Investigating Octree Generation for Interactive Animated Volume Rendering

by

David Reilly, B.Sc. in Computer Science

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

August 2011

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

David Reilly

August 30, 2011

Permission to Lend and/or Copy

Acknowledgments

I'd like to thank my supervisor John Dingliana for all his help and guidance throughout the project, and also throughout the year. I would also like to thank my fellow classmates for helping me get throughout the year and getting through those late nights. I'd also like to thank my family and friends for their great support throughout the whole year, it is greatly appreciated.

David Reilly

University of Dublin, Trinity College
August 2011

Investigating Octree Generation for Interactive Animated Volume Rendering

David Reilly
University of Dublin, Trinity College, 2011

Supervisor: John Dingliana

Volume rendering is a huge field which has received heavy research for its use in domains such as medical imaging, physics simulations and special effects for use in films. A big problem within the volume rendering field is rendering animated volumetric data at interactive rates, due to the large information sets associated with the data.

A commonly used approach to optimise volume rendering is to use octrees, but it has not fully been shown how this can be done for animated data, in particular physically based animations or user-defined deformations.

This dissertation investigates the use of octrees in the field of volumetric rendering and analyses their use in the rendering of animated volumes in interactive scenes. Certain octree optimization approaches are investigated with the view to reduce the overall time required to generate an octree at interactive rates. Furthermore, the dissertation studies the generation of octrees over a number of frames, progressively drawing the octree as it is generated. Multiple threads are utilized to achieve faster octree generation. This dissertation also investigates methods to adapt an octree for modifications made to the volumetric data.

To evaluate octrees for animated volumetric rendering, this dissertation will analyse the performance of octree generation during run-time, including the advantages and disadvantages of certain optimizations, and conclude how applicable octree generation at interactive rates is for animated volumetric rendering.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Volume Rendering is an alternative approach to triangle-based rendering, which is used in fields such as medical imaging, geophysics, fluid dynamics, image processing, and more recently in special effects for films to represent fuzzy objects such as clouds and smoke. The recent success of games such as Minecraft®(Mojang) and Crysis®(Crytek, Electronic Arts), which used voxels for it's terrain engine, have also brought attention to volume rendering in games, with some voxel engines built specifically for games such as the Atomontage engine [7].

Despite great advances in volume rendering, as demonstrated from recent research by Crassin et al. [8] and Laine and Karras [9], implementation of volume rendering in interactive entertainment has been limited to a small number of commercial games and independent games. One of the reasons for the limited use of volume rendering is the difficulty associated with rendering animated volumes. Animated volume rendering at interactive rates is currently the subject of heavy research, and is a major stumbling block for the use of volume rendering in interactive entertainment such as games, with current implementations strictly for static objects only, such as terrain.

This introduction chapter will first give a general introduction to the concepts relevant to the work in this dissertation. This will allow for full appreciation of the challenges involved in animated volume rendering, which are discussed next. Following this, the aims and motivation of this dissertation are presented, followed by the layout of the dissertation.

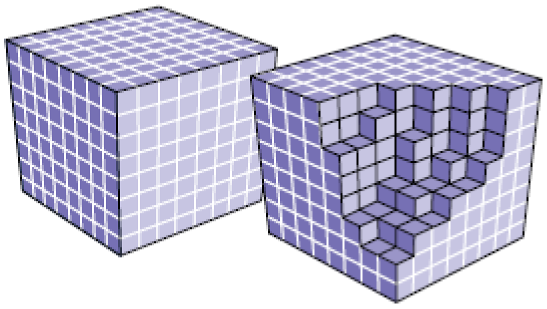


Figure 1.1: A cube volume divided into voxels [1].

1.1 Concepts

Volumetric data is an array of *voxels* (short for volume picture element), in three-dimensional space. Voxels, generally organized in a regular three dimensional grid allow for easy access and storage, are cuboids that typically have a scalar value associated with them (see Figure 1.1). They are used to represent three-dimensional objects by combining many voxels together (millions are required for complex objects) (see Figure 1.2). By increasing the resolution of the volume, the voxels will become smaller and less noticeable, making the object look less blocky (voxels can become smaller than the pixels used to present them). As mentioned, voxels have a scalar value associated with them, they do not have positional information, instead this is derived based on the position of the voxel relative to other voxels within the volume.

To help visualise and understand volumetric data, consider a two dimensional image. A two dimensional image consists of an array of pixels (picture elements), which store the colour information for the image. A volumetric data set can be considered as a collection of two-dimensional images stacked on top of each other, with each voxel considered as a three-dimensional pixel representing a scalar value calculated by sampling the local area surrounding the voxel.

Volume rendering is a set of techniques that project the three-dimensional volumetric data onto a two-dimensional plane, the view plane. There is a vast number of volume rendering techniques used to achieve this, such as *volume ray casting*. An in-depth discussion on the popular volume rendering techniques can be found in Chapter

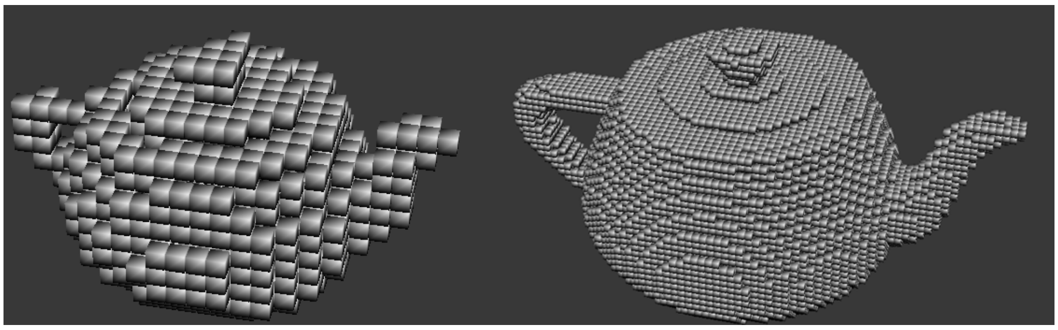


Figure 1.2: An example of a voxel model of a teapot at a low resolution (662 voxels), left, and at a high resolution (7,474 voxels), right.

2. *Animated volume rendering* involves applying these volume rendering techniques to animated volumetric data. Similar to the concept of animations in films, an animated volume is a collection of volume datasets, who typically have the same dimensions.

As mentioned, voxels typically only store scalar values, no other information is generally stored in the volume data set, including colour information, allowing the voxels to represent various properties such as bone density or fluid velocity. In order to represent the volume in colour, a *transfer function* is required to map the scalar values to colour values.

Another concept discussed in this dissertation is an *Octree*. An octree is a spatial sub-division hierarchy that sub-divides data (in this case, volume data) into eight sections, which can be further subdivided (octrees are also used in other areas such as collision detection and physics, but this dissertation is only concerned with octrees for volume data). The purpose of octrees in volume rendering is to reduce the computation required during sampling the volume. Volume data can contain a lot of empty voxels which do not contribute to the rendered volume. An octree only stores non-empty voxels, resulting in the renderer only sampling filled voxels, and not wasting time sampling empty voxels.

The final concept discussed in this dissertation is multi-threading. Multi-threading is when an application (or process) has multiple threads running, where each thread has a set of instructions and stack memory. All threads run independently of each other, but have access to the data stored on the heap for that application. Why would an application require multiple threads? Applications can perform tasks that require a number of clock cycles before the task is complete. When this happens, the rest of the application has to wait for the task to complete before continuing with other tasks. This may be acceptable for applications where there is no user-interaction or where performance does not matter, but for other applications, especially computer and video games, this wait is unacceptable. An example of this would be a web browser. If your web browser was downloading a file, without multi-threading, you would be unable to search the internet or view websites until the file was finished downloading. As far as the user is concerned, the browser would freeze when it is downloading a file. To prevent the user from waiting, multi-threading is used (in fact, some modern web browsers use multiple processes). By using multiple threads, your application can set specific tasks (such as downloading a file) to a thread which executes separately from the main application, meaning the application can continue performing other takes (like opening a new website) while the thread performs it's own task, without slowing down the main application. As a result, great performance gains can be achieved using multi-threading.

1.2 Challenges

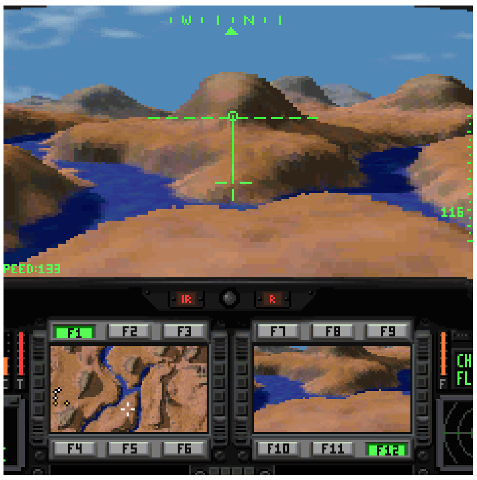
The size of animated volumetric data is a major challenge for rendering with interactive rates. In most state of the art engines, a single voxel requires at least 1 byte of memory. With a volume resolution of 256^3, this requires 16 MB of space. With today's push for high definition within interactive entertainment, such a resolution would not be appropriate. Instead a more suitable resolution would be 512^3, which results in 128 MB of space required. To achieve an animation at 30 frames a second, that would require 30 volumes, resulting in 3.75 GB of space needed to store the data, just for a 1 second animation. An appropriate animation length, a 3 second animation, would require 11.25 GB. With modern graphics cards generally capable of storing memory between 512 MB to 1024 MB, such data cannot be stored on the GPU. The CPU also has a limited amount of memory, with the current limit being 192 GB on the 64-bit Windows 7 Professional operating system, although commonly most systems can only have a maximum of 4 GB (the limitation on existing 32-bit Windows operating systems).

As mentioned previously, volumes are spatially sub-divided, commonly with an octree, to reduce computation during rendering. For animated volume rendering, this requires an octree for every frame of the animation. Using the previous example of a 3 second animation at 30 frames per second, this would require 90 octrees, further adding to the amount of memory consumed. Another option is to reconstruct or to build a new octree during the animation. Currently, the preferred approach is to create an octree for every frame or to load in the octrees from external files, both during pre-processing. This approach is appropriate for recorded, pre-set animations such as a character walking or running. However, for physically based animations or user defined deformations which feature heavily in modern games, this approach is not applicable. Instead, reconstructing the octree for each frame is needed, however the construction of an octree is a time consuming process, even more-so for high resolution volumes.

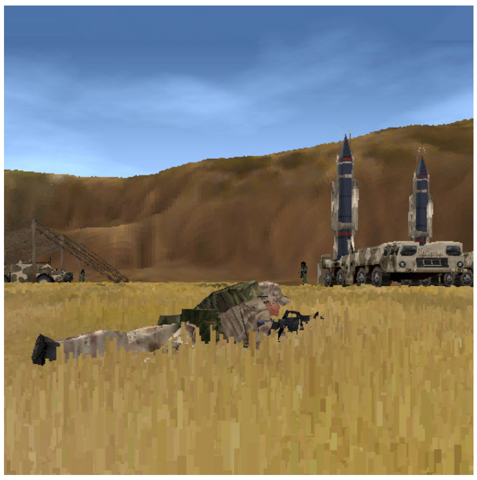
1.3 Motivation and aims

The motivation behind this project is the implementation of animated volumes in interactive entertainment technologies, including both pre-recorded fixed animations as well as physically based animations and user defined deformations. This would allow for fully complete volume rendering engines to be used in the development of interactive entertainment technologies without the use of any polygon-based rendering engines to perform specific tasks, such as character animations, enabling games to be built entirely from voxels.

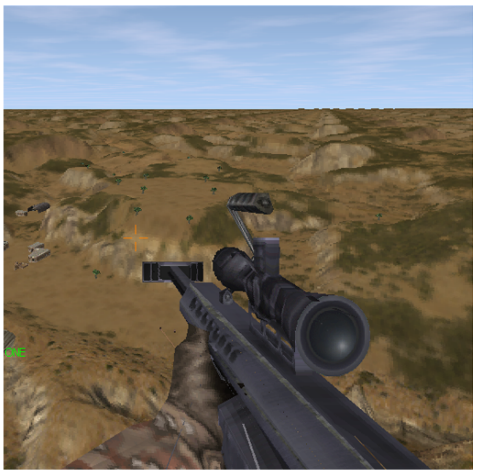
Volume rendering with voxels have many advantages over traditional triangle-based rendering. They are a more natural representation of objects, as voxels can be treated as the atoms that make up the object. As a result, voxels are suitable for representing natural details such as rocks, cliffs and caves. In fact, this is the primary use for voxels in modern games such as Crysis ®(Crytek, Electronic Arts), which implemented voxels to create overhangs, something that is not possible using height maps with triangle-based rendering, in the games terrain. Another great advantage of voxels for interactive entertainment is its use for destructible environments, which has become popular in modern games such as Red Fraction ®(Volition Inc, THQ) and Battlefield 3 ®(EA Digital Illusions CE, Electronic Arts). For example, blowing a hole in a wall simply requires removing the relevant voxels from the wall. Performing a similar task for triangle-based rendering is a lot more complex, requiring cutting holes in polygons and



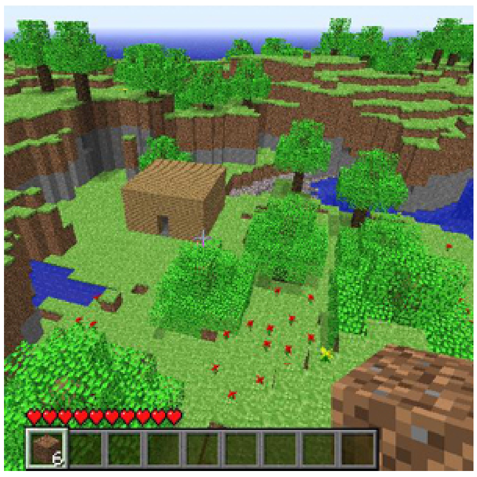
(a) Comanche ®by NovaLogic featured voxels for rendering.



(b) Voxels used for grass in Delta Force 2 ®by NovaLogic.



(c) Voxels used for the terrain in Delta Force 2 ®by NovaLogic.



(d) Minecraft ®by Mojang implements voxels.

Figure 1.3: Existing games that implement Voxel Rendering.

building the geometry behind it while setting up the new texture coordinates. Voxel rendering commonly incorporates an octree to reduce rendering costs. With this octree, level of detail also comes at no extra cost, as the rendering engine can traverse the octree to an appropriate depth based on the distance the volume is from the camera. This means that, in theory, you can have unlimited geometry detail in a scene (limited by available system memory). Voxels also allow for complete texturing, where each voxel can be assigned a different colour without requiring a large number of textures, which is the main source of memory usage for current triangle-based rendering engines. Volume rendering is also ideal for representing real world phenomena such as smoke, fire and clouds, which cannot be easily represented using triangle-based rendering techniques. Instead, these phenomena are represented using particle systems in modern games. Particle systems are not ideal, as the particles, usually point sprites, can intersect with scene geometry resulting in artefacts that ruin the illusion.

These advantages of volume rendering and voxels over traditional triangle-based rendering have not gone unnoticed by the interactive entertainment industry, with a number of video games taking advantage of voxel rendering. Most notably, the Delta Force ®series (Delta Force 1 and Delta Force 2) and the Comanche ®series (see Figure 1.3a), which was the first commercial flight simulation game rendered using voxels (in fact this game used a more restrictive definition of voxels, where the voxels were 2D bricks as the engine was not a 3D engine, but a 2.5D engine), both developed by NovaLogic who used voxels in a number of their games. Delta force 2 ®exploited voxels for generating landscapes and environments, including grass which was implemented to allow players to take cover (see Figure 1.3b), in the game world with practically unlimited draw distance, which was a major feature for the game involving sniping enemies from a distance. This was achieved by tiling the base terrain of a level in all directions, allowing the game to feature levels that exceed the size of most levels in many games to this day, despite having been released in 1999 (see Figure 1.3c). Voxels are heavily used to generate game worlds and allow the user to build and destroy objects, including the game world, in the recently successful game called Minecraft ®(Mojang) (see Figure 1.3d). The use of voxels in games has not been strictly limited to terrain generation. Command & Conquer: Tiberian Sun ®and Command & Conquer: Red Alert 2 ®(both developed by Westwood Studios and published by Electronic Arts) used voxels to render some of the game's units (such as vehicles) and buildings. Blade Runner ®, also by Westwood Studios used voxels to render characters and some objects in the game.

It is clear that the games industry can and want to use volume rendering for developing their games. However, for volume rendering to be universally implemented by modern interactive entertainment, it is important that it can perform physically based animations. There is an increasing number of games released every year, such as Grand Theft Auto 4 ®(Rockstar North, Rockstar Games), Red Dead Redemption ®(Rockstar San Diego, Rockstar Games) and Star Wars: The Force Unleased ®(LucasArts), where character and object animations are physics-based, using engines such as Havok ®and Euphoria ®, and not pre-recorded animations. The popularity of such engines is no surprise as games strive to make believable, immersive game worlds, realistic and believable animations are a must, where characters react to the game terrain and user interactions through physic-based animations, instead of playing pre-scripted animations formed in 3D animation software or motion capture.

This dissertation is aimed at investigating the use of octrees in animated volume rendering, performing an analysis of certain optimizations to octrees to reduce computation time in interactive scenes. Furthermore, the use of dedicated threads for octree generation will be investigated and evaluated to determine how applicable octree generation is to interactive scenes. This will include progressive octree generation, where the octree is drawn as it is being subdivided, with interactive rates. The main aims are:

- Investigate the length of time required to generate an octree to specific depths (for example, depth 1 to depth 8).
- Study the performance of octree generating over a series of frames, rendering each level as it is built.
- Investigate the implementation of dedicated threads for progressive octree generation.
- Study and investigate possible approaches to adapt the octree, instead of reconstructing the octree.
- Develop a prototype application for volume rendering and octree generation at interactive generation. The application should implement multiple threads and progressive octree generation. Furthermore, the application should have a user friendly interface and provide the user with options to modify the volume data, generate a new octree and also visualize the octree. The application should be platform independent where possible, only using platform specific code when required.

- Evaluate octree generation for animated volumes and it's applicability to interactive entertainment technologies

1.4 Dissertation Layout

This dissertation report is organised in the following structure:

The introduction, just covered in this chapter, introduces the most relevant concepts related to this work, with a brief description of each concept. Following, the challenges in the field of animated volume rendering were introduced. The motivations behind this research and application developed as a result was explained in detailed, with the aims outlined.

The next chapter gives a brief revision of related work performed in volume rendering, animated volume rendering and octrees, presenting the numerous approaches developed to implement these concepts.

After the chapter on related work, the design of the system is introduced. This chapter introduces the development plan for this dissertation and also breaks the system down into it's relevant components, and explains the functionality of each component, the options and decisions made when designing the system.

Next, a chapter on the implementation is presented. This chapter provides a detailed description of how each component in the application was constructed based on the design of the application.

The application is then evaluated by presenting and analysing the results obtained from the application.

The final chapter will present the conclusions reached based on the application evaluation, and suggest possible future work that is related to this dissertation.

Chapter 2

Related Work

Volume rendering and animated volume rendering have been researched in great depth, which has resulted in a wide variety of techniques to render volume data. Octrees have received similar attention with regard to the volume rendering domain and other domains such as collision detection. This chapter will discuss and review related work in the following fields:

- Volume rendering techniques.
- Animated volume rendering.
- Octrees.

2.1 Volume Rendering Techniques

There is vast array of volume rendering techniques researched in this field, and a detailed discussion on these techniques will follow. It is important to discuss the various rendering techniques available to determine what are the advantages and disadvantages of these techniques, and which is most appropriate for this dissertation.

Volume rendering can be divided into two categories, direct volume rendering and indirect volume rendering, and much work has been researched in both fields. A brief discussion will be had on indirect volume rendering, before discussing direct volume rendering in depth as it is more relevant to this research.

2.1.1 Indirect Volume Rendering

Indirect volume rendering techniques transform volume data into a different domain, generally into a set of polygons representing the surface of the volume. This can then be rendered with existing polygon rendering techniques to produce an image of the volume.

A common indirect volume rendering technique is the marching cubes algorithm, first presented by Lorensen and Cline [10] which creates a surface from the volume data. A cube is created from a total of eight neighbouring voxels which is then compared to the surface of the volume to calculate an index. This index is used to determine polygonal representation of the volume based on a pre-calculated table. The algorithm not only outputs the vertices of the polygonal representation of the volume, but also its normals. There are several cases where holes can appear in the final image, which was later improved upon by extensions to the algorithm such as in Kobbelt et al's research [11].

Kobbelt et al. [11] present an extended marching cubes algorithm and enhanced distance field representation which increases the quality of the final triangle mesh, resulting in an approximation error below 0.25%.

A common technique implemented for medical applications prior to the marching cubes algorithm was contour tracing. This approach consists of a four step process which involves:

- Take a 2D slice of the volume and traverse adjacent pixels to find closed contours, forming polylines.
- Based on iso values, identify structures.
- Connect contours from adjacent slices that represent the same structure, and use them to form a triangle
- Finally, render the created triangles.

There are many algorithms to implement contour tracing, such as that proposed by Yan and Min [12], which could be adapted for use with volumetric data.

2.1.2 Direct Volume Rendering

Direct volume rendering renders every voxel in a volume directly without transforming the volume to a polygonal representation or other domain. This section will discuss common volume rendering techniques such as shear-warp and splatting, with particular attention given to ray casting as it is most relevant to this research.

Ray Casting

Ray casting is a common technique for volume rendering which has been used for many years [13]. With this technique, rays are created for each pixel and traced from the camera (or eye point) into the volume (see Figure 2.1). The volume is sampled at discrete positions along each ray and, using a transfer function, the data obtained at each position is mapped to colour values. These values are accumulated to form the final colour for the pixel. Ray casting can be costly and achieving interactive rates for large data-sets was impossible for many years as it was only performed on the CPU.

The graphics processing unit (GPU) has became very powerful with advanced functionality such as dynamic looping in recent years. With this new powerful processing unit, it has become possible to implement ray casting on the GPU to achieve interactive rates [1].

Kruger and Westermann [5] exploit the features of the GPU, such as the ability to render to texture targets, per-fragment texture fetch operations and texture coordinate generation, to render volume data via ray casting on the GPU with improved performance. They apply acceleration techniques to further increase performance, such as early ray termination, where the ray is terminated once an opacity threshold of the accumulated pixel value has been met or a selected iso value is reached. They also implement empty-space skipping, where regions are checked to determine if they are empty while sampling and skipped if that is the case.

Extending on from this, Scharsach [14] implement a process called hitpoint refinement. Once the opacity threshold has been exceeded, the ray is stepped backwards up to 6 times to find the position where the threshold is met. This provides up to 64 times more accurate results than the original estimation. Also introduced is interleaved sampling and the ability to fly-through the rendered volume, which was previously not possible [14].

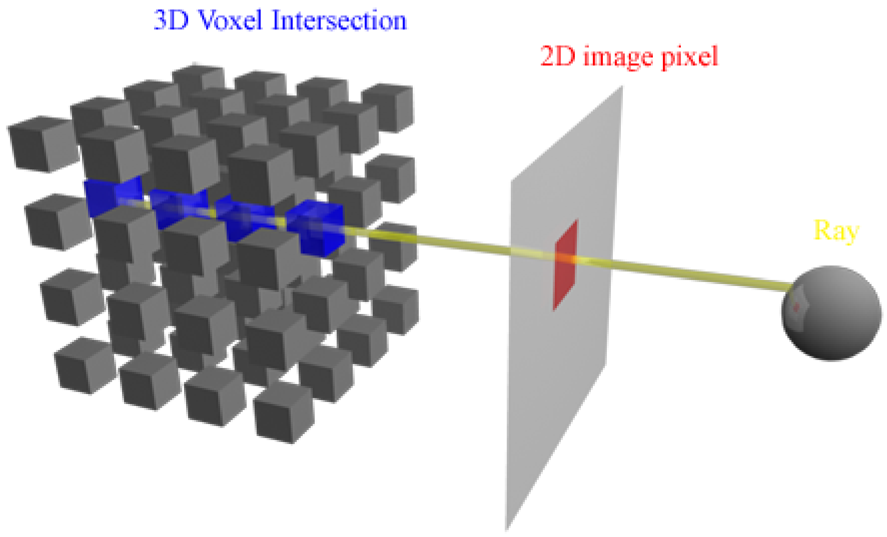


Figure 2.1: Ray casting volumetric data [2].

Spatial Sub-Division During volume rendering, a large portion of the volume does not contribute to the final image, as a result of occlusion or due to being outside the camera's field of view. While using ray casting, ray intersection tests are required to determine which parts of the volume to render. The time taken to do these intersection tests can be reduced by spatial sub-division.

Spatial sub-division divides the volume space into sections, which can be further sub-divided for more detail (see Figure 2.2). Each section is represented as a bounding volume. These can be sub-divided to 8 equally sized bounding volumes, which combined equal the size of the parent section. As a result, if a ray does not intersect the parent section, then it cannot intersect the children of that section. This reduces the number of ray intersection test performed, and therefore increases performance of volume rendering. This can be achieved by using a tree data structure such as binary-space partitioning trees and octrees, however discussion will be limited to octrees for this section.

Gobbetti et al. [15] note that when rendering, the entire volume does not need to be in memory. They organize the data in a spatial sub-divided octree, where empty

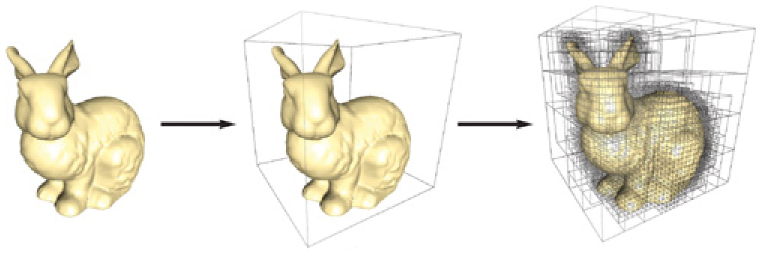


Figure 2.2: Subdividing a model into an octree [3].

space can be skipped while ray casting. The octree contains nodes which point to bricks, which are voxel grids of a set size M^3, or indicate empty space. All the bricks are grouped in a 3D texture, referred to as the brick pool, on the GPU. At all times, it is ensured that all leaf nodes visible from the current view point are in the brick pool. The CPU also has the same data structure which is used to perform modifications to the data structure and these changes are then sent to the GPU. The CPU streams any required data that is missing to the GPU. When rendering, rays are traversed through the octree using an algorithm similar to kd-restart [16], which is normally implemented for real-time ray tracing.

Crassin et al. [8] extend the work done by Gobbetti to achieve better quality and performance. To improve on the previous work, they reduce the number of pointers to one pointer per node in the octree by storing all nodes in a 3D texture, the node pool. This is organized into blocks of nodes and allows for all children of a node to be stored in one of these blocks. By using the 3D texture, it takes advantage of texture caching on the GPU. Crassin et al. [8] also determine the correct level of detail in the shader while performing ray traversal, and do not need a data structure unlike Gobbetti et al. [15].

A similar approach to Crassin et al. [8] is presented by Laine and Karras [9] [17]. They introduce a compact data structure for storing voxels, including geometry and shading data for each voxel. The data is divided into contiguous areas of memory called blocks. For each block, information on the topology of the octree, the geometrical shape of the voxel and shading properties of the voxel is stored. When a ray hits geometry, a shader is executed based on the shading properties stored in the geometry's respective block.

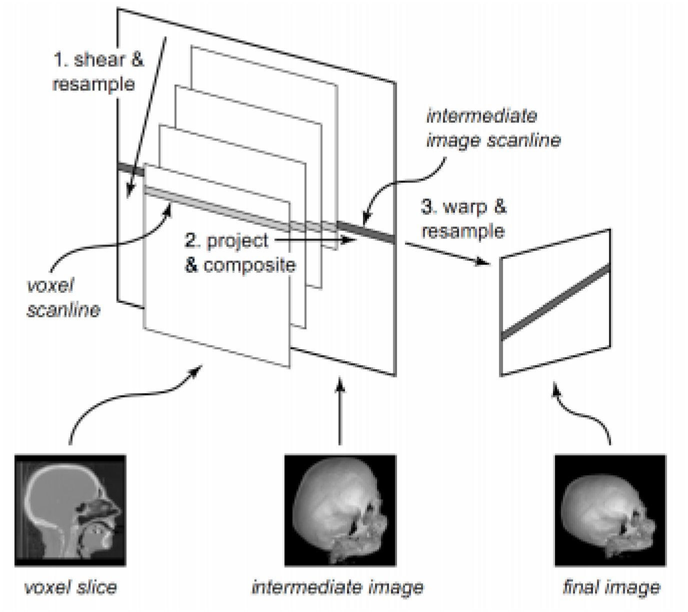


Figure 2.3: The steps involved in the Shear-Warp volume rendering technique are shear, project and warp [4].

The result is efficient sparse voxel octrees capable of voxel representation that are competitive with triangle based representations. While not as accurate (such as rendering corners and edges), it contains unique colour and normal information for each voxel, allowing unique geometry to be represented at high resolutions.

Due to the ever increasing power of the GPU, and the simple yet flexible technique that is ray casting on the GPU, ray casting will be a popular technique for the foreseeable future in volume rendering.

Shear-Warp

Shear-warp is a popular algorithm first introduced by Cameron and Undrill [18] and later by Lacroute and Levoy [4] to produce various views of a volume at interactive rates. In this algorithm, the volume is sliced, or sheared at multiple positions (see Figure 2.3). These samples remain parallel to each other, however their relative positions change. Using this approach, if one was to look directly at the volume (where only the front of the volume was visible), after the shearing process, the side of the volume would also be visible. The samples are projected onto an intermediate image. Due to random sample positions during shearing, the intermediate image after projection can be distorted, and so this is warped to produce the correct image. Each voxel position is transformed onto the projection plane by a simple rotation and projection.

Lacroute and Levoy [4] note that this technique is "more than five times faster" than a raycaster for datasets of size 128^3, and "more than ten times faster" for datasets of size 256^3. However the image results are of lesser quality than that produced by ray casting and incorporating zooming with acceptable results is problematic. This algorithm is also slower if large parts of the volume contain voxels of low opacity.

Wu et al. [19] present a hybrid approach of both shear-wrap and ray casting. The volume is sliced just like in the shear-wrap approach. Then rays are cast through the image plane and samples taken at the intersection with each slice. The approach does not warp the image, unlike the shear-warp method. Wu et al. note that this method requires the image to be separated into a number of partitions to avoid rays that diverge and become almost parallel to the slices of the image. This effects the number of passes required for rendering, one pass per partition.

Splatting

Splatting is a technique introduced by Westover [20] to perform fast volume rendering, at the cost of some quality. With this technique, the volume elements are projected onto the viewing plane, which Westover describes as "like a snowball" [20], which are integrated into 2D images called footprints. The projection is approximated by a Gaussian splat, based on the colour and opacity of the voxel. The footprints are then composited on top of each other in a back-to-front manner based on the camera orientation to produce the final image.

This is a fast technique for volume rendering, and is capable of running on low end graphics cards. However, the quality of the final rendered image is lower than that produced by ray casting, and results in fuzzy, blurry images.

Further improvements to splatting were proposed by Laur and Hanrahan [21], who implement a splatting algorithm based on a pyramidal representation of the volume. This differs from Westover as a set of footprints are built for each level of the pyramid. Recently more work has been done on splatting by McDonnell et al. [22], who present

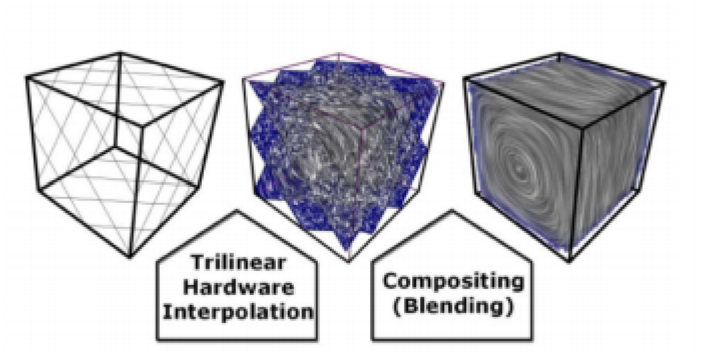


Figure 2.4: Three dimensional textures [5].

subdivision volume splatting. They "demonstrate that splatting in particular is the ideal subdivision volume visualization algorithm".

Three Dimensional Textures

Three dimensional texture based volume rendering is another method for volume rendering which was introduced by Cabral et al. [23] and Wilson et al. [24]. Three dimensional texture based volume rendering is where 3D textures are slices of a texture block which have been sliced in a back-to-front order with planes oriented parallel to the viewing plane [5]. For each slice, the 3D texture is sampled and the result blended with the pixel colour in the colour buffer [5] (see Figure 2.4). It is possible to render the slices orthogonal to the viewing plane due to the trilinear interpolation that is available on the GPU.

Westermann and Ertl [25] extend volume rendering with 3D textures by exploiting pixel textures for volume rendering in spherical domains, possible due to the pixel texgen extension in OpenGL.

Three dimensional texture based volume rendering produces images of reasonable quality at interactive rates.

Direct Volume Rendering Recap

This section has discussed some of the direct volume rendering techniques that are commonly used. There are a number of other volume rendering techniques such as wavelet volume rendering [26] [27] and fourier volume rendering [28] [29] which have not been discussed in detail however.

2.2 Animated Volume Rendering

Visualizing animated volume data is an important ability for scientists to study time-varying phenomena. Another application for it has arisen in recent years in computer and video games, with the number of games utilizing voxels increasing. Visualizing animated, or time-varying, volume data is problematic due to the large dataset that is associated with it. The volume data itself can be quite large, such as in static volume data, however this needs to be repeated for each time-step of the animation, which can be thousands of time-steps [30]. There has been a lot of research dedicated to this field, with a number of techniques presented to achieve interactive visualization of time-varying volume data.

2.2.1 Encoding

Encoding is one such technique which reduces the volume dataset size through compression, therefore reducing the amount of memory required to store the dataset, making it possible in some cases to store the data as a texture on the GPU, and also allowing it to be stored in main memory [30].

One implementation of encoding is to separate the time dimension from the spatial dimensions by extracting the differential information between sequential time steps [31], this technique also incorporates ray casting. Shen and Johnson [31] note that this technique has limitations if the changes in elements exceed over 50%. Additionally, the data must be browsed from the first time step [30]. Ma and Shen [32] use scalar quantization with an octree and difference encoding to achieve compression. They exploit the spatial and temporal coherence to fuse neighbouring voxels into macro voxels if they are similar. They also merge two subtrees if they are consecutive and identical to reduce space and increase rendering speeds. They use a ray casting volume renderer to render the first tree, and then only render modified trees in subsequent time-steps. Sohn et al. [33] use a block-based wavelet transform, which is also used for image compression, to compress isosurfaces and volumetric features while maintaining minimal image degradation. The volume is decomposed into small blocks which are only then encoded. To reduce space, only significant features of the volume are encoded.

Another approach to encoding is to treat the data as 4D data with the use of an extended octree, 4D tree, where time is the fourth dimension [30]. Linsen et al. [34] introduced $\sqrt[4]{2}$ subdivision with quadrilinear B-spline wavelets. Ma [30] notes that the characteristics of the data should determine if we should treat the data as 4D data. If the resolutions of the temporal and spatial differ greatly, they should be then be separated due to the difficulty in locating temporal coherence [35].

Shen et al. [35] introduce a time-space partitioning (TSP) tree for temporal and spatial coherence, allowing the user to adjust error tolerances for trade off in image quality and rendering speed. The TSP tree is similar to an octree, however each node in the tree is a binary tree which represents the difference time span for that voxel, thus reducing the amount of data required for rendering.

2.2.2 Transfer Functions

A transfer function assigns RGBA values to voxels to determine how they are represented visually, and so is a major aspect of rendering the volume. They can be used for both static and animated volume rendering, however, there is no single accepted transfer function for time-variant data, with a number of various approaches proposed [36].

Ma [30] notes that for time-varying volume data, the transfer function needs to identify features of regular (feature that moves steadily through the volume), periodic (features that appear and disappear over time) or random patterns in the time-varying data.

Jankun-Kelly and Ma [37] discuss various ways of generating transfer functions, as well as studying when multiple transfer functions are needed including how to capture the time-varying data in as few transfer functions as possible. They find that time-varying volume data that have a regular structure can be rendered with a single transfer function.

2.2.3 Animated Sparse Voxel Octrees

While sparse voxel octrees are suitable for efficient volume rendering of static objects [9], the structure of the octree makes them unsuitable for animation. Each node in the octree does not contain position or orientation information, instead this information is determined by the position of the node in the octree. This results in a lack of an organized hierarchy which can be exploited for animation (such as toes connected to a foot, which is connected to a leg etc.). Therefore, animating a model would require an octree for each frame of the animation. This approach results in large amount of data, well beyond current hardware memory limitations.

Bautembach [38] introduces animated sparse voxel octrees and presents examples of rotation and skinning. Rotating voxel octrees are achieved by traversing the octree to the leaf level, computing every nodes minimum and maximum vectors on the fly based on the nodes current position within the octree. A rotation matrix is then applied to every leaf node by determining the node's centre position and rotating it. This approach introduces visual errors, but these are negligible given a high sampling rate. Bautembach notes that this approach can be applied to every rigid and non-rigid transformation, however different procedures are required for different transformations in order to maintain the model's proportions. An example of this is provided where a simple formula is used to adapt the size of the voxels in order to prevent holes in a skinned model animation. To create the animated voxel octree, Bautembach, D. [38] takes an animated model, in triangle mesh form, from another standard storage format, and creates a compressed octree upon initialization. The Nodes in the octree are stored sequentially as if they were traversed by breadth-first search. For each frame of the animated model, another octree is created based on the original.

While this approach achieves animated voxels, it is only suitable for pre-recorded animations, and so rag-doll animations, for example, would not be possible.

2.3 Octrees

So far, discussion of octrees has been related to volume rendering, however octrees are used in other fields such as collision detection, and as a result has received heavy research outside of the volume rendering domain. This research is still very relevant for volume rendering.

Octrees have traditionally been implemented on the CPU, however the GPU has become a powerful processing unit, and is now used extensively for rendering, such as the previously mentioned GPU ray casting techniques. However, if rendering is now taking place on the GPU, the octree on the CPU needs to be accessed by the GPU if it is to be integrated with the renderer.

Pharr and Fernando [3] store an octree created on the CPU in a 3D texture called the indirection pool. The indirection pool is subdivided into indirection grids, which are cubes of 2 x 2 x 2 pixels (referred to as cells). Each indirection grid represents a node in the octree. The cells in each indirection grid represent the nodes children in the octree (see Figure 2.5). The RGB values of the cell can contain nothing if the corresponding octree node is empty or it can contain:

- data if the child node is a leaf
- or the index position of another indirection grid if the child node is not a leaf node.

The alpha value of the cell indicates what type of data is stored in the RGB component of that cell. An alpha value of 0 indicates the cell is empty, a value of 0.5 indicates the cell contains an index position, and a value of 1 indicates data is stored. This texture is then uploaded to the GPU and stored in texture memory, for access by the fragment shader. To find if a point is within the tree, a lookup function is called which starts at the root node (at texture coordinate [0, 0, 0]), and successively visits all nodes that contain the point until a leaf node is found.

Madeira and Lewiner [39] present an octree search structure for the GPU based on a hash table. To take advantage of the parallel structure of the GPU, their search algorithm treats all search locations and octree depths independently of each other. They also introduce a parallel version of their optimized search algorithm, allowing for efficient searching of a single point at any time in the octree.

Ganovelli et al. [40] introduce a bucketTree where the leaves of an octree data structure contain primitives inside the corresponding box, which is referred to as a bucket. The primitives stored in the buckets can be vertices, polygons or even volume data. This is made possible as the bucketTree views the object as a soup of primitives.

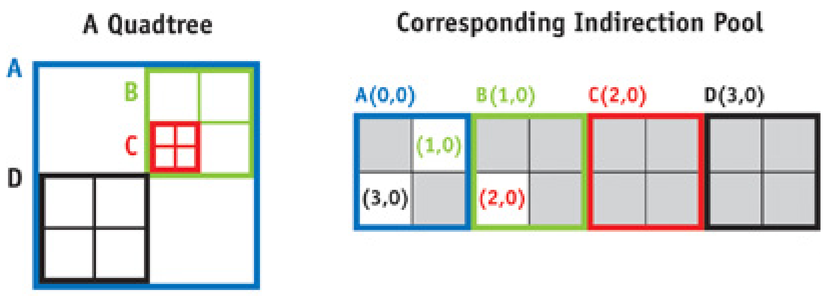


Figure 2.5: Storing an octree generated on the CPU in a 3D texture for use on the GPU, using a quadtree (2D equivalent of an octree) and a 2D texture for illustrative purposes [3].

At every time step of a simulation or animation, the primitives are assigned to the correct buckets.

Gu and Wei [41] implement octree ray casting on multi-core CPUs to achieve parallel optimization of an octree. The octree is optimized to exclude redundant data and to skip empty voxels to reduce storage and computation. Computation is further reduced with the implementation of the octree with multi-core CPUs, reducing the computation time associated with the octree and ray casting.

Chapter 3

Design

The main research areas of this dissertation can be divided into two areas; the first is volume rendering at interactive rates; the second research area is progressive octree generation at interactive rates. The goal of this dissertation is to create an application that implements both these research areas, and to produce technical results which can be evaluated, with particular attention given to progressive octree generation, in terms of performance and applicability to interactive entertainment technology.

3.1 Development Plan

The development process plan for this dissertation is divided into the following:

- Research existing volume rendering techniques, animated volume rendering and octree generation.
- Design an application to incorporate volume rendering and octree generation (including progressive octree generation) with a user friendly interface.
- Implement the application, using cross-platform compatible code where possible so the application can be used on multiple operating systems, based on the design outlined in the previous step.
- Using the application, evaluate the implemented volume renderer and octree generation.

Research into existing volume rendering techniques, animated volume rendering and octree generation has been discussed in depth in Chapter 2, with the knowledge gained in this research applied while designing the application. The application is divided into the following components:

- File management component to load volume data from external files and also store data to external files.
- Volume renderer component to display the volume to the screen.
- Octree component used to generate octrees. The octree will also be incorporated with the volume renderer.
- A user friendly interface.

The following sections will discuss the design of each of these components.

3.2 File Manager

The File Manager component is responsible for loading volume data into the application from external files. The application must be capable of reading volume data, regardless of the volume data dimensions and spacing. The application should be capable of loading 8-bit and 16-bit volumetric datasets. The datasets will be loaded from binary files, or .RAW files, which are a common storage format for volume datasets and are freely available on websites such www.volvis.org. The loaded data should then be stored in an array, which can be used to create a texture.

Furthermore, the file manager will be responsible for writing data to external files, in particular octree information for specific volume datasets. Subdividing a volume with an octree can take a number of seconds to complete, depending on the depth of the tree and the resolution of the volume. This subdivision will occur every time a new volume is loaded into the system. Instead of generating the same octree for a volume every time it is loaded, the octree should be stored to an external file. This file can be loaded into the system when loading new volume data, reducing the loading time of the application. This also allows octrees for pre-recorded animated volumes to be loaded into the application. It is important to note that the octree will only be loaded from a file when a new volume is loaded and initialized, it will play no part in the progressive generation process.

3.3 Volume Renderer

The Volume Renderer is responsible for displaying the volume data-set on screen. This involves implementing a volume rendering technique, many of which have been discussed in detail in Chapter 2. For this dissertation and research project, ray casting has been chosen. There are numerous reasons as to why ray casting was chosen over other techniques.

The fundamental reason for choosing ray casting is that the technique allows for optimum usage of spatial sub-division hierarchies, such as octrees, which can be incorporated into the raycaster without much code adaptation. Since octrees are a core research topic for this dissertation, this rendering technique is an appropriate choice.

An additional reason for using ray casting is it provides the best balance in terms of image quality and processing speeds. This was not always the case however. Traditionally, to perform ray casting at interactive rates required top of the range CPU's. An example of this is Outcast®, a computer game released in 1999 by Appeal and Infogrames, which used ray casting to render the environments. This provided impressive visuals, but a top of the range CPU was necessary to be capable of running the game at interactive frame rates. With the introduction of the GPU, and it's ever increasing processing power, it is now possible to perform ray casting on the GPU, with significant speed boosts over the CPU.

3.3.1 GPU Ray Casting

Ray casting will be performed on the GPU to achieve the best possible frame rates. This will require the volume data to be loaded into a three dimensional texture and passed to the GPU. Furthermore, the transfer function will also need to be passed to the GPU as a texture so the colour information can be mapped to the volume data in the fragment shader. Additionally, the volume should be rendered with diffuse shading and with a simple ambient light. More complex lighting algorithms could be implemented, such as Phong [42] or Blinn-Phong [43], however this is beyond the scope of this dissertation.

The volume renderer should be capable of extracting iso surfaces from the volume also. This is a common feature of volume rendering used to analyse three dimensional data in greater detail; for example removing the skin layer of a volume representing a human head, so the skull is revealed for a clearer inspection.

3.3.2 CPU Ray Casting

A CPU raycaster will also be implemented as a measure against the GPU raycaster, demonstrating the difference in both performance and image quality between both implementations. The CPU raycaster will incorporate the CPU generated octree to reduce rendering costs.

3.3.3 Transfer Function

Transfer functions map colour information to volumes, as volumetric datasets do not store any colour information. These transfer functions can be complex, implementing mathematical algorithms such as the Fourier transform to produce sophisticated functions, in particular to volumes relating to velocity fields and fluids. However, such implementations are beyond the scope of this dissertation, where the application of a transfer function is to simply render volumes with colour. As a result, this application will apply simple one dimensional transfer functions, which are saved to a one dimensional texture for use on the GPU.

3.3.4 Pre-Recorded Volume Animation

The application should be capable of rendering pre-recorded volume animations loaded in from an external file, enabling the application to support both forms of volume animation, pre-recorded and physically based or user-defined deformations. These animations are simply a set of volumes, rendered one after another sequentially. Rendering animated volumes, that is pre-recorded animated volumes, is an area that is currently heavily researched with a number of approaches being proposed and investigated, such as compression of the animated data [44], to aid in rendering animated volumes at interactive rates.

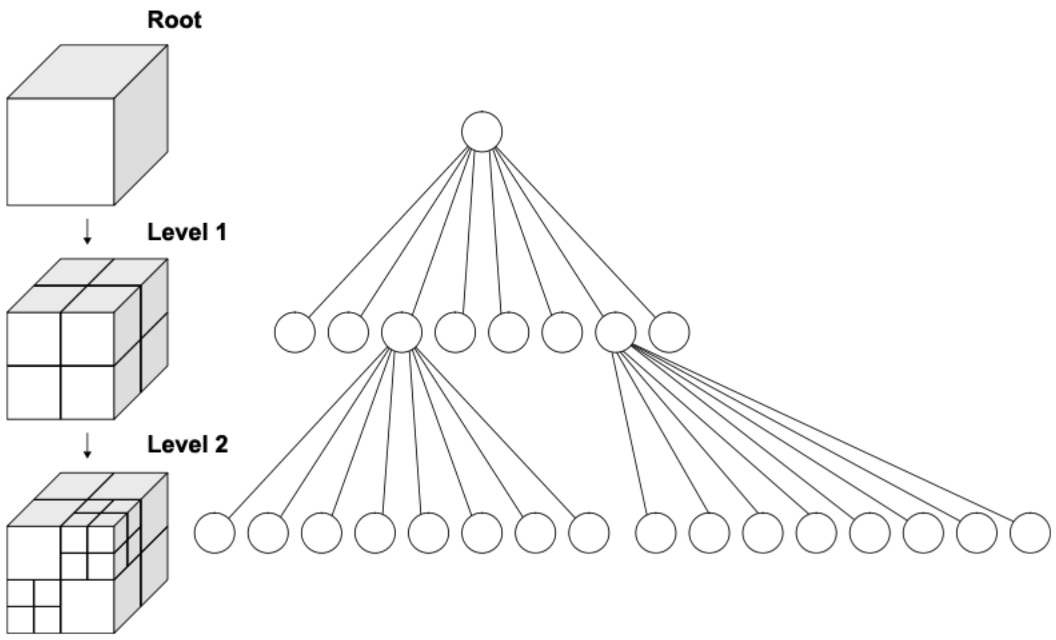


Figure 3.1: Subdividing data into an octree to a depth of level 2 [6].

It is important to note that this dissertation is focussed on using octrees for user-defined deformations or physically based animations, and not on pre-recorded animated volume rendering. With this in mind, this application will be capable of only rendering small animated volumes, both in volume resolution and animation length. Large animated volumes, such as the example presented in section 1.2 (page 4) of a 3 second animated volume of resolution 512^3, will not be possible due to the large amount of data associated with such datasets which require optimizations for rendering at interactive rates. The implementation of these optimizations is not possible for this application due to the time constraints opposed on this dissertation.

3.4 Octree

The octree component will be responsible for subdividing volume data into an octree on the CPU. An octree is a spatial sub-division hierarchy that sub-divides the volume data into nodes. A node represents a part of the volume, storing a value which represents the average of all the voxels within that area of the volume. This node can be further subdivided into 8 more nodes (hence the name *oct*ree), which represents a smaller area of the volume, and therefore stores more accurate information about the volume as there are less voxels within the node (the value stored in a parent node represents the average of all the child nodes)(see Figure 3.1).

In this application, the octree will be a pointer-based octree, that is each node in the octree has 8 pointers which point to its child nodes. Alternatively, a node's child pointers can be empty if that node is not subdivided. Each node in the octree should store as little data as possible. This data includes:

- The 8 pointers to the child nodes.
- The three dimensional position of the nodes centre.
- The nodes width.
- A boolean flag to indicate if the node is subdivided (i.e. contains children) or not.
- An average of the volumes voxel data that is located inside the node.

3.4.1 Octree Generation

Octree generation involves creating a root node that encompasses the volume, and then subsequently subdividing the node into eight new nodes, which are also subdivided and so on. This subdivision should continue until a desired tree level is reached or until each node presents only one voxel. To prevent unnecessary processing and memory usage, empty voxels are ignored, resulting in nodes being created only if all the voxels within the node are not empty. Further optimizations are applied to the octree generation process to reduce computation time. These optimizations are as follows:

- For a node to be created, the voxels it encloses not only have to be non-empty, the average of all the voxels must be above a certain threshold. This will prevent nodes being created for areas of the volume that have a very low opacity and do not contribute to the final rendering, saving both computation and memory.

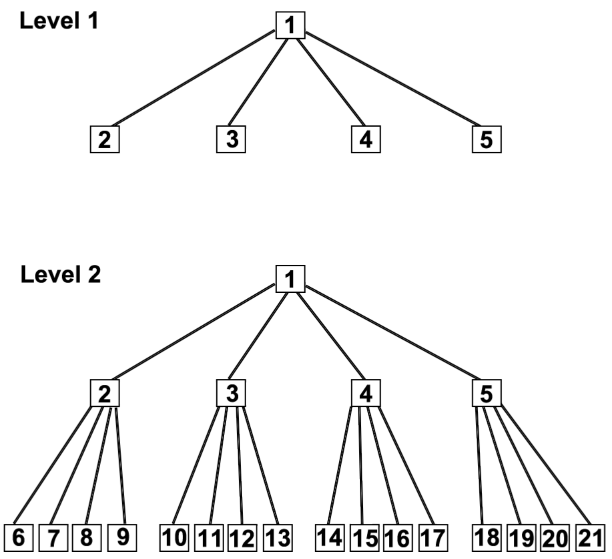


Figure 3.2: Subdividing a quad-tree (two dimensional version of an octree) using breadth-first approach. The numbers on the nodes indicate the order in the nodes are subdivided.

- When subdividing a node, a check will be made to see if the average voxel values for the soon to be created child nodes are the same. If this is the case, this means that the average value of each of the child nodes, that are to be created, will be the same as the parent node. As a result, the node will not be subdivided as the children of that node would not contain any more detailed information that the current node already provides.

Two different approaches will be implemented to generate the octree, breadth-first and depth-first. This will allow for a comparison between the performance of the two approaches, determining which is the faster approach, if any, for the generation of octrees at interactive rates.

Breadth-First

Generating an octree with breadth-first results in the octree been divided level by level, that is level 2 of the octree is not subdivided until level 1 is fully subdivided first (see Figure 3.2).

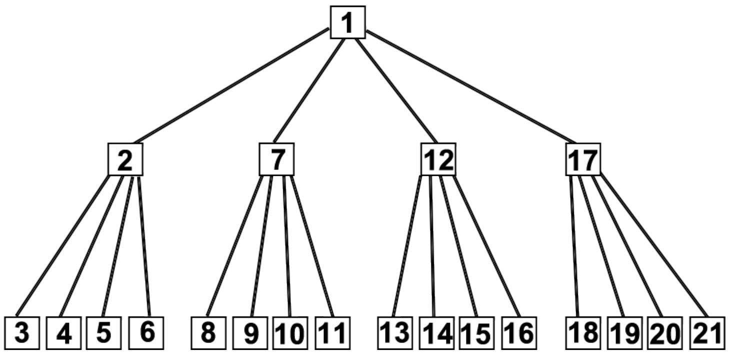


Figure 3.3: Subdividing a quad-tree (two dimensional version of an octree) using depth-first approach. The numbers on the nodes indicate the order in which these nodes are subdivided.

Furthermore, two implementations of this approach will be used, the first a recursive implementation, the second a non-recursive implementation. The decision to include two implementations of this approach is due to the desire to achieve the best performance possible when generating the octree. Recursive implementations are effective, but they have overheads associated with stack management. By performing both implementations, they can be evaluated to determine which, if any, provides the best performance.

Depth-First

Depth-first subdivides one node as far as possible before backtracking to subdivide the rest (see Figure 3.3). How far a node is subdivided is based on the maximum allowed depth of the octree, the number of voxels a node represents (if the current number is 1, further subdivision is unnecessary) or if the average voxel value is below the previously mentioned threshold. As a result, one side of the octree is completed before moving onto the next. This approach is not suitable for progressive octrees, as levels are not fully completed before moving onto the next, meaning the octree cannot be drawn as it is subdivided as only parts of each level will be visible. However, it is useful to determine which approach is faster, breadth-first or depth-first.

3.4.2 Progressive Octree Generation

Progressive octree generation should generate an octree during run-time while maintaining interactive rates. This should take advantage of the previously mentioned octree generation implementations. Three forms of progressive octree generation will be implemented. These are:

- Progressive octree generation on the application's main thread.
- Implementing a dedicated thread for the generation process.
- Implementing multiple dedicated threads.

The progressive octree generation will incorporate the breadth-first approach to generating the octree. This will allow for the octree to be drawn while it is still being subdivided. For example, if level 5 is currently being subdivided to create level 6, with breadth-first this means that levels 1, 2, 3, 4 and 5 are all completed, and so can be displayed. This will be important, as generating the octree will potentially take seconds (for large volumes), and instead of drawing nothing while the octree is created, it is better to show feedback to the user, demonstrating the current progress.

Main Application Thread

The application's main thread is responsible for everything else, such as rendering and updating application states. It is important that the octree generation does not cause the main thread to slow down. As a result, the progressive octree generation should be the last operation performed on the main thread after the draw call. This will allow for an approximation of how much time was spent on the current frame, and how much time can be spent generating the octree before the frame rate drops below the desired level. Once the progressive octree generation has exceeded this time limit, it should stop the generation and resume in the next frame.

For example, if the desired frame rate of the application is 60 frames per second, that means each frame can take a maximum of 16.6666 milliseconds to complete. If one frame has performed all updates and draw calls within 10 milliseconds, this means there is 6.6666 milliseconds left to perform operations if needed, in this case to progressively generate an octree. When generating the octree, a timer will be used to determine how much time has been spent so far. Once this time has exceeded the allocated time, 6.6666 milliseconds in this case, it should stop the generation and resume in the next frame.

This requires the ability to stop the generation of the octree, storing the current stage of the generation process and resuming from the same point the next time it is called.

Single Dedicated Thread

A dedicated thread will be created for the sole purpose of generating the octree. This will allow for the progressive octree generation process to be abstracted from the main application's thread, allowing the dedicated thread to utilize all it's available resources to generate the octree as quickly as possible.

Multiple Dedicated Threads

Multiple dedicated threads will also be implemented for generating the octree, 9 threads to be precise. The first thread will be used to initialize the octree, copying volume data, setting the dimensions and creating the root node of the tree. This thread will also be responsible for subdividing the root node, creating potentially eight child nodes and setting the octree depth level to 1. This thread will then create a new thread for each of the root node's children which are not empty. These threads will subdivide their assigned node, using breadth-first traversal, down to the desired maximum tree level or until further subdivision is not possible. This will result in potentially 8 threads working separately to subdivide the same octree to the desired level, without interrupting the main application.

It is important that these threads are loosely coupled, and do not depend on accessing shared information. Otherwise, locks will be required to prevent multiple threads from accessing the same data at the same time. These locks, while very important for concurrency control, can reduce the efficiency of threads.

3.4.3 Octree Rotation

The user will have the ability to rotate the volume, which will invalidate the octree for that volume. Instead of rebuilding the octree, the octree will be adapted by rotating it, based on the implementation described by [38], where the center of each node in the tree is rotated about the origin, in a similar to how the volume is rotated.

3.4.4 Visualizing the Octree

The application should be capable of visualizing an octree. This will be achieved by rendering 12 lines to create a wire-frame for each node. When evaluating the performance of octree generation, the octree visualization implementation will be disabled to get a true performance measure.

3.5 Volume Deformation

The focus of this dissertation is progressively generating octrees for volume deformations or physically based volume animations. To demonstrate how applicable progressive octree generation is, it will be important to allow the user to modify the volume data on the fly. Therefore, the application must be capable of allowing the user to adjust the volume data that is used for generating the octree. A simple approach will be undertaken to achieve this. The user will be capable of setting what percentage of the volume data should be used for creating the octree. A clipping plane will be used to clip the appropriate area of the volume based on a user defined percentage value. For demonstrative purposes, a single clipping plane will be used to clip the volume along the x axis. For example, with a volume of size of 64^3, if the user determines to clip 50% of this volume data, that will result in the octree been generated for voxels between [0, 0, 0] and [32, 64, 64] of the volume. This is a simple approach for user-defined deformations, but is suitable for testing progressive octree generation.

As previously mentioned, the user will be capable of rotating the volume, invalidating the current octree. This is another simple form of volume modification that is appropriate for this application.

3.6 User Friendly Interface

The application will provide the user with a user friendly interface, to allow for modifications to system variables and adjustments to the display settings. The user interface should allow the user to:

- Generate an octree at the press of a button.
- Set the depth at which the octree should be subdivided to.
- Determine which approach to generate the octree with, breadth-first or depth-first.
- Determine whether the octree should be generated on the application's main thread, a single thread or using multiple threads.
- Switch octree visualization on or off, and set which level of the octree to visualize.
- Enable and disable isosurface extraction, and setting which isosurfaces to extract from the volume.
- Set what percentage of the volume should be used when generating the octree (as part of the user volume modification system). Adjusting this value will automatically re-generate the octree.
- Enable and disable rotation of the volume.

3.7 Software Development Methodology

The project will adopt the Agile development process called SCRUM [45]. This development methodology is effective for quickly developing and testing applications, allowing for changes and fixes to be applied to the application quickly and easily with ease. Weekly sprints are used to designate tasks to be completed within the time allocated for this dissertation.

Chapter 4

Implementation

This chapter will discuss the implementation of the application based on the system design set out in the previous chapter. The implementation of each component is discussed, followed by a brief discussion on the external libraries used in the application.

4.1 File Manager

The volume data is stored in the binary format as a .RAW file. Loading the data into the application involves reading the binary data and storing it in an array. This array is then used to create a three dimensional texture of the volume for use with the GPU raycaster. The array type is determined by the volume data. If the volume data is 8 bit, then the data in the array is of type GLubyte, whereas GLushort is required for 16 bit volume data.

Storing the octree to an external file involves creating two files. The first file is a header file with information regarding the octree, such as it's centre, dimensions and depth. The second file stores the actual octree. The approach used to store the octree to an external file is described by Pharr and Fernando [3].

4.2 Volume Renderer

Once the volume is loaded into the application, it can be integrated into both the GPU and CPU raycaster. A discussion regarding the implementation of these follows.

4.2.1 GPU Ray Casting

As the focus of this dissertation is on octree generation for volume rendering, and considering GPU ray casting is a well researched topic that is used frequently [1] [5] [14], it was decided that if possible, an existing implementation would be used as opposed to writing it from scratch. As a result, the GPU ray casting code is based on the tutorial by Hayward [46]. An additional fragment shader is created for isosurface extraction. This is to prevent the GPU from performing unnecessary checks to see if isosurface extraction is enabled within the raycaster, as the user has the option to enable or disable it.

As previously discussed, ray casting involves casting a ray for each pixel and tracing it from the camera into the volume, sampling the volume at discrete positions along the ray. To cast the ray into the volume, the ray needs to be defined first, that is, the ray's origin and direction need to be determined. To do this, the coordinates of the volume's bounding box need to be obtained. This can be achieved by rendering the front and back buffers, where the colours represent the coordinates of the bounding box, storing each buffer to a texture using multi-pass rendering. These textures are then passed to the GPU raycaster, along with the three dimensional volume texture, and used to create a ray for the current pixel. The ray's origin is the coordinates from the front buffer, while the direction of the ray is determined by subtracting the back buffer from the front buffer.

Samples of the volume data are then taken along the ray at discrete positions, based on the step size. A sample involves a texture look-up on the three dimensional volume texture, using the ray's 3D position as the look-up coordinates. For each sample, front to back compositing is performed to blend the colour value obtained from the current sample with the colour value obtained from the previous sample. Following this, the ray's position is advanced based on the step size passed to the fragment shader and the ray's direction. Once the ray's position is advanced, a check is performed to determine if the ray is outside the volume. If the ray is outside the volume, no more samples are taken and the current pixel colour is obtained from all previous samples and is returned from the fragment shader. Otherwise, the volume is sampled at the ray's new position.

Early ray termination is implemented by checking the opacity of the current pixel value after each sample. If the opacity is above a certain threshold, for example 0.95, then no more samples are taken and the colour obtained from all previous samples is returned from the fragment shader.

4.2.2 CPU Ray Casting

The CPU raycaster integrates the octree into the rendering progress. For each pixel, a ray is cast into the scene in the direction the camera is looking at. For each ray, a ray to cube intersection test, as described by de Bruijne [47], is performed with the root node of the octree. If the ray is found to intersect the root node, further intersection tests are performed with the children of the root node. This process is repeated until a leaf node is found. Using the average voxel value stored in each node, the transfer function is sampled to get the colour for that node, which is then stored in an array, and later rendered to the screen. Further optimizations to the CPU raycaster were beyond the scope of this dissertation, as were improvements to the rendering quality as the purpose of this implementation is to determine whether significant improvements in performance can be gained from using a GPU implementation.

4.2.3 Transfer Function

The transfer function implemented in this application is a simple 1D texture. As each voxel contains a scalar or density value that ranges from 0 to 255, the width of the texture is 256 allowing for a colour to be mapped to each possible scalar value. The transfer function can be created using any image editing software, however this application created the transfer function based on the implementation by Hayward [48] where cubic splines are used to give greater control over the function.

Integrating the transfer function into the raycaster involves performing a texture look-up on the transfer function. As the transfer function is a 1D texture, the texture is looked-up using the alpha component of the colour value (which represents the voxel's scalar or density values) obtained from sampling the volume data. This is then integrated with the compositing blending, and returned from the fragment shader, drawing it to the screen.

4.2.4 Pre-Recorded Volume Animation

Basic pre-recorded volume animation is achieved by loading in each frame of the animated volume into an array, where each frame is stored in a separate .RAW file. An octree is required for each frame of the animation. If an octree for each frame already exists in an external file, these files are loaded into the application and used to create the octrees for all the frames. Otherwise, each octree is generated during the preprocessing stage of the application when the volume data is loaded. The application updates the current frame of the animation and draws the appropriate volume that is associated with the current animation frame.

4.3 Octree

The octree is created on the CPU and consists of node objects. Each node contains as little information as possible to reduce memory usage. Therefore, a node only contains its width and centre position, a flag indicating if the node is subdivided or not, 8 pointers to child nodes which are by default set to zero, and finally the average value of all the voxels the node represents.

Due to the structure of the node object, that is each node has pointers to child nodes, the octree object is only required to store the root node. Storing other nodes in the octree class is not necessary, as they can be accessed via the root node. Also stored in the octree is the volume data in a three dimensional array. This data is required to subdivide each node. A possible implementation was to store the array of voxels that each node of the octree encloses within the node object. However by simply storing the voxels in the octree, each node in the tree can easily access the data. Furthermore, because the volume data is stored in a three dimensional array, traversing voxels only relevant to a node can be easily done based on the node's minimum and maximum coordinates, derived from the node's centre and width.

The root node of the octree is created based on the 3D coordinates of the volume's centre and the dimensions of the volume. The width of the node is set to the maximum volume dimension. For example, for a volume of size 128 x 64 x 256, the width of the node is set to 256, as the octree should enclose the entire volume while maintaining a cube shape (that is, the height, width and depth are all equal). The average voxel value for the root node is found by traversing the entire volume data and calculating the average value.

4.3.1 Subdividing Octree Node

To subdivide a node, the voxels within that node need to be traversed, as previously mentioned. Since the voxels are stored in a three dimensional array, traversing only the voxels within the node is a simple procedure. The minimum and maximum coordinates of the node is found based on the node's centre and width. These minimum and maximum coordinates are then used to loop through the three dimensional array, starting at the minimum coordinate and ending at the maximum coordinate.

Once the node's voxels can be accessed, each voxel is sampled, taking the voxel's density value and accessing the transfer function to get the alpha value associated with that voxel. The transfer function stores RGBA values to map colours to the density of a voxel, but, for subdividing the node, only the alpha value is required, the RGB components can be ignored. The minimum and maximum vectors for each of the 8, soon to be created, child nodes are calculated, and used during sampling to determine which child node the current voxel will be assigned to. While sampling the voxels and determining which child node they will belong to, the average alpha value is calculated for each child.

Once sampling is complete, two checks are performed to determine if the child nodes provide more detailed information of the volume than the parent node. The first check involves determining if the average alpha value for each of the child nodes are the same. If they are the same, this indicates that they have the same alpha value as the parent node and as a result no further detailed information will be obtained by subdividing the current node. Hence the parent node remains a leaf node and the child nodes are not created. Otherwise, if the alpha values are not the same, each of the child nodes are created if the average alpha value is above a specified threshold. This check ensures that nodes are only created for voxels that contribute to the final rendering, because if a node's average alpha value is below the threshold, assuming an appropriate threshold value is set, then the alpha is so low it is unlikely to contribute to the rendering of the volume and therefore can be ignored.

4.3.2 Breadth-First Octree Generation

The main technique used in this application to generate the octree is the breadth-first approach, as this allows for the octree to be drawn as it is generated for the progressive octree implementation. Breadth-first octree generation involves subdividing, using the subdivision approach previously discussed, the root node, creating potentially 8 new child nodes. These child nodes are then subdivided, creating new nodes which are subsequently subdivided. This results in the octree been built level by level.

Two implementations of breadth-first octree generation are implemented, a recursive and a non-recursive approach, as outlined in the application design to determine which, if any, provide the best performance for octree generation.

Recursive Breadth-First Traversal

The recursive implementation of breadth-first involves calling a subdivide function, passing the octree's root node. This function checks if the node is already subdivided. If it is not, the node is subdivided and the function returns. Otherwise, the subdivide function is called for each of the child nodes. This function is capable of subdividing one whole level of the tree. To completely subdivide the octree to the maximum level, a while loop is implemented where the subdivide function is called with the octree's root node. Once the max depth has been reach, or further subdivision is not possible, the while loop ends.

Non-Recursive Breadth-First Traversal

For the non-recursive implementation, a queue is used to determine the nodes that need to be subdivided next. The queue is created with the root node added initially. A while loop is used to continually subdivide the octree until the max depth is reached or no further subdivisions is possible. For each iteration of the while loop, the node at the front of the queue is popped. If this node is already subdivided, then it's children are pushed to the back of the queue. Otherwise, the node is subdivided and it's children are pushed to the back of the queue. This results in the octree's nodes being subdivided in the order they are created without recursively calling a function.

4.3.3 Depth-First Octree Generation

Another approach to generating an octree is by using the depth-first approach outlined in the design section. This is achieved by first subdividing the root node, potentially creating 8 new child nodes. The first child node is then subdivided, creating more child nodes, with the first of these nodes subsequently subdivided. This process continues until the max tree depth is reached. At this point, the process goes back up one level and subdivides the next child (see Figure 3.3 for the order of node subdivision). This process is repeated until all nodes are subdivided to the max level.

4.3.4 Progressive Octree Generation

As discussed in design, progressive octree generation is performed over three different implementations, using the main application thread, a dedicated thread and multiple dedicated threads. This application uses the boost library, discussed in section , to perform threading in C++.

Main Application Thread

The main application thread is responsible for running the application, updating states and rendering. As a result, when the thread is generating the octree, it cannot update states, respond to user input or render objects as it is waiting for the generation process is complete, the application essentially becomes unresponsive while generating the octree. To maintain interactive rates, it is therefore important to limit the amount of time spent generating the octree for each frame. A naive implementation would be to use a pre-set amount of time, where each frame spends a set amount of time generating the octree. However, a more adaptable approach would be to base it on the time spent on the current frame. The aim is to achieve 60 frames per second, a standard frame rate in computer and video games. This means each frame has approximately 16 milliseconds to perform all operations for that frame. By performing the octree generation at the end of every frame, it can be determined how much time has been spent so far for the current frame, and therefore how much time is available for generating the octree before the frame exceeds an execution time of 16 milliseconds. For example, if all operations for a frame took 10 milliseconds, then 6 milliseconds are available for generating the octree.

Octree generation is performed with the breadth-first technique using recursion. As the octree is generated over multiple frames, it is required to interrupt the generation process and resume it during the next frame. In order to achieve this, a queue is implemented in the octree, and used in a similar approach to that outlined in section 4.3.2, non-recursive breadth-first traversal. The queue maintains the list of nodes that need to be traversed. The node at the top of the queue is popped and removed from the queue. If the popped node is a leaf node, it is subdivided, otherwise the node's children is traversed. If each child is a leaf node, they are subdivided, otherwise each child is pushed onto the back of the queue. This ensures that the queue has a list of nodes that are required to be traversed before the octree is fully generated. After a node is popped from the queue and the steps just outlined are performed, a check is made to ensure that the time spent subdividing the node has not gone over the allocated time. If it has, the generation process is stopped, otherwise the process continues with another node popped off the front of the queue. When the queue is empty, this indicates that a level has been fully generated. If further subdivisions is possible, the rootnode is pushed into the queue and the process repeated.

By using the queue, the generation process can resume in the next frame by popping the node at the front of the queue. However, a disadvantage is that the generation process cannot be interrupted during the subdivision of a node, only before or after a node is subdivided. As the subdivision process is the most time consuming part, this can result in the generation process over-running it's allocated time while subdividing a node, affecting the frame rate of the application. This can be taken into account by subtracting a set amount of time from that allocated to the octree generation.

Using this approach, the octree is generated over several frames. A state is kept to maintain the current depth of the tree, using it to draw the octree while the current level is subdivided to create the next level.

Single Dedicated Thread

Another approach is to use a single dedicated thread to generate the octree. This allows the main thread to continue updating and rendering the application while the dedicated thread is generating the octree, using breath-first. As the octree is generated on a separate thread from the main application thread, no time limitation is associated with the generation process, the thread continually subdivides the octree until it is finished. Once the generation of the octree is complete, the thread finishes executing.

Multiple Dedicated Threads

The multiple thread implementation separates the generation process across a number of dedicated threads, in this case up to 9 threads. The first thread creates the root node and subdivides it, creating potentially 8 new nodes. Then it creates a new thread for each of the child nodes. These threads are assigned to subdivide the specified node to the desired depth or until no further subdivisions are possible using the breadth-first technique. As a result, each thread is responsible for subdividing one octant of the octree.

When using multiple threads, it is important to ensure that no two threads try to access an object at the same time, as this can result in concurrency issues. With this implementation, these concurrency issues are avoided as each thread is assigned to only subdivide one octant of the tree, and as a result a node will only ever be accessed by one thread.

4.3.5 Octree Rotation

Rotating the octree involves rotating the centre of each node around the origin. Despite rotating the octree, the nodes of it are still axis aligned (see Figure 4.1). However this introduces errors into the octree representation of the volume (see Figure 4.1). These errors include gaps between nodes and nodes overlapping each other. The gaps between the nodes will cause holes and an inaccurate representation of the volume when the octree is rendered. Furthermore, the overlapping nodes will cause rays to hit multiple nodes during ray casting, resulting in more intersection tests and an inaccurate rendering of the volume.

Bautembach [38] dealt with such problems by scaling the nodes to cover the empty space for an animated model. Nodes around areas where two bones meet were selected to be scaled as this is the area where gaps are most likely to occur. Implementing a similar technique is not possible with standard animated volumes, as no bones exist. Instead, a distance check between the nodes is needed to accurately determine if scaling

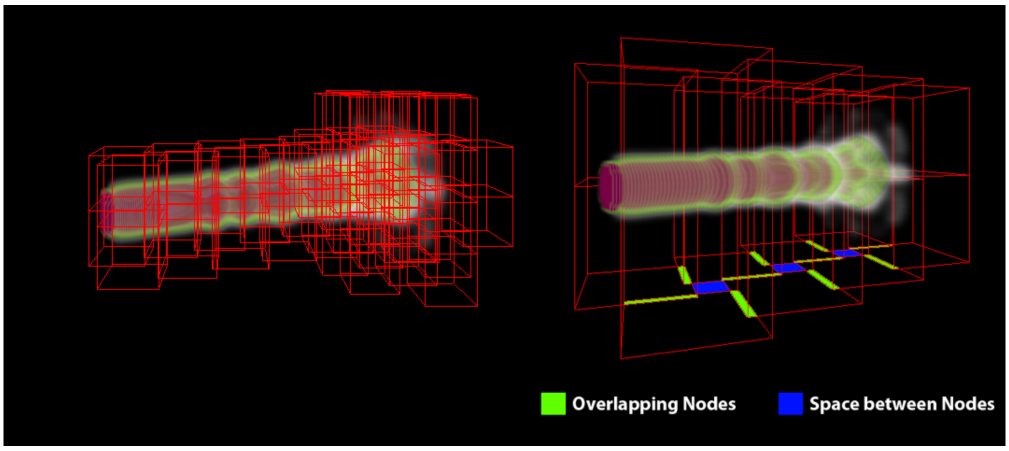


Figure 4.1: Rotating the octree (left) and errors introduced by rotating an octree (right). The green rectangles highlight overlapping nodes, while the blue square highlight gaps between nodes.

is required, and to which nodes. Such distance checks are expensive, and the cost of performing them would negate the benefit of rotating the octree instead of rebuilding it.

4.3.6 Visualizing the Octree

To visualize the octree, a wire-frame of each node within the octree is created during subdivision, and stored in a vector container allowing for the user to visualize different levels of the octree, see figure 4.2.

The visualization information is stored in a separate object in the main thread. This required the use of mutexes for the multiple threaded implementation, as it is possible for more than one thread to access the vector container, to read or write data, at the same time. A mutex is a synchronization mechanism used to ensure only one thread accesses common resources at a time. When a thread enters a section of code enclosed by a mutex, it locks that mutex, unlocking the mutex when it leaves. When another thread attempts to access the same code and the mutex is locked, the thread is denied access and is forced to wait until the mutex is unlocked before proceeding. This can slow down the generation of the octree as threads are forced to wait for other threads to complete. However a real world application, such as a game, would not

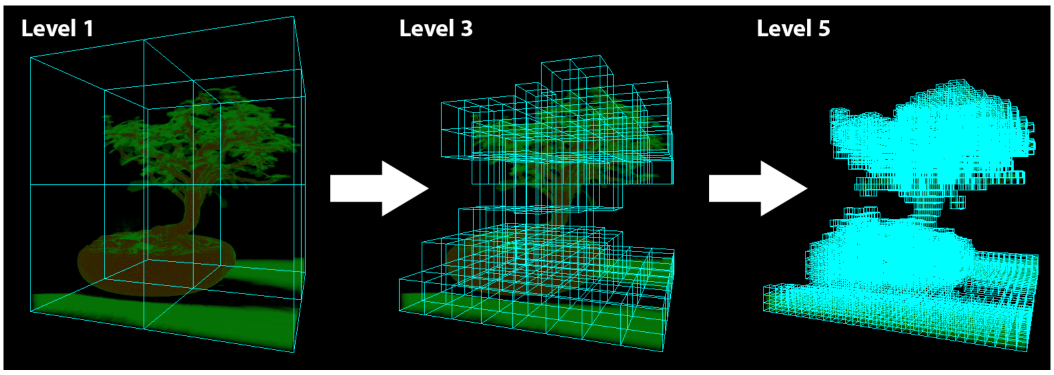


Figure 4.2: Visualizing the subdivision process of the bonsai tree, size 256^3.

require octree visualization, and so such mutexs would not be required.

4.4 Volume Deformation

To demonstrate the use of progressive octrees at interactive rates, the user is given the ability to deform the volume. The deformation approach implemented specifies what percentage of the volume to render. That is, if 100% is selected, the whole volume is rendered, if 50% is selected then half the volume is clipped along the x axis (see Figure 4.3). When the volume is deformed, the volume's octree is invalidated and is required to be rebuilt. If the volume is deformed again while the octree for the previous volume state is not complete, the generation process is interrupted and restarted with the new volume information.

Interrupting the octree generation process involves setting the octree's maximum depth to 0. This is because a thread, using boost threads, cannot be killed instantly. Doing so would be unsafe as the memory and resources it is using would be not be deleted. Instead, a thread will end when it has completed it's task or is interrupted, at certain pre-defined points set up by boost. During the octree generation process, the current depth of the tree is checked against the maximum depth of the tree regularly. Therefore, setting the octree's max depth to 0 is a quick way to ensure the thread finishes executing the octree generation process. This can result in multiple threads running however, where the old thread for creating the old octree has not finished executing before the new thread is started.

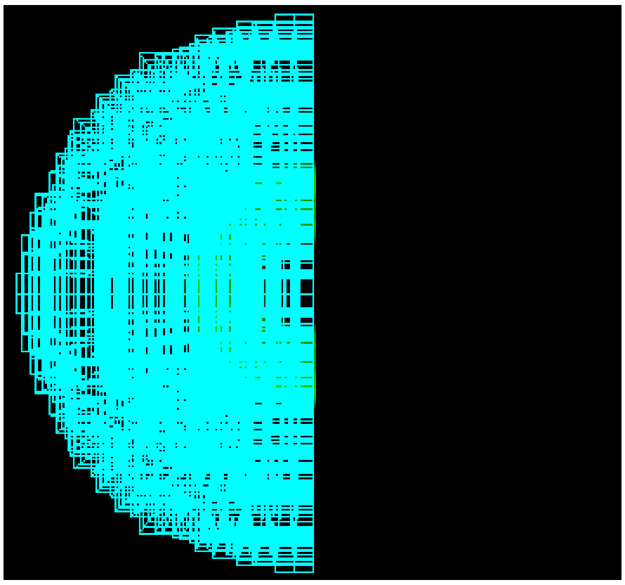


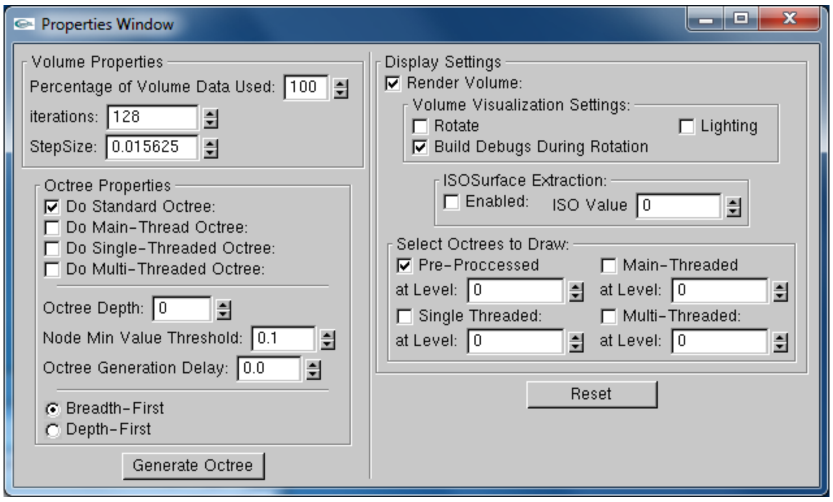
Figure 4.3: Octree generated for a deformed volume, where only 50% of the volume along the x axis is rendered.

4.5 User Friendly Interface

Using GLUI, a user friendly interface is created to allow the user to adjust settings and generate octrees. Among the options available to the user, as seen in figure 4.4, is a slider to allow the user to quickly deform the volume, as discussed in section 4.4. Furthermore, the user can determine which octree implementation to use and to what level to generate the octree to, change the generation type (i.e. breadth-first) and rotate the octree. The user can also visualize the octree, specifying the level of the octree to visualize.

4.6 External Libraries

The application is built using C++ and the OpenGL 2.1 graphics library. The application implements a number of external libraries and toolkits, such as GLUT, GLUI, glext.h, vmath, Boost library and the Nvidia Cg shading language, and was developed on Windows 7 using Visual Studio 2010 Professional, both by Microsoft. Followed is a brief description of these libraries and toolkits and their use within the system.


Properties Window

Volume Properties
Percentage of Volume Data Used: 100
iterations: 128
StepSize: 0.015625

Octree Properties
[✓] Do Standard Octree:
[ ] Do Main-Thread Octree:
[ ] Do Single-Threaded Octree:
[ ] Do Multi-Threaded Octree:
Octree Depth: 0
Node Min Value Threshold: 0.1
Octree Generation Delay: 0.0
● Breadth-First
○ Depth-First

Generate Octree

Display Settings
[✓] Render Volume:
Volume Visualization Settings:
[ ] Rotate
[ ] Lighting
[✓] Build Debugs During Rotation
ISOSurface Extraction:
[ ] Enabled: ISO Value 0
Select Octrees to Draw:
[✓] Pre-Proccessed at Level: 0
[ ] Main-Threaded at Level: 0
[ ] Single Threaded: at Level: 0
[ ] Multi-Threaded: at Level: 0

Reset

Figure 4.4: The user friendly interface with the application allowing the user to generate, rotate and visualize octrees and change settings within the application.

Libraries that are cross-platform were chosen where possible to meet the design goal of cross-platform compatibility.

4.6.1 OpenGL

OpenGL (Open Graphics Library) is an API developed by Silicon Graphics Inc. for 2D and 3D computer graphics. It was chosen over DirectX, a competing graphics API by Microsoft for the Windows operating system, as it is a cross-platform API allowing OpenGL programs to run on different operating systems without requiring code to be adapted. OpenGL version 2.1 is employed in this application.

4.6.2 GLUT and GLUI

GLUT (OpenGL Utility Toolkit) is a set of utilities that interfaces with the operating system to create windows, read keyboard inputs and mouse controls and to draw basic primitives such as cubes. This utility is used within this application to render the main window and read commands in the form of the mouse and keyboard. GLUT version 3.7 is used in this application.

GLUI (OpenGL User Interface Library) is a user interface library in C++ based on the functionality of GLUT. It provides quick and easy access to create UI buttons, drop-down menus, spinners, check-boxes and radio buttons. This library is implemented to create a user friendly user interface for the user to change parameters in the application such as the depth of the octree and options to visualize the octree. GLUI version 2.36 is used in this application.

4.6.3 glext.h

Glext.h is a header file with opengl extensions and constants defined for use in OpenGL 1.2 code. It is required to implement 3D textures, which are used in this application, in OpenGL 1.2 and OpenGL 2.0. Glext.h is designed for all operating systems, making the application more adaptable to different operating systems without requiring major code changes.

4.6.4 vmath

vmath is a set of Vector and Matrix classes for C++ designed for use in computer graphics. It contains built in functions to create Vector3 which are used extensively throughout the application. Furthermore, the library supports Vector2, Vector4, Matrix3, Matrix4 and Quaternions. A Vector3 is a class that is used to store 3D vectors, which is commonly used for storing 3D coordinates. Vector2 and Vector4 are 2D and 4D implementations of Vector3. Matrix3 stores a 3 x 3 matrix which can be used for matrix multiplications and transformations, while the Matrix4 class stores a 4 x 4 matrix. Quaternion class is used to represent a quaternion, another approach for achieving rotation instead of using matrices. Including the vmath library allowed time to be focused on the core aspects of the project, without needing to create these Vector classes. The version included in this application is vmath version 0.9.

4.6.5 Boost Library

Boost library is a set of c++ libraries available for free and proprietary software projects. It contains over 80 different libraries to extend C++ functionality. The library is designed to be cross-platform allowing for its use in both Linux and Windows which makes it ideal for the application developed as part of this research. The purpose of boost for this research project is for the use of threads, which are not part of the current C++ standards. Boost version 1.47 is used in this research project.

4.6.6 Cg Shading Language

Cg is implemented to support shaders which are used by the volume rendering pipeline. Cg is a high-level shading language that incorporates programmable vertex and pixel shaders on the GPU. It was developed by Nvidia in collaboration with Microsoft and as a result it closely matches the syntax of Microsoft's HLSL. This similarity with HLSL is the core reason for using Cg instead of GLSL, OpenGL's shading language, as we have prior experience with HLSL. The version of Cg included in this application is Cg version 2.0.

Chapter 5

Results & Evaluation

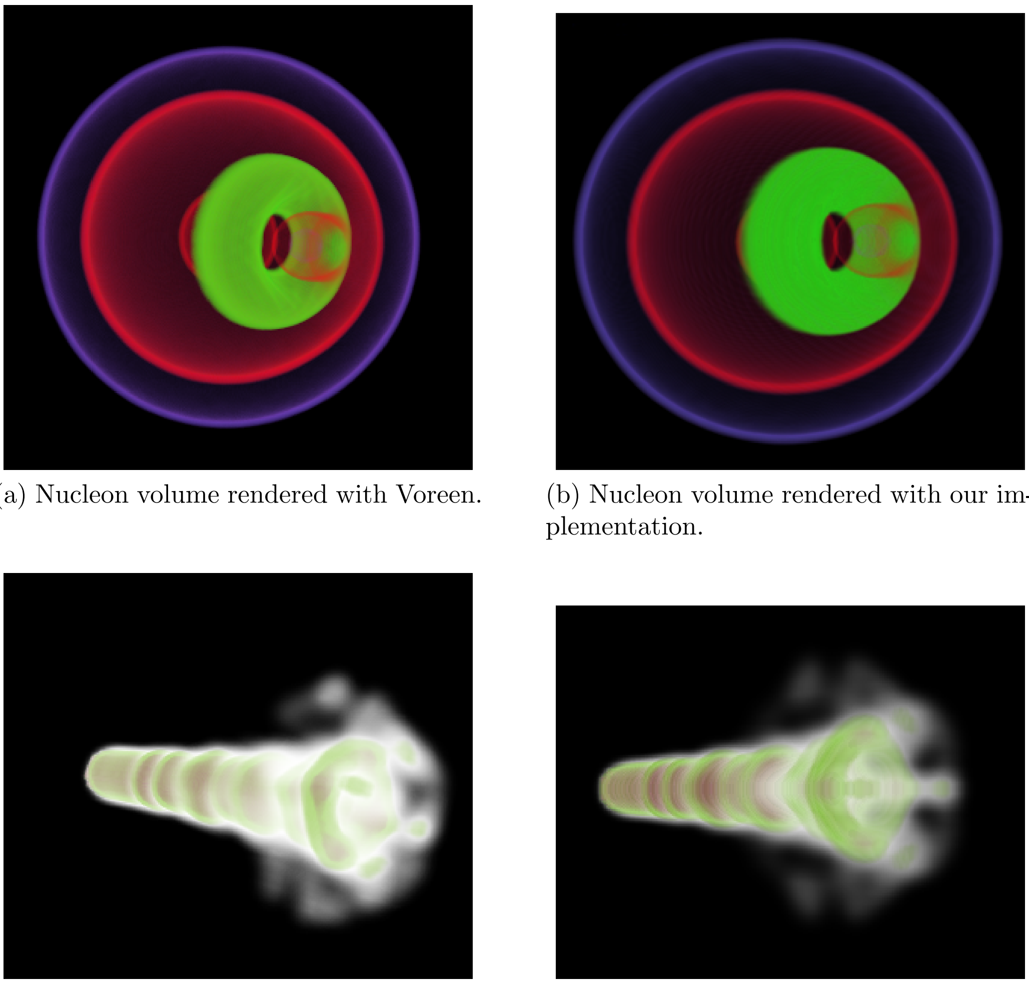
This chapter will evaluate the quality and performance of the implemented volume renderer and how successful the implementation of progressive octree generation was including octree rotation, with the objective to achieve progressive octree generation at interactive rates which can be applied to physically-based volume animations or user defined deformations.

This chapter will first evaluate the implemented volume renderer, before presenting result of the progressive octree generation implementation, including octree rotation. Finally, the chapter will end with an overall evaluation on the implementation based on the results presented previously. All tests were performed on an Intel Xeon W3520 CPU at 2.67GHz with 3.00GB RAM and on an NVIDIA Quadro FX 580.

5.1 Volume Renderer

Renderings from the GPU raycaster implemented are compared to equivalent renderings from a state of the art volume rendering engine, the Voreen Volume Rendering Engine [49]. The renderings are of the same volume data and use the same transfer function. Due to only simple lighting being implemented in this research, lighting in Voreen was disabled to compare both renderings under the same lighting conditions. As can be seen in figure 5.1, the implemented simple raycaster matches the quality of Voreen's simple raycaster, with Voreen producing slightly higher quality renderings.

The application ran at an average of 60 frames per second for a volume of 64^3,


(a) Nucleon volume rendered with Voreen. (b) Nucleon volume rendered with our implementation.

(c) Fuel volume rendered with Voreen. (d) Fuel volume rendered with our implementation.

Figure 5.1: Comparasion between volumes rendered with Voreen and our implementation.

and an average of 20 frames per second for a volume of size 256^3 at a resolution of 800 x 600. However, the application's frame rate reduces the closer the camera gets to the volume, as more rays are hitting the volume as a result. Although certain obvious optimizations were implemented for the raycaster such as early ray termination, further optimizations were beyond the scope of this project and we refer the reader to [5] for more information on such optimizations. Integrating the octree into the raycaster would also significantly increase rendering performance.

These results show that simple GPU ray casting can produce quality renderings

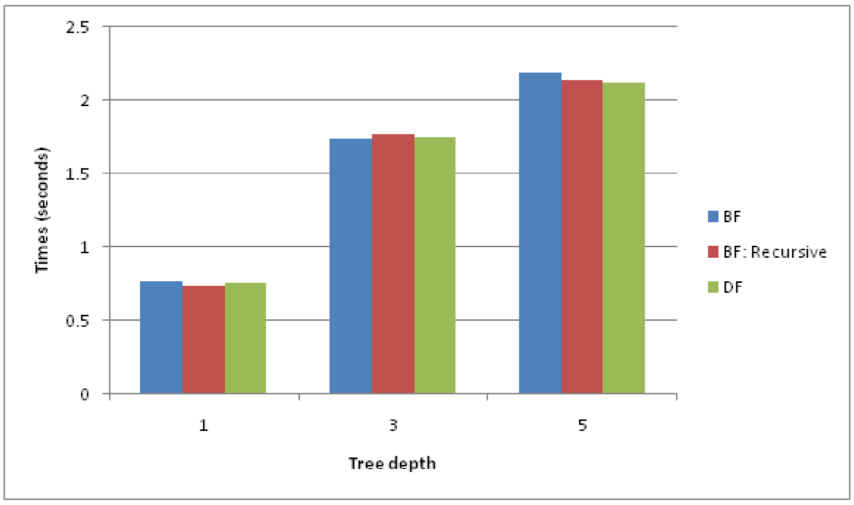


Figure 5.2: The octree generation times for the bonzai volume (256^3) using breadth-first, breadth-first with recursion and depth-first.

at interactive rates. The CPU ray caster, however, performed significantly worst than the GPU raycaster, as expected, averaging 2 frames per second for the nucleon volume (41^3), 2.5 frames per second for the fuel volume (64^3) and 1.5 frames per second for the bonsai volume (256^3).

5.2 Octree Generation Results

For evaluating the generation times between the different approaches, i.e. breadth-first using recursion and without recursion, as well as depth-first, tests were performed on the main thread during pre-processing of the bonsai volume (256^3). The progressive octree generation implementations were not used to gather these results as an accurate timing, without taking rendering and updating the application into account, on each approach was needed to gauge which is the best, if any. The results show that no particular implementation is better, with the differences between each approach negligible (see Figure 5.2). The timings for the recursive and non-recursive implementation of breadth-first show that neither approach offers significant gains over the other.

These results demonstrate that breadth-first is a suitable approach to generate the octree progressively, not only in terms of the ability to draw each level as it is generated, but also due to depth-first offering no significant gains in performance.

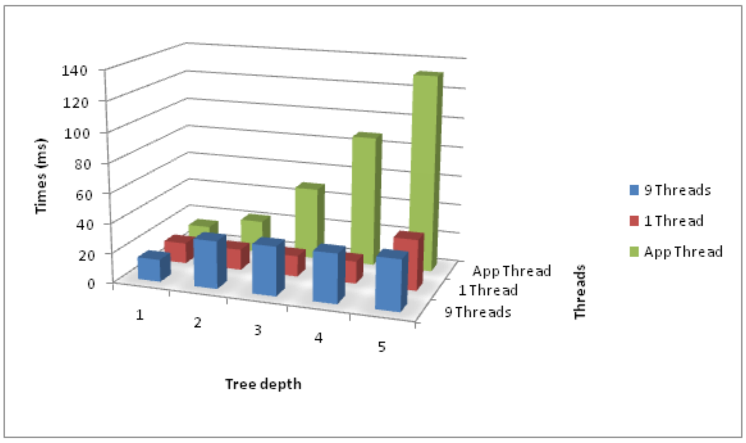
5.3 Progressive Octree Results

Evaluation of the progressive octrees involved performing progressive octree generation for three different volume sizes, 41^3, 64^3 and finally 256^3. For each test, the volume was subdivided to a maximum of 7 different levels, levels 1 to 7 or until the maximum depth of the tree is reached, that is each node in the tree represents only one voxel. These tests are performed using the main thread, a single dedicated thread and multiple threads, 9, for generating the octree. The results of each test are broken into 3 parts:

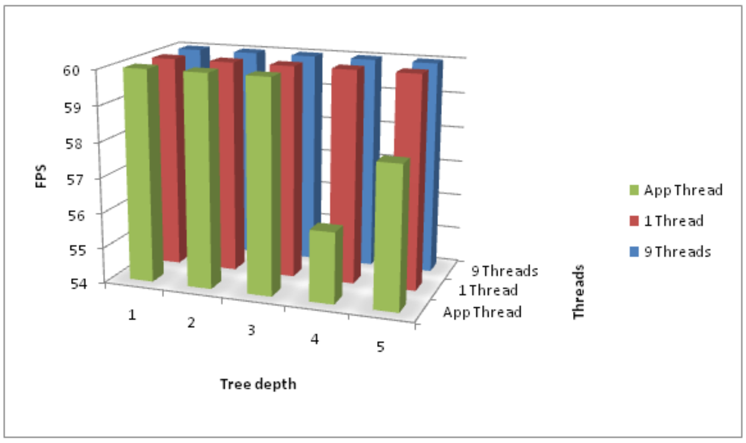
- The time taken to completely generate the octree.
- The average frame rate (fps) of the application during the octree generation process.
- The minimum fps of the application during the octree generation process. This is noted as the aim is to achieve octree generation at interactive rates. While the average fps for the generation process may be high, it is possible that the application is drastically slowed down for a small amount of time during the generation process, making the application not interactive for that duration. The aim is to achieve a consistent 60 frames per second.

While performing the tests, the application was rendering the volume been subdivided, with the camera located approximately 150 units from the volume. Furthermore, octree visualization is disabled as it is used only for debugging purposes, and would not be implemented in a real world application.

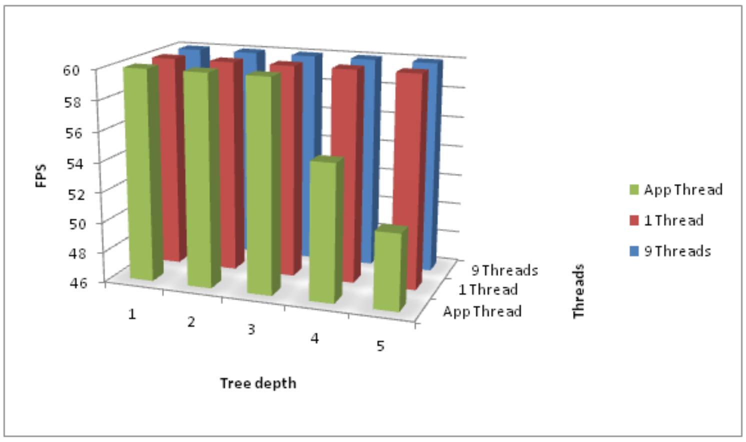
It is worth noting that an interactive application would be performing several other tasks as well, such as physics calculations and AI calculations. Such tasks are not taken into account for these tests. This is to get accurate results for octree generation as an individual task. With these results, it can be determined if full integration into a game is possible, or if further optimizations are required before hand.



(a) Time taken to progressively generate octree to specific depths.



(b) Average fps during the progressive octree generation.



(c) Minimum fps during the progressive octree generation.

Figure 5.3: Results for progressive octree generation for volume size 41^3.

5.3.1 Results for Volume size 41^3

For a volume size of 41^3, the results (see Figure 5.3a) show that the use of multiple threads have no advantage over a single thread for small volume sizes. In this case, the time spent setting up the threads and managing them negates the benefit of using multiple threads. However, for an octree depth of 5, multiple threads complete the octree generation process in the same time as the single thread (35 milliseconds), which indicates that multiple threads become beneficial over the use of single threads for low depths on these small volumes. The main thread implementation took the longest time to generate, as expected, requiring an additional 98 milliseconds more than then single and multi-threaded implementations.

The results for the frame rate of the application throughout the generation process (see Figure 5.3b and 5.3c) shows the application's fps remained consistent for all threads, maintaining an average fps of 60, with the main thread reaching the lowest fps of 51 for a tree depth of 5. While this is below the goal fps of 60, it is acceptable for interactive applications.

Further subdivision beyond depth 5 is not necessary for this volume, as a depth of 5 results in each node representing only one voxel, therefore the use of multiple threads have no advantage over single threads for volumes of this size.

5.3.2 Results for Volume size 64^3

For a volume size of 64^3, the multi-threaded implementation, requiring 49 milliseconds, was only 4 milliseconds faster than the single-threaded implementation for a tree depth of 6 (see Figure 5.4a), providing a speed increase of 8% over the single-threaded implementation, and over 400% faster than using the main thread.

The performance of the application (see Figure 5.4b and 5.4c) remained at 60 fps for the single and multi-threaded implementations. However, despite having an average of 55 fps, the main-thread slowed the application's performance to a minimum fps of 50, again below the desired threshold of 60 fps. This can be prevented by adjusting the time given to generate the octree each frame, by increasing the pre-determined amount of time deducted from the time left in the current frame, as discussed in section 4.3.4. However, it is not possible to guarantee the generation process will not overrun its allocated time without the ability of interrupting the subdivision of a node.

It is worth noting that a volume size of 64^3 is a reasonable size for volumes to represent objects and items in games, however a higher resolution would be required for main characters.

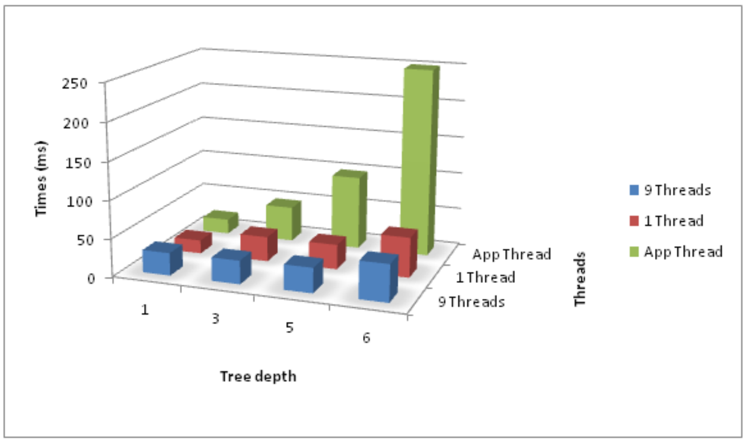
Once again, a depth of 6 for this volume is the maximum depth the tree can go, when each node represents one voxel.

5.3.3 Results for Volume size 256^3

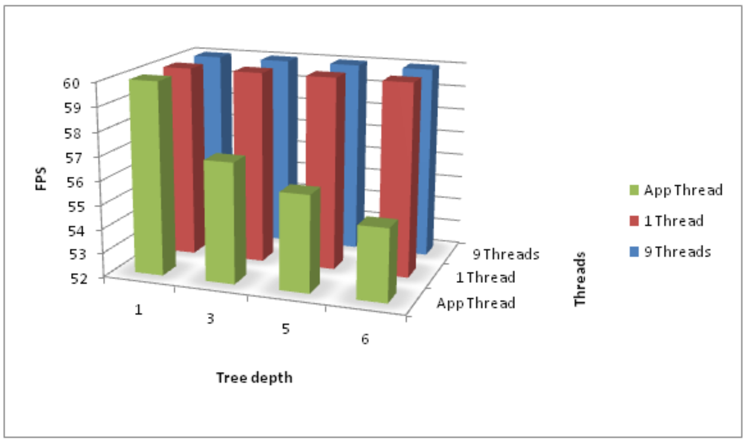
Generating an octree for this large volume, while possible at interactive rates, is not suitable due to the generation times reaching and exceeding several seconds (see Figure 5.5a). The main thread implementation took the longest time, as expected, requiring over 9 seconds to generate to a depth of 7. Furthermore, the frame rate dropped to 0 while generating the octree on the main thread, making the application unresponsive during the generation process.

As expected, the multi-threaded implementation generated the octree the fastest, over 2 times faster than the single-threaded solution. However these timings are not acceptable for interactive applications. Despite the multi-threaded implementation generating faster, like the previous results, generating the first level takes longer than the other implementations. This is because subdividing the root node can only be performed by a single thread, and therefore the remaining eight threads serve no purpose. By simply not creating these threads, the generation time would be reduced, matching that of the single-threaded implementation. Subdividing the children of the root node, that is generating level 2, takes significantly less time as demonstrated by the multi-threaded timings as the eight additional threads are implemented, where generating level 2 only took an additional 12 milliseconds, compared to the 1.15 seconds needed to create level 1. Compared to the single-threaded, which took 98 milliseconds to generate level 1, and an additional second to generate level 2, the benefits of multi-threading in terms of speed is significant.

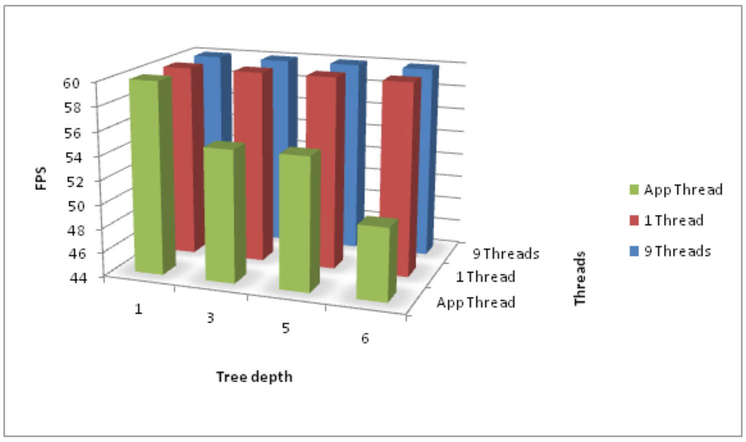
Furthermore, the processing of such large volume data over 8 threads drastically impacted the performance of the application (see Figure 5.5b and 5.5c). Despite maintaining an average of 55 fps, the application hit a low of 15 fps during the generation process, whereas the single-threaded solution only hit a low of 55 fps, maintaining a consistently high frame rate.



(a) Time taken to progressively generate octree to specific depths.

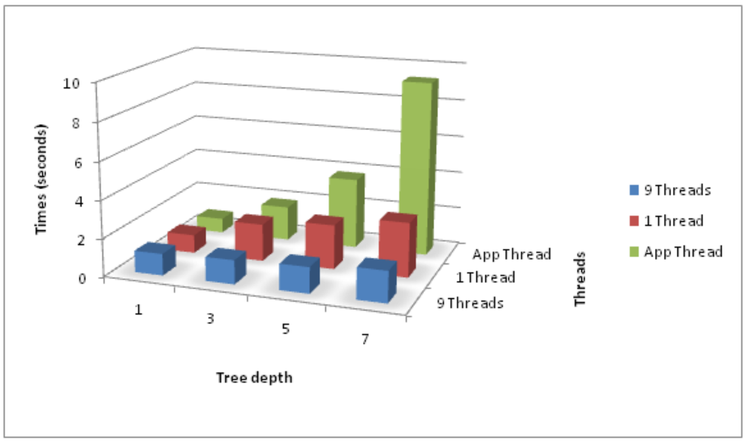


(b) Average fps during the progressive octree generation.

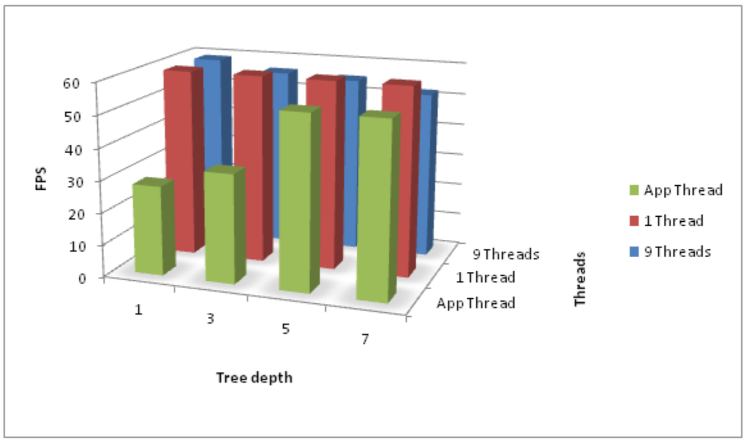


(c) Minimum fps during the progressive octree generation.

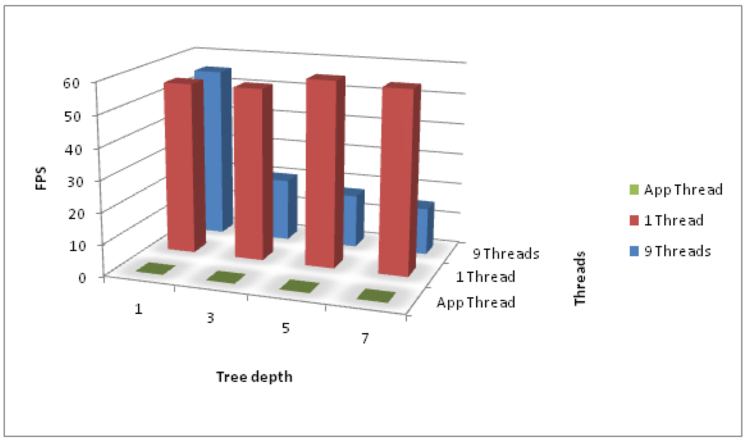
Figure 5.4: Results for progressive octree generation for volume size 64^3.



(a) Time taken to progressively generate octree to specific depths.



(b) Average fps during the progressive octree generation.



(c) Minimum fps during the progressive octree generation.

Figure 5.5: Results for progressive octree generation for volume size 256^3.

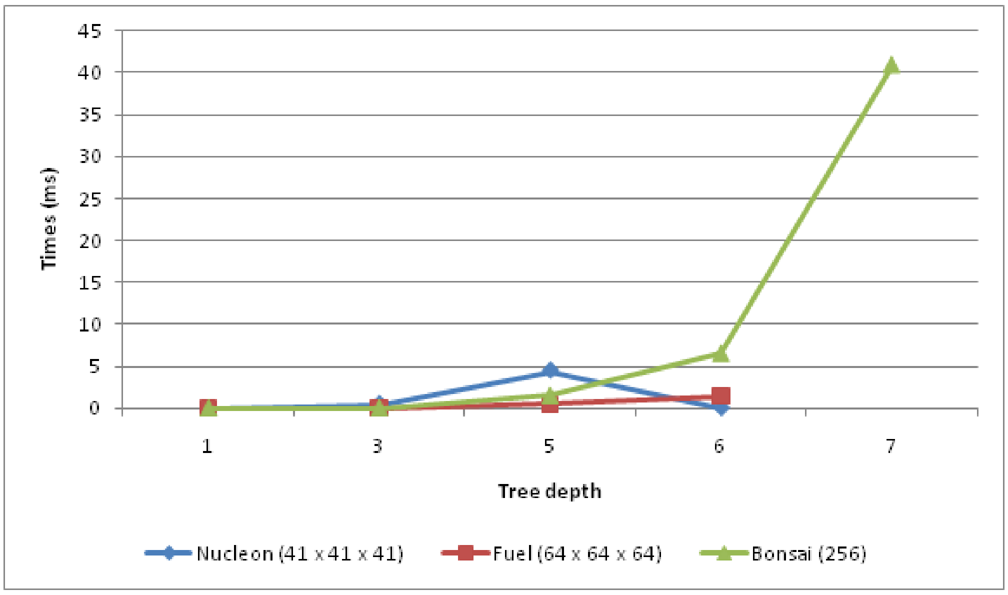


Figure 5.6: The times, in milliseconds, taken to rotate octree of different depths for different volumes.

5.4 Octree Rotation

The results for rotating the octree show that significant time is saved compared to building the octree (see Figure 5.6). For low depths of 1 to 3, the time taken to rotate the octree were negligible, with accurate timings difficult to gather as the time taken was less than 1 millisecond. While rotating the octree saves significant time, over 25 times faster compared to some of the progressive octree generation results, errors are introduced into the octree representation of the volume (see Figure 4.3), which are difficult to solve efficiently.

5.5 Overall Evaluation

From these results, we can see that progressive octree generation using breadth-first traversal at interactive rates is achievable for small volumes, however larger volumes are currently not possible, with long generation times, exceeding seconds, required to fully generate the octree to low depths. It is worth noting that while a volume size of 64^3 is relatively small compared to the standard sizes of existing volumes, such a size is more than appropriate for use within video games, and is of a high enough resolution to represent game objects and units. Furthermore, medical and visualisation applications typically render a single object at high resolutions. In games, it is likely that several small volumes are rendered instead of a single large volume.

Furthermore, the single-threaded implementation produces the octree consistently faster than the multi-threaded solution for small volumes, with the exception of the 64^3 volume where the multi-threaded implementation is 4 milliseconds faster at a depth of 6. However, multiple-threads provide the best solution for large volumes in terms of speed, but at a cost of consistent frame rate. The single-threaded implementation produced the most consistent frame-rate of all the implementations. This is important to note, as a game is likely to have several threads running for various tasks such as physics and AI. Therefore, adding 9 additional threads for generating the octree will impact the overall system performance, as using multiple threads have costs associated with them, such as context switching where the CPU stores and restores a state when switching between threads and processes on the same CPU.

An interesting point worth noting is that the size of the volume is not necessarily the deciding factor for the length of time required to the generate the octree. In the fact, the most significant impact the size of the volume has is the time required to subdivide the root node, as it requires all voxels in the volume data to be traversed. What impacts the octree generation time is the number of non-empty voxels in the volume, which requires more nodes to be created and subdivided. For example, a volume of size 512^3 would take a significant time to subdivide the root node, as all 134,217,728 voxels have to be traversed whether they are empty or not. However, after the root node is subdivided, further subdivisions require traversing only the voxels that are associated with the node. Since nodes are only created for voxels that contribute to the final rendering, large sections of empty voxels are skipped reducing the number of voxels traversed as the octree is subdivided further.

Rotating the octree is a quick way of adapting the octree for a rotated volume. However, the gaps and overlays formed as a result mean that rotating the octree no longer represents the volume accurately, and multiple voxels will be sampled more than once when the octree is integrated with the raycaster. A naive solution to this would be to rotate the octree if the angle of rotation for the volume is below a threshold. This results in an acceptable amount of errors in the octree, that is, gaps between nodes and overlapping nodes. However, once the angle exceeds the threshold, the entire octree is rebuilt. Regardless, these results show that significant time can be saved by adapting the octree instead of rebuilding it, and that this area is worth further research.

Chapter 6

Conclusion

For this dissertation, we proposed to investigate octree generation for interactive animated volume rendering, presenting an approach to progressively generate an octree for the rendering of physically-based animated volumes or for user-defined volume deformations. The main research area of this dissertation was to investigate progressive octree generation over several frames, rendering the octree as it is being generated, at interactive rates and to implement it in a user-friendly application that allows the user to modify volume data during run time, requiring an octree to be progressively built. Furthermore, an approach to adapt the octree was implemented in the form of rotating an existing octree, a process used instead of rebuilding the octree when the volume the octree represents is rotated.

The implementation of the progressive octree generation was done using three approaches, one using the main application's thread, the second using a dedicated thread created by the main application, and the third using multiple dedicated threads, in this case 9. The implemented approaches were tested and the results indicated that progressively generating octrees at interactive rates is possible for small volumes (while these volumes are small, such as 64^3, compared to the standard size of volumes, this size is more than suitable to represent objects within a game). Larger volumes, such as 256^3, are not currently possible as the generation times exceed one second to generate an octree beyond a depth of 1.

Furthermore, the single-threaded implementation provided the best overall results for small volumes in terms of both generation times and application frame rates. The multi-threaded approach generated the octree faster than the other 2 approaches for large volumes. Interestingly, for the large volume, 256^3, the application maintained a more consistent frame-rate for the single-threaded approach, only reaching a low of 55 frames per second while generating the octree to a depth of 7, whereas the multi-threaded approach reached a low of 15 frames per second. While the generation times for the single-threaded approach were over a second longer than the multi-threaded approach, these results show that the single-threaded approach maintains a more consistent frame-rate, and is the best option for applications where a consistent frame rate is more desirable than generation speed.

Results showed that implementing the breadth-first approach to progressively generate the octree is a suitable approach, with no significant gains to be achieved in terms of generation times by using depth-first instead.

Additionally, results demonstrated that adapting the octree by rotating it can save significant time, compared to rebuilding the octree for a newly rotated volume. In some cases, the time taken to rotate the octree was negligible, below 1 millisecond. However, problems presented with this approach demonstrate the limitations with this technique and highlight the need for further investigation into possible solutions.

We are pleased with the results achieved from the progressive octree implementation, and the resulting application that allows the user to modify volume data and subsequently generating a new octree progressively, demonstrating that this is applicable for interactive applications, games in particular where small volumes would be more likely to be used than large volumes, and can be implemented for user-defined volume deformations at interactive rates.

6.1 Future Work

Despite the work done in this research project, there is still further work which we would like to undertake in this area, and these are outlined in this section. These have not been implemented in this research project because they were either beyond the scope of this dissertation or due to time constraints.

6.1.1 Optimizations to GPU RayCaster

As previously mentioned, certain optimizations such as early ray termination have been implemented for the GPU raycaster in this research project. However, there are many more optimizations that can be applied to increase the performance of the renderer.

We would like to implement some of these optimizations presented by Kruger and Westermann [5]. Furthermore, we would like to successfully incorporate the CPU octree with the GPU raycaster, implementing the octree texture approach presented by Pharr and Fernando [3], where the octree is stored in a three dimensional texture and uploaded to the GPU. Integration into the GPU raycaster involves performing texture lookups on the octree texture with the position of the current ray during the ray casting process. This would vastly increase rendering performance as it can be determined quickly if a ray is going to hit the volume or not.

6.1.2 Investigate Potential Solutions to Rotation Errors

As highlighted in Figure 4.3, the errors introduced by rotating the octree limit the benefits of octree rotation. A naive solution of performing distance tests between each node was proposed, but such an implementation would be slow for volumes subdivided to low levels of depths such as 6, 7 or 8. The benefits of rotating the octree instead of rebuilding are very significant, and so we feel further investigation into this problem is warranted.

6.1.3 Octree Adaptation

As demonstrated by the octree rotation results, modifying an existing octree is much more time efficient than rebuilding the octree from scratch. This could be extended further to only recreate parts of the octree. By detecting the locations of changes to a volume, it can be determined which nodes are no longer valid. This would allow for affected nodes of the octree to be rebuilt, while non-affected nodes can be maintained as they are, reducing the time required to re-adjust the octree for the newly modified volume.

6.1.4 Octree Generation on the GPU

The GPU has fast become a powerful processor, with modern GPU's experiencing a rapid increase in performance on a yearly basis. As a result, the performance of the GPU is well beyond that of the CPU. This has led to the GPU being used for performing computationally expensive operations which are traditionally performed on the CPU.

We would like to benefit from the GPU processing power by generating the octree on the GPU, based on the implementation presented by Goradia [50], taking advantage of the high performance of the GPU to reduce the construction times of the octrees.

6.1.5 Implementation within a Game

The evaluation tests were performed on our application which was designed purely for rendering volumes and generating octrees. For real-world use, progressive octrees would be integrated to a large application, such as a game, which would also be performing various processing intensive tasks. For example, a game would be required to process collision detection, physic simulations, AI calculations, network processing and so forth for every frame. Many commercial games perform so many operations every frame that they are unable to reach 60 frames per second, but instead aim for 30 frames per second.

For future work, we would like to integrate progressive octree generation into a game application which performs many of the operations mentioned above, such as physic simulations and AI. This would enable an analysis and evaluation into the use of progressive octree generation in a real world application, highlighting potential problems that may be associated with integration within a game.

6.1.6 User Perception Tests

The fundamental aspect of the research is drawing an octree as it is being generated at interactive rates. This results in the octree depth level 1 being displayed while level 2 is being generated, and so forth. It is important to test how user's perceive this transition from one level to the next and to check if the delay between drawing each level is noticed by the users, determining what are acceptable generation times before the transition from one level to the next is noticed by the user, and how such transitions affect the overall user experience.

For such tests, it would be important to set up an interactive application, such as a short game, where the user can interact with volumes, such as modifying and deforming the volume. This would make the results obtained from the tests applicable to animated volume rendering in computer and video games.

Appendix

Acronym	Definition
CPU	Central Processor Unit
GB	GigaByte
GLSL	OpenGL Shading Language
GLUI	OpenGL User Interface Library
GLUT	OpenGL Utility Toolkit
GPU	Graphics Processor Unit
HLSL	High Level Shading Language
MB	Mega Byte
RGB	Red Green Blue
RGBA	Red Green Blue Alpha

Table 1: Acronyms commonly used in the text.

Bibliography

[1] K. Engel, M. Hadwiger, J. Kniss, A. Lefohn, C. Salama, and D. Weiskopf, "Real-time volume graphics," in *ACM Siggraph 2004 Course Notes*, pp. 29–es, ACM, 2004.

[2] V. Visualization, "Ray casting." `http://www.volviz.com/images/VolumeRayCastingWithRay2.png` [Retrieved 20-August-2011], 2011.

[3] M. Pharr and R. Fernando, "Gpu gems 2: Programming techniques for high-performance graphics and general-purpose computation," 2005.

[4] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 451–458, ACM, 1994.

[5] J. Kruger and R. Westermann, "Acceleration techniques for gpu-based volume rendering," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, p. 38, IEEE Computer Society, 2003.

[6] Wikipedia, "Octree — wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Octree&oldid=416514939l` [Retrieved 20-August-2011], 2011.

[7] Atomontage, "Atomontage engine." `http://www.atomontage.com/` [Retrieved 26-August-2011].

[8] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 15–22, ACM, 2009.

[9] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pp. 55–63, ACM, 2010.

[10] W. Lorensen and H. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM Siggraph Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[11] L. Kobbelt, M. Botsch, U. Schwanecke, and H. Seidel, "Feature sensitive surface extraction from volume data," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 57–66, ACM, 2001.

[12] L. Yan and Z. Min, "A new contour tracing automaton in binary image," in *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, vol. 2, pp. 577 –581, june 2011.

[13] M. Levoy, "Display of surfaces from volume data," *Computer Graphics and Applications, IEEE*, vol. 8, no. 3, pp. 29–37, 1988.

[14] H. Scharsach, "Advanced gpu raycasting," *Proceedings of CESCG*, vol. 5, pp. 67–76, 2005.

[15] E. Gobbetti, F. Marton, and J. Iglesias Guitián, "A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *The Visual Computer*, vol. 24, no. 7, pp. 797–806, 2008.

[16] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive kd tree gpu raytracing," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pp. 167–174, ACM, 2007.

[17] S. Laine and T. Karras, "Efficient sparse voxel octrees–analysis, extensions, and implementation," 2010.

[18] G. Cameron and P. Undrill, "Rendering volumetric medical image data on a simd-architecture computer," in *Proceedings of the Third Eurographics Workshop on Rendering*, pp. 135–145, 1992.

[19] Y. Wu, V. Bhatia, H. Lauer, and L. Seiler, “Shear-image order ray casting volume rendering,” in *Proceedings of the 2003 symposium on Interactive 3D graphics*, pp. 152–162, ACM, 2003.

[20] L. Westover, “Footprint evaluation for volume rendering,” *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4, pp. 367–376, 1990.

[21] D. Laur and P. Hanrahan, “Hierarchical splatting: A progressive refinement algorithm for volume rendering,” in *ACM SIGGRAPH Computer Graphics*, vol. 25, pp. 285–288, ACM, 1991.

[22] K. McDonnell, N. Neophytou, K. Mueller, and H. Qin, “Subdivision volume splatting,”

[23] B. Cabral, N. Cam, and J. Foran, “Accelerated volume rendering and tomographic reconstruction using texture mapping hardware,” in *Proceedings of the 1994 symposium on Volume visualization*, VVS ’94, (New York, NY, USA), pp. 91–98, ACM, 1994.

[24] O. Wilson, A. VanGelder, and J. Wilhelms, “Direct volume rendering via 3d textures,” tech. rep., Santa Cruz, CA, USA, 1994.

[25] R. Westermann and T. Ertl, “Efficiently using graphics hardware in volume rendering applications,” in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 169–177, ACM, 1998.

[26] M. Gross, L. Lippert, R. Dittrich, and S. Haring, “Two methods for wavelet-based volume rendering,” *Computers & Graphics*, vol. 21, no. 2, pp. 237–252, 1997.

[27] L. Lippert, *Wavelet-based volume rendering.* PhD thesis, SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, 1998.

[28] B. Lichtenbelt, *Fourier volume rendering.* Hewlett-Packard Laboratories, Technical Publications Dept., 1995.

[29] A. Entezari, R. Scoggins, T. Moller, and R. Machiraju, “Shading for fourier volume rendering,” in *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pp. 131–138, IEEE Press, 2002.

[30] K. Ma, “Visualizing time-varying volume data,” *Computing in Science & Engineering*, vol. 5, no. 2, pp. 34–42, 2003.

[31] H. Shen and C. Johnson, “Differential volume rendering: A fast volume visualization technique for flow animation,” in *Proceedings of the conference on Visualization'94*, pp. 180–187, IEEE Computer Society Press, 1994.

[32] K. Ma and H. Shen, “Compression and accelerated rendering of time-varying volume data,” in *Proceedings of the 2000 International Computer Symposium-Workshop on Computer Graphics and Virtual Reality*, pp. 82–89, Citeseer, 2000.

[33] B.-S. Sohn, C. Bajaj, and V. Siddavanahalli, “Feature based volumetric video compression for interactive playback,” in *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, VVS '02, (Piscataway, NJ, USA), pp. 89–96, IEEE Press, 2002.

[34] L. Linsen, V. Pascucci, M. A. Duchaineau, B. Hamann, and K. I. Joy, “Hierarchical representation of time-varying volume data with "4th-root-of-2" subdivision and quadrilinear b-spline wavelets,” in *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*, PG '02, (Washington, DC, USA), pp. 346–, IEEE Computer Society, 2002.

[35] H.-W. Shen, L.-J. Chiang, and K.-L. Ma, “A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree,” in *Proceedings of the conference on Visualization '99: celebrating ten years*, VIS '99, (Los Alamitos, CA, USA), pp. 371–377, IEEE Computer Society Press, 1999.

[36] H. Pfister, B. Lorensen, W. Schroeder, C. Bajaj, and G. Kindlmann, “The transfer function bake-off (panel session),” in *Proceedings of the conference on Visualization '00*, VIS '00, (Los Alamitos, CA, USA), pp. 523–526, IEEE Computer Society Press, 2000.

[37] T. Jankun-Kelly and K. Ma, “A study of transfer function generation for time-varying volume data,” in *Volume graphics 2001: proceedings of the joint IEEE TCVG and Eurographics workshop in Stony Brook, New York, USA, June 21-22, 2001*, p. 51, Springer Verlag Wien, 2001.

[38] D. Bautembach, "Animated sparse voxel octrees," March 2011.

[39] D. Madeira, A. Montenegro, E. Clua, and T. Lewiner, "Gpu octrees and optimized search,"

[40] F. Ganovelli, J. Dingliana, and C. OSullivan, "Buckettree: Improving collision detection between deformable objects," in *Proc. of Spring Conference on Computer Graphics SCCG00*, vol. 11, Citeseer, 2000.

[41] J. Gu and S. Wei, "An octree ray casting algorithm based on multi-core cpus," in *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, vol. 2, pp. 783–787, IEEE.

[42] B. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, no. 6, p. 317, 1975.

[43] J. Blinn, "Models of light reflection for computer synthesized pictures," in *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pp. 192–198, ACM, 1977.

[44] D. Nagayasu, F. Ino, and K. Hagihara, "Two-stage compression for fast volume rendering of time-varying scalar data," in *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia, November*, Citeseer.

[45] K. Schwaber, *Agile project management with Scrum*, vol. 7. Microsoft Press Redmond (Washington), 2004.

[46] K. Hayward, "Volume rendering 101." `http://graphicsrunner.blogspot.com/2009/01/volume-rendering-101.html` [Retrieved 26-August-2011], 2009.

[47] R. de Bruijne, "ray/aabb intersection." `http://pastebin.com/PCmvDFK` [Retrieved 26-August-2011], 2011.

[48] K. Hayward, "Volume rendering 102." `http://graphicsrunner.blogspot.com/2009/01/volume-rendering-102-transfer-functions.htm` [Retrieved 26-August-2011], 2009.

[49] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs, "Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations," *Computer Graphics and Applications, IEEE*, vol. 29, pp. 6 –13, nov.-dec. 2009.

[50] R. Goradia, "Gpu-based adaptive octree construction algorithms." `http://www.cse.iitb.ac.in/~rhushabh/publications/octree.pdf` [Retrieved 26-August-2011].