Lochlann Baker

Real-Time Reconstruction and Rendering of RGB+Depth Video Scenes

Part II Dissertation

Queens' College

May 13, 2024

Declaration of originality

I, Lochlann Baker of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: Lochlann Alexander Baker

Date: May 13, 2024

Proforma

Candidate Number: **2337G**

Project Title: **Real Time Reconstruction and Rendering of RGB+Depth Video Scenes**

Examination: **Computer Science Tripos - Part II - 2024**

Word count: 11,692[1]

Code line count: 14,271[2]

Project Originator: **Dr Rafal Mantiuk**

Project Supervisor: **Dr Rafal Mantiuk**

Original Aims of the Project

The original aim of the project was to, given a set of RGB and depth video streams, render the reconstructed geometry in an application where the user can view the scene at any angle.

The core aim was to implement one geometry reconstruction technique called voxel-based volume integration, and to compare this to a different technique based off ray marching. The two approaches would be compared to each under performance (refresh rate) and accuracy (differences between screenshots of the reconstruction and the original renderings) metrics. Extensions included a VR port to illustrate a use case of the work, an optimisation for both techniques, and specular surface detection and estimation for more accurate lighting results.

Work Completed

The success criteria were exceeded.

The core and some of the extensions were completed. Image-based ray marching was chosen to be the core module, and the voxel-based method was chosen to be the extension module due to its complexity. The image-based method yielded accurate results with real-time processing and data streaming. Of the extensions, modifications were made to both techniques to improve performance in both, and robustness for the image-based technique. A VR implementation not completed since the results could be sufficiently demonstrated in a desktop application, and the specular surface detection and reconstruction methods were not real-time and so were not relevant to this dissertation.

Special Difficulties

None.

[1]Computed using `texcount` on Overleaf.
[2]Submitted source code line count may be less since code will be tidied up before submission.

Contents

List of Figures

Introduction

3D reconstruction is the process in which a three-dimensional representation of an object or scene[1] is created from several 2D images, points or scans, as shown in figure 1.1. Having a holistic view of an entire object or scene allows us to gain a more intuitive grasp of the underlying structures of an object, as seen in applications such as medical imaging[1]. Being able to reconstruct a scene in real-time from several sensors has particularly seen traction recently in robotics [2], 3D teleconferencing [3] and mixed-reality (XR)[4]. Within XR applications, capturing a 3D representation of the scene is not only essential for tracking the user in their 6 degrees of freedom, but also for being able to establish a correspondence between digital and physical objects.

Such reconstructions are often computationally expensive and subject to limitations imposed by the capturing or display technology, however. For example, if only a mobile phone camera were available, then the reconstruction method can only depend on sequential RGB images as its input[5]. Conversely, a VR headset may have several RGB and depth cameras, so the reconstruction method would be different in that it can utilise several input sources[2] at once. This dissertation explores the latter use case - processing multiple RGB and depth streams at once - since the additional depth information leads to more precise reconstructions as depth estimation is not required. This comes with the challenge of maintaining real-time reconstruction speeds due to the voluminous input data requiring processing.

As such, there are several considerations in regards to generating and rendering the 3D scene:

- **Resolution**: Sampling or storing data at a higher resolution increases the level of detail we can render, but increases the processing time due to there being more data to process. This results in a tradeoff between detail and rendering speeds, which holds true across all geometry representations.
- **Artefacts**: The reconstruction process may introduce artefacts into the scene at points the capturing technology was not able to properly record[3]. These artefacts should be minimised which could require extra processing time.
- **Dynamic updating**: If videos are taken of the scene, then the changes in the setting should be reflected in the scene in real-time (i.e. without stuttering, lag or buffering). Generating a mesh animation from video frames can also introduce difficulties when objects change shape and topology, which is difficult to capture in a mesh-based representation of the scene efficiently, for example.

Previous works in this field attempt to reconcile these tradeoffs by carefully selecting the capturing and reconstruction process to best suit the specific usage scenario [3] [4], as discussed in 1.2.

[1]A *scene* in this context is defined to be a space containing one or more objects that undergo continuous action.

[2]RGB and depth are different in nature, leading to fundamentally different techniques to just RGB, largely because depth no longer needs to be estimated.

[3]which could be due to noise or inaccurate depth measurements

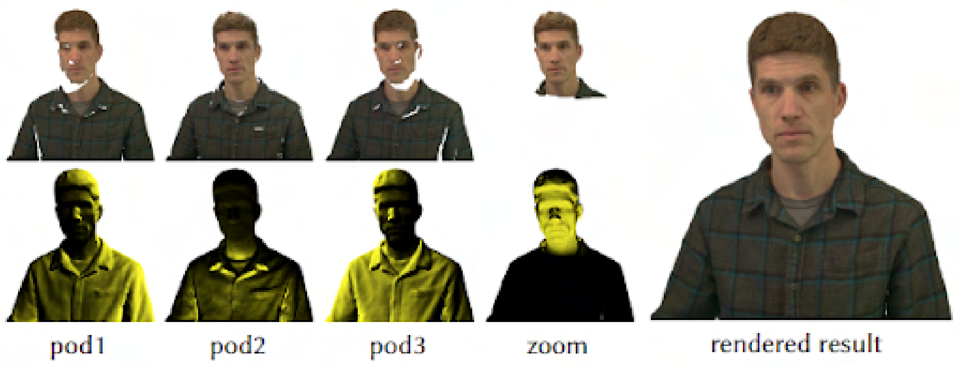


Figure 1.1: An example of 3D reconstruction, in this case using multiple RGB (top) and depth (bottom) images. They are combined to produce a final 3D rendered result (see right). Obtained from Lawrence et al. [3]

1.1 Project Summary

This project takes several RGB and depth videos of a scene as input, and outputs a 3D rendering of the reconstructed scene that the user can view from any position and angle[4].

Specifically, this dissertation focuses on real-time geometry reconstruction with several concurrent input streams of RGB and depth because of its rising significance in robotics and mixed reality. A 3D scene is rendered and updated to show the 'action' of the video, which can be displayed in a VR or desktop application on consumer hardware.

In particular, the ray marching method presented by Lawrence et al. is implemented due to its capturing process being similar to the aims of this dissertation (see §1.2) while boasting real-time results. This project also implemented a series of modifications to the original method which makes it more robust for arbitrary scene types and virtual camera positions. The voxel-based method presented by Curless and Levoy [6] is also implemented which serves as a baseline due to its high accuracy. Comparisons are made to the ground-truth of the original scenes that were recorded to see how accurate the method is in an absolute sense, in addition to the voxel-based method to see how it compares to an accurate reconstruction technique.

Both techniques are evaluated under the following metrics which aim to analyse the effectiveness of each technique in different usage scenarios:

- **Rendering speed** - the refresh rate of the output application is measured, along with the degree to which real-time video playback (i.e. not in slow motion or sped up) occurs.
- **Presence of artefacts** - a recording of the 3D scene from a desired viewpoint is created, and will be compared with a recording from the Blender scene that the scene was taken from, at the same viewpoint using PSNR measurements.

1.2 Previous Work

Previous works present reconstruction techniques which work under different capture technologies and usage scenarios, with several of them described below. I used this to determine which methods would be most applicable to this project.

[4]As such, the user can view a captured scene from more angles than what is available just through the finite number of input views.

1.2.1 Other works' use cases

Curless and Levoy[6] present a **highly accurate** geometry reconstruction technique using voxel grid integration, whose impressive results still serve as **a common baseline for comparison** in literature today despite the slow reconstruction speeds. This method has been widely used in literature for some time, so the method can be used to see how my project fares against other RGB+Depth reconstruction techniques.

Chen et al. [7] adapt this voxel-based approach to introduce **real-time reconstruction at scale**, but only for a single depth and RGB camera stream. This doesn't apply to my use case since I hope to reconstruct several RGB and depth camera streams in real-time.

Sun et al. [8] also offer a **real-time** reconstruction of a scene recorded with a single moving camera, but without the need for a depth camera. The application they create has uses in augmented reality (AR), but does not re-texture fused geometries. Whilst also real-time and sees applications in XR, it is again limited by the single camera and lack of depth information compared to my dissertation.

Overbeck et al. [4] expand on this by presenting a **real-time VR application** that renders panoramic light field stills, using a compressed video stream algorithm, at 90Hz display refresh rate. However, the reconstructed scenes can only be viewed in the convex hull of the data cameras' view frustums. My project allows the user to view the scene at any position and, since I can use depth streams, panoramic light fields aren't necessary for my project.

Mildenhall et al. [9] present a neural network-based method for **novel view synthesis** with impressive results at the expense of long training times for each scene. Requiring training for each scene means that arbitrary scene data can't be processed in real-time, which is what my project focuses on.

Lawrence et al. [3] present a **real-time** 3D telecommunications system that uses **several RGB+depth video streams** and control the lighting and setting so as to avoid 'flat' looking surfaces and specular reflections. My dissertation expands on this technique by implementing modifications which improve reconstruction quality at arbitrary positions since this paper uses a fixed virtual camera position and relies on continuous depth map geometry.

1.2.2 Comparison with my use cases

To justify the choice in reconstruction technique, the capturing and rendering requirements of this project were compared against those of the methods that already exist in literature. Table 1.1 show how Lawrence et al.'s method is the only one which fulfills all four of my requirements. Additionally, Curless and Levoy's method is sufficient for comparison on an individual-frame basis since their capturing process is compatible with my method. As such, these are the two methods I explore in this dissertation.

Previous Work	Real-time	Several concurrent cameras	RGB and depth	Videos
Curless and Levoy	✗	✓	✓	✗
Chen et al.	✓	✗	✓	✓
Sun et al.	✓	✗	✗	✓
Overbeck et al.	✓	✓	✗	✗
Mildenhall et al.	✗	✓	✗	✗
Lawrence et al.	✓	✓	✓	✓

Table 1.1: Comparison with previous works

Preparation

In this chapter, I discuss the work undertaken to prepare a suite of RGB and depth video scenes that I operated on using Blender, along with a discussion of the file formatting used. I then explain the theory behind geometry reconstruction, in particular ray marching and volume integration.

2.1 RGB with depth Video Scenes

The first piece of preliminary work we must do is to prepare a collection of video scenes, where each scene has recordings (from several camera viewpoints) of RGB and depth. There is a growing selection of RGB-Depth cameras on the market, but for the purposes of this dissertation, I chose to produce such recordings digitally on virtual scenes. This has a number of advantages.

Firstly, the measurements produced are exactly accurate. Cameras in the real world produce noise, and depth cameras may produce inaccuracies, depending on the technology used[10]. Rendering virtual scenes means I have greater control of noise and atmospheric effects, and also means I have perfect depth information. This gives a fairer test on the geometry reconstruction methods that I will be evaluating.

Secondly, I can obtain re-renderings of the digital scenes which I can use as a ground-truth comparison to my results. This is fairer than re-recording a physical scene which could change due to lighting or atmospheric effects, or camera poses not being in precisely the same place.

I found several digitally animated scenes, which are either realistic or stylized, and have multiple camera view points record the scene such that each camera produces one RGB feed and one depth feed. I imposed that the cameras do not move, which means I do not need to store a stream of camera poses, nor do I need to estimate pose on-the-fly at each frame.

2.1.1 Depth Maps

This project uses depth maps because they allow for precise reconstruction along the third dimension. A depth map is a 2D image in which the measurement of intensity (which could be colour or brightness) at each pixel represents the distance between the camera and the physical point which projects onto that pixel, as shown in figure 2.1. Internally, one can think of a depth map as a function $d : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$ from a set of discrete points to a continuous depth measurement. Depth maps are useful because they preserve the depth information to each pixel, which is lost during projection onto the camera plane (§2.1.2).

There are several rendering and reconstruction methods which don't rely on depth information, but at different expenses. Foveated light fields[11] are able to render novel views of a captured scene, but only within the viewing frustum of the capture setup used, making them unsuitable for this project where the user should be able to view the scene at any position. The capture process itself also relies on dozens of cameras in a particular assortment. Neural radiance fields also don't rely on depth, but models need to be trained on each scene, making it unsuitable for streaming as this project does.

Techniques with uncalibrated camera poses, such as those involving monocular video [8] or a single RGB and depth camera pair [7] suffer from camera drift as detected features used for tracking change between frames.

Therefore, by using depth maps with known camera poses, this project prevents all of the shortcomings of the aforementioned techniques.

By knowing both the depth information and the intrinsic and extrinsic camera parameters, we can obtain an almost perfect re-projection of points from 2D image space to 3D world space.

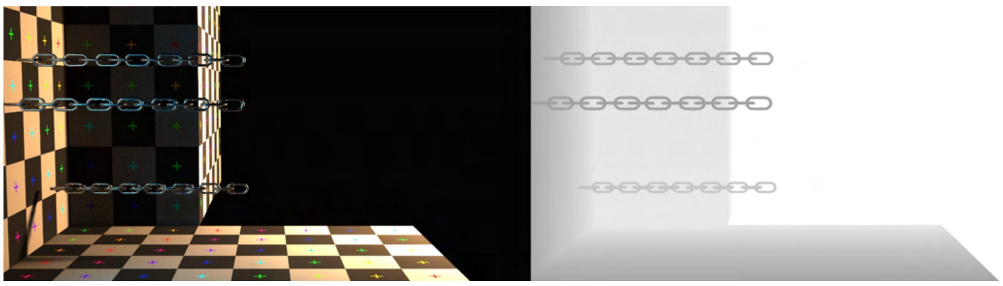


Figure 2.1: An example of a depth map. Left is the original image, and the right is the depth map, where darker is nearer to the camera.

2.1.2 Camera Models

This project requires knowledge of the type of cameras used to capture scenes so that the projection process done by the camera can be undone on each pixel to obtain the precise world coordinates needed for an accurate reconstruction.

Physical cameras are able to obtain a digital image by allowing light to pass through a lens, which focuses the light on to a sensor plane. The sensor plane measures the intensity of light and produces an electrical signal which is used to generate the digital image. Mathematically, cameras can be modelled by a transformation from a 3D world point to a 2D image point.

The *extrinsic* matrix of a camera $\mathbf{E}$ describes the pose of a camera (its position and local rotation) relative to a world-space origin. The *intrinsic* matrix of a camera $\mathbf{K}$ describes the transformation from a 3D camera view coordinate to a 2D image plane coordinate. In other words, the extrinsic matrix is likened to the view matrix, and the intrinsic matrix is likened to the projection matrix.

The pinhole camera model is an idealised camera model, which ignores distortions such as lens distortion[1] and discrete image coordinates. Its components are illustrated in figure 2.2 and its camera matrices are as follows:

$$\mathbf{E} = \mathbf{R_z R_y R_x T} \tag{2.1}$$

[1]Distortions are modelled as additional constants in the intrinsic matrix, and thus can be present in a camera while still yielding accurate projection and unprojection. Since, by default, Blender does not use distortion, this project also happens to not use lens-distorted cameras, so the derivations do not model distortion here for simplicity.

$$\mathbf{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

Where $\mathbf{R_i}$ is the rotation matrix about axis i, T is the translation matrix, f_i is the focal length along axis i (measured in mm), and c_i is the center i-coordinate.

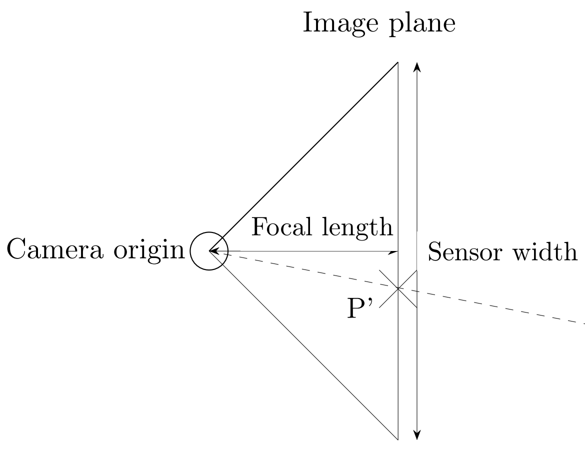


Figure 2.2: Pinhole camera model used by Blender. The camera is defined with the focal length and sensor size. A point P is projected onto the image plane at point P'.

The image coordinate components can be derived as shown below:

$$x_{img} = \frac{f_x x_{world}}{z_{world}} + c_x \tag{2.3}$$

where

$$f_x = \frac{\text{focal length} \times \text{resolution}_x}{\text{sensor width}} \tag{2.4}$$

and similarly for y_{img} and f_y.

Thus, we can ‘unproject’ camera coordinates back to 3D, given a known depth (z-coordinate), which we can do since we have the depth coordinate (in view space) from the corresponding depth map:

$$x_{world} = z\left(\frac{x_{img} - c_x}{f_x}\right) \tag{2.5}$$

2.1.3 Blender

Blender is a free 3D modelling, animation and video editing program. It has Python as its scripting language with its own API, which I use to help prepare the RGB and depth feeds for each camera in some scene. It also features a compositor; a node-based editor which supports various operations on images, such as modifying channels, outputting depth or alpha values, and mathematical or image-based operations.

I chose to use Blender for preparing video scenes because of its suite of creative-commons example scenes and its scripting features which allow me to extract the RGB and depth data in the format I need, for several cameras.

The Blender Camera

The default Blender camera operates under a pinhole model, with no lens distortion (though configurable), with the following parameters:

- Focal length: 50mm
- Sensor size: 36mm - this is both the sensor width and the sensor height, since Blender uses square pixels by default
- Resolution: 1920x1080 pixels

These parameters are assumed for all scenes, but having them vary would not affect results since they are captured in the projection and unprojection formulas.

Rendering each scene into image sequences and video

This project requires the streaming of each frame of video (for both RGB and depth), for each camera, to be used as inputs for reconstruction.

For each scene, I configured Blender to output both image sequences and formatted videos for RGB and depth so that streaming performance and visual quality could be compared[2].

For images, I chose to render RGB frames as PNG files due to its lossless compression, which maintained image quality. For depth, I chose OpenEXR files with a single 32-bit float alpha channel since its high precision exactly matched the depth data Blender provided.

For videos, the RGB camera feed was saved as an MKV file with h264 encoding due to its high visual quality while using less bandwidth. For depth videos, there are limited options for encoding formats which support 16 or 32 bit colour channels, so I instead used 16-bit floats[3] for depth and saved each 8 bits across the red and green channels of each pixel. We still have to use a video which has 3 channels since the alternative is a single-channel grayscale video which does not offer a high enough bit depth.

To ensure that these values are never modified by the encoding process[4], I have the option between FFV1 and HuffYUV encoding, both of which are lossless and so preserve pixel values. FFV1 has much a higher compression ratio, but HuffYUV has better streaming performance at the tradeoff of a larger file size, as shown in Appendix C.

2.2 Isosurfaces

Isosurfaces are used in this project to represent the surface of signed-distance fields, which both methods in this project rely upon.

[2]The exact process I follow to obtain the data is explained in §3.2

[3]It is shown in the evaluation that the lower resolution for depth is still sufficient for reconstruction.

[4]This is because most video encoders are lossy, and changing some of the bits in the red channel could change the exponent of the float, which drastically changes the depth value.

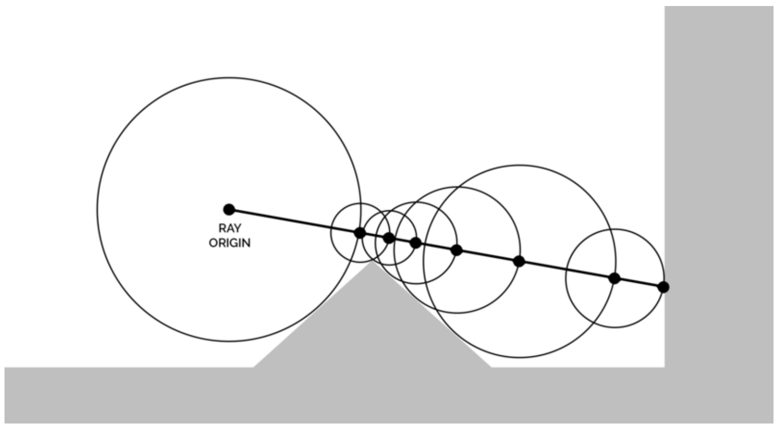


Figure 2.3: An example of the ray marching process. At each step, a circle is drawn which shows the closest distance to the surface. This distance is used as the next step size until an intersection is reached. Obtained from Teadrinker[13], CC BY-SA 4.0

An isosurface is a level set L_c of some continuous 3-space scalar field $f : \mathbb{R}^3 \to \mathbb{R}$ for some constant $c \in \mathbb{R}$, given by $L_c = \{\ \mathbf{x} \mid f(\mathbf{x}) = c\ \}$. The surface itself is defined as the domain of L_c, with the inside or outside of the surface being chosen arbitrarily for points whose range is greater or lesser than c. In other words, isosurfaces can be thought of as the generalisation of an implicit surface.

Isosurfaces have the advantages over other geometry representations in that they can define complex topological shapes mathematically, and can easily be transformed arbitrarily by applying operations directly to f. However, since isosurfaces are defined in a continuous space, we must discretize them in order to render them. Voxel grids are used to sample a subset of 3D space at regular intervals. By sampling f over a set of voxels V, a mesh can be generated directly from the voxel grid using a sampling algorithm such as the Marching Cubes algorithm[12].

2.3 Ray Marching

This project uses follows the reconstruction method presented by Lawrence et al., which includes ray marching in order to render the fused camera views.

Ray marching is a rendering technique which rays are cast through each pixel and then traversed iteratively along a scene until an intersection is found. At each point[5] along the ray, the signed distance to one or more surfaces are sampled. The signed distance has two purposes: to determine if the ray has intersected the surface, if it is close to zero; and to determine the size of the next step. These are shown in figure 2.3.

Ray marching is often used in scenarios where the intersection between the ray and scene geometry or data has no closed-form solution. This makes it applicable in situations where ray tracing will not work, such as rendering arbitrary 3D data. In particular, it can be used to find an intersection between a ray and several depth maps as this project requires.

2.4 Voxel-based volume integration

To provide a means for comparison, this project additionally implements the voxel-based reconstruction method presented by Curless and Levoy due to its accurate results.

[5]also known as a step, or march

The process can be seen in figure 2.4 and is comprised of three main steps:

1. Tessellate each depth map to produce a mesh containing vertex position, normal and texture values;
2. Use each mesh to update a voxel grid by finding the smallest signed distance from each voxel near the mesh to the mesh itself by ray casting;
3. Perform the Marching Cubes algorithm on the voxel grid in order to produce the output mesh.

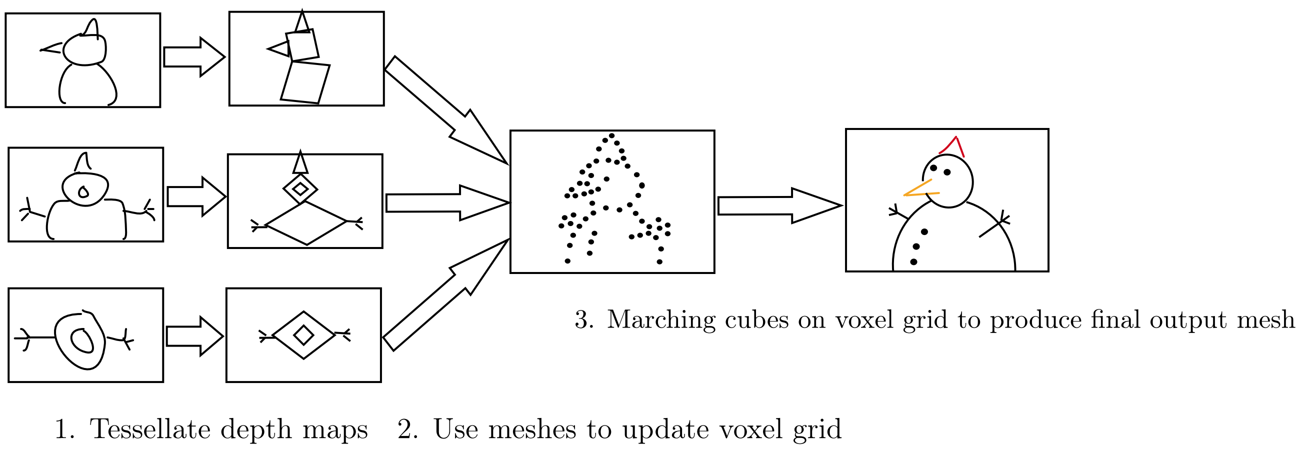

3. Marching cubes on voxel grid to produce final output mesh
1. Tessellate depth maps
2. Use meshes to update voxel grid

Figure 2.4: An overview of voxel-based volume integration. We begin with depth maps as input, tessellate them, update the voxel grid using the meshes, and then perform the marching cubes algorithm to produce an output mesh.

Each of these steps are elaborated further below.

2.4.1 Tessellation

Tessellation is the process of transforming a set of points into a triangular mesh by connecting edges between each point. An example of tessellation is shown in figure 2.5.

If we have the mesh data of all depth maps, in addition to camera pose information, then we could align the meshes directly and stop here. This is actually a valid method of geometry reconstruction as long as the overlaps between meshes are filtered to avoid *z-fighting*[14].

We choose to continue the processing instead of just fusing the meshes together for 2 reasons. Firstly, voxel grids help to reduce noise more explicitly by assigning weights to voxels whose physical space may have been measured less accurately, which can be the case with optical triangulation lasers on surfaces almost parallel to the laser. Secondly, for situations where many scans occur, updating a voxel grid with one mesh at a time is more memory efficient than holding dozens of meshes in memory, or even one very large mesh.

2.4.2 Voxel grids

A voxel[6] grid is a 3D grid of points that are uniformly spaced. They discretize a signed distance field (SDF), as each point in the grid can hold a value which samples the SDF at that point. Each mesh[7] is used to update the values of each voxel in the grid by finding the shortest distance from the voxel to the mesh.

[6] *volume-pixel*, defined as the smallest subdivision of a region of 3D space
[7] from each depth camera

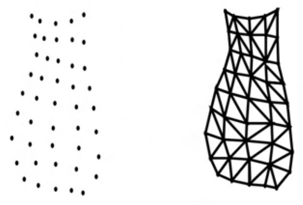


Figure 2.5: An example of how a point cloud can be tessellated to produce a 3D mesh. In our case, each pixel of the depth map will represent a point in the point cloud. Obtained from Curless and Levoy [6].

This distance is approximated by tracing a ray from the depth camera, through the voxel in question, and finding the point of intersection with the mesh. The sign of the distance is taken by comparing intersection points, as shown in figure 2.6. This process is repeated for all meshes.

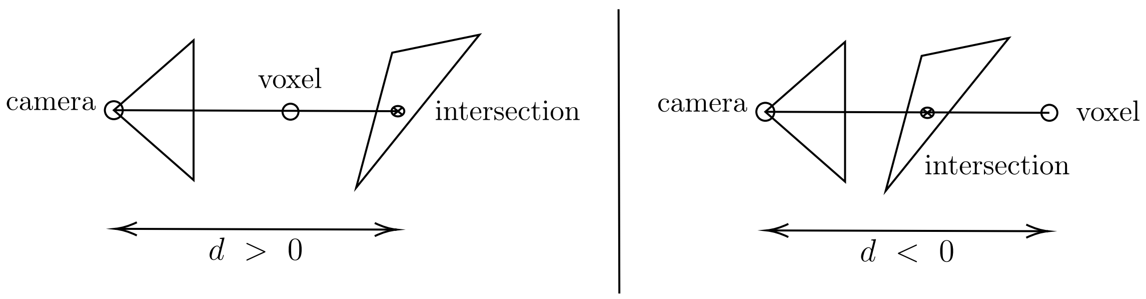


Figure 2.6: Comparison of a positive and negative signed distance, depending on the arrangement of the camera, voxel and intersection point.

When considering how this process operates with several meshes, the question arises of how a voxel should store the signed distance to several meshes at once. This is the crux of the reconstruction, since finding a way to do this allows the voxel grid to represent the signed distance field of the surface represented by *all* depth maps at once.

For a voxel at position $\mathbf{x}$, its signed distance to the fused surface $D(\mathbf{x})$ is updated according to the signed distance $d(\mathbf{x})$ at depth map i and weight at that point $w(\mathbf{x})$, explained below:

$$D(\mathbf{x}) = \frac{\Sigma w_i(\mathbf{x}) d_i(\mathbf{x})}{\Sigma w_i(\mathbf{x})} \tag{2.6}$$

$$W(\mathbf{x}) = \Sigma w_i(\mathbf{x}) \tag{2.7}$$

Weights are used to give less importance to regions of the surface whose depth may be measured inaccurately. For many capturing technologies, this is often when measuring surfaces that are close to parallel with the viewing axis. This introduces a projection error, so the weighting function w_i is set to the dot product between the viewing direction of the depth camera and the normal of the triangle found in depth map i.

In order to use equations 2.6 and 2.7 in the updating process, they must be expressed incrementally since we update the voxel grid with one depth map at a time:

$$D_{i+1}(\mathbf{x}) = \frac{W_i(\mathbf{x})D_i(\mathbf{x}) + w_{i+1}(\mathbf{x})d_{i+1}(\mathbf{x})}{W_i(\mathbf{x}) + w_{i+1}(\mathbf{x})} \tag{2.8}$$

$$W_{i+1}(\mathbf{x}) = W_i(\mathbf{x}) + w_{i+1}(\mathbf{x}) \tag{2.9}$$

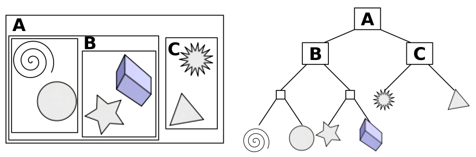
These are our final functions used to update the voxel grid. To avoid comparisons for voxel grids a large distance from the mesh, only the voxels near each mesh vertex position are considered.

To find the signed distance in the first place, ray tracing is used. This is also an expensive process when implemented naively since a comparison would have to be made to every triangle to check for an intersection. As such, acceleration structures must be used in order to reduce the number of intersection tests on the mesh.

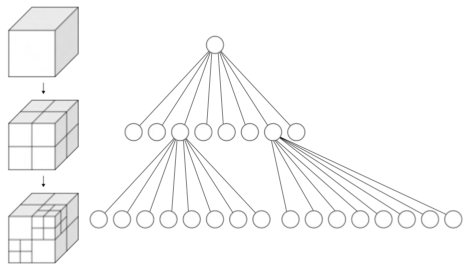
2.4.3 Ray tracing acceleration structures

A common way to reduce the number of comparisons is to split the mesh or space into large divisions that contain smaller divisions recursively, creating a tree-like structure.

We can choose to divide either the physical space or the object data itself. When split using *axis-aligned bounding boxes* (AABBs), dividing physical space results in an *octree* data structure, whereas dividing objects or primitives results in a *bounding volume hierarchy* (BVH). The difference between these data structures can be seen in figure 2.7. To find an intersection with ray tracing, the hierarchies are traversed until a leaf node is reached, where the actual object intersection tests occur.



(a) A bounding volume hierarchy. Space is divided using object positions. Obtained from Schreiberx[15], CC BY-SA 3.0



(b) An octree. Space itself is divided. Obtained from WhiteTimberWolf[16], CC BY-SA 3.0

Figure 2.7: Comparison between a BVH and octree. Note how they divide space differently.

Curless and Levoy use octrees, but I chose to use BVHs as they are widely used in ray-tracing applications today, and thus many efficient implementations of them have been devised since the publication of their paper[17]. I use a GPU-based BVH implementation described by Karras [18] which allows for both construction and traversal to happen entirely on the GPU, described further in Appendix B.

Now that the voxel grids can be updated in an efficient manner, a final output mesh must be generated from them.

2.4.4 The marching cubes algorithm

The marching cubes algorithm is used to generate a mesh from a voxel grid[8], which will be the mesh that represents our final reconstructed geometry.

The algorithm itself proceeds as follows and is illustrated in figure 2.8:

1. For each voxel, form a cube by treating the voxel as the bottom-left vertex and finding the neighbouring voxels which form the other vertices.
2. Record the sign of the values (signed distances) of each voxel in the cube to produce an 8-bit binary string[9].
3. Use the binary string to do a lookup in a table to obtain the mesh configurations for that cube of voxels, where the vertices of the mesh lie on the edges of the cube.
4. Use linear interpolation on each edge of the cube to find the specific position of each vertex in the mesh along the edge of the cube. Repeat for vertex normal values and UV texture coordinates.
5. Update the corresponding buffers for the overall mesh with each vertex attribute.

This process produces a mesh and its vertex attribute information that can be rendered using OpenGL, ready for viewing.

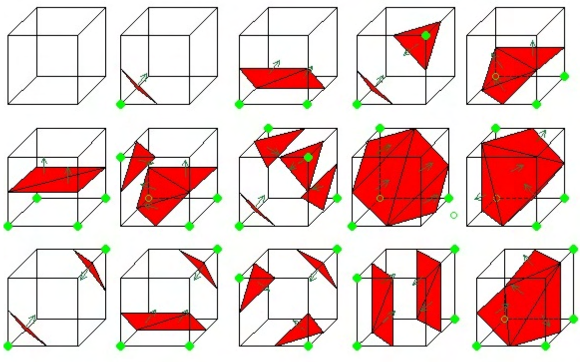


Figure 2.8: An illustration of all unique voxel cube mesh configurations. Each cube is a different case, with vertices being voxel positions. Green highlighted vertices are positively (or zero) valued voxels, and regular vertices are negatively. Obtained from Lorensen and Cline [12].

2.5 Starting Point

This project does not build on any existing code or project, but common graphics and file-handling libraries are used such as OpenTK (a C# wrapper for Khronos APIs such as OpenGL and OpenXR), TinyEXR and FFmpegAutoGen (an auto-generated C# binding for the FFmpeg C++ API, used for video handling). Besides OpenGL and Blender, I had no prior experience to any of the methods and libraries used in this dissertation.

2.6 Requirements Analysis

In order to deliver an application which can reconstruct a scene from RGB and depth, and display this to the user across a variety of scenes, there are several core goals which must be

[8]which represents a SDF

[9]Where a 0 is positive or zero (outside the surface) and 1 is negative (inside the surface), chosen arbitrarily.

Deliverable	**Priority**
Data Preparation	
Blender Python script for generating RGB and depth data	Required
Reconstruction Application	
Input processor for streaming image files	Required
Input processor for streaming video frames	Extension
Image based ray marching	Required
OpenGL application where the user can fly a virtual camera around a rendered scene	Required
Voxel based volume integration	Extension
Improve image based ray marching by making it robust to arbitrary camera poses	Extension
Improve voxel based volume integration with a GPU implementation	Extension
Evaluation for both methods	
Performance analysis using refresh rate of application	Required
Accuracy analysis through PSNR measurements	Required
Use case analysis	Required
Performance analysis through playback rate of streamed data	Extension

Table 2.1: Project deliverables

fulfilled. These give rise to the success criteria of the project, which are unchanged from the proposal in Appendix D:

1. At least 5 Blender scenes are rendered with RGB and depth output to produce a suite of streams to be used as tests and input;
2. An interactive application that takes RGB and depth video streams as input is produced which outputs a 3D scene created using a reconstruction strategy;
3. A modification or new strategy is implemented which is evaluated on the same scenes and same metrics as the original method;
4. The strategies are compared according to metrics that measure rendering speed (framerate) and the presence of artefacts, and are evaluated in the context of scenes of varying geometries and number of cameras.

As justified in sections 2.1.1 and 1.2, the primary technique I implement to fulfill the reconstruction is *image-based ray marching* [3]. In addition to comparisons with the original renderings of the scenes, a comparison to *voxel-based volume integration* [6] is preferable so that conclusions can be drawn on how different techniques fare under different circumstances[10]. Based on this, the project deliverables are listed in table 2.1.

[10]Curless and Levoy's method is chosen because it yields accurate results for both large and small scale scenes. The use cases for both techniques are discussed in the Conclusion.

2.7 Software Engineering Practices

Development of the project proceeded under an agile model. Naive versions of each method were produced under a small sample, such as a single depth map without colour. Each version was iterated on to address particular issues, such as performance, artefacts, and including support for several depth maps. Since the overall output and objective of the project remained for both techniques, a strategy pattern was utilised for the largest modules, allowing me to quickly test and change each reconstruction or rendering method, and reusing individual components. This also lent itself to effective comparisons between methods. Git was used for version control, allowing me to compare performance between my laptop and desktop computer to gauge robustness on different platforms. Since .NET aims to be platform-independent, I also aimed to minimise the use of platform-specific libraries where possible.

2.7.1 Tools Used

Blender was used to prepare the RGB and depth video scenes to be used as input. OpenTK was used for its C# OpenGL binding, in addition to its mathematics libraries. FFmpegAutoGen was used as a C binding for the FFmpeg API, and StbImage was used for image handling. For evaluation, OpenCV was used to calculate PSNR values.

2.7.2 Licensing

All Blender scenes were those obtained under Creative Commons (CC) Licenses. All C# libraries used are publicly available, and code within my project that is taken from open source repositories under CC licenses is clearly attributed with a link, credit and license type when they appear.

Implementation

This project implements **two** methods for reconstructing and rendering 3D scenes. In particular, section 3.4 describes the implementation of voxel-based volume integration, which is a geometry reconstruction technique that produces a mesh; and section 3.5 describes the implementation of image-based ray marching, which produces a direct rendering of the scene. Before this, the RGB and depth video streams must be produced using Blender.

3.1 Repository Overview

The repository is split up into C# projects, which can be thought of as high-level modules that each project can reference. The overview can be seen in figure 3.1.

Geometry handles geometric operations on mesh data and triangles, which is frequently used by all other modules.

Application is the entry point to the application, and initialises the reconstruction method to be used. It also contains the interactive application code which is separate from the reconstruction implementations themselves, in addition to the ray marching fragment shader.

Strategies contains the bulk of the reconstruction implementation for both the image-based method and the voxel-based method, including compute shaders.

Video Handler contains classes which communicate with the FFmpeg C binding, which requires its own memory management since C code is not garbage collected by the C# compiler.

Grouping each modules into these C# libraries made it easier for each reconstruction method to reuse commonly used modules, in addition to giving a clearer, hierarchical view of the project.

3.2 Obtaining RGB and depth video streams

RGB and depth videos are used as the input data for reconstruction, for which we use Blender. The camera positions are also needed so each depth map can be aligned to world space.

Once each scene is opened in Blender, I prepared several video feeds by placing cameras at different arbitrary points in the scene and recording their pose. A Python script is then run which records RGB and depth for each camera across all frames.

I used Blender's video editor to convert the sequence of RGB and depth PNG images into a .mkv files. For depth videos, each pixel has a 16-bit floating point depth value by storing each of the upper and lower 8 bits into the red and green channels respectively. These channels are unpacked into a floating point value manually when I use them in GLSL since it doesn't support integer to half float operations. HuffYUV, a lossless video encoding format, is used since it yields the fastest decoding speeds, as shown in Appendix C.

C# Project	Key Modules	Description
Geometry	`Mesh.cs`	Describes mesh data and provides operations to format the data in an OpenGL-useable way.
	`MeshLayout.cs`	Describes the buffer layout for data to be used by OpenGL.
Application	`Application.cs`	Initialises and chooses which reconstruction strategy to use.
	`MultiViewFramePreparer.cs`	Prepares texture data for each video camera for each frame.
	`BVHReconstruction.cs`	Initialises, runs and outputs the voxel-based strategy.
	`RaycastReconstruction.cs`	Initialises, runs and outputs the ray marching strategy.
Application/shaders/	`raycaster.frag`	Implementation of the ray marching algorithm.
Strategies	`VoxelGrid.cs`	Implementation of the voxel grid.
	`VoxelGridDeviceBVH.cs`	Implementation of the voxel grid for the GPU.
Strategies/shaders/		Compute shaders used for tessellation and the BVH.
Video Handler	`VidStreamHandler.cs`	Interfaces with the FFmpeg C bindings to produce video frame data.

Table 3.1: Repository overview

3.3 A user-interactive application for exploring reconstructed scenes

A simple application was developed using OpenTK which can either render a mesh[1] or ray march a scene[2] that the user can explore by controlling the virtual camera. The camera controls are derived from the OpenTK tutorial[19] so details are omitted from this dissertation.

3.4 Geometry Reconstruction: A voxel-based approach

The first geometry reconstruction technique I implemented was voxel-based volume integration. I implemented the three main steps discussed in §2.4. Additional modifications are discussed when they arise for each step.

[1]If the voxel-based method is used, which produces a mesh
[2]If the image-based ray marching method is used, which directly draws to each pixel

3.4.1 Tessellating Depth Maps

This section assumes knowledge of how index buffers work in OpenGL. A refresher can be found in Appendix A.

Each depth map needs to be tessellated to produce a mesh whose values can be used to updated the voxel grid. Within the depth map, neighbouring pixels are joined to form triangles as shown in figure 3.1.

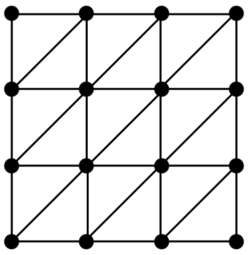


Figure 3.1: Tessellation example. Each vertex forms a triangle with its right and upper-right neighbours, and a triangle with its upper-right and top neighbours.

The entire depth map can be tessellated by only considering, for each point, the triangle formed from its right and upper-right neighbours, and the triangle formed from its upper and upper-right neighbours.

Before triangles are created, however, the x and y coordinates of each point are transformed from image space to world space using the unprojection formula described in §2.1.2.

At each triangle, if any edge lengths are over some threshold value, then the entire triangle is discarded. This is to avoid rendering material where there should be gaps in the depth map. The depth map is tessellated at a lower resolution than the original image by only sampling every n_x or n_y pixels[3]. Empirically, a threshold distance of $\frac{1}{80}(n_x+n_y)$ was found to be sufficient.

The index of each vertex can be determined by managing a dictionary which has the vertex information as an index, and the integer index as the value. A hash set is used to query whether a particular vertex has been seen yet. This ensures that duplicate vertices are not stored in their respective OpenGL buffers, thus providing a memory optimisation.

Tessellation on the GPU

Since it is guaranteed that every pixel can be used to produce at most two globally unique triangles, the tessellation process can be parallelised across pixels. An OpenGL Compute Shader is used to do this.

The process is the same as the CPU implementation, with the exception of index calculations because we don't have access to a dictionary or hash set implementation in GLSL. Instead, when allocating buffers to pass to the GPU, the worst case[4] is assumed, resulting in 6 possible indices per pixel. The position i_{idx} in the index array that we place each index idx is calculated as follows:

[3]I chose 4 for both since it has only a minor impact on quality whilst reducing the data size by a factor of 16.

[4]The worst case is when the maximum number of triangles are formed. That is, none of the triangles are discarded due to them having an edge that is too long.

$$i_{idx} = 6\left(\frac{y_{img}w}{n_x n_y} + \frac{x}{n_x}\right) + i_{v_{local}} + \text{offset}$$

Where x_{img}, y_{img} are the current pixel image coordinates, w is the width of the original image, n_x, n_y are the number of pixels we skip between samples, $i_{v_{local}}$ is the index of the current triangle in the range [0,2]$\subseteq \mathbb{Z}$, and offset is 0 if we are processing the first triangle or 1 if we are processing the second triangle for that point.

The idea is to allocate 6 possible indices per pixel in the array (for up to two triangles to be drawn), with the specific index being determined by the current vertex of the triangle, and which of the two triangles are being created, as shown in figure 3.2.

The value of the index itself is the flattened index (in column-major order) of the current pixel coordinate.

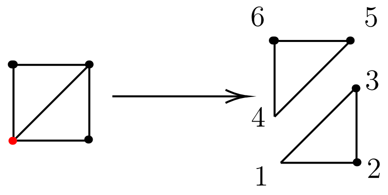


Figure 3.2: A demonstration of how triangle indices are determined on the GPU. The pixel in question (red) can generate up to two triangles. These are given the indices 1 through 6, which are added on to the flattened index of the pixel in the depth map.

This process repeats for each depth map, producing a set of meshes that can be used to update the voxel grid.

3.4.2 Voxel Grid Initialisation

The voxel grid is used to discretize an SDF which will represent the surface of the final fused geometry.

As such, a voxel grid implementation must hold the values of each voxel in the grid, in addition to supporting indexing operations to update the voxel at a particular point in space.

I defined the voxel values using a flattened array of floating-point values.

The space that the voxels occupy themselves are defined by giving the first voxel position[5], the number of voxels in each direction[6], and the spacing between each voxel. This enforces the voxels to have the same uniform spacing in any axis, which aids in indexing the grid.

To index into this array, a function $I : \mathbb{R}^3 \to \mathbb{N}$ is defined which takes any world position, finds the nearest voxel position, and its corresponding integer index into the voxel value array:

$$I(\mathbf{v}) = \mathbf{I}(\mathbf{v} - \mathbf{b}, t) \cdot \begin{bmatrix} t^{-1} \\ st^{-1} \\ s^2t^{-1} \end{bmatrix} \tag{3.1}$$

[5]the minimum x, y and z coordinates of the grid

[6]which I define to be the *resolution* of the voxel grid

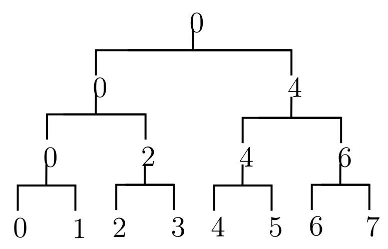


Figure 3.3: A bottom-up reduction. The numbers represent an index of the data set. In each pair, the result of the comparison is stored in the first index of the pair, and the index is used as the value of the parent node. This proceeds until the root is reached, where the root node contains the smallest or largest value of the data set.

where $\mathbf{I}$ is a function which rounds the values in a vector to the nearest scalar value, $\mathbf{b}$ is the smallest position in the voxel grid, $\mathbf{v}$ is the voxel position, t is the distance between two voxels along any axis and s is the number of voxels along any axis.

Due to limitations of OpenGL buffers, a voxel grid of size 500^3 is the maximum available. The size of the voxel grid is determined based on the AABB of all tessellated meshes. As such, the minimum and maximum coordinate values of each mesh need to be found. Since tessellation is completed on the GPU, no comparisons of the mesh data can be made to each other[7], so another efficient search is required.

3.4.3 Parallel bottom-up reductions

In order to find the coordinate ranges of each mesh, a *parallel bottom-up reduction* is implemented and performed on each mesh in a Compute Shader. Finding these ranges in parallel avoids having to compute ranges for every triangle of every mesh sequentially, which can take several seconds per mesh.

A bottom-up reduction works by first placing all objects to be compared at the bottom level of a tree, as leaf nodes. Each pair of leaf nodes is compared, with the chosen object becoming the value of the parent node. The same process proceeds at each level until the root is reached. The value at the root is the largest (or smallest) value in the group. The process can be visualised as a single-elimination tournament, as seen in figure 3.3.

To do this process in parallel, the data is split up into groups of size 2^n, with the last group having values of infinity to fill space if the data is not exactly of size 2^n. Each group is processed in parallel, with each group having their own reduction processes occuring.

2 passes of this are sufficient to reduce the data enough to do a small search on the CPU. My implementation uses a work group size of 256 to ensure that enough threads are being occupied whilst avoiding having to force threads to wait in the case that there are too many threads for the GPU to process at once.

The result is used to determine the start voxel position in the grid. This, along with the size of the grid, is used to define the voxel grid entirely.

[7]Since, otherwise, atomic min and max values need to be stored for each coordinate to avoid race conditions, and forcing atomic operations for every comparison will reduce the performance to a sequential implementation.

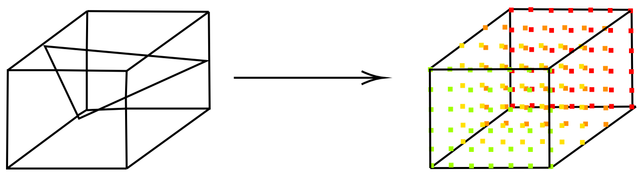


Figure 3.4: How voxels are chosen for updating. A bounding box is determined around the triangle (left) and all of the voxels within the box (right) are added to the list of voxels to be updated.

3.4.4 Determining which voxels need to be updated

The voxel grid must now be updated with each mesh. Since, due to the definition of an implicit surface, we are only interested in places where the sign of neighbouring voxel values changes[8], we only need to consider voxels close to the mesh[9].

To do this, we take each triangle of the mesh, and find its AABB rounded to the interval of the distance between neighbouring voxels. Every voxel position within the AABB is added to a list, as shown in figure 3.4.

3.4.5 Bounding Volume Hierarchies

Each voxel's value must now be updated with the weighted signed distance to all meshes. To aid the ray tracing involved in this, a BVH is created for each mesh, where individual triangles are the leaf nodes. To do this, I implemented an entirely GPU-based BVH construction algorithm presented by Karras[18].

Usually, BVHs are constructed in a bottom up fashion: each neighbouring leaf node is paired via a parent node, and parents nodes are recursively paired until a root is reached. This bottom-up approach is not effective on the GPU since a thread that joins two nodes must wait until its neighbouring thread[10] has joined their nodes. This produces the problem of *occupancy*: most of the work the GPU performs is at the bottom level, which is usually on more leaf nodes than there are threads. As the algorithm proceeds upwards, less threads are required, which wastes potential GPU computation time. This also introduces *data divergence* since higher levels will be accessing vastly different regions of the BVH array.

To make this construction fully parallel, every internal node is defined such that they represent a range of leaf nodes that they contain. Morton codes are used to represent the positions of triangles and are used by each internal node to determine their range.

3.4.6 BVH construction with Morton codes

Morton codes are used to represent the centroid of each triangle. They are used because they encode the position according to a *z-order curve*. When the Morton codes are sorted, they follow a space filling curve. By having the triangles be represented as a sorted array of Morton codes, the neighours of triangles can easily be found by looking up their adjacent entries in the array.

In order to find the Morton code of a triangle, we find the Morton code of its centroid. A Morton code is produced by mapping a point in the range $[0, 1]^3$ to an unsigned integer. We map world positions to this space using the ranges found in §3.4.3.

[8]indicating a 0 crossing of the surface
[9]since the mesh defines the surface for that depth map
[10]That is, the thread which processes the neighbouring nodes

The encoding proceeds by interleaving each bit of the x, y and z coordinates into a single bit string, as shown in figure 3.5.

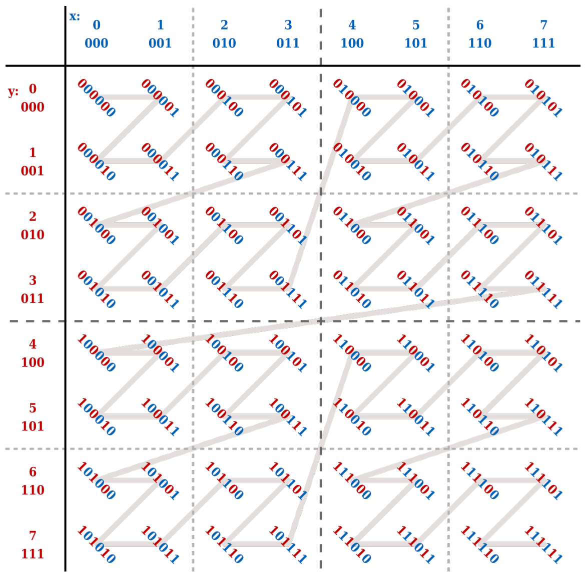


Figure 3.5: How Morton codes produce a Z-order curve in the 2D. Bits of each coordinate are interleaved to produce an ordering that fills the entire space. Public domain.

I implemented this encoding process using a Compute Shader which processed every triangle in parallel. The Morton codes are stored as 64-bit unsigned integers to increase precision, with the bottom 32 bits being set to the thread's global invocation ID to ensure uniqueness among all codes. I then used the ILGPU library to do a GPU-based parallel radix sort on the pairs formed by the index of the original triangle and the corresponding Morton code. This is to maintain the correspondence between triangle and Morton code for when we test for ray intersections with leaf nodes.

3.4.7 BVH Construction on the GPU

Now that we have a sorted list of Morton codes, we can use them to construct a BVH in a fully parallel manner on the GPU. This is done using the algorithm presented by Karras[18] which I explain in Appendix B.

Following this, we must then assign AABBs to every node. We can proceed again using a parallel bottom-up reduction (see §3.4.3). Each node's AABB is the union of its childrens', with leaf nodes obtaining theirs from their corresponding triangle. A Compute Shader is used to do this, with one thread launched per leaf node which traverse to the root. Since each layer of the BVH is processed at a time, the AABB of the children will always be available for the next node even if the current thread did not compute it.

We now have a BVH that is ready for updating the voxel grid via ray tracing.

3.4.8 Updating the Voxel Grid using the BVH

In order to obtain a SDF that represents the fused surface of all the depth maps, we need to update the voxel grid's values with the signed distance from each voxel to the meshes. This is done by tracing a ray through each voxel and intersecting it with the mesh, as described in section 2.4.2. This process is done on a Compute Shader, parallelised across rays in groups of 10,000 to avoid timing out the GPU[11]. Since GLSL does not support recursion or pointers, an explicit stack is used to keep track of which nodes are to be explored next in a depth-first fashion.

A ray can either intersect with an AABB or a triangle. In addition to updating the voxel grid, voxels through which a ray intersection occurs are stored in order to optimise the marching cubes algorithm, which happens once the voxel grid is fully updated with all of the meshes.

3.4.9 Producing an output using the Marching Cubes algorithm

The SDF stored using the voxel grid can now be interpreted to give a final reconstructed mesh. Only the voxels which were 'seen' by the voxel grid updating process are considered since these are the only voxels whose neighbours might have a sign change. Besides this, the algorithm is implemented exactly as given in §2.4.4.

When collecting mesh data, the same data structures used during §3.4.1 are used to keep track of unique vertex data and vertex indices.

The outputted mesh can be rendered as any other OpenGL mesh, concluding the process. A screenshot of a final reconstructed scene can be found in figure 3.6.

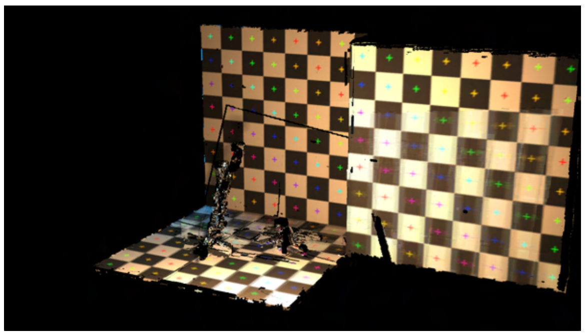


Figure 3.6: Screenshot of the output mesh from the voxel-grid method. Artefacts are discussed in §4.3.2.

3.5 Real-time direct rendering using an image-based approach

The second geometry fusion method I implemented was a direct rendering approach using ray marching, adapted from Lawrence et al. in their 3D telepresence system [3]. Whilst I follow most of the ray marching process itself, I had to make some fundamental modifications to the ray marching algorithm in order to meet the constraints of my project, as discussed in 3.5.1. Additionally, parts of their method are skipped due to differences in our capturing processes making them unnecessary. The description of my implementation below includes some of the modifications which made my version more effective for arbitrary geometry types and virtual camera poses.

3.5.1 Ray casting for geometry fusion

Ray marching is used to see if a ray cast through each pixel of the output screen intersects any of the depth maps. If they do, then the pixel is coloured in using the corresponding RGB map.

[11]Windows forcefully shuts down any GPU process which takes longer than two seconds.

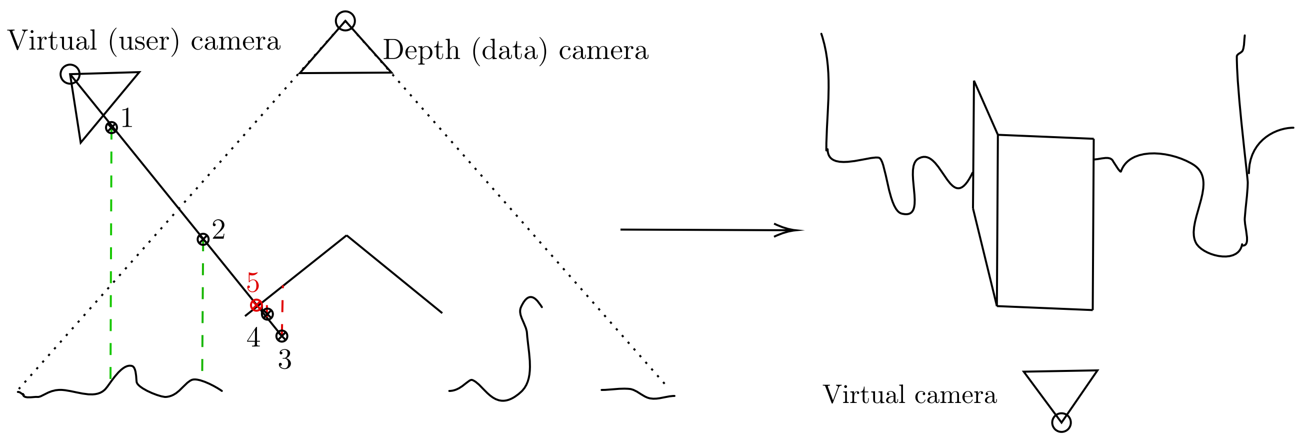


Figure 3.7: The ray marching process for one depth map. On the left, a plan view of the scene is shown, with a ray cast from the camera intersecting the face of the cuboid in 5 steps. Distances are looked up by querying the depth map from the data camera, which is why the direction of the depth lookups are in the same direction as the depth camera's viewing axis. The right shows the front view of what the virtual camera may see when the scene is rendered.

This way, the ray marching process will naturally render the depth maps fused together, thus reconstructing the original scene.

Since ray marching uses rays cast through each pixel of the screen, the entire ray marching process occurs in an OpenGL fragment shader.

In order to directly render the fused geometry to the screen, rays are marched along each depth map until the first intersection is found. This intersection point is used to calculate a colour for that pixel. The overall process for ray marching in this manner is shown in figure 3.7 which shows how ray marching happens for one depth map.

To avoid having to keep track of a different ray for each depth map, we work with just one ray for each pixel and transform the ray as needed. This requires two actions to happen: initially projecting the ray from screen space to world space; and projecting the ray from world space to the view space of each depth camera.

Ray setup and projection onto depth maps

The former step is consistent with any other ray marching process. We first define a viewing frustum of the virtual camera, which can easily be done using the viewing angle f and aspect ratio a of the output window. Then, we convert the fragment coordinates (given in screen space, e.g. 600x800px) to normalised device coordinates $\mathbf{ndc} \in [-1, 1]$. Using the NDC and virtual camera parameters, it is easy to obtain the ray direction r_{cd} in view space:

$$\mathbf{r_{cd}} = \begin{bmatrix} a\mathbf{ndc}_0 \tan(\frac{f}{2}) \\ \mathbf{ndc}_1 \tan(\frac{f}{2}) \\ -1 \end{bmatrix} \tag{3.2}$$

The origin of the ray is still the origin of the coordinate system since we are operating in camera space. I used a field-of-view which matches the FOV of the Blender camera during the Evaluation.

To convert the ray into world space, we need only transform the ray origin and direction (in homogeneous coordinates) using the inverse of the camera view matrix $\mathbf{V}$:

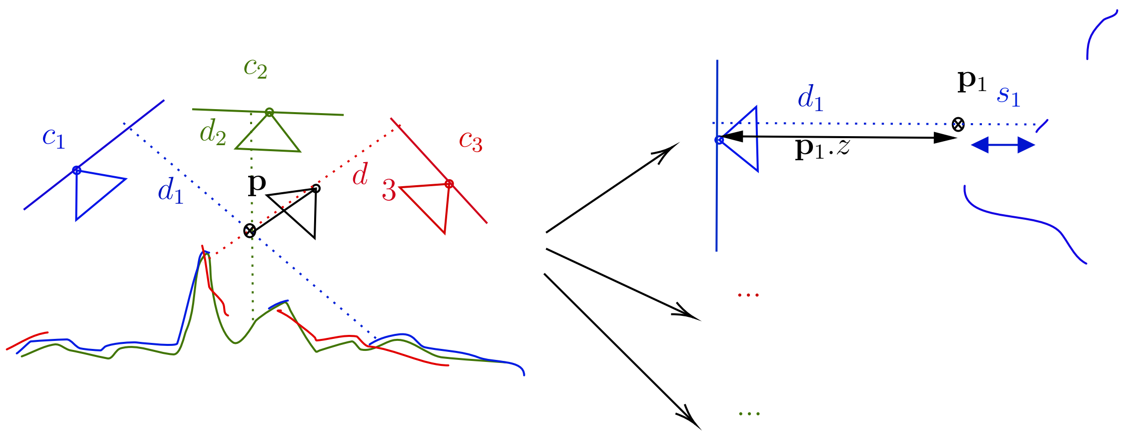


Figure 3.8: The process for obtaining the smallest signed distance s. At each ray march step, a depth lookup is performed for each depth map c_i by projecting $\mathbf{p}$ into their local image space. The depth, d_i, is subtracted from the z component of $\mathbf{p}_i$, which is $\mathbf{p}$ in the view space of c_i, to obtain signed distance s_i. The smallest s_i becomes s.

$$\mathbf{r_{wd}} = \mathbf{V}^{-1} \begin{bmatrix} \mathbf{r_{cd}} \\ 0 \end{bmatrix} \tag{3.3}$$

$$\mathbf{r_{wo}} = \begin{bmatrix} \mathbf{V^{-1}}_{20} \\ \mathbf{V^{-1}}_{21} \\ \mathbf{V^{-1}}_{22} \\ 1 \end{bmatrix} \tag{3.4}$$

With a ray origin and direction in world space, it is easy to march the ray along by calculating $\mathbf{r_w} = \mathbf{r_{ro}} + s\mathbf{r_{wd}}$ for any step size $s \in \mathbb{R}$.

In order to perform depth lookups given a point on the ray, we are required to project the ray into the view space of the depth camera we are interested in because depth values are given from the perspective of the depth camera. This is simply done by multiplying the homogeneous position vector of the point by the transformation matrix describing the depth camera's pose.

Marching the ray

In order to find an intersection point, we march the ray along some step size s until an intersection is found. In this approach, the step size is set to clamp$(s, -T, T)$ for some truncation distance T. In this project, I chose the truncation distance to be 0.02, which equates to 2cm in Blender's measurement system. This is in-line with the approach taken by Lawrence et al., and is shown to produce the best results in Appendix C.

s is chosen to be the smallest signed distance found across the depth maps, and is calculated by finding $\mathbf{p}_i.z - d_i$, where $\mathbf{p}_i$ is the projected point onto depth map i and d_i is the depth measured at this point on i, as illustrated by figure 3.8. This only approximates the signed distance to the surface of the depth map, as discussed in §3.5.1.

Adaptive truncation

Having the truncation distance be too small may result in a ray never reaching potential geometry after the iteration limit has been reached. However, a too large distance may result

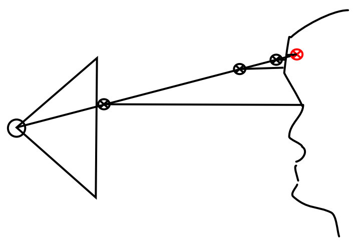


Figure 3.9: A demonstration of why Lawrence et al. were able to stop their ray march early. The structure of the human face approximates to a continuous surface, so any sign change indicates a crossing with the surface (shown in red).

in small details being skipped when an intersection should have occurred. As such, the ray should have larger truncation values in regions where it is likely that details will not be missed. I increase the truncation value in two circumstances.

The first is when a ray is outside all camera's view frustums, which can easily be checked by seeing if the projected image coordinates of the ray are in the range $(0, 0) - (1920, 1080)$[12].

The second is if the signed distance to all depth maps is larger than some value[13]. This can still risk missing details, but is less likely to occur since it requires all depth map lookups to be far away. I show that adaptive truncation can be used to reduce the iteration limit in 4.2.1, which improves the lower bound of performance. It is also a lightweight alternative to the point-splatting method that Lawrence et al. [3] use to determine the search bounds of each ray, which involves rasterizing all depth maps on every frame.

Knowing when an intersection is found

If an intersection is found, then the ray marching stops and the pixel is coloured in. The condition for finding an intersection is different from the original paper, which the Evaluation shows produces more accurate results.

Lawrence et al. march the ray along until the sign of s changes, which denotes a ray going behind a depth map. For their use case, this works because the virtual camera is in a similar position and direction to the depth cameras, and the human face and torso are continuous surfaces, as seen in figure 3.9. These assumptions do not hold in my case, in which the user can move the virtual camera to any position and can render any type of surface.

If the virtual camera is positioned such that a ray goes behind an object, but does not intersect it, then its sign will change since the direction to the depth map at these points is reversed. However, prematurely ending the algorithm at this point will cause the shadows cast by occluding objects to be treated as continuous objects that extend to the next surface. Figure 3.10 illustrates this case.

The problem persists if we naively allow the algorithm to continue marching, since the ray will oscillate around the discontinuity without ever getting a small enough signed distance. This causes holes to be cast behind all objects in the virtual camera's viewing direction, since the algorithm fails due to never finding a surface, as shown in figure 3.11.

Instead, I continue to march the ray in its original direction if a sign change is detected, and define an intersection to be found when the signed distance is smaller than some epsilon. I found setting this threshold distance to 0.001 to yield the best results in Appendix C. This

[12]for videos of resolution 1920x1080

[13]I choose 3 metres which is a safe upper bound considering the truncation distance is set to only go up to 0.1 metres.

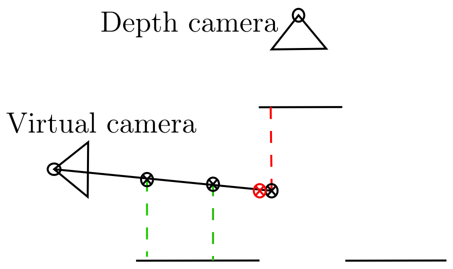


Figure 3.10: A demonstration of how a sign change doesn't imply an intersection. When a negative signed distance is recorded (red line), a surface is not crossed in this case because the depth map is discontinuous, even when the potential root is found (red cross).

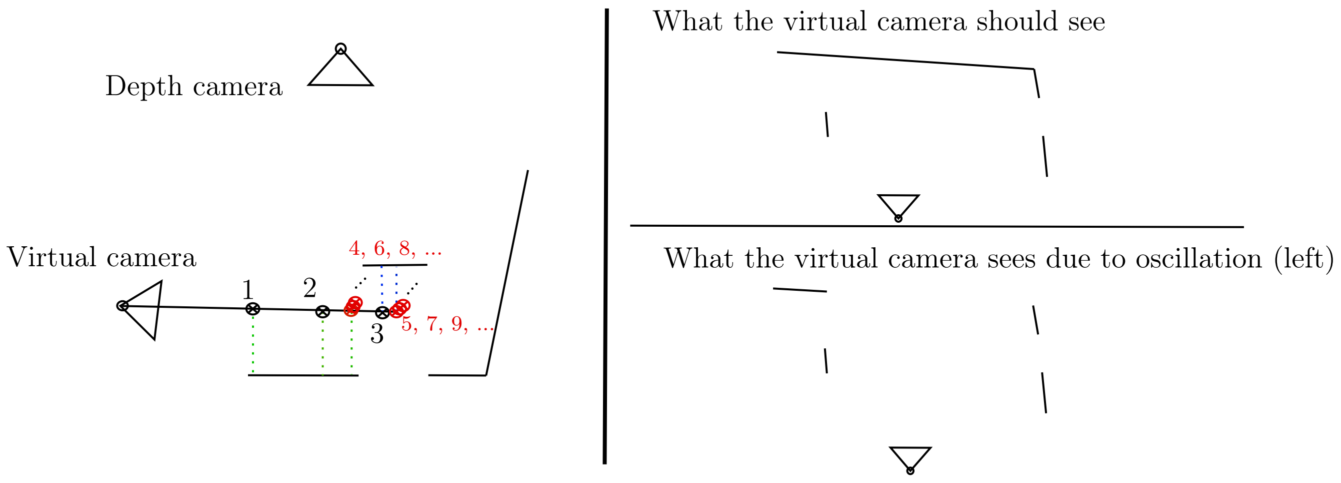

Depth camera
Virtual camera
1
2
3
4, 6, 8, ...
5, 7, 9, ...
What the virtual camera should see
What the virtual camera sees due to oscillation (left)

Figure 3.11: A demonstration of how continuing to march naively results in holes in the output. Whenever a camera's ray reaches the discontinuity, it oscillates around the surface without ever finding a root, as shown on the left. This results in the ray never reaching the wall at the end of the scene, so it will not be rendered to the screen, as shown on the right.

means that the ray will continue to pass behind objects, but stop when a surface (such as a wall) is found, as seen in figure 3.12.

This does not entirely solve the problem of finding correct intersections, however.

Truncated signed distance function of a depth map

The depth values used for depth maps are not the same as the signed distances used in traditional ray marching, which can cause visual artefacts to occur.

The process of projecting the ray onto each depth map and then performing lookups can be thought of as a way to approximate the SDF of the geometry described by all depth maps. In traditional ray marching, the signed distance to the nearest surface is given mathematically by the geometry. For example, for a sphere at position $\mathbf{p_s}$ and radius r, the signed distance from any point $\mathbf{p}$ to the closest point on the surface of the sphere is given by $d = ||\mathbf{p} - \mathbf{p_s}|| - r$. However, it is difficult to find the exact closest distance to the surface described by a depth map since the value at each coordinate gives the distance only along the camera's viewing axis, as shown in figure 3.13.

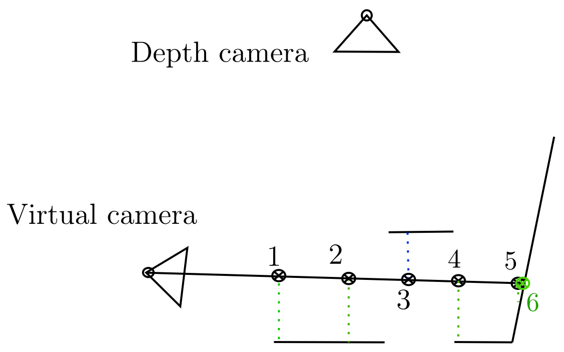


Figure 3.12: Continuing to march in the same direction avoids ray oscillation. When a negative signed distance is found, continue marching in the same direction instead of changing directions which would result in an oscillation.

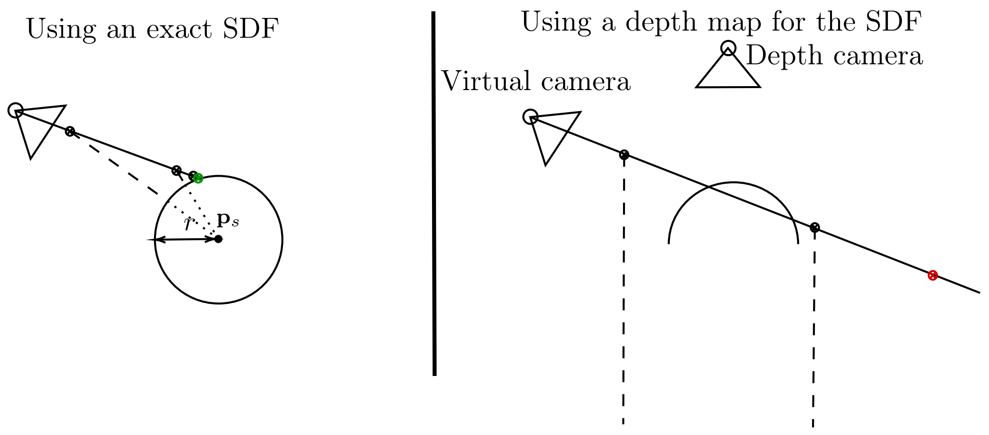


Figure 3.13: An exaggerated illustration of how depth maps cannot be accurate SDFs. Looking up depth could cause the ray to jump over an object that it was actually adjacent to.

One solution is to find the shortest distance by finding the minimum distance to every point on the depth map, which is clearly too expensive since, for a 1920x1080 resolution depth map, we would do 2,073,600 comparisons on every march step for every depth map. Instead, a suitable approximation can be found by just using the depth lookup at that point because any intersection is valid when the ray is on a depth map surface. Additionally, a conservative estimate for the step size can be found by truncating the signed distance and then marching 0.8 times that distance, of which both this project and Lawrence et al. found to be a good estimate empirically.

Reducing over shooting

The above approach works well when the virtual camera is looking in a similar direction to the depth camera because the depth values provide a more accurate step size, allowing the ray to converge more easily on the surface. However, it is easy to overshoot surfaces close to parallel to the depth camera's viewing direction since the depth value is a less reliable estimate of the distance to the surface along the ray direction.

This produces the visual artefact of there being rings carved out in surfaces that are are at a grazing angle to the depth camera's viewing direction as seen in figure 3.15. Since the gradient of these surfaces are high in this direction, the step sizes remain high, making it more likely for a ray to pass through the surface and miss that an intersection occured. The rendered surfaces between rings are regions in which the ray steps happen to land close enough to the surface to

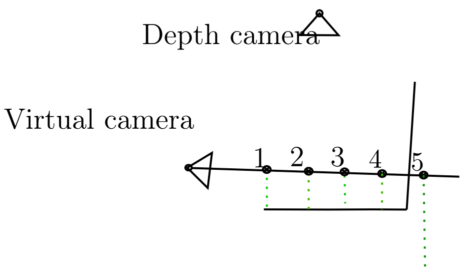


Figure 3.14: An exmaple of how ray marching overshoots almost perpendicular surfaces.

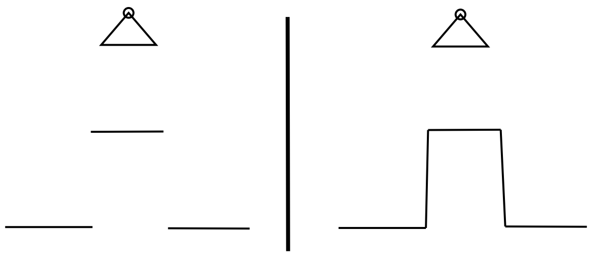


Figure 3.16: A discontinuity (left) and a steep slope (right). Due to the extreme angle, these appear almost identical to an RGB camera in the depth camera's position.

register.

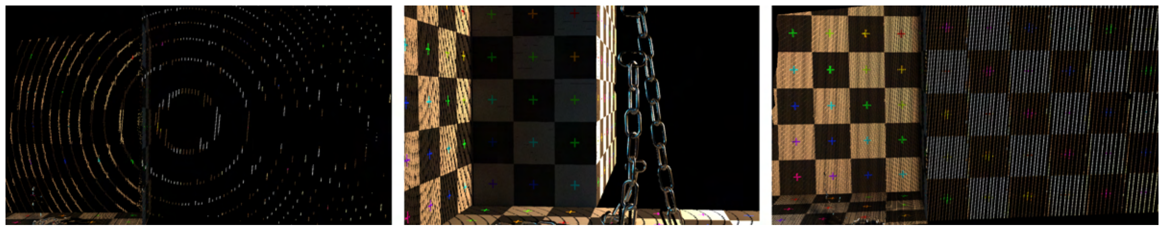


Figure 3.15: Visual artefact produced by not doing a root search. When the virtual camera is looking at a surface almost parallel to the data camera's view axis, there are gaps in the result (left). These go away when looking at the scene from the angle of the depth camera (middle). When root searching is used, more of the surface is visible (right).

It is still possible to tell when a ray has passed through a surface, however. For a small enough step size, the sign of the TSD is guaranteed to change. This is because, in order for a surface to be visible at all, it cannot be parallel to the viewing direction through those pixels. As such, observing a sign change with a very small step size may imply the crossing of a surface.

In practise, an infinitesimally small step size cannot be used since the iteration limit would quickly be reached without finding any surface (see §4.3). Conversely, using a large step size risks skipping the surface entirely. However, the step size currently used can have a change in sign for all but the most extreme surfaces in terms of angle.

Once a sign change is detected, more work must be done to avoid the issues described in §3.5.1. In particular, the algorithm needs to determine whether the ray has passed through a discontinuity or through a very steep slope as seen in figure 3.16.

To combat this, the precise intersection point is attempted to be found using a bisection search with the intial bounds being the previous step and current step positions.

The number of steps in the bisection search is equal to $\log_2(\frac{d}{t})$, where d is the maximum step size and t is the intersection threshold distance. If the search hasn't found an intersection at this point, then it's assumed that a discontinuity has been passed through and the ray continues to march forward. Otherwise, the point is returned and the colouring process begins.

Computing the colour of the pixel

The ray marching stops at two possible points - as soon as the ray meets an intersection on any depth map, or when the iteration limit is reached. If the iteration limit is reached then the background colour of the original Blender scene is used to aid an image-based comparison. Otherwise, when an intersection is found on depth map, the corresponding RGB image is used to obtain the value using the same pixel coordinates that were used to do the depth lookup.

In my implementation, no colour blending is used since there are no noise or projection errors. Additionally, there is a one-to-one correspondence between RGB and depth maps, so no further work is needed unlike in the case of Lawrence et al. The final colour is used for the current pixel of the fragment shader.

3.5.2 Efficient streaming of RGB and depth input data

A potential performance bottleneck of this method is how quickly RGB and depth data can be streamed in for processing. This can happen in two ways. If the application is not rendering because it is processing input data, then the application itself may freeze on each frame. If each frame of input data is streamed in slower than the original video's playback speed, then the reconstructed scene may play out in slow-motion.

I avoid the first problem by utilising multi-threading to handle input data. OpenGL is inherently single-threaded and so must run on the main thread. As such, I launch background threads for every camera feed. A shared concurrent priority queue is created for the frames for each video.

Each background thread asynchronously reads the frame data of each image or video frame and stores them in the camera's priority queue. The use of this data structure ensures that two threads that read subsequent frames will have their frames stored in order. For image data, TinyEXR and StbImage libraries are used to read in depth and RGB data respectively. For video data, an FFmpeg context is created locally on the thread which handles the decoding of the video data. If these threads read in data too quickly then too much memory can be allocated, resulting in freezes due to the C# garbage collector having to do more work. A circular buffer is used for video frames, since the memory used by each frame must be manually freed due to FFmpeg being a C++ library.

When the time between video frames has passed on the main thread, it reads the data from the priority queues and updates the OpenGL textures that store the RGB and depth data if the data is available. This allows the application to update the reconstructed scene in real-time at the original playback speed.

3.6 Summary

An application was successfully created which takes several RGB and depth video feeds, and renders the reconstructed geometry that the user can see from any angle. An image-based ray marching method and a voxel-based reconstruction method were implemented in order to do this. For the voxel-based method, several GPU implementations made the reconstruction process fast enough so that it is feasible for comparison against the image-based method.

My contributions to image-based ray marching further enhance the rendering accuracy and performance for a wide variety of scene types, as I will now show.

Evaluation

This project exceeded all success criteria and successfully implemented all core and extension modules. In §4.1, I show that the success criteria were met, thus demonstrating that a working, real-time geometry fusion application was created. In §4.2, I evaluate the performance of ray marching, with a demonstration of how my contributions to the method impact performance. In §4.3, I evaluate the accuracy of each method to ground-truth Blender scenes, and demonstrate how my contributions to the ray marching method yielded better results than the original.

4.1 Review of the project requirements

The project successfully implemented all core and extension requirements listed in §2.6, thus fulfilling the success criteria. The ways in which this project fulfills the success criteria are substantiated below:

1. **At least 5 Blender scenes are rendered with RGB and depth output to produce a suite of streams to be used as tests and input.** These were successfully rendered and can be found in figure 4.1. Videos of these scenes, along with my real-time reconstruction can be found in the project submission.
2. **An interactive application that takes RGB and depth video streams as input is produced which outputs a 3D scene created using a reconstruction strategy.** A screenshot of this application working can be found in figure??. Its 3D interactivity is shown in the videos attached to the project submission.
3. **A modification or new strategy is implemented which is evaluated on the same scenes and same metrics as the original method.** Voxel-based volume integration was implemented as a new strategy in §3.4, and a modification to the ray marching method was shown in §3.5.1.
4. **The strategies are compared according to metrics that measure rendering speed (framerate) and the prescnce of artefacts, and are evaluated in the context of scenes of varying geometries and number of cameras.** This evaluation is carried out in §4.2 and §4.3, demonstrating real-time results for ray marching, and an accurate reconstruction for both methods.

For extension modules, an evaluation of playback speed is shown in §4.2, which shows full speed playback at lower resolutions and real-time playback at a range of resolutions. Video streaming was necessary for this to happen, and was discussed in §3.5.2. All 5 reconstructed scenes are shown in figure 4.2, but it is encouraged to watch the video attached to this dissertation to understand what is being evaluated.

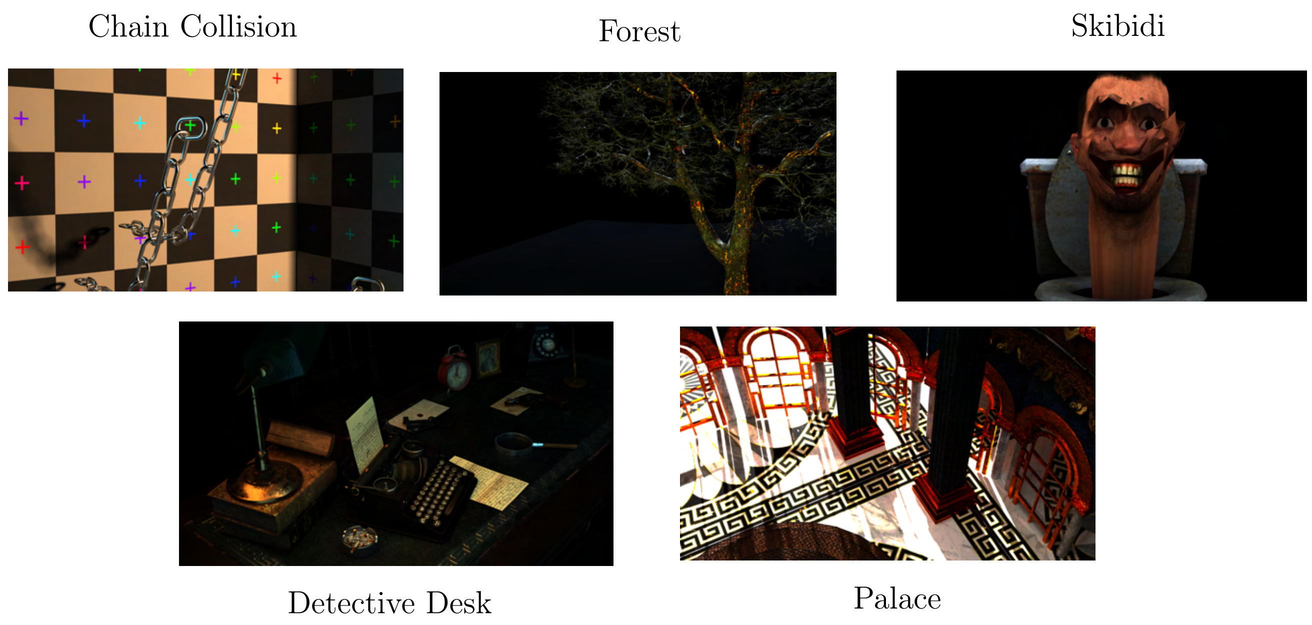


Figure 4.1: Example of all 5 Blender test scenes. In particular, scene "*Skibidi*" was used under CC-BY-4.0 with attribution to 'Noob'[20]. All others are CC0 or do not require attribution.

4.2 Performance benchmarks

To allow the user to explore reconstructed 3D scenes with a smooth experience, the screen must be updated often to present the illusion of motion.

In this regard, performance of this application can be evaluated from two perspectives:

Refresh rate is the term I give to how often a new image is rendered to the screen. At a real-time refresh rate, the user would still be able to fly around the scene smoothly even if the next frame of the video hasn't been processed yet.

Frame rate is the term I give to how often the next frame of RGB and depth videos are processed and used to update the scene itself.

Both of these performance metrics are measured in Hertz (Hz).

I define *real-time* to be 24Hz, which is the standard used for movies and motion pictures[21].

4.2.1 Performance of image-based ray marching

All of the processing for this method can be split into two parts: the loading of video data, and the rendering of the reconstructed scene using ray marching.

Each of these introduce their own bottlenecks. The following parameters were found to produce optimal[1] results, as justified in Appendix C:

- Video encoding: HuffYUV
- Output resolution: 1280x720px
- Ray marching truncation distance: 0.02m
- Ray marching intersection epsilon: 0.001m
- Video frame buffer size: 5 frames per video

[1]for both performance *and* accuracy

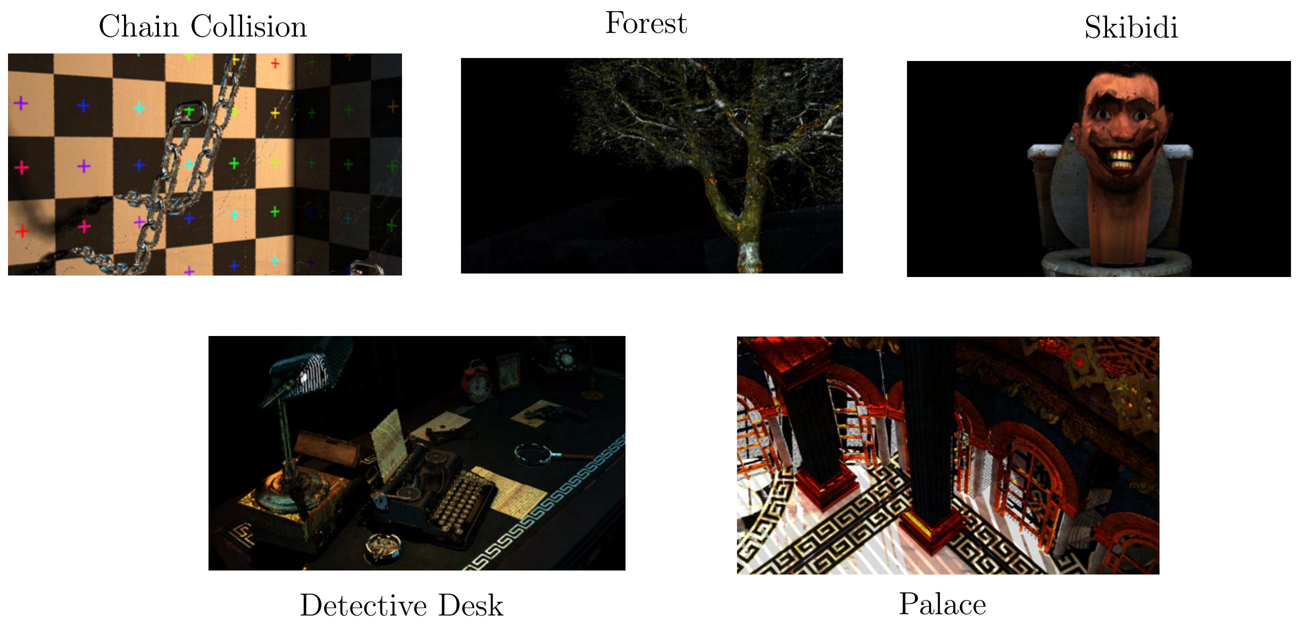


Figure 4.2: All 5 reconstructed scenes.

The project achieves real-time refresh rates across all 5 test scenes, as demonstrated in the figure 4.3 when using the above parameters. Refresh rates are recorded using a rolling average of the previous 60 render cycles. The refresh rates were calculated for every render cycle for all 5 scenes, with each scene being tested 3 times to capture anomalies[2]. The graph shows the average, minimum and maximum refresh rates recorded for each scene, with error bars representing the variance in these measurements across the three tests. All tests were performed using my personal desktop computer which has an RTX 4070Ti GPU, Ryzen 9 7950X CPU and 32GB RAM.

Despite the refresh rate being real-time on average, there is a large variance in possible refresh rates as indicated by the minimum and maximum refresh rates. Refresh rates are inherently volatile because the performance of the CPU and GPU, which process the video and render respectively, depends on other processes that are currently running in addition to hardware specific factors such as scheduling and temperature throttling. There is no explicit memory management available in C#, so garbage collection may also briefly lower performance momentarily.

For frame rates, it is similarly shown that real-time video processing was achieved in figure 4.3 for all 5 scenes. I show in Appendix C that full-speed video playback is available at lower resolutions, along with a discussion on why this is the case.

Performance of root searching

In order for my ray marching method to be more robust to arbitrary geometry types and camera positions, a modification to the original ray marching method [3] was implemented in §3.5.1.

This involves a bisection search carried out every time the ray's signed distance changes sign. The following graph demonstrates that, despite the decreased performance, real-time performance is still observed for both refresh rate and frame rates. As we will see in §4.3.1, this provides a noticeable increase in the rendered output's quality, making the tradeoff worth it.

[2]due to other applications using the computer's resources, for example

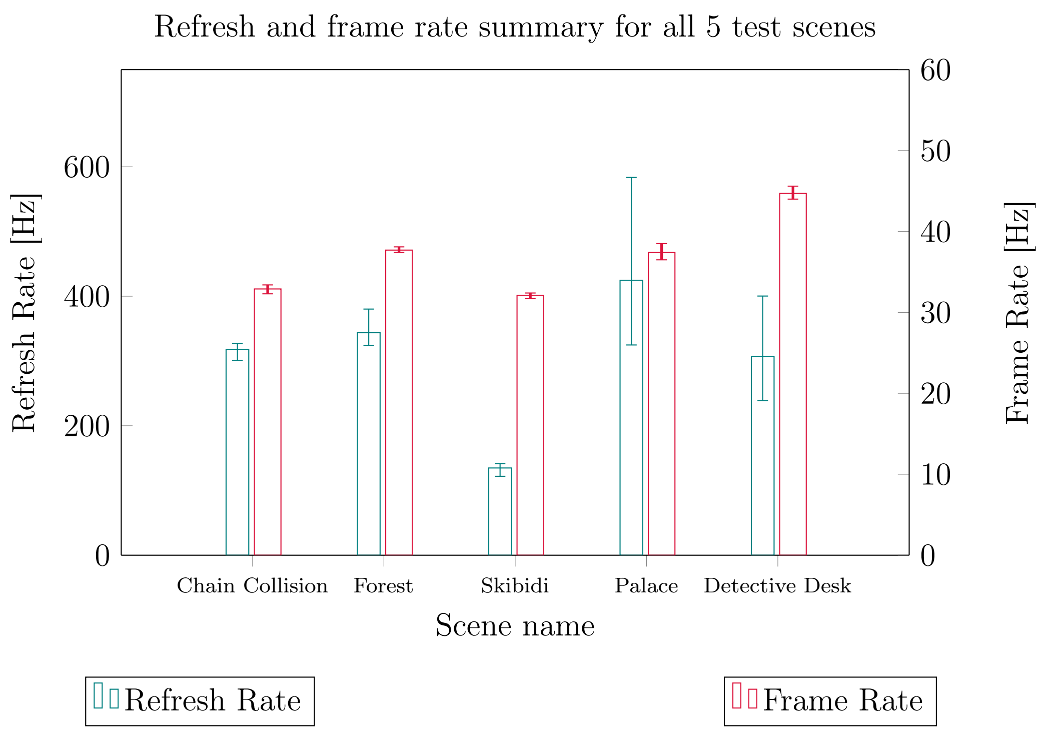


Figure 4.3: Performance summary. Both refresh rate and frame rates are better than real-time for all 5 scenes, which can be viewed in the video submission. "*Skibidi*" has a much lower refresh rate than the other scenes due to it mostly being empty space, resulting in the iteration limit being hit for most rays.

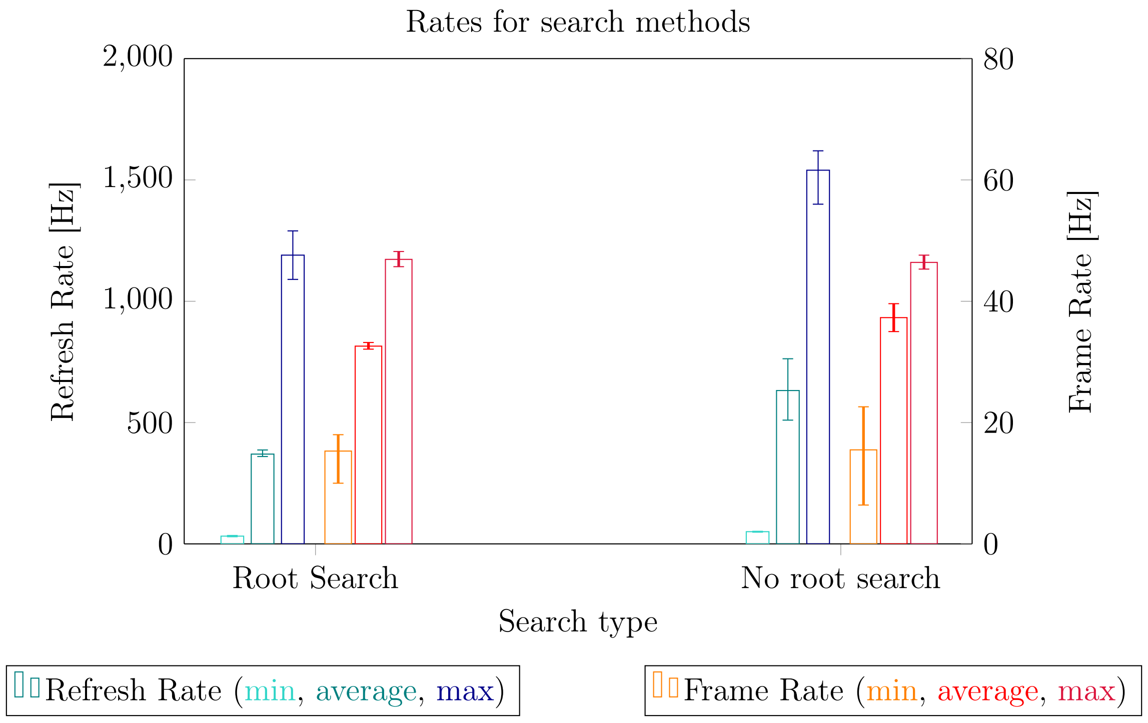


Figure 4.4: A comparison of the minimum, average and maximum refresh rates for root searching versus no root searching. There is a clear decrease in performance when bisection search is used, but it is shown later that this performance tradeoff is worth it for the improved accuracy. Unlike with refresh rates, the frame rates see a much smaller change when root searching is turned on. This is expected as the search only takes place within the fragment shader and so should not impact CPU performance for video decoding.

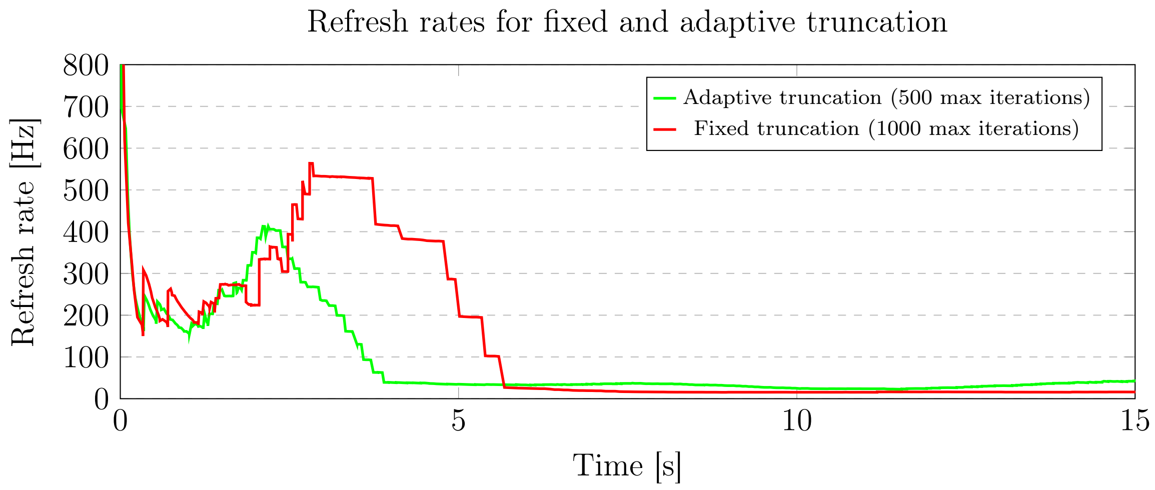


Figure 4.5: Refresh rates recorded for the scene "*Chain Collision*" for fixed and adaptive truncation schemes. Interestingly, the refresh rate is lower for adaptive truncation despite having less iterations. This is particularly the case early on in the scene when the camera is closer to the geometry. This may be due to there being more boolean checks for out-of-bounds rays, resulting in execution divergence in the GPU.

Performance of adaptive truncation

The iteration limit for the ray marching chosen so far was 1000 iterations, providing a maximum viewing distance of 20m (given a truncation distance of 0.02m). When looking at regions with no geometry, this decreases the performance drastically, as discussed in figure 4.3, so it is expected that adaptive truncation performs better in these situations.

Figure 4.5 shows the performance effects of this method. There is lower maximum performance when the virtual camera is close to the rendered geometry, which is likely due to execution divergence as each pixel is using different ray step sizes for many iterations. However, there is a noticeable increase in the minimum performance, as shown in figure 4.6.

This indicates that this scheme is suitable for lower-end consumer hardware. There is also a performance increase when the camera is looking in a region with little geometry. This is expected as it is in these regions where most of the rays will hit the iteration limit.

Overall, my adaptive truncation is a suitable scheme which benefits both refresh rate and frame rate in most cases.

4.3 Accuracy benchmarks

It is recommended that the videos included in the submission are viewed to gauge the reconstruction quality subjectively. However, we can quantify these accuracy measurements by taking a screenshot of a particular frame of the reconstructed scene, and then rendering the same frame from the same position in Blender. This gives a ground truth that I can compare my reconstructions to. I measured the quality of reconstruction using a *peak signal-to-noise ratio* (PSNR) on each frame. PSNR, measured in decibels (dB), is used commonly when measuring the reconstruction schemes of lossily-compressed images.

A higher PSNR value corresponds to a more accurate reconstruction, with infinity being exactly the same image. 30-50dB is standard for most lossy compressions, with 20dB being considered the "lowest acceptable" PSNR value[22]. Since our reconstructions can be viewed from any

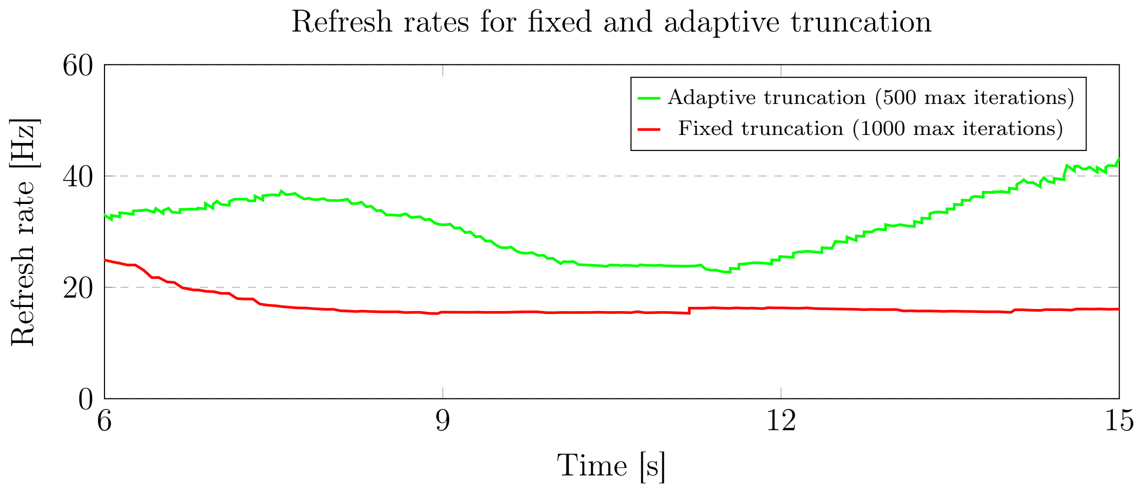


Figure 4.6: Upon closer examination, it is clear that in the worst case performance, adaptive truncation yields greater refresh rates, placing the performance above the real-time threshold of 24Hz that fixed truncation fell behind in this region.

angle, and are not just the original image, this view is held lightly. Rather, PSNR is used to quantify an accurate reconstruction.

4.3.1 Accuracy of image-based ray marching

To measure the accuracy of image-based ray marching, I animated the virtual camera to fly around the scene (as the scene is playing) along the same route as a camera in Blender. I also ensure that the camera parameters (see §2.1.3) are also identical between the two cameras. Since there are an arbitrary number of camera poses at every time step, it is infeasible to evaluate accuracy for all possible camera angles. Therefore, to reduce selection bias, I animated the camera such that it views close-up details and far away details of the scene. Since the reconstruction process does not account for that which is outside the data cameras' view frustums, I ensure that the virtual camera only moves within the convex hull of the data cameras.

Results

The average PSNR values for each scene test can be seen in figure 4.7. Overall, the reconstruction is not only recognisable, but it can be compared to a lossily compressed image, since the average PSNR value is around 20dB. This is an impressive value because it means we receive an accurate scene at any viewing angle.

The PSNR value drops in regions where the camera is close to the reconstructed geometry. This is because colour values may be taken from a data camera far away from that region of geometry, resulting in a lower-resolution output. This also results in some of the finer details on surfaces appearing blurred since relatively larger pixels from far away cameras are being coloured. This could be resolved by down-weighting further away data cameras during the pixel colouring process, at the cost of extra processing for each pixel.

There is a discrepancy in the way Blender calculates its depth maps compared to its RGB renders, resulting in pixels of an object being coloured the same as an object just in front of it, as seen in figure 4.8. This is due to Blender's Cycles renderer calculating the depth pass perfectly by querying the geometry itself, whereas it uses path tracing for RGB colour, resulting in noise. This difference is visualised in figure 4.8[3]. This is prevalent in the scene "*Forest*".

[3]They use path tracing in their Cycles engine, which is inherently random in some regions due to its Monte-

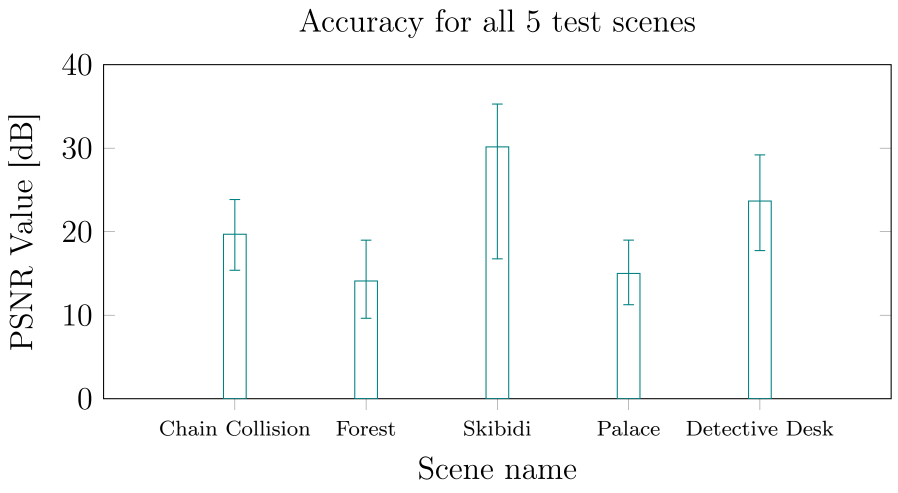


Figure 4.7: Accuracy summary. All scenes have a maximum PSNR of at least 20dB with 3 of them having 20dB as their average. Scenes "*Forest*" and "*Palace*" highlight future potential work of the project due to their small details and specular surfaces not being reconstructed accurately.

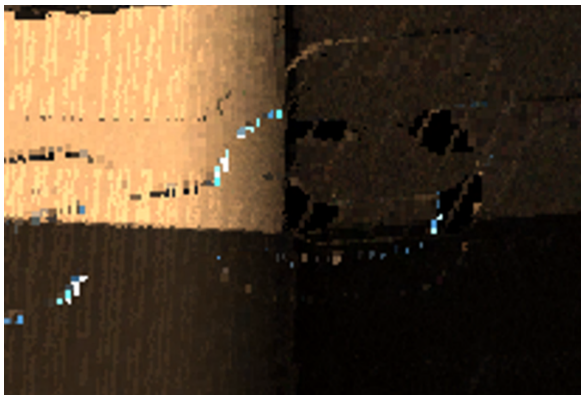


Figure 4.8: Here we see a visual artefact where the edges of the chain appear on the wall. It is shown that this is due to differences in the way Blender calculates depth versus colour.

Another source of inaccuracy is with specular surfaces, as seen in scene "*Palace*". They are view-dependent, so there is no way to account for them with ray marching since it relies on the specific viewing angle of the data cameras. As discussed in the proforma, there does not currently exist a way to estimate and reconstruct specular surfaces in real-time.

The final source of any inaccuracy is in regions where surfaces are at a scathing angle to the data cameras which capture it. This is explained in §3.5.1, and these artefacts are greatly reduced (see §4.3.1), although not completely eliminated due to infinitesimal step sizes not being practical.

Accuracy of root searching

Lawrence et al. use a bisection search on an intersection to improve the accuracy of their root for the pixel colouring process. I use it to determine whether an intersection actually happens[4] as the geometry I render is more unpredictable than their use cases (see §3.5.1).

Figure 4.10 clearly demonstrates the difference that root searching makes. This scene in par-

Carlo sampling process.

[4] I also do not necessarily end the ray marching algorithm at this point, which is another difference.

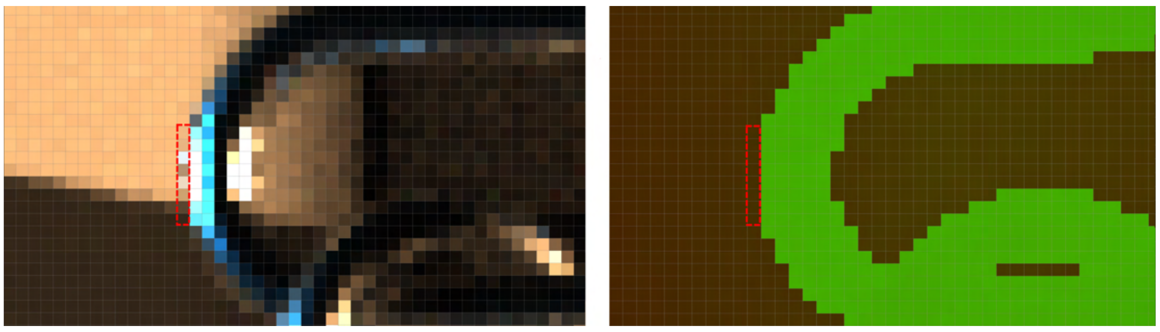


Figure 4.9: The inaccuracy with Blender's Cycles renderer is explicitly shown. The highlighted red pixels are coloured as a chain on the left image, but are actually part of the wall in the right-hand depth map (darker is further away in this case). This has a direct correspondance to the artefact in figure 4.8.

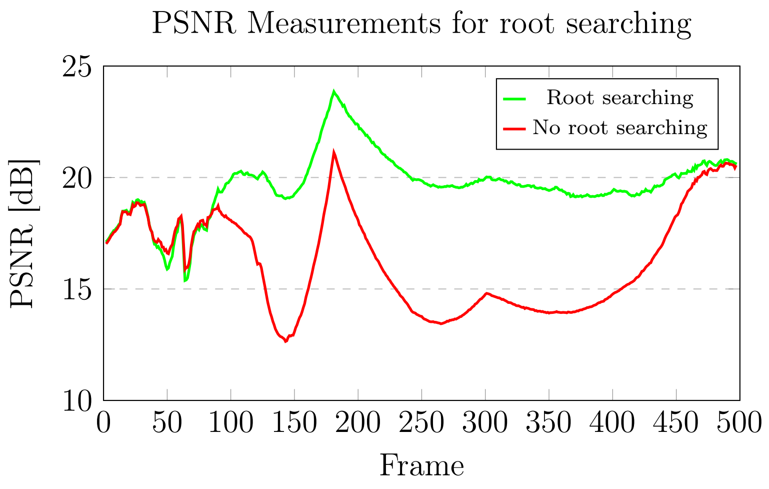


Figure 4.10: The graph shows the PSNR measurement for each frame in the test camera animation for the scene "*Chain Collision*". There is a clear increase in PSNR in regions where artefacts would otherwise be seen, showing an increase in accuracy across the board.

ticular was chosen for testing individual elements of the ray marcher due to its high variance in geometry types: fine details are seen in the chains; quickly-moving elements are seen in the swinging chains; still and flat areas are shown in the walls and floor; the data cameras are positioned both close to the geometry and far away; a range of colours with sharp edges are seen in the coloured crosses; and smoother colours are seen in the shadows cast by the chains.

4.3.2 Accuracy of voxel-based volume integration

The accuracy of the voxel-based method can be measured using the PSNR value for individual frames. Figure 4.11 demonstrates the reconstruction quality for a frame from "*Chain Collision*". There is no noise unlike the image base method, but the voxel-based method was not able to capture smaller details as seen in the blocky chains. This was because a hard limit was imposed on my voxel grid implementation of 500^3 due to the maximum buffer size that OpenGL allows. This could easily be alleviated if an alternative GPU API was used, such as CUDA. Additionally, there are regions of uncoloured or blurry voxels. The same weighting scheme for voxel values was used for colour blending, and colours between voxels were linearly interpolated during the Marching cubes algorithm, which evidently appears to be ineffective[5]. This may be because

[5]Curless and Levoy do not propose any scheme for colouring their reconstructed meshes, so these colouring decisions were devised by me.

colour weights should not have depended on voxel weights, since the value of the voxel already dictates which pixel colours to use from the RGB maps.

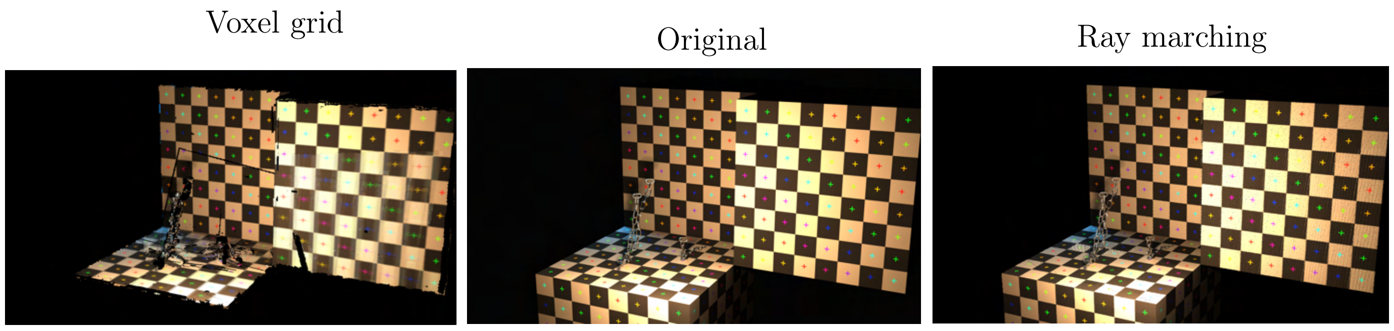


Figure 4.11: Comparison of reconstruction methods. The PSNR value for the voxel based method on this frame is 13.5dB, which is much lower than 23.7dB for ray marching.

This culminates in the image-based method being more accurate overall than the voxel grid method. However, if larger voxel grids were available, then the voxel grid method may be more accurate due to it being less prone to noise.

4.4 Summary

Overall, I have demonstrated not only that my ray marching implementation produces accurate, real-time results with adaptability for lower-end devices, but also that my contributions to the algorithm yield more consistent performance and greater rendering accuracy. The major artefacts are an unavoidable consequence of using Blender to prepare the RGB frames, and the lack of detail for the voxel-based method is due to unforeseen consequences of OpenGL's buffer size limitations.

Conclusion

The project was a success. An interactive, 3D rendered, real-time and accurate geometry reconstruction application was developed using ray-marching, and was shown to have positive results. My contributions to the algorithm were shown to improve the robustness of the ray marcher for both accuracy and performance.

A correct implementation of voxel-based volume integration was completed, and was shown to have promising results if not for the limitations of OpenGL. It was also optimised using a GPU-based BVH to provide a feasible (from over an hour to <1 minute) reconstruction time, with the potential to make it real-time if a different BVH implementation was used.

Screen recordings comparing the original video inputs to the final reconstructed outputs can be found with the project submission, substantiating my claims of a successful application being built.

5.1 Lessons Learnt

This was the largest software engineering project I have worked on, particularly the reconstruction techniques which have many smaller constituent parts to them. I learnt how to properly manage, break down and test each individual component, and I also learnt how re-implementations of techniques (such as the voxel-based technique) by other papers may involve heavy optimisations which are both omitted and are beyond what I was capable of in a short time[1]. Having to learn new programming paradigms, such as GPU programming, is a substantial time sink, especially when done for difficult GPU-based techniques such as BVH implementations.

Before beginning difficult implementations that may have a risky payoff[2], more due diligence should be needed to assess whether it is entirely suitable for my use case. Building smaller scaled prototypes of each component of the techniques proved useful in both debugging and understanding the implementations. Prioritising a minimum viable product over stubbornly sticking to a difficult technique would have alleviate a lot of the stress when completing the code for this project.

5.2 Future Work

Despite contributing to the ray marching process presented by Lawrence et al. [3], there are still several aspects of their implementation that I could implement to see their impact on my results. For example, rasterizing the depth maps with point splatting is a technique they used

[1]Lawrence et al. also implement the voxel based technique for comparison, but only described it as ‘efficient’ without referencing their implementation of it, which turned out to be substantially more advanced than the original paper by Curless and Levoy[6] in order to increase performance.

[2]as was the case with my BVH implementation, which, although quicker, did not make it real-time despite it taking several weeks to implement

to accurately determine the search bounds for each ray. I could compare this to my adaptive truncation, or perhaps use them together to further reduce the max number of ray iterations, which I showed to be performance bottleneck in my project. I could also implement their colour blending to see if it reduces the artefacts I found in §4.3.1.

The voxel-based method could be improved by using a different BVH implementation and GPU programming API, such as CUDA. NVIDIA's Optix provides ready-to-use BVH libraries that this method could benefit from.

The application itself also has practical use. It could be expanded upon with different reconstruction methods to produce a 'testing ground' for different reconstruction techniques in combination with different capturing techniques. This could allow the community to easily guage which combination is best suited for their usage scenario.

Bibliography

[1] Sushitha Susan Joseph and Aju Dennisan. “Three Dimensional Reconstruction Models for Medical Modalities: A Comprehensive Investigation and Analysis”. en. In: *Curr Med Imaging* 16.6 (2020), pp. 653–668.

[2] William Agnew, Christopher Xie, Aaron Walsman, et al. “Amodal 3D Reconstruction for Robotic Manipulation via Stability and Connectivity”. In: *Proceedings of the 2020 Conference on Robot Learning*. Ed. by Jens Kober, Fabio Ramos, and Claire Tomlin. Vol. 155. Proceedings of Machine Learning Research. PMLR, 16–18 Nov 2021, pp. 1498–1508. URL: https://proceedings.mlr.press/v155/agnew21a.html.

[3] Jason Lawrence, Danb Goldman, Supreeth Achar, et al. “Project Starline: A High-Fidelity Telepresence System”. In: *ACM Trans. Graph.* 40.6 (Dec. 2021). ISSN: 0730-0301. DOI: 10.1145/3478513.3480490. URL: https://doi.org/10.1145/3478513.3480490.

[4] Ryan S. Overbeck, Daniel Erickson, Daniel Evangelakos, et al. “A System for Acquiring, Processing, and Rendering Panoramic Light Field Stills for Virtual Reality”. In: *ACM Trans. Graph.* 37.6 (Dec. 2018). ISSN: 0730-0301. DOI: 10.1145/3272127.3275031. URL: https://doi.org/10.1145/3272127.3275031.

[5] Oleg Muratov, Yury Slynko, Vitaly Chernov, et al. “3DCapture: 3D Reconstruction for a Smartphone”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2016, pp. 893–900. DOI: 10.1109/CVPRW.2016.116.

[6] Brian Curless and Marc Levoy. “A Volumetric Method for Building Complex Models from Range Images”. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 303–312. ISBN: 0897917464. DOI: 10.1145/237170.237269. URL: https://doi.org/10.1145/237170.237269.

[7] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, et al. “Real-Time 3D Reconstruction at Scale Using Voxel Hashing”. In: *ACM Trans. Graph.* 32.6 (Nov. 2013). ISSN: 0730-0301. DOI: 10.1145/2508363.2508374. URL: https://doi.org/10.1145/2508363.2508374.

[8] Jiaming Sun, Yiming Xie, Linghao Chen, et al. “NeuralRecon: Real-Time Coherent 3D Reconstruction from Monocular Video”. In: *CVPR* (2021).

[9] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, et al. *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis*. 2020. arXiv: 2003.08934 [cs.CV].

[10] Cailian Lao, Yu Feng, and Han Yang. “Depth distortion correction for consumer-grade depth cameras in crop reconstruction”. In: *Information Processing in Agriculture* 10.4 (2023), pp. 523–534. ISSN: 2214-3173. DOI: https://doi.org/10.1016/j.inpa.

2022.05.005. URL: https://www.sciencedirect.com/science/article/pii/S2214317322000518.

[11] Chen Gao, Yifan Peng, Rui Wang, et al. "Foveated light-field display and real-time rendering for virtualreality". In: *Appl. Opt.* 60.28 (Oct. 2021), pp. 8634–8643. DOI: 10.1364/AO.432911. URL: https://opg.optica.org/ao/abstract.cfm?URI=ao-60-28-8634.

[12] B.R. De-Araújo, Daniel Lopes, Pauline Jepp, et al. "A Survey on Implicit Surface Polygonization". In: *ACM Computing Surveys* 47 (May 2015), pp. 1–39. DOI: 10.1145/2732197.

[13] Teadrinker. *Ray marching - Wikipedia.* URL: https://en.wikipedia.org/wiki/Ray_marching#/media/File:Visualization_of_SDF_ray_marching_algorithm.png.

[14] *LearnOpenGL - Depth Testing.* Accessed: 01/05/2024. URL: https://learnopengl.com/Advanced-OpenGL/Depth-testing.

[15] Schreiberx. *Bounding volume hierarchies.* URL: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy#/media/File:Example_of_bounding_volume_hierarchy.svg.

[16] WhiteTimerWolf. *Octree.* URL: https://en.wikipedia.org/wiki/Octree#/media/File:Octree2.svg.

[17] Daniel Meister, Shinji Ogaki, Carsten Benthin, et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum* 40 (May 2021), pp. 683–712. DOI: 10.1111/cgf.142662.

[18] Tero Karras. "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees". In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics.* EGGH-HPG'12. Paris, France: Eurographics Association, 2012, pp. 33–37. ISBN: 9783905674415.

[19] *OpenTK Tutorial.* Accessed: 3/11/2023. URL: https://opentk.net/learn/.

[20] *Skibidi Toilet Blender animation.* Obtained under CC-BY-4.0. Additional credit to Dafuq!?Boom!, the creator of Skibidi Toilet. Original work was modified to make the animation run at 60fps and to add more cameras. URL: https://sketchfab.com/3d-models/skibidi-toilet-animation-cc616eca6c964656828c0a02f65614d8.

[21] *A Beginner's guide to frame rates in movies.* Accessed: 2/05/2024. URL: https://www.adobe.com/uk/creativecloud/video/discover/frame-rate.html#:~:text=Motion%20pictures%2C%20TV%20broadcasts%2C%20streaming,focus%20due%20to%20quick%20movement..

[22] N. Thomos, N.V. Boulgouris, and M.G. Strintzis. "Optimized transmission of JPEG2000 streams over wireless channels". In: *IEEE Transactions on Image Processing* 15.1 (2006), pp. 54–67. DOI: 10.1109/TIP.2005.860338.

[23] *OpenGL Vertex Specification.* Accessed: 1/05/2024. URL: https://www.khronos.org/opengl/wiki/Vertex_Specification.

[24] *OpenGL Tutorial*. Accessed: 1/05/2024. URL: https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing/.

OpenGL Index Buffers

Within the OpenGL rendering pipeline, there are two stages that the user is required to write a shader for: the Vertex shader and the Fragment shader. This appendix focuses on the former.

The Vertex shader operates on the vertices of the objects that the user specifies to OpenGL; that is, it is called once for every vertex that the user supplies to OpenGL via a Vertex Array Object (VAO). The properties of each vertex can be specified using a Vertex Attribute, which is supplied directly to the shader. The user then manually specifies which attributes they would like their VAO to access. Properties can be customised and specified by the user (with limitations[23]), and commonly include items such as position, UV texture coordinates and normals.

The data itself that describes the values for each Vertex Attribute is supplied by Vertex Buffer Objects (VBOs). Each VBO is linked to a vertex attribute and is then attached to the VAO, so the Vertex shader knows which VBO to query for each vertex in the VAO. Specific details about the offset, stride and how many components each attribute has are omitted here for clarity but can be found in the OpenGL specification[23].

To specify the *order* in which vertices are processed, the user can supply an Index Buffer for their VBOs. This has 2 advantages; the first being that it gives the user more control over the processing order of vertices, and the second being that the shader can reuse vertex attribute values as shown in figure A.1. The drawing order of triangle needs to be specified to OpenGL, which naturally duplicates vertices that border several triangles. Without indexing, this would require us to repeat Vertex Attribute values for every drawn vertex, but with indexing, we need only repeat the index. OpenGL then knows it can reuse the other values in the Vertex Attribute since the Index Buffer specifies where they are.

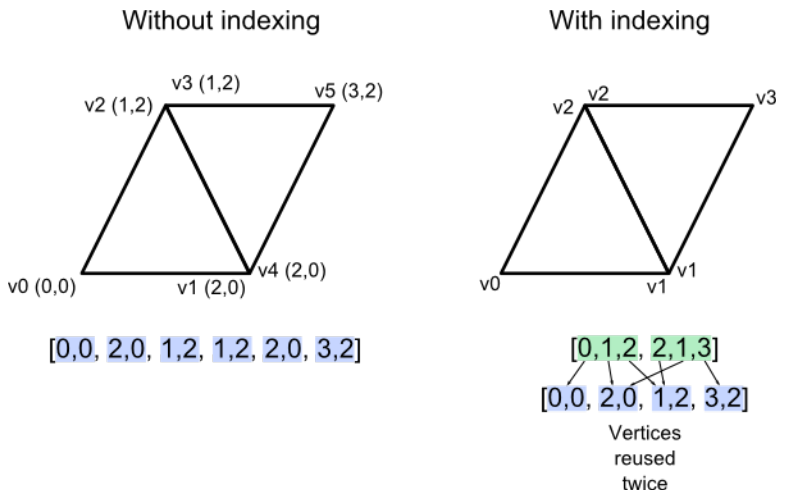


Figure A.1: Indexing example. Since every vertex needs to be specified for each triangle to be drawn, vertex information (such as position) needs to be repeated without indexing. With indexing, only the Index Buffer needs to specify the drawing order, so only its indices are repeated. Obtained from *OpenGL-Tutorial*[24].

BVH Construction on the GPU

This algorithm involves generating a binary radix tree structured bounding volume hierarchy (BVH) in an entirely parallel way. This method was presented by Karras[18], and proceeds as follows.

By determining each internal node's children, the entire BVH can be constructed. As such, the algorithm focuses on finding these for each internal node independently, which lends itself to a simple GPU implementation.

Each internal and leaf node *represents* something different about the tree, which can be seen in figure B.1.

Each *leaf node* represents a Morton code (triangle). We refer to Morton codes as *keys* since there is a one-to-one correspondence between Morton codes and node indices. That is, the index of a node can be used as an index into the sorted Morton code array.

Each *internal node* represents a *range* of keys. This range captures all of the leaf nodes of whom are descendants of the internal node. If an internal node is a *left* child of a parent node, then the key of that node is the *end* of its range[1]. If an internal node is a *right* child of a parent node (or is the root itself), then the key of that node is the *beginning* of its range. This is seen in figure B.1, where each internal node has a horizontal bar that represents its range.

Note that the index of an internal node is the same as the leaf node's index at side of the range that the internal node is positioned at. This can be seen by the vertical alignment of the internal node and its leaf node of the same index in the figure. We can use this property to determine specifically what the range of an internal node should be, and therefore its children.

To determine the specific range of keys that an internal node covers, we examine the common number of leading bits of keys within its range. For indices i and j, define $\delta(i, j)$ to be the length of the common prefix of the keys indexed by i and j.

Firstly, we must determine whether an internal node is the beginning (right child) or the end (left child) of a range. We do this comparing the number of leading bits between the internal node's key to its left and right neighbours' keys $\delta(i, i+1)$, $\delta(i, i-1)$. Whichever neighbour has more leading bits is the neighbour that is in the internal node's range because the corresponding triangles will be closer to each other in 3D space[2]. As such, the range proceeds in the direction $d \in 1, -1$ of the chosen neighbour. An example of this can be seen in figure B.1 with internal node 2; key 1 shares 2 leading bits, whereas key three shares 4, so its range is in the direction of 3 (and hence 2 is a 'beginning' range node, i.e. a right child).

We already have one end of the range of keys by construction of the internal nodes. Now we must find the other end of the range. We know that $\delta(i, i-d)$[3] can be used as a lower bound for the common prefix length, since we would like the internal node to contain objects that are more closely grouped than objects outside its range, and therefore should have a higher δ. Thus, we look for the highest key $i + ld$ such that $\delta(i, i+ld) > \delta(i, i-d)$. This is done by

[1]i.e., the final Morton code

[2]This is the goal of a BVH: to group parts of an object that are close to each other.

[3]$i - d$ is the index of the rejected neighbour

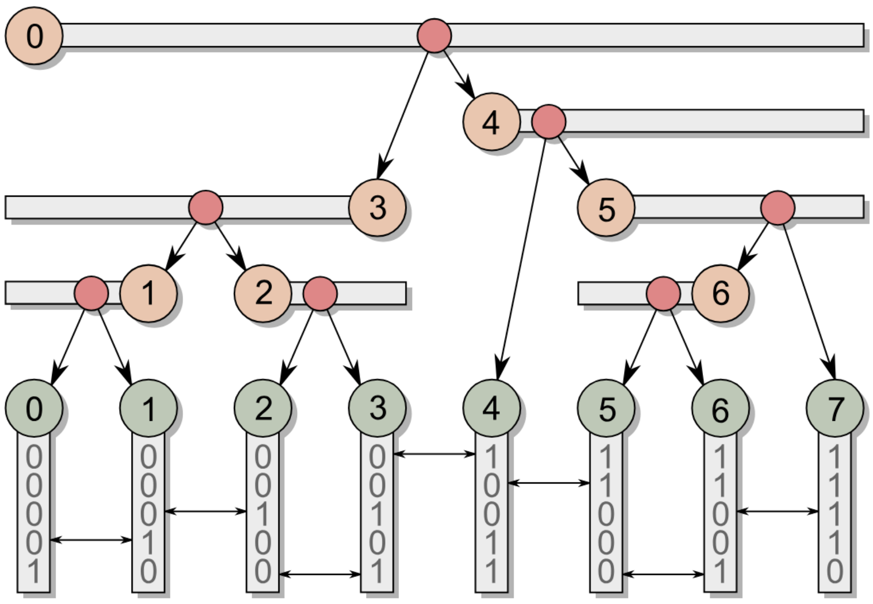


Figure B.1: An example of a binary radix tree of a BVH. Leaf nodes contain Morton codes, which encode triangle positions. Internal nodes represent a range of leaves, indicated by the horizontal bar. Each internal node index corresponds with a leaf node index, as shown by the vertical alignment. Internal nodes are either the start or end of their range as indicated by their position along the bar. Red circles represent split positions, whose closest indices are the children of that node. Obtained from Karras [18]

exponentially increasing l until $\delta(i, i + ld) \leq \delta(i, i - d)$. This gives us an upper bound of the key. Then, the specific end of the range $i + td$ can be found by doing another binary search on the current range, until $\delta(i, i + td) > \delta(i, i - d)$. We now have the range that the internal node covers, $[i, i + td]$, which can be used to determine the indices of its children.

This can be done by finding the *split-position* of the keys within the range. We want the split position to be between the highest-differing bit of the keys. This ensures that all keys either side of the split have a greater δ than the previous δ for the entire range. Again, this can be found using a binary search, giving us a split index γ. The split index is the index of one of the children; the other child can be found by finding $\gamma + d$. For example, with internal node 0 in figure B.1, its split position is between 3 and 4 since those keys are where the most significant bit differs.

If a child's range - the difference between the index of the child and either the start or end range - is one, then the child must be a leaf node since it cannot have any more children.

We now have a BVH structure that can be fully constructed on the GPU. In order to make use of this, all that remains is associating this tree with the space it occupies by assigning AABBs to each node.

Finding the optimal ray marching parameters

It was discussed in §4.2.1 that each parameter was carefully chosen, which I now demonstrate by comparing parameter choices under both performance (frame rate and refresh rate) and accuracy (PSNR). These tests were all done on the scene *Chain Collision* for the same reasons given in §4.3.1.

Resolution

Real-time refresh rates are observed for all resolutions besides 1440p, and real-time frame rates are observed for all resolutions besides 1080p and 1440p. The performance of these are still fast enough to be perceived as motion (>10Hz). A comparison of refresh rates and frame rates can be seen as the scene *"Chain Collision"* plays, as shown in figures C.1 and C.2.

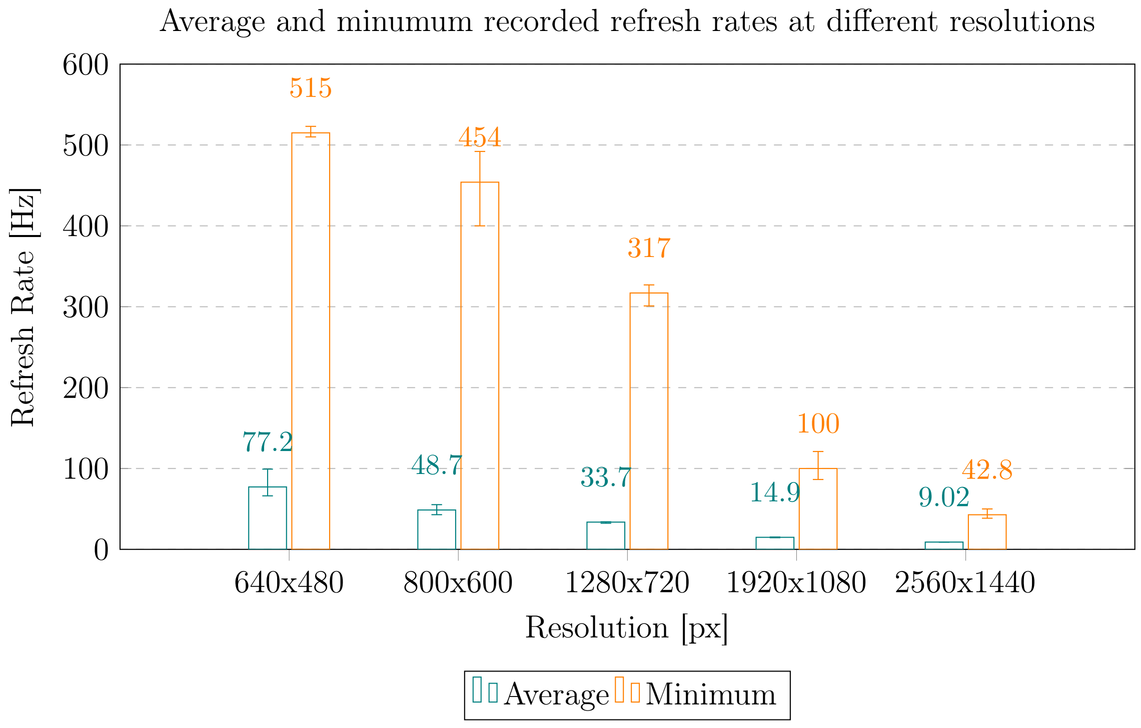


Figure C.1: Refresh rates at different resolutions for *"Chain Collision"*. It is clear that greater average and minimum performance is seen at lower resolutions due to less rays being marched in the fragment shader.

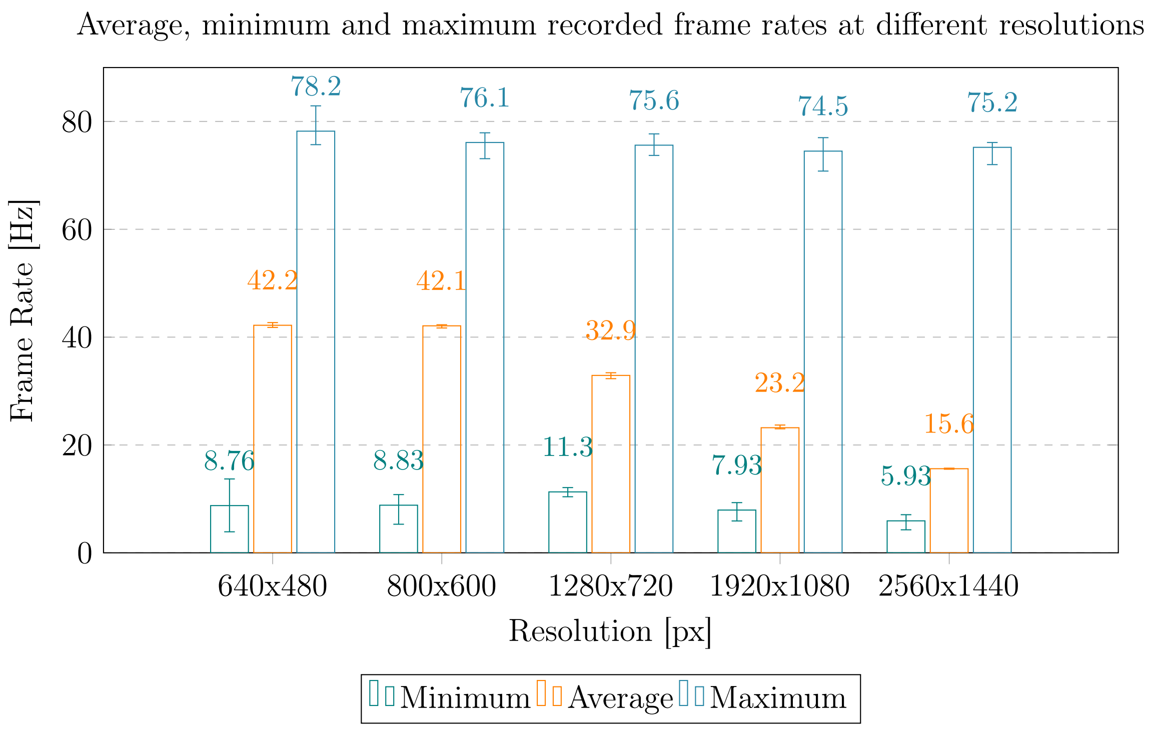


Figure C.2: Frame rates at different resolutions for scene *"Chain Collision"*. There is a similar correlation to the refresh rates, except it is clear that resolution no longer becomes a bottleneck below 1280x720 resolution as the frame rates are similar at these lowest resolutions.

There is a clear negative correlation between both refresh rate and frame rate, and the resolution. This is due to the ray marching algorithm being performed on more pixels at once, which bottlenecks GPU performance, as expected from any ray marching implementation. At higher resolutions, frame rate also drops, which is interesting because the video decoding happens CPU-side on background threads. It may be the case that FFmpeg still utilises the GPU under the hood since GPUs are optimised for video decoding. Ultimately, 1280x720 is the highest resolution that still supports real-time frame rate, so this resolution is chosen. It should be noted that lower resolutions yield greater performance, making the application effective on lower-end hardware still.

Truncation distance

The truncation distance[1] has different consequences depending on whether it is too big or too small, so it must be chosen carefully. A truncation distance that is too small results in the ray not traversing enough distance before hitting any geometry. A truncation distance that is too large may skip smaller parts of the geometry altogether. Hence, it is shown that a truncation distance of 0.02 sufficiently covers all geometry whilst remaining small enough to capture fine details in figure C.3. There is a small difference in most truncation values, but since larger values can miss details in the chains (which are more likely to have blocky artefacts since they are finer details captured from far away cameras), they have a slightly higher overall accuracy. However, I chose not to use these since it is important that all details are captured even if they are inaccurate, rather than omitting elements of the scene.

[1]The maximum/minimum step sizes for each ray

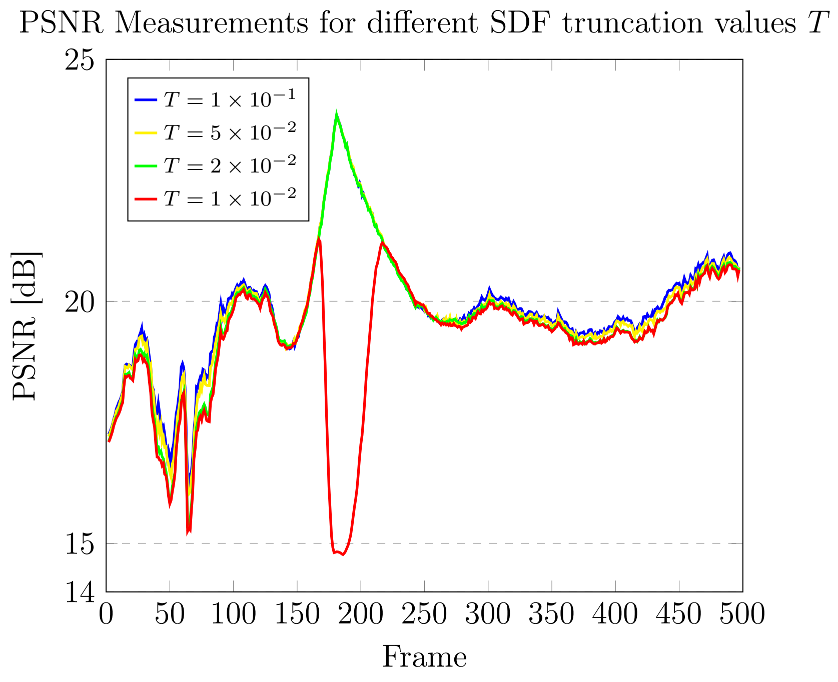


Figure C.3: PSNR measurements recorded as the *"Chain Collision"* scene was played from the test camera's perspective. There is a marginal difference in most truncation values, with smaller truncation values having a slightly lower overall accuracy, since larger values are more likely to miss regions of chain that are more likely to have artefacts. The lowest truncation distance has a much lower PSNR in the middle because the test camera is too far away to render geometry.

Intersection threshold value

An intersection threshold value also presents problems when it is too large or too small. When too small, an intersection may not be reliably found with some of the geometry, resulting in holes in the geometry. Additionally, a small threshold results in a larger number of bisection search iterations, reducing performance. When too large, false positives arrive when checking for intersections, resulting in distorted geometry.

It is shown in figure C.4 that a threshold value of 0.001 produces the best outcome when searching for precise geometry intersections. At frame intervals [0, 75] and [450, 500], we see that $t = 10^4$ and 10^{-5} outperform larger t values. This is because at these parts of the animation, the test camera is closer to the objects in the scene, making distortion more apparent for larger t. However, when further away from the geometry in the middle of the animation, the holes become much more apparent, resulting in larger t values outperforming lower t values.

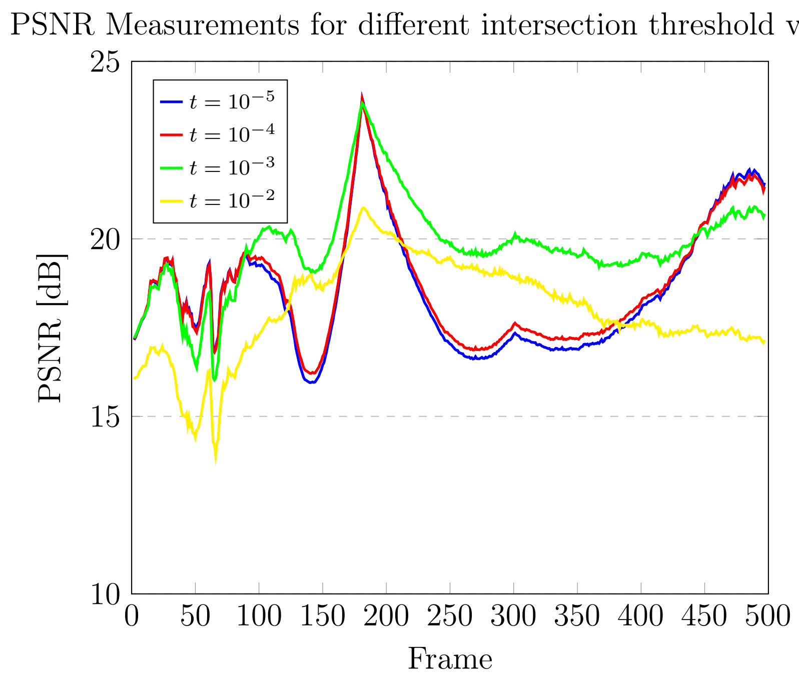


Figure C.4: PSNR measurements for different intersection thresholds for the scene *"Chain Collision"*. The fact that different t values perform better during different parts of the animation highlight the tradeoff mentioned above.

Video encoding format

As discussed in §2.1.3, the choice of video format was limited due to the requirement of lossless frame compression. FFV1 offers better storage compression at the tradeoff of slower frame decoding speeds, whilst HuffYUV offers little storage compression at the tradeoff of fast decoding speeds. Figure C.5 demonstrates how, overall, HuffYUV provides better framerates, although it should be noted that FFV1 has more consistent frame rates. This, and its lower storage footprint may make it more suitable for lower-end devices, but I chose HuffYUV for its greater average frame rate.

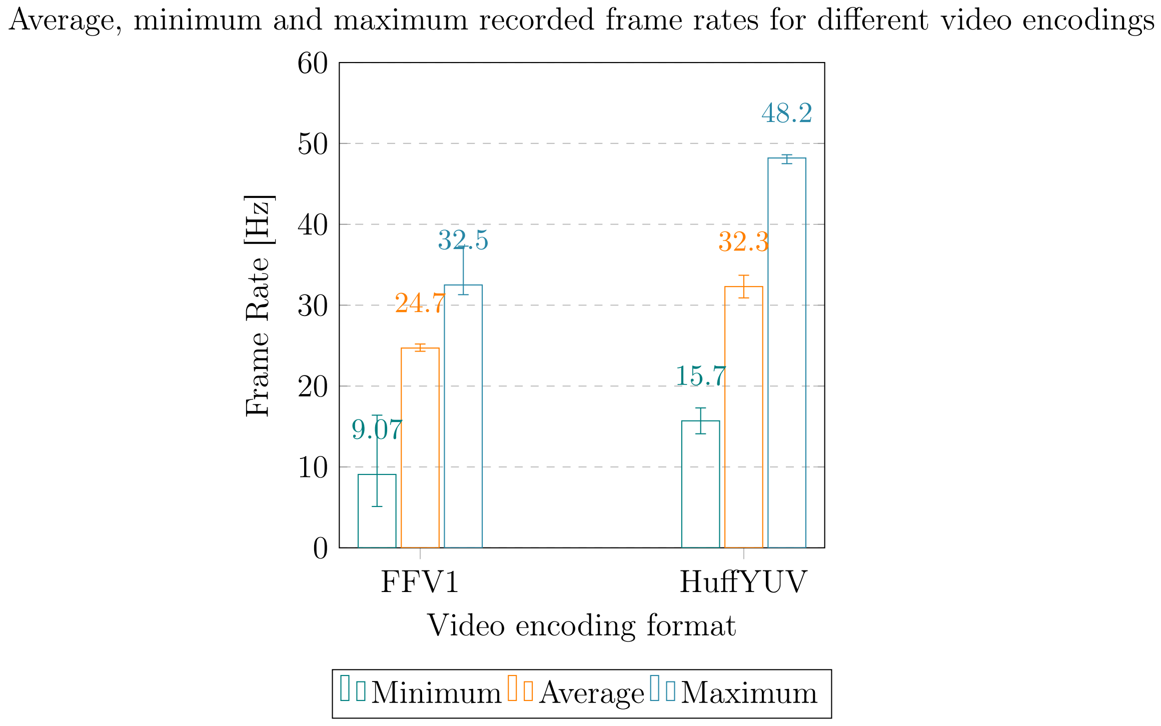


Figure C.5: Frame rates for different encoding formats for the scene "*Chain Collision*". HuffYUV sees greater framerates across the board, but the lower range for FFV1 indicate more consistent frame decoding speeds.

Video frame buffer size

Each background thread stores its processed video frames in a circular buffer. A larger buffer size means more frames can be prepared during slower rendering periods, and frames can be accessed more quickly during faster rendering periods, instead of waiting for them to be decoded. Figure C.6 demonstrates that the buffer size has little effect on neither the refresh rate nor the frame rate, indicating that the memory management of frames using the circular buffer was successful. Instead, the bottleneck for refresh rates lie in the resolution, and the bottleneck for frame rates lie in the decoding speed of FFmpeg. Overall, a buffer of 5 frames is chosen to reduce the memory footprint of the application, making it more suitable for lower-end devices.

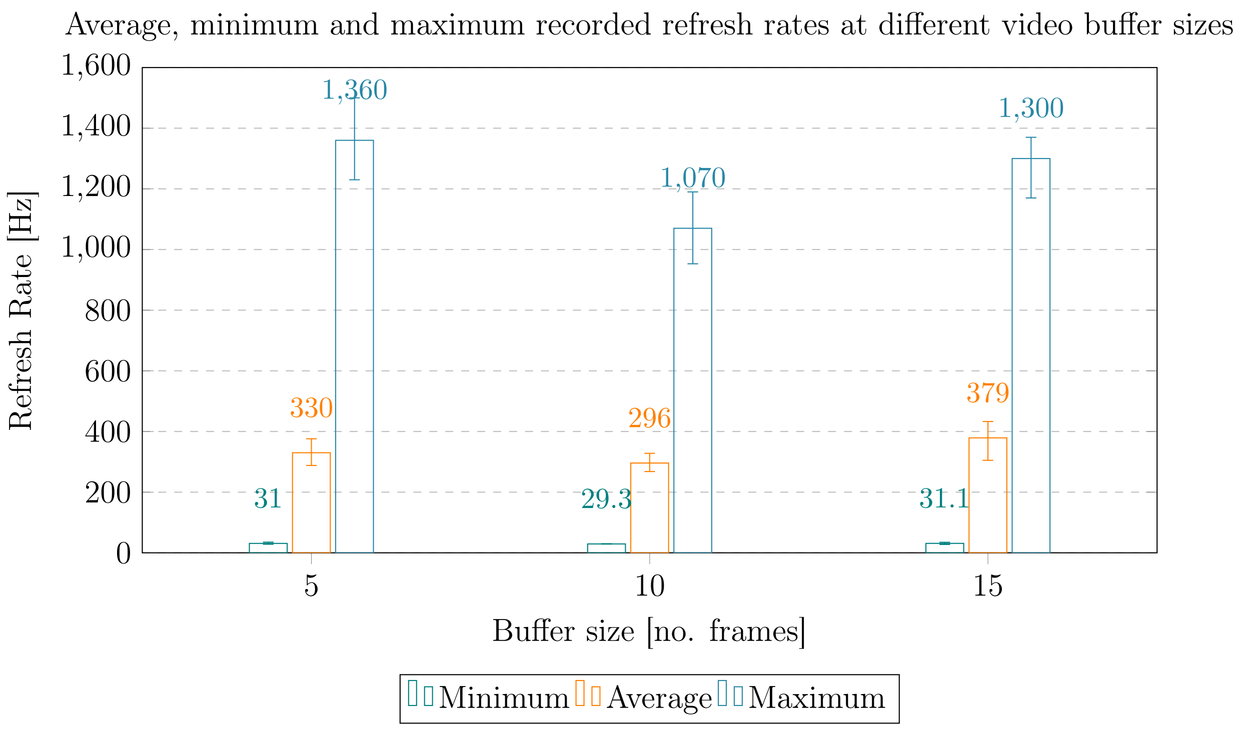


Figure C.6: Average refresh rates at different video buffer sizes for the scene "*Chain Collision*". Average refresh rates are within 10% of each other, indicating no change in performance.

Proposal

Real-Time Reconstruction and Rendering of RGB+Depth Video Scenes

Part II Project Proposal

2337G

October 16, 2023

Contents

1 Project Details

Originator: Dr Rafal Mantiuk
Supervisor: Dr Rafal Mantiuk
Director of Studies: Dr Ramsey Faragher

2 Introduction and Description

Constructing a 3D representation of some setting (a 'scene') captured by several images or videos has seen a variety of applications from more immersive video teleconferencing, to effective and safe traversal of areas by remotely operated or automated machinery [1][2]. Having a holistic view of the entire captured setting in a digital format is useful because it provides a continuous viewing space that isn't offered by just the set of recordings of cameras and sensors, and further provides a more intuitive way to explore a scene for human use. For VR and AR applications, a 3D representation is necessary in order to take advantage of the 6 degrees of freedom that human movement has. However, 3D reconstruction is still limited and expensive. Artefacts can impair visibility and can cause confusion for users; rendering times and computation resources still make this technology unfeasible for some portable devices.

There are several ways in which a setting can be captured in order to provide both a practical and effective method for constructing and rendering the 3D geometry. One such way, used by Lawrence et al, uses RGB video cameras, along with corresponding depth sensors for each camera, to generate depth maps of the captured setting [1]. The depth maps are combined to produce a digital representation of the geometry, and the RGB camera inputs are used to texture the geometry. The result is a 3D rendered representation of the captured setting as seen in Figure D.1.

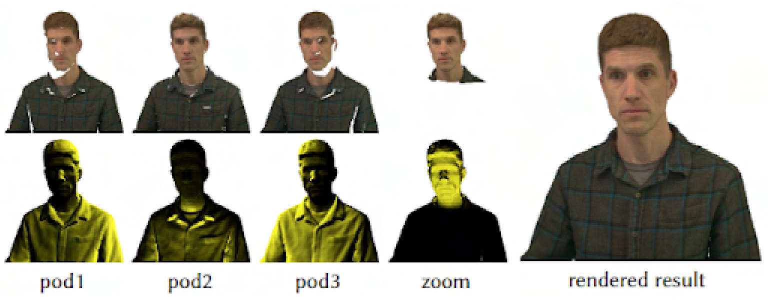


Figure D.1: A representation of measured RGB (top) and depth maps (bottom) combined to produce a final rendered result (see right). Obtained from Lawrence et al. [1]

Many challenges are posed both in the processes of capturing the setting, and in generating and rendering the 3D representation of the scene. When capturing the setting, it would be ideal to capture as much of the setting as possible (from a variety of positions and angles) in order to later generate the most accurate presentation of the scene. Not only is this impractical in terms of the amount of time and physical equipment needed, but also generates a copious amount of data to be processed, leading to long geometry generation times. Noise when using physical equipment, especially in places with poor visibility, is often inevitable, and can produce artefacts when the scene is rendered [3].

When generating and rendering the 3D scene, there are several factors and tradeoffs:

- Rendering high-fidelity geometry can provide a more precise result, but high polygon counts can result in expensive graphical processing, potentially leading to lower framerates.
- The geometry construction process may produce artefacts in the scene, especially at points where few instruments were able to record that part of the setting, or where depth measurements are inaccurate.

- If videos are taken of the scene, then the changes in the setting should be reflected in the scene in real-time (i.e. without stuttering, lag or buffering). Generating a mesh animation from video frames can also introduce difficulties when objects change shape and topology, which is difficult to capture in a mesh-based representation of the scene.

Previous works in this field attempt to reconcile these tradeoffs by carefully selecting the capturing and reconstruction process to best suit the specific usage scenario [1][4]. Lawrence et al. [1] present a 3D telecommunications system that uses several RGB+depth video streams and control the lighting and setting so as to avoid 'flat' looking surfaces and specular reflections.

Overbeck et al. [4] offer a real-time VR application that renders panoramic light field stills, using a compressed video stream algorithm, at 90Hz display refresh rate. For real-time applications, the bandwidth used when transferring several video streams (particularly at higher resolutions) may create a real bottleneck for performance.

Sun et al. [5] also offer a real-time reconstruction of a scene recorded with a single moving camera. The application presented has uses in augmented reality (AR), but does not require the retexturing of fused geometries.

My project will take several video streams with corresponding depth maps as input, and render a 3D scene that updates to show the 'action' of the video in real-time, and will be displayed in a VR application as an extension. Presenting the work in a VR application imposes stricter resolution and framerate conditions, but the production of a 3D scene in VR means that the resulting sense of parallax that one can obtain can be appreciated. As such, we would like to aim to render the scene at 90Hz for a comfortable viewing experience, and up to 120Hz as an ideal. I will be using OpenXR to develop the VR application, which is both cross-platform, and provides more flexibility at the rendering level. Allowing for use across several VR headsets will be useful if I need to borrow one in case of my own hardware failure.

The video streams, in addition to the depth maps, will be obtained by rendering a variety of open-licence Blender demo animations from several stationary viewpoints. The scripting feature within Blender will be used to easily obtain both of these streams from each camera.

The project will explore two geometry fusion strategies: the traditional voxel-based reconstruction, presented by Curless and Levoy [6] in the core; and an image-based approach offered by Lawrence et al. [1], with additional acceleration strategies for the voxel-based method presented by Nießner et al. [5], which will both be explored as an extension. The project will explore these techniques under the following evaluation metrics:

- **Rendering speed** - in which the techniques will be compared to the desired 90Hz of a VR display, with real-time video playback (i.e. not in slow motion or sped up).
- **Presence of artefacts** - a recording of the 3D scene from a desired viewpoint is created, and will be compared with a recording from the Blender scene at the same viewpoint, using ColorVideoVDP [7].

I will analyse the effectiveness of these techniques under those metrics, given the usage scenario and type of scene (for example, scenes with few cameras, or with complex geometries).

Finally, I will modify these rendering strategies to optimise the application for this specific project. This will be done by implementing and evaluating a mechanism which uses the temporal history of the video to determine which parts of the scene have changed, and to only schedule a re-rendering of specific regions. Lawrence et al. [1], for instance, exploits no temporal history, and so recomputes the entire scene for each frame. Its performance will be evaluated against the other strategies above under the same metrics.

3 Starting Point

I have experience with Blender and scripting using Python (Blender's scripting language). I have no experience with developing for VR, including SDKs or game engines for any VR platform. I am familiar with the concept of 3D scanning and geometry reconstruction from IB Further Graphics, but have no practical experience. I have practical experience with rendering, particularly in OpenGL, and am familiar with ray tracing and path tracing from IA Introduction to Graphics and Further Graphics, but have no practical experience of either of these outside of their respective courses. I also have no experience with ray-marching.

4 Success Criteria

The project will be a success if the following objectives are met:

1. At least 5 Blender scenes are rendered with RGB frames and depth to produce a suite of streams to be used as tests and input.
2. An interactive application that takes RGB and depth video streams as input, and a 3D scene is outputted using a geometry reconstruction strategy.
3. A modification or new strategy is implemented which is then evaluated on the same scenes using the same metrics.
4. The strategies are compared according to metrics that measure rendering speed (framerate) and presence of artefacts, and are evaluated in the context of scenes of varying geometries and number of cameras.

5 Possible Extensions

1. **An additional geometry reconstruction technique**: As discussed earlier, one technique for geometry fusion was proposed by Lawrence et al. [1] which utilises an image-based fusion technique. This extension would focus on implementing and evaluating this according to the same metrics as previous techniques, in addition to implementing an acceleration strategy presented by Nießner et al. [5]. The same proposed modification will also be applied to this technique to judge potential extra performance gains or losses over the image-based strategy with the modification, if any.
2. **Implementation for VR display**: Rendering 3D scenes for display on VR comes at an added cost due to the extra viewport and increased resolution. In addition, rendering speed constraints are introduced in order to provide a comfortable viewing experience for the user. As such, we would aim to produce a VR application which can render these scenes in real time at 90Hz, with 60Hz as a bare minimum. There are benefits to doing so: VR displays allow the user to make use of their sense of parallax, providing an extra layer of both immersion and appreciation for the 3D scene. Such constraints may also be reflected in practical applications, such as in remotely-operated machinery. I will be using OpenXR, a cross-platform development framework, to implement this. It also offers more flexibility at lower levels compared to game engines such as Unreal or Unity.
3. **Specular surfaces**: Reconstructing specular surfaces is difficult because the reflection changes as the viewing point moves. As such, since the video cameras are stationary, these surfaces need to be modelled explicitly. DelPozo and Savarese [8] offer a method for detecting specular surfaces, and Godard et al. [9] offer a method of reconstructing

specular surfaces. It may be possible to implement and apply these methods together, using the rest of the 3D scene as a reference point, to model a specular surface that will behave as a specular surface should; with the reflection changing as the user moves around.

6 Plan of Work

Weeks 1 & 2 (16th Oct - 29th Oct)

- Read through papers in more detail, learn about VR development (read through OpenXR documentation for MetaQuest), and set up necessary tools and SDKs.
- Prepare Blender RGB video + depth streams.

Milestone: All video+depth test suites prepared, and work environment is set up.

Weeks 3 & 4 (30th Oct - 12th Nov)

- Implement unoptimised voxel-based reconstruction strategy and simple 3D interactive application to view output (implemented using OpenGL).

Milestone: An output for voxel-based strategy is produced and ready for testing.

Weeks 5 & 6 (13th Nov - 26th Nov)

- Implement one acceleration strategy (voxel-hashing).
- Evaluate both technique and acceleration strategy for both rendering speed and artefact presence.

Milestone: Core techniques implemented except for my own.

Weeks 7 & 8 (27th Nov - 10th Dec) (end of Michaelmas term)

- Implement modifications for strategies to take advantage of temporal locality.
- Evaluate all strategies implemented so far with modifications for both rendering speed and artefacts.

Milestone: All strategies implemented and tested. Core project implementation finished.

Weeks 9 & 10 (11th Dec - 24th Dec) (Christmas)

- Buffer time to catch up
- Produce simple rasterized scene and VR application that can view it

Weeks 11 & 12 (25th Dec - 7th Jan)

- Implement image-based reconstruction strategy.

Weeks 13 & 14 (8th Jan - 21st Jan) (start of Lent term)

- Complete port onto VR display
- Evaluate techniques in the context of VR rendering
- Prepare progress report

Weeks 15 & 16 (22nd Jan - 4th Feb)

- Implement specular surface detection
- Submit progress report

Weeks 17 & 18 (5th Feb - 18th Feb)

- Implement specular surface reconstruction

Milestone: Specular surface work completed

Weeks 19 & 20 (19th Feb - 3rd Mar)

- Evaluate metrics for scenes with specular surfaces

Milestone: All project implementation finished

Weeks 21 & 22 (4th Mar - 17th Mar) (end of Lent Term)

- Buffer time to catch up

Weeks 23 & 24 (18th Mar - 31st Mar)

- Dissertation

Weeks 25 & 26 (1st Apr - 14th Apr)

- Dissertation

Weeks 27 & 28 (15th Apr - 28th Apr)

- Dissertation

Milestone: Finish dissertation draft

Weeks 29 & 30 (29th Apr - 10th May)

- Final tweaks

Milestone: Dissertation submitted

7 Resource Declaration

I will use my personal computer (Ryzen 9 7950X, RTX 4070Ti, 32GB RAM) for development and testing. I will also be using my own VR headset (Meta Quest 3) for testing. I have made contingency plans to mitigate hardware and software failures. Rafal Mantiuk has agreed to let me use a VR headset from the Department in case of emergency. The code I write for VR will be cross-platform.

I will use my personal laptop (i7-11800H 2.50GHz, RTX 3050Ti Laptop GPU, 16GB RAM) as a backup. I will make periodic backups to both a remote repository and to a local USB flash drive of my project and dissertation.

References

[1] Jason Lawrence, Danb Goldman, Supreeth Achar, et al. “Project Starline: A High-Fidelity Telepresence System”. In: *ACM Trans. Graph.* 40.6 (Dec. 2021). ISSN: 0730-0301. DOI: 10.1145/3478513.3480490. URL: https://doi.org/10.1145/3478513.3480490.

[2] William Agnew, Christopher Xie, Aaron Walsman, et al. “Amodal 3D Reconstruction for Robotic Manipulation via Stability and Connectivity”. In: *Proceedings of the 2020 Conference on Robot Learning.* Ed. by Jens Kober, Fabio Ramos, and Claire Tomlin. Vol. 155. Proceedings of Machine Learning Research. PMLR, 16–18 Nov 2021, pp. 1498–1508. URL: https://proceedings.mlr.press/v155/agnew21a.html.

[3] Nader Aldeeb and Olaf Hellwich. “3D Reconstruction Under Weak Illumination Using Visibility-Enhanced LDR Imagery”. In: *Advances in Computer Vision.* Jan. 2020, pp. 515–534. ISBN: 978-3-030-12138-9. DOI: 10.1007/978-3-030-17795-9_38.

[4] Ryan S. Overbeck, Daniel Erickson, Daniel Evangelakos, et al. “A System for Acquiring, Processing, and Rendering Panoramic Light Field Stills for Virtual Reality”. In: *ACM Trans. Graph.* 37.6 (Dec. 2018). ISSN: 0730-0301. DOI: 10.1145/3272127.3275031. URL: https://doi.org/10.1145/3272127.3275031.

[5] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, et al. “Real-Time 3D Reconstruction at Scale Using Voxel Hashing”. In: *ACM Trans. Graph.* 32.6 (Nov. 2013). ISSN: 0730-0301. DOI: 10.1145/2508363.2508374. URL: https://doi.org/10.1145/2508363.2508374.

[6] Brian Curless and Marc Levoy. “A Volumetric Method for Building Complex Models from Range Images”. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH ’96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 303–312. ISBN: 0897917464. DOI: 10.1145/237170.237269. URL: https://doi.org/10.1145/237170.237269.

[7] Rafal Mantiuk. URL: https://github.com/gfxdisp/ColorVideoVDP.

[8] Andrey DelPozo and Silvio Savarese. “Detecting Specular Surfaces on Natural Images”. In: *2007 IEEE Conference on Computer Vision and Pattern Recognition.* 2007, pp. 1–8. DOI: 10.1109/CVPR.2007.383215.

[9] Clément Godard, Peter Hedman, Wenbin Li, et al. “Multi-view Reconstruction of Highly Specular Surfaces in Uncontrolled Environments”. In: *2015 International Conference on 3D Vision.* 2015, pp. 19–27. DOI: 10.1109/3DV.2015.10.