Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Crouch, Liam Svanåsbakken

The application of voxel octrees for 3d video

Masteroppgave i MSIT
Veileder: Hardeberg, Jon Yngve
Medveileder: Umetani, Nobuyuki
Juli 2023

NTNU
Kunnskap for en bedre verden

Crouch, Liam Svanåsbakken

The application of voxel octrees for 3d video

Masteroppgave i MSIT
Veileder: Hardeberg, Jon Yngve
Medveileder: Umetani, Nobuyuki
Juli 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU

Kunnskap for en bedre verden

NTNU
Kunnskap for en bedre verden

Department of Computer Science

IT3920 - Master's Thesis

The application of voxel octrees for 3d video

Author:
Liam Svanåsbakken Crouch

Advisors:
Jon Yngve Hardeberg (NTNU)
Nobuyuki Umetani (University of Tokyo)

10th July 2023

Abstract

The growing popularity of virtual reality (VR) and augmented reality (AR) headsets has opened up new opportunities for immersive video experiences. This thesis explores the potential of fully immersive video-like experiences using depth camera-based recordings, leveraging the voxel octree data structure for storing these recordings. The goal is to investigate the feasibility of lossy encoding methods for voxel octrees, aiming to achieve efficient video compression while preserving spatial and temporal similarities within the data.

Two main research questions are addressed: (1) the feasibility of lossy subtree substitution for encoding sparse voxel octrees, and (2) the suitability of Discrete Cosine Transform (DCT) for lossy encoding of color information in voxel octrees. The thesis presents a comprehensive analysis of these research questions through experiments and evaluations. The results demonstrate that while subtree substitution showed limitations, DCT encoding proved to be a promising technique for color compression in voxel octrees.

The thesis contributes novel methods for lossy encoding of voxel octrees and provides insights into the potential future directions for 3D video encoding. However, computational cost, the exploration of hybrid approaches (such as combining voxel octrees with B+-trees), better methods for performing octree comparison, and better ordering of color data before DCT compression remain as areas for further research and improvement.

Sammendrag

Med en økende interesse for VR og AR-utstyr har det åpnet opp nye muligheter for mer oppslukende video-opplevelser. Disse kan bl.a lages ved å ta opp video med dybdekameraer, og large disse i "*voxel octree*" datastrukturer. Denne oppgaven undersøker bruk av "*voxel octree*" for å levere oppslukende video-opplevelser tatt opp med dybdekamera. Målet er å undersøke muligheten til å bruke diverse metoder for å large "octrees" med tap - der unødvendig informasjon blir fjernet for å spare plass. To research-spørsmål blir presentert: (1) Hvorvidt det er mulig å redusere filstørrelsen på et "octree" ved å fjerne deler som ligner, og (2) hvorvidt det er mulig å bruke "Discrete Cosine Transform (DCT)" for å komprimere fargene i disse trærne.

Oppgaven presenterer en analyse av disse spørsmålene ved hjelp av eksperimenter og numerisk analyse av testmateriale. Resultatene viser at DCT virker lovende som teknologi, og at substitusjon av trær trenger mer arbeid for å være lovende.

Oppgaven presenterer også nye teknikker for å lagre "voxel octree" med tap. Videre presenteres flere videre temaer som kan tas opp i senere arbeid, som hybrid-datastrukturer, bedre sammenligning av volumet i to trær, og bedre teknikker for å sammenligne fargen i to trær.

1 Acknowledgements

This page contains acknowledgements from the main author of the paper.

I would like to start by thanking my thesis advisor, Jon Yngve Hardeberg, for his guidance and support throughout my research. His presence has been a solid anchor for me throughout the entire thesis process. The course I took at his lab in Gjøvik, Colorlab, was a great experience that opened up my mind to a lot of new concepts that I would not have been exposed to otherwise. I am truly grateful for his mentorship and for giving me the opportunity to learn from him.

I would also like to extend my heartfelt gratitude to my professor in Japan, Nobuyuki Umetani, for his unwavering support during trying times. When the pandemic made exchange programs difficult, he was instrumental in helping me fulfill my goals. Staying at his lab has been an amazing experience that has allowed me to explore and play around with technologies that I never thought I would have the chance to. His insights and encouragement have been invaluable to me, and I am honored to have had the chance to work with him.

Furthermore, I would like to thank the members of the various labs I have been associated with for their support and inspiration. They have provided me with a wealth of knowledge and insights that have helped me navigate through the various challenges I encountered during my research.

I would like to thank all the people who participated in the user studies carried out as part of this thesis, whom I relied on for subjective data collection.

Another thank you goes to Bendik Dyrli for lending me frankly obscene amounts of compute resources when time was short, allowing me to perform one of my experiments with very little time to spare.

Finally, I would like to express my gratitude to my family and friends, who have been my pillars of strength and encouragement throughout my academic journey. Their unwavering support and love have given me the courage and determination to push through the difficult times and strive for excellence.

Thank you all for your support and encouragement.

Table of Contents

List of Figures

List of Tables

Part I

Introduction

This part discusses the overall project motivation, goal, research questions, and contributions.

2 Motivation

In recent years, the sale of VR and AR headsets have increased in both the consumer and professional market. Multiple sources expect the market to continue growing, such as (*Virtual Reality Headset Market Share & Growth Report, 2030* 2023). As a result, many companies are investing a lot of resources into the field. A notable example is Meta(previously Facebook), which purchased Oculus in 2014 and renamed it to Meta in a show of intention to pivot its business model towards this new human interfacing frontier. Meta isn't alone - other large tech companies also have a pawn in the game. Microsoft has been releasing its own AR kit aimed at professionals, the HoloLens, since 2016. Apple recently joined the game by announcing its AR headset, Apple Vision Pro.

With more VR/AR gear in the wild, new opportunities are opening for entertainment and productivity. One such opportunity is video. While traditional two-dimensional videos are already enjoyed in VR/AR through applications that simulate cinemas (Rodgers, 2022), there is a potential for fully immersive video-like experiences where the user is able to walk around the video as it is being played. While this is already achievable using traditional 3d animation with rigged meshes, it is worth exploring how the same goal can be achieved for depth camera-based recordings.

These three-dimensional videos have many applications in both entertainment and productivity. In entertainment, they could allow for media where the user is encouraged to view the video multiple times, following a different actor every time. One could claim that the emphasis on viewing a video multiple times from different perspectives teaches the viewer values such as nuances and considering different people's viewpoints. In this manner, these videos could be used in an educational context by teaching children about the value of considering other's viewpoint.

Productive applications also exist. A popular example is teleconferencing, where 3d video could be used to unite people around the world such that they perceive each other as existing in the same meeting room. Another possible application is training videos. For example, many fast food chains have historically used videos to help onboard new employees on how to operate machines and serve customers. Today, some companies use VR training software in order to train new employees (Sisson, 2020). 3d videos may be another useful tool for this application.

Voxel octrees are an interesting data structure due to the quality of having an unlimited level of detail and the ability to store large, sparse structures efficiently. As you will see in Part II, they are well-studied, and see a lot of adaptation in both academia and practical applications. Famously, John Carmack spoke well about the use of octrees for storing 3d assets in a 2008 interview(Shrout, 2008). Carmack spent the interview talking about how he thought Octrees were the future of 3d asset storage due to their infinite detail quality, and that he wanted hardware that was designed to render them efficiently. However, his dream never came true. Figure 1 shows a Google Trends search, showing the popularity of the search term "Octree". By the time Carmack was interviewed, octrees were already becoming less popular, and their popularity never recovered.

This was 10 years before NViDIA released their first line of RTX graphics cards, effectively making raytracing available to the masses. While octrees never took off in the way Carmack wished, they have never been completely off the table. With current-generation graphics cards putting raytracing-capable hardware in the hands of consumers, now is as good a time as any to reconsider its use.

Figure 1: Screencapture from Google Trends (*Google Trends* 2023) showing relative popularity of the term "octree" worldwide since 2004

Voxel octrees, as a spatial tree-structure, is an interesting data structure for video encoding. This is due to the fact that child nodes recursively re-define the volume of their parent in double the resolution. Because of this quality, it is easy to store scenes where the resolution varies locally depending on need. It is also possible to perform various forms of adaptive rendering where the renderer decides how far down the tree it wants to iterate depending on some factor. This is similar to how one uses tesselation to get a similar result when rendering polygons. Doing this has an obvious benefit: speeding up rendering and being able to better prioritize where to spend rendering resources.

One example of adaptive rendering is using the current distance to the camera to render objects far away with a lower quality. One could also apply this technique to lower the bandwidth needed to stream the recording over the internet. In VR/AR applications, one could perform foveated rendering. This is a technique where an iris tracker is used to prioritize rendering parts that the eye is looking at.

While adaptive rendering techniques exist for polygon rendering, they have some shortcomings that octrees don't have. For example, octrees can easily render different parts of the same tree in varying levels of detail, without risking jagged edges or seams where two levels of detail intersect. Figure 2 illustrates this - notice how the model is not watertight along the seam between the two levels of detail. As long as the two levels of detail do not share the same vertices at the seams, this is bound to happen.

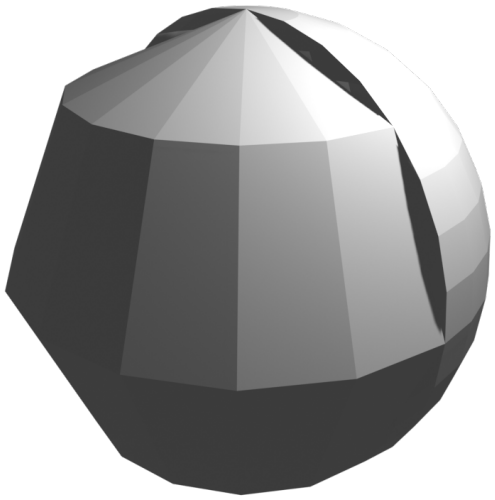


Figure 2: Illustration showing problematic mesh seams in a sphere model where the front and back are rendered at different levels of quality.

With the current modern technological landscape, we think it is about time Octrees gets some attention again. Therefore, in this thesis, we will apply the Voxel octree data structure to the encoding of three-dimensional videos. It is our hope that by doing this, we contribute valuable information to the scientific literature about possible ways forward for 3d video encoding in the future.

3 Goals

The overarching goal of this thesis is to **present a sparse voxel octree(SVO)-based lossy video encoding method**. In a previous preparatory course, software for capturing and building SVOs was written (as will be explained in Section 13). In this thesis we will build on this by Investigating lossy encoding of voxel octrees in two manners, which are futher broken down in Section 14. The core of this thesis is designing a video encoding scheme for Voxel octrees that:

- Is able to benefit from the temporal and spatial similarities that may exist within an octree and across multiple frames in a video sequence.
- Is able to encode the tree data structure in a space-efficient manner

In addition to the aforementioned goals, we will generally work to reduce the filesize of the voxel octree video by experimenting with efficient schemes for encoding SVOs.

Although the ability to selectively load only parts of the octree is mentioned as a motivation for the choice of voxel octrees, we do not consider it a necessity for showing off the format. It will therefore be considered out of scope for this thesis. The same goes for techniques for efficiently rendering these octrees - there exists a lot of literature on this already.

3.1 Subtree substitution

The first technique we want to use to lossily compress sparse voxel octrees with is what we will call subtree substitution. A sparse voxel octree is a tree structure with many layers of nodes. Previous solutions like (Kämpe et al., 2016) have already used this fact by finding subtrees that are identical and de-duplicating them by using the same subtree as child of multiple parent nodes. This can be done within one octree, or even across multiple octrees, to save space. In octrees that are digitally rendered with a lot of static geometry between each frame, there are some serious savings to be made by doing this.

In this thesis, we will keep to a theme of using real-life captures that therefore will see less savings when deduplicating by looking for completely identical subtrees. Instead, we will make this process lossy by combing through all nodes of a given voxel level, marking nodes as similar if they are *close to identical* using some comparison function.

We will bundle together multiple frames into chunks, creating a DAG containing the data of multiple SVOs. We will then perform the deduplication process once per chunk. Figure 3 shows an example tree structure before deduplication has been performed, while Figure 4 shows the resulting DAG with fewer nodes encoding the same information.

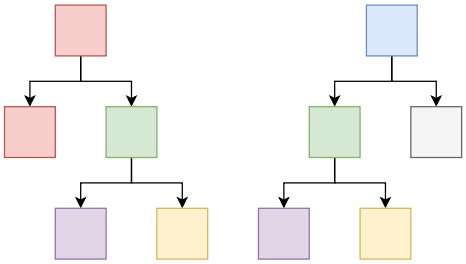


Figure 3: An illustration of two trees before deduplication by subtree substitution

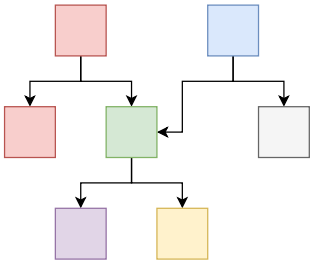


Figure 4: An illustration of the same trees as in Figure 3, but after deduplication has been done.

3.2 DCT encoding of color data

DCT is famously used in JPEG, among other applications. It's a technique for signal encoding where a signal, usually a two-dimensional one, is encoded as a set of weights that when multiplied by corresponding cosine waves become an estimation of the original system. As the human vision system is bad at recognizing high-frequency noise, we can quantize these weights depending on the frequency of the cosine wave they correspond to in such a manner that sacrifices detail in high-frequency areas. Depending on how much quantization is applied, this technique can give us impressive gains without much noticeable loss of detail.

4 Research questions

The goals from Section 3 can be summed up by an overall question we want to answer: "*How well can we encode sparse voxel octrees in a lossy manner?*", or simply put: "*Are voxel octrees a suitable data structure for 3d video encoding?*". Using the two overall goals in Section 3 and the two approaches we wish to use for lossy encoding, we present our research questions in Table 1. Research questions are a systematic way to seek and gain knowledge about a subject. In this thesis, we will use them to divide-and-conquer the big question into smaller, more easily researchable questions.

RQ1	How feasible is lossy subtree substitution for lossy encoding of SVOs?
RQ2	How feasible is DCT for lossy encoding of color information in a SVO?

Table 1: Research questions proposed by this thesis

.

Oates, 2005 suggests that one research question is usually related to one research strategy. In this thesis, we will follow this way of thinking by using research questions to organize the experiments we later perform in Part V. The goal is for the research questions to lay the base for the later experiments to thoroughly the subject. We will now go through each research question and explain the motivation behind them.

4.1 RQ1

RQ1 aims to answer whether or not lossy subtree substitution is a feasible method for lossy encoding of SVOs. This is one of the simpler approaches, as in theory, all we have to do is find parts of the trees that are similar and replace them. However, there are two main reasons why we think this may be challenging:

- Even with hundreds or thousands of nodes, there may still be very few subtrees that are similar enough. This is because the number of possible combinations of children increases exponentially, and is very large even for only a few layers.
- Even if similar trees are found, the colors also have to match as well

As with many things in life, the devil is in the detail. As will be explained later in this thesis, finding a solid method for searching *similar* octrees is hard to both define and implement. It is a goal of this thesis to contribute experience with this technique to the scientific literature, and this is therefore one of our research questions.

4.2 RQ2

RQ2, in a similar manner to **RQ1**, aims to figure out whether or not DCT is a valuable technique for color encoding in SVOs. It makes sense to try applying technologies we already know. With DCT already being popular in applications like JPEG (Wallace, 1992), applying it again to SVOs might be an "easy win". RQ2 exists out of curiosity as to what would happen if we gave it a try.

5 Contributions

This thesis contributes the following to the scientific literature:

- A method for performing lossy encoding of voxel octrees using multiple techniques, which we believe are novel.

- A simple yet configurable testing platform for performing online tests of perceived quality using videos.

There are similar solutions out there. For example, Museth, 2013 is a time series-compatible encoding scheme for volumetric data. There are also solutions that use two-dimensional videos as their ground fundament. In fact, (Kämpe et al., 2016) suggests what we aim to do as a possible future work. As you will find in Part II, there are to our knowledge no papers that attempt to encode sequential trees in a lossy manner for the sake of rendering, making our method novel.

Part II

Background

This section provides the background necessary to read the rest of the Thesis, an (abridged) literature review of the field, as well as a description of the current state of the art.

6 A primer on video encoding and 3d scene encoding

6.1 Octrees

An octree is a tree structure where each node may have up to 8 children.

A **voxel octree** is an octree where the goal is to represent a voxel in variable detail by allowing each node to be subdivided into 8 smaller nodes, each $\frac{1}{8}$ the volume of their parent. This can be represented as each node containing 8 *slots* corresponding to fixed areas, in which a child may or may not exist. The absence of a child implies that the area is empty, while the opposite implies that the area is defined by the child node.

A octree datastructure allows for some neat benefits:

- Infinite detail
- Empty areas to not consume memory - memory consumption is affected by model complexity, similar to 3d models.
- Level of Detail can be varied in different areas of the octree
  - You could load higher detail in areas closer to the camera.
  - Using meshes with variable LoD means you are inherently dealing with different models, which may cause seams in the sections where they join.

There are also some inherent downsides to this data structure. Being a tree structure, you have to dereference a pointer per layer traverse. This is not optimal on modern hardware, which is often reliant on a high amount of cache hits. With octrees having a low branching factor(i.e few children per node - a deeper tree is needed to reach a given resolution), octrees are slow on memory-bound systems.

Other n-tree data structures with usage in spatial encoding exist. A good example is the quadtree, which is already used for traditional 2d video encoding today **TODO: Cite**. Figure 5 illustrates an example of a quadtree, which in this context serves as a simplified explanation of how spatial tree structures are able to efficiently encode detail by avoiding encoding areas with no data.

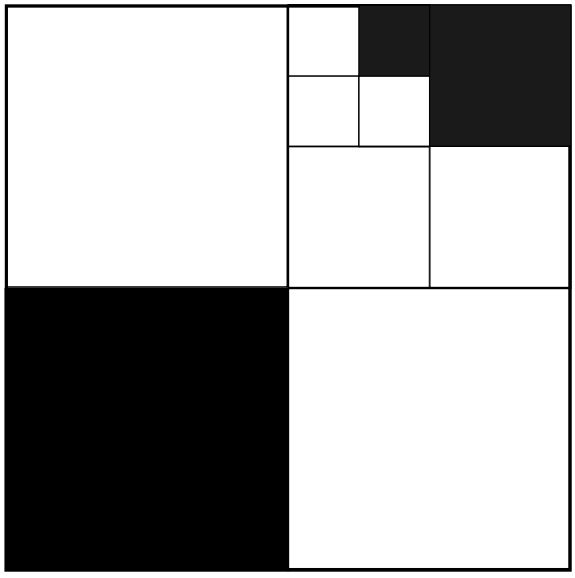


Figure 5: An example quadtree with a resolution of $2^3x2^3 = 8x8$

The quadtree in Figure 5 can be drawn as a tree structure where each node contains four child pointers, one for each of the four areas you get if you subdivide the root box into 4 equally sized

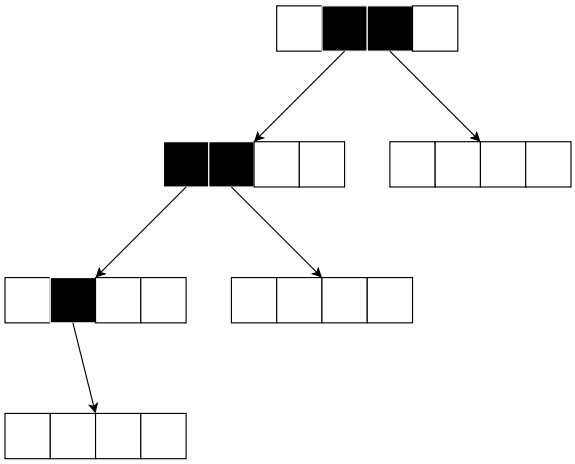


Figure 6: Illustration showing child pointers for each node of the quadtree shown in Figure 5

boxes. Figure 6 illustrates this. We use the order top-left, top-right, bottom-left, bottom-right for the child pointers in this exampe. A black pointer box means it contains a pointer, while a white one means no pointer exists. If encoded correctly, this kind of data structure can be very spare efficient for sparse data.

John Carmack is a known fan of voxel octrees, having talked extensively about how it could be the content format of the future in Shrout, 2008. Carmacks wishes did not come true - the industry is still very focused on triangle-based rendering. However, recently, with the introduction of NVIDIA RTX, raytracing is suddenly in the mainstream. Therefore, we believe a reevaluation of raytracing as a rendering technique is useful.

Volumetric video is already in widespread use in the entertainment industry. However, voxel octrees haven't seen much use. There are many reasons for this, with one of the main probably being the need for new technologies to tightly integrate with existing tools.

7 Background

7.1 Surface reconstruction

Modern surface reconstruction from a Pointcloud is usually based on Poisson Surface reconstruction (Kazhdan et al., 2006), where an indicator function is used to determine if an arbitrary given point is inside or outside a shape. A threshold can then be picked and used to build a surface in the gradient between the inside and outside that will approximate the pointcloud. This is in itself a tough problem, as you rely on normals to be accurate in order for the indicator function field to be accurate. Recent papers like Xu et al., 2023 show promising progress in this field.

7.2 Octrees in research

Rendering an octree is nothing new - Octrees have been heavily investigated in scientific literature. We have been able to render octrees in real-time for more than 10 years. For example Laine and Karras, 2011 presents an advanced data structure and rendering method that was able to render octrees of high resolution using a GPU.

(Wurm et al., 2010) is very similar to what we are doing in that they are using octrees for the storage of depth sensor data. However, their focus is purely on spatial data, and do not care about RGB data.

Another worthy mention is the use of octrees in many research applications. For example, Takikawa

et al., 2021 uses an Octree data structure to encode feature vectors for a Neural SDF renderer. Another paper, Yu et al., 2021, uses octrees to generate a lookup table of from NeRF(Mildenhall et al., 2020) network, that can be used to speed up rendering.

We are not the first people to apply DCT to an octree either. (Baziyad et al., 2021) did it for the purpose of steganography - hiding information in plain sight.

7.3 In video games

Although voxel octrees never experienced the mainstream adoption in video games as Carmack wanted, they are still in use. A great example of this is Oortmerssen et al., 2004 - a quake-style shooter that uses voxel octrees for their level format.

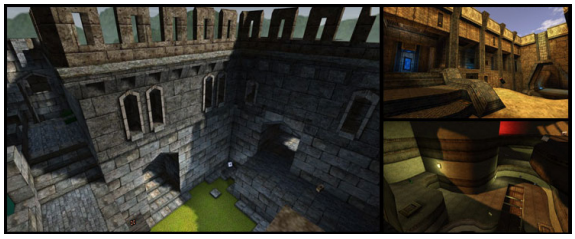


Figure 7: Illustration of Cube2: Sauerbraten, from Oortmerssen et al., 2004

8 Other approaches to 3d video encoding

There exists many non-octree approaches to 3d video encoding. These approaches can generally be categorized into three different groups depending on their approach.

One common approach is to work with "proxy meshes" - pre-defined meshes which are then transformed in some manner. A normal issue in this camp is dealing with textures - when combining captures from multiple cameras it is normal that the resulting textures may appear blurry. (Casas et al., 2015) is an example of such a paper, which solves the texturing problem using optical flow. However, this camp is in generall uninteresting to our goal, as a proxy model makes the solutions incompatible with general video encoding without known models to pick from.

Another approach is to generate videos from novel viewpoints by blending together videos from multiple viewpoints. This can be called IBR, Image Based Rendering, and is often considered to be pioneered by papers such as the lumigraph paper (Cohen et al., 1996). (Zitnick et al., 2004) a more modern paper on this subject - they use color segmentation to segment objects believed to exist on the same depth layer. They then use these color segments to build a depth map for each camera, which can be used to create novel views of the scene by blending together parts from different views.

The last approach is different variations on voxel grids. An example is OpenVDB (Museth, 2013), a B+tree-inspired format. It is a "hierarchical data structure and a suite of tools for the efficient manipulation of sparse, time-varying, volumetric data discretized on three-dimensional grids" *OpenVDB - about* 2023. The biggest difference between the OpenVDB approach and octrees is the resulting tree depths - OpenVDB allows for a configurable number of children per node, allowing for faster access due to less pointer dereferencing and more sequential access. On the other hand, octrees are fixed to 8 possible children. In reality, OpenVDB can be seen as a hybrid grid/octree solution, consisting of a few layers of pointers followed by a layer containing only leaf nodes.

This is good for its intended usecase - efficient storage of baked/computed volumetric data later to be used when rendering production movies. However, this approach makes it inflexible for captured video: The number of pointer layers is fixed compile-time, and the "infinite level of detail" benefit is completely gone.

9 3d video in commercial industries

4DViews provides HOLOSYS, a video capture system that generates textured meshes. This seems to be the current golden standard, being a commercial product that has seen use in pop music videos and similar. It can also easily be used in video games and other realtime software through Unity and Unreal Engine plugins (*4Dviews - Volumetric video capture technology* 2023).

Sony is also working on volumetric capture techniques that can produce traditional meshes as output. This makes them useable in a traditional 3d workflow, an important feature (Shrout, 2008). Additionally, Unity provides a product called Metacast - which is used to deliver volumetric video of sports events. However, none of these approaches utilize octrees as far as we know.

10 Existing works on video encoding using SVO's

While SVO's have received much attention in the research scene, not much work has been put into video encoding using it. There are some exceptions, however.

(Kämpe et al., 2016) is the closest to what we are doing, being similar in exploiting temporal and spatial consistency by encoding multiple SVOs in a single Direct Ascyclic Graph. Their results are impressive, but note that their lack of lossy encoding makes their method weak against datasets based on real life data. This is something they note themselves, and something that is a suggested future work.

Their kinect dataset, which is the only real-life based dataset for which they provided the number of points in the source material, has on average 126k points according to them. With this amount of data, they are able to encode SVOs at 11.2 megabytes per second, and 5.15 megabytes per second with a reduced DAG. Note that they use 24 frames per second, while this master will use 30.

There is also the work of the MPEG Standardization group, who are developing V-PCC Jang et al., 2019 as a solution to encoding 3d video as a sequence of pointclouds. V-PCC works by recursively dividing the area containing points into smaller boxes until there are sufficiently few points within the subdivided bounding box. The contents of the boxes box are then orthogonally projected onto each of the 6 sides of the box, and the resulting images are stitched together and treated as normal 2-dimensional frames in a video. This allows V-PCC to borrow from the decades of work on 2d video encoding.

There are two major downsides to this approach. Primarily, the biggest downside is that compression artifacts cause geometric distortions. Works such as Akhtar et al., 2021 attempt to mitigate this issue. You also risk losing details if the rendering boxes are too big, as some points may be occluded by others from some angles.

In many ways, this paper is mostly a successor to (Kämpe et al., 2016). However, the use of lossy image encoding is similar to that of (Jang et al., 2019), with the main difference being that our solution only uses this encoding scheme for color data, while V-PCC uses it for spatial encoding as well.

Part III

Design

In this section we will discuss the planning of the software project. This includes a risk assessment and awareness of certain project constraints. We will also discuss design decisions, wanted features and qualities, and plan a timeline for the implementation of the Software.

11 Risk assessment

No matter the project, there are always risks. With this thesis aiming to implement novel solutions, the risk is even larger. In fact, according to (Oates, 2005), what we are doing is the design and creation research strategy. One of the main downsides to this strategy is the additional risk. As they put it in (Oates, 2005, p. 122), "*It is risky if you do not have the necessary technical or artistic skills. Enthusiasm is no substitute.*"

There are several risks that need to be considered. Proper risk management is important in order to reduce unexpected situations. In the following sections, we will discuss the main risks we considered, and how we chose to work to reduce them.

11.1 No significant benefits are found over the current state of the art, time is wasted

What we are doing is implementing novel technology. However, until we have a working prototype, we cannot guarantee that there are any benefits of the method we are implementing.

However, we are not completely in the dark. What we ended up doing to guard us against this risk, was to perform literature research. By looking at papers doing similar things to us, we could get an idea of the performance to be expected from our solution. This is also helpful in that it allows us to build on earlier papers, saving research and implementation time by allowing us to base our software on earlier experiences by other researchers.

While we are not able to fully guard ourselves against this risk, performing this measure will help us feel confident in our work as we design and implement it. While we can't completely guard ourselves against the risk of our our experiment being a step in the wrong direction, we can at least get an idea of what we can expect from similar studies. As Oates puts it, the literature review provides the foundation for our research (Oates, 2005, p. 73).

11.2 Software is not completed by the deadline

The thesis has a deadline, and it is according to the university only possible to get an extension if you have an extremely good reason. The worst thing that can happen should the project not be completed on time is that the thesis could be failed. This is considered the worst outcome for us as a lot of time has to be re-invested in order to retry later. There is also the argument that by not completing on time, or by finishing with something subpar in order to deliver on time, we are not contributing as good quality findings to the scientific literature as we should.

Again, proper literature research and knowledge of the field is important to avoid this. Proper scheduling as done in Section 15 is also useful as it allows us to detect that we are behind schedule as fast as possible by comparing actual implementation time to what we expected. However, it is simply a tool for spotting issues - it cannot magically make us more time. What we do with the knowledge that we are behind schedule is also important. In our case, we would solve being behind schedule by re-defining goals in order to reduce the scope of the project.

11.3 External factors

Sometimes, unexpected things can happen outside of the project. Since it is developed by humans, factors such as personal ambition, working towards long-term personal goals, natural disasters, and loss of close friends or family may lead to less time being able to be put towards development.

This is simply a fact of life and nothing you can completely guard yourself against. We solve it by planning in some extra margin, such that unforeseen events could appear without jeopartizing the project.

12 Stakeholders

12.1 Me

I am the main stakeholder of this project, as my degree is dependent on it being successful, at least to the point of being able to deliver an acceptable thesis. There are many possible consequences I may have to face depending on the outcome of this thesis. For example, I may fail to graduate on time, which hurts my current career trajectory and visa situation as I intend to keep working in Japan after graduation. On the other side of things, if the research has not been done properly(f.ex by not properly respecting privacy regulations) there may be organizational or even legal punishments down the line.

12.2 Advisors

I am advised by Nobuyuki Umetani(JP) and Jon Ynge Hardeberg(NO). While they are not actively partaking in most of the design and implementation outside of providing advice, their name is on the thesis and therefore bear some of the responsibility for what the thesis contains. As researchers, these people are also dependent on metrics like paper count when their work is assessed, and as such the failure to finish a thesis may hurt them in this manner as well.

12.3 Research organizations

Especially NTNU, but also the University of Tokyo, may in some manner face consequences if this thesis ends up being problematic. Outside of this, as a student of NTNU, they have a stake in my thesis being successful for the sake of statistics that could be used to determine university-related policies in Norway and direct funding towards them.

12.4 The industry and society

Lack of proper assessments in this paper may have both positive or negative consequences for the industry and society. If the thesis ends up presenting Octree video as something better than it is, we risk other researchers wasting time and investors wasting money trying to build on the method. In addition, we risk bad derivative work being pushed onto consumers, which may give them a worse experience than necessary if another method was used for 3d video encoding.

If the method is better than this thesis is able to prove, society may lose out on a good solution to a problem, which comes at a cost of unnecessary use of any resource the solution ends up utilizing better than other state of the art solutions.

13 Existing work

In a precursor to this thesis, work has been put into writing software to assist with the capture and rendering of voxel octrees. The work was put in as a part of another course, *IT3915*, and therefore does not count as work performed on the thesis for the sake of avoiding self-plagiarising. However, an overview of the system will be presented here for the purpose of context as to what systems we were using and their capabilities.

13.1 Overview

The voxel octree capture system consists of multiple tools that together form an entire capture pipeline. The pipeline is able to capture Pointclouds from multiple depth cameras, stitch them

together to one coherent Pointcloud, convert it to a SVO, and render it. The following is a list of the binaries that resulted from this work:

- *OctreeMasterCapture* - captures multiple Pointclouds and stitches them together.
- *ply2octree* - reads Pointclouds stored as PLY and converts them to octrees.
- *OctreeMasterPlayback* - renders an octree using Raymarching(CUDA), or as a mesh. Alternatively, it can also render the entire octree as a wireframe.

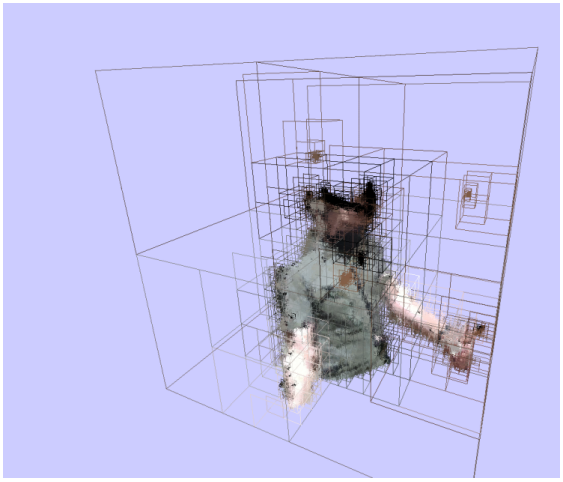


Figure 8: A screenshot from the *OctreeMasterPlayback* application rendering in wireframe mode

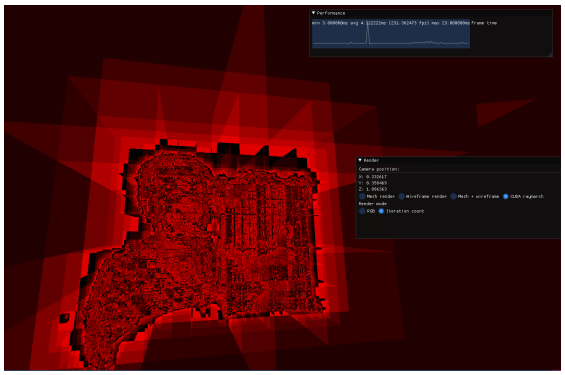


Figure 9: A screenshot from the *OctreeMasterPlayback* application in CUDA rayamrch mode, illustrating how many iterations each ray had to take before reaching a solid voxel.

13.2 Capture

A capture application was written using C++ that reads Pointclouds from multiple Intel Realsense D415 Depth cameras. The code was written such that other cameras could be introduced later. The application can use an ArUco marker to calibrate the depth cameras such that their pointclouds are all in the same coordinate system. This calibration is provided by OpenCV (Bradski, 2000)

The application writes single-frame Pointclouds to disk in the PLY file format. The ply format was chosen due to high level of adaptation with other Pointcloud software, like Cignoni et al., n.d.

13.3 Conversion

The conversion tool is an executable that takes a single PLY file and outputs an octree. To invoke it on multiple files at once, an external tool must be used. We used well-known UNIX tools such as *find* and GNU parallel (Tange, 2021):

```
find . -name "*.ply" |
parallel -j 10 "
    ~/git/uni/octree-suite/bin/ply2octree --in {} --out \$( echo {} | sed 's/ply/oct/g' )
"
```

The conversion algorithm is simple and leaves much to be desired. Consider the following psuedo-code, where *TREE_MIN* is minimum depth at which a leaf node may exist, *POINTS* are the pointcloud points, and *TREE* is the octree being built.

$TREE \leftarrow \{\}$
while $point = POINTS.next(); point \neq NULL$ **do**
  $current\_tree = TREE$
  $current\_level = 0$
  while $current\_level < TREE\_MIN or current\_tree.children.length \neq 0$ **do**
    $slot \leftarrow determine_quadrant(point, current\_tree)$
    $current\_level \leftarrow current\_level + 1$
    $current\_tree \leftarrow current\_tree.children[slot]$
    if $current\_level > TREE\_MAX$ **then** $current\_tree.add\_average(point)$ =0

This algorithm recursively subdivides an octree until there is a voxel corresponding to every point. However, if subdividing voxel further surpasses the resolution limit, it will instead average the color of all points within the region voxel that hit the limit and use this color.

13.4 Rendering

The rendering is performed either using triangles, a wireframe(Figure 8), or CUDA Raymarching(Figure 9). The former two renderers were written for debugging purposes, while the latter was meant as the final rendering solution. The CUDA Raymarcher was also heavily inspired by (Laine and Karras, 2011), notably using the same bit-flipping technique to reduce the number of comparisons that had to be made in order to look for ray-voxel intersections.

14 Functional requirements

Functionality requirements are high-level goals for the functionality of the software. Put simply, they explain what the software is expected to **do**. Even though there was only one developer, we chose to add prioritization to certain, such that "nice to have"s could be separated from requirements critical to the paper's success.

Table 2 shows the list of functional requirements, with a description and attached priority. Note that FR6 is at the bottom of the list and has the highest integer associated with it, even if its priority was 1. This is due to being discovered late on in the development cycle. This will be touched on more in Section 16.

ID	Requirement	Comments	Priority
FR1	Render animated octrees	There should be an application that can render sequential octrees after each other	1
FR2	Octree compression	There should exist an application that can load a sequence of octrees, combine them, and produce new files containing multiple octrees where similar subtrees are combined to save space	1
FR3	Frame output	The octree renderer should be able to export every frame of the octree sequence to an image such that we can create videos to use during a subjective assessment test.	2
FR4	Pre-defined animations	When exporting images as explained in FR3, it should be possible to pre-set some camera pose or movement to use for said image export, to ensure that candidates are presented equally in subjective tests	4
FR5	Rendering of competing solutions	In order to ensure a fair comparison between my thesis work and current state-of-the-art, the renderer should be able to render competing 3d video formats and export the renders using FR3.	3
FR6	DCT-encoding of color	In order to save space, we want to experiment with the application of DCT for lossily encoding octree colors	1

Table 2: Functional requirements for the project

14.1 Quality requirements

Quality requirements complement functional requirements by setting expectations for **how well** the aforementioned functionalities are to perform in different ways. The exact priority of quality requirements is very context-dependent, and depends on the use-case of the software. We will discuss two qualities that are especially wanted, some of which similar papers also touch upon.

14.1.1 Low file size

As media is primarily streamed over the internet nowadays (Stokel-Walker, 2021), it is important that the file sizes are as low as possible. This is because not everyone has a good internet connection, and cellular plans usually give you a fixed amount of data per month that you are allowed to transmit and receive. (Kämpe et al., 2016) uses file size(and its ability to compress octrees in a good manner) as one of their main measures of success.

There is also the argument of not putting a burden on society. Streaming services like Netflix have been known for years to hog up a considerable amount of the bandwidth available in the global Internet infrastructure. ISPs have even been trying to get these streaming services to pay for the traffic they are generating (Brodkin, 2023). If SVO-based 3d video is ever adopted in mainstream use, and it completely saturates the internet, our technology has hurt society.

14.1.2 Low generation time

Converting pointcloud data to SVO takes compute power. There are two main motivations for wanting lower generation time. First, if we reduce computational requirements enough, we can perform the encoding in realtime, enabling live-streaming. Current literature hints to this not

being possible at the moment, with both (Kämpe et al., 2016) and (Jang et al., 2019) being unable to perform realtime computation at the moment. However, it is definitely a wanted quality that should be aimed for.

14.1.3 Low loading time

Before SVO-based videos can be rendered, they have to be loaded. This process also takes compute time. Shortcuts taken during generation in order to minimize file size may have negative effects on load time. For example, (Kämpe et al., 2016) chose to store their octree in a predetermined order on disk in order to not have to store pointers. However, this has some negative effects on load time, as you now need to perform extra logic work to deduce correct pointers in order to make the octree traversable in a random order.

14.2 Planned timeline, use of agile development methods

Due to only being one person, agile tools like Scrum (Kniberg, 2015) were considered overkill. Scrum is a popular Agile framework in which big goals are divided into smaller tasks that developers themselves estimate the cost of. The next 2-4 weeks are then planned by sorting the smaller tasks by priority and filling the schedule until the total expected development time of all the tasks matches an expected upper limit for how much work can be done. Being a framework, Scrum also suggests teams adjust how things work to their liking. Meetings are then held at the end of every cycle to discuss what went right and what went wrong, in order to better optimize time estimations and avoid unnecessary blockers.

Since there was only one developer on this project, and only one deliverable at the end, we ended up going for a more traditional style of waterfall development. Waterfall-development is, in contrary to scrum, a style of development with longer development cycles and fewer deliverable deadlines. The biggest downside to this approach is the inability to detect blockers and other issues in a reasonable time, as deliverables are often few and far between.

The development time on hand was planned to be from January to April 2023, which corresponds to 4-8 scrum "sprints". With the number of functional requirements shown in Table 2, using scrum wouldn't really work, as there would have been too few items to convert into user stories. In addition, there were some dependencies: FR3 and FR4 were dependent on FR2. FR2 was not testable without FR1. Therefore, it made more sense to allocate a rough development time per functional requirement and then take it from there.

While we chose not to use Scrum, it is worth noting that many of the advertised benefits of agile development are very much wanted for the development of prototype software. There are many things that can go wrong, and you have to be able to modify your plan to handle these unexpected events. For example, upon implementing a feature, you might figure out it doesn't perform as expected. In this situation, experimental software development requires one of the main benefits of Agile: Being flexible and able to quickly respond to unexpected problems when developing software.

15 Schedule

The writing and research for this thesis was scheduled from the beginning of January to the 10th of July. In addition to this, there were some soft deadlines. The authors stay at the University of Tokyo ended on the 31st of May. After this date, they would be unable to use university resources such as the lab pc and the on-campus supercomputer. In addition to this, Professor Umetani's lab was a community with many skilled individuals with interests in similar fields. It was therefore important that both software development and experiments were done by the end of May in order to fully take advantage of the available resources.

Table 3 shows the rather optimistic schedule that was made for the project

January	**February**	**March**	**April**
Implement video encoder(FR2)	Implement video encoder(FR2)	Implement video renderer(FR1)	Run experiments
May	**June**	**July**	
Run experiments	Writing	Writing	

Table 3: Planned schedule of the thesis

Considering the time it takes to write a Master's thesis, and the scope of tests we wanted to perform on our method, we considered it important to dedicate a lot of time to non-programming tasks. If anything, the main author was confident in their ability to write software and wanted more leeway in places they had less experience.

As for dividing "code" into more granular components, we found this hard due to the lack of distinguishable features of the applications being developed. As discussed in Section 14, there are few requirements, and those that do exist often hang together such that it makes the most sense to develop them in tandem. Due to these reasons, we considered dividing the programming effort planning-wise into "Video encoding" and "Video rendering" sufficient. The stretch goals(not priority 1), were considered to be implicitly part of either of the aforementioned categories, should there be available time.

Part IV

Implementation

In this section we will go over the process of implementing the software as per Part III design parameters.

16 Implementation

In this section, we will discuss how implementing the planned software and encoding methods went. We will first touch on the discrepancies between the planned development timeline and the actual development time. Then, we will touch on how the software was implemented.

In total, two different lossy compression types have been implemented and will later be assessed in Part V: Subtree substitution compression, and Discrete Cosine Transform (DCT). Subtree substitution compression uses the fact that we are working with a tree data structure by attempting to find and merge similar-looking subtrees, saving space at the cost of some visual degradation to the user. Discrete cosine transform(DCT) is the same trick used by JPEG to save space (Wallace, 1992), and works by applying the discrete cosine transform to groups of nodes in the octree together. Like in JPEG, resulting values are then quantized to reduce quality in the high-noise part of the images users are less likely to notice.

16.1 Timeline

As discussed in Section 15, the implementation of the software was set to happen between January and March. In reality, this schedule was far exceeded, and the software was still being written until the middle of july. Table 4 shows the actual development schedule.

January	**February**	**March**	**April**
Implement video encoder(FR2)	Implement video encoder(FR2)	Implement video renderer(FR1)	Implement video encoder(FR2)
May	**June**	**July**	
Implement video encoder(FR2) + Run experiments	Run experiments + Writing	Run experiments + Writing	

Table 4: Actual schedule

The biggest timesink was implementing the encoding software. In total, multiple weeks were spent solving memory-related bugs in the software. In addition, it was a challenge to get the software fast enough for it to be usable. Multithreading using *pthrads* was applied to speed things up, but this was also a major contributor to the aforementioned memory bugs. After a while, the analysis tool *valgrind*(Seward, n.d.) was applied to find and fix hard-to-find bugs faster. In hindsight, using it from the beginning as a routine when testing code most likely would have saved a lot of time. Had valgrind been used earlier, some bugs could have been found early on in the implementation phase before they became part of large, complex, and hard-to-debug multi-thread state machines.

The main author's experience with C++ is partially to blame for this. While the main author has known about C++ for at least 10 years, he has no practical experience with it outside smaller video games written on a hobby basis, and C/C++ courses taken at NTNU. Even with good grades, working on small assignments doesn't give a lot of experience in writing larger long-term projects where technical decisions could cause issues in the long run.

In addition to technical issues, real-life problems also affected the schedule. This thesis was written in Japan during an exchange, and the main author found out he wanted to continue living in Japan after the end of his studies. A job hunting effort followed, the difficulty of which was orders of magnitude harder than anything previously experienced in the Norwegian market. In total, over a week was lost to job interviews, technical assignments, and other job-related activities.

Whether or not job hunting-related issues could be avoided is up for discussion. On one hand, "simply return to Norway" or "be more prepared" are both valid arguments. Job hunting in a foreign market like Japan is hard - you don't know your competition well, many companies require you to be native in the local language, and there are many people who want to get in. However,

for the author's long-term goal of working in Japan, getting a job was necessary.

16.2 Goal achievements

In Table 2 we defined the functional requirements we wanted for our application. Table 5 shows the status of these requirements at the end of the project.

ID	Requirement	Succeeded?	Priority
FR1	Render animated octrees	Yes	1
FR2	Octree compression	Yes	1
FR3	Frame output	Yes	2
FR4	Pre-defined animations	**No**	4
FR5	Rendering of competing solutions	**No**	3
FR6	DCT-encoding of color	**Yes**	1

Table 5: The functional requirements and whether or not they were implemented

As you can see, FR4 and FR5 were not implemented. This is due to time constraints and changes in the test design. We initially wanted to perform a full one-on-one perceptual test against other state-of-the-art solutions, but due to difference in rendering techiques it would quickly become a case of comparing apples and oranges. FR4 would have been a nice-to-have for the tests, but was not considered a necessity. This is reflected in the relatively low priority of 4. FR6 was implemented. In Section 14 we touched on the fact that FR6 was added on late in the development process. DCT as a feasible option was discovered in May. Making a choice on whether or not to implement it was hard. Adding more features this late in the development cycle, after the planned end date for development, is risky. However, some quick experiments found that the results were too good not to consider, and that it would be a great addition to the thesis.

16.3 Tree substitution compression

16.3.1 Octree similarity

The hardest part of this thesis was implementing tree substitution compression. The goal is to compress an octree structure consisting of nodes with an associated color in a lossy manner by removing trees that look similar. This is hard, as what "similar" means is hard to define.

There are several reasons why this is the case. First of all, there are multiple ways in which a tree can be similar. For example, two octrees can be structurally identical but have different colors. On the other hand, two octrees may be very structurally dissimilar but have the same overall look color-wise. There may exist many situations where one of the two cases is preferred over the other, but making this choice is hard. Making rules for making the choice will therefore be even harder, as you cannot rely on the same "hunch" we are using when performing experiments on human perception of quality.

Even if we were able to find a good way of comparing the structure of two octrees, there are many problems. Due to not being a uniform grid, we cannot easily compare the color of two octrees either. For example, how would one compare the color of two octrees where the intersection of their volume(i.e. the volume they share) is empty?

For the problem of calculating the similarity between two octrees, we ended up using a score system where a pair of trees were rated 0-1 on how similar they were volumetrically. In this scale, 0 means the intersection of their volumes is empty and 1 means they are completely identical. The C++ code used to calculate their similarity is attached in Section A. A quick summary of the algorithm is the following:

- Both trees have no children(leaf node)? Return 1 - completely similar
- One tree is a leaf node? Return the fill rate of the non-leaf node that is filled - the more filled the more similar
- No tree is a leaf node? Calculate the average similarity of each possible child slot:
  - Child doesn't exist for neither? Similarity is 1
  - Child exists for one of them? Similarity is $1 - fillrate$ - Completely filled means completely dissimilar
  - Child exists for both? Use this algorithm to calculate their similarity.

16.3.2 Comparison order

There is also the question of how we make the comparisons - do you compare every possible subtree(from every layer) with each other? Do you compare every node of the same level of the tree? Both imply $O(n^2)$ comparisons, meaning comparisons will get very computationally expensive if there are too many trees.

The solution we ended up with was based on multiple steps:

- Perform node pruning sequentially layer by layer, starting from the top. This allows us to skip nodes in lower layers that were already pruned as part of pruning at higher levels.
- Sort nodes into buckets based on similarity
- Find the node that is the most similar to other nodes in the same bucket
- Mark all nodes that are similar enough to the best node as pruned, recursively

For step 2, a Hash Map using the child occupancy flags of each node as key was used. As there are 8 possible child positions, and 2 possible states for each position(Occupied, Non-occupied), this results in a hash map with $2^8 = 256$ buckets. The key can be calculated by setting the bits in an 8-bit integer corresponding to the slots of the tree that have a child, while the octree is being loaded from disk.

Step 3 was implemented using a modified version of the clustering algorithm k-means. K-means works by selecting k "cluster centers" and then iteratively redefining them based on the average position of the elements in their corresponding cluster. However, due to the lack of an Euclidean problem space, we am unable to use Euclidean distance and the concept of a "cluster center". Instead, we used the function for calculating octree similarity as previously mentioned, and had a single node per cluster considered its best candidate. This increases the time complexity of k-means to n^2.

Due to the added time complexity of step 3, the whole process was written to run paralellized. This was done by inserting each bucket of the aforementioned hash map into a job list, which was then processed by a pool of threads. Step 4 was also ran during this process. Had the graph been cyclic, this could have caused race condition issues. However, due to the nature of trees, and the fact that we are simply *marking* nodes as trimmed(and specifying their replacement), a race condition would at most cause a tree to be marked as pruned several times.

16.3.3 Using CUDA to speed up comparison

At one point we refactored the comparison code such that it could run in CUDA. However, this gave no speedups whatsoever. In most cases, the results were identical or slightly slower than our CPU-only comparisons. This makes sense, as GPUs are generally memory-bound once you start accessing main memory from the compute kernel. One also has to factor in the fact that all the needed data has to be transferred to the GPU. We ended up removing the CUDA code due to it not being helpful.

16.4 Lossy color compression using DCT

Discrete Cosine Transform(DCT) is a method by which data is converted to a list of weights that when applied to cosine functions produces an estimation of the original input. It is famously used by JPEG (Wallace, 1992). After applying DCT to our storage format, it was found that even an imperfect implementation greatly reduced file size. Our implementation works like this:

- Converts color data from RGB to YUV
- Applies DCT to color data in groups of 64 bytes. We treat them as an 8x8 image even though there is no correlation on the y-axis.
- Quantizes the result, reducing the resolution on high-frequency signals
- Compression using ZLIB

16.4.1 RGB to YUV

YUV is a different color space from RGB, meaning it is a different coordinate system for representing colors. RGB uses the three primary colors Red, Green, and Blue as axes. On the other hand, YUV encodes color with one axis corresponding to Luminance(Y, Darkness/lightness) and two axes of color(UV). This is beneficial to image encoding, as empirical tests show there is more entropy along the Luminance axis (Wallace, 1992). This is backed by the fact that the human vision system is optimized for recognizing contrast.

In our dataset, the U and V color arrays post-compresson are around 50% more compressed than the Y array. This indicates that the U and V layers have less entropy, and are therefore possible to compress to a smaller size.

16.4.2 DCT

We use the same DCT type as JPEG, DCT-II (Wallace, 1992). The algorithm is run separately on the Y, U, and V channels. The main difference between us and JPEG is the existence of a Y-axis in the source data. While JPEG divides the image to be compressed into 8x8 chunks, we simply divide the array of Y, U, and V color data for each layer into chunks of 64 bytes that are treated as 8x8 images. Additionally, we do not subsample the U and V layers.

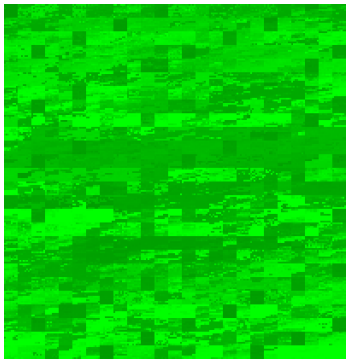


Figure 10: A montage of 8x8 DCT-encoded blocks from the Y color channel of layer 10. Taken from a chunk in the *wave* dataset, scaled 400% using nearest neighbor

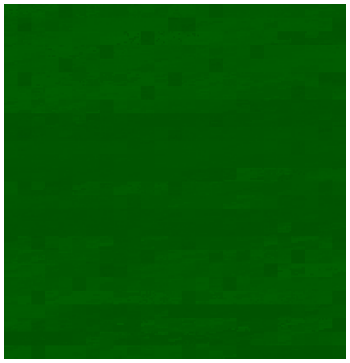


Figure 11: A montage of 8x8 DCT-encoded blocks from the U color channel of layer 10. Taken from a chunk in the *wave* dataset, scaled 400% using nearest neighbor

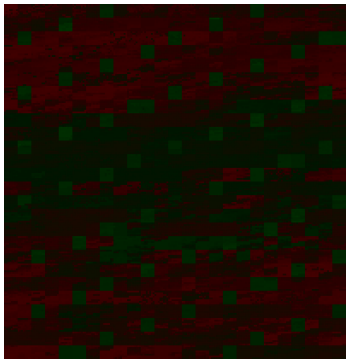


Figure 12: A montage of 8x8 DCT-encoded blocks from the V color channel of layer 10. Taken from a chunk in the *wave* dataset, scaled 400% using nearest neighbor

Not organizing color data such that there is a correlation on the Y axis has an obvious downside - there may be a lot of noise on that axis. However, in practice, spatial coherence in colors is strong enough for it to not be a showstopper. Figure 10, Figure 11, and Figure 12 show examples of the data we are DCT-encoding. As you can see, most 8x8 squares have some kind of consistency. However, there are also 8x8 squares with a lot of noise and sudden color changes. These areas are hard for DCT to accurately encode, and we should therefore aim to remove them in a later version.

A solution to the aforementioned y-axis alignment issue is to perform DCT on a per-node basis on every third layer, only performing the transformation on points where there is a voxel. Nodes on layers that don't have a DCT encoding could get their color by averaging the color of their DCT-having children. We believe this would yield great results, but did not have time to implement a prototype.

16.4.3 Quantization

Quantization is performed by dividing the DCT-transformed signals by a predetermined number, effectively limiting the number of discrete signal levels a signal can have. Each signal can have a different divisor, usually with low-frequency signals having lower divisors and therefore being allowed more signal levels. In our case, we build a lookup table with divisors for each signal by linearly interpolating between a wanted lowest for the low-frequency signals, and a wanted highest number for high-frequency signals.

This is different from jpeg, where 100 quantization tables are pre-determined and used to give the user a more user-friendly compression quality range selection range from 1-100. We strayed from this approach due to time constraints.

16.4.4 Compression

Compression was done using zlib (Gailly and Adler, 2022) as it is a well-known compression library that was easily accessible and therefore low-risk to use given the time constraints. Compression is important, as it is where the space saving is we take the quantized data with most high-frequency signals discarded, and compress it by using the fact that most of the quantized table is zeroes. JPEG uses Run-length encoding followed by Huffman encoding. We found zlib to be better in our use-case than simply using run-length encoding and ran out of time to consider any sort of Huffman encoding (Huffman, 1952).

17 Optimizations

The base data structure used by our octree video file format is very wasteful, and can be optimized in several ways. Howver, due to to time constraints, we decided on implementing optimizations in a best-effort manner after proof of concept was already working. This meant that optimizations were mostly done in the May-June months.

17.1 Original data structure

This subsection goes over the original file format used to store octree videos before any optimizations were made.

Figure 13 is a description of the original file format used to store octree, in pseudo c. Note that every child pointer is a 32-bit integer, and that 3 bytes are used to store color. Note also the fields `childCount` and `childFlags`, which are redundant and can be deduced at load-time. However, we will not count the removal of these fields as an optimization as they could have been removed from the start. Figure 14 is an illustration of how Figure 13 looks in memory. Figure 15 shows how the data structure will look in memory when we are finished with this section.

```
#define TREE_DEPTH 20
#define OCTREE_POSSIBLE_CHILD_COUNT 8

struct OctreeVideoHeader {
  int magic;
  int tree_depth;
  int headerSize;
  int layerSizes[TREE_DEPTH];
};

struct OctreeVideoLayer {
  OctreeVideoNode nodes[layerSize];
};

struct OctreeVideoNode {
  char r;
  char g;
  char b;
  char childCount; // Number of children
  char childFlags; // Bitmask indicating what child pointers are not null
  char leafFlags; // Bitmask indicating what children are leaf nodes
  int childPointers[OCTREE_POSSIBLE_CHILD_COUNT];
};
```

Figure 13: Original octree data structure

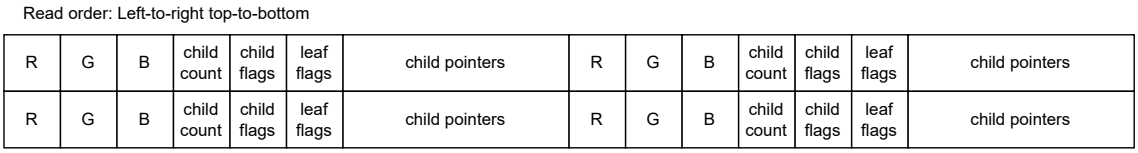


Figure 14: An illustration of how data is laid out in memory as in Figure 13

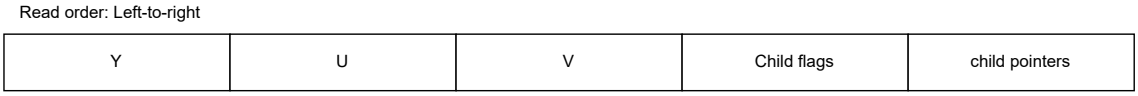


Figure 15: An illustration of how the data structure a layer with a few nodes will look in memory with the modifications discussed in this section

In this structure, `OctreeVideoNode` is the structure of the actual nodes in the tree. `r`, `g`, and `b` encode the node color. Even non-leaf nodes have a color, in order to standardize the data structure. It also allows us to render more coarse versions of the octree to save on compute resources, should it be wanted. `childCount` is the number of children of a node. It is not necessary and can be derived by calculating the population count of the `childFlags` field. The `childFlags` is used as 8 binary flags representing whether or not each of the 8 possible child slots contains a child or not. `leafFlags` is similar to `childFlags`, but instead of referring to the existence of a child or not, each bit represents wether or not the child is a leaf(i.e has no children) or not.

`childPointers` contains pointers to the nodes children. When written to and from disk, only pointers to children are written. That is, if a node has 3 children, 3 pointers are written to disk. The `childFlags` field is used to determine how many pointers exist, and where to place them in `childPointers`.

17.2 Write order

The original file format is written the same way it is stored in-memory: node-major. This means that all the fields for a single node are written together. By writing all values for a single field type in a continuous array, we are able to better compress them, as the chance of there being repetition is greater compared to when all fields for a given node are written together. It is hard to numerically prove this as most of the optimizations mentioned in this chapter were done at the same time, and altering how data was written to disk was important to many of the tricks we will be discussing further.

Instead, we perform numerical investigations in Section 19 on the data format

17.3 Optimizing pointer sizes

If the octree is written depth first to disk, when no nodes are substituted, all child nodes exist right after another. Even with a large amount of tree substitution, most nodes still have a very short distance between their children. This can be abused.

Instead of writing the actual child pointers to disk, we can write the difference between the current and previously written child offset. When no lossy tree substitutions are made, this means the value ‘1‘ is continuously written to disk. As a result, it can be trivially compressed.

Part V

Testing and evaluation

In this part we carry out experiments and evaluate metrics of our suggested solution.

18 Dataset

For the purpose of this thesis, two datasets were made: *wave* and *walk*. These datasets were generated with the goal of testing our suggested encoding algorithm with different levels of difficulty. Sadly, the *walk* dataset was corrupted some time after we lost access to the sensors used to capture test data, and as such we could not use it for this thesis.

wave is considered the simplest dataset, and is a 2.4 second clip of a human waving recorded at 30fps for a total of 75 frames. It is considered simple as most of the video remains relatively stationary throughout the recording. *walk* was a 9.9 second clip of a human walking around at 30fps for a total frame count of 297 frames. It was considered harder to compress than *walk* as the subject is moving a lot around.

Figure 16 illustrates a frame from *wave*. The subject is wearing plain clothes, something we are aware makes compression easier, as there is more uniformity and less variation. Had there been more time, we would have prepared harder-to-compress clothing, such as clothing with advanced patterns.

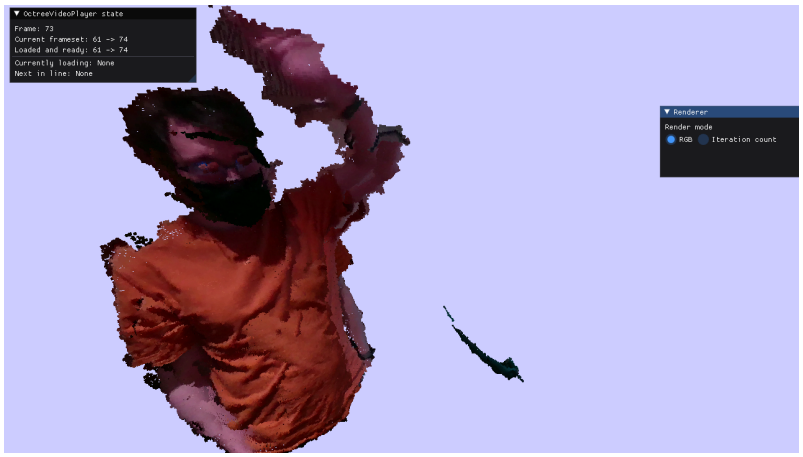


Figure 16: A still frame from the wave dataset(No lossy compression applied)

Table 6 shows the number of discrete points per frame in the source pointcloud dataset that is then converted to octrees.

	Wave	**Walk**
Min	351024	266395
Average (Rounded)	375290	498059
Max	389660	801660

Table 6: Number of points in dataset source material

Figure 17 shows the process used to generate octree video files using our solution.

Unlike other papers, like (Kämpe et al., 2016), the resolution of our dataset is dynamic. This is due to the algorithm, we used to convert pointclouds to octrees, as explained in Section 13. Figure 18 shows how many nodes we had in on average per layer in the*wave* dataset. Note that the minimum resolution was configured to 7 layers, or $2^7 = 128$ per axis. As you can see, layer 11 is the layer with most nodes, at a resolution of $2^11 = 2048$ per axis. Table 7 shows the **total** number of nodes per layer over the entire dataset.

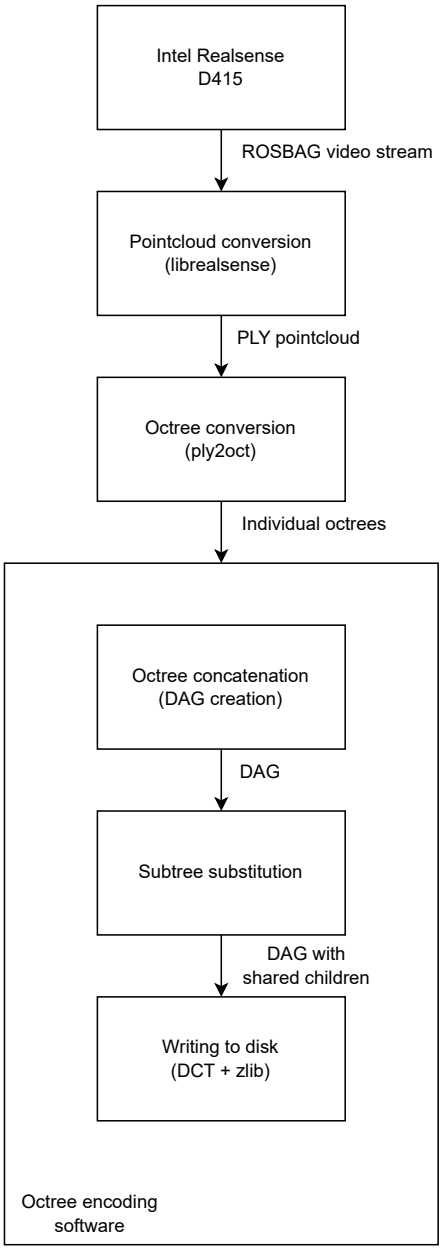


Figure 17: The processing pipeline that creates octree videos from raw Intel RealSense captures

1	**2**	**3**	**4**	**5**
75	225	1158	4791	16371
6	**7**	**8**	**9**	**10**
62034	254343	1126116	4780329	18013617
11	**12**	**13**	**14**	**15**
50406255	52650486	937506	87420	10932
16	**17**	**18**	**19**	**20**
1425	204	0	0	0

Table 7: Node count per layer over all frames in the *wave* dataset

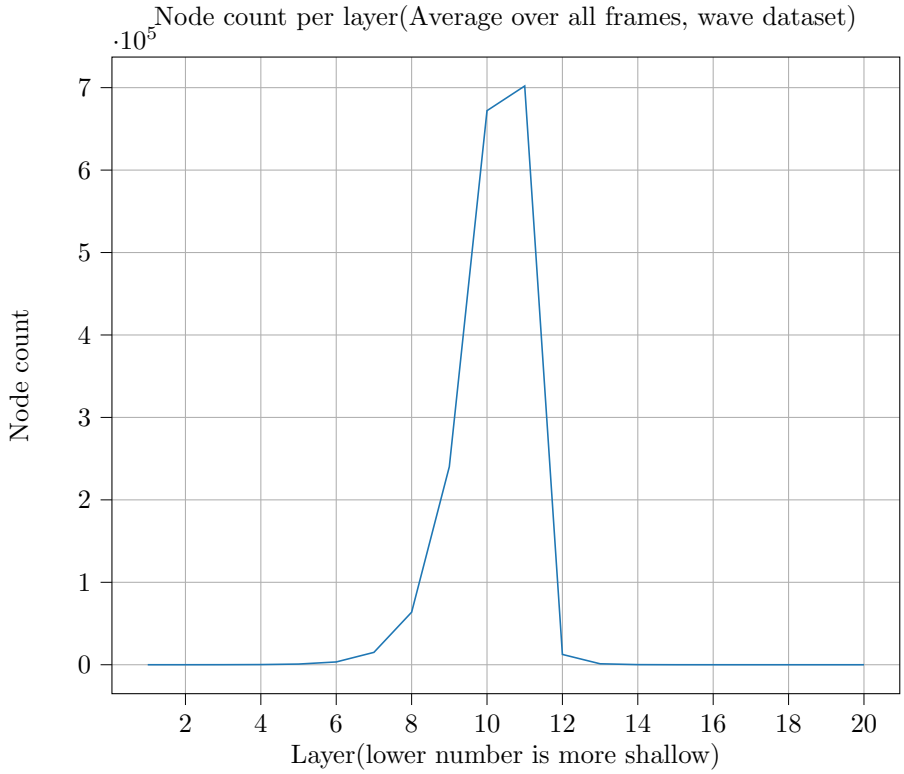


Figure 18: Chart showing the average number of nodes per frame in the *wave* dataset

19 Numeric evaluation

In this section we perform numeric evaluation based on empirical data gathered from the thesis.

19.1 File size comparison

Table 8 shows the filesizes for the two datasets, both in their original pointcloud form, and in their raw octree form. We also show the compressed filesizes using `gzip` to give a certain measure of how much entropy the files contain. The *Pointcloud* rows refer to the raw `.ply` pointcloud files, while *Raw octree* refers to the files containing octrees that are converted from `.ply`. There is one octree file per pointcloud file, and no compression is done. In our pointcloud-to-octree-video pipeline(See Figure 17), these octree files can be considered an intermediate product.

	Wave	**Rotate**
Pointcloud	403 MiB	2118 MiB
Pointcloud (Compressed, `gzip`)	352 MiB (87% of original)	1749 MiB (82% of original)
Raw octree	368 MiB	1931 MiB
Raw octree (Compressed, `gzip`)	221 MiB (60% of original)	1040 MiB (53% of original)

Table 8: File sizes of the datasets

19.2 Compression gains by number of frames per chunk

The octree substitution compression works by concatenating multilpe octrees together into one common data structure and then removing similar entries. Since the end result is then compressed, there is a theoretical compression gain through storing more trees together, as there is more data for zlib to work with.

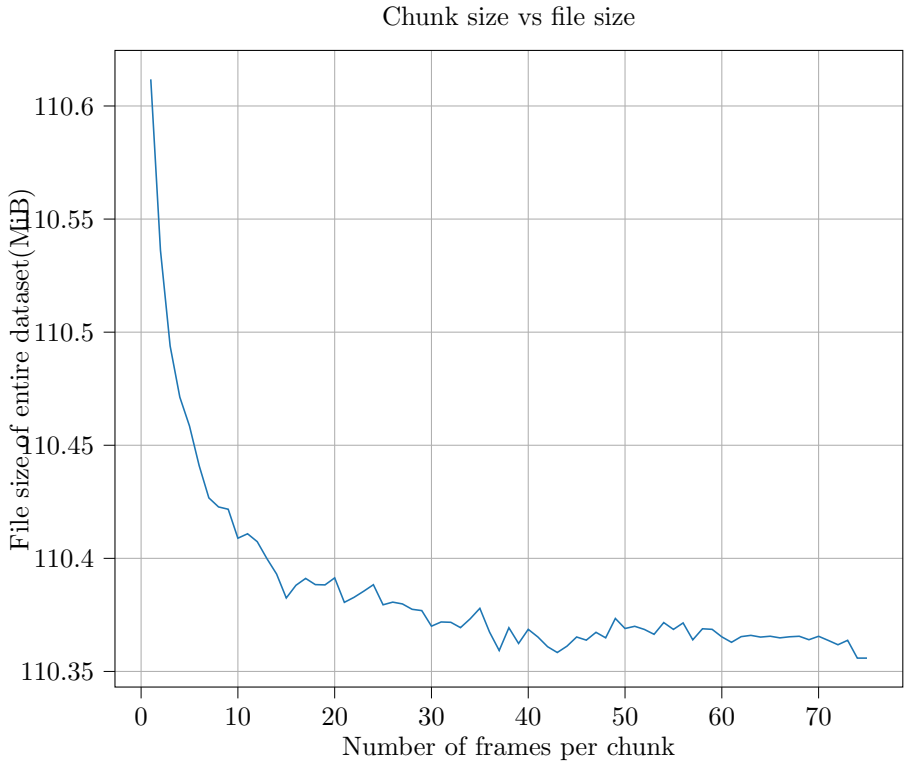


Figure 19: Chart showing the relationship between number of frames being saved together and the total filesize in MiB

We performed an experiment to measure how much chunk size affects the overall file size of the output DAG video. We performed this by iteratively running our octree encoder on the `wave` dataset with 1-75(number of frames in the dataset) frames per chunk, resulting in 75 to 2 chunks in total. We disabled substitution of similar octrees, and used the highest possible settings for DCT encoding. These settings were picked due to computational cost, as well as more data theoretically resulting in better compression results.

Figure 19 shows the amount of file shrinkage obtained depending on how many frames are compressed together. As we can see, after around 18 frames, the gain of compressing more frames together quickly disappears. After 40 frames, the line has for all practical purposes flatlined. Note that, even if compressing 20 frames together results in a file size improvement, we are still only saving 250 KiB of space per chunk, which is almost negligible.

The reason for the negligible amount of gain from compressing multiple frames is caused by the manner in which we compress the DAGs: we compress data layer-by-layer. Most layers have thousands, if not hundreds of thousands of nodes, as can be seen in Table 7. The only gain by bunching more frames together happens in the uppermost and lowermost layers(1-6 and 14-20), that in general have fewer nodes. The most extreme example is the uppermost layer, which has exactly one node, the root node, per frame in the chunk.

19.3 Child pointer compression

There are several methods for storing an octree on disk depending on needs. (Kämpe et al., 2016) Used a file structure in which no child pointers were used - instead the nodes were stored in a predetermined order and needed to be loaded into memory and have child pointers deduced before you could access the structure manually. We experimented with a scheme for encoding these pointers in a very efficient way. Figure 20 illustrates how efficient it was.

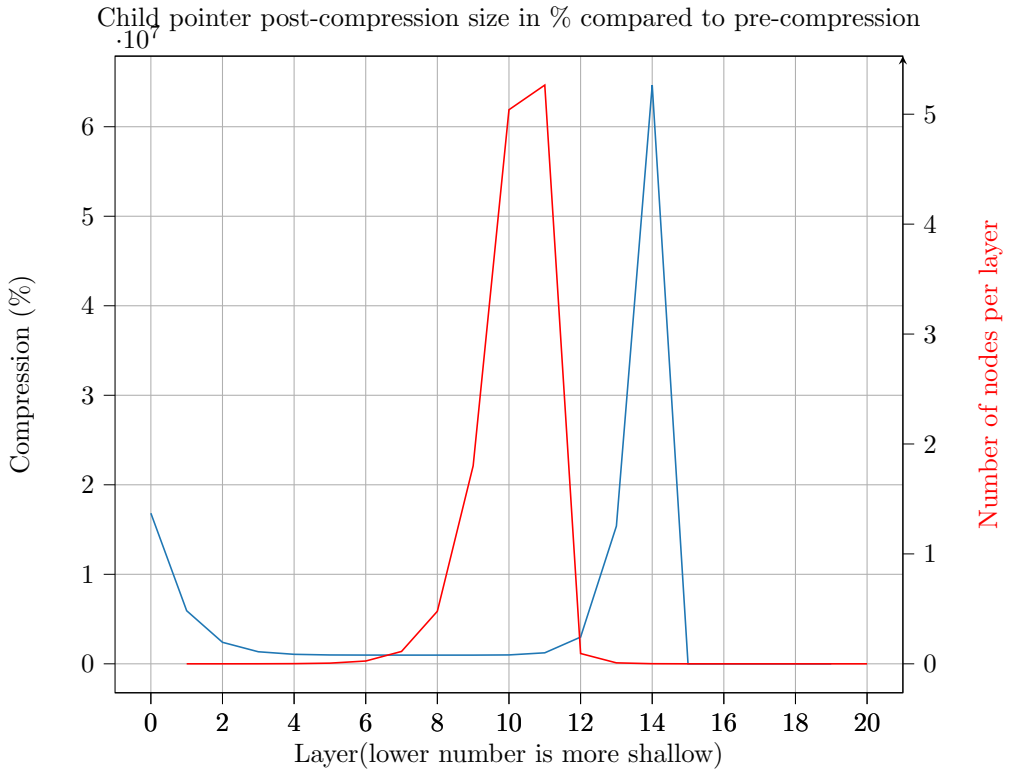


Figure 20: Chart showing the child pointer compression ratio, overlaid with the number of nodes per layer

From Figure 20, there is a reverse relationship between the compression ratio and number of nodes. This can be explained by the layers with a few nodes being harder to compress due to not having enough repetitive content. The sudden drop at layer 15 can also be explained: There are no more nodes after that point(See Table 7), and there is therefore nothing to compress.

19.4 Color data compression

Figure 21, Figure 22, and Figure 23 show the relationship between compression ratio and overall filesize for the Y, U, and V channels in our *wave* dataset compressed with DCT. Note that we use the YUV color space for color compression. While the relationship between the compression ratios varies as the size decreases, we can see that the Y channel is generally 1.5-2x less compressed than U and V. This supports the idea that most interesting detail is captured in the Y channel, something JPEG cleverly uses by downsampling the U and V channels (Wallace, 1992).

While we did not downsample the U and V channels, it may be interesting to look into for future works, especially when the spatial coherence of color data in an octree is taken into account.

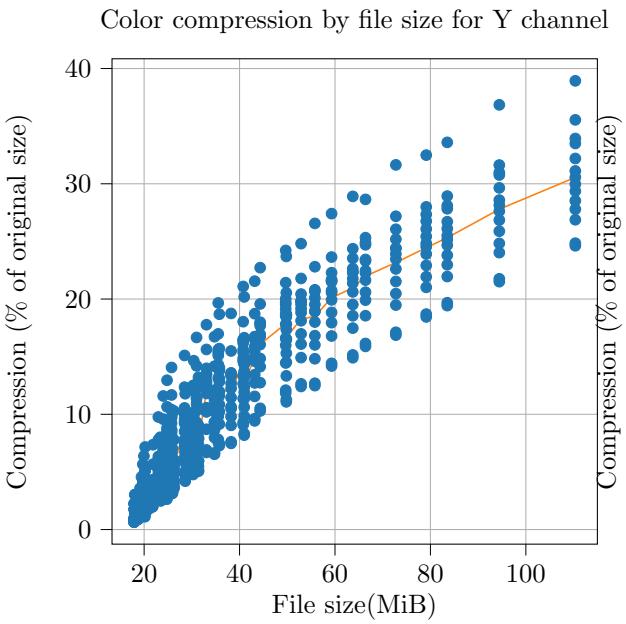


Figure 21: A chart showing how the compression ratio improved for the Y channel as file size got lower

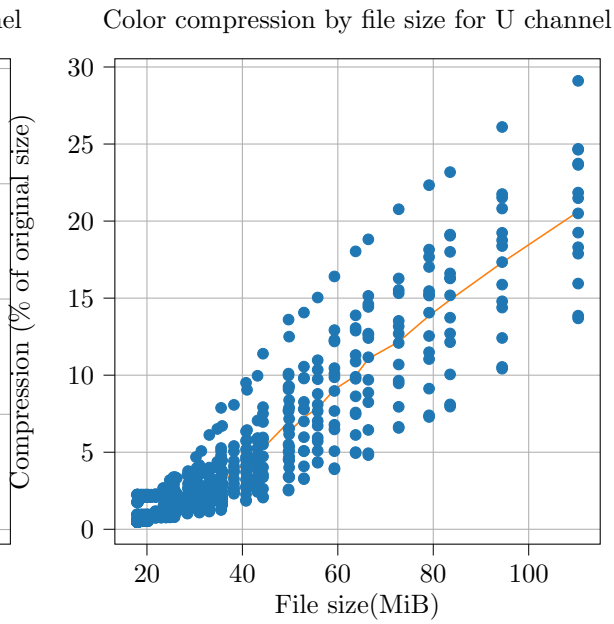


Figure 22: A chart showing how the compression ratio improved for the U channel as file size got lower

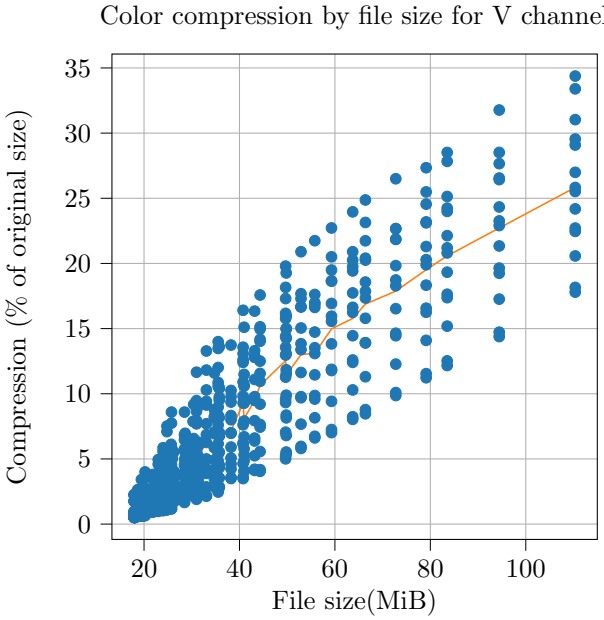


Figure 23: A chart showing how the compression ratio improved for the V channel as file size got lower

19.5 Substitution encoding speed as a function of chunk size

As mentioned earlier in Section 16.3.2, we use a $O(n^2)$ method when we search for subtrees that are similar. As a result, we expect the encoding speed to increase rapidly as the number of frames per chunk increases.

In order to visualize and verify this, we ran the encoder once using default settings with chunk sizes ranging from 1 to 17. The PC used was a laptop with a *Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz*. As a result, it is possible that the first results are slightly faster than expected due to no thermal throttling, but this is not expected to be a huge factor, as we ran using only one thread on the 6-core CPU. Figure 24 shows the results we gathered. Table 9 shows the number of seconds per chunk for chunks of size 1-6 frames

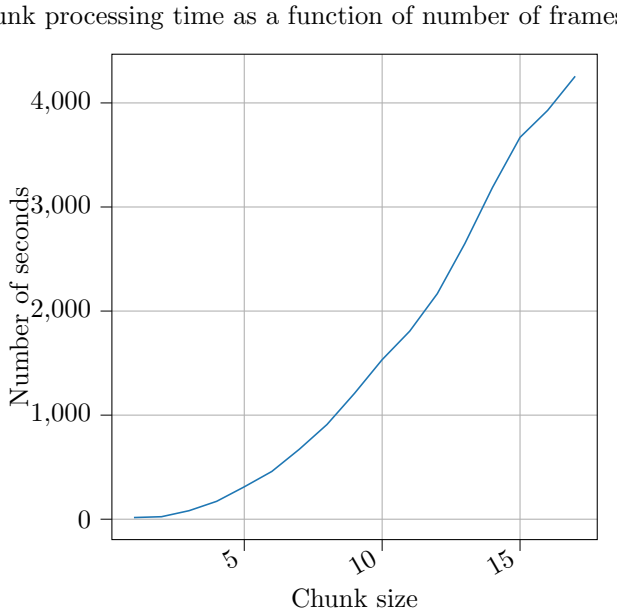


Figure 24: Chart showing encoding time as a function of number of frames per chunk

1 frame	**2 frames**	**3 frames**	**4 frames**	**5 frames**	**6 frames**
16s	24s	82s	172s	311s	459s

Table 9: Number of seconds for encoding a small number of frames per chunk(Same data as in Figure 24)

An interesting point is the shape of the graph in Figure 24. It appears exponential until around a chunk size of 10 frames. After this, the line continues in a mostly linear manner. Why this happens is unknown, but may be due to imprecision in the results caused by running on a laptop.

20 Subjective test design

As the goal of this thesis is to investigate a novel scheme for 3d video encoding, it is useful to test how artifacts of the encoding scheme is experienced by humans. For this purpose, some subjective tests were carried out. In this section, we explain how we designed our subjective tests and why we made the choices we did. We will then describe the tests we carried out, and discuss the results.

The goal of the subjective tests we carried out was to get a mapping between encoding quality and the perceived quality. We carried out two tests in total, based on the research questions defined in Section 4:

- A test of our DCT compression (**RQ2**)
- A test of our tree substitution compression (**RQ1**)

The reason for the tests being in opposite order of the research questions is one of practicality - the comptuational cost of testing **RQ2** was much lower, so it was used as a preliminary test of our test hardware. The experiment for **RQ1** was performed last-minute.

20.1 Test material generation

As we did not have a web-compatible renderer for our octree data, it was necessary to render it to some intermediate format suitable for web consumption. Another worry was the computational cost of rendering our octrees - if the powerful computer we used to render struggled, would test subjects' computers be able to render anything at an usable framerate?

Of course, this question can have serious implications for feasibility of our suggested method. However, we chose to blame limited time on the performance issues we are facing. Based on prior experience when experimenting with octree rendering, we believe better performance is very much achievable had there been more time to optimize the renderer.

With realtime rendering in the browser of of the question, the choice quickly fell on video. This decision was made early on in the process, and was defined as FR3 in Table 2. The videos were generated by taking the frames exported by our software and concatenating them together in ffmpeg. Figure 25 shows the steps taken to go from an octree video file to mp4 video files. The following command was used to build the video files:

```
ffmpeg -framerate 30 -pattern_type glob -i "$FRAME_DIR/${FOLDER_NAME}/*.png" \
    -c:v libx264 -crf 10 -pix_fmt yuv420p "${VIDEO_DIR}/${FOLDER_NAME}.mp4"
```

The bash variables come from this copy-pasted command being part of a bash script. As often seems customary, we will now explain the ffmpeg command line flags used:

- `-pattern_type glob -i` configures ffmpeg to load frames from disk using a provided glob pattern.
- `-c:v libx264` specifies the encoder - a H.264 implementation
- `-crf 10` specifies the *constant rate factor* (*Encode/H.264 - Ffmpeg* 2022-09-23), the "*recommended rate control mode for most uses*". A rate factor of 10 is 11 values better than the default of 21. Having completely lossless video was considered to generate too big video files, so a compromise was selected by picking a CRF value low enough that it was impossible to see any major compression artifacts.
- `-pix_fmt yuv420p` Specifies the pixel format of the output video. *yuv420p* is a chroma subsampling scheme.
- `"$VIDEO_DIR/$FOLDER_NAME.mp4"` output filename

20.2 Test tool

Finding a testing tool was harder than expected, and multiple candidates were considered, but our requirements made it hard to make a good fit. The following requirements were put in place after performing experiment design:

- Needed to support video playback - if only still images are shown we are unable to test for visual artifacts that may be more visible when they vary a lot frame-by-frame.
- Needed to support randomization of samples - we want to show each user the same videos in a different order to reduce the chance of known bias sources such as lack of familiarity with the dataset.

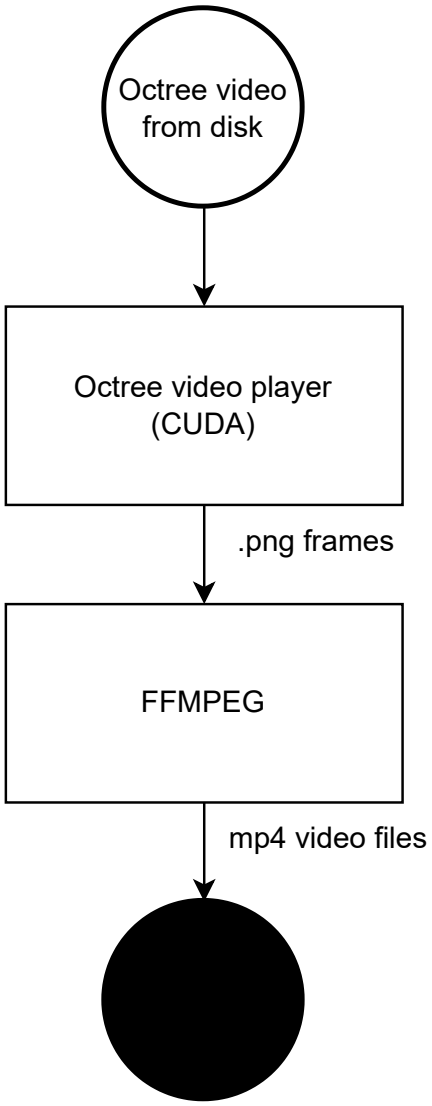


Figure 25: The processing pipeline that converts octree video files to mp4 files suitable for an online test

- The ability to perform some kind of training - We want to show the user some examples of what to expect before the actual test starts in order to combat the aforementioned biases.
- Needs to be ran online - due to the situation surrounding the exchange this thesis was written during, the main author physical access to campus before the thesis was finished, and needed to be able to gather test data online. This has an added benefit of being able to run the experiments on even more people.

QuickEval (Ngo et al., 2015) is a Psychometric image evaluation tool designed by the Colorlab at NTNU Gjøvik - the lab of the Norwegian supervisor for this paper. It is a great tool for carrying out subjective experiments on images. However, it has no video support, and therefore had to be dropped.

Another worthy mention is Versus (Vuong et al., 2018), an online tool that even supports being used together with Mechanical Turk (*Amazon Mechanical Turk* 2023). Mechanical Turk is a tool by Amazon that lets you pay humans a small sum to do small tasks for you. Mechanical Turk has a great history within this field, being especially popular to label data for machine learning purposes. However, Versus is made for 2AFC and does not support video out of the box. Even if Versus did support video, 2AFC was considered too complex for our use case, as it requires the user to make a lot of comparisons. We considered this to be too much effort for our test subjects.

Google Forms (*Google Forms* 2023) could have worked, but did not support randomizing pairs of videos and input fields, and was therefore also dropped. Forms does have a feature that allows you to randomize the order of questions. However, if done, it caused all videos to appear at the top of the form, with all questions then following in random order. Google Apps Script could be used to generate forms where videos and questions are correctly randomized in a pairwise manner. However, this would require every test taker to use a unique form, which makes distributing the test harder.

In the end, a custom solution for video testing was developed. There is no reason to reinvent the wheel, but after a long search for a fitting evaluation tool, nothing was found that met all our requirements. The sunk costs fallacy was also an item of consideration - at one point it would be more productive to make a tool that could do the job rather than keep looking.

The custom solution was written in an afternoon - less time than was spent looking for an evaluation tool. As a guide to avoiding common experiment pitfalls, (Del Pin and Amirshahi, 2022) was used as a guide. To make deployment and management easier, extra requirements were added. Table 10 shows the functional requirements we gave ourselves for the development of this test tool.

ID	**Requirement**	**Comments**
FTR1	Video playback	We want to show video recordings of our encoding scheme
FTR2	Randomization of test videos	By randomizing the order of videos being shown to the test subjects, we reduce the effect of biases such as the effect of users slowly learning how the dataset looks (Del Pin and Amirshahi, 2022)
FTR3	Training support	Needs to be able to show users test videos before the experiment begins. Needs to clearly indicate that they are being trained on how things are supposed to look.
FTR4	Accessible from online	Needs to run in a web browser and should be accessible through the internet, as we didn't have reliable access to a big amount of test subjects.
FTR5	Configurable through a simple configuration file	It should be simple to configure aspects of the test, such that we can iteratively work on our test up until we show it to actual users.
FTR6	Easy to deploy	We had access to bare metal hardware that already ran Docker and a HTTP Reverse Proxy chat could handle TLS termination. If the software could be packaged as a single HTTP web server in a Docker container, it could be deployed in minutes using existing infrastructure.
FTR7	Ability to ensure the video is shown properly	Today, many people use mobile devices. As a result, if the URL to the test is shared on social media or other platforms, it is likely that users will visit the test site from a mobile device. As these devices often have limited resolution and are often small, they are not optimal devices for performing visual assessments. We, therefore, want the ability to stop users with too small screens from participating in the test.

Table 10: Functional requirements for the test tool. **NOTE:** These requirements are not ordered, as they were all considered equally critical to be implemented.

It is easy to argue that the lack of a prioritization of the tasks in Table 10 is foolish, as it puts you in an "all or nothing" situation where you either implement everything or end up with a product that lacks critical features yet has other less critical(in the eyes of someone - user? stakeholder?) items implemented. However, in this case, the choice of no prioritization fell on the opinion that the failure to implement any of the wanted goals would make the tool useless for our use case. Had we not been able to implement everything, other tools that also lacked in certain areas yet were better developed over a longer time frame would have been better choices.

`FTR1`, `FTR2`, `FTR3`, and `FTR4` were all original requirements when we were looking at other alternatives. If any of these were not fulfilled, other options would have probably been better. `FTR5` and `FTR6` were added to avoid slowing us down by causing big amounts of extra work. `FTR5` was added as it was important that the tool was fast to use. `FTR6` was added as spending a lot of time getting the tool into production was unwanted. `FTR7` was added as a "nice to have" that we could enforce due to owning the test platform. While it was not critical to the test's execution, it has an effect of the quality of the results we gathered.

In the end, a SvelteKit (Harris et al., 2022) web application was developed using the requirements in Table 10 as a guide. The choice of SvelteKit fell on the main author having used Svelte (Harris et al., 2016) for Single Page Apps in the past, and knowing SvelteKit allowed a similar workflow to Svelte with added server-side rendering and business logic capabilities without the need to write a separate web server.

The web application worked by using a UUID provided as part of the URL the user would visit the site with as a lookup key for finding a `json`-based manifest file stored in the filesystem on the server. This manifest file configures the web browser for the test, containing information on what videos to show, what documents to show, etc. Section C shows the manifest files used for the two subjective tests. The manifest file supports configuring the following parameters:

- Title
- Author and contact information, which is shown before the experiment. Once the experiment is complete, the contact information is again shown to the subject.
- Filename of explanation file - An explanation of the test about to be performed is written in markdown, loaded from the server, and sent to the client upon loading of the manifest. It is then rendered as HTML.
- Range configuration - how many points, and what to label the first and last point as.
- Minimum screen size
- List of training videos with title and description, that are to be shown chronologically
- List of video files to run the experiment on, to be randomized

`FTR7` Was implemented using the aforementioned "minimum screen size" field in the manifest file. We ended up settling on making sure there were enough pixels on the browser's HTML viewport. The logic behind this decision was that if the viewport is wider than the videos being shown, there is most likely space to show the videos at their full resolution, which we considered the most important.

Experiment results are submitted by storing the rated subjective quality of each video in a JavaScript object, and transmitting it to the server using a HTTP POST request serialized as JSON. On the server, the data is validated to be JSON and then dumped to a directory on disk using a server-determined random filename. These steps were made to avoid Unrestricted File upload-related security vulnerabilities, as described by OWASP (Dalili et al., 2023). While a database could be used, there was no expectation of this data needing to be read again by the web server at a later date. Additionally, as this data would later be programmatically manipulated in order to generate graphs, it was thought that keeping the data in a machine-readable format made it easier to make graphs later on.

The test tool is Open-Source and is available at https://github.com/petterroea/videotest

20.3 Test flow

We will now discuss the flow followed by the test subject during the experiment. The test consists of 4 phases:

- Introduction
- Training
- Experiment
- Submission

20.4 Introduction

The user arrives at the test by following an URL. They are shown a document that outlines the test, giving some context and instructions. This document is the `explanation file` as described earlier and is customized on a per-test basis by us. The goal of this stage is to provide the test subject with the knowledge required to properly carry out the test (Del Pin and Amirshahi, 2022), as well as reassure them that they are not taking an exam and that they should follow their gut feeling in order to provide the most accurate results. The user may proceed with the experiment by pressing a button labeled **OK** located at the bottom of the document.

20.5 Training

The user is sequentially shown a set of videos, one by one. Above each video, a title and description explaining how the video fits in the dataset is shown. For example, a video could be labeled "100% compressed" with the description "This video is 100% compressed and should be considered the worst-case". The user can proceed to the next video by clicking a button labeled **OK**, but this button does not appear until the video has been played in its entirety.

In our case, we use two videos for training - the highest compressed example we had, and the lowest compressed video. We chose to not provide an example of a "medium compressed" video or similar as it is hard to determine what "medium" or "50%" means - it could refer to what is perceived by the user as half-way compressed, or it could mean the file size is halfway between the minimum and maximum compression example. By not giving an example of a "medium compressed" video, we make the user use their own perception to determine "medium".

20.6 Experiment

The user is shown videos in a similar manner to during training. However, instead of a title per video, the current test progress is shown. Unlike the training stage, the user is also given a slider they can use to rate the video on a Likert scale from 1-7. For each video, the test subject is forced to watch through the entirety before they are allowed to select their rating. They can proceed to the next video by clicking a button labeled **OK**

20.7 Submission

After all the videos have been viewed and rated, pressing the aforementioned **GO** button takes the user to a screen where they are asked to submit the data. The intention of this screen is to give the test subject the ability to back out from submitting their results before anything is sent to the server. Upon pressing an **Submit** button, data is uploaded to the server, and the user is prompted with a final screen.

This screen provides the author's e-mail address in case of questions and tells them that they can safely close their browser window.

21 DCT subjective test

This subjective experiment was done to answer **RQ2**: *How feasible is DCT for lossy encoding of color information in a SVO?*. Most of the test design is explained in Section 20. The explanatory document is attached in Section B.1.

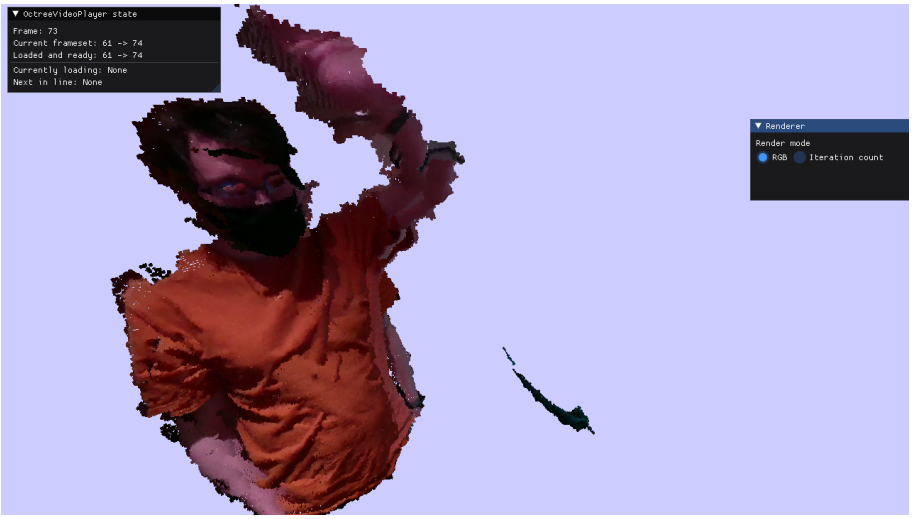


Figure 26: A frame from the DCT test, with no quantization (Same as Figure 16)

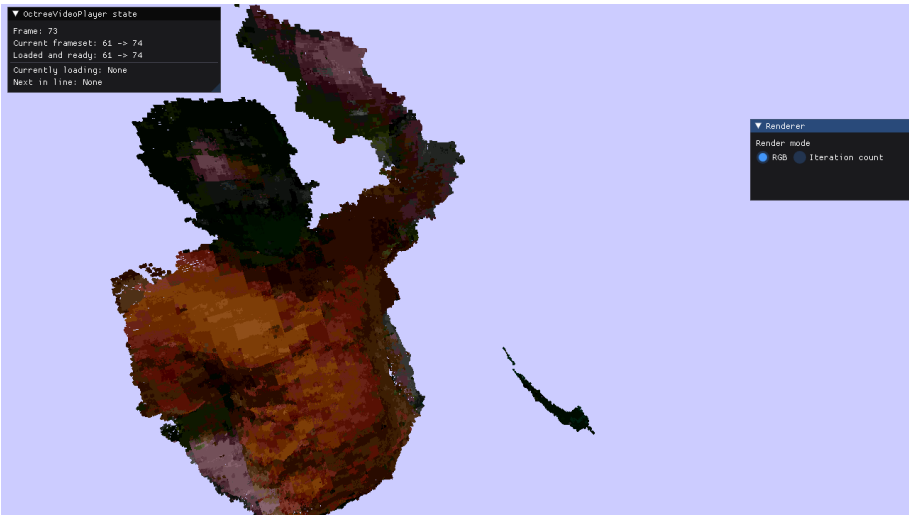


Figure 27: The same frame, but with the maximum amount of quantization applied.

The test subjects were shown a range of videos rendered from our octree rendering software, with varying degrees of quantization applied to DCT encoding. This affects the file size, as it is in the quantization step that detail is discarded. Recordings with the best possible(Figure 26) and worst possible(Figure 27) quantization settings were shown to the user prior to the experiment, in order to train them on how the dataset would look.

Figure 28 shows the relationship between perceived quality and the file size of the whole animation. Note that it is the zlib compression of the quantized data that actually reduces the file size, as this is where we eliminate repetitive patterns of zeroes caused by high-frequency data being reduced to

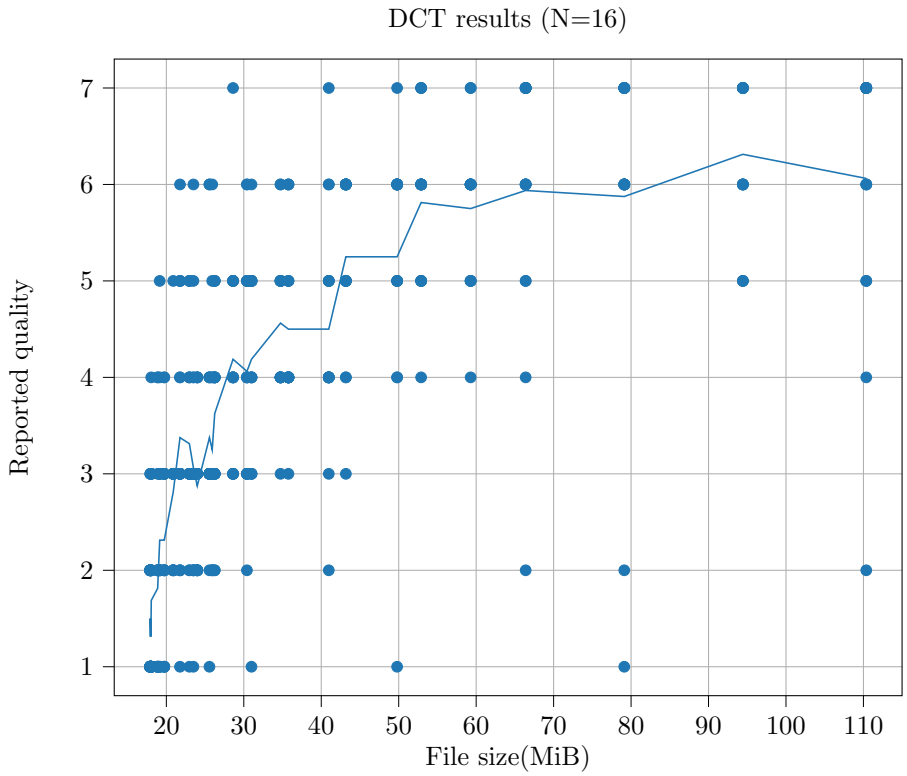


Figure 28: Chart showing line plot and scatter points for the relationship between file size and perceived quality for our DCT test

0. You can also see Figure 19 for an overview of how the file size is affected by compression when there is little data to compress.

One thing that immediately sticks out is that the test subjects, on average, never really rated anything in the dataset more than 6. It almost seems as if the training we did was inefficient, or that the users rebelled against our suggestion of "best quality".

This abnormality can be blamed on the dataset, which due to being captured on consumer-grade cameras had a considerable amount of noise, especially around the edges. One user made contact to complain about the dataset's quality, claiming it "made it hard to figure out how bad the compression was". As giving further guidance would skew the result, no extra guidance was provided to said user other than to reinforce the fact that they were to respond using their gut feeling.

The dataset could have been cleaned up more, but this is something we lacked the necessary time to do. In hindsight, using a premade dataset from a professional source could have given us better results.

When it comes to the appearance of compression, it seems we can reduce the file size by around 50% before humans notice. From there, the quality drops rapidly. Had there been time to implement DCT in a manner that better accounted for spatial coherence, it is very possible that better numbers could have been achieved. We leave this as a possible future work.

22 Substitution subjective test

This subjective experiment was done to answer **RQ1**: *How feasible is lossy subtree substitution for lossy encoding of SVOs?*. Most of the test design is explained in Section 20. The explanatory document is attached in Section B.2.

This dataset was built by running a script that encoded the *wave* dataset once for each permutation of two lists of pre-defined values that affect our tree substitution: color importance and nearness factor. The nearness factor is how similar two trees need to be substituted, and color importance is how much the nearness is "punished" depending on how dissimilar they are in color. We used 10 frames per chunk when building the dataset for these tests, as we considered it a good balance between giving the substitution algorithm enough nodes to search through in order to find good matches, and required compute time. As we showed in Figure 24, increasing the number of frames per chunk greatly increases compute time, of which we had little at this point in the thesis process.

Sadly, an unexpected bug was hit during the encoding phase, and most of our encoding processes did not complete. Of those that did complete, most of them had almost no nodes substituted. Due to time constraints, we opted not to fix the bug, as it could have been a week of debugging away. Instead, we opted to continue with the small dataset we had, removing most of the encoded data with similar file sizes. What we were left with were 6 compressed octrees with file sizes ranging from 38 to 110MB. The number of participants in this test was low. This was also caused by a lack of time.

Another issue met during test generation was the difficulty of tuning the compression parameters. While we ran many encodings, we found many of them to either not remove nodes at all, or remove too many nodes(60+%), like in the example we have with the least quality. We spent a considerable amount of time trying to tune these parameters to get good in-between results, but even if we were able to do so, the aforementioned bug stopped most of these encoding runs from completing.

Figure 29: A frame from the substitution test, with no quantization

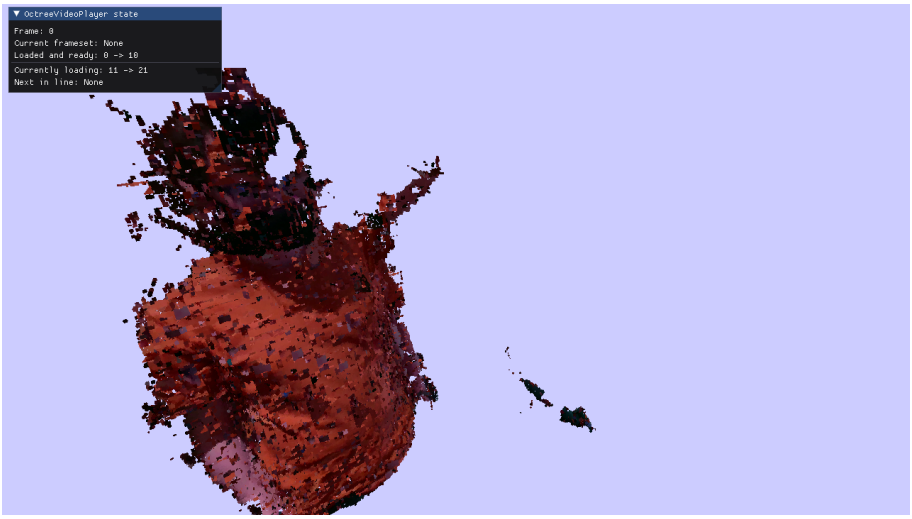


Figure 30: The same frame, but with a lot of substitutions(60%) being made

This lack of test data is bad. However, given the almost instantaneous drop in perceived quality, we believe this test says enough about our method. Figure 31 shows the results we collected. Unlike with Figure 28, the perceived quality drops off almost instantaneously.

It is possible to argue that the immediate drop in quality from the 110MB output to the 105MB output is due to human factors. It could be that the test subjects felt the need to mark anything not "perfect" with a lower grade than 7. However, we can see that quality grades 3 and 4 were used for the 105MB output, while the 110MB output never saw a grade lower than 5.

Another possible issue is the fact that the best entry in the dataset was still rated as low as 5. The same video was shown to the test subject during the training phase. We, therefore, suggest this to be caused by ineffective training or a bad dataset. Section 21 experienced a similar phenomenon, where even the best video was given a 6 on average.

One fact that was interesting was how the octrees looked at a high amount of substitution. It is clear that the differentiation algorithm did work, as you would often see certain parts of the subject's body be completely still for the duration of a 10-frame chunk, while other parts were moving. This shows that the algorithm was successfully detecting similar parts and merging them.

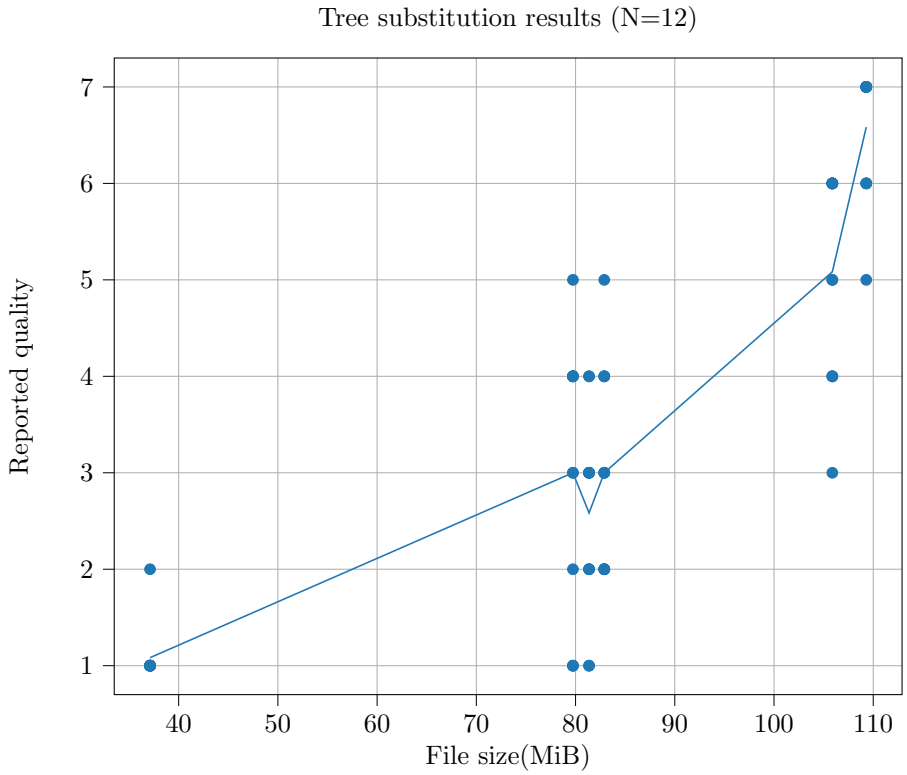


Figure 31: Chart showing how perceived quality decreases as the file size of the tree substituted data decreases.

Part VI

Discussion and conclusion

23 Discussion

23.1 RQ1 and RQ2 - wrapping up our experiments

In Section 4 we discussed two main RQs we wanted answers to, which we later researched in Part V. We will now consider what we believe the answer to these research questions is.

23.1.1 RQ1

The results from the **RQ1** experiment(Section 22) were underwhelming. Unlike in **RQ2** (Section 21), the perceived quality fell off almost immediately. We believe this is a result of our similarity algorithm, which is unable to take into account octrees that appear similar but mostly occupy slightly different parts of the voxel area being compared (See Section 16.3).

Another issue is the color similarity. As we only use the average color of all children when comparing, we are unable to properly take into account high-frequency noise and small details. As a result, many substitutions where there is a tiny bit of skin can be seen on the clothing, and vice versa, in the test data we generated.

However, this does not completely dispel tree substitution as a viable method. (Kämpe et al., 2016) showed that lossless tree substitution was a viable method, even for real-life data. Instead, we believe our methods should be worked on. For example, it may be better to use a Signed distance field when comparing two trees, as it would allow you to treat two octrees with no volume in common as similar if the volume they occupy is similar but not the same.

Additionally, some benefits may be gained from using other techniques for comparing colors. For example, DCT-encoding subtrees could help, as it would allow us to compare the amount of different signal frequencies in the color of the child node. This would be useful, as high-noise colors would better stand out in a scheme like this, compared to just using the average color of all children.

23.1.2 RQ2

For **RQ2** we seem to have gotten satisfactory results - a compression rate of 50% without humans noticing is a good start. It is important to consider that the dataset quality may have affected these results, as noise in the base data may have stopped test subjects from spotting compression artifacts earlier. Note also that our DCT implementation is relatively rudimentary, as discussed in Section 16.4.

Had we had time to implement DCT in a way that better utilized the spatial nature of the data, we think even better compression is possible. Had the dataset been more noisy(i.e the subject was wearing more noisy clothes), it is also possible that compression artifacts would have been hidden better, improving the perceived quality.

We originally went for SVO late in the development cycle, and we believe we made the right choice by doing so. This was a good find, and a promising subject to keep investigating. We therefore conclude that **DCT is a feasible solution for performing lossy encoding of SVOs**.

23.2 The effect of branching factors - comparing with VDB

As implied by (Museth, 2013), octrees are an inferior data structure for deep trees due to their branching factor. The dataset we used is 10 levels deep at many places, meaning at least 10 pointer dereferences is needed for rendering. If you factor in the fact that the Raymarcher may have to march great portions of the tree in order to hit anything, it is clear that we are talking about a lot of pointer references. However, there are multiple possible compromises that could be made, depending on needs.

(Museth, 2013) suggests that increasing the Branching factor gives benefits over octrees. In fact, VDB often only needs a 3-4 layers to store a considerable amount of data, due to their large Branching factor. However, this comes as the cost of memory - each "node" in VDB has many children, 64++. With this kind of configuration, if only one child is actually populated, a node may be extremely memory-wasteful. It is however important to note that in such a situation, the pointer list would still compress well. This means that the cost would mostly be in RAM after the file is loaded, not in file size.

The solution probably lies in a compromise. We can take inspiration from VDB, by storing the first N layers in nodes with many children, and then use octrees to encode additional detail beyond these many-child nodes. This could give amortized performance similar to that of VDB, with the added benefit of supporting infinite detail as with normal octrees. Figure 32 illustrates how this could look.

The actual gain of using a VDB-octree-hybrid would heavily depend on how often the octree part of the structure is hit, as well as GPU-related issues like branch divergence caused by needing new logic in order to handle octree traversal in some cases. In addition, it is important to consider that VDB details like layer count and node size are determined compile-time based on the needs of the project. Determining this compile-time adds extra branching to the code, which may reduce any possible gains brought forward by this improvement.

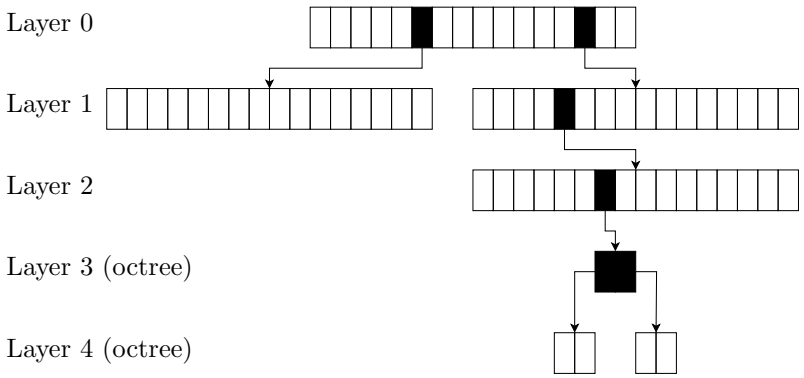


Figure 32: A possible VDB/octree compromise? Illustration with simplified nodes.

VDB is not a lossy format. It's main goal was to efficiently store simulation data for production movies. However, by introducing some of the qualities of VDB into our solution, we believe it may be possible to gain some additional benefits.

23.3 Our method vs others - encoding efficiency

(Kämpe et al., 2016) is the most similar other method, and is able to present much lower file sizes than us - why? The only dataset for which they provide number of points, *Kinect*, has on average 126k points per frame. With this amount of data, they achieve 54.9 mbit/s at 24fps. This is 33% of the size of our *wave* dataset, which averages 375k points per frame. At the lowest filesize at which we can perform DCT without humans noticing any difference, the whole dataset is 50MB. This is 16 Megabytes per second at 24 FPS, or 128 megabits/s. With 45.9 mbit/s being 42% less than 128 megabits/s, the fact is that we are actually able to compress more data per second, not taking into account that we, unlike them, still encode pointers in the file. Our DCT method is therefore comparable to other state of the art solutions.

23.4 The computational cost

Just like (Kämpe et al., 2016), our method is not fast enough to be used in real-time. In fact, it is currently quite far from it. The same goes for (Jang et al., 2019). It is important to note that 2d encoding is also computationally expensive, with hardware encoding often being utilized. Still, the

amount of nodes we have per layer are nothing compared to the amount of pixels in an image(with the maximum number of nodes in one frame of the *wave* dataset being 389660, while a 4k image has around 8 million pixels - 21 times the amount of color data.

It is clear that the computational cost is still a huge issue and that future work should look into how this cost can be reduced. From our experience, the biggest bottleneck is memory - evidenced by the fact that we tried moving our entire encoding pipeline to CUDA with no benefit, as discussed in Section 16.3.3. This is further evidence that a B+-tree-based compromise as discussed in Section 23.2 may be a way forward, as it allows us to greatly reduce pointer dereferencing.

23.5 Meta-discussion: The thesis process

In the end, the planned and actual development time ended up being very different. The difference between Table 4 and Table 3 is large, and it is clear that the project was very big for one person to work on alone. However, we don't believe incorrect use of developer methodology was to blame, meaning we are satisfied with our planning. Even if it took time before we realized we were short of time, the changes we would have done would have happened during the planning phase of the project, before a line of code was written. Had we done this again we would have relied more on earlier work and reduced the scope of the project to a simpler one.

The biggest take-away from this project as a developer has been to play it safe, understand your limits, and stand on the shoulder of giants as much as you can.

24 Conclusion

In this thesis, we have demonstrated a working system for performing lossy octree compression in two distinct manners. We have also performed experiments on octree data output from encodings made using these techniques. We believe the use of DCT for compression of color in octree videos, and substitution of trees in a lossy manner using a similarity function novel, and consider these our contribution to the scientific literature. This includes the implementation details, and the experience we gained and documented in this thesis. The tools developed for this thesis, as well as tools written earlier in order to support the thesis, are available at https://github.com/petterroea/Octree-suite

We have also contributed a simple tool, for performing subjective experiments with videos over the internet. Its source code can be fetched from https://github.com/petterroea/videotest

While there were many deviations from the development plan, both in terms of time schedule and in features implemented, we find these to be acceptable and consider the end-result satisfactory. There are things that could have been done differently, and we have learned from this. Our method is not perfect and can for example not be used to perform live encoding of 3d video. However, when comparing with similar solutions, we conclude that we have contributed novel methods that are feasible and deserve further research.

We also contributed three major items which we suggest could be further investigated:

- Using DCT in SVOs
- Tree substitution using better comparison methods.
- Octree-b+tree hybrids.

25 Future works

As mentioned in Section 23.2, we believe investigating a OpenVDB-inspired B+-tree Octree hybrid may be worthwhile. There are many downsides to using octrees that can be mitigated by using

the best of both worlds.

In addition to this, we believe we have shown that our implemented systems are of such merit that they are worthy of being worked on further. We have shown that we can use DCT to greatly compress the size of our color data without affecting the apparent quality. Further improving how DCT is applied to SVOs could lead to even better results.

Tree substitution is also an interesting path go to down. Even if it did not give as good results as we expected, we believe there are many possible improvements that can be made by investigating novel ways of comparing the volume and color of SVOs.

Bibliography

4Dviews - Volumetric video capture technology (2023). URL: https://www.4dviews.com/volumetric-systems (visited on 9th July 2023).

Akhtar, Anique et al. (2021). 'Video-based Point Cloud Compression Artifact Removal'. In: DOI: 10.48550/ARXIV.2107.14179. URL: https://arxiv.org/abs/2107.14179.

Amazon Mechanical Turk (2023). URL: https://www.mturk.com/ (visited on 27th June 2023).

Baziyad, Mohammed, Tamer Rabie and Ibrahim Kamel (2021). 'Utilizing Spatio-Temporal Redundancy to Maximize the Performance of Video Steganography: An Octree Approach'. In: *2021 International Symposium on Networks, Computers and Communications (ISNCC)*, pp. 1–6. DOI: 10.1109/ISNCC52172.2021.9615833.

Bradski, G. (2000). 'The OpenCV Library'. In: *Dr. Dobb's Journal of Software Tools*.

Brodkin, Jon (4th Mar. 2023). 'Netflix fights attempt to make streaming firms pay for ISP network upgrades'. In: *Ars Technica*. URL: https://arstechnica.com/tech-policy/2023/03/netflix-fights-attempt-to-make-streaming-firms-pay-for-isp-network-upgrades/ (visited on 9th July 2023).

Casas, Dan et al. (Oct. 2015). '4D Model Flow: Precomputed Appearance Alignment for Real-time 4D Video Interpolation'. In: *Computer Graphics Forum (Proceedings of Pacific Graphics)* 34.7, pp. 173–182. DOI: 10.1111/cgf.12756. URL: http://richardt.name/4d-model-flow/.

Cignoni, Paolo et al. (n.d.). *MeshLab*. DOI: 10.5281/zenodo.5114037.

Cohen, Michael et al. (Aug. 1996). 'The Lumigraph'. In: Association for Computing Machinery, Inc. URL: https://www.microsoft.com/en-us/research/publication/the-lumigraph/.

Dalili, Soroush, Dirk Wetter, Landon Mayo et al. (28th Jan. 2023). *Unrestricted File Upload*. URL: https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload (visited on 1st July 2023).

Del Pin, Simon and Seyed Ali Amirshahi (2022). 'Subjective Quality Evaluation: What Can be Learnt From Cognitive Science?' In: *The 11th Colour and Visual Computing Symposium 2022*.

Encode/H.264 - Ffmpeg (2022-09-23). URL: https://trac.ffmpeg.org/wiki/Encode/H.264 (visited on 10th July 2023).

Gailly, Jean-loup and Mark Adler (13th Oct. 2022). *zlib*. URL: https://zlib.net/.

Google Forms (2023). URL: https://www.google.com/forms/about/ (visited on 1st July 2023).

Google Trends (2023). URL: https://trends.google.com/trends/ (visited on 9th July 2023).

Harris, Rich et al. (26th Nov. 2016). *Svelte*. URL: https://svelte.dev/.

Harris, Rich et al. (14th Dec. 2022). *SvelteKit*. URL: https://kit.svelte.dev/.

Huffman, David A. (1952). 'A Method for the Construction of Minimum-Redundancy Codes'. In: *Proceedings of the IRE* 40.9, pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

Jang, Euee S. et al. (2019). 'Video-Based Point-Cloud-Compression Standard in MPEG: From Evidence Collection to Committee Draft [Standards in a Nutshell]'. In: *IEEE Signal Processing Magazine* 36.3, pp. 118–123. DOI: 10.1109/MSP.2019.2900721.

Kämpe, Viktor et al. (2016). 'Exploiting Coherence in Time-Varying Voxel Data'. In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '16. Redmond, Washington: Association for Computing Machinery, pp. 15–21. ISBN: 9781450340434. DOI: 10.1145/2856400.2856413. URL: https://doi.org/10.1145/2856400.2856413.

Kazhdan, Michael, Matthew Bolitho and Hugues Hoppe (2006). 'Poisson Surface Reconstruction'. In: *Symposium on Geometry Processing*. Ed. by Alla Sheffer and Konrad Polthier. The Eurographics Association. ISBN: 3-905673-24-X. DOI: 10.2312/SGP/SGP06/061-070.

Kniberg, Henrik (2015). *Scrum and XP from the Trenches. How we do Scrum*. 2nd ed. C4Media.

Laine, Samuli and Tero Karras (2011). 'Efficient Sparse Voxel Octrees'. In: *IEEE Transactions on Visualization and Computer Graphics* 17.8, pp. 1048–1059. DOI: 10.1109/TVCG.2010.240.

Mildenhall, Ben et al. (2020). 'NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis'. In: *ECCV*.

Museth, Ken (July 2013). 'VDB: High-Resolution Sparse Volumes with Dynamic Topology'. In: *ACM Trans. Graph.* 32.3. ISSN: 0730-0301. DOI: 10.1145/2487228.2487235. URL: https://doi.org/10.1145/2487228.2487235.

Ngo, K. V. et al. (2015). *Quickeval: a web application for subjective image quality assessment, Image Quality and System Performance XII*. URL: https://www.quickeval.no/.

Oates, Briony J (2005). *Researching information systems and computing*. Sage.

Oortmerssen, Wouter van, Lee Salzman and Mike Dysart (6th May 2004). *Cube 2: Sauerbraten*. URL: https://sauerbraten.org/.

OpenVDB - about (2023). URL: https://www.openvdb.org/about/ (visited on 22nd Feb. 2023).

Rodgers, Reece (9th Jan. 2022). 'How to Watch Movies in Virtual Reality'. In: *Wired*. URL: https://www.wired.com/story/how-to-stream-movies-in-vr/ (visited on 6th Mar. 2023).

Seward, Julian (n.d.). *Valgrind*. URL: https://valgrind.org/.

Shrout, Ryan (12th Mar. 2008). *John Carmack on ID Tech 6, Ray Tracing, Consoles, Physics and more*. URL: https://pcper.com/2008/03/john-carmack-on-id-tech-6-ray-tracing-consoles-physics-and-more/ (visited on 25th July 2022).

Sisson, Patrick (2020). *How VR Training in the Workplace Is Transforming Learning on the Job*. URL: https://redshift.autodesk.com/articles/vr-training-in-the-workplace (visited on 9th July 2023).

Stokel-Walker, Chris (27th Nov. 2021). 'Why Do DVDs Still Exist?' In: *Wired*. URL: https://www.wired.com/story/why-do-dvds-exist/ (visited on 9th July 2023).

Takikawa, Towaki et al. (2021). 'Neural Geometric Level of Detail: Real-time Rendering with Implicit 3D Shapes'. In.

Tange, Ole (Aug. 2021). *GNU Parallel 20210822 ('Kabul')*. DOI: 10.5281/zenodo.5233953. URL: https://doi.org/10.5281/zenodo.5233953.

Virtual Reality Headset Market Share & Growth Report, 2030 (2023). URL: https://www.grandviewresearch.com/industry-analysis/virtual-reality-vr-headset-market (visited on 6th July 2023).

Vuong, Jenny et al. (2018). 'Versus—A tool for evaluating visualizations and image quality using a 2AFC methodology'. In: *Visual Informatics* 2.4, pp. 225–234. ISSN: 2468-502X. DOI: https://doi.org/10.1016/j.visinf.2018.12.003. URL: https://www.sciencedirect.com/science/article/pii/S2468502X18300433.

Wallace, G.K. (1992). 'The JPEG still picture compression standard'. In: *IEEE Transactions on Consumer Electronics* 38.1, pp. xviii–xxxiv. DOI: 10.1109/30.125072.

Wurm, Kai et al. (Jan. 2010). 'OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems'. In: vol. 2.

Xu, Rui et al. (2023). *Globally Consistent Normal Orientation for Point Clouds by Regularizing the Winding-Number Field*. arXiv: 2304.11605 [cs.GR].

Yu, Alex et al. (2021). 'PlenOctrees for Real-time Rendering of Neural Radiance Fields'. In: *ICCV*.

Zitnick, C. Lawrence et al. (Aug. 2004). 'High-Quality Video View Interpolation Using a Layered Representation'. In: *ACM Trans. Graph.* 23.3, pp. 600–608. ISSN: 0730-0301. DOI: 10.1145/1015706.1015766. URL: https://doi.org/10.1145/1015706.1015766.

Glossary

AR Augmented reality - technology that aims to **augment** the real life environment around you by superimposing extra information. i, 2, 3

ArUco Printable black-and-white markers. This allows software to understand where a camera is in a room, relative to the marker.. 15

Branching factor The number of children a node has. In the context of volumetric formats, a lower branching factor means needing to traverse more nodes to access the same information in a given spatial resolution.. 47

CUDA SDK for writing software that runs in parallel on NViDIA GPUs. 15, 16, 23

DAG Directed Acyclic graph - a graph where the relationship between nodes has a direction and there are no cycles. 4

DCT Discrete Cosine Transform - a lossy compression technique that decomposes a signal into cosine waves of different frequencies and amplitudes. i, ii, v, viii, 4, 5, 17, 21, 22, 24, 25, 33, 46–49

Depth camera A special type of camera which uses various techniques is able to capture the depth of what is in front of it as a 2d height map.. 15

Euclidean The "normal" coordinate space we are used to, where the distance between two points on the same axis is always uniform. 23

PLY File format for 3d data(including pointclouds) with large adoption. 15

Pointcloud A data structure which encodes points in a 3d-space. Usually points with RGB data attached, gathered from depth+RGB sensors.. 9, 14, 15

Raymarcher Rendering software using the Raymarching technique. 16, 46

Raymarching A rendering technique that simulates rays coming out of the camera, slowly moving the ray forward until it hits something. 15, 16, 52

RGB A color space often used general purpose applications. Represents color on three axes, each axis referring to of the three primary colors the human vision system is able to distinguish.. 24

Signed distance field A mathematical way of representing geometry as a continuous function where the output value is how a given coordinate is from the closest object.. 46

SVO Sparse voxel octree. 3–5, 15, 17, 18, 42, 46, 48, 49

UNIX Family of operating systems. Linux is a UNIX-*compatible* operating system. 15

VR Virtual Reality - technology that aims to **replace** the real life environment around you. i, 2, 3

YUV A color space in which color is represented by three axis, one correlating two lightness and the other two correlating to some mix of red, green, and blue. 24, 33

Appendix

A Octree similarity algorithm

A.1 Similarity

```
// **Structural** octree similarity
template <typename T>
__host__ __device__ float layeredOctreeSimilarity(layer_ptr_type lhs,
↪  layer_ptr_type rhs, int layer, T* container) {
    if(layer == OCTREE_MAX_DEPTH) {
        #ifdef __CUDA_ARCH__
            return 0.0f;
        #else
            throw "Octree is too deep";
        #endif
    }
    auto lhs_node = container->getNode(layer, lhs);
    auto rhs_node = container->getNode(layer, rhs);

    int lhs_children = lhs_node->getChildCount();
    int rhs_children = rhs_node->getChildCount();

    if(!lhs_children && !rhs_children) {
        // Both leaf nodes? 100% similar
        return 1.0f;
    } else if(lhs_children && !rhs_children) {
        // rhs has no children, meaning it's a leaf node.
        // The more space lhs occupies, the more similar the nodes are
        return layeredOctreeFillRate(lhs, layer, container);
    } else if(!lhs_children && rhs_children) {
        // Same as above but opposite
        return layeredOctreeFillRate(rhs, layer, container);
    }
    // Both nodes are populated - calculate similarity of the children
    float sum = 0.0f;
    for(int i = 0; i < 8; i++) {
        auto lhs_child = lhs_node->getChildByIdx(i);
        auto rhs_child = rhs_node->getChildByIdx(i);
        if(lhs_child == NO_NODE && rhs_child == NO_NODE) {
            // Both child spots are empty, 100% similar
            sum += 1.0f;
        } else if(lhs_child != NO_NODE && rhs_child != NO_NODE) {
            // Both child spots are occupied, get their similarity
            sum += layeredOctreeSimilarity<T>(lhs_child, rhs_child, layer+1,
↪  container);
        } else if(lhs_child != NO_NODE && rhs_child == NO_NODE) {
            // lhs child is occupied, rhs child is empty. They are 100% similar
↪  if lhs is empty
            sum += 1.0f - layeredOctreeFillRate<T>(lhs_child, layer+1,
↪  container);
        } else { //lhs_child == NO_NODE && rhs_child != NO_NODE
            // Only rhs child is occupied, fill rate is inverse of lhs fill rate
            sum += 1.0f - layeredOctreeFillRate<T>(rhs_child, layer+1,
↪  container);
        }
    }
```

```
    return sum / 8.0f;
}
```

A.2 Fillrate

```
template <typename T>
__host__ __device__ float layeredOctreeFillRate(layer_ptr_type tree, int layer,
↪   T* container) {
    if(layer == OCTREE_MAX_DEPTH) {
        #ifdef __CUDA_ARCH__
            return 0.0f;
        #else
            throw "Octree is too deep";
        #endif
    }
    auto tree_node = container->getNode(layer, tree);
    // No children? Leaf node - 100% fill
    if(!tree_node->getChildCount()) {
        return 1.0f;
    }
    /*if(
        lhs->getLeafFlags() == rhs->getLeafFlags() &&
        lhs->getChildFlags() == rhs->getChildFlags() &&
        lhs->getLeafFlags() ^ lhs->getChildFlags() == 0) {
        return 1.0f;
    } */
    auto childFlags = tree_node->getChildFlags();
    auto leafFlags = tree_node->getLeafFlags();

    //All children are leafs? Use bit magic instead
    if( leafFlags == childFlags ) {
        // How many % of the children are filled?
        return (static_cast<float>(popcount(childFlags)) / 8.0f);
    }

    // Not all children are leafs, we need to recurse
    float sum = 0.0f;
    // Add all leafs to the sum
    sum += static_cast<float>(popcount(leafFlags));

    #pragma GCC unroll 8
    for(int i = 0; i < OCTREE_SIZE; i++) {
        // Recurse all children that aren't leafs
        if(((childFlags ^ leafFlags) >> i) & 1) {
            sum += layeredOctreeFillRate<T>(tree_node->getChildByIdx(i), layer+1,
↪   container);
        }
    }
    return sum / 8.0f;
}
```

B Subjective assessment data

B.1 DCT test materials

This appendix presents information given to the user as part of the subjective assessments performed in this thesis.

This experiment aims to measure how humans perceive output from a novel 3d video system looks. The goal is to measure how much we can compress a file without a noticable degradation in perceived quality.

Please read the entire document before proceeding.

What is compression?

Compression is how we save space when storing or transferring media. There are two types of compression: Lossy and lossless. Lossless compression lets you make data smaller withut losing any information. You may have used this yourself using the `.zip` file format.

Lossy compression works by throwing away information we think you won't notice is missing. For example, `.jpeg` files as well as most video files are compressed in a lossy manner. You may have experienced lossy compression through services like Netflix and YouTube.

The following image is compressed with relatively little information being thrown away:

(*credit https://unsplash.com/photos/iMdsjoiftZo*)

The next image is compressed with most of the information thrown away. Note that while the file size is almost 10 times smaller than the original, you can still make out the subject of the image:

The easiest way to see lossy compression in action is to look up videos where confetti or similar effects are used, as they are notoriously difficult to compress in a lossy manner.

How the experiment works

You will be shown video files and asked to rate them on a scale from 1-7. On this scale, 1 means "It looks completely compressed, all detail is gone", and 7 means "It doesn't look like it was compressed at all.". For each video, use your gut feeling to decide how much you think the video is compressed.

Before the experiment starts, you will be shown 2 videos to "train" you on how the source material is supposed to look with no compression.

This experiment takes around 5 minutes.

There is no right or wrong answer - the experiment aims to learn about how you as a human perceive our data

Please note that you will download around 100MB of data by running this experiment - it may not be a good idea to partake from a cellular connection. Connecting to a WiFi network or an Ethernet cable is recommended.

B.2 Subtree substitution materials

C Subjective experiment configuration

This appendix presents the manifest files used to set up the two subjective experiments we ran.

C.1 Subjective test 1: DCT

```
{
    "title": "DCT test",
    "author": "Liam S. Crouch",
    "contact": "me@petterroea.com",
    "explanation": "lobby.md",
    "range": {
        "max_range": 7,
        "min_label": "All detail is gone",
        "max_label": "No noticeable loss of detail"
    },
    "min_screen_size": 1300,
    "training": [
        {
            "title": "0% compression",
            "description": "This video file is not compressed at all",
            "file": "videos/1_1.mp4"
        },
        {
            "title": "100% compression",
            "description": "This video file is compressed to the maximum possible
↪   extent",
            "file": "videos/128_128.mp4"
        }
    ],
    "files": [
        "videos/1_1.mp4",
        "videos/128_128.mp4",
        "videos/128_64.mp4",
        "videos/128_16.mp4",
        "videos/128_8.mp4",
        "videos/64_64.mp4",
        "videos/64_32.mp4",
        "videos/64_8.mp4",
        "videos/32_64.mp4",
        "videos/16_128.mp4",
        "videos/32_16.mp4",
        "videos/8_128.mp4",
        "videos/32_8.mp4",
        "videos/1_128.mp4",
        "videos/2_128.mp4",
        "videos/8_64.mp4",
        "videos/4_64.mp4",
        "videos/8_32.mp4",
        "videos/16_8.mp4",
        "videos/4_32.mp4",
        "videos/8_16.mp4",
        "videos/1_32.mp4",
        "videos/4_16.mp4",
```

```
        "videos/2_16.mp4",
        "videos/4_8.mp4",
        "videos/4_4.mp4",
        "videos/4_1.mp4",
        "videos/2_4.mp4",
        "videos/1_4.mp4"
    ]
}
```

C.2 Subjective test 2: Tree substitution

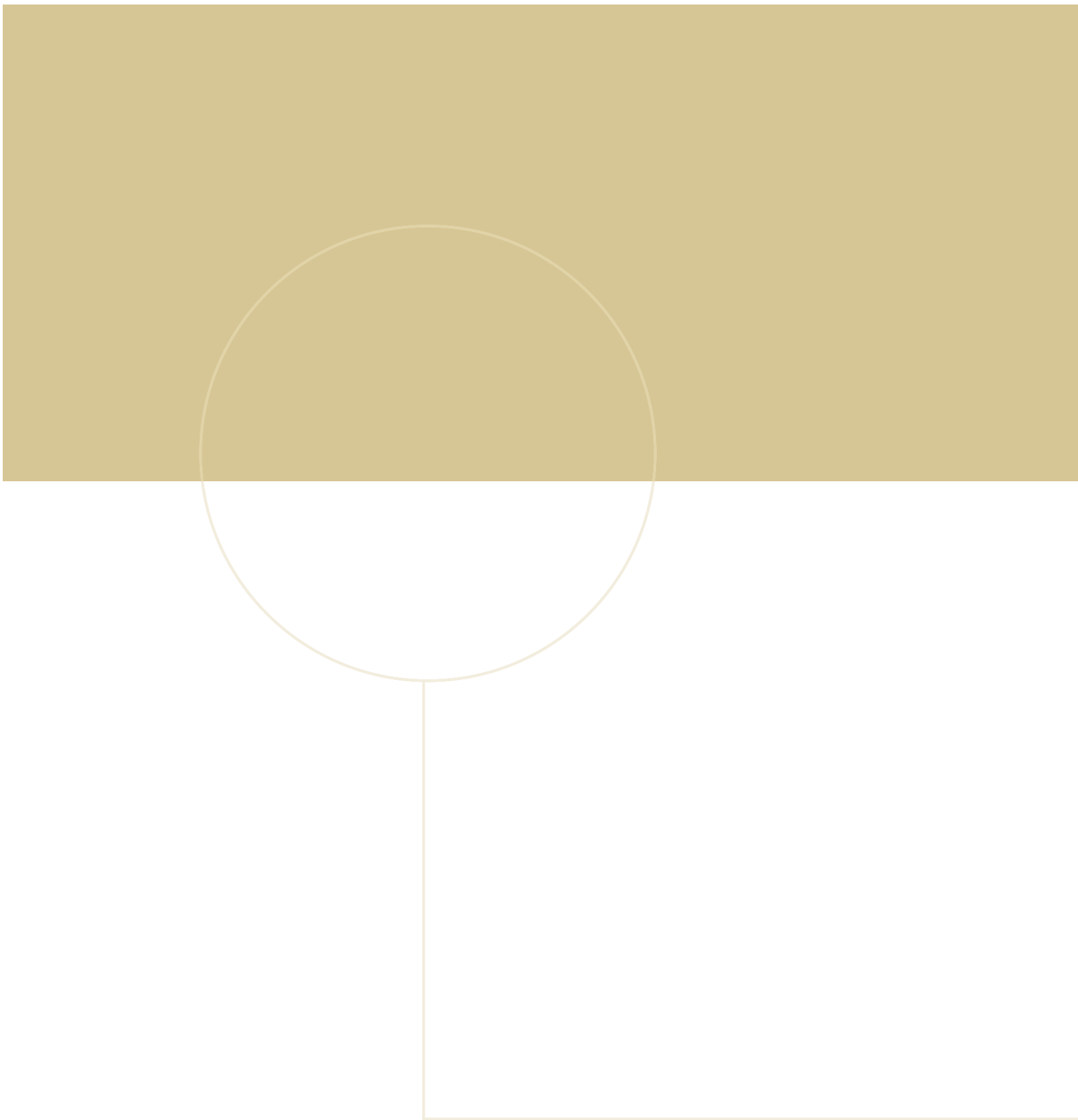
```
{
    "title": "Tree substitution test",
    "author": "Liam S. Crouch",
    "contact": "me@petterroea.com",
    "explanation": "lobby.md",
    "range": {
        "max_range": 7,
        "min_label": "Complete loss of quality",
        "max_label": "No apparent change in quality"
    },
    "min_screen_size": 1300,
    "training": [
        {
            "title": "0% compression",
            "description": "This video is not modified at all",
            "file": "videos/c0.1_n0.995.mp4"
        },
        {
            "title": "100% compression",
            "description": "This is the most modification you can expect",
            "file": "videos/c0.02_n0.9.mp4"
        }
    ],
    "files": [
        "videos/c0.03_n0.95.mp4",
        "videos/c0.05_n0.95.mp4",
        "videos/c0.02_n0.9.mp4",
        "videos/c0.09_n0.95.mp4",
        "videos/c0.01_n0.99.mp4",
        "videos/c0.1_n0.995.mp4"
    ]
}
```

D Use of Open Source Libraries

The following Open Source libraries were used in this project. We would like to thank their authors for their contribution to the open-source community, which has been of great use to us.

- OpenCV
- libRealsense2
- ImGUI
- glm

- stb
- rapidjson
- SDL2
- GLEW
- zlib
- libstdc++



NTNU

Kunnskap for en bedre verden