IMPERIAL

MEng Individual Project

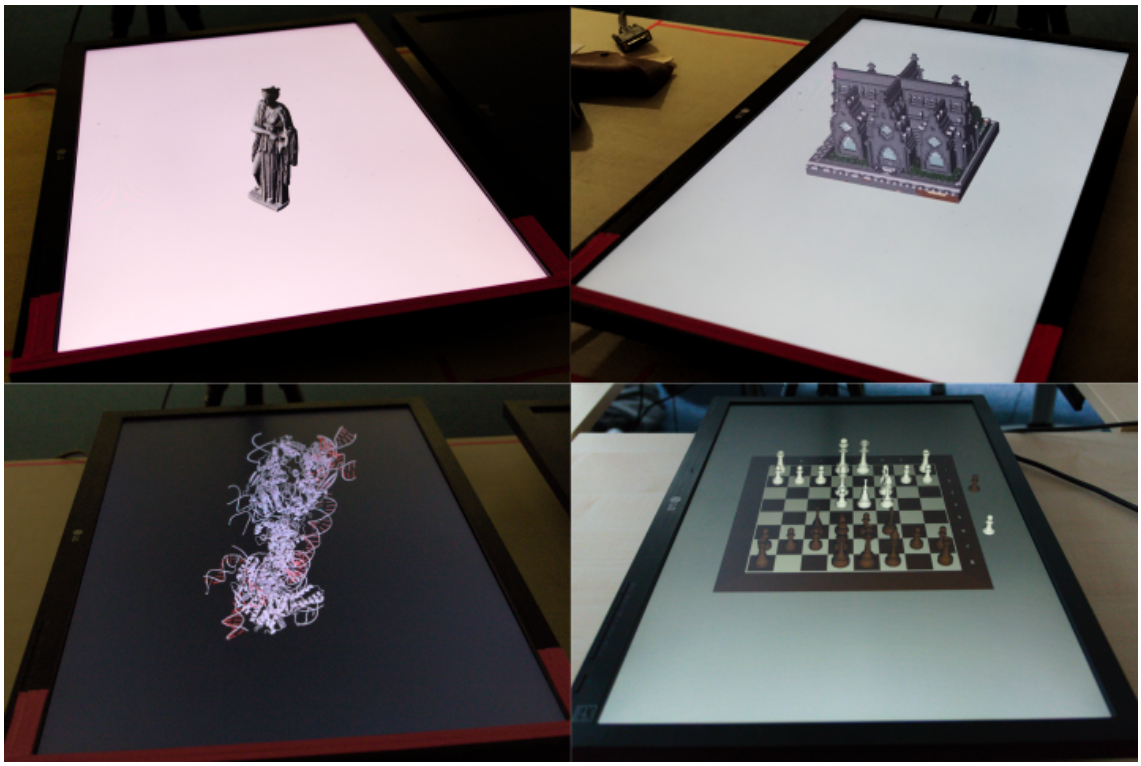
Department of Computing

Imperial College of Science, Technology and Medicine

A Virtual Volumetric Screen

Author:
Robert Searby Buxton

Supervisor:
Dr Nicole Salomons



January 1, 1980

Submitted in partial fulfillment of the requirements for the Computing MEng of Imperial College London

Abstract

TODO

Acknowledgments

I would like to thank **in order of importance**:

- My supervisor Dr Nicole Salomons for her guidance and support throughout this project so far.
- ChatGPT
- My loving family who still think I am studying physics for some reason.

Contents

Chapter 1

Introduction

1.1 Motivations

A volumetric display is a type of graphical display device that can provide a visual representation of objects natively in 3D. They can be viewed from any angle without the need for special visual apparatus by multiple people simultaneously [20]. These displays differ from more traditional virtual reality devices in that they are not immersive, but rather they are a window into a virtual world (See Fig 1.1.1 and Fig 1.1.2). There is no real consensus on what exactly constitutes a volumetric display, let alone the best way to build one. As we cover in the background section there are currently many different approaches being attempted by research groups both academic and industrial to create these displays.

Figure 1.1.1: One of Columbia University's passive optical scattering based volumetric displays [46]

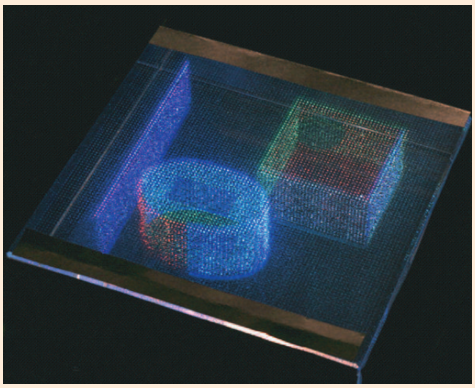
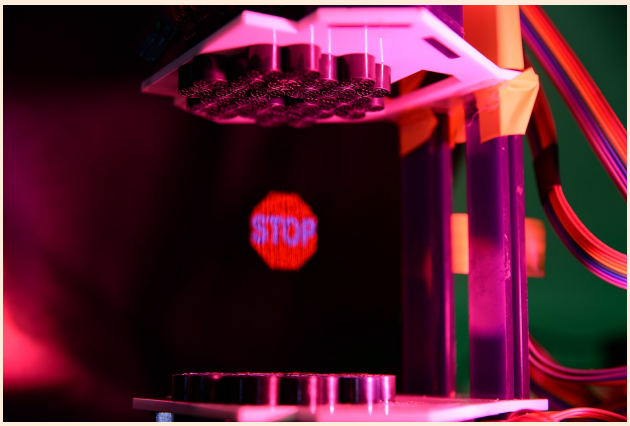


Figure 1.1.2: One of Bristol University's acoustic trapping based volumetric displays [23]



At the time of writing it is difficult to conduct human-computer interaction (HCI) research into the field of volumetric displays because these devices are not widely available, expensive and difficult to manufacture, and have extreme bandwidth requirements which makes it difficult to conduct user studies and experiments. People have created virtual simulations of volumetric displays before to try and solve this problem, but these solutions are often complicated and expensive to replicate.

1.2 Objectives

With the conclusion of this project, we aim to produce a cheap, multi-platform, lightweight and simple platform for simulating volumetric displays through the use of a Fish tank virtual reality (FTVR) device (See background). We hope that this will reduce the barrier to entry for researchers who wish to conduct HCI research in the field of volumetric displays. We aim to make the following contributions:

1.2.1 Volumetric Simulator

We plan to create a platform for simulating volumetric displays that is:

- **Multi-platform**: We have packaged our platform in the Nix package manager [15] which allows it to be easily run and ported to any platform and hardware that Nix supports.
- **Lightweight**: We use simple rending algorithms in OpenGL [50] to create our virtual volumetric display allowing our software to be computationally cheap to run compared to more fully-fledged rendering/games engines that might be used typically for HCI research like Unity.
- **Cheap**: By relying on only a generic depth camera and a standard display our software requires minimal hardware to run, making research conducted on our platform cheap and easy to run.
- **Reproducible:** By building with Nix we can guarantee that any experiments conducted using this platform will be completely reproducible. (See background)

1.2.2 User experiment

We plan to conduct an HCI user study to demonstrate the utility of our volumetric display simulation platform. We will conduct the user study to compare the relative effectiveness of using hand tracking to interact directly with an ethereal/incorporeal volumetric display compared to a via teleoperation with a corporeal/tangible display (See evaluation).

Chapter 2

Ethical Discussion

2.1 User Study

We ran a user study to evaluate our simulator titled ”A Virtual Volumetric Screen User Study”. We undertook Imperial College London’s ”Science Engineering Technology Research Ethics Committee process” which was reviewed by the ”Research Governance and Integrity Team” and the head of the computing department. We gained approval on the 2nd May 2024 and ran our study between May and July find first and last date of participants.

2.1.1 Human participants

As we ran our study on human participants we had to ensure we conducted our study in an ethical way. We consulted the Equality Act 2010 to make sure we did not exclude any participants ToCite . We were also careful to not use any participants that might feel perceived pressure to participate or might feel influenced by the researchers conducting the study. Participants were presented with an information sheet and a consent form before the study began.

2.1.2 Data collection

While collecting data for our study we were careful to comply with the local regulations (**GDPR**) ToCite . All data was stored on a secure computer on campus and was only accessible by the researchers, or was stored on Imperials secure cloud network. We also made sure to anonymise the data when presenting it in our report. Data will be retained until 30th June 2034.

2.2 Volumetric Simulator

2.2.1 Military applications

This technology in theory could be used for military applications, however, we believe is unlikely to be used for such purposes, and if it was, it would not be directly used in combat and would be no more dangerous than other existing technologies.

2.2.2 Copywrite Limitations

Open Source

We will be using the Azure Kinect SDK which is licensed under the MIT license. We will also be using the Nix package manager which is licensed under the LGPL-2.1 license. Furthermore, we will also be using the Nixpkgs repository which is licensed under the MIT license. We will also be using the `dlib` library which is licensed under the Boost Software License 1.0 (BSL-1.0). We will also be using the OpenGL library which is licensed under the open-source license for the use of sample Implementation (SI). Furthermore, we will also be using the GLFW library which is licensed under the zlib license. We will also be using the GLM library which is licensed under the MIT license. We will also be using the OpenCV license under the Apache License.

Proprietary

Furthermore, we use Microsoft's proprietary depth engine designed to work with the Azure Kinect SDK which is not open source.
Add other libraries we use

Chapter 3

Background

3.1 Nix/NixOS

3.1.1 Introduction to Nix

Nix [15] is an open-source, ”purely functional package manager” used in Unix-like operating systems to provide a functional and reproducible approach to package management. Started in 2003 as a research project Nix [14] is widely used in both industry [1] and academia [11] [39] [10], and its associated public package repository `nixpkgs` [22] as of Jan 2024 has over 80,000 unique packages making it the largest up-to-date package repository in the world [42]. Out of Nix has also grown **NixOS** [16] a Linux distribution that is conceived and defined as a deterministic and reproducible entity that is declared functionally and is built using the **Nix** package manager.

Nix packages are defined in the **Nix Language** a lazy functional programming language where packages are treated like purely functional values that are built by side effect-less functions and once produced are immutable. Packages are built with every dependency down to the `ELF` interpreter and `libc` (C standard library) defined in nix. All packages are installed in the store directory, typically `/nix/store/` by their unique hash and package name as can be seen in Fig 3.1.1 as opposed to the traditional Unix Filesystem Hierarchy Standard (FHS).

Figure 3.1.1: Nix Store Path

```
/nix/store/sbldylj3clbkc0aqvjjzfa6slp4zdvlj-hello-2.12.1
Prefix      Hash part                       Package name
```

Package source files, like tarballs and patches, are also downloaded and stored with their hash in the store directory where packages can find them when building. Changing a package's dependencies results in a different hash and therefore location in the store directory which means you can have multiple versions or variants of the same package installed simultaneously without issue. This design also avoids ”DLL hell” by making it impossible to accidentally point at the wrong version of a package. Another important result is that upgrading or uninstalling a package cannot ever break other applications.

Nix builds packages in a sandbox to ensure they are built exactly the same way on every machine by restricting access to nonreproducible files, OS features (like time and date), and the network [55]. A package can and should be pinned to a specific NixOS release (regardless of whether you are using NixOS or just the package manager). This means that once a package is configured to build correctly it will continue to work the same way in the future, regardless of when and where it is used and it will never not be able to be built.

These features are extremely useful for scientific work, CERN uses Nix to package the LHCb Experiment because it allows the software ”to be stable for long periods (longer than even long-term support operating systems)” and it means that as Nix is reproducible; all the experiments are completely reproducible as all bugs that existed in the original experiment stay and

ensure the accuracy of the results [10].

To create a package Nix evaluates a **derivation** which is a specification/recipe that defines how a package should be built. It includes all the necessary information and instructions for building a package from its source code, such as the source location, build dependencies, build commands, and post-installation steps. By default, Nix uses binary caching to build packages faster, the default cache is `cache.nixos.org` is open to everyone and is constantly being populated by CI systems. You can also specify custom caches. The basic iterative process for building Nix packages can be seen in Fig 3.1.2.

Figure 3.1.2: Nix Build Loop

1. A hash is computed for the package derivation and, using that hash, a Nix store path is generated, e.g `/nix/store/sbldylj3clbkc0aqvjjzfa6slp4zdvlj-hello-2.12.1`.
2. Using the store path, Nix checks if the derivation has already been built. First, checking the configured Nix store e.g `/nix/store/` to see if the path e.g `sbldylj3clbkc0aqvjjzfa6slp4zdvlj-hello-2.12.1` already exists. If it does, it uses that, if it does not it continues to the next step.
3. Next it checks if the store path exists in a configured binary cache, this is by default `cache.nixos.org`. If it does it downloads it from the cache and uses that. If it does not it continues to the next step.
4. Nix will build the derivation from scratch, recursively following all of the steps in this list, using already-realized packages whenever possible and building only what is necessary. Once the derivation is built, it is added to the Nix store.

3.1.2 Example of a Nix package

To give an example of what a Nix package might look like. We have created a flake (one method of defining a package) in Listing 3.1.3 that builds a version of the classic example package ”hello”.

Listing 3.1.3: flake.nix

```
{
  description = "A flake for building Hello World";
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";

  outputs = { self, nixpkgs }: {
    defaultPackage.x86_64-linux =
      let
        pkgs = nixpkgs.legacyPackages.x86_64-linux;
      in
      pkgs.stdenv.mkDerivation {
        name = "hello-2.12.1";
        src = self;
```

```
13          # Not strictly necessary as stdenv will add gcc
14          buildInputs = [ pkgs.gcc ];
15          configurePhase = "echo 'int main() { printf(\"Hello World!\"); }' > hello.c";
16          buildPhase = "gcc -o hello ./hello.c";
17          installPhase = "mkdir -p $out/bin; install -t $out/bin hello";
18        };
19    };
20 }
```

To dive deeper into what each line does we have given a breakdown below for the `flake.nix`

- **Line 2:** We have specified that we want to build our flake with the stable **nix channel** `nixos-23.11`, the most recent channel at the time of writing. This ”channel” is just a release branch on the `nixpkgs` GitHub repository. Channels do receive conservative updates such as bug fixes and security patches but no major updates after the initial release. The first time we build the hello package from our `flake.nix` a `flake.lock` is automatically generated that pins us to a specific revision of `nixos-23.11`. Our built inputs will not change until we relock our flake to either a different revision of `nixos-23.11` or a new channel entirely.
- **Line 5:** Here we define `outputs` as a function that accepts, `self` (the flake) and `nixpkgs` (the set of packages we just pinned to on line 2). Nix will resolve all inputs, and then call the `output` function.
- **Line 6:** Here we specify that we are defining the default package for users on `x86_64-linux`. If we tried to build this package on a different CPU architecture like for example ARM (`aarch64-linux`) the flake would refuse to build as the package has not been defined for ARM yet. If we desired we could fix this by adding a `defaultPackage.aarch64-linux` definition.
- **Line 7-9:** Here we are just defining a shorthand way to refer to x86 Linux packages. This syntax is similar if not identical to Haskell.
- **Line 10:** Here we begin the definition of the derivation which is the instruction set Nix uses to build the package.
- **Line 14:** We specify here that we need `gcc` in our sandbox to build our package. `gcc` here is shorthand for `gcc12` but we could specify any c compiler with any version of that compiler we liked. If you desired you could compile different parts of your package with different versions of GCC.
- **Line 15:** Here we are slightly abusing the configure phase to generate a hello.c file. You would usually download a source to build from with a command like `fetchurl` while providing a hash. Each phase is essentially run as a bash script. Everything inside `mkDerivation` is happening inside a sandbox that will be discarded once the package is built (technically after we garbage collect).
- **Line 16:** Here we actually build our package
- **Line 17:** In this line we copy the executable we have generated which is currently in the sandbox into the actual package we are producing which will be in the store directory `/nix/store`.

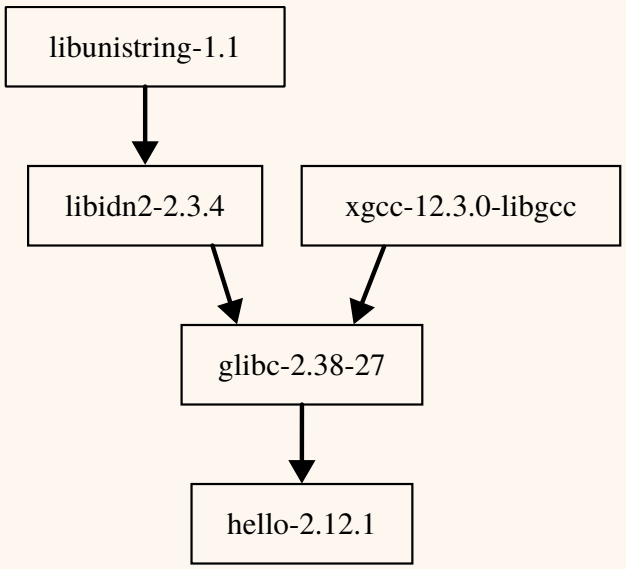
Below we have given some examples of how to run and investigate our hello package in Listing 3.1.4.

Listing 3.1.4: Terminal

```
[shell:~]$ ls
flake.lock  flake.nix

[shell:~]$ nix flake show
└──defaultPackage
   └──x86_64-linux: package 'hello-2.12.1'

[shell:~]$ nix run .
Hello, world!

[shell:~]$ nix path-info .
"\nix\store\sbldylj3clbkc0aqvjjzfa6slp4zdvlj-hello-2.12.1"

[shell:~]$ tree $(nix path-info .)
"\nix\store\sbldylj3clbkc0aqvjjzfa6slp4zdvlj-hello-2.12.1"
└──bin
    └──hello

[shell:~]$ nix-store --query $(nix path-info .) --requisites
/nix/store/s2f1sqfsdi4pmh23nfnrh42v17zsvi5y-libunistring-1.1
/nix/store/08n25j4vxyjidjf93fyc15icxwrxm2p8-libidn2-2.3.4
/nix/store/lmidwx4id2q87f4z9aj79xwb03gsmq5j-xgcc-12.3.0-libgcc
/nix/store/qn3ggz5sf3hkjs2c797xf7nan3amdxmp-glibc-2.38-27
/nix/store/sbldylj3clbkc0aqvjjzfa6slp4zdvlj-hello-2.12.1
```

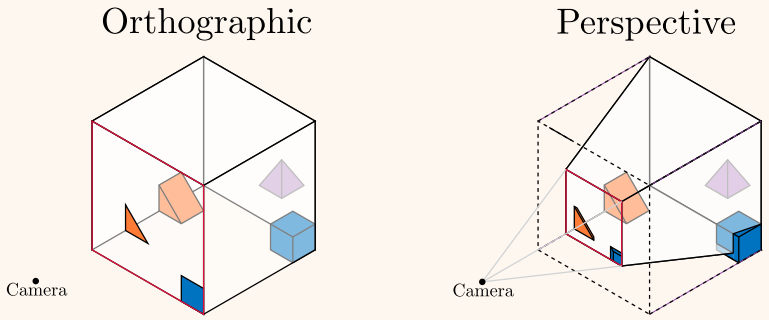
In Fig 3.1.5 we can see the package dependency graph of our `hello` package. We are only dependent on 4 packages `libunistring`, `libidn2`, `xgcc`, `glibc` all of which Nix have installed and configured separately the rest of the non-nix system (assuming we are not on NixOS).

Figure 3.1.5: Dependency graph



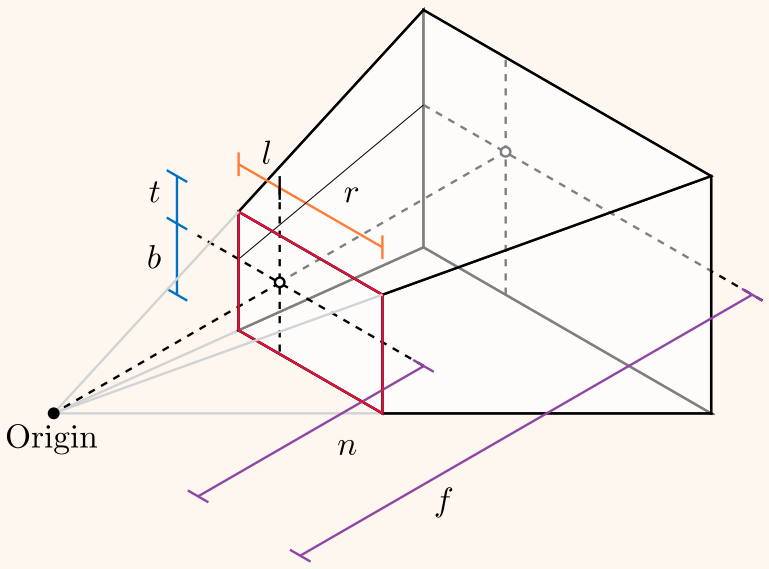
3.2 Perspective Projection

Figure 3.2.1: Orthographic and perspective projections



To represent 3D space on a 2D surface OpenGL supports two types of projections: perspective and orthographic as seen in Fig 3.2.1. Orthographic features parallel projection lines (orthogonal to the projection plane), which means that it does not depict the effect of perspective. Distances are preserved, making it useful for technical drawings where measurements need to be precise and not skewed by perspective (For example all diagrams in this report are from the orthographic perspective). Unlike orthographic projections, perspective projections simulate the way the human eye perceives the world, with objects appearing smaller the further away they are from the viewpoint as the projection lines converge at a vanishing point on the horizon. If we wish to create the illusion of volumetric display in this project we must use a perspective projection.

Figure 3.2.2: Using frustum to generate a perspective projection



OpenGL provides the `frustum` function as seen in Fig 3.2.2 which can be used to construct the perspective matrix (it is worth noting that OpenGL uses homogeneous coordinates so the matrix is 4x4):

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

This maps a specified viewing frustum to screen space (with intermediate steps handled by OpenGL) [28]. This viewing frustum is specified by six parameters: l, r, b, t, n and f which represent the left, right, bottom, top, near, and far extents of the frustum. These parameters define the sides of the near-clipping plane, highlighted in red, relative to the origin of the coordinate system. These parameters do not represent distances or magnitudes in a traditional sense but rather define the vectors from the center of the near-clipping plane to its edges.

The l and r parameters specify the horizontal boundaries of the frustum on the near-clipping plane, with the left typically being negative and the right positive, defining the extent to which the frustum extends to the left and right of the origin. Similarly, the b and t parameters determine the vertical boundaries, with the bottom often negative and the top positive, expressing the extent of the frustum below and above the origin.

The n and f parameters are scalar values that specify the distances from the origin to the near and far clipping planes along the view direction. Altering the value of n will change the angles of the lines that connect the corners of the near plane to the eye, effectively changing the "field of view". Changing the value f affects the range of depth that is captured within the scene but not the view.

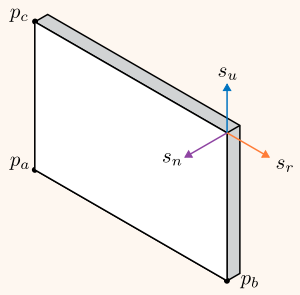
If we can track the position of a viewer's eye in real time then we can create the illusion of a 3D scene behind and in front of a display using this `frustum` function. This can be done fairly trivially following Robert Kooima's method he sets out in "Generalized Perspective Projection" to calculate f, l, r, b, t, n as the viewer's eye changes position [38].

3.2.1 Generating the perspective projection

The first step we must take is to record the position of the screen we are projecting onto in 3D space relative to the coordinate system of the tracking device, "tracker-space". To encode the position and size of the screen we take 3 points, p_a, p_b and p_c which represent the lower-left, lower-right and upper-left points of the screen respectively when viewed from the front on. These points can be used to generate an orthonormal basis for the screen comprised of s_r, s_u and s_n which represents the directions up, right and normal to the screen respectively as seen in Fig 3.2.3. We can compute these values from the screen corners as follows:

$$s_r = \frac{p_b - p_a}{||p_b - p_a||} \quad s_u = \frac{p_c - p_a}{||p_c - p_a||} \quad s_n = \frac{s_r \times s_u}{||s_r \times s_u||}$$

Figure 3.2.3: Defining a screen in 3D space



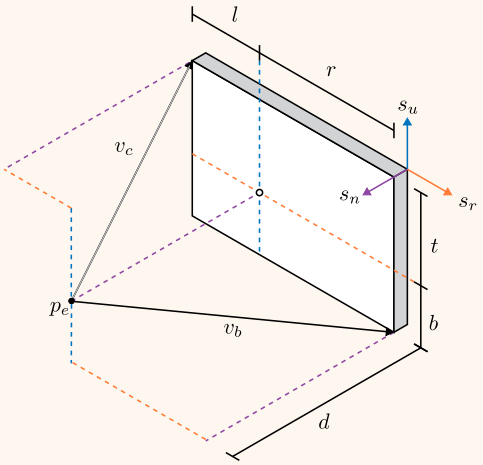
Next, we introduce the viewer's eye which we will refer to as p_e. We can draw 2 vectors v_b, v_c from the viewer's eye p_e to the corners of the screen p_b, p_c as seen in Fig 3.2.4. In the diagram, we also have labeled the components of each of these vectors in the basis of the screen. We can compute these as follows:

$$v_a = p_a - p_e \quad v_b = p_b - p_e \quad v_c = p_c - p_e$$

To calculate the required values for our `frustum` OpenGL function we must first find the point where a line drawn perpendicular to the plane of the screen that passes through p_e strikes the screen. We refer to this point as the *screen-space-origin*, it is worth noting that this point can lie outside the screen (the rectangle bounded by p_a, p_b, p_c). We can find the distance of the *screen-space-origin* from the eye p_e by taking the component of the screen basis vector s_n in either of the vectors v_b and v_c. However, as s_n is in the opposite direction we must invert the result. Similarly, we can calculate t by taking the component of v_c in the basis vector s_u, b by v_b in s_u, l by v_c in s_r and lastly r by v_b in s_r. We can compute these as follows:

$$d = -(s_n \cdot v_a) \quad l = (v_c \cdot s_r) \quad r = (v_b \cdot s_r) \quad b = (v_b \cdot s_u) \quad t = (v_c \cdot s_u)$$

Figure 3.2.4: Screen Intersection with view

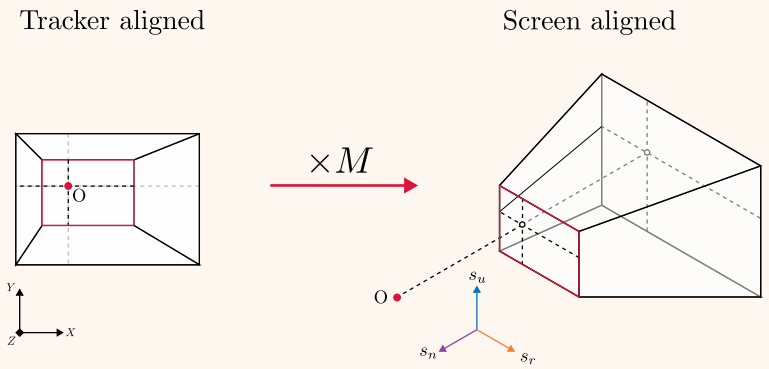


We can now generate a projection matrix by calling `frustum` using d as our near-clipping plane distance n with an arbitrary value for the far-clipping plane f depending on our required scene

depth. We have now successfully generated our viewing frustum but we still have a few issues. Firstly our frustum has been defined in tracker space so it is aligned with the direction of our camera not the normal of our screen. We can remedy this by applying a rotation matrix M to align our frustum with s_n, s_u and s_r, the basis of our screen as seen in Fig 3.2.5. M is defined as follows:

$$\begin{bmatrix} v_{rx} & v_{ry} & v_{rz} & 0 \\ v_{ux} & v_{uy} & v_{uz} & 0 \\ v_{nx} & v_{ny} & v_{nz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

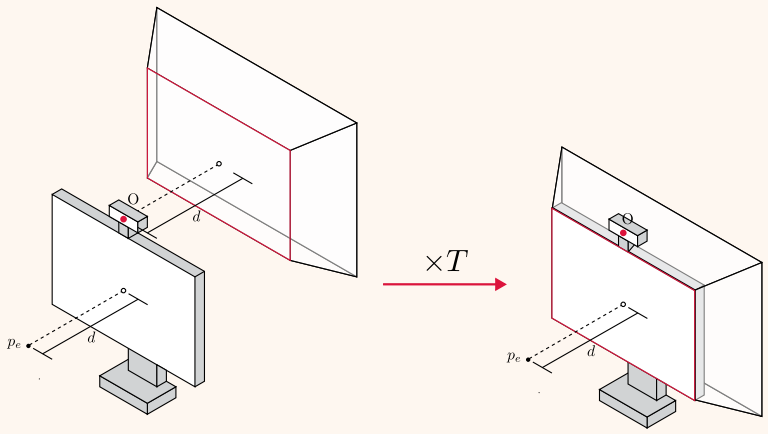
Figure 3.2.5: Rotating the frustum from tracker space alignment into screen space alignment



The second problem we have is that we want our projection matrix to move around with the viewer's eye however the mathematics of perspective projection disallow this, with the camera assumed to be at the origin. To translate our viewing frustum to our eye position we must instead translate our eye position (and the whole world) to the origin of our frustum. This can be done with a translation matrix T as seen in Fig 3.2.6. T can be generated with the OpenGL function `translate` where we want to offset it by the vector from our Origin to the viewer's eye p_e. T is defined as follows:

$$\begin{bmatrix} 1 & 0 & 0 & -p_{ex} \\ 0 & 1 & 0 & -p_{ey} \\ 0 & 0 & 1 & -p_{ez} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3.2.6: Translating the viewing frustum to sit inside the screen

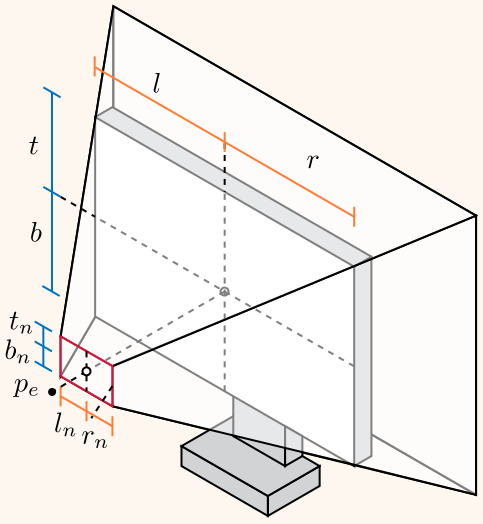


We now have a working method for projecting virtual objects behind our screen onto our screen however it is also possible if we desire to project objects in front of our screen onto the screen as well as long as they lie within the pyramid formed between the edges of the screen and the viewer's eye. We can scale the near-clipping plane from the plane of the screen to a small distance n from our eye as seen in Fig 3.2.7 giving us scaled-down values of t, b l and r we can use for our new viewing frustum which we call t_n, b_n l_n and r_n. They are defined as follows:

$$l_n = (v_c \cdot s_r)\frac{n}{d} \quad r_n = (v_b \cdot s_r)\frac{n}{d} \quad b_n = (v_c \cdot s_u)\frac{n}{d} \quad t_n = (v_b \cdot s_u)\frac{n}{d}$$

So our final viewing frustum takes in the frustum extents t_n, b_n l_n and r_n and n and f defining the distances to the near and far clipping plane.

Figure 3.2.7: Extending the near plane to not clip out objects in front of the screen



Following these steps, we can create an accurate projection providing the perspective we would

expect to see if there was a scene in front and behind our screen.

3.2.2 Sample code

Below in Listing 3.2.8 we have given an example of a function implementing the process we have just described in C++.

Listing 3.2.8: projection.cpp, Sample code for creating the 3D illusion projection

```cpp
#include <glad/gl.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

using namespace glm;

mat4 projectionToEye(vec3 pa, vec3 pb, vec3 pc, vec3 eye, GLfloat n, GLfloat f)
{
    // Orthonormal basis of the screen
    vec3 sr = normalize(pb - pa);
    vec3 su = normalize(pc - pa);
    vec3 sn = normalize(cross(sr, su));

    // Vectors from eye to opposite screen corners
    vec3 vb = pb - eye;
    vec3 vc = pc - eye;

    // Distance from eye to screen
    GLfloat d = -dot(sn, vc);

    // Frustum extents (scaled to the near clipping plane)
    GLfloat l = dot(sr, vc) * n / d;
    GLfloat r = dot(sr, vb) * n / d;
    GLfloat b = dot(su, vb) * n / d;
    GLfloat t = dot(su, vc) * n / d;

    // Create the projection matrix
    mat4 projMatrix = frustum(l, r, b, t, n, f);

    // Rotate the projection to be aligned with screen basis.
    mat4 rotMatrix(1.0f);
    rotMatrix[0] = vec4(sr, 0);
    rotMatrix[1] = vec4(su, 0);
    rotMatrix[2] = vec4(sn, 0);

    // Translate the world so the eye is at the origin of the viewing frustum
    mat4 transMatrix = translate(mat4(1.0f), -eye);

    return projMatrix * rotMatrix * transMatrix;
}
```

3.3 Volumetric displays

Volumetric displays [20] provide a three-dimensional viewing experience by emitting light from each voxel, or volume element, in a 3D space. This approach enables the accurate representation of virtual 3D objects while providing accurate focal depth, motion parallax, and vergence. Vergence refers to the rotation of a viewer's eye to fixate on the same point they are focusing on. Moreover, volumetric displays allow multiple users to view the same display from different angles, providing unique perspectives of the same object simultaneously.

3.3.1 Swept Volume Displays

Swept volume displays are one prominent category of volumetric displays. They employ a moving 2D display to create a 3D image through the persistence of vision effects. This is achieved by moving the 2D display through a 3D space at high speeds while emitting light from the display where it reaches the position of each corresponding voxel. Common techniques for achieving this include using a rotating mirror [21], an emitting screen, typically an LED-based [25], or a transparent projector screen [33]. There currently exist commercial products that implement this technique as can be seen in Fig 3.3.1 and Fig 3.3.2.

Figure 3.3.1: The VXR4612 3D Volumetric Display, a projector-based persistence of vision display produced by Voxon Photonics. [60]

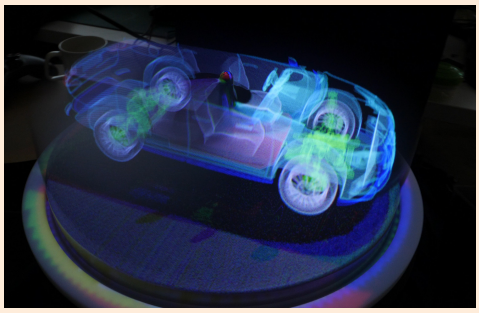


Figure 3.3.2: A Volumetric Display / Holographic Signage, an LED-based persistence of vision display produced by Brightvox Inc. [9]

3.3.2 Static Volume Displays

Static volume displays are another category. They employ a static transparent medium that when interacted with creates a 3D image. The result is that light is emitted from the display at each point in a 3D space. Techniques for achieving this range from using a 3D array of LEDs [51], lasers and phosphorus gas [61], or a transparent laser-induced damaged medium that can be projected into [46]. There has been research into photon-activated dye [48] and even quantum dot-based displays [31].

3.3.3 Trapped Particle Displays

Acoustic Trapping Displays displays are a relatively new category of volumetric displays. They employ a 3D array of particles that are suspended in air using acoustic levitation. [23] [30] This is achieved by using an array of ultrasonic transducers to create a standing wave that can trap particles in the nodes of the wave. By moving the nodes of the wave through a 3D space and illuminating the particles with light, a 3D image can be created. This technique is still in its infancy and can struggle to provide a convincing persistence of vision effect. Another direction some researchers have taken is to use a photophoretic trap to trap particles in air [54]. The advantage of this sort of display is that space that is not being used to display an object is empty and can be passed through. This is in contrast to swept volume displays/most static volume displays where the space not being used to display an object is filled with the display's hardware.

3.3.4 Issues

Volumetric displays often require custom/cutting-edge hardware (e.g. extremely high refresh rate projectors, transparent micro LEDs, complex laser systems) which makes them expensive, difficult to manufacture and calibrate and not widely available. For example, the Voxon VX1, one of the few if only commercially available volumetric displays costs, $11,700 USD [49] per unit.

Volumetric displays are also held back by their inherent high bandwidth requirements: To render objects in real-time at equivalent resolutions to current 2D displays while taking a raw voxel stream (as opposed to calculating voxels on hardware from primitive shapes) has an extremely high bandwidth requirement. If we want to render at `60fps` on a $4096 \times 2160 \times 1080$ voxel display with `24 bit` color, it would require a bandwidth of 1.37×10^3 bits per second/13.7 terabits per second which is orders of magnitude higher than what a normal display requires. To achieve that currently would require about 170 state-of-the-art Ultra High Bit Rate (UHBR) (80 gigabit) DisplayPort cables simultaneously. It was predicted in 2021 [6] that due to these limitations and based on the historic trends of bandwidth in commercially available displays, volumetric displays will only become feasible in 2060 at the earliest. There are ways to reduce this bandwidth requirement through compression and other techniques [67] but this still provides a major issue.

3.3.5 Volumetric Screen Simulations

Because of these issues, there has been some research into simulating volumetric displays. One commonly used method is the so-called fish tank virtual reality (FTVR) display [63] which has been commonly used to simulate volumetric displays, [19], [65]. A FTVR comprises a singular or set of 2D displays that are positioned in front of a user. The viewer's eyes are tracked in 3D space and the image on the displays is adjusted accordingly so that there appears to be a 3D image present in front of them. This is a relatively cheap and easy way to simulate a volumetric display, but it has some major drawbacks. The user is limited to a single focal depth and is limited to a single vergence (This can be fixed by wearing glasses to filter different images to

each eye providing a stereo view [58]). This system is also limited to just a single user at a time unless image filtering is used.
Another approach that has been taken is to take advantage of virtual reality (VR) headsets. VR headsets are a relatively cheap and easy way to simulate a volumetric display. They are also able to provide a stereo view and can be used by multiple users at once [18].

Chapter 4

Implementation

4.1 Overview

Building a Volumetric Simulator from scratch was a complex task that required the integration of various technologies and components. The most significant parts of the system are listed below, and we will discuss them in more detail in the following implementation sections.

Simulator: The simulator (`libvolsim.so`) is a shared library written in C++ that provides a graphical interface creating the illusion of a 3D display that can be interacted with.

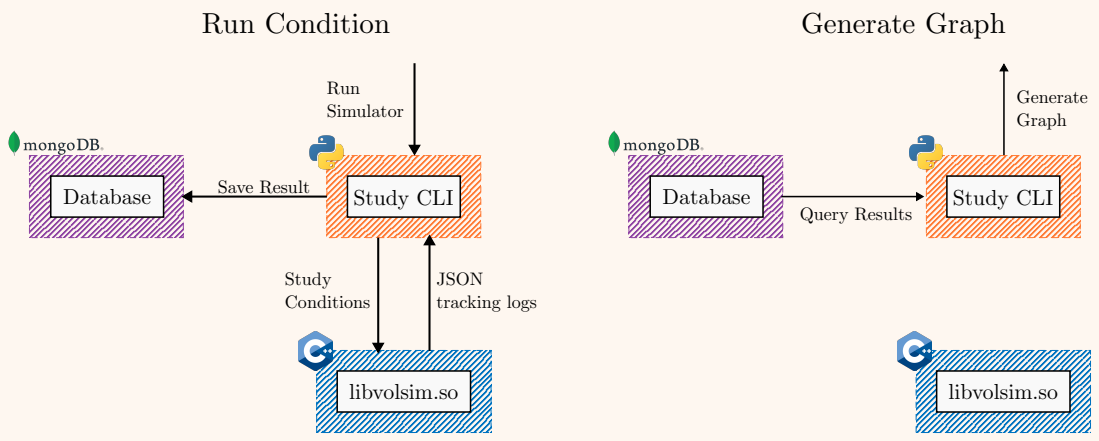
- **Renderer Subsystem**: The rendering system is responsible for displaying the volumetric scene and the virtual objects within it. It uses OpenGL to render the scene based on user positional data from the tracking system, thus creating the illusion of 3D.
- **Tracking Subsystem**: The tracking system monitors the user's hands and eyes within the scene. It employs machine learning models and a depth camera to track the user's head and eyes.

User Study CLI: The User Study CLI (command-line interface) orchestrates the running of the simulator and stores the results of the user study in a MongoDB database. It also provides functionality to automatically analyze study data. This component is written in Python.

Build System: The build system compiles the simulator and rendering system, as well as manages the execution of the user study. It uses Nix to handle dependencies and build the system in a declarative manner.

The user interacts with the `Study CLI`, to orchestrate running simulations by invoking the simulator's shared library `libvolsim.so`. The simulator returns its results/logs to the User Study CLI (in JSON format), which then stores the data in a MongoDB database. The User Study CLI can subsequently be used to analyze the results. A schematic of the system layout is shown in Fig 4.1.1.

Figure 4.1.1: Two examples of using our system



4.2 Build System

The build system plays a crucial role in compiling the simulator and rendering system, as well as in running the user study. Our simulator is a complex system that requires the compilation of C++, CUDA, and Python code, the management of large machine learning models, object files for rendering, and the handling of components such as camera drivers.

Portability is essential to ensure that the user study can be conducted on various systems. Given the author's previous experiences with graphical and hardware-dependent research projects, getting a project to build consistently on different machines is often a significant challenge.

One of the key goals of this project was to make the system as easy to build and run as possible on a variety of machines. This was important to ensure that the system could be used by other researchers in the future. To address these challenges, we chose to use Nix for our build system due to its declarative nature and ease of dependency management, including the ability to modify packages globally using overlays.

4.2.1 Overview

The build system provides two main sets of functionalities to the user.
Firstly, it offers the Nix package `volumetricSim-0.0.1`, which serves as an automated set of instructions for compiling the simulator and its dependencies into a shared library (`libvolsim.so`) from scratch. Secondly, it provides a set of development environments designed for running the user study and for developing the Volumetric Simulator using Visual Studio Code (VSCode).
Both of these functionalities are accessible through the `nix flake` interface, as demonstrated in Listing 4.2.1.

Listing 4.2.1: Terminal

```
[VolumetricSim]$ nix flake show
git+file:///home/robbieb/Projects/VolumetricSim
├──devShells
│   └──x86_64-linux
│       ├──default: development environment 'volumetricSim'
│       ├──start-mongodb: development environment 'mongodb-shell'
│       └──userstudy: development environment 'userstudy-shell'
└──packages
    └──x86_64-linux
        └──default: package 'volumetricSim-0.0.1'
```

4.2.2 VolumetricSim Package

You can build our simulator as a shared library using the following one-liner command from inside the main repository, as shown in Listing 4.2.2:

Listing 4.2.2: Terminal

```
[VolumetricSim]$  nix build
```

Alternatively, if you do not want to clone the repository, you can build the simulator without cloning it by taking advantage of Nix's ability to build from GitHub, as demonstrated in Listing 4.2.3.

Listing 4.2.3: Terminal

```
[home]$  nix build github:RobbieBuxton/VolumetricSim
```

Although these may appear to be simple commands, they perform a significant amount of work behind the scenes. Firstly, they fetch all the dependencies required to build the simulator from source (or a public binary cache). You do not need to have any of these dependencies installed on your system, as Nix will manage all of this for you.
Our packages configure the following components:

1. **CUDA:** Since we use the CUDA parallel computing platform [40] in our simulator, we need to build the CUDA toolkit. Fortunately, Nix allows for a more fine-grained approach, enabling you to build only the components you need. We utilize the CUDA Deep Neural Network library (cuDNN), CUDA Basic Linear Algebra Subprograms library (cuBLAS), CUDA Random Number Generation library (cuRAND), and CUDA Dense Linear Solver library (cuSOLVER), along with the necessary libraries to interact with them. We override and recompile OpenCV and Dlib to be CUDA-enabled.

2. **MKL:** We use the Intel Math Kernel Library (MKL) [62] for some of our linear algebra operations, as it is significantly faster than the default BLAS library. We override and recompile OpenCV and Dlib to use the MKL BLAS libraries.

3. **Azure Kinect Sensor SDK:** We have created our own Nix package for the Azure Kinect SDK [45], as it was not previously packaged. This package provides the necessary drivers and stubs for the Azure Kinect camera to function.

4. **OpenGL:** We download and configure GLFW [26] (a lightweight OpenGL utility library) and GLAD [29] (hardware-specific OpenGL drivers) for the programming interface used in rendering 2D and 3D vector graphics with OpenGL [64].

5. **Tracking Models:** We download and configure Dlib [36] and MediaPipe [41], which are machine learning libraries used for our tracking models. We also automatically download the two required models for Dlib from the internet (verified by hash). Additionally, we download and build OpenCV for image management.

Once all the dependencies are built, the simulator is compiled in a sandbox environment before being copied into the Nix store as a package containing the shared library `libvolsim.so` and all files the simulator depends on (tracking models, OBJ files, shaders). This shared library can

be accessed from the User Study CLI to run the simulation. An overview of the final package contents is provided in Listing 4.2.4.
move to appendix

Listing 4.2.4: Terminal

```
[VolumetricSim]$  tree result
/nix/store/gasc1x5y75rz6qdjz33jq1ffid3az9q7-volumetricSim-0.0.1/
├── bin
│   └── libvolsim.so
└── data
    ├── challenges
    │   ├── demo1.txt
    │   ├── demo2.txt
    │   ├── task1.txt
    │   ├── task2.txt
    │   ├── task3.txt
    │   ├── task4.txt
    │   └── task5.txt
    ├── mediapipe
    │   └── modules
    │       ├── hand_landmark
    │       │   ├── handedness.txt
    │       │   ├── hand_landmark_cpu.binarypb
    │       │   ├── hand_landmark_full.tflite
    │       │   ├── hand_landmark_landmarks_to_roi.binarypb
    │       │   ├── hand_landmark_model_loader.binarypb
    │       │   ├── hand_landmark_tracking_cpu.binarypb
    │       │   └── palm_detection_detection_to_roi.binarypb
    │       └── palm_detection
    │           ├── palm_detection_cpu.binarypb
    │           ├── palm_detection_full.tflite
    │           └── palm_detection_model_loader.binarypb
    ├── dlib
    │   └── modules
    │       ├── face_detection
    │       │   └── mmod_human_face_detector.dat
    │       └── face_landmark
    │           └── shape_predictor_5_face_landmarks.dat
    ├── resources
    │   ├── materials
    │   │   └── scene.mtl
    │   └── models
    │       ├── cube.obj
    │       ├── cylinder.obj
    │       └── sphere.obj
    └── shaders
        ├── camera.fs
        ├── camera.vs
        ├── image.fs
        └── image.vs
```

4.2.3 Development Environments

To facilitate our user study, we have created a development environment that includes all the necessary dependencies, primarily Python libraries, required to run the user study. Our shell script will also set up an alias that enables you to run the study via a CLI easily, as shown in Listing 4.2.5.

Listing 4.2.5: Terminal

```
[VolumetricSim]$ nix develop .#userstudy
Type 'study' to start the user study application.
[VolumetricSim]$ study --help

  Command line interface for running the Volumetric User Study.

Options:
  --help  Show this message and exit.

Commands:
  add   [ user ]
  list  [ users | results ]
  run   [ debug | eval | demo | task | next ]
  save  [ user ]
  show  [ result | task | eval ]
```

Additionally, we have developed a separate environment containing all the dependencies needed to run the analysis and fully reproduce the graphs and tables presented in this document.
Finally, there is a development environment designed to automatically launch and manage a local MongoDB database for storing the results of the user study. This environment includes GUI tools, such as MongoDB Compass, for viewing and editing the database. This can be activated by running the command shown in Listing 4.2.6.

Listing 4.2.6: Terminal

```
[VolumetricSim]$ nix develop .#start-mongodb
```

4.2.4 Additional Efforts

One significant drawback of using Nix is that if a package is not already available, you usually have to package it yourself. Fortunately, almost everything we required was already available in the Nix package repository (nixpkgs), but there were a few exceptions. Typically, if a package is not available, it is because it is either not widely used or it is challenging to package.

Azure Kinect Package

The first project we had to package was the Azure Kinect SDK. This was not available in nixpkgs, and the only official package was a poorly ported, outdated Ubuntu binary from Windows. To package it in Nix, we had to manually patch the rpaths (run-time search paths hard-coded in an executable file) and resolve build and driver issues. Microsoft officially stopped supporting the Azure Kinect in August 2023 [44], so we decided to package a fork of `https://github.com/microsoft/Azure-Kinect-Sensor-SDK` that addressed the build issues we encountered. This process was quite challenging and required about a week of work. We have not yet upstreamed this package to nixpkgs but hope to do so in the future; it is currently available for all to use on GitHub.

Dlib Package

The second project involved packaging Dlib. Although Dlib was available in nixpkgs, we discovered that CUDA support had been incorrectly implemented (the maintainer had simply toggled a CMake flag without actually adding CUDA support). We were able to implement this locally using overlays (a functional method for globally mapping changes to all packages in Nix). We submitted a pull request (PR) to nixpkgs to fix this issue for everyone. This task took much longer than expected, as we ended up resolving many other issues in the package that did not initially affect us. This effort required about a week of work.

MediaPipe

The last challenge we faced was with MediaPipe. MediaPipe is a machine learning library built with Bazel [5], a build system that, while supported in nixpkgs, is difficult to work with due to design conflicts with Nix. To integrate it into our C++ project, we had to convert MediaPipe into a shared library by first wrapping it in a C interface before packaging it in Nix. This task was particularly time-consuming and difficult, taking about a week of work.

4.3 Rendering System

4.3.1 Introduction

The rendering system is a key component of the Volumetric Simulator, responsible for displaying models in a manner that makes them appear three-dimensional. It needs to be fast and responsive to maintain the illusion of 3D.

4.3.2 OpenGL

Our project requires the rendering system to render 3D models in real-time with low latency and a high frame rate. We decided to use OpenGL [64] to achieve this due to its low-level control over the rendering pipeline and cross-platform compatibility. We briefly considered using fully fledged game engines like Unity [57] or Unreal Engine [17], but ultimately decided against them due to the significant overhead and lack of fine control over the rendering pipeline. Additionally, we utilized the OpenGL-compatible GLM [24] mathematical library for matrix manipulation and projections, as it was sufficiently fast and simple to use.

4.3.3 Perspective

As covered extensively in the background section, the user's perspective is crucial for creating the illusion of 3D. To achieve this, we used positions provided by the tracking system to calculate the correct dimensions for the viewing frustums, rendering the scene from the user's perspective. The results of this method, used to render a chessboard, can be seen in Fig. 4.3.1 and Fig. 4.3.2.

Figure 4.3.1: Right Perspective

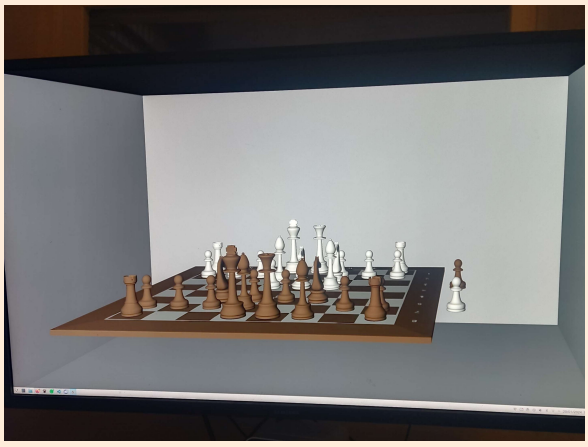
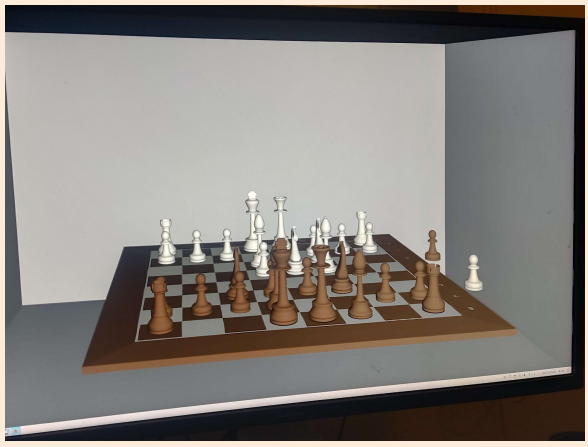


Figure 4.3.2: Left Perspective



4.3.4 Object Loading

Object loading support was added to the rendering system to facilitate the rendering of complex 3D models. We used the tinyobjloader library [56] to load `.obj` object files. We chose

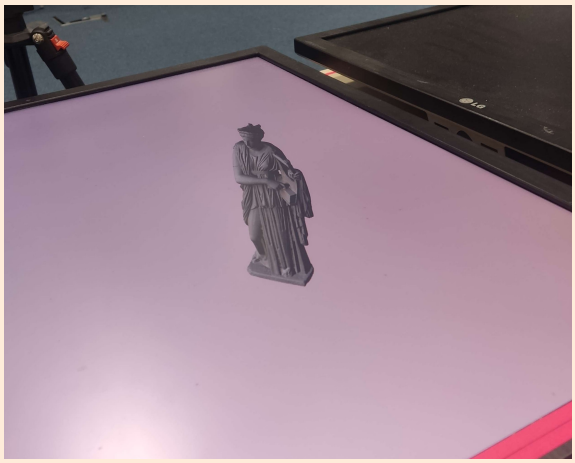
this library over alternatives like Assimp [3] due to its lightweight nature. We constructed our challenge for the user study by modifying primitives such as spheres, cylinders, and cubes loaded as `.obj` files using tinyobjloader to create an interactive task for the user.

4.3.5 Lighting

As shadows enhance the illusion of depth [34], we added a simple lighting model to the scene. We employed the Blinn-Phong [7] lighting model, which uses ambient, diffuse, and specular lighting. A comparison of the scene with and without lighting can be seen with the Erato Model [43] in Fig. 5.1.14 and Fig. 4.3.4, respectively.

Figure 4.3.3: Erato

Figure 4.3.4: Erato With Shadows



4.4 Tracking System

4.4.1 Introduction

To accurately simulate a volumetric display, it is essential to determine the positions of the user's face and hands, which allows for rendering the correct perspective for the user. To achieve this, our tracking system needed to meet the following requirements:

- **High resolution:** Precisely track the user's eyes and hands.
- **High framerate:** Ensure smooth tracking of the user's eyes and hands.
- **Low latency:** Provide near real-time tracking of the user's face and hands.

4.4.2 Hardware

Figure 4.4.1: Azure Kinect [4]

For this project, we used a Microsoft Azure Kinect camera (Fig 4.4.1). The Azure Kinect camera is equipped with two sensors: a depth sensor (utilizing an IR camera) and a color camera.

We configured the camera to capture images at its widest field of view (FoV) of $90° \times 74.3°$, with an exposure time of 12.8 ms and a framerate of 30 fps. To use this configuration, we compromised on the resolution of the color images, running the RGB camera at 2048×1536 instead of its maximum resolution of 4096×3072. Similarly, we used the depth camera at a 2×2 binned resolution of 512×512, instead of its maximum unbinned resolution of 1024×1024.

Because we utilized the depth camera in wide FoV mode ($120° \times 120°$), rather than the narrower FoV mode ($75° \times 65°$), the maximum operating range of the depth sensor was reduced to 2.88 m, compared to 5.46 m in the narrower FoV mode. This reduction in range was acceptable for our purposes, as the user was expected to be within 1.5 m of the camera.

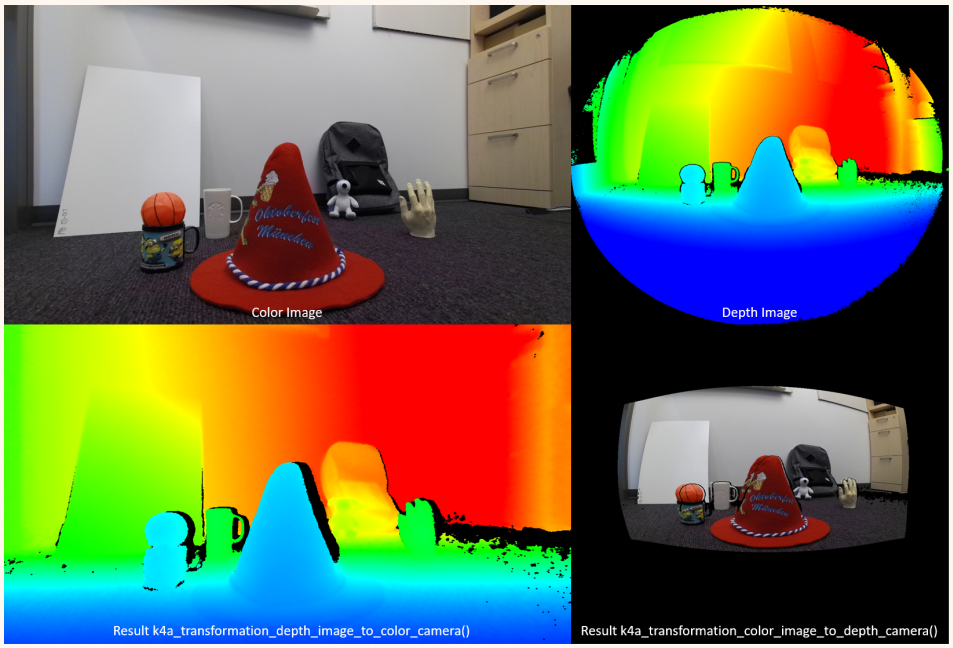
4.4.3 Core Libaries

Azure Kinect SDK (K4A)

We utilize the Azure Kinect SDK (K4A) [45] library to retrieve captures from the Kinect and handle the spatial transformations necessary to calculate the positions of points in 3D.

When the Kinect camera is polled using the K4A library, it returns a ”Capture,” which is a struct containing a color image, a depth image, and an IR image. It is important to note that the depth image is in a different coordinate space compared to the color image, as illustrated in Fig 4.4.2. This discrepancy arises because the color image and depth/IR image are captured using physically offset sensors, resulting in a slight variation in perspective. The depth image resides in what is known as ”depth space,” while the color image resides in ”color space.” To utilize these two images together, they must be converted to the same coordinate space.

Figure 4.4.2: Different Spaces: Depth and Color Images [59]



Using a ”calibration” that is generated at the start of the program, K4A allows conversion between the four different ”spaces”: ”Depth 2D,” ”Depth 3D,” ”Color 2D,” and ”Color 3D.” There are notable performance implications when using different spaces for various tasks. For instance, converting from ”Depth 2D” to ”Depth 3D” is significantly faster than converting from ”Color 2D” to ”Color 3D.”

An interesting side effect of converting between spaces is that, due to the physical offset and different diffraction characteristics of the IR and color cameras, ”depth shadows” [35] can occur, as visible in the bottom left image in Fig 4.4.2. These depth shadows can complicate tracking thin objects, such as fingers, because they increase the likelihood of encountering

invalid depth data. We explored using the IR image for tracking to mitigate these depth shadows; however, we found that tracking models struggled with IR images, leading us to continue using the original color image method.

OpenCV

We use OpenCV [8] to handle the images obtained from the Azure Kinect SDK. OpenCV is a comprehensive library that provides a wide range of functions for image processing and computer vision. We utilize OpenCV to convert the images from the Azure Kinect SDK into a format that can be efficiently processed by Dlib and MediaPipe. We leverage OpenCV's GPU/CUDA-accelerated functionalities, such as image pyramids for downscaling, to enhance the performance of our tracking system. Additionally, OpenCV is used for debugging purposes to render images to the screen.

Dlib

We use Dlib [36] for tracking the user's face. Dlib is a modern C++ toolkit that includes machine learning algorithms and tools for creating complex software solutions in C++. As discussed in greater detail later in this section, we use Dlib's face tracking model to track the user's eyes. We also utilize Dlib's GPU-accelerated functions to improve the performance of our tracking system.

MediaPipe

We use MediaPipe [41] for tracking the user's hands. MediaPipe is a cross-platform framework designed for building multimodal applied machine learning pipelines. We employ MediaPipe's hand tracking model on color images to track the user's hands in real-time. MediaPipe's hand tracking model is a deep learning model capable of accurately tracking hand movements in real-time.

4.4.4 Overall Tracking System Design

The primary goal of the tracking system is to convert the captures provided by the Kinect camera into 3D points that represent positional information, such as the positions of the user's eyes and fingers. The current system focuses on returning the position of the left eye and the tips of the index and middle fingers on the hand closest to the camera. The system operates smoothly at the same frame rate as the Kinect camera, which is 30 frames per second (fps).

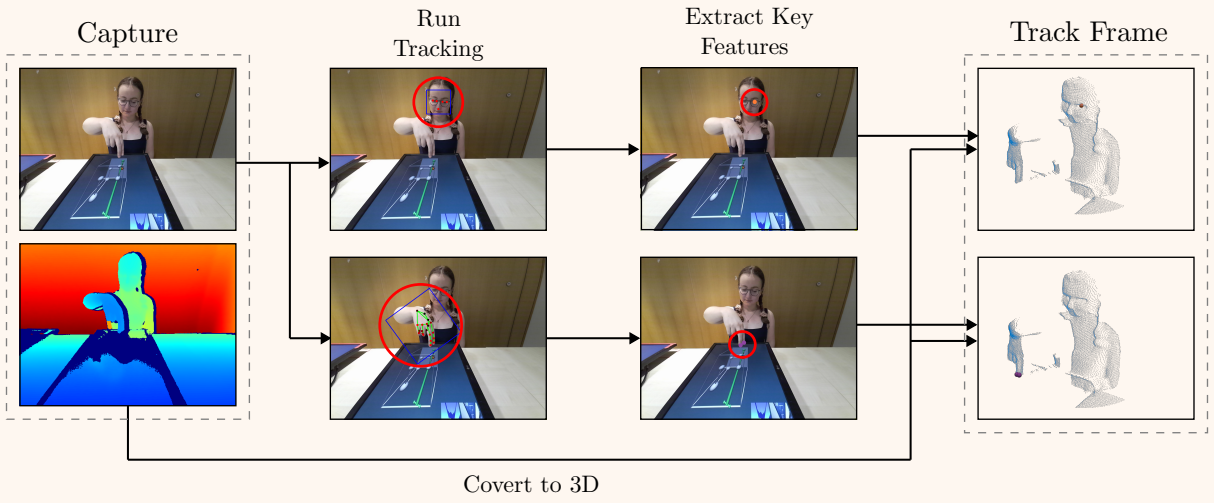
Several different approaches were considered for the tracking system design. Ultimately, we chose to use a method that tracks the positions of the user's face and hands separately using 2D color images. These images are then processed and converted into 3D points.

An alternative approach we considered was to track the face and hands directly in 3D. However, we decided against this method for several reasons. Although the rendering system is an important aspect of this project, it was not our primary focus, and we were concerned that tackling 3D tracking would be too complex given our time constraints. The ecosystem for

2D tracking is more mature, partly due to advancements driven by mobile phone technology, allowing us to leverage existing work, such as the Dlib, MediaPipe, and OpenCV libraries discussed previously. Additionally, we were concerned about the performance of 3D tracking. The increased data volume associated with 3D tracking could potentially slow down the system, making it difficult to achieve real-time tracking of the user's face and hands.

Despite opting for 2D tracking, there are several downsides to this approach. For example, as discussed further in the evaluation section, it can be challenging to accurately determine the positions of objects in 3D space that are occluded, such as fingers behind other fingers. Fingers, being relatively small objects, require sampling a general area to identify their position, and selecting the closest valid point. If only the predicted point is sampled, there is a risk of missing the finger and encountering a "depth shadow".

Figure 4.4.3: Tracker Overview



As illustrated in Fig 4.4.3, the process flow of the tracking system is as follows:

1. **Retrieve Capture:** The Kinect Camera is polled to receive a capture.
2. **Run Tracking:** Hand and head tracking are performed on the color image.
3. **Extract Key Features:** The positions of the tracked points are extracted from the hands (middle and index fingertips) and face (left eye).
4. **Convert to 3D:** The depth image is sampled to convert the 2D points into 3D points, which are then placed in a "tracking frame" to be sent to the renderer.

While it might seem unconventional, we maintain separate instances for tracking the left eye and the thumb and index fingertips. This approach is necessary because there is no guarantee that the tracker will detect both the face and hands in a single pass through. For instance, if the user holds their hand in front of their face, the tracker may only detect the hand. Additionally, there is a possibility that either of the tracking models may fail to detect the intended features. In such cases, our system will reuse the last known position and the corresponding

capture, which serves as a reasonable approximation for the eye or finger positions. This strategy also helps to reduce the jerkiness effect that can occur with sporadic dropouts.

It is important to note that, for performance reasons, we do not calculate the 3D positioning of the points until the renderer requests them. This approach minimizes resource wastage. If we can process a capture faster than the renderer can render a frame, we only calculate the 3D positioning for the most recent capture.

4.4.5 Tracking Models

Dlib

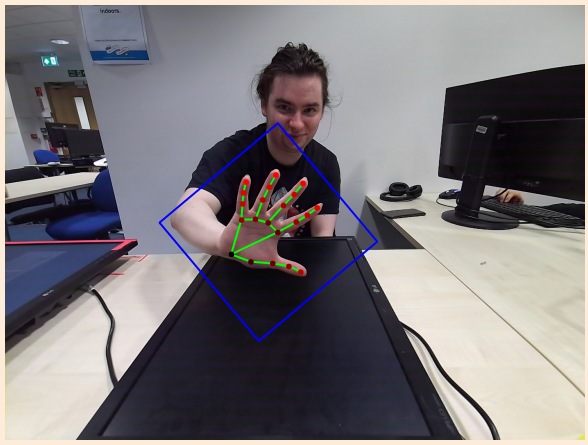
We use Dlib to track the left eye's position through a two-stage process. In the first stage, we utilize a Max-Margin Object Detection model [37, 12] implemented with a convolutional neural network (CNN) [53]. Instead of training our own model, we used a pre-trained model provided by Dlib [13], known as `mmod_human_face_detector`. Initially, we ran the face detection model on the CPU; however, this created a bottleneck in our tracking pipeline. Therefore, we opted to run our CNN on a GPU using CUDA-accelerated functions to meet our performance requirements.

Once the user's face is detected, we proceed to the second stage of our facial landmark detection model. We chose to use a five-landmark pose estimator called `shape_predictor_5_face_landmark`, leveraging Dlib's implementation of a method proposed in "One Millisecond Face Alignment with an Ensemble of Regression Trees" [32], trained on the iBUG 300-W face landmark dataset [52]. This pose estimator operates efficiently enough on a CPU, so GPU acceleration was unnecessary.

An example of the results obtained from running the two Dlib models can be seen in Fig 4.4.4.

Figure 4.4.4: Dlib Face Tracker

Figure 4.4.5: MediaPipe Hand Tracker



MediaPipe

We use MediaPipe to track the position of two fingers on the user's hand, employing a two-stage process. MediaPipe uses a two-stage model to track hands [66]. The first stage involves a palm detection model that identifies the position of the hand within the image. The second stage is a hand landmark model that detects the positions of 21 points on the hand. MediaPipe provides an interface that allows us to feed images as a stream, abstracting away most of the detection logic, unlike Dlib.

We chose to track the positions of the index and middle fingers because this configuration proved to be more stable than tracking the thumb and index finger, and it led to fewer instances of accidental occlusion. To enhance the tracking accuracy, we depth-sample from the surface of the hand and apply a constant offset to make it appear as though the point is inside the hand. An example of the results from running the two MediaPipe models can be seen in Fig 4.4.5.

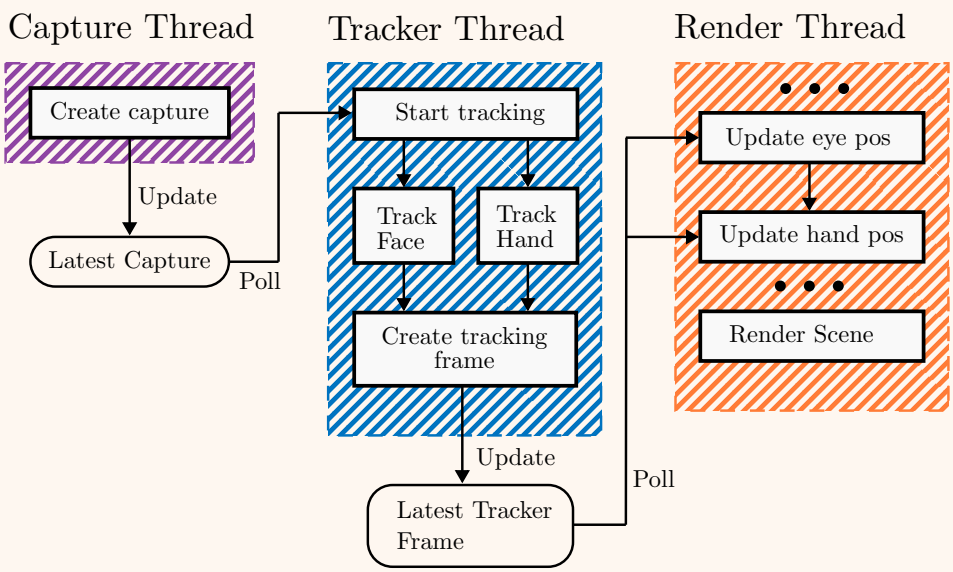
Downscaling

To ensure that our tracking system operates at a sufficient frame rate, we downscale the images obtained from the camera. We found that downscaling the images using an image pyramid [2] by a factor of 2 still yields accurate tracking results. This significantly improves the performance of our tracking system. Further details can be found in the evaluation section.

4.4.6 Multithreading

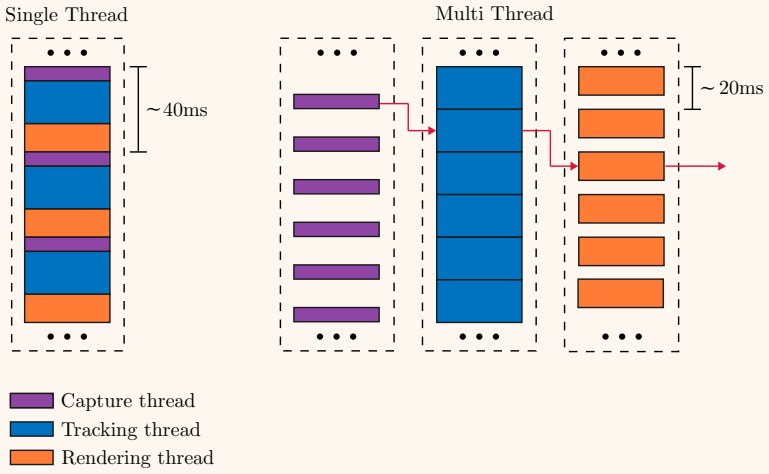
One of the more challenging aspects of this project was making our tracking system performant enough to feel smooth and responsive. To achieve this, we implemented a multi-threaded design, as outlined in Fig 4.4.6. We used separate threads for tracking, capturing, and rendering.

Figure 4.4.6: Multi-threaded Design



The purpose of this design was to ensure that the tracking thread and models were utilized 100% of the time, as they are the most computationally intensive components of the system and represent the main bottleneck. While using a multithreaded design does not reduce the system's latency (as discussed further in the evaluation section), it significantly increases the application's throughput and frame rate.

Figure 4.4.7: Single vs Multi-threaded Design



Since the Kinect camera operates at 30 fps, we need to process an image every $\frac{1000\text{ ms}}{30} = 33.3$ ms to ensure that we handle every frame. In our initial single-threaded implementation, we were unable to achieve this rate. By switching to a multi-threaded design, we decreased the time required to produce a new tracking frame to match the duration of the slowest thread (the tracker thread), as shown in Fig 4.4.7. This also allowed us to run the simulation at a

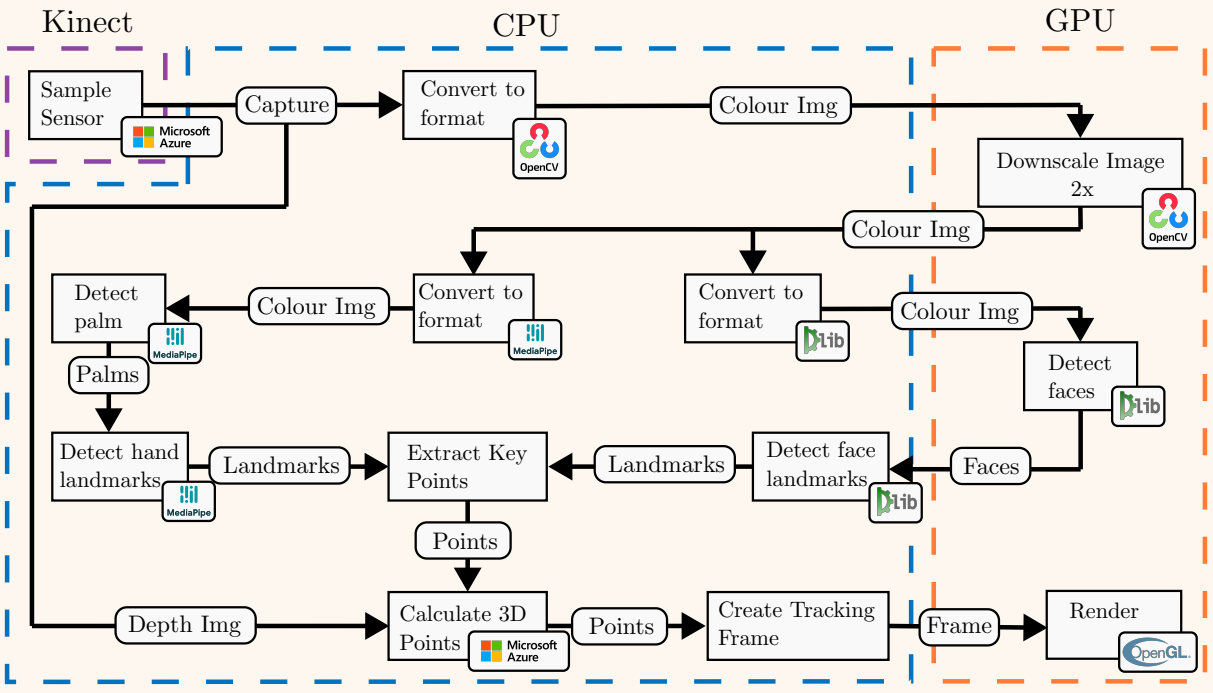
frame rate independent of the tracker. Although this design resulted in the capture thread often being idle, the overall system was light on resources, making this trade-off worthwhile for the significant frame rate improvement it provided.

4.4.7 GPU Acceleration

Another method we utilized to enhance the performance of our tracking system is GPU/CUDA acceleration. Both Dlib and MediaPipe support GPU acceleration. However, we only needed to use GPU acceleration in Dlib because the CPU speed was already sufficient for our tracking pipeline in MediaPipe. The reported speedup of 12.27 ms with GPU acceleration versus 17.12 ms without [27] did not justify the effort required to enable CUDA in MediaPipe, especially given the complexities involved in building with Nix (see the build systems section for more information).

As illustrated in Fig 4.4.8, we only used GPU acceleration for two parts of our tracking system, excluding rendering. We utilized OpenCV's GPU-accelerated pyramid down function to downscale our color images, as this task is highly parallel and benefits significantly from acceleration. Additionally, we executed the Dlib CNN for face detection on the GPU.

Figure 4.4.8: Overall Tracking System Design



4.4.8 Camera Positioning

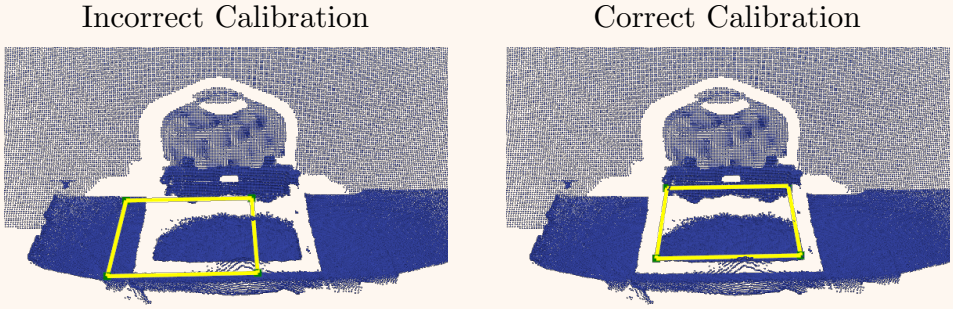
To ensure that the tracking system is calibrated correctly and that the user sees the correct perspective, it is crucial to know the relative position of the camera to the screen. Misalignment can lead to a distorted or incorrect user experience, where objects may appear to be

in the wrong location or orientation relative to the user's point of view. To achieve this, the orientation of the camera and the dimensions of the screen must be accurately determined. We developed a calibration system to expedite the process of determining accurate position and orientation values. The calibration system operates as follows:

1. The camera's position and orientation are measured in 3D space, and the screen's position is also measured in 3D space.
2. The relative positions of the camera and screen are input into the system.
3. The predicted position of the screen is rendered in 3D.
4. The position of the screen is iteratively adjusted until the rendered position aligns with the actual position.

An example of correct and incorrect calibration can be seen in Fig 4.4.9.

Figure 4.4.9: Display Calibration



4.5 User Study

4.5.1 Introduction

To validate the effectiveness of our system, we conducted a Within-Subjects User Study. The study was designed to demonstrate the system's capability for research applications. We aimed to select a study that would both showcase the system's capabilities and contribute novel insights to the field. We settled on a study to test user performance with volumetric displays under two different conditions as outlined below.

4.5.2 Experimental Variables

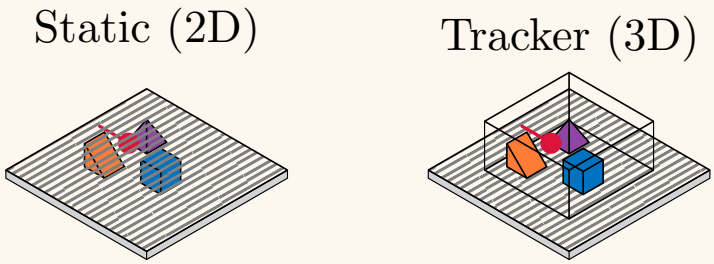
We aimed to test the following two hypotheses:

- **H1**: Is there a difference in task performance when interacting with the volumetric display in 3D as opposed to 2D?
- **H2**: Is there a difference in task performance when interacting with the volumetric display directly with hands as opposed to via teleoperation?

To test these hypotheses, we designed a 2x2 within-subjects experiment. Our two independent variables were:

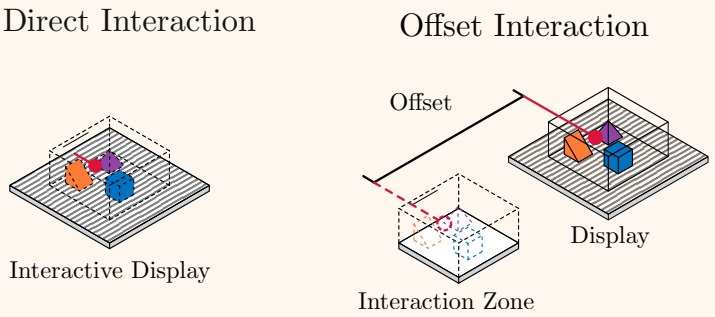
- **Perspective:** `Static` vs. `Tracker`. This variable controls whether the system uses a tracking mechanism to create the illusion of a 3D volumetric display (`Tracker`) or a fixed perspective on a standard monitor (`Static`), as shown in Fig 4.5.1. This tests **H1**.

Figure 4.5.1: 2D & 3D



- **Interaction Offset:** Offset vs. No Offset. This variable controls whether the display is directly in front of the participant or offset by a fixed amount, as shown in Fig 4.5.2. This tests **H2**.

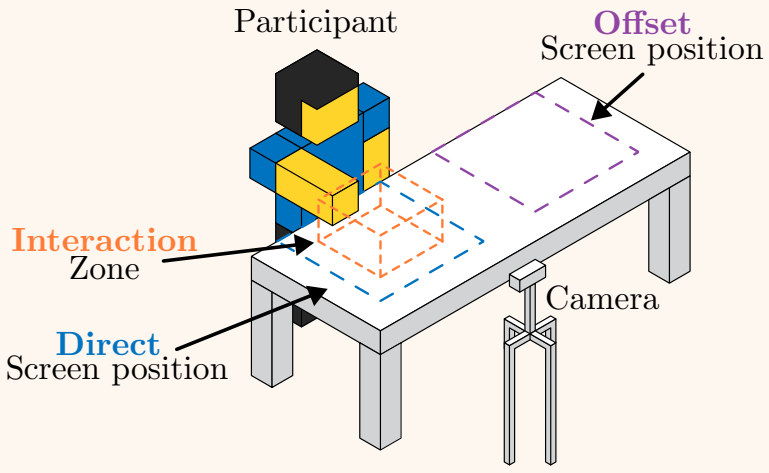
Figure 4.5.2: Direct & Offset Interaction



We controlled for the following variables during the study:

- **Tasks:** We ensured that the five tasks were identical in each condition.
- **Device Calibration:** The positions of the participant, the tracking camera, and the interaction zone were kept consistent across conditions, as shown in Fig 4.5.3. When using an offset position, another display was placed where the original display was to maintain consistent tracking.
- **Lighting:** We maintained consistent lighting conditions in the room, as lighting significantly impacts the system's tracking quality.

Figure 4.5.3: Test Setup



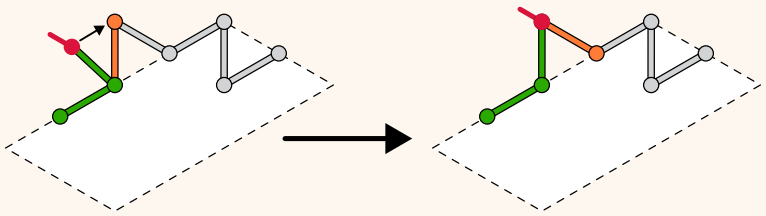
4.5.3 Tasks

In each of the four combinations of conditions, participants must complete the same five tasks in the same order. The tasks are designed to be simple to understand but challenging to complete. They were crafted so that, from any perspective, the components would visually overlap.

To complete a task, participants must trace the path between points using their index and middle fingers in the order presented by the simulator. A green point indicates a completed

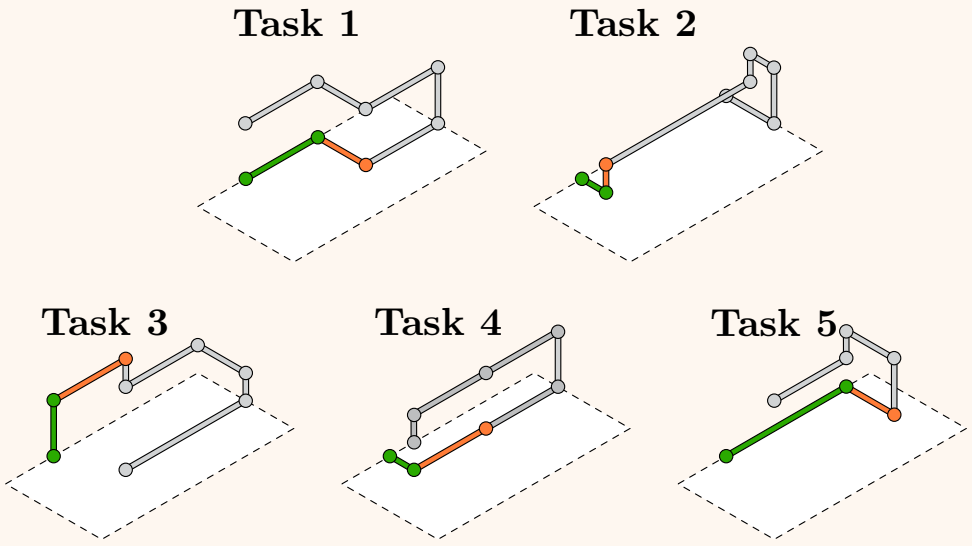
segment, an orange point represents the next segment to be completed, and a red point shows the current position of the participant's hand. Each time a task segment is completed, the colors update accordingly. An example of a completed task can be seen in Fig 4.5.4.

Figure 4.5.4: Completing a Task



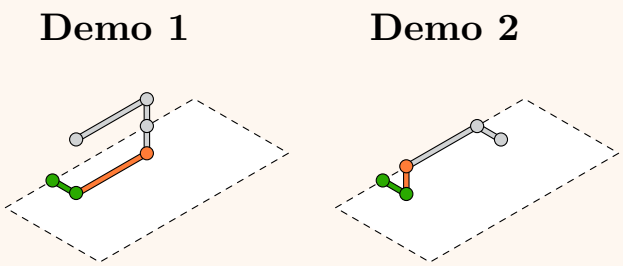
Participants have a time limit of one minute to complete each task. The time at which each point is completed, as well as the position of the hand and eye throughout the task, is recorded. If participants do not complete the task within the time limit, the task is marked as incomplete, and all completed segments are logged. The five different tasks are shown in Fig 4.5.5.

Figure 4.5.5: The Five Tasks



Participants are also given two demonstration tasks to familiarize themselves with the system, as seen in Fig 4.5.6.

Figure 4.5.6: Demo Conditions



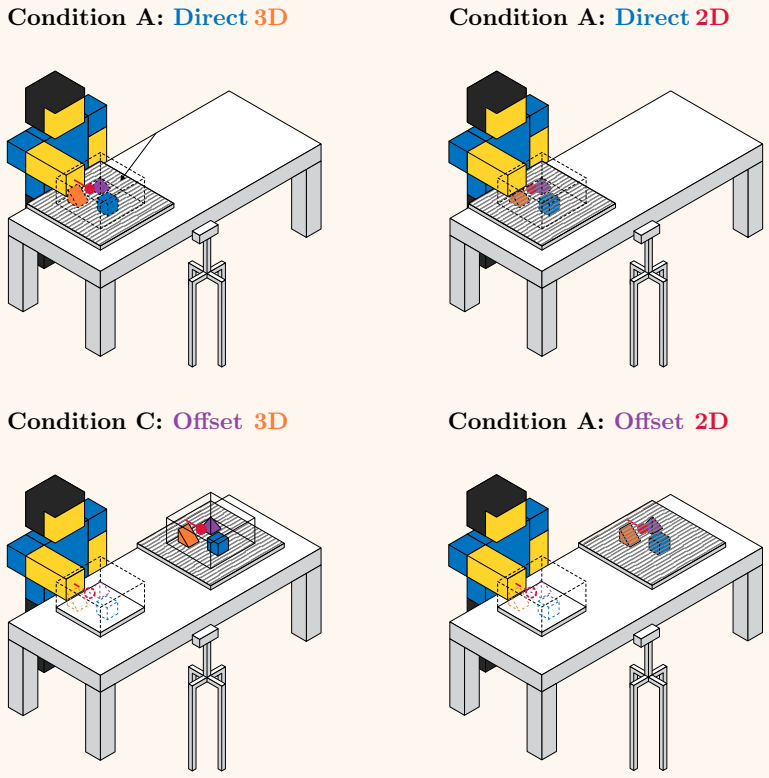
4.5.4 Participants

Participants were recruited for the study via email and text message using a standardized script to ensure the process was unbiased. Upon arrival, participants were required to fill out a consent form and complete the first page of a questionnaire. This initial questionnaire collected demographic and personal information that could influence the study results, such as age, handedness, previous experience with VR/AR, and whether they wore glasses (the full questionnaire is provided in the appendix).

Following this, participants received a brief overview of the system and had the opportunity to run through two demo tasks in each of the four different configurations to familiarize themselves with the system. Participants were instructed to keep their non-dominant hand on their lap and place their dominant hand face down on the monitor at the start of each task to facilitate tracking. They were permitted to repeat the demo tasks up to three times if necessary.

Once the participants felt comfortable with the system, they were entered into the study's system and were automatically assigned a random sequence of the four experimental conditions, as shown in Figure 4.5.7.

Figure 4.5.7: Study Conditions



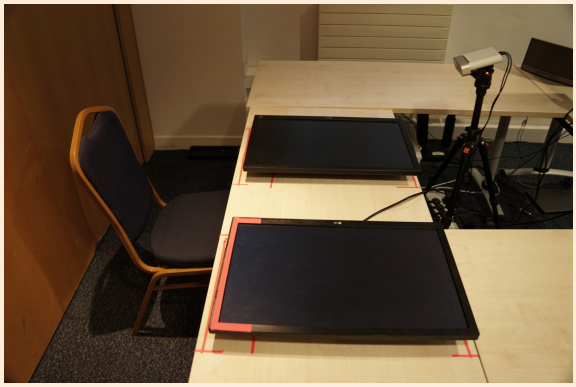
Each condition comprised five tasks, each lasting one minute, which participants were required to complete sequentially. An audio cue signaled the start and end of each task. Participants were allowed to take breaks between conditions. After completing each condition, they filled out a survey regarding that condition. At the end of the study, they completed a survey about the overall system. Both surveys are included in the appendix.

4.5.5 Setup

The study was conducted on the ground floor of the Huxley building in Room 218 at Imperial College London's South Kensington Campus. Considering that the study would span multiple days, we took special care to set up the system to minimize the risk of accidental changes to the setup. As depicted in Fig 4.5.8, the camera was mounted on a tripod, and the positions of the tripod legs were marked on the floor to maintain consistency. Additionally, we surrounded the camera with a barrier of tables to prevent it from being knocked over.

Figure 4.5.8: Study: Front View

Figure 4.5.9: Study: Side View



The displays used for the study were $24''$ 1920×1200 `LG IPS LED 24EB23` computer monitors, which were detached from their stands and placed horizontally on a table. There was a 25 cm gap between the bottom of the farther monitor and the top of the closer monitor, as shown in Fig 4.5.9.
The camera was positioned such that the user's head was approximately 1 meter away, although this distance varied slightly with participant height. The interaction zone on the far display was set up such that participants interacted with the scene at distances ranging from 30 to 70 cm from the camera. These distances were selected to fall within the optimal tracking range of our system (discussed in more detail in the evaluation section).
To facilitate the study and monitor participants effectively, we set up two additional monitors opposite the participants, as shown in Fig 4.5.10. These monitors allowed the study conductor to control the system and observe the participants. Using large monitors helped to block the view of the study conductor, thereby reducing the possibility of participants feeling observed, which might introduce unintended bias.

Figure 4.5.10: Study: Control View

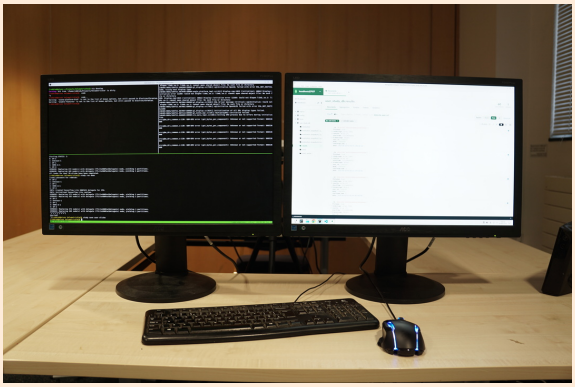
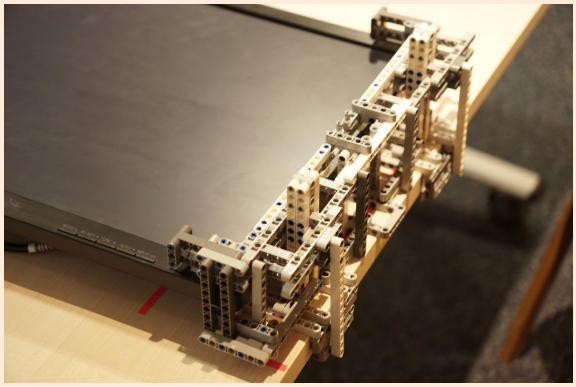


Figure 4.5.11: Study: Calibration Device



We also designed and built a calibration device to realign the displays after each participant completed their tasks, as depicted in Fig 4.5.11. Participants often inadvertently moved the displays during the tasks, and this simple device, made from Lego Technic (A line of Lego inter-

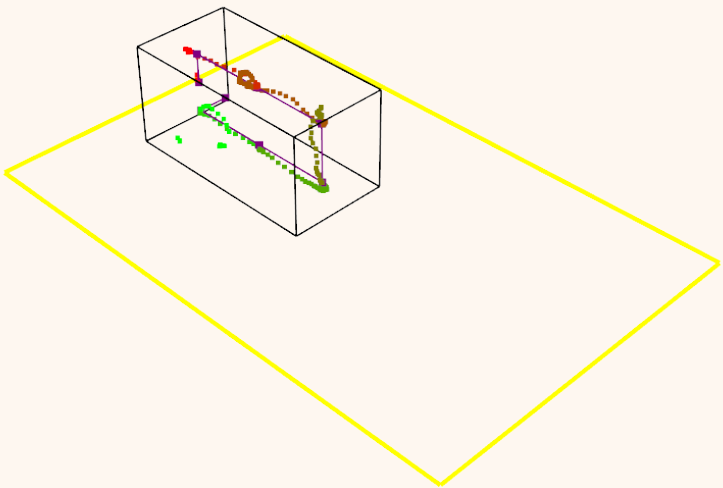
connecting plastic rods and parts, which is useful for creating a variety of mechanical systems), enabled us to easily and accurately reposition the displays to their original alignment.

4.5.6 Evaluation Metrics and Collected Data

The evaluation of the system was based on two primary metrics: the time taken to complete each task and the number of subtasks completed within the designated time frame. For precise measurement, a time stamp was recorded at the beginning of each task and at the completion of each subtask. This allowed us to accurately track task duration and identify any performance patterns. Additionally, we monitored and recorded instances of tracking failures to ensure that we could filter out erroneous data during the analysis phase, thus maintaining the integrity of our results.

Throughout each task, we continuously logged the positions of the participants' eyes, middle fingers, and index fingers, along with corresponding time stamps. This comprehensive data collection enabled a detailed analysis of participant movements and interactions. Specifically, the logged data allowed us to plot the paths taken by participants, which provided valuable insights into their interaction patterns. An example of such a plot is shown in Fig 4.5.12, illustrating the trajectory of a user's finger movements during Task 4.

Figure 4.5.12: Example of logging a user's path for Task 4



To ensure comprehensive evaluation, we also collected additional data points such as error rates, which included the frequency and types of errors participants made during the tasks. This data provided deeper insights into the usability and reliability of the system. Moreover, participant feedback was collected through post-task and post-condition surveys, offering qualitative data that complemented the quantitative metrics. This holistic approach ensured that we could thoroughly assess both the performance and user experience aspects of the system.

4.5.7 Study Implementation

The study was run from a python based CLI using the Click [47] library. To make the study experience as seamless as possible, we designed a user-friendly CLI interface that guided study runner through the study process. The CLI would automatically start the next task with a simple one line command as can be seen in List 4.5.13.

Listing 4.5.13: Terminal

```
[VolumetricSim]$ study run next {USER_ID}
```

Chapter 5

Evaluation

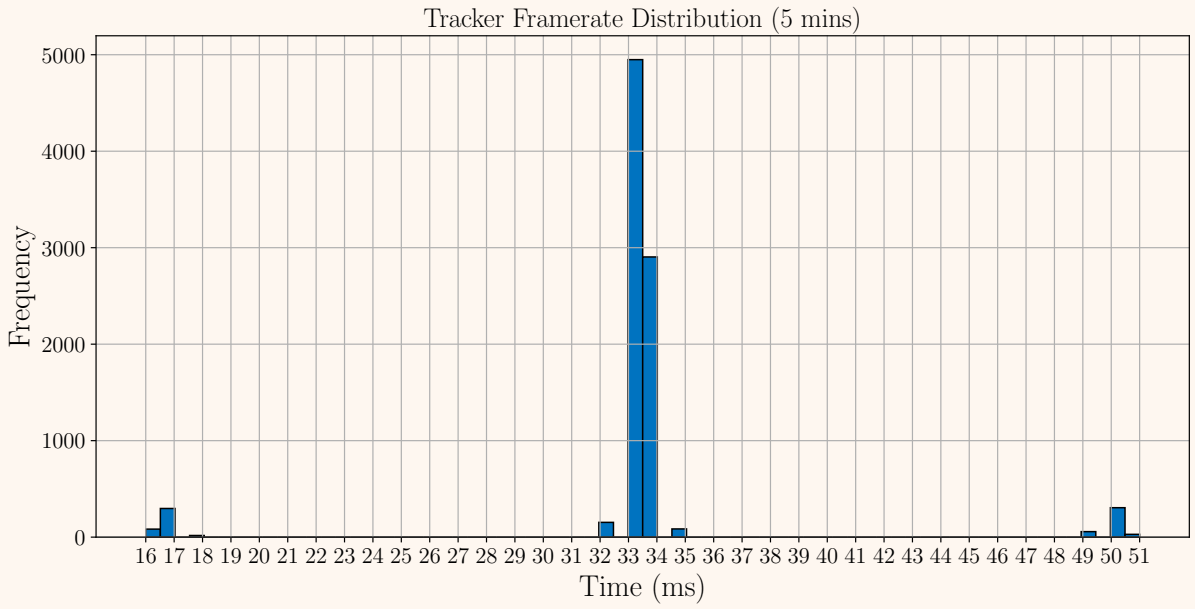
5.1 Simulator Evaluation

5.1.1 Overall System

1. Get benchmark of the system i.e GPU usage and CPU usage.

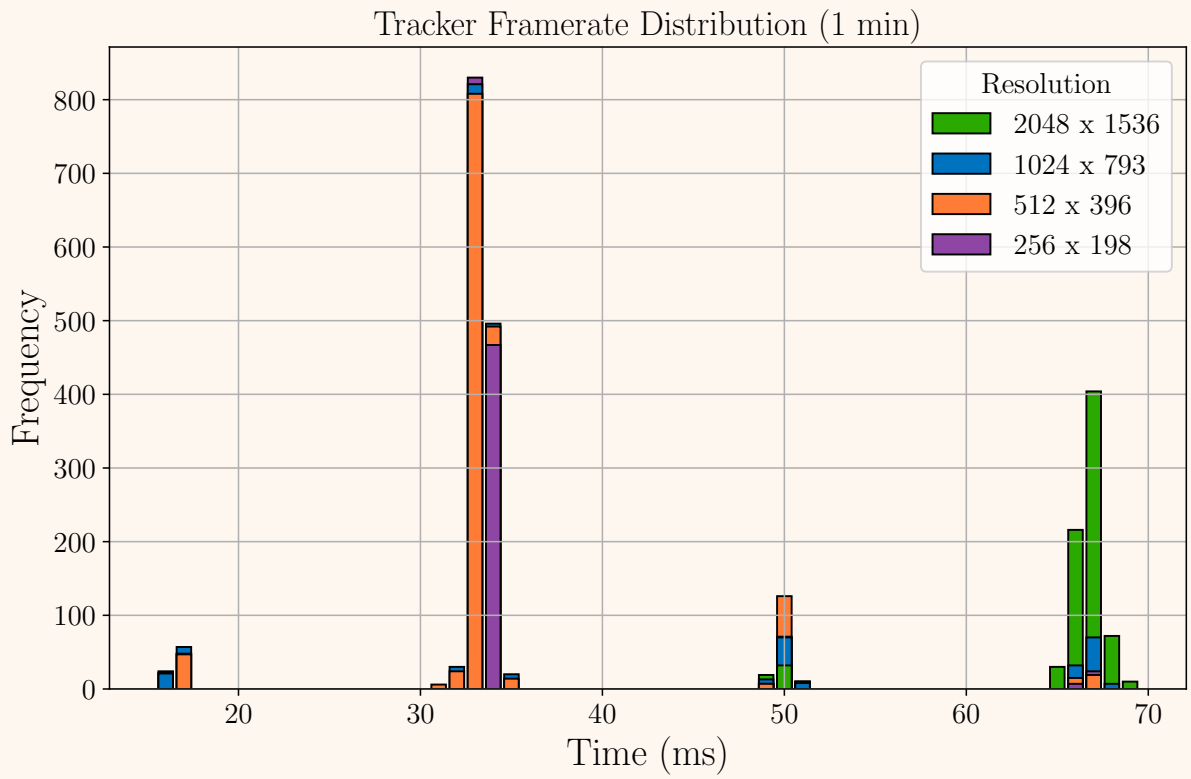
5.1.2 Tracking System: Frame Rate, Latency and Timings

Figure 5.1.1: Overall Framerate



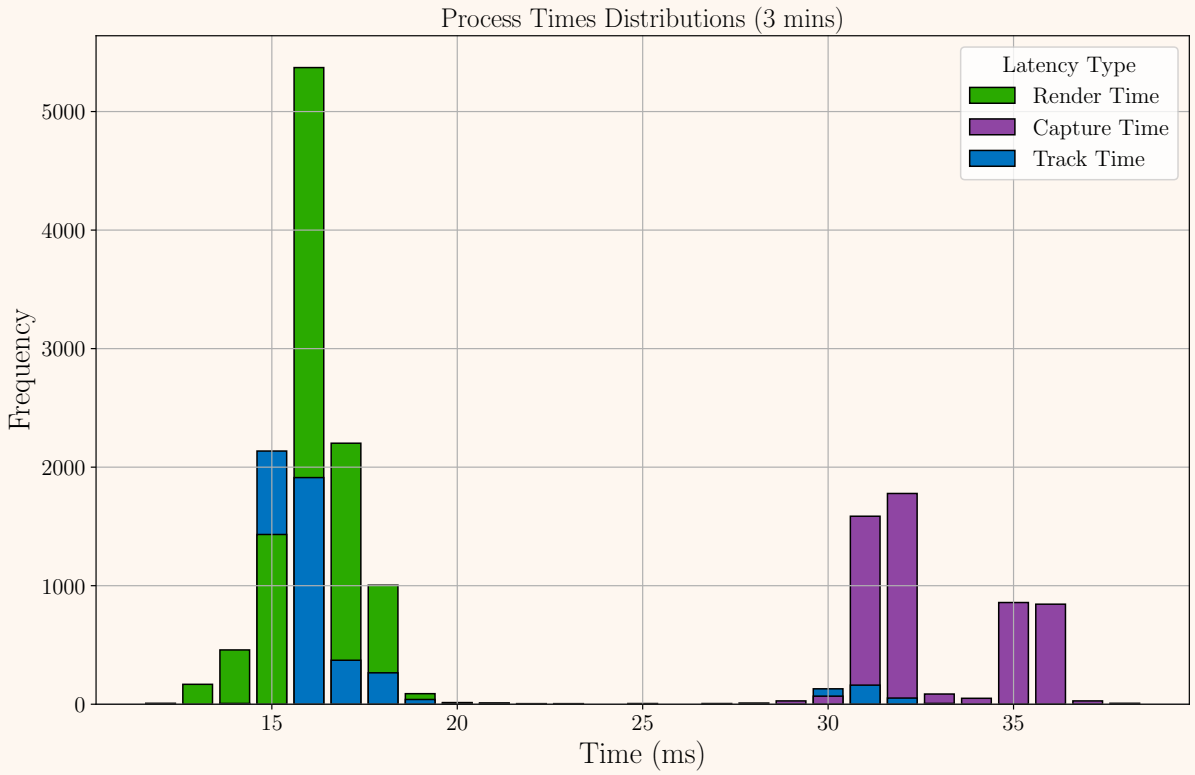
As can be seen in Figure 5.1.1, the framerate of the system is quite low. 90.83% of the time the framerate is between 30 and 35ms. There is what at first glance a strange phenomenon of a group of latencies at 16-18ms (4.46%) and 49-51ms (4.39%). This is due to the framerate of the renderer, as it lazily converted on request it basically chunked to the nearest 1000/60 = 16.666. Even though the tracker is running at 30fps, it is possible to have a latency of 16ms if the tracker fails to the meet the 30fps requirement for a frame buffering the result for the next frame. This means the next capture is already ready by the next frame so it makes it appear like the camera has a higher framerate than 30. rewrite this in a less shit way
We investigated down scaling the image to increase performance. We only found that down scaling it once was enough to get the performance we needed. Down scaling it further did not offer any significant performance increases and significantly decreased tracking quality. As can be seen in Figure 5.1.2, the performance of the system increases noticeably between going from 2048 x 1536 to 1024 x 793 but reducing the resolution further does not provide any speed benefit at all as we are already running at 30fps which is the cameras refresh rate.

Figure 5.1.2: Comparing Resolutions



As can be seen in Figure 5.1.3, of the three separate threads (captures from the Kinect camera, tracking the hand and eye, and rendering with OpenGL) which are running at the same time, the bottleneck is waiting for the inect camera to return the capture. This means there isn't any benefit to speeding up the tracking algorithm.

Figure 5.1.3: Comparing Processing Times



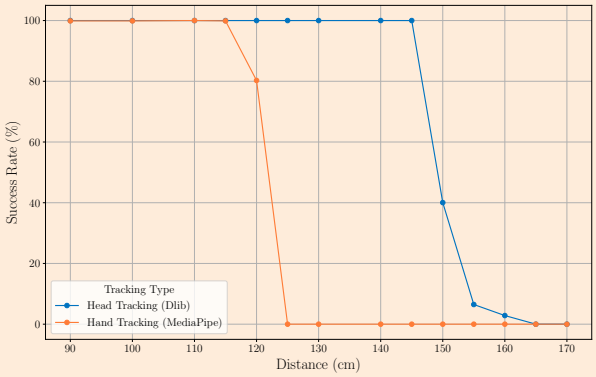
Estimate Latency, 12.8 exposure which you should cite, plus workout how the framerate effects it + 15 ms for tracking plus work out how the random chance of being selected during rendering works and compare to it similar systems like oculus quest and maybe fishbowl and vision pro?

5.1.3 Tracking System: Accuracy

We measured the effective tracking range of our system. The method we used for this was to sit in a chair and slowly wave your hand and move you your head side to side over a period of 30 seconds as can be seen in Fig 5.1.4. We took a variety of samples at different distances measuring the percentage of captures that were successfully able to detect a face or hand. The resolution of the colour camera was 1024 x 793.

Figure 5.1.4: Setup for distance testing

Figure 5.1.5: Tracking success rate at different distances



As can be seen in our results for Fig 5.1.5 the success rate for MediaPipes hand tracking model rapidly deteriorates after about 1 meter to the point of being completely unable. We suspect this is probably due to the data it was trained on and the fact it was primarily designed as a model to track hands from a mobile phone camera ToCite .

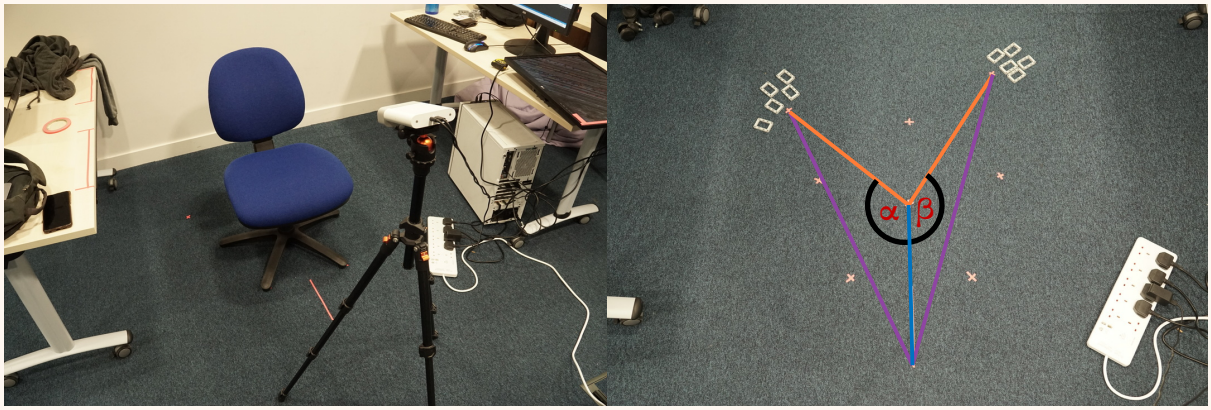
Dlib's headtracking model was better deteriorating at about 1.5 meters for suspected similar reasons to MediaPipes. We took these results into consideration while designing our user study positioning the camera so the users hand would be in the range of FIND OUT STUDY DISTANCE RANGE

Head Tracking

1. Test with glasses on?

One of the important aspects of the system is the head tracking. We wanted to evaluate how well the head tracking system worked. One important aspect of the head tracking system is the angle of the head at which it fails to detect a face. We create a fairly redumentary setup as can be seen in Fig 5.1.6 to measure the angle of the head at which the head tracking system fails. We used a swivel chair to rotate the user's body and head (we asked the participants to keep their body still and rigid). We measured the location of the midpoint of their feet when the tracking finished. Using GCSE trigonometry we were able to calculate the angle of the head at which the tracking failed.

Figure 5.1.6: Angle Setup



We tested this with 5 different people and the results can be seen in Fig 5.2.4. The technique was not particularly precise so the results should be taken with a grain of salt. The angle at which the head tracking failed was typically between $120° - 150°$. This might result might see strange as at that angle the participant is very much facing away from the camera however the way the face tracking model works is it first detects a face and then maps a landmark to the face.

Figure 5.1.7: Angle of failure for headtracking

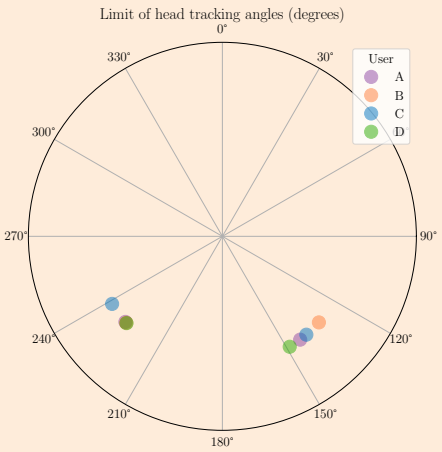
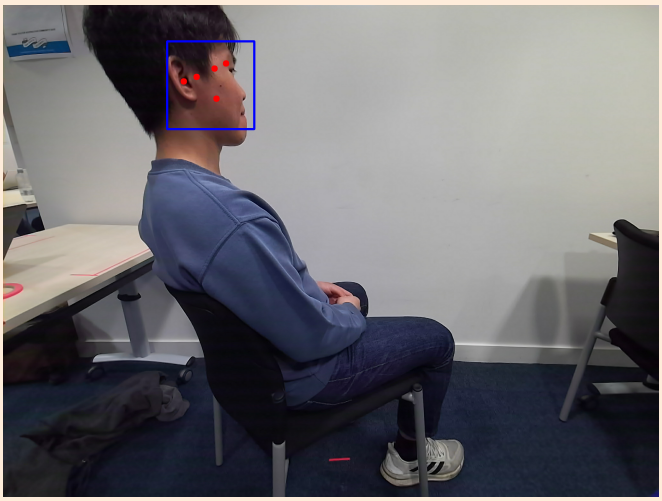


Figure 5.1.8: Incorrect Head Tracking



As can be seen in Fig 5.2.5 the face tracking model can still detect a face even when the face is not visible as it is detecting the side of this participants head. Acadotally the model seems to fail when it can only see hair. We would be interested to test this on a bald participant so see if it always detects a head no matter the rotation. Although this might seem like a problem at first glance it isn't. If the tracker cannot see the face then the face can't see the display either, so it doesn't matter if the display is rendered incorrectly. It also has the added benefit of tracking the eyes in an approximate position to where they would be if the face was

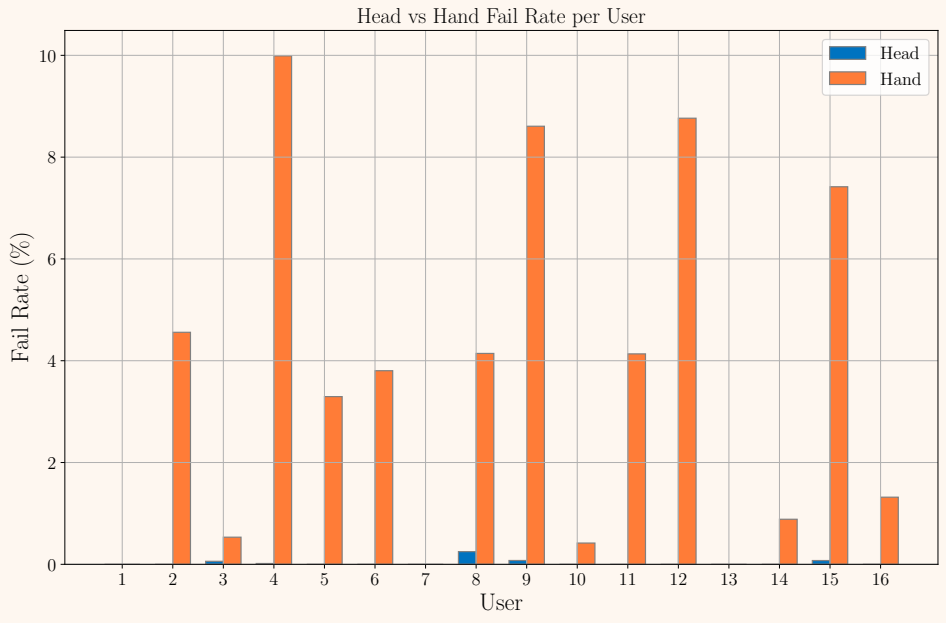
visible, so as the user turns their head and the face comes into view the display does not need to correct much and the user will not notice the correction.

Hand tracking

We found the hand tracker model was significantly less reliable than the head tracking model as can be seen in Fig 5.1.9. During our user study the hand tracking model failed for some users up to 10% of the time, usually when it was first trying to find their hand (we didn't record this time in our final results). We found it was very particular with the conditions it would work under. It required a well-lit room and a plain background and required us to remove all objects out of the scene. We found people wearing t-shirts with intricate patterns would massively reduce the success rate so we had a plain spare jumper to give users to wear if we noticed this being an issue.

We think we might have been able to fix this issue by using a segmentation model to segment the hand from the background first before trying to track it however we did not have time to experiment with this solution and we were worried it might significantly negatively impact the latency performance of the system.

Figure 5.1.9: Failrates during user study



One of the big issues with using a depth sampling method is that it cannot deal with occlusion. Even though Mediapipe is capable of predicting the position of points of the hand when it can't see them we can't sample their 3D positions accurately. This is a big issue as the hand is often occluded by itself as can be seen in Fig 5.1.10. Here the fingers are occluded by the back of the palm meaning when we try and sample the depth we instead sample the depth of the palm in front. This also at first thought sounds like it might be an issue for glasses wearers,

however, we found that the depth sample (it uses an IR method) was able to pass through glass.

Figure 5.1.10: Occlusion Sampling

There are methods for getting around this such as sampling points you know are not occluded and then using the depth of the hand to infer the depth of the occluded points. This is a fairly complex problem and we did not have time to implement it in a way that was both accurate and fast as mediapipes estimated depth is not that good. We think the best solution would just be to switch to a point cloud based tracking model. maybe find a paper to cite about hand occlusion

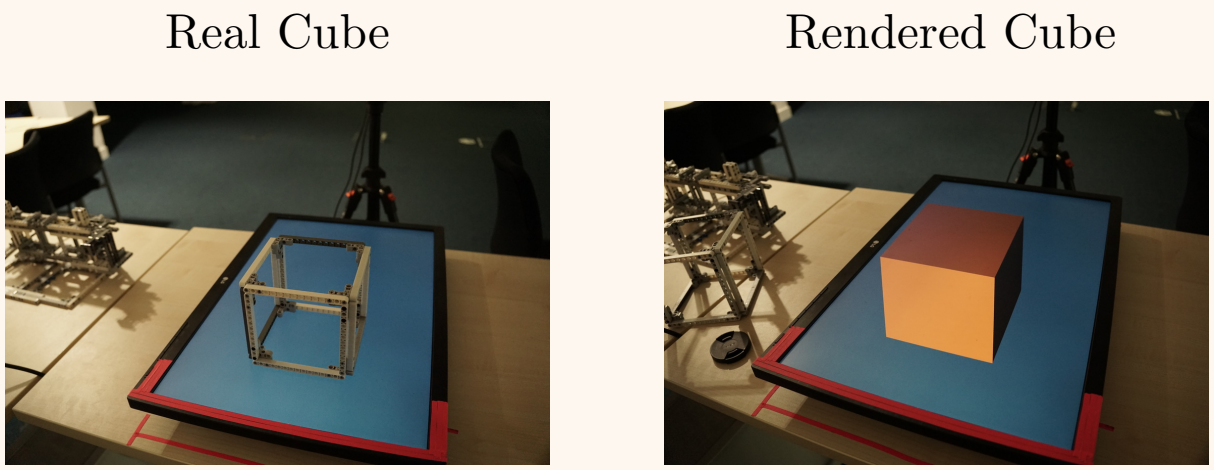
5.1.4 Renderer

TODO

1. Talk about and give an image of the rendering system displaying a complex scene.
2. Get framerate?

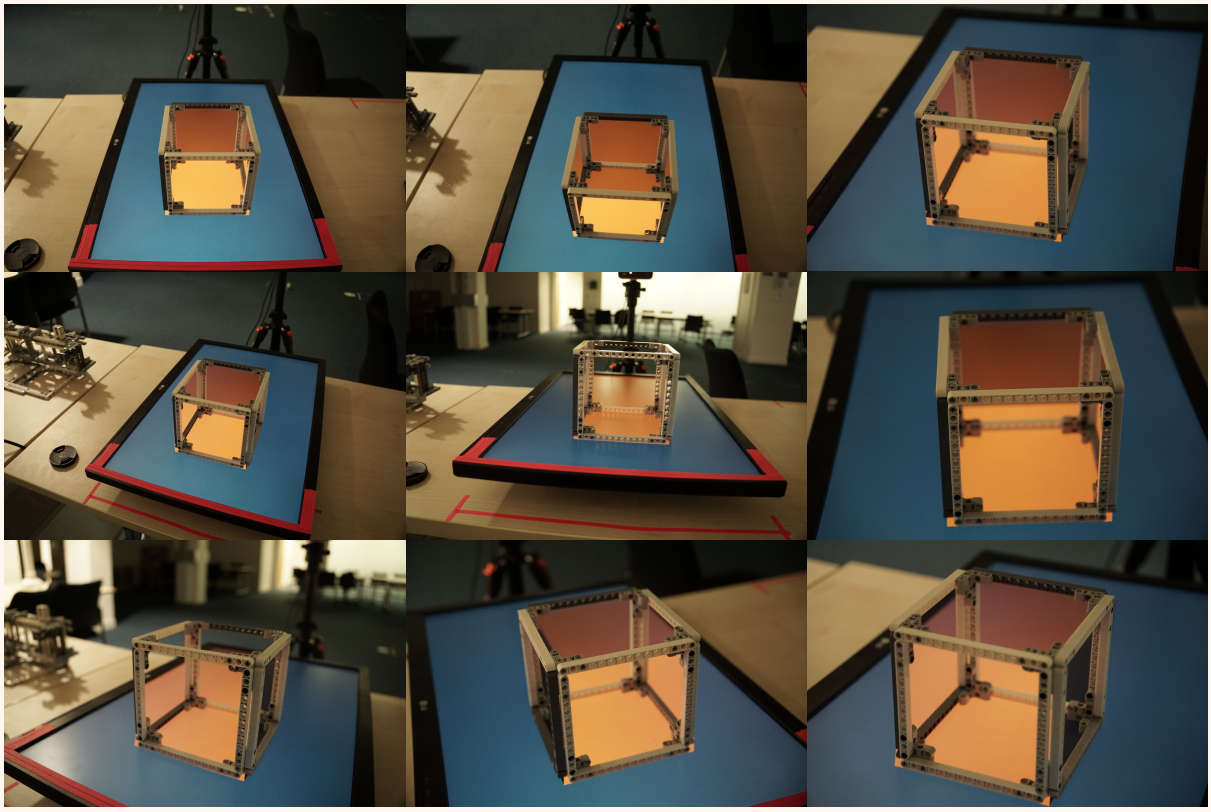
We wanted to evaluate how accurate our rendering system. As our rendering system propose is to create the illusion of 3D objects in the real world, we wanted to see how well it could recreate a real object. We chose a cube as it is a simple object that is easy to recreate. We created a physical cube out of lego and created a virtual cube in our simulator both with the same dimensions of 13.5cm x 13.5cm x 12cm as can be seen in Fig 5.1.11.

Figure 5.1.11: A real and rendered cube



Because the physical cube is hollow this allowed us to easily to compare the two cubes by superimposing them. We tested a variety of different perspectives and verified that the two cubes were indeed the same size and shape and completely overlapped. Some examples can be seen in Fig 5.1.12, it is worth noticing that the images might be slightly off as the system was tracking my eye and not the camera lense which was both below and about 10cm in front of my eye. This shows that our rendering system is accurate and can be used to create the illusion of 3D objects in the real world.

Figure 5.1.12: Superimposed real and rendered cube



5.1.5 Render quality

TODO cite: rendering objects from this repo: https://casual-effects.com/data/index.html cite where they are all from and why I picked them (e.g rending a protein)

Figure 5.1.13: Chess Set

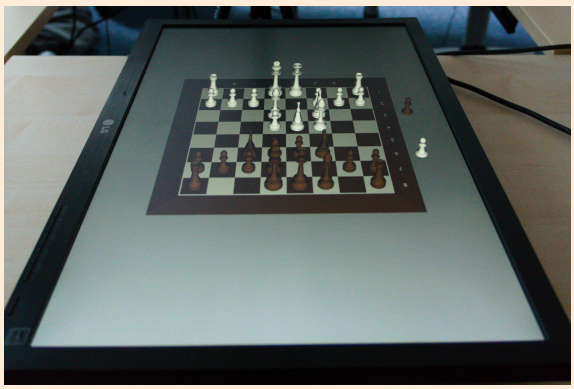


Figure 5.1.14: Erato

Figure 5.1.15: Minecraft House

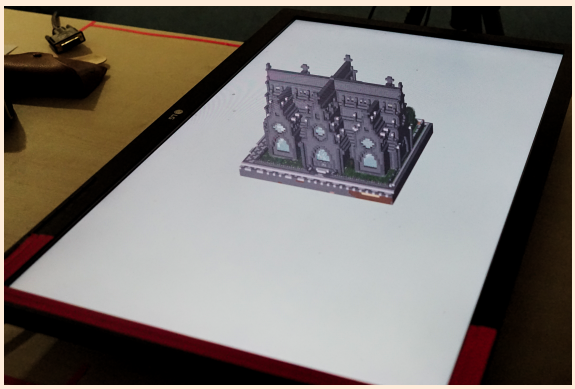
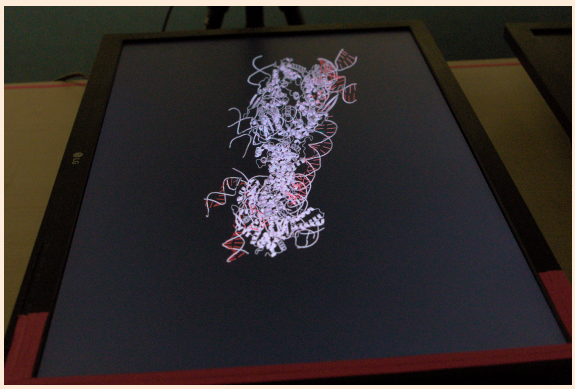
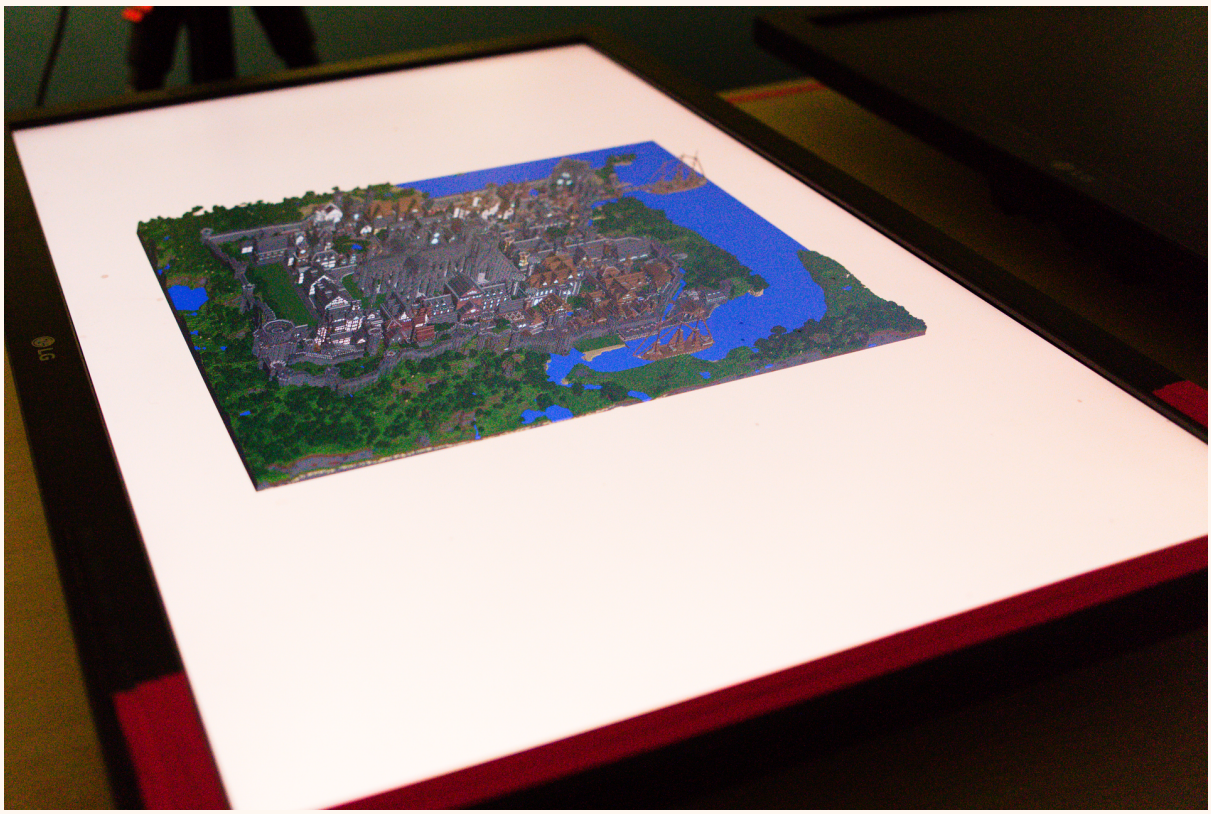


Figure 5.1.16: Retron-Eco1 filament with ADP-ribosylated Effector



Talk about why this is important (as the file is very complex)

Figure 5.1.17: Large Minecraft File

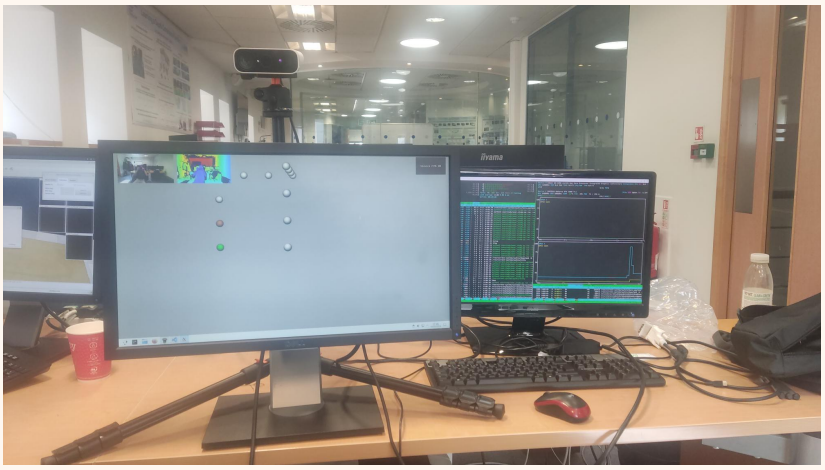


5.1.6 Portability

One of the key aspects of the system is that it is portable and should be runnable from a fresh linux machine from scratch with a single command. We developed the system on a

Linux Machine (NixOS) using an Intel i5 9600k, with a Nvida 2070 Super GPU. We tested the system on a different linux machine (NixOS) with an Intel i7 4770k and a Nvidia 1080 as can be seen in Fig 5.1.18. We were able to run the system on this machine with no issues first try. We also tested the system on a Windows machine using WSL2 TODO .

Figure 5.1.18: Running on a different machine



5.2 User Study Evaluation

We ran the study between begin and end date in Huxley 218 on the Imperial College London South Kensington Campus. In the end we managed to collect 16 participants, 14 of which agreed to be photographed as can be seen in Fig 5.2.1.

Figure 5.2.1: Participants

5.2.1 Participants

We found the easiest way to get participants was to ask acquaintance who were already on campus to take part.

This had the downside of our participant population not being a particularly representative section of the wider population. As can be seen in Fig population stats our participants were overwhelmingly clustered around the age of 22 and were male skewed with a disprotionate percentage of the population wearing glasses and having used vr before compare to national stats?

Figure 5.2.2: Age

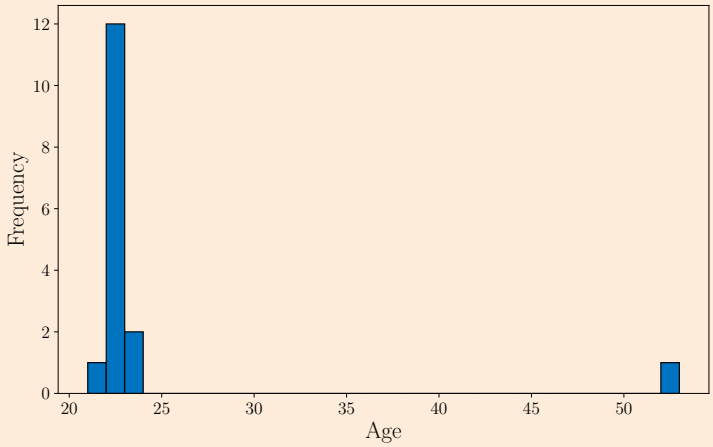


Figure 5.2.3: Glasses

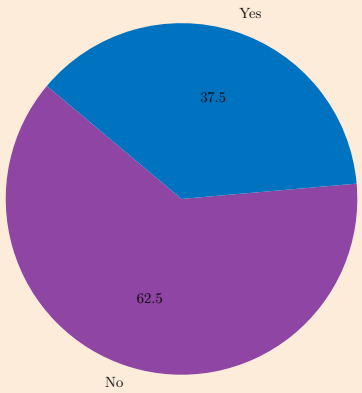


Figure 5.2.4: Used VR/AR

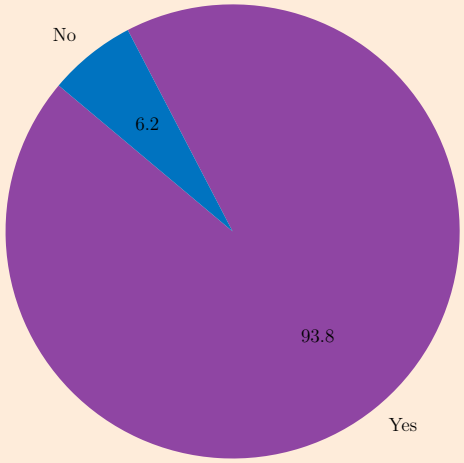
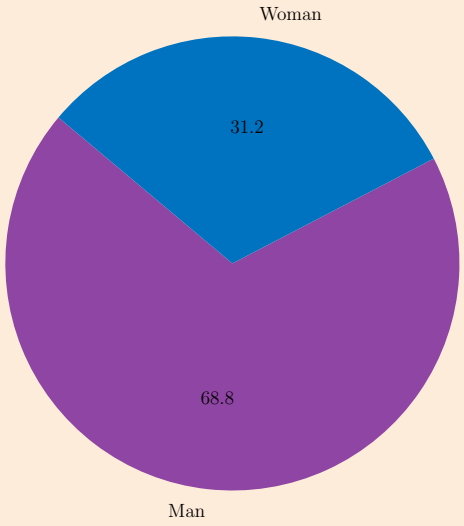


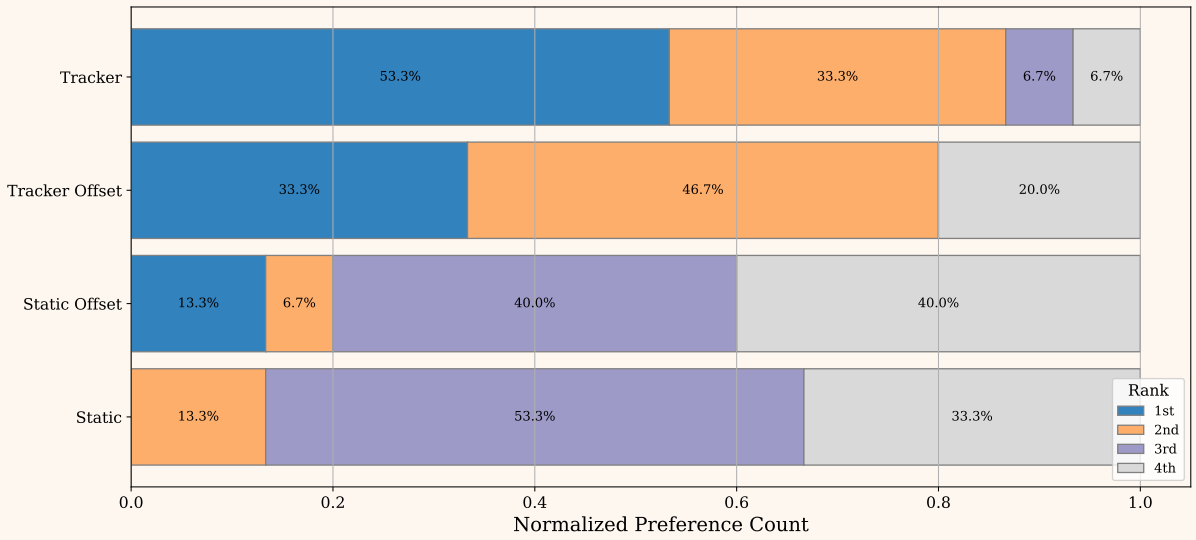
Figure 5.2.5: Gender



5.2.2 Survey Results

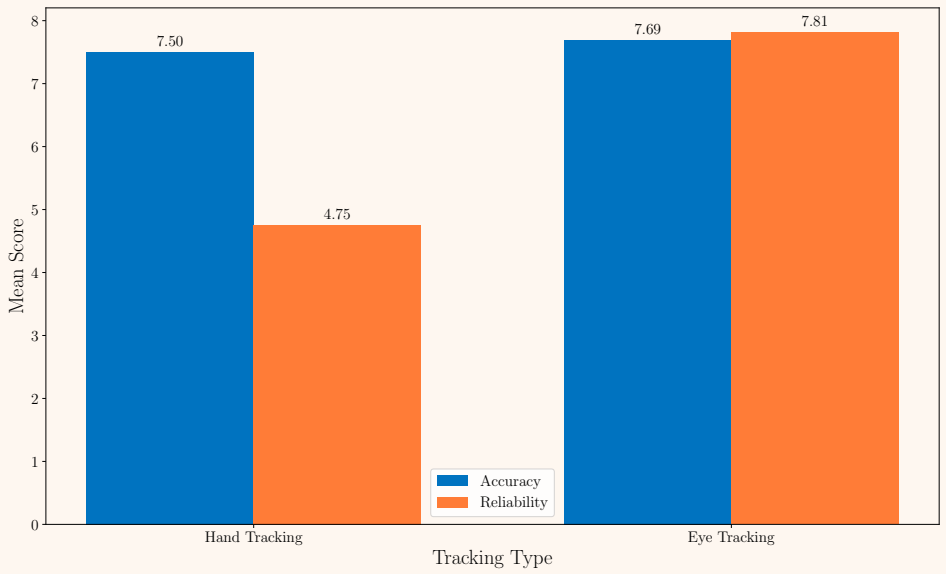
When asked to rank the conditions they undertook in order of preferences people overwhelmingly seemed to prefer the Tracker conditions as can be seen in 5.2.6. Only 13% of people picked either of the Static conditions as their first choice. write more and provide some analysis

Figure 5.2.6: Study Condition Rankings



Unsurprisingly people found the head tracking more reliable when compared to the head track as can be seen in 5.2.7. This is backed out by the fail rates we observed in both systems as can be seen in insert other figure currently in system, maybe move it to this study section?

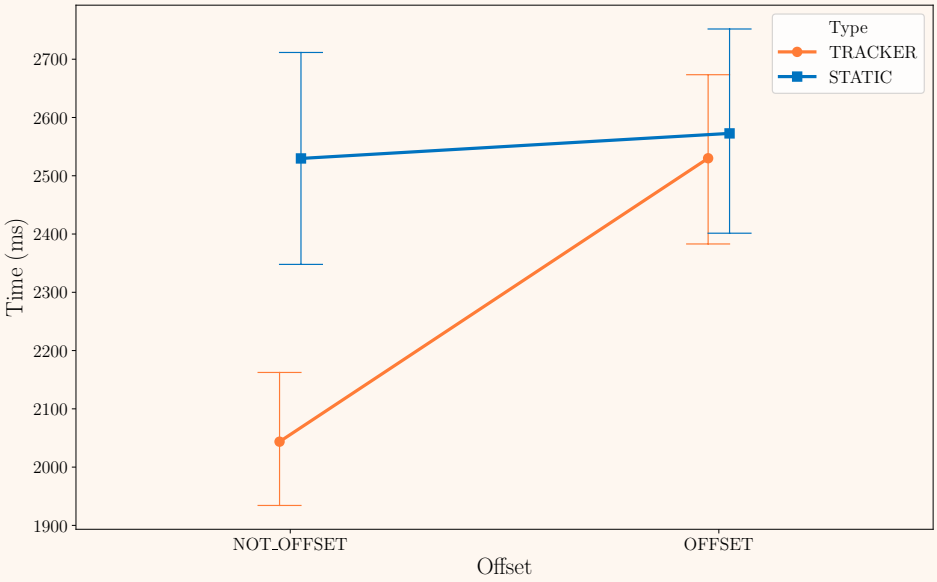
Figure 5.2.7: Mean Accuracy and Reliability of Hand and Eye Tracking



5.2.3 Data results

We analysed the time taken to complete each segment of each task and found that the type of view and the offset had a significant effect on the time taken to complete the segment. This can be seen in 5.2.8.

Figure 5.2.8: Interaction between Type and Offset of segment times



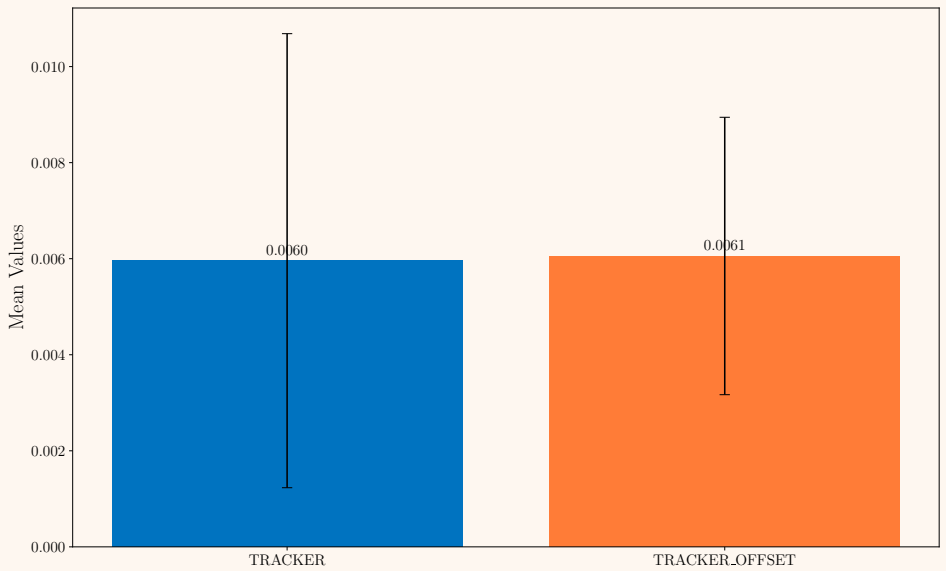
We ran a type 2 anova test to verify that these results are indeed significant as can be seen in Table 5.1.

Table 5.1: ANOVA Table for Fig 5.2.8

Source	Sum of Squares	df	F	p-value
`C(Type)`	$3.363\,186\times10^{7}$	1.0	10.698 212	0.001 092
`C(Offset)`	$3.341\,680\times10^{7}$	1.0	10.629 800	0.001 132
`C(Type):C(Offset)`	$2.344\,484\times10^{7}$	1.0	7.457 744	0.006 375
`Residual`	$5.985\,586\times10^{9}$	1904.0	NaN	NaN

People were best at completing the segments when using the TRACKER mode (3D view) directly in front of them. They approximately equally bad at completing all other conditions. This implies the 3D view is only really useful when directly in front of you. This is understandable as the further an object is away the less information you can get from it by moving your head. Interestingly we found that users still moved their head the same amount regardless of the view type as can be seen in Fig 5.2.9.

Figure 5.2.9: Combined Mean Eye Movement Values Per Millisecond and Standard Deviations



As can be seen from Table 5.2 the t-test we ran suggests the offset condition does not have a significant effect on eye/head movement.

Table 5.2: T-Test Result for Fig 5.2.9

Statistic	Value	p-value
T-statistic	−0.4170	0.6767

Interestingly the data seems to suggest that offset interaction does not have a significant effect on the time taken to complete the segment. This is surprising as we expected the offset to have a significant effect on the time taken to complete the segment. One people reason might be the downside of the offset might be accounted for by the fact the participants hands no longer occlude their view of the scene.

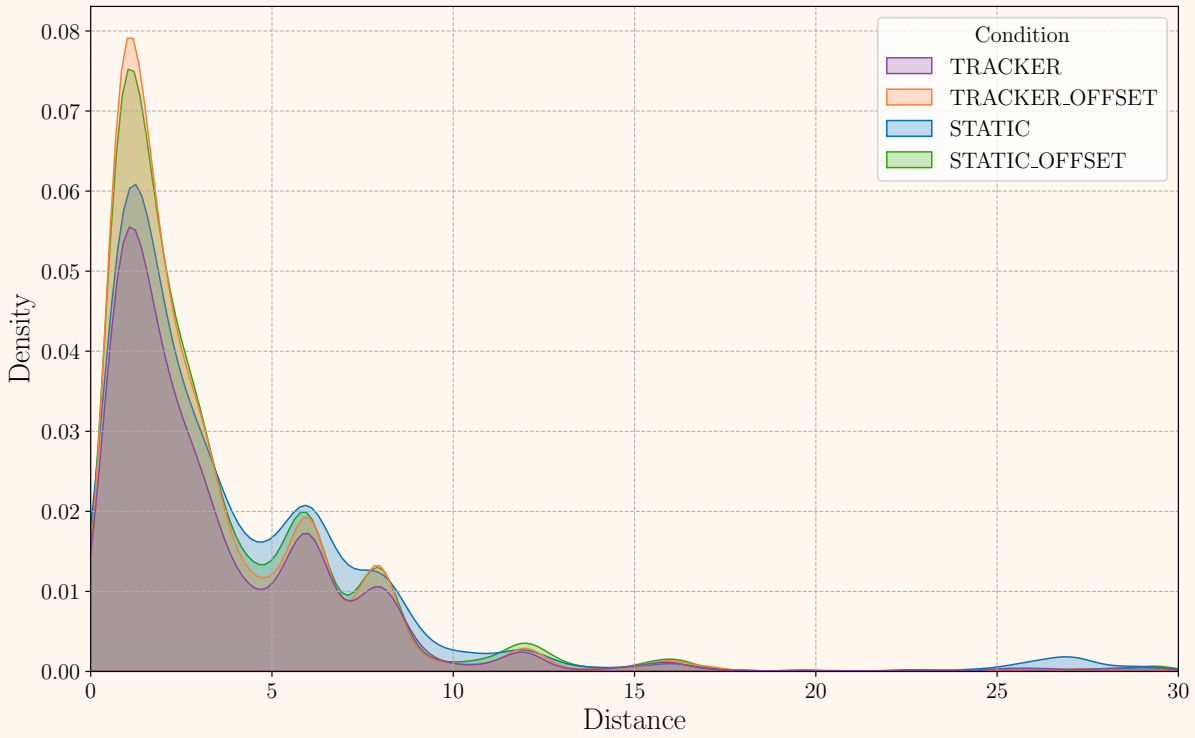
One study we would like to run in the future is to see what happens if we offset the interaction zone instead of the view, this would control for the apparent factor that 3D is less useful far away.

Another interesting observation we made was that in offset conditions users were significantly worse at precisely completing the point at the end. This can be seen in Fig 5.2.10 where the conditions TRACKER_OFFSET and STATIC_OFFSET have a much larger peak just before the distance of zero (the point where participants are trying to precisely select the end of the segment).

It is worth noting the that appear are almost certainly a result of the time when users have realised they have completed a segment as they appear at the same distances as the lengths

of the segments (2cm x 1, 3cm x 8, 6cm x 12, 8cm x 10, 12cm x 3, 16cm x 1).

Figure 5.2.10: Distribution of Sampled Points Distances by Segment End



Add more graphs in the appendix and talk about them here, put the task side by side with the results

Chapter 6

Conclusions and Future Work

6.1 Conclusions

1. We have developed a novel system for visualising 3D data in a 2D space.
2. We were able to validate it with a user study.
   (a) Write about results form user study.
3. We are unaware of any other system that can do this with hand tracking.
4. We have built a system that is easy to use and intuitive and portable.

6.2 Future Work

- **Adapt to be compliant with OpenXR:** OpenXR is an open standard for virtual reality and augmented reality. It is supported by all the major players in the industry including Microsoft, Valve, Oculus, Google, and many more. It would be a good idea to adapt the project to be compliant with this standard, so it can load into any OpenXR compatible application. I am currently not sure how difficult this is or if it is even feasible.
- **Anaglyph 3D:** Anaglyph 3D is a method of displaying 3D images using filters typically red and green color filters and does not require special hardware. The 3D effect currently requires 1 eye to be closed so adding 3D support would make it a more immersive experience. I predict this task will take a day or two as I just need to duplicate the perspective per eye.
- **Realtime light detection:** Taking inspiration from what I have learned from advanced graphics this term, it might be interesting to add another camera, a fish eye lens and use that to generate a real-time light map. This would allow the virtual scene to be lit by real-world lighting. I have already talked to Prof Abhijeet Ghosh about this idea, and he thinks it is feasible. However, this is going in a slightly different direction with the project. I predict this task will take a week or two.
- **Improve compatibility:** Currently the project only works on Nvidia GPUs. It would be good to improve compatibility to work on AMD GPUs and Intel GPUs and also run without a GPU (albeit slowly). It would also be good to support different depth cameras other than the Kinect (Like Intels Intellisense) as this has been discontinued by Microsoft. This would require a lot of refactorings and would probably take a week or two.
- **Switch hand tracking model:** By a fairly significant margin, the hand tracking was the weakest part of the project. It would be good to switch to a more robust hand tracking model. I think it would be good to switch to a model that tracks using the depth image rather than the RGB image. This would probably be a fairly large undertaking as there is unlikely to be an off the shelf model that does this. I think implementing this paper may be promising ToCite . I predict this task will take a month or two.
- **Publish User Study**: ask nicole about this

- **Further User Study**: Ran the experiment again this time moving the offset zone not the display

6.3 Contributions

6.4 Novel Contributions

Once this project is complete we expect to have made the following novel contributions:

- A **volumetric display simulator** that is Multi-platform, Lightweight, Cheap, and Reproducible.
- A **user experiment** that compares the effectiveness of using hand tracking to interact directly with an ethereal/incorporeal volumetric display compared to a via teleoperation with a corporeal/tangible display.

Bibliography

[1] URL: https://nixos.wiki/wiki/Nix_Community (visited on 01/09/2024).

[2] E. H. Adelson et al. "1984, Pyramid methods in image processing". In: *RCA Engineer* 29.6 (1984), pp. 33–41.

[3] *assimp/assimp*. original-date: 2010-05-05T12:53:45Z. June 10, 2024. URL: https://github.com/assimp/assimp (visited on 06/10/2024).

[4] *Azure Kinect developer kit – Microsoft*. Microsoft Store. URL: https://www.microsoft.com/en-gb/d/azure-kinect-dk/8pp5vxmd9nhq (visited on 06/11/2024).

[5] *bazelbuild/bazel*. original-date: 2014-06-12T16:00:38Z. June 10, 2024. URL: https://github.com/bazelbuild/bazel (visited on 06/10/2024).

[6] Pierre-Alexandre Blanche. *Holography, and the future of 3D display*. 2021. DOI: 10.37188/lam.2021.028. URL: https://www.light-am.com//article/id/82c54cac-97b0-4d77-8ed8-4edda712fe7c.

[7] James F. Blinn. "Models of light reflection for computer synthesized pictures". In: *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '77. San Jose, California: Association for Computing Machinery, 1977, pp. 192–198. ISBN: 9781450373555. DOI: 10.1145/563858.563893. URL: https://doi.org/10.1145/563858.563893.

[8] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[9] *brightvox 3D June , 2023 NEXMEDIA exhibition - Holographic Signage #volumetric*. June 2023. URL: https://www.youtube.com/watch?v=mxyw6LkAtiQ (visited on 01/23/2024).

[10] Burr, Chris, Clemencic, Marco, and Couturier, Ben. "Software packaging and distribution for LHCb using Nix". In: *EPJ Web Conf.* 214 (2019), p. 05005. DOI: 10.1051/epjconf/201921405005. URL: https://doi.org/10.1051/epjconf/201921405005.

[11] Bruno Bzeznik et al. "Nix as HPC Package Management System". In: *Proceedings of the Fourth International Workshop on HPC User Support Tools*. HUST'17. Denver, CO, USA: Association for Computing Machinery, 2017. ISBN: 9781450351300. DOI: 10.1145/3152493.3152556. URL: https://doi.org/10.1145/3152493.3152556.

[12] *dlib C++ Library: Easily Create High Quality Object Detectors with Deep Learning*. URL: https://blog.dlib.net/2016/10/easily-create-high-quality-object.html (visited on 06/11/2024).

[13] *Dlib provided files*. URL: http://dlib.net/files/ (visited on 06/11/2024).

[14] Eelco Dolstra. *The purely functional software deployment model*. Utrecht University, 2006.

[15] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. "Nix: A Safe and Policy-Free System for Software Deployment." In: *LISA*. Vol. 4. 2004, pp. 79–92.

[16] Eelco Dolstra and Andres Löh. "NixOS: A Purely Functional Linux Distribution". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 367–378. ISBN: 9781595939197. DOI: 10.1145/1411204.1411255. URL: https://doi.org/10.1145/1411204.1411255.

[17] Epic Games. *Unreal Engine*. June 10, 2024. URL: https://www.unrealengine.com.

[18] Dylan Fafard et al. "FTVR in VR: Evaluation of 3D Perception With a Simulated Volumetric Fish-Tank Virtual Reality Display". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, pp. 1–12. ISBN: 9781450359702. DOI: 10.1145/3290605.3300763. URL: https://doi.org/10.1145/3290605.3300763.

[19] Dylan Brodie Fafard et al. "Design and implementation of a multi-person fish-tank virtual reality display". In: *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*. VRST '18. Tokyo, Japan: Association for Computing Machinery, 2018. ISBN: 9781450360869. DOI: 10.1145/3281505.3281540. URL: https://doi.org/10.1145/3281505.3281540.

[20] G.E. Favalora. "Volumetric 3D displays and application infrastructure". In: *Computer* 38.8 (2005), pp. 37–44. DOI: 10.1109/MC.2005.276.

[21] Gregg E. Favalora et al. "100-million-voxel volumetric display". In: *Cockpit Displays IX: Displays for Defense Applications*. Ed. by Darrel G. Hopper. Vol. 4712. International Society for Optics and Photonics. SPIE, 2002, pp. 300–312. DOI: 10.1117/12.480930. URL: https://doi.org/10.1117/12.480930.

[22] NixOS Foundation. URL: https://github.com/NixOS/nixpkgs (visited on 01/09/2024).

[23] Tatsuki Fushimi et al. "Acoustophoretic volumetric displays using a fast-moving levitated particle". In: *Applied Physics Letters* 115.6 (Aug. 2019), p. 064101. ISSN: 0003-6951. DOI: 10.1063/1.5113467. eprint: https://pubs.aip.org/aip/apl/article-pdf/doi/10.1063/1.5113467/13562800/064101_1_online.pdf. URL: https://doi.org/10.1063/1.5113467.

[24] *g-truc/glm*. original-date: 2012-09-06T00:04:56Z. June 10, 2024. URL: https://github.com/g-truc/glm (visited on 06/10/2024).

[25] Matthew Gately et al. "A Three-Dimensional Swept Volume Display Based on LED Arrays". In: *J. Display Technol.* 7.9 (Sept. 2011), pp. 503–514. URL: https://opg.optica.org/jdt/abstract.cfm?URI=jdt-7-9-503.

[26] *glfw/glfw*. original-date: 2013-04-18T15:24:53Z. June 10, 2024. URL: https://github.com/glfw/glfw (visited on 06/10/2024).

[27] *Hand landmarks detection guide | Google AI Edge*. Google for Developers. URL: https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker (visited on 06/11/2024).

[28] Donald Hearn, M Pauline Baker, and M Pauline Baker. *Computer graphics with OpenGL*. Vol. 3. Pearson Prentice Hall Upper Saddle River, NJ: 2004.

[29] David Herberth. *Dav1dde/glad*. original-date: 2013-07-29T10:54:13Z. June 10, 2024. URL: https://github.com/Dav1dde/glad (visited on 06/10/2024).

[30] Ryuji Hirayama et al. "A volumetric display for visual, tactile and audio presentation using acoustic trapping". In: *Nature* 575.7782 (Nov. 2019), pp. 320–323. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1739-5. URL: https://doi.org/10.1038/s41586-019-1739-5.

[31] Ryuji Hirayama et al. "Design, Implementation and Characterization of a Quantum-Dot-Based Volumetric Display". In: *Scientific Reports* 5.1 (Feb. 2015), p. 8472. ISSN: 2045-2322. DOI: 10.1038/srep08472. URL: https://doi.org/10.1038/srep08472.

[32] Vahid Kazemi and Josephine Sullivan. "One millisecond face alignment with an ensemble of regression trees". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 1867–1874. DOI: 10.1109/CVPR.2014.241.

[33] Sean Frederick KEANE et al. "Volumetric 3d display". WO2016092464A1. June 2016. URL: https://patents.google.com/patent/WO2016092464A1/en (visited on 01/17/2024).

[34] D Kersten, P Mamassian, and D C Knill. "Moving cast shadows induce apparent motion in depth". en. In: *Perception* 26.2 (1997), pp. 171–192.

[35] D Kersten, P Mamassian, and D C Knill. "Moving cast shadows induce apparent motion in depth". en. In: *Perception* 26.2 (1997), pp. 171–192.

[36] Davis E. King. "Dlib-ml: A Machine Learning Toolkit". In: *Journal of Machine Learning Research* 10 (2009), pp. 1755–1758.

[37] Davis E. King. *Max-Margin Object Detection*. 2015. arXiv: 1502.00046 [cs.CV].

[38] Robert Kooima. "Generalized perspective projection". In: (2009).

[39] Markus Kowalewski and Phillip Seeber. "Sustainable packaging of quantum chemistry software with the Nix package manager". In: *International Journal of Quantum Chemistry* 122.9 (2022), e26872. DOI: https://doi.org/10.1002/qua.26872. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.26872. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.26872.

[40] David Luebke. "CUDA: Scalable parallel programming for high-performance scientific computing". In: *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 2008, pp. 836–838. DOI: 10.1109/ISBI.2008.4541126.

[41] Camillo Lugaresi et al. *MediaPipe: A Framework for Building Perception Pipelines*. 2019. arXiv: 1906.08172 [cs.DC].

[42] Dmitry Marakasov. Jan. 2024. URL: https://repology.org/repositories/graphs (visited on 01/09/2024).

[43] Morgan McGuire. *Computer Graphics Archive*. July 2017. URL: https://casual-effects.com/data.

[44] *Microsoft's Azure Kinect Developer Kit Technology Transfers to Partner Ecosystem*. TECHCOMMUNITY.MICROSOFT.COM. URL: https://techcommunity.microsoft.com/t5/mixed-reality-blog/microsoft-s-azure-kinect-developer-kit-technology-transfers-to/ba-p/3899122 (visited on 06/10/2024).

[45] *microsoft/Azure-Kinect-Sensor-SDK*. original-date: 2018-12-10T17:50:05Z. June 8, 2024. URL: https://github.com/microsoft/Azure-Kinect-Sensor-SDK (visited on 06/10/2024).

[46] Shree K. Nayar and Vijay N. Anand. "3D volumetric display using passive optical scatterers". In: *ACM SIGGRAPH 2006 Sketches*. SIGGRAPH '06. Boston, Massachusetts: Association for Computing Machinery, 2006, 106–es. ISBN: 1595933646. DOI: 10.1145/1179849.1179982. URL: https://doi.org/10.1145/1179849.1179982.

[47] *pallets/click*. original-date: 2014-04-24T09:52:19Z. June 11, 2024. URL: https://github.com/pallets/click (visited on 06/11/2024).

[48] Shreya K. Patel, Jian Cao, and Alexander R. Lippert. "A volumetric three-dimensional digital light photoactivatable dye display". In: *Nature Communications* 8.1 (July 2017), p. 15239. ISSN: 2041-1723. DOI: 10.1038/ncomms15239. URL: https://doi.org/10.1038/ncomms15239.

[49] *Products*. en-AU. URL: https://voxon.co/products/ (visited on 01/17/2024).

[50] Randi J Rost et al. *OpenGL shading language*. Pearson Education, 2009.

[51] Anthony Rowe. "Within an ocean of light: creating volumetric lightscapes". In: *ACM SIGGRAPH 2012 Art Gallery*. SIGGRAPH '12. Los Angeles, California: Association for Computing Machinery, 2012, pp. 358–365. ISBN: 9781450316750. DOI: 10.1145/2341931.2341937. URL: https://doi.org/10.1145/2341931.2341937.

[52] Christos Sagonas et al. "300 Faces in-the-Wild Challenge: The First Facial Landmark Localization Challenge". In: *2013 IEEE International Conference on Computer Vision Workshops*. 2013, pp. 397–403. DOI: 10.1109/ICCVW.2013.59.

[53] Jürgen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *CoRR* abs/1404.7828 (2014). arXiv: 1404.7828. URL: http://arxiv.org/abs/1404.7828.

[54] D. E. Smalley et al. "A photophoretic-trap volumetric display". In: *Nature* 553.7689 (Jan. 2018), pp. 486–490. ISSN: 1476-4687. DOI: 10.1038/nature25176. URL: https://doi.org/10.1038/nature25176.

[55] Jörg Thalheim. *About Nix sandboxes and breakpoints (NixCon 2018)*. Youtube. 2018. URL: https://www.youtube.com/watch?v=ULqoCjANK-I (visited on 01/09/2024).

[56] *tinyobjloader/tinyobjloader*. original-date: 2012-08-15T02:44:30Z. June 8, 2024. URL: https://github.com/tinyobjloader/tinyobjloader (visited on 06/10/2024).

[57] *Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine*. Unity. URL: https://unity.com/ (visited on 06/10/2024).

[58] Hakan Urey et al. "State of the Art in Stereoscopic and Autostereoscopic Displays". In: *Proceedings of the IEEE* 99.4 (2011), pp. 540–555. DOI: 10.1109/JPROC.2010.2098351.

[59] *Use Azure Kinect Sensor SDK image transformations | Microsoft Learn*. URL: https://learn.microsoft.com/en-us/azure/kinect-dk/use-image-transformation (visited on 06/11/2024).

[60] *Voxon features on CNET's What The Future*. en-AU. URL: https://voxon.co/voxon-features-cnet-what-the-future/ (visited on 01/23/2024).

[61] Shigang Wan et al. "A Prototype of a Volumetric Three-Dimensional Display Based on Programmable Photo-Activated Phosphorescence". In: *Angewandte Chemie International Edition* 59.22 (2020), pp. 8416–8420. DOI: https://doi.org/10.1002/anie.202003160. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/anie.202003160. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/anie.202003160.

[62] Endong Wang et al. "Intel Math Kernel Library". In: *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*. Cham: Springer International Publishing, 2014, pp. 167–188. ISBN: 978-3-319-06486-4. DOI: 10.1007/978-3-319-06486-4_7. URL: https://doi.org/10.1007/978-3-319-06486-4_7.

[63] Colin Ware, Kevin Arthur, and Kellogg S. Booth. "Fish tank virtual reality". In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. Amsterdam, The Netherlands: Association for Computing Machinery, 1993, pp. 37–42. ISBN: 0897915755. DOI: 10.1145/169059.169066. URL: https://doi.org/10.1145/169059.169066.

[64] Mason Woo et al. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[65] Manfredas Zabarauskas. "3D Display Simulation Using Head-Tracking with Microsoft Kinect". Examination: Part II in Computer Science, June 2012. Word Count: 119761. Project Originator: M. Zabarauskas. Supervisor: Prof N. Dodgson. Unpublished master's thesis. Wolfson College: University of Cambridge, May 2012.

[66] Fan Zhang et al. *MediaPipe Hands: On-device Real-time Hand Tracking*. 2020. arXiv: 2006.10214 [cs.CV].

[67] Matthias Zwicker et al. "Multi-view Video Compression for 3D Displays". In: *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*. 2007, pp. 1506–1510. DOI: 10.1109/ACSSC.2007.4487481.