fl

Performance comparison on rendering methods for voxel data

Oskari Nousiainen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo July 24, 2019

Supervisor

Prof. Tapio Takala

Advisor

Dip.Ins. Jukka Larja

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Oskari Nousiainen

Title Performance comparison on rendering methods for voxel data

Degree programme Master's Programme in Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Prof. Tapio Takala

Advisor Dip.Ins. Jukka Larja

Date July 24, 2019 **Number of pages** 50 **Language** English

Abstract

There are multiple ways in which 3 dimensional objects can be presented and rendered to screen. Voxels are a representation of scene or object as cubes of equal dimensions, which are either full or empty and contain information about their enclosing space. They can be rendered to image by generating surface geometry and passing that to rendering pipeline or using ray casting to render the voxels directly.

In this paper, these methods are compared to each other by using different octree structures for voxel storage. These methods are compared in different generated scenes to determine which factors affect their relative performance as measured in real time it takes to draw a single frame.

These tests show that ray casting performs better when resolution of voxel model is high, but rasterisation is faster for low resolution voxel models. Rasterisation scales better with screen resolution and on higher screen resolutions most of the tested voxel resolutions where faster to render with with rasterisation.

Ray casting performed better only when model resolution was high compared to screen resolution or the model had high surface filling rate but low coherency. Differences between octree models where significant, bit in some cases even the base voxel data was faster.

Keywords Computer Graphics, Real-Time Rendering, Voxels

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Oskari Nousiainen

Työn nimi Vokselidatan piirtämismenetelmisen tehokkuusvertailu

Koulutusohjelma Master's Programme in Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja Prof. Tapio Takala

Työn ohjaaja Dip.Ins. Jukka Larja

Päivämäärä July 24, 2019 **Sivumäärä** 50 **Kieli** Englanti

Tiivistelmä

Kolmiulotteisia kappaleita ja ympäristöjä voidaan esittää ja piirtää useammilla tavoilla. Vokseleilla malli esitetään kuutioiksi jaettuna alueena, jossa kuutio voi olla joko tyhjä tai täysi ja sisältää muuta informaatiota, jota tilan piirtämiseen voidaan käyttää. Vokseleita voidaan piirtää, joko luomalla niille pintageometria rasterointia varten tai käyttämällä ray cast -menetelmiä.

Tässä tutkimuksessa menetelmiä vertailaan mittaamalla kuvan piirtämiseen käytettyä aikaa molemmilla menetelmillä muuttamalla piirettävien kolmiulotteisten mallejen ja piirretävän kuvan resoluutiota. Lisäksi testataan erilaisten octree-rakenteiden käyttämistä piirtämisen nopeuttamiseen.

Tutkimuksen testit osoittavat, että rasterointi-menetelmä on nopeampi kun vokselimallin resoluutio on pieni ja kuvan resoluution kasvaessa se ottaa kiinni ray cast -menetelmää myös keski ja suuri resoluutioisilla malleilla. Rasterointi menetelmä on hyvin riipuvainen siitä, että piirrettävä malli pakkautuu hyvin octree-rakenteeseen, mutta eri octree-toteutusten erot suorituskykyyn ovat hyvin vähäisiä.

Ray cast menetelmät ovat nopeampia, kun vokselimallin resoluutio on korkea tai sen sisältö ei ole koherentia, mutta täyttöaste lähellä laitoja on korkea. Erot eri octree menetelmien välillä ovat merkittäviä, mutta joissain tapauksissa myös alkuperäinen vokselidata oli nopeampi.

Avainsanat Tietokone grafiikka, Realiaikainen piirtäminen, vokselit

Preface

I want to thank my friends, only people who where actually actively interested in what my thesis was about and not just if it is ready.

I would also like to thank other researchers of this field who take time to write their papers in a clear and concise manner and include definitions of their mathematical notation in their papers.

Thanks.

Otaniemi, July 24, 2019

Oskari M. Nousiainen

Acknowledgments

My employment by Frozenbyte Oy or my education at Aalto University might have influenced writing of this paper.

Contents

Definitions and Notations

Abbreviations

API	Application Programming Interface, part of software that allows other software to use some functionality it offers
CPU	Central Processing Unit, generic computation unit found in most computers
GPU	Graphics Processing Unit, computation device optimised for parallel floating point arithmetic needed in rendering
MIMD	Multiple Instructions, Multiple Data, architecture where each processing unit has their own instructions and parts of data
SIMD	Single Instruction, Multiple Data, architecture where same instructions are run parallel with different data
SIMT	Single instruction, multiple threads, extension of SIMD with multithreading
FPS	Frames Per Second, common way of measuring graphical performance as number of images shown each second
ms	Millisecond, One thousandth of a second, unit usually used to measure time in context computer performance
b	Bit
B	Byte
kB	Kilobyte, 1000 bytes
MB	Megabyte, 1000 kilobytes
KiB	Kibibyte, 1024 bytes, factor of two friendly version of kilobyte, common measure of storage
MiB	Mibibyte, 1024^2 bytes, factor of two friendly version of megabyte
Hz	Hertz, unit of frequency, 1 cycle or event per second

Terms

Bit	Smallest unit of information, a single on/off storage
Byte	8 bits, common storage size unit
Pixel	A regular sized and shaped discrete part of a picture
Voxel	A regular sized and shaped discrete part of a space
Ray Tracer	Algorithm which follows a ray in scene as light would travel in it
Ray Caster	Rendering method which traces a ray to its first intersection with scene
Ray Marching	Method of ray tracing in which ray is advanced iteratively
Frame Time	A time that it takes to render single frame.
Frame Rate	Frames rendered per unit of time.

Notations

x	A scalar
$\vec{x}$	A vector
$\langle x, y, z \rangle$ A vector's components	
$\mathbf{X}$	A matrix
$\frac{\mathrm{d}}{\mathrm{d}t}$	derivative with respect to variable t
$\sum_i$	sum over index i
$\mathbf{a} \cdot \mathbf{b}$	dot product of vectors **a** and **b**
$\lvert x \rvert$	Absolute value of x, per element for matrix or vector
$\lvert\lvert \vec{x} \rvert\rvert$	Euclidean length of a vector
$\mathbb{R}$	Group of real numbers
$\mathbb{N}$	Group of natural numbers including zero and positive integers, 0, 1, .., ∞
$\mathbb{F}$	Group of numbers presentable by IEEE 754 32 bit binary single precision floating point numbers

1 Introduction

Rendering is the process of turning description of three dimensional object or scene into a two dimensional image. Volume rendering is process of rendering volumetric data. Voxel rendering is a subclass of volume rendering, where the volume is stored as samples taken at discrete, regular intervals.

Besides vertex polygons approximating surface geometry, voxels can be used to represent three dimensional models. Voxels are usually cubic discrete volumes of space, containing information about whatever the volume is filled or not and additional properties, such as colour, transparency or texture of space within the voxel's boundaries.

Advantages in generic calculation on graphics computation unit have made it viable to render images using methods other than standard triangle rasterisation pipeline for real time graphics.

Voxels can be rendered either by generating their surface geometry, usually optimised to only surfaces that might be visible to camera and then using rasterisation pipeline for drawing them, or by using ray casting or tracing to directly draw the voxel data.

Most research on the voxel rendering has been on the side of ray tracing, but most of current graphics hardware is optimised for rasterisation. Interest in ray tracing solutions is likely because of better support for more advanced lighting models and transparency calculations. But how do these methods fare against rasterisation pipeline when lighting methods common to current real time applications are required.

In this paper we examine and compare rasterisation and ray casting methods of rendering voxel data for real time applications.

1.1 Research motivation

Aim of this paper is to find ways in which performance of real time programs using voxel graphics as part of their scenes can be improved.

Voxels are efficient and understandable way of modeling objects and materials, which change mainly trough addition or subtraction, rather than bending or stretching. While this allows voxels quite a lot of flexibility, they are usually slower to render than models based on modeling surface geometry with vertices and edges connecting them. This is generally, because voxel models need to be turned into surface geometry to be rendered in normal rendering pipeline or use custom rendering methods which rarely are as optimised.

On the other hand, voxels are quite simple and efficient presentation of model for ray tracing and similar rendering methods. They are already axis aligned on uniform grid and bounding volumes hierarchy can be build quite efficiently using mipmapping, k-tree or similar techniques. These further improve the efficiency of ray tracing options as ray can advance with a longer step in empty areas.

By finding conditions in which ray casting or rasterisation methods are clearly faster than the other, we can optimise whole rendering pipeline to use faster method for each model or by changing the method for all voxel objects in scene when certain

parameters are meet. It is still possible that one method might be significantly faster in all realistic use cases, which might lead to possibility of general recommendation.

1.2 Research Questions

This paper seeks to establish whatever or not the alternative rendering methods might be better for performance in real time applications and if so, under what conditions should they be used over each other.

First, we must establish whatever or not the rendering methods can provide similar enough results to be used as alternatives to each other. This means that the resulting images should be visually indistinguishable from each other and should be within floating point calculation accuracy on data level.

Second, we examine whatever these methods are comparable in performance and what are the main bottlenecks in each method. While the theoretically interesting differences on these methods come from the complexity of rendering equation in terms of picture size and voxel count, more likely real world bottleneck is the memory channel between CPU and GPU.

Finally, we compare the performance of these methods in different scenes to establish which is most powerful method and what are the major factors in selecting a method for each application.

1.3 Scope of this paper

This section defines the scope of this research. The optimisations searched for in this paper aim for improvements in common deferred rendering pipeline. This section also explains why certain rendering features were excluded or why specific platforms were not used for testing.

1.3.1 Deferred shading pipeline

At the moment most real time interactive graphics, which require complex lighting calculations are done using deferred shading. In deferred shading, the scene is first sampled for data needed for rendering, such as screen space positions, normals and materials, which are stored into a texture usually referred as geometry buffer or G-buffer. Then second pass is performed, in which this information is combined to rendered result.

Main advantage of this method is that it allows possibly expensive material and lighting calculations to be calculated only for visible pixels. It is usually further optimised by dividing the second step into multiple smaller steps were only lights affecting specified area are considered and final collection step which composes complete image from these pieces.

Disadvantages of the method are in difficulties of transparency and reflection calculations, which must either be done with only screen space information or rendered on top of the image in a separate pass, and that the texture used for second pass takes lots of memory if there are any complex materials in the scene.

For purposes of this paper, all the differences between different rendering methods happen in the vertex and fragment shader programs of the first rendering pass that generates the G-buffer or in preprocessing done before the first pass.

This is both practical for research as it decreases differences between the different rendering methods as well as necessity to keep the research relevant to many modern real time applications.

Concerns of lighting and rendering equation are left out of this paper. By having only differences in the first pass of deferred rendering pipeline, same lighting calculations can be used for both tested methods as would be used for any vertex based models in the scene. For more complete consideration of voxel based rendering equation, see for example [24].

1.3.2 Examined features

As the paper is targeting real time applications where the rendered model might change at each frame, optimisations which depend on heavy precalculations on the model are left out.

For rasterisation, combination of voxels into larger boxes or precalculating the visible surfaces of each voxel and generating a rasterisation ready surface geometry beforehand would no doubt speed up the rendering, but we would loose the modifiability and other benefits of the voxels. If such optimisations would be feasible for the purposes, then model format other than voxels should be considered. Instead we will do run-time optimisations of voxels by using hierarchical data structures as date representation for our voxels.

Similarly, as results of each rendering method needs to be comparable to others, more complex lighting models are not examined and transparency, refraction and reflection are left outside of the scope of this paper. With deferred rendering these can be done at the latter steps of the rendering without caring for the original presentation of the model.

While voxels have advantages in modeling of heterogeneous translucent objects, these are left outside of this paper's scope as these advantages quickly disappear when using methods of other than full ray tracing for rendering them.

In some older contexts, term voxel is also used refer to certain kinds of height map based ground models. For example VoxelSpace rendering engine used in Comanche games and their recreations use height map and colour texture to render ground. [29] This method would be quite usable for situations where only single layer of voxels from single side of the model is ever visible, as with ground without arcs or caves. These use cases are not general enough for our use of modifiable voxel data, as these requirements would easily be broken by quite simple modifications to voxel data.

We will also only use cubical voxels contained in a regular rectangular grid. While ball shaped voxels, a interlaced hexagonal grid or meta data of surfaces inside the voxels[25] could be used to model natural shapes more accurately, they increase the cost of the rendering and make the comparisons between the rendering methods more difficult. For this reason voxels rendered at this paper will look like voxels and the models will be blocky if the resolution of the model is smaller then the

visible resolution. Other than voxels containing vertex or contour data, there is also significant lack of easily accessible prior research on non-cubical voxels.

As an additional note, this paper will only consider situations where voxels only changing attribute is whatever it is filled or not. This means that possible other rendering information, such as textures don't need to be updated during normal rendering. In this paper we will assume that if any other information about the voxel is required, it is available form texture that can be sampled with voxel's position on the model, and that this data doesn't change between frames, so transferring the data doesn't need to be considered as part of the rendering each frame.

Implementations presented in this paper will work with little tweaking regardless if a single 3D texture is used for whole model or each voxel has their own textures, as long as there is a way of retrieving correct textures and sample them based on position inside of the model. Individual textures for each voxel could be implemented, for example, by redirection 3D texture where texture coordinates corresponding with voxels position in the model contain the index of a texture for that voxel in some collection of textures to sample with the collision position relative to the voxel's corners.

1.3.3 Targeted Platforms

This work only considers desktop computers with dedicated consumer level graphics computing hardware.

While some development has been done in specialised ray tracing hardware [40][36], these are not yet widely available, and with process on the generic programming on GPUs and new graphics APIs giving more low level control over rendering pipeline, it might be that they will not be common in consumer products any time soon. Considering our practical applications for this research, targeting hardware that is not widely available, would not give as useful results at the moment.

In autumn 2018 nVidia released series of graphics cards with ray tracing optimised generic calculation cores and machine learning based ray tracing postprocessing tools [33].

As the research was already underway, these are left outside of this paper's scope. The postprocessing smoothing is unlikely to be useful for target purposes, but the generic processing cores optimised for ray tracing might offer advantages over examined rendering methods. This should be investigated in further research.

Most modern consoles also have graphics processing units very similar to these available to consumer computers, so they are not considered separately. Mostly everything that is found to be true for modern desktop computers, should also apply to console hardware, even if the requirements for optimisation are even tighter. Because hardware differences between consoles using same version of the software are minimal, more hardware specialised optimisations might be available, which are not as easily duplicteable on personal computers with higher variety of hardware setups for any version of distributed executable.

This thesis will also ignore mobile platforms. Mobile platforms might have differences on what level of customisation and generic computing their rendering

pipelines allow and most have integrated graphics processing coprocessors on CPU or in case of Tegra[1] chips, more of a GPU with integrated CPU. Such processors likely face different kind of bottlenecks and might be of interest for future research.

Exact configurations of hardware and operating software used for testing are documented in the methods part of this thesis.

2 Background

This section examines the historical and theoretical background as well as previous research in the area of subject.

2.1 History of the topic

2.1.1 Time line of Programmable Graphics Processing Units

Implementations discussed in this paper rely heavily on programmability of modern GPU and thus it would be useful for reader to understand outlines of how technology has reached current state. One of the main things to note is that new features are usually first moved from CPU to external devices as quite case specific hardware solutions, and then latter transform into more generic devices, with more features and more programmable behaviour.

Specialised graphics progressing hardware started as either integrated chips or as extension hardware to offload specific parts of graphics computations from CPU. [31]

First specialised graphics chips that where not completely separate super computers, where arcade machine video chips in 1970s, usually quite specific for each game. [20]

First part of graphics computation to be offloaded from CPU was the functionality transfer and compile image data. For 3D applications this meant rasterisation of simple primitives to frame buffer and in 2D copying masked images to frame buffer. [31]

Video Shifters where chips handling timing signals and transferring frame data received from CPU memory to serial data stream for video output. [9]

In 1980s some CPUs where accompanied with blitters, which where specialised in moving large amounts of data from parts of memory to others. They where named after Xerox Alto processors bit blit operation. In graphics processing they where used to drawing bit map images on top of each other. From standard movement operations blitters differed by their ability to handle transparency with conditional overwrite and multiple pixels per byte formats by masking and shifting. [35] [14]

Released 1984, the IBM Professional Graphics Controller was one of the first video cards designed to handle both 2D and 3D image rendering. It was a board of Intel 8088 microprocessors that could take over most of video processing from CPU, but because of its high price and limited software support was not widely used. It is significant in signalling transition from specialised integrated chips to separate processors in graphics processing. [31]

1989 Silicon Graphics Inc. introduced OpenGL, first widely used platform independent API for graphics programming. OpenGL also formalised the concept of graphics pipeline for rastering 3D vertex geometry. [31] OpenGL appeared first as a professional graphics API, but originally suffered from performance issues which allowed competing Glide API to be the dominant force on the PC market until the late 1990s.

In 1990s general purpose 2D graphic coprocessors died out in favour of separate chips, which usually where specialised for 3D operations, such as transforming,

clipping and specific lighting calculations, which give them name T&L chip, Transform and Lighting chip. Such hardware was mostly found on arcade machines and 5th generation video game consoles.

In 1993 SGI released RealityEngine, fixed pipeline graphics processing unit that could handle the latter stages of the graphics pipeline while still leaving the earlier phases for the CPU. [31] In late 1990s SGI chips where mainly targeted for workstations while other manufacturers brought cheaper GPUs to consumer computers, among them the current primary manufacturers ATI and nVidia.[31]

Before late 90s graphics processing components were optimised for sequential throughput and could usually only handle single pixel per clock cycle. This changed in late 90s when manufacturers started including multiple parallel pipelines.

In 1997, Fujitsu marketed Pinolite as the first 3D geometry processor for personal computers. It was an extension chip capable of performing floating point operations for large number of 3 dimensional vectors simultaneously and while not actually faster than higher tier CPUs available for home computers of the time, it freed the processing time for other tasks.[17]

The first hardware T&L GPU on home video game consoles was the Nintendo 64's Reality Coprocessor, released in 1996. It is likely first commercially available graphics processing unit that has each of the pipeline stages it takes over from CPU programmable. [41]

First consumer level graphics cards to implement whole 3D rendering pipeline first appeared right at the end of 90s. These pipelines where completely fixed, allowing only limited types of input data and configuration variables, with no options for code execution or accessing any intermediate data.

Released in 2000, PlayStation 2 had separate vertex processor that supported complete implementation C programming language, giving it full CPU level of programmability. [30]

In 2001, nVidia GeForce 3 was marketed as first programmable graphics card, and likely was that for consumer PCs. Its NV20 chip supported vertex shaders with up to 128 instructions and pixel shader with 4 texture look up instructions and 8 blending instructions. These shaders where written in hardware specific assembly level language. The pipeline was otherwise fixed and these instructions where sent along the rendered data. [8] [38] [31] [27] [7]

In 2002, new cards allowed for fully programmable vertex and fragment shaders as well as some configuration of input and output operations. Still both shaders had their own specialised hardware, and their instruction sets where somewhat limited. [31]

2003 saw the introduction of DirectX 9 and new GPUs, which allowed full floating point arithmetic, multiple rendering targets and advanced texture processing. General processing on GPU was first handled trough proprietary languages specific for card manufacturer or platform or by abusing programmability of fragment shader for compute shaders.

Also in 2003, Cg was introduced as C-like language and systems for programming vertex and fragment streaming processors in GPUs. It differed from earlier attempts of higher level shading languages like RenderMan by not being application specific.

It could work with both major graphics APIs of the time, Direct3D and OpenGL. [30] Other early examples where Brook and Sh from 2004. [31] Soon after, API specific shader languages, HLSL and GLSL (initially known as glslang based on the reference front end) followed. [13]

After OpenGL and DirectX started supporting shaders written on relatively higher level of abstraction and hardware was capable to execute almost any code on vertex and fragment shaders, fixed T&L hardware become unnecessary and the GPU where no longer limited to processing graphics.

By 2006 vertex, geometry and fragment processing units where unified into single fully programmable processor that could handle all of them. The pipeline that had been important part of graphics processing design became merely a abstraction useful for software side. [31]

Besides incrementally improved memory bandwidth and parallel processing power, recently the GPU development has seen introduction of more generic programming performance, even separate generic programming cores, and features optimised for ray tracing or neural network systems.

2.1.2 Previous uses of voxel graphics

It would be difficult to understand the current state of voxel graphics if one is unfamiliar with how and where they have been used in past. This section aims to provide some outlines for previous uses of voxels.

There currently are two main areas of use for voxel graphics, scientific visualisation and entertainment. In scientific visualisation voxels are used to render three dimensional scalar or vector fields. Common applications include topography scans and magnetic resonance imagining. Visualising medical scans is one of the most important areas in scientific visualisation. [24] Other, entertainment, applications of voxel rendering are mostly on computer games and animations, in which voxels have been used to represent materials and objects that can change trough addition and deletion.

There are some differences between these fields. In scientific visualisation only single model is usually rendered at time, while entertainment might require hundreds or thousands of models on screen at time. Scientific models usually have much larger amounts of data for each voxel and the aim is to accurately relay this information to viewer. In entertainment uses, the voxels usually only have the minimum amount of data that they require for their intended purposes and the aim is to render the data in a way that looks visually pleasing or realistic. Both fields have use for real time applications of voxel graphics, but in scientific visualisation updates to voxel data at run time are less common and updates to rendering parameters other than transforms and lighting are more common.

In entertainment, first examples of voxels where found on terrain rendering of games like Comanche: Maximum Overkill from year 1992. These early examples where in most cases actually height maps, though the rendering methods where similar to what was used for full voxel models at the time. Command and Conquer series used voxels for animating most vehicles of the games. Most common use cases

in games for voxels are currently in destructible terrain and other environment. In most instances of voxels used in video games, they have used software rendering, which means that the calculations mostly took place on CPU side and GPU or other graphics acceleration hardware was not utilised.

On side of movies, voxels have been mostly used for rendering volumetric smoke, dust and liquid effects, because voxels represent interiors of heterogeneous materials better than surface geometry and are more efficient format for ray tracing than particle effects. Lately voxels have also been used for solid objects when extreme level of detail is required.[12] Use cases in movie industry are mostly on the side of offline rendering, and run-time rendering is only used for previews during editing.

Voxels have quite a few algorithms with which they can be rendered. Common algorithms fall into categories of ray casting, splatting, cell projection and multi-pass resampling. The categories are based on the order at which the data is handled if algorithm is executed on single thread. There are also few which are combinations of multiple methods such as sheer-factorisation. [24]

Ray casting algorithms take a view direction ray for each pixel and sample the voxel data in its direction, splatting algorithms go trough voxel data and render each to screen as its own object, cell projection algorithms transform the voxel data into vertex data and draw it as polygons and multi-pass resampling works by sampling voxel data at fixed distances like it was a set of two dimensional textures on planes orthogonal to view direction.

One of the traditional ways of extracting polygonal data from voxel model is marching cubes algorithm, which approximates possible isosurfaces based on values at each of the cubes corner. Given the eight points with specific values showing if they are inside or outside of the surface, there are limited number of ways a somewhat smooth surface can go between them. [28] Given the simple model of filled or empty voxels used in this paper, the algorithm would take eight voxels at time, check which are filled and empty and then place triangles to place surfaces so that they pass between filled and empty voxels.

Original algorithm had troubles of creating topologically correct surfaces and leaving holes in few cases, which were later corrected by increased set of possible surfaces and heuristic on deciding which to use based on bilinear interpolation between the sampled values [10].

The method is most commonly used to render scalar fields in scientific visualisation, when seeing certain value boundaries as surfaces is beneficial. The name comes from the fact that the sample area is a cube and once it has tested one set of eight voxels, it 'marches on' by replacing four of the voxels with four new voxels. [28] While marching cubes can lead to a smoother presentation of voxel model, it is too expensive to be used in run-time creation of surfaces for any larger data set as complete set of surface geometry would need to be updated to GPU after each update.

Earliest methods used to render voxels directly started by sampling voxel data from far plane of the camera and rendering these as a points or vertical lines. The algorithm then repeatedly took a new plane slightly closer to the camera and repeated the sampling process either overwriting pixels in the existing image or blending new samples to existing image if there was transparency in the sample.

The voxel data could also be sampled for two dimensional textures along these planes and then used as textures for rendering geometry corresponding with the sampling planes with the sampled textures trough graphics pipeline.

Next step of volume rendering was turning this method from back-to-front order to front-to-back order. Rendering the near planes first allowed decreasing level of detail on farther planes and terminating the rendering of the further planes once the colour of the pixel has reached specified level of opacity. This required additional numbers for the distance of last effective voxel and/or combined opacity of the voxels already sampled for each pixel. [29]

Because each of the pixels on the image would have to be tested for each plane used for sampling, it made sense to instead start ray from each pixel and sample the scene along the ray until the ray reached some specific termination conditions. If the sampling was done at predetermined positions along the ray, such as with even distance between each of the samples, it was quite likely that the ray would pass trough a corner of opaque voxel without sampling it. This could be partially avoided by sampling the data more often along the ray, but such high number of samples would make the algorithm prohibitively expensive. Limitations of early fragment shaders required ray casting to be performed in multiple steps, each blending results of last step with sample from tracing the ray one step forward. [23] Usually the samples where from predetermined planes trough the model.

Voxels have been also been used as an optimisation for ray tracing vertex based surface geometry from at least since 1980s, and because of this there are quite a few optimised representations and traversal algorithms for voxels available. [3]

2.2 Earlier research on topic

This section further examines earlier research on the topic and attempts at improving the performance of voxel rendering on GPU hardware.

Splatting is a way of rendering volumetric data by presenting each voxel as a separate 3D object having a 2D footprint that is projected on the target surface in the back to front order. [39] This can utilise the point rendering pipeline found in normal rasterisation optimised graphics computing hardware. Splatting benefits greatly from view space culling, and can handle transparency quite efficiently, but for fully opaque objects, the need for rendering inner voxels as well increases cost.

Biggest problem in the method is requiring the voxels to be sorted in back to front order, though the sorting can be avoided when there is no transparency and the visibility of the splat can be determined by depth testing. Splatting differs from the our rasterisation method in that the primitives that are rendered in splatting are simpler than the polygon cubes we use.

It would be usable option if the voxel model was of such accuracy that 2D footprints would not make visual errors on flat surfaces. Extraction of surface features such as normals, which we need for the lighting pass might also be more difficult.

There has also been research on compression of bounding volume hierarchies, some of them based on specialised or mobile hardware. (see [37] for example). Because type of voxel data examined in this paper, there are no benefits for storing positions

or sizes of the nodes in the hierarchical presentation as these are implicitly available from the tree structure. These methods are more useful for ray tracing vertex data, but might also be useful for containing multiple voxel objects with different rotations on same scene.

Parallelism of creating hierarchical container structures for scene geometry has also been researched (for example [22]). While these are meant for vertex data, similar radix based algorithm might improve point voxel tests needed for voxel rendering and editing the data. Octree implementations tested in this paper include a depth first stored voxel data, so the usefulness of memory locality from such ordering will be tested along other features.

In 2009 Crassin et al. [12] managed to achieve interactive rendering times for volumetric data exceeding size of the GPU memory by allowing the GPU renderer to signal CPU about missing parts of the voxel data, so that only need parts of the voxel model have to be stored at GPU memory at time. By having the voxel data stored as a "bricks", small grids of voxels with a size of 16^3 or 32^3 voxels, the most memory intensive part of the model could be updated as small packages. The bricks where stored on a octree where they presented the actual voxel data when stored at leaf nodes or lower level of detail combination of nodes children when stored at non-leaf nodes. If a resolution of a brick on higher level of a node was close to a size of a pixel it might be rendered to, the ray traversal could be done trough this lower level of detail brick, which required fewer samples than if leaf level bricks had been used and as added bonus provided efficient anti-aliasing at the same time. [12]

This method is quite suitable for uses where voxel size is often smaller than a single pixel, when more complex data is stored for each voxel or when more complex lighting model is used. The brick system might also be useful for simpler voxel data by providing memory coherence benefits for nearby rays or to update parts of the voxel model when voxel data changes without touching other parts as the changes are usually concentrated to small area at time. Unfortunately, the method also wastes lots of memory when the voxel data is not truly volumetric[25].

Laine et al.[25] presented multiple methods for compressing voxel representation of a scene, accelerating ray casting and improving the quality of generated image using said representation. Their main contributions are introduction of contours, which allow for more accurate representation of flat surfaces not aligning with the primary axes of the voxel model. Another important part is the voxel data structure, which is implemented as sparse octree with 64 bit node. Nodes are stored in memory pages of quite small size, and contain information about where its children are stored and whatever the each of the children are leaf nodes, non-leaf nodes or empty, as well as information about the contour describing the surface on the nodes area. [25]

In 2014 survey conducted by Beyer et al. [6] it was found that there are quite a many names for methods where once the size of a voxel on screen is small enough, lower level of detail data is used for traversal and sampling the voxel data. Basic idea is that if closer to root nodes in hierarchical data structure representing the voxels contain approximation of the data in their child nodes, then the parent can be sampled instead of the children. This avoids flickering caused by ray occasionally passing trough empty part of the model and occasionally stopping at one of the filled

children, when user doesn't see even pixels worth of movement in rest of the model. Most of the other technologies mentioned on the survey are only useful once the voxel data gets too large to be stored on GPU.

In our octree implementations this level of detail optimisation could be used to terminate ray traversal on non-leaf nodes if whole node would fit inside the pixel cone for current ray, even if they are only partially full. For rasterisation implementation, using such method would be trickier as the generation of surface geometry would need to be aware of the angle resolution of target image and the distance of the model from the camera.

2018 Assarsson et al.[4] presented method of transferring hierarchical voxel representations into directed acyclic graphs, examined its benefits in compressing sparse voxel octree representations. The method is quite simple in that if two subtrees are identical, other can be removed and any pointers to it directed to other. Their paper is also one of the few which consider the possibility of animation, mostly by reusing subtrees between the frames. While finding of the equal subtrees is quite expensive, even with a efficient method based on subtree hashes, this allows rendering whole scenes which would not normally fit into GPU memory with in-core methods. They also examine using similar technique to store shadow maps. [4]

Ray tracing on GPU is quite difficult to approximate from theoretical intensity of the computations so another approach for estimating and comparing efficiency of algorithms is to simulate ideal work distribution and memory accesses for given hardware and algorithm. While simulations and tests made on ray tracing efficiency by Aila and Laine [2] were made for vertex data, their observations about node traversal order might hold for nodes containing voxels as well.

Unfortunately there appears to be little relevant information on rendering dynamic or mutable voxel data.

2.3 Computational intensity of rendering methods

This section examines the computational requirements of the voxel rendering methods and to form hypothesis on the comparative efficiency of these methods.

In rasterisation method we calculate screen space information for each triangle, which might be visible and then for each pixel check if it is within the bounds of that projection of the triangle. If the pixel is within the triangle and the corresponding point on the triangle is closer to camera than any previous point written to that pixel, the pixel is updated by overwriting it with data sampled from the corresponding point or interpolated from the vertices of the triangle. At its base this process for image with n pixels and voxel data with m^3 voxels, takes $O(n \times m^3)$ pixel information update operations, but in practice early visibility testing and other optimisations greatly reduce the number of operations required.

When using rasterisation methods for voxel data we also need to transform the voxel data into surface geometry. As the rasterisation is heavily impacted by the number of objects drawn, instead of sending surfaces of each voxel as a vertex cube with filling information or just all filled voxels, it is beneficial to use a tree structure, such as octree to combine completely filled subtrees as single cube. This operation is

hard to measure because it is not fully dependent just on how fine scale the voxel model has, but also on the structure and fill balance of the tree. At best case scenario, the model is completely full or empty, and only single node must be processed. At worst case scenario, one of the eight the voxels for each tree's node is empty and the algorithm needs to traverse all $\frac{1}{7}(8m^3 - 1)$ nodes as well as create geometry for $\frac{7}{8}m^3$ voxels. Memory footprint of surface geometry would still be smaller than for fully filled model without any optimisations, as that would require creating geometry for m^3 voxels.

When rendering is done using the ray casting method, we only require rasterisation of few surfaces. Actually just single quad or triangle covering the screen would work, but by using a single bounding volume cube, we get additional model and word space information from shader pipeline and can benefit from some minor optimisations. While the ray casting requires only one cube to be drawn, computations on each of the pixels within these surfaces is much more complex, requiring sampling the voxel data until a collision between the ray is found. As the data is sampled on a line passing trough the voxel model with m^3 voxels at most $3m$ voxels need to be sampled. Thus the computational cost rendering n pixels with the method should be around $O(n \times 3m)$. Hierarchical methods might further decrease this but depending of the model might also cause additional overhead from having to sample lower level nodes.

Given these complexities it would be easy to claim that for any significant size of voxel data, the ray casting method would be faster, but using it trades many of the highly efficient optimisations found on the modern GPU graphics pipeline. Complexity of per pixel operations might mean that on large image resolutions, the rasterisation has and edge over the ray casting method.

It should be expected that rasterisation method would benefit much more from good acceleration structure and scale more heavily on the resolution of the voxel model. Likely hidden details are also more costly for the rasterisation approach. On the other hand, ray casting method doesn't benefit from early culling based on depth, so rasterisation might gain an edge on situations where it is partially hidden or out of view.

There is also great likelihood that most of the rendering operations are bound by memory bandwidth between the CPU and the GPU rather then calculation speed on either end. In these cases the rasterisation benefits more from the octree acceleration structures, as it doesn't have to transfer the tree structure to the GPU. On the other hand plain voxel filled or empty state can be stored in much smaller space than surface description of the same voxel model.

3 Implementations and Testing Methods

This chapter examines how the rendering methods were implemented and how they were tested. First part examines generation and storage of the voxel models. Second part describes the implementations of the ray casting and rasterisation methods. Third part examines the methods used to accurately time different parts of the program executions and the fourth part describes testing setup used to benchmark the implementations against each other. The final part of this section enumerates the different test scenarios.

All implementations were written on C++ in a way that they could be compiled for Windows and Unix operating systems and with OpenGL 4.4 core profile used for rendering. GLFW3 was used for window management and IO handling, Glad for handling OpenGL bindings and GLM OpenGl Mathematics library for vector and matrix mathematics on the CPU side. [19][21][18]

3.1 Scene generation

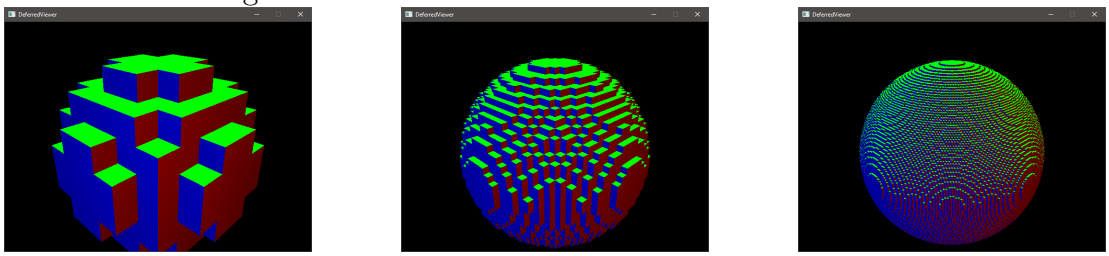
This section describes the data that is rendered and how it was produced.

Most artist tool chains are geared toward creating and editing surface geometry and textures and thus there are not as many voxel models easily available to be used for testing. It also appears that there is no good standard format for storing voxel data, so loading found models would not be trivial.

To get suitable test data, voxel models can be generated from surface models or other descriptions. Surface models could even be used to generate voxel data from vertex data in rasterisation pipeline at real time rates[15] [16] [34]. For purposes of this paper, offline generated voxel models were used on testing.

Two methods of generation were used for the generated models, first method generated filled balls of different sizes by setting voxels within specified distance from the centre of the model to be filled. The ball was chosen as it is simple but also gives good visual indication of how well model of given size and resolution can represent natural or curved surfaces. Filled ball also has a an interesting property for the acceleration structures as it has quite even amount of both short and long paths from the root to completely filled or empty node. Figure 1 shows the model rendered at different resolutions.

Figure 1: Ball model at different voxel resolutions

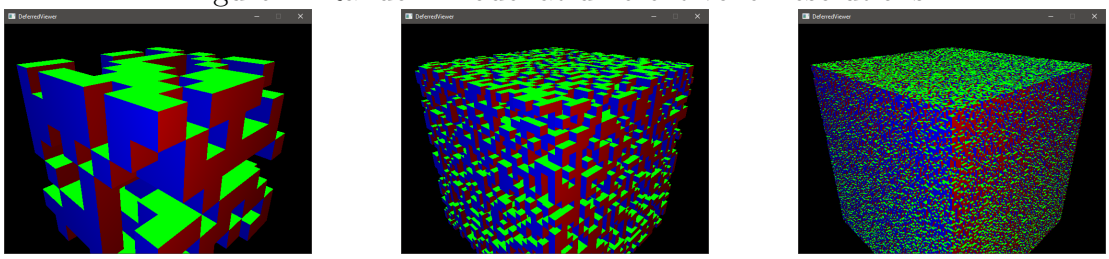


8^3 32^3 128^3

Second model was created by randomly deciding for each voxel if it should be

filled or empty. This allows for adjusting the filling rate of the voxels in the model as well as gives an unbiased data point for comparing the different methods as methods can not be directly optimised for data that doesn't have known patterns. It also gives a good representation of partially filled model, as randomisation gives both small clusters of empty or full voxels as well as areas containing both somewhat evenly. The used model had only small clusters of filled or empty space and its exterior had large number of small visible details. The interior was similarly complex, making the model difficult for octree storage as in most cases the full level of detail was needed. For 50% fill rate, only 1 in 128 nodes on the level right above the voxels would be empty or full. Figure 2 shows the randomly filled model, with 50% fill rate at different model resolutions.

Figure 2: Random model at different voxel resolutions

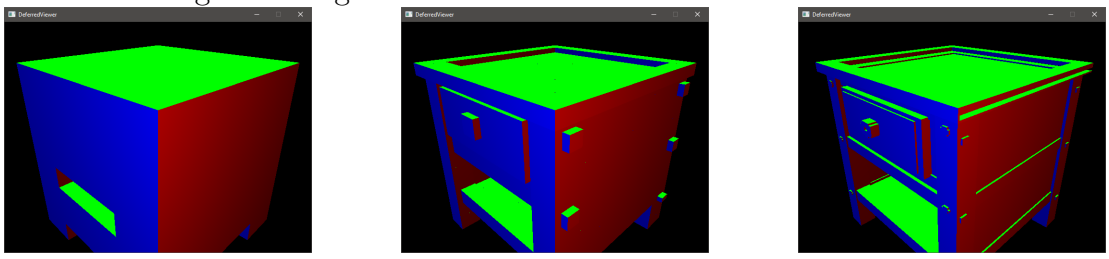


8^3 32^3 128^3

Additionally, a model created from vertex data using surface filling was used for testing. Fastest methods of creating voxel data from vertex geometry are based on rendering a depth map from different direction with orthogonal projection and combining this information, but these methods can't account for empty space inside the model. Because voxelisation phase is not within the scope of this research, a simple brute force method was used. For each triangle, a bounding box was calculated and then voxels overlapping with the bounding box were set to be filled.

The selected model is a nightstand with a drawer. The model has both visible and hidden empty spaces inside in it. The filled areas are quite narrow and require more levels of octree to be presented accurately, but the large empty areas should benefit form empty space skipping. The empty space also means that for the linear array presentation and rasterisation the model has relatively small number of filled voxels. The figure 3 shows the model rendered with different voxel resolutions.

Figure 3: Nightstand model at different voxel resolutions



8^3 32^3 128^3

To simplify the testing process and to avoid overhead from the other parts of rendering, the simplest testing scenes only contain a camera, a single voxel model and a directional light source.

Each model is generated at 7 different levels of accuracy, from 2^3 to 2^9 voxels per edge. The upper limit was chosen because the there was an integer overflow on the random model when trying use accuracy of 2^{10} voxels per edge.

Camera and the model are positioned so that the model is completely or almost completely visible and that covers most of the screen space. In test scenes, where nothing else is mentioned, the camera is positioned on 45 degrees to side and 25 degrees up from the model.

3.2 Voxel storage

Format of storage for the voxels is one of the main optimisation points for both methods. Storage format affects both time it takes to update the voxel data and time it takes to transfer the updated data to GPU. It can also be used as an acceleration structure for empty space skipping in ray tracing and to combine fully filled areas for the rasterisation method.

As the memory requirement increases linearly to number of the voxels or cube of number of divisions in one dimension, accurate voxel models require significant amounts of memory. To store only the filling status of the voxel, only single bit is needed and byte can hold eight voxels, but if the model has been divided to voxels so that it has resolution of 256 voxels per direction, we already need 2 MiB of storage. Given that in many real use cases the voxel data has to be accompanied with a quite large 3D textures for other information the renderer requires of the voxels, it is important to not use too much of memory for the voxel structure. Given that the voxel information is updated for each frame, it is also important that the memory requirements don't exceed the available bandwidth.

Simplest implementation tested in this paper stores the voxels in scan line order. Voxels, with same x and y coordinates, are in order of rising z coordinate, then followed by another row with coordinates x and $y + 1$. This method is used for its simplicity, but a more complex linearisation order, such as indexing with Morton encoded coordinates [32], might allow for better locality of data[5].

The storage for this scan line linearised data is a array of 32 bit integers. Position of voxel is in the model transformed into index in the linear representation by multiplying position along x-axis with square of the voxel model resolution and position along the y-axis with the voxel model resolution, these are then added to position along z-axis. This result is then divided by 32 to gain index in this array and the remainder of the division gives us the offset in the given integer.

The voxels can also be stored in a hierarchical tree structure, such as octree. The octree can be stored on memory in multiple ways. Because we need to be able to send the octree data to GPU we need to store it in a continuous memory block. Octree can be represented in a memory as a dense tree, a sparse tree or a graph.

In dense tree approach, every node and leaf voxel is stored in the memory. This method doesn't save space over storing just the voxels, but it is useful as acceleration

structure for both methods. For example, a model with resolution of 16 voxels per dimension takes 512 bytes of memory to store the voxel data but 1168 bytes to store voxels on a tree structure.

The nodes and the voxels can be stored in the memory either in depth first order or breath first order. Depth first order allows updating any subtree as overwriting a continuous part of memory, so it handles partial updates between CPU and GPU better, while breath first order might improve the locality of data required in the higher levels of the tree which are accessed more often than the leaf nodes and the voxels in ray traversal. Because the position of the node in the tree and its location in the memory can be inferred from each other, the nodes need to only be 2 bits in size, indicating either full, partially filled or empty node.

In sparse tree, children of a node are only stored if they don't have same values. This method saves space when there are nodes close to the root that are completely empty or filled, but because the memory location of a node and its location on the tree don't relate to each other, each node needs to contain memory addresses or offsets of its children and if stackless algorithm is required, also its parent's address.

If breath first order is used to store the nodes, only first of the children needs to have its address stored, but for depth first order either addresses of every children needs to be stored in its parent, or then each node must also contain position of its next sibling. The address of children might be either offset from current node or from start of the array. There is also a hybrid order obtained by storing all children of a node as continuous block but selecting next block to store in depth first order. This method preservers the benefit of only needing one child pointer per node, but will also give locality benefits of depth first order. For larger trees, storing it in smaller parts, and having the address as part number and index in that part might be beneficial for updates.

The sparse tree versions of our rendering methods use the hybrid version of this data structure. For each node we store the index of its first children. As root is not children of any node, its index, zero, is used to indicate empty subtree. The largest unsigned integer value representable by the storage unit for each node is used to signify that sub tree is full. This leaves all the values between these two available as indexes of the the tree nodes, meaning 32 bit nodes can represent octree with 11 layers of nodes completely but with sparse presentation, even deeper trees are possible if there are any fully empty or filled areas.

If we used offset from current node to its first children, we could store more voxels for same size node, but in breath first storage order the the additional complexity is not really worth the gained addressing space on models used for the experiment.

Because of changing a value of voxel that is not already a node in the sparse tree requires new nodes to be created, this operation is quite expensive. Creating a node requires us to move the nodes after the addition position forward to make room for the new subtree and we have to update any pointers that point to the moved area, which includes siblings of the any changed voxel and anything after the moved part.

Similarly, when node changes in a way that its parent now has all of its children with a same value, it must also be changed to match that value. Now the children are no longer needed and should be removed to minimise memory footprint and maintain

the order, but any nodes pointing over the removed nodes need to be changed to match the new indexes, and the nodes after the removed nodes need to be moved toward the beginning of the memory area and changed if they are pointers. If we don't remove the nodes, we will either have to keep track of this empty space or create new space at the end of the array when the node's children would be added again.

This is one of the places where using offsets can be helpful, as only the nodes pointing over the changed area, which are latter siblings of the root of the changed subtree, must be changed, rest can just be copied to their new places.

A graph is an extension of sparse tree with equal subtrees joined at their root. This saves lots of memory when there are identical subtrees at different parts of the tree, and can be effective optimisation for huge trees with regular or repeating patterns. Unfortunately this means that ordering of nodes in memory is very complex after first few nodes from the root. Construction of the graph and checking for duplicate subtrees is also quite expensive and updating a single voxel leads to having to make a copy of the subtree before the change, to avoid changing other subtrees that use the same data. The update might also lead to creating subtree that is identical to other subtree and these should be joined periodically to avoid losing the benefits of the graph. With the construction and update costs so much higher this is unlikely to be suitable for real time applications with mutable voxels and is not implemented in tests presented in this thesis.

In this paper, 4 storage methods for the voxels have been used, linear array for voxel data, dense depth first octree, dense breadth first octree and sparse hybrid order octree.

To compare results without overhead of texturing only filling data is used in the test renderer. Information needed for texture sampling is generated as part of the rendering methods, but no sampling is done. If textures or other voxel information would be required, the sampling information can easily be used to check these from additional textures. This means that the method can be easily adapted to situations where each voxel has its own textures as well as in places where the whole model has single texture with resolution lower than that of the voxel model.

Since voxel models are not generated on run time in applications that are within the scope of this paper, the generation itself is not performance critical. Instead the critical parts of the data structure are the time and memory required to render it and how quickly small edits can be done to the model.

Table 1 shows that with quite a large and coherent model of filled ball, the sparse octree takes significantly less space than the other methods. The dense octrees take bit more than a double of the space required by the linear array representation and don't depend on the content. The sparse octree on the other hand does depend on data and the size of the random model is 20 times larger than that of ball with doubled resolution and thus eight times more voxels.

3.3 Rendering Implementations

This part describes implementation in each of the rendering methods.

Table 1: Size requirements of different voxel representations.

Model	Ball			Random	
Voxel resolution	8	32	1024	32	512
Linear Array	68 B	4 100 B	128 MiB	4 100 B	16.0 MiB
Depth first octree	148 B	9 364 B	293 MiB	9 364 B	36.6 MiB
Breadth first octree	148 B	9 364 B	293 MiB	9 364 B	36.6 MiB
Sparse octree	1 828 B	27.3 KiB	28.7 MiB	145 KiB	581 MiB

3.3.1 Rasterisation of generated triangles

The rasterisation pipeline relies on the use of instanced rendering to render multiple cubes with single draw call. First the a cube model and shader program are loaded to the GPU, then an array containing positions and size of filled voxels or octree nodes is sent to the GPU and instanced draw is called.

Initialisation of the shader programs and environment as well as the creation of the cube model work as in any modern rendering application. The vertex data for the cube requires normals for each side as well as the corners set at model space coordinates $\langle 0, 0, 0 \rangle$ and $\langle 1, 1, 1 \rangle$. UV coordinates can also be supplied in vertex data, but can also be taken from the other data if needed.

To render the model for each frame, we need to collect the filled voxels and their positions from the voxel storage to a buffer of model space positions and scales.

For the linear array this is quite simple but expensive part, we iterate over the voxel data and if the voxel is filled, append its location to a buffer. For the other storage methods, we need to traverse trough the tree starting from the root node and add new elements to the buffer when ever a full node is found, the scale of the voxel is dependant on the depth of the voxel in the tree and its position from the path that was taken to find it. This requires 4 numbers per voxel, or in case of the octree, for each fully filled node. These number need to be large enough that they can accurately describe the voxels position within the model. For model with resolution of 256 voxels per dimension, 8 bits would be enough for each model. Any reasonably sized voxel models should be representable with 16 bit numbers. In the implementation used in this paper a 32 bit floating point numbers were used.

This part can be parallelised by splitting the model to regions of continuous memory or subtrees, creating a buffer for each of these parts and then joining the parts together once they have been filled in different threads. With single thread implementation, the drawing with this method took around 6.6 ms for a linear array stored ball with resolution of 64 voxels. With 4 threads the same process took 6.2 ms, so there is room there but not much. On other models the parallelisation had similar results, so if other CPU cores or hyper threads are idle, they can be utilised for this. The tree structures were parallelised based on the 8 child nodes of the root node. Given the tree structure and linear end result, use of computation shaders or massive parallel methods would likely not be beneficial.

After the buffer of positions and scales have been generated, they must be passed to the shader program. In modern OpenGL this can be done trough uniform types or instance vertex attributes. Uniforms are a simpler method, but only about 500 instances could be send per draw call. This worked quite well for smaller data resolutions, but became excessively expensive on larger resolutions. Another approach is to use instanced vertex attribute arrays. Rabidly changing buffer data has its own problems, as the operations are not done synchronously but their use decreased the time needed for the 64 voxel ball from around 6 ms to around 4 ms. At this point the drawing of the linear array ball of higher resolution still takes too long to measure, but the octree versions are in real time frame rates.

While using ray casting, we have to send the voxel data to GPU, with rasterisation method we have to send the offsets and scales. Table 2 shows the sizes of the completed instance data that must be send to GPU. All the different octree options tested in this paper end up sending the exact same data. The combination of full subtrees into single cubes saves significant amount of memory for coherent data.

Table 2: Size of offset and scale data sent for instanced rendering.

Model	Ball			Random	
Voxel resolution	8	32	512	32	512
Linear Array	4 480 B	270 KiB	1.05 GiB	258 KiB	1023 MiB
Octree Versions	3 584 B	43.0 KiB	12.5 MiB	256 KiB	1017 MiB

Vertex shader is quite simple, it takes the cubes vertex position, normal and UV-coordinates and applies the instance offset and scale to them, transforms positions and normals to world space and projection space, and then stores them to output variables. For the UV coordinate, we expand it into 3D position within the model space, so that it can be used to identify both the voxel and the position on its surface. The graphics pipeline then performs primitive assembly, view space culling, early z-buffering and other operations and optimisations, which leaves us with possibly visible fragments on fragment shader. The fragment shader is equally simple, and only transfers the position and normal data to G-buffer. If the voxels were textured, this would also include the fetching of the texture data, which can be found based on the UV coordinates.

3.3.2 Ray casting voxel data

In ray casting method we use the rasterisation pipeline to render single cube scaled and oriented to match the outer bounds of the voxel model. The vertex shader simply applies model, world and camera transformations and stores the parameters we need to interpolate for each pixel. Fragment shader performs the ray casting for pixels which are within the bounding volumes projection to screen space.

Ray is defined by equation $\vec{r} = \vec{s} + t\vec{d}$ where $\vec{s}$ is the starting point of the ray, $\vec{d}$ is vector in the the rays direction. t is a positive scalar that goes from zero to infinity,

defining each of the points along the ray as $\vec{r}$. [26]

There are two options for starting position of the ray. If the fragment shader is invoked for camera facing face of the bounding cube, we start from the surface of the cube and move in the direction from camera position to surface position.

If the fragment is for a surface facing away from the camera, we check if the camera is inside the voxel model and start from the camera position if it is. If the camera is outside of the model boundary, the same pixel will be tested by front facing face fragment, and can be discarded, unless the front facing bound is between near plane and camera position, in which case we only have the back face for rendering, but to avoid artefacts and inconsistency, we have to calculate respective position on the front face and handle the fragment as it was front face fragment.

To simplify later calculations, we transform the rays starting position and direction from world space, in which they are received in the fragment shader, to model space. Model space of the cube has its coordinates configured so that that they fall to range $[0-1]$ and can be used to query the voxel data or textures.

After the ray starting position and direction has been selected, we must find place where it collides with filled voxel. When finding a collision between ray and another object the analytic way is to solve the t for group of equations where $\vec{r}$ is also defined as a point of another object.

For example for ray–ball intersection we would solve equations $||\vec{r}-\vec{c}|| = r, \vec{r} = \vec{s}+t\vec{d}$ where $\vec{c}$ is centre of the ball and r its radius. If there are multiple objects, the pair of equations is solved for each object and smallest value of t is used to find the first collision point.

In case of our voxel models we avoid unnecessary equation solving by checking collisions in order the ray travels trough them until we find a collision with existing voxel or pass out of the voxel models bounding volume. Because the voxels are in uniform grid we only have to test which of the three cardinal planes corresponding with the current voxel's back boundaries the ray first passes trough.

In uniform grid, such as when using the linear array storage for the voxels, we can also reuse two of the distances to boundaries, and update the one that was closest with a constant describing distances along the ray of two boundaries in that cardinal direction.

Main optimisation point and trade off of this method is between the memory use and ray traversal speed. 3D array or texture is simple to sample for voxel data and the travelling can be done with simple fixed steps, but large areas of empty voxels will require sample for each of them. The another option is to store the voxel data in a tree structure, which allows us to skip larger empty areas at cost of bit more complex calculations for next boundary and more memory accesses required for sampling a voxel, especially in partially filled subtrees. Earlier research suggests that the using of tree structure is faster. [26]

With the octree structures it is possible to skip empty areas of the model by calculating the next collisions based on the layer of the tree, where the first completely empty node for that position was found. This method exchanges time needed for the voxel traversal to time used to fetch data from the voxel storage.

If the ray reaches the back facing outer bounds of the model, we will discard

the fragment. If there is a filled voxel on the path of the ray, we will return t corresponding with first boundary of that voxel and then calculate the collision position using the rays parametric equation. We will also store the normal of the collision boundary. The model space normal can be deduced from the signs of the rays direction vector and the direction in which the found boundary was from the last crossed boundary. The position and the normal can then be transformed from the model space to world space and stored to G-buffer data as well as used to gather additional voxel data from 3D texture or other source.

The method for voxel traversal for ray tracing has not changed significantly from the format it was presented by Amanatides et al. in 1987. [3] Only differences are in inclusion of acceleration structures and context in which the voxel traversal is performed.

Another common optimisation of ray tracing, early ray termination, is automatically included to our solution as all rays terminate on first contact with filled voxel or on reaching outer boundary of the voxel model. Unfortunately GPU architecture doesn't fully benefit from early ray termination as each thread in same execution block will have to execute same instructions. This means that large groups of threads have to wait for the slowest thread in the execution block before they can take new set of rays to trace.

The voxel data is send to the GPU as texture buffer, because uniform buffers had size limitations too strict for the massive voxel data.

3.4 Timing methods

This section explains the setup used for timing the execution of the programming.

Best way to compare performance cost of rendering methods is to compare their effects on the frame time, or time it takes to render one frame. [11] In real time applications targeting 60 FPS, frame time should be under 16 ms, but we should expect that only small portion of that is available for rendering a single voxel model.

To avoid extra overhead from the timing, we store the system time at highest accuracy available from the system. For each context and model, 100 frames are rendered before the starting time is recorded. After 1000 frames have been rendered from the start of the timing, end time is measured and the difference between start and the end is recorded. Another 100 frames is rendered, the time as well as other information about the test is printed, the next model and rendering parameters are initialised and the process is started from the beginning. For comparison, the rendering process without any models takes 0.33 ms at resolution of 640×480 and 0.63 ms at resolution of 2560×1440. This includes clearing the frame buffers, activating the shader for geometry pass, passing the camera and model transformation uniforms, activating lighting pass shader, swapping the G-buffer to input texture and rendering a single whole screen covering quad with the G-buffer as texture. Table 3 shows the full list of times for rendering an empty frame, so that these can be used as reference to latter results. Accuracy of the measurements is of magnitude ± 0.03 ms.

By measuring the time for thousand frames, instead of just one, we will decrease the order of effect from measuring overhead by factor of 1000 and also smooth out

lots of random variance caused by other processes and hardware timings on the computer.

Table 3: Rendering times for frame without any model.

Resolution	Time
640x480	0.33 ms
800x600	0.33 ms
1280x720	0.33 ms
1366x768	0.33 ms
1600x1024	0.33 ms
1920x1200	0.42 ms
2560x1440	0.63 ms

3.5 Testing platform

This section describes the system used for the testing.

The primary computer used for the testing of the rendering methods is a few years old mid-range gaming computer. Table 4 describes the the technical specifications of the hardware and the operating system. All test result reported on this thesis where taken on this hardware, though others were also used during development. To avoid driver software from interfering with the system, all forms of v-sync and frame limits were disabled from the driver settings. For the timing runs, the program is also run as an individual application, without attached debugger or profiling software, which would affect the execution with added overhead.

Table 4: Technical specification of the main test hardware.

Operating System	Microsoft Windows 10 Pro N, 64-bit version 10.0.17763 build 17763
CPU	Intel Core i5-3570K CPU 3401 MHz, 4 Cores, 4 Logical Processors
GPU	NVIDIA GeForce GTX 1060
Graphics clock	1569 MHz
Memory data rate	8.01 Gb/s
Memory interface	192 bits
Memory bandwidth	192.19 GB/s
Driver Version	430.86
Dedicated Video Memory	3072 MB GDDR 5
Total Video Memory	7115 MB

3.6 Test cases

This section describes test scenes used for rendering.

3.6.1 Test case 1: Filled ball with changing model resolution

First test case renders a scene where only object is a voxel model of a filled ball. The ball and camera are positioned so that the ball almost fills the height of the view.

This scene has completely filled interior and empty exterior, with a curved surface between the two. Empty corners of the model allow for empty space skipping, and tree presentations require only few layers of depth near the centre or the corners, but full depth at the boundary.

This scene examines the effects of increased model resolution in case where the octree structure can be utilised to quite high degree. It would be expected that acceleration structures showed performance gains in both methods. Tests were repeated with increasing model resolution until the time required for average frame was above the 16 ms real time requirement, for most storage and rendering methods.

This test was performed with target resolution of 1280×720.

3.6.2 Test case 2: Random voxels with changing model resolution

This scene is created to test how different methods perform on less regular voxel model, the test is initialised with each voxel randomly assigned to either filled or empty state. Each voxel's state is assigned independently with equal change for both states.

As there is little coherency in the voxel data created in this way, the efficiency of acceleration structures is likely greatly reduced. Similar to test case 1, the resolution of the model is increased until most of the storage and rendering method combinations take more than 16 ms on average to render a frame.

This test was performed with target resolution of 1280×720.

3.6.3 Test case 3: Filled ball with changing target resolution

This test uses same ball scene from test 1 with model resolutions of 16 and 128 voxels per dimension. The target image resolution is changed for each measurement, using common monitor resolution as target resolutions. Purpose of this test is to uncover the extend of target image resolution dependency in the rendering cost.

Voxel models of resolution 16 and 128 where used in this test.

3.6.4 Test case 4: Best for the specs

This test iterates over all target and model resolutions and finds the fastest way to render the ball model with them. The search is done by rendering for 1000 frames with each storage and rendering method combination for each resolution combination.

3.6.5 Test case 5: Nightstand model with changing model resolution

This scene uses voxel version of a night stand drawer vertex model. The model is only filled from surface and has a large concave area at front. While the thin layers are more difficult for octree structures the concave area should show benefits of the empty space skipping.

4 Results

This part presents the results of the test cases described in previous section.

4.1 Results of test case 1: Filled ball with changing model resolution

Tables 5 and 6 show the effects of the model size for different storage methods. For the ray casting method the highest resolution benefits from the breadth first or sparse octree storage methods, while depth first octree is always a bit slower. With rasterisation the benefits of the octrees are quite clear, the resolution 128 voxels per dimension takes 12 longer to render without a tree structure than with the slowest tree structure.

It should be noted that the 256 and 512 voxels per dimension resolutions could not be measured for linear array version using the rasterisation method, as the rendering time was too long. It is possible that the memory requirements for the voxel displacement and scale data were larger than testing hardware allowed.

Comparing fastest method for both implementations, we see that rasterisation starts faster at lower model resolutions, but scales worse than ray casting that catches up on higher resolutions.

Table 5: Ball scene rasterised at different model resolutions.

Resolution	Array	Depth First	Breadth First	Sparse
8	0.597 ms	0.536 ms	0.535 ms	0.536 ms
16	0.894 ms	0.618 ms	0.620 ms	0.620 ms
32	1.705 ms	0.794 ms	0.793 ms	0.793 ms
64	4.159 ms	1.084 ms	1.083 ms	1.084 ms
128	31.45 ms	2.007 ms	2.568 ms	1.999 ms
256		8.567 ms	11.41 ms	8.509 ms
512		36.92 ms	54.85 ms	36.52 ms

4.2 Results of test case 2: Random voxels with changing model resolution

Results of this test are visible on tables 7 and 8.

Tests could not be performed for the largest size of the sparse octree with the ray casting method as parts of the tree were not successfully transferred to GPU, even though the frame rate still appeared to be within reasonable numbers. Other than the sparse tree, where memory run out, the ray casting seems to have taken less time on the random model than it did with the ball model. This is likely because the most rays on the random model were terminated quite early as half of the surface voxels were filled.

Table 6: Ball scene with ray casting at different model resolutions.

Resolution	Array	Depth First	Breadth First	Sparse
8	0.619 ms	1.022 ms	0.804 ms	0.949 ms
16	0.790 ms	1.597 ms	1.148 ms	1.409 ms
32	1.148 ms	2.442 ms	1.567 ms	2.102 ms
64	1.846 ms	3.616 ms	2.080 ms	3.080 ms
128	3.332 ms	5.939 ms	3.039 ms	5.041 ms
256	6.768 ms	9.245 ms	4.702 ms	7.661 ms
512	15.71 ms	17.06 ms	10.14 ms	11.99 ms

The linear array rasterisation works within the same time limit it did with the ball scene, but the octree versions now behave worse once the model resolution gets a bit bigger. In fine grained data, octree structure doesn't reduce the amount of cubes that need to be send for GPU and processed by it, but it does introduce slight overhead to generating this data.

Table 7: Random scene rasterised at different model resolutions.

Resolution	Array	Depth First	Breadth First	Sparse
8	0.604 ms	0.611 ms	0.608 ms	0.608 ms
16	1.022 ms	1.016 ms	1.018 ms	1.013 ms
32	2.033 ms	2.000 ms	2.012 ms	2.008 ms
64	4.697 ms	5.801 ms	6.984 ms	5.708 ms
128	30.40 ms	48.38 ms	58.93 ms	47.00 ms

Table 8: Random scene ray casted at different model resolutions.

Resolution	Array	Depth First	Breadth First	Sparse
8	0.591 ms	0.842 ms	0.709 ms	0.827 ms
16	0.605 ms	1.140 ms	0.850 ms	1.036 ms
32	0.643 ms	1.502 ms	1.043 ms	1.348 ms
64	0.672 ms	1.982 ms	1.273 ms	1.823 ms
128	0.748 ms	2.731 ms	1.645 ms	3.265 ms
256	0.969 ms	3.880 ms	2.425 ms	15.05 ms
512	3.610 ms	8.178 ms	7.844 ms	
1024	25.54 ms	56.84 ms	56.66 ms	

4.3 Results of test case 3: Ball with changing target resolution

Tables 9 and 10 show the effects of increasing the rendering target resolution. With rasterisation the changes are quite minor. In the case of the larger model resolution, the linear array, any changes seem to be hidden below measurement error and random variance, though it is already the most expensive method tested. Other storage methods show very similar results for rasterisation, where the change is within factor of 4 for the smaller model resolution and factor of two for the larger model resolution.

Ray casting methods show much higher dependency on the resolution. The required rendering time for the highest target resolution is 4 to 7 times more than the time needed for the lowest resolution. This is still below the difference between the target resolutions themselves, as the highest resolution has 12 times more pixels than the lowest one. If the model covered whole rendering area, the differences might be closer to that.

4.4 Results of test case 4: Best for the specs

Table 11 shows the fasted method of rendering the ball model with given model and target resolutions. Table lists combination of rendering method, voxel storage method and average time it took to render a frame. It must be noted, in few cases there were multiple options within a very small margin and these results could very well vary between tests. Most commonly the closest tied methods were the different octree versions of rasterisation, as they only have minor differences in the buffer generation part and send exactly same data to GPU. On the other hand, ray casting seems to fit best with the breadth first order.

4.5 Results of test case 5: Nightstand model with changing model resolution

Tables 12 and 13 show the rendering times for the nightstand model at different voxel model resolution. This scene was rendered to 640×480 resolution. There seems to be increasing cost on using octrees with rasterising this model, likely because the rasterisation only works with the filled area and they are in most places only single voxel wide. Same also seems to be true for the ray casting as there is no evidence of octree structures empty space skipping benefits.

Table 9: Ball rasterized at different target resolutions.

Target Resolution	Pixel count	16 Voxels	128 Voxels
Array			
640x480	307200	0.520833 ms	33.1744 ms
800x600	480000	0.73671 ms	33.0761 ms
1280x720	921600	0.920398 ms	31.4991 ms
1366x768	1049088	1.05492 ms	31.3225 ms
1600x1024	1638400	1.59914 ms	31.0305 ms
1920x1200	2304000	2.12471 ms	30.9439 ms
2560x1440	3686400	2.92995 ms	31.2283 ms
Breadth First			
640x480	307200	0.432222 ms	2.85937 ms
800x600	480000	0.501319 ms	2.911 ms
1280x720	921600	0.637064 ms	2.7023 ms
1366x768	1049088	0.716699 ms	2.65254 ms
1600x1024	1638400	1.08089 ms	2.69529 ms
1920x1200	2304000	1.42368 ms	3.52196 ms
2560x1440	3686400	1.99414 ms	4.62582 ms
Depth First			
640x480	307200	0.425012 ms	2.3064 ms
800x600	480000	0.502187 ms	2.28395 ms
1280x720	921600	0.635996 ms	2.12206 ms
1366x768	1049088	0.719455 ms	2.24645 ms
1600x1024	1638400	1.08139 ms	2.71024 ms
1920x1200	2304000	1.42114 ms	3.52862 ms
2560x1440	3686400	1.99466 ms	4.63546 ms
Sparse			
640x480	307200	0.42135 ms	2.26507 ms
800x600	480000	0.499356 ms	2.26412 ms
1280x720	921600	0.640705 ms	2.0554 ms
1366x768	1049088	0.719918 ms	2.12352 ms
1600x1024	1638400	1.08057 ms	2.70861 ms
1920x1200	2304000	1.4221 ms	3.5291 ms
2560x1440	3686400	1.99409 ms	4.63606 ms

Table 10: Ball ray casted at different target resolutions.

Target Resolution	Pixel count	16 Voxels	128 Voxels
Array			
640x480	307200	0.476711 ms	1.84445 ms
800x600	480000	0.616808 ms	2.60847 ms
1280x720	921600	0.854705 ms	3.60626 ms
1366x768	1049088	0.898847 ms	3.75736 ms
1600x1024	1638400	1.40666 ms	6.10015 ms
1920x1200	2304000	1.85738 ms	8.00283 ms
2560x1440	3686400	2.63917 ms	11.1832 ms
Breadth First			
640x480	307200	0.656961 ms	1.81473 ms
800x600	480000	0.90145 ms	2.482 ms
1280x720	921600	1.24681 ms	3.08097 ms
1366x768	1049088	1.28603 ms	3.42392 ms
1600x1024	1638400	2.04373 ms	5.25886 ms
1920x1200	2304000	2.67372 ms	6.82278 ms
2560x1440	3686400	3.76686 ms	9.15793 ms
Depth First			
640x480	307200	0.94513 ms	3.67431 ms
800x600	480000	1.27486 ms	4.92084 ms
1280x720	921600	1.71131 ms	6.43214 ms
1366x768	1049088	1.81519 ms	6.63728 ms
1600x1024	1638400	2.86434 ms	9.87629 ms
1920x1200	2304000	3.72246 ms	12.5897 ms
2560x1440	3686400	5.24207 ms	16.6415 ms
Sparse			
640x480	307200	0.81623 ms	3.13034 ms
800x600	480000	1.12724 ms	4.22451 ms
1280x720	921600	1.53161 ms	5.08934 ms
1366x768	1049088	1.59829 ms	5.64201 ms
1600x1024	1638400	2.51629 ms	8.27159 ms
1920x1200	2304000	3.2797 ms	10.501 ms
2560x1440	3686400	4.60142 ms	13.7503 ms

Table 11: Fastest method for each model and target resolution. Time is in milliseconds. Key: A = Array, BF = Breadth First, DF = Depth First, S = Sparse, RC = Ray Cast, RS = Rasterisation

Resolution	8	16	32	64	128	256	512
640x480	S-RS	A-RC	S-RS	S-RS	BF-RC	BF-RC	S-RC
	0.412	0.468	0.549	0.946	1.780	3.037	8.316
800x600	DF-RS	DF-RS	S-RS	S-RS	BF-RC	BF-RC	BF-RC
	0.526	0.557	0.731	1.277	2.570	3.923	9.106
1280x720	DF-RS	BF-RS	S-RS	BF-RS	S-RS	BF-RC	BF-RC
	0.558	0.650	0.836	1.136	2.170	4.811	10.51
1366x768	S-RS	DF-RS	BF-RS	BF-RS	S-RS	BF-RC	BF-RC
	0.627	0.727	0.934	1.297	2.208	5.317	11.25
1600x1024	DF-RS	S-RS	BF-RS	BF-RS	DF-RS	BF-RC	BF-RC
	0.903	1.091	1.437	1.941	2.744	7.657	14.46
1920x1200	DF-RS	S-RS	S-RS	S-RS	S-RS	DF-RS	BF-RC
	1.224	1.434	1.877	2.568	3.596	9.568	17.40
2560x1440	DF-RS	BF-RS	DF-RS	DF-RS	S-RS	S-RS	BF-RC
	1.747	2.029	2.587	3.445	4.730	9.752	21.41

Table 12: Nightstand scene rasterised at different model resolutions.

Resolution	Array	Depth First	Breadth First	Sparse
8	0.396 ms	0.394 ms	0.424 ms	0.408 ms
16	0.520 ms	0.406 ms	0.415 ms	0.407 ms
32	0.719 ms	0.631 ms	0.695 ms	0.627 ms
64	1.733 ms	2.229 ms	2.708 ms	2.241 ms
128	6.605 ms	9.211 ms	11.62 ms	8.996 ms
256	31.90 ms	41.80 ms	56.33 ms	41.53 ms

Table 13: Nightstand scene with ray casting different model resolutions.

Resolution	Array	Depth First	Breadth First	Sparse
8	0.494 ms	0.405 ms	0.407 ms	0.398 ms
16	0.410 ms	0.454 ms	0.396 ms	0.427 ms
32	0.431 ms	1.173 ms	0.727 ms	1.029 ms
64	0.629 ms	1.764 ms	0.995 ms	1.537 ms
128	1.064 ms	2.698 ms	1.442 ms	2.439 ms
256	2.147 ms	4.356 ms	2.529 ms	4.237 ms

5 Evaluation

This chapter presents evaluation of the results and how well they are generaliseable.

5.1 Reading the results

Rasterisation method without octree optimisation was able to render models only to resolutions of 128 or 256 voxels per dimension, depending on model. That the nightstand model could be drawn with higher model resolution is due to lower filling rate of the model, as total number of instances is equal to number of filled voxels in this method. Octree structures allowed coherent models like the ball to be rendered up to model resolution of 1024 voxels per dimension, but frame times were significantly higher than what would be required for real-time application even with resolution of 512.

For rasterisation, it seems that any of the acceleration octrees will significantly improve performance over the flat linear array representation, when there are continuous large completely filled areas in the model. In these cases the differences in different octrees performance are quite small, likely within margin of error from the test and timing setup as well as random variance from background processes.

On cases where there is a low coherency or filling rate, the octrees cause a noticeable performance penalty on higher resolutions and while depth first and sparse octrees perform similarly, breath first seems to be somewhat slower.

Performance of the rasterisation method could likely be improved by using smaller numbers to contain the offset and scale of each instance that needs to be rendered. Using 16 bit numbers would require only half of the memory used by 32 bit numbers used in this implementation. Though the savings from decreased memory use would be slightly offset by need to pack and unpack numbers that don't match with internal register sizes and memory alignment.

Ray casting shows more differences between the different storage variants. For small model resolutions all formats seem to be within error margin, but on larger resolutions plain linear array and breadth first octree seems to be faster than the other octrees. Sparse octree performs bit better than

Most tests don't show meaningful performance gains from empty space skipping, and in cases where model coherency is low there seems to be a performance penalty for using tree structures. This might be from the size of the data transferred to GPU each frame or from additional branching and voxel data accesses on the traversal algorithm.

The best for the specification test shows that for small model resolutions, the rasterisation method is faster. As the target resolution rises, the resolution of the model that can be rendered faster with rasterisation methods grows as ray casting methods are more dependent on the resolution of the target image.

5.2 Generalising results

Comprehensive testing was only performed on single hardware setup, but given the similarity of most modern computer and console hardware, results should be indicative of the performance on most similar platforms.

Sparse octree solutions fared rather poorly in our test case, this is likely because the nodes where much larger in size. If other rendering data, such as textures, where included into the voxel data, and thus voxel required more than 1 bit to be stored, it would likely be much more beneficial to use sparse structure as more data could be stored closer to the root of the octree.

Results for the best for the spec test are consistent with the theoretical complexity of the rendering methods, it is unlikely that any optimisations on the rendering methods would fundamentally change the results, just adjust the line at which the faster method changes.

Because of targeting highly dynamic and changing voxel data, implementations on this paper required the whole voxel data to be updated on each frame. If such requirements are not met or a method to track changes and create partial updates is used many of the larger models will see significant performance gains.

In the test scenarios, the rendered voxels were without any texture data. In real applications where these textures would be present, each fragment shader would have to do few texture sampling operations for each fragment it needs to calculate. This would apply a small performance penalty to each method. As ray casting method already does multiple texture fetches for the voxel data and only handles maximum of two fragment per pixel, effects of textures would likely be quite minimal for it. On the other hand, rasterisation method may have to perform quite a many texture fetches for fragments that are hidden from the view, because another fragments can latter overwrite it.

In our tests, the voxel model was a only model on the scene. In real applications, there would likely be multiple objects, both voxel and vertex models, on the scene, in these situations, the ray casting methods inability to use early depth testing would likely give the rasterisation method a slight advantage.

6 Conclusions

This chapter presents the conclusions which can be drawn from the evaluated results and questions left open for further research.

6.1 Recommendations

Given the results of the tests represented here, it seems that ray casting is better option for larger model resolutions. Though situations in which the ray casting methods are faster, are usually quite close to a line where their use in real time applications needs to be considered with care as rendering times with larger model resolutions were in 10 to 20 ms general area. If model is likely to be coherent, using breadth-first octree is recommendable. For low coherency models array of voxels will work better.

Rasterising the models seems suitable for small and medium resolution voxel models with large coherent areas that can be easily optimised by hierarchical structures. Being less affected by rendering resolution might be useful in some situations as well. If rasterisation solution is needed, it would be beneficial to try to keep the models quite coherent.

6.2 Further research topics

There are still further optimisations available for both methods, which could not be implemented or tested in the limited time of this thesis work.

This thesis did not examine situations where model was only partly visible. In rasterisation version the view space culling and early z-buffer testing should remove most of the overhead from fragmentation shader, but the amount of data send to GPU and the workload of the vertex shader and need for memory bandwidth could be greatly reduced if the parts that are out of view or hidden behind other objects could be detected and ignored on the instance offset and scale buffer generation. Similarly parts hidden by the models own geometry could be ignored if there was a cheap way of detecting them.

For ray casting, view space culling removes most of the unnecessary work for partly out of view models, but because the z-buffer needs to be adjusted for the voxels depth, early z-buffer testing can't be used for the voxel object. Manual z-buffer testing before or during the ray traversal might improve the performance, when model is partly covered.

Tests performed with the ray casting showed benefits from acceleration structures, but not as great as could be expected based on prior research. It might require that algorithm for ray traversal was adjusted so that last place examined from the octree was kept in memory and used as a starting point for the the next voxel status query, instead of starting form the root node at each query. Using of brick approach where leaf nodes of the model are not singular voxels but blocks of multiple voxels might also be beneficial.

With dynamic voxel data, tracking changes in the data separately and updating only the necessary parts of the voxel data each frame might help reduce the memory bandwidth requirements.

Texturing of the voxels was mostly ignored on this paper. Testing how texture or other rendering data fetches affect the different rendering methods, would provide useful information on wider range of application areas. It is quite likely that the rasterisation method might scale worse with the rendering resolution if texture sampling was performed on each fragment that ends up being computed on fragment shader. On the other hand, ray cast method already makes multiple texture fetches for each fragment, and is computed for much smaller number of fragments, so the penalty there might be lower.

In tests presented here we did not address optimisations based on level of detail. In both methods, if we knew that a partially filled node in the octree would be far enough from the camera that its children could not be visually differentiated, the node could be rendered as full or empty node, depending the majority of the children. Our 2 bit system for the octrees would even have room for this information.

As noted in the introduction of this paper, there seems to be lack of experiments with non-cubical voxels. Given the efficiency of the rasterisation solution and that the instancing other shapes would be trivial, it would by interesting to perform similar study to non-cubical voxels. Do they require different hierarchical structures and what level of detail on voxel model leads to similar level of visual detail? Is there efficient algorithms for traversing non-cubical voxels.

While the testing system is quite descriptive of the algorithms' performance on modern hardware, it might be of interest to extend the tests to other hardware and software setups and compare the results to hardware features such as GPU memory bandwidth and amount of processing cores. This might lead to some insight on how well the algorithms will scale on future hardware.

Given the recent hardware releases, in which consumer GPUs have included increasing number of generic programming specialised cores, as well as hardware and software optimisations for ray tracing, performing similar experiment with algorithms optimised for the new GPUs might provide interesting data on their performance.

7 Summary

At the beginning of this thesis we examined the history of programmable graphics hardware and the previous research of voxel graphics. Graphics processing units formed when specific operations where moved from CPU to external chips, chips used in graphics processing were combined and extended for other features and eventually opened for application programmers. Voxel data has uses in both scientific and medical fields as well as entertainment industry. Most research on rendering voxels in real time has been on area of more complex static models and lighting methods for ray casting.

Two voxel rendering methods, aimed for highly dynamic, but simple voxel data, were build on top of simple standard deferred rendering pipeline and extensively tested using four different implementations of voxel storage.

The tests showed that rasterisation method was faster for low resolution voxel models and less dependent of resolution of the rendering target. Ray casting was most suitable for high resolution voxel models on smaller rendering target resolutions. Coherency of the model and use of octree storage structures for optimisation was more important for rasterisation method.

Benefits of the octree structures where dependent on the coherence of the model, and low coherence models even saw performance drops over plain voxel data. For rasterisation, the octree storage methods where about equal to each other, but ray casting benefited most from breadth first storage order.

While these results are likely to generalise for many use cases, anyone implementing voxel rendering should consider their needs for other rendering information as well as context and type of the voxel models rendered.

References

[1] Nvidia tegra x1, January 2015. Accessed. 2019-20-07, https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149. ACM, 2009.

[3] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987.

[4] Ulf Assarsson, Markus Billeter, Dan Dolonius, Elmar Eisemann, Alberto Jaspe, Leonardo Scandolo, and Erik Sintorn. Voxel dags and multiresolution hierarchies: from large-scale scenes to pre-computed shadows. *Proc. EUROGRAPHICS Tutorials*, page 6, 2018.

[5] Jeroen Baert, Ares Lagae, and Philip Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th high-performance graphics conference*, pages 27–32. ACM, 2013.

[6] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. A survey of gpu-based large-scale volume visualization. In *Eurographics Conference on Visualization (EuroVis)(2014)*. IEEE Visualization and Graphics Technical Committee (IEEE VGTC), 2014.

[7] Katherine Bourzac. Chip hall of fame: Nvidia nv20. *IEEE Spectrum*, 2018. Accessed: 2019-15-06, https://spectrum.ieee.org/tech-history/silicon-revolution/chip-hall-of-fame-nvidia-nv20.

[8] PCQ Bureau. A programmable graphics chip, 2001. Accessed: 2019-06-03, https://www.pcquest.com/a-programmable-graphics-chip/.

[9] Davide Cammarata. The evolution of the gpu. Accessed: 2019-07-07, http://piermarcobarbe.github.io/informatics_history_HCI_atelier_2015/html/hardware/evolution_gpu.html.

[10] Evgeni V. Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Technical report, 1995.

[11] OpenGL Wiki contributors. Opengl wiki: Performance. Accessed: 2019-24-06, http://www.khronos.org/opengl/wiki_opengl/index.php?title=Performance&oldid=12362.

[12] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22. ACM, 2009.

[13] Thomas Scott Crow. Evolution of the graphical processing unit. *A professional paper submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science, University of Nevada, Reno*, 2004.

[14] Johann Jakob Eberle. *Development and analysis of a workstation computer*. PhD thesis, ETH Zurich, 1987.

[15] Elmar Eisemann and Xavier Décoret. Fast scene voxelization and applications. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 71–78, New York, NY, USA, 2006. ACM.

[16] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, GI '08, pages 73–80, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[17] Fujitsu develops world's first three dimensional geometry processor. Accessed: 2019-06-03, http://pr.fujitsu.com/jp/news/1997/Jul/2e.html.

[18] G-Truc Creation. Opengl mathematics. Accessed: 2019-05-07, https://glm.g-truc.net/0.9.9/index.html.

[19] GLFW. Glfw. Accessed: 2019-05-07, https://www.glfw.org/.

[20] James Hague. Why do dedicated game consoles exist? Accessed: 2019-16-06 via https://web.archive.org/web/20150504042057/http://prog21.dadgum.com/181.html.

[21] David Herberth. Multi-language vulkan/gl/gles/egl/glx/wgl loader-generator based on the official specs. Accessed: 2019-05-07, https://github.com/Dav1dde/glad.

[22] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012.

[23] Jens Krüger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38. IEEE Computer Society, 2003.

[24] Philippe Lacroute and Marc 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*, pages 451–458. ACM, 1994.

[25] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[26] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics (TOG)*, 9(3):245–261, 1990.

[27] Erik Lindholm, Mark J Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM, 2001.

[28] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[29] Sebastian Macke. Voxelspace, 2017. Accessed: 2019-16-06, https://github.com/s-macke/VoxelSpace.

[30] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics (TOG)*, 22(3):896–907, 2003.

[31] Chris McClanahan. History and evolution of gpu architecture. *A Survey Paper*, page 9, 2010.

[32] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

[33] Nvidia turing gpu architecture, 2018. Accessed: 2019-16-06, https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[34] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph.*, 29(6):179:1–179:10, December 2010.

[35] Ken Shirriff. Restoring ycombinator's xerox alto day 5: Microcode tracing with a logic analyzer. *Ken Shirriff's blog*, 9 2016. Accessed: 2019-06-03, http://www.righto.com/2016/09/xerox-alto-restoration-day-5-smoke-and.html.

[36] Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. Trax: A multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1802–1815, 2009.

[37] Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Kalle Immonen, and Jarmo Takala. Fast hardware construction and refitting of quantized bounding volume hierarchies. In *Computer Graphics Forum*, volume 36, pages 167–178. Wiley Online Library, 2017.

[38] Li-Yi Wei. A crash course on programmable graphics hardware. *Microsoft Research Asia, Tsinghua University, Beijing*, 2005.

[39] Lee Alan Westover. *Splatting: a parallel, feed-forward volume rendering algorithm*. PhD thesis, University of North Carolina at Chapel Hill Chapel Hill, NC, 1991.

[40] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 434–444. ACM, 2005.

[41] Xenol. The nintendo 64 is one of the greatest gaming devices of all time, 2015. Accessed: 2019-16-06, https://xenol.kinja.com/the-nintendo-64-is-one-of-the-greatest-gaming-devices-o-1722364688.