/IEEE TCVG Symposium on Visualization; VisSym 2003

[16] Durand and Cutler; *Lecture Notes: Graphics Pipeline*; MIT OpenCourseWare, Computer Graphics; Fall 2003

[17] Wilhems, Gelder; *Multidimensional trees for controlled volume rendering and compression*; Proceedings of ACM Symposium on Volume Visualization; 1994

[18] R. Yagel, Z. Shi; *Accelerating Volume Animation by Space Leaping*; Proceedings of SIGGRAPH ‘93; ACM Siggraph 1993

[19] Aleksey Berillo; *S3TC and FXT1 texture compression*; retrieved from http://www.digit-life.com/articles/reviews3tcfxt1/ on the 24th of March 2007

[20] K. G. Nguyen and D. Saupe; *Rapid high quality compression of volume data for visualization*; Computer Graphics Forum #20; March 2001

[21] Martin Kraus, Thomas Ertl; *Adaptive Texture Maps*; Proceedings of Eurographics; SIGGRAPH Graphics Hardware Workshop; 2002

[22] D. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, S. Worley; *Texturing and Modeling: A Procedural Approach*; Academic Press; July 1998

[23] C. E. Shannon; *Communication in the presence of noise*; Institute of Radio Engineers, vol. 37, no.1; 1949

[24] M. Hadwiger, T. Theußl, H. Hauser, E. Gröller; *Hardware-accelerated high-quality filtering on PC hardware*; In Proceedings of Vision, Modelling, and Visualization 2001; 2001

[25] General Electric Company; *United States Patent 4,710,876*; Filed on the 5th of June 1985