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.
- 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 visually represents objects and scenes natively in 3D. These displays differ from more traditional virtual reality devices in that they are not immersive; rather, they coexist with their surroundings. They can be viewed from any angle without the need for special visual apparatus and can be observed by multiple people simultaneously [30]. 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 3.2, there are currently many approaches being explored by both academic and industrial research groups to develop these displays.

Figure 1.1.1: PerspectaRAD being used to view a CT image of a patient's head [64]

Figure 1.1.2: Graph Visualization on the Voxon Photonics VX1 [86]

Due to their capacity to display objects and scenes to multiple viewers simultaneously, volumetric displays have been applied to a variety of professional and academic fields, including but not limited to medical imaging [39] (see Fig 1.1.1), scientific visualization (see Fig. 1.1.2), computer-aided design [77], and military visualization [69] [30] [6].

As volumetric displays become more advanced and accessible, they have the potential to revolutionize the way we interact with digital content, providing benefits over alternative mediums like Virtual Reality (VR) (see Background 3.2). However, the high cost and complexity of these devices have limited their adoption and hindered research into their usability in the past.

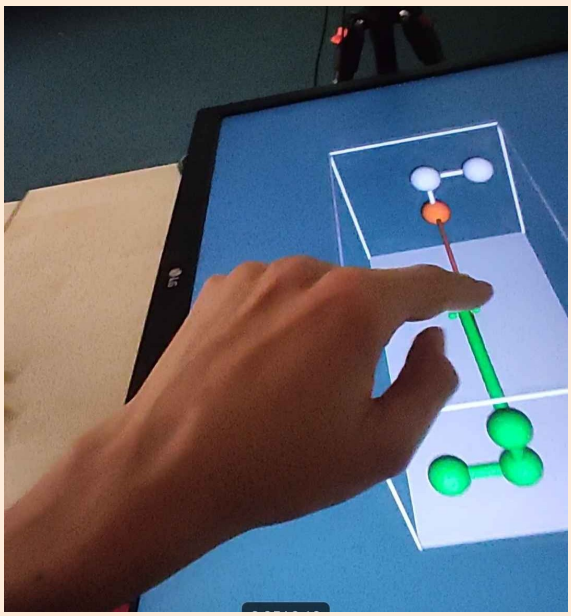
It is currently difficult to conduct human-computer interaction (HCI) research in the field of volumetric displays because these devices are not widely available and are prohibitively expensive, costing upwards of $10,000 USD [68]. There have been attempts to create devices that simulate volumetric displays to address this issue (see Background 3.2), but these solutions are often complex and expensive to replicate.

1.2 Contributions

Our project makes two novel contributions to the field of volumetric displays. Firstly, we built a system for simulating volumetric displays for use in HCI research. Secondly, we utilized our device to conduct a user study that successfully demonstrated the research viability of our system.

Figure 1.2.1: Our simulator rendering Imperial College London in 3D

Figure 1.2.2: POV perspective completing a task our user study



Volumetric Display Simulator

We created our Volumetric Display Simulator (see Implementation 5.1) with the intention of forming a strong foundation for conducting future user studies involving volumetric displays.

Our simulator (as seen in Fig 1.2.1) uses head tracking to render the correct perspective for the viewer using the device. This allows the user to change the view while maintaining the illusion of a 3D scene in front of them. We also implemented native hand tracking integration, allowing the user to interact with the scene using their hands.

We designed our system in a such a way that it is:

1. **Cost-Effective:** The system was designed to operate on a standard desktop computer with regular monitors using common hardware. It utilizes Microsoft's widely used Azure Kinect Camera for its head and hand tracking. We believe our system's affordability and accessibility will encourage more researchers to explore volumetric display technology.

2. **Reproducible:** The system was developed using Nix (see Background 3.3.2), a package manager that facilitates the easy reproduction and compilation of software environ-

ments. This means the system can be run on any computer with Nix installed using a single command (see Implementation 5.2.4). This not only simplifies execution but also enhances the ease of sharing and reproducing results.

3. **Simple and Lightweight:** The system was intentionally designed to be as simple as possible by leveraging well-established libraries (see Implementation 5.3.5 and Implementation 5.4.8). The simulator comprises only approximately 2,000 lines of C++ code, making it straightforward to understand and modify. We hope that this simplicity will encourage other researchers to build upon our work and develop new features.

User Study

We conducted a full user study to validate the effectiveness of our simulator. Our aim was to demonstrate that our system could be used to conduct meaningful research into the usability of volumetric displays.

We chose to investigate the performance of participants in a task that required them to trace a path with their hands under different conditions (see Implementation 5.5.7). An example of the task can be seen in Fig. 1.2.2. We focused on two factors: the perspective of the display (2D vs. 3D) and the method of interaction (direct hand interaction vs teleoperation), resulting in a total of four conditions.

The findings indicated that using head tracking for a 3D perspective significantly improved task completion speed and accuracy compared to a 2D display. Additionally, direct hand interaction with the volumetric display yielded superior results compared to teleoperation, particularly when paired with a 3D view.

We observed a strong user preference for the 3D perspective while using direct hand interaction, suggesting that users value natural and intuitive interaction modes. The findings reaffirm the advantage of motion parallax in 3D tasks, showing that the benefits of 3D tracking are diminished with positional offsets.

The study identifies several areas for further investigation, including the effects of varying offset distances and the potential impact of offsetting the interaction zone rather than the display itself. These recommendations can guide future research efforts to refine and improve the usability of volumetric displays. (For more information see Evaluation 6.2.3).

Chapter 2

Ethical Discussion

2.1 User Study

We conducted a user study to evaluate our simulator, titled "A Virtual Volumetric Screen User Study." This study followed the ethical guidelines and approval process as outlined by the Science Engineering Technology Research Ethics Committee at Imperial College London. The process was reviewed by the Research Governance and Integrity Team and the head of the Computing Department. We received approval on 2nd May 2024 and conducted the study between the 1st and 5th of June.

2.1.1 Human Participants

Since our study involved human participants, we ensured that it was conducted ethically and in accordance with relevant guidelines. We adhered to the Equality Act 2010 [81] to avoid excluding any participants. Additionally, we took care to avoid involving participants who might feel pressured to participate or influenced by the researchers. Prior to the commencement of the study, participants were provided with an information sheet and a consent form.

2.1.2 Data Collection

During data collection, we complied with local regulations, including the General Data Protection Regulation (GDPR) [27]. All data were securely stored on a computer located on campus, accessible only to the researchers, or on Imperial's secure cloud network. We ensured that data presented in our report were anonymized. The data will be retained until 30th June 2034.

2.2 Volumetric Simulator

2.2.1 Military Applications

Although this technology could theoretically be used for military applications, we believe it is unlikely to be employed for such purposes. If it were to be used, it would not be for direct combat and would pose no more danger than other existing technologies.

2.2.2 Copyright Limitations

Open Source

We utilize several open-source libraries and tools in our project. The Azure Kinect SDK is licensed under the MIT license. We use the Nix package manager, licensed under the LGPL-2.1 license, and the Nixpkgs repository, which is also under the MIT license. The `dlib` library is used under the Boost Software License 1.0 (BSL-1.0). The OpenGL library is employed under its open-source license for the Sample Implementation (SI). We also utilize the GLFW library, licensed under the zlib license, and the GLM library, licensed under the MIT license. Additionally, we use OpenCV, which is licensed under the Apache License, and MediaPipe, which is released under the Apache License 2.0. The tinyobjloader is used under the MIT license.

Proprietary Software

We also use proprietary software in our project. This includes Microsoft's proprietary depth engine, designed for use with the Azure Kinect SDK, which is not open source. Additionally, we use CUDA and related libraries, which are proprietary software developed by NVIDIA.

Chapter 3

Background

3.1 Volumetric displays

Volumetric displays [30] 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.

3.1.1 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 [31], an emitting screen, typically an LED-based [36], or a transparent projector screen [47]. There currently exist commercial products that implement this technique as can be seen in Fig 3.1.1 and Fig 3.1.2.

Figure 3.1.1: The VXR4612 3D Volumetric Display, a projector-based persistence of vision display produced by Voxon Photonics. [85]

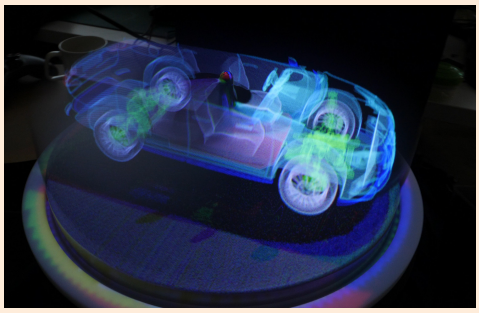


Figure 3.1.2: A Volumetric Display / Holographic Signage, an LED-based persistence of vision display produced by Brightvox Inc. [14]

3.1.2 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 [70], lasers and phosphorus gas [87], or a transparent laser-induced damaged medium that can be projected into [65]. There has been research into photon-activated dye [67] and even quantum dot-based displays [45]. An example of one such display can be seen in Fig 3.1.3.

3.1.3 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. [34] [44] 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 [74]. 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. An example of one such display can be seen in Fig 3.1.4.

Figure 3.1.3: Columbia University's passive optical scattering volumetric display [65]

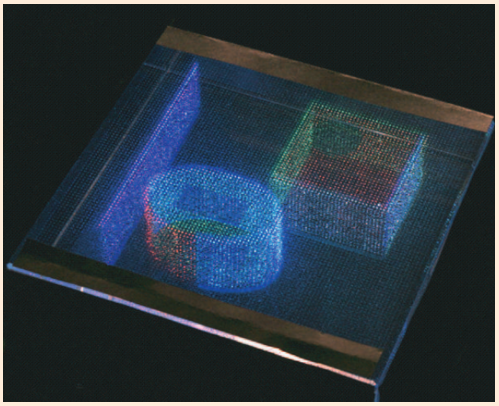
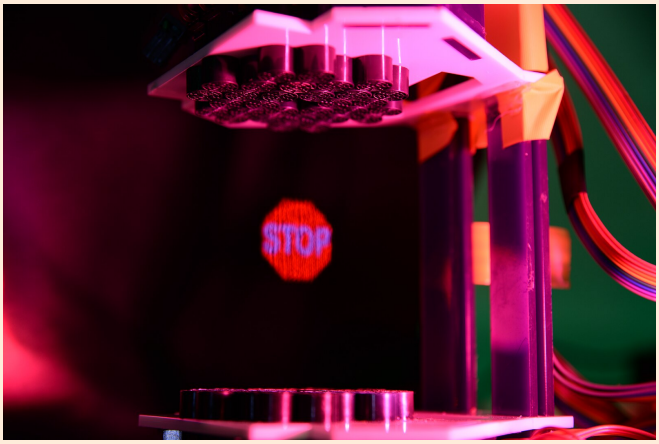


Figure 3.1.4: Bristol University's acoustic trapping volumetric display [34]



3.1.4 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 [68] 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 [11] 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 [95] but this still provides a major issue.

3.2 Volumetric Screen Simulations

Because of these issues, there has been research into simulating volumetric displays. One commonly used method is the so-called fish tank virtual reality (FTVR) display [89] which has been commonly used to simulate volumetric displays, [29], [91]. 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 [83]). 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 [28].

3.3 Nix/NixOS

To achieve our goal of making our system reproducible and buildable with a single command (See Implementation 5.2.4) we have used the Nix package manager. It would not be an understatement to say this would have been impossible without Nix. Nix can be likened to the Python package manager 'pip,' but with significantly enhanced capabilities and applicability that extend beyond just package management to encompass the entire system environment.

3.3.1 Introduction to Nix

Nix [23] 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 [22] is widely used in both industry [1] and academia [16] [56] [15], and its associated public package repository `nixpkgs` [33] as of Jan 2024 has over 80,000 unique packages making it the largest up-to-date package repository in the world [59]. Out of Nix has also grown **NixOS** [24] 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 3.3.1 as opposed to the traditional Unix Filesystem Hierarchy Standard (FHS).

Figure 3.3.1: 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 [79]. 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 [15].

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 3.3.2.

Figure 3.3.2: 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.

3.3.2 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 3.3.3 that builds a version of the classic example package ”hello”.

Listing 3.3.3: flake.nix

```
1 {
2   description = "A flake for building Hello World";
3   inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";
4
```

```
5   outputs = { self, nixpkgs }: {
6     defaultPackage.x86_64-linux =
7       let
8         pkgs = nixpkgs.legacyPackages.x86_64-linux;
9       in
10      pkgs.stdenv.mkDerivation {
11        name = "hello-2.12.1";
12        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`.

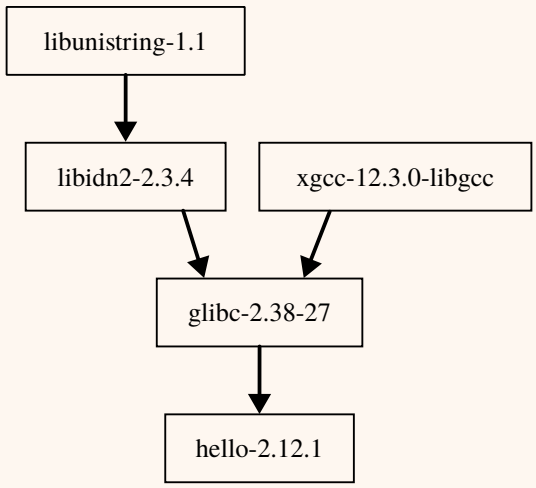
Below we have given some examples of how to run and investigate our hello package in Listing 3.3.4.

Listing 3.3.4: 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 3.3.5 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 3.3.5: Dependency graph



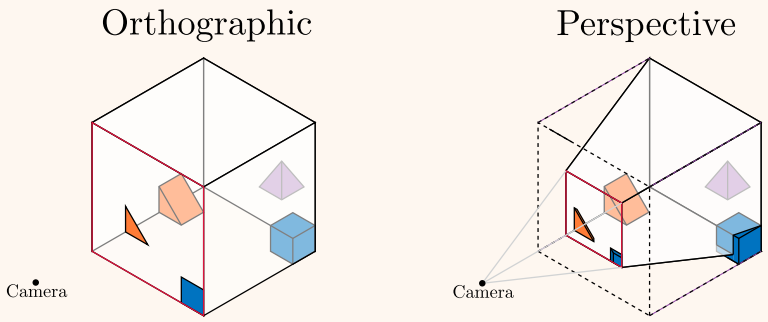
Chapter 4

Methods

4.1 Perspective Projection

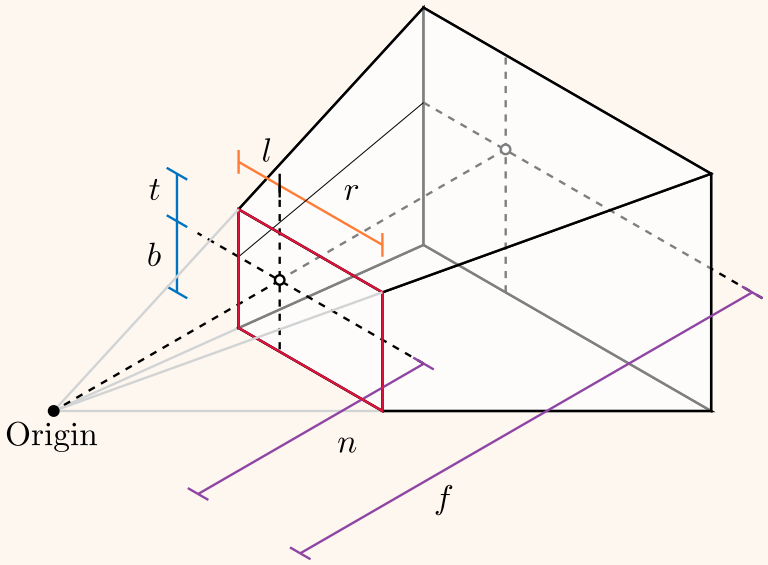
To create the illusion of a 3D scene behind and in front of a display we must use a perspective projection. This is a method of representing 3D space on a 2D surface that simulates the way the human eye perceives the world. For our project we have followed the methods laid out in ”Generalized Perspective Projection” by Robert Kooima [55] to calculate the required values for the OpenGL function `frustum` to generate a perspective projection.

Figure 4.1.1: 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 4.1.1. 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 4.1.2: Using frustum to generate a perspective projection



OpenGL provides the `frustum` function as seen in Fig 4.1.2 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). 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

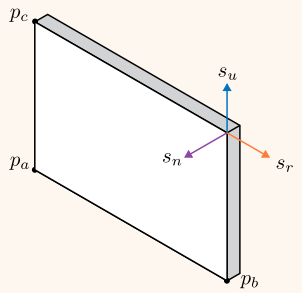
3D scene behind and in front of a display using this `frustum` function. This can be done fairly trivially following Robert Kooima's method he sets out in "Generalized Perspective Projection" to calculate f, l, r, b, t, n as the viewer's eye changes position [55].

4.1.1 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 4.1.3. 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 4.1.3: Defining a screen in 3D space



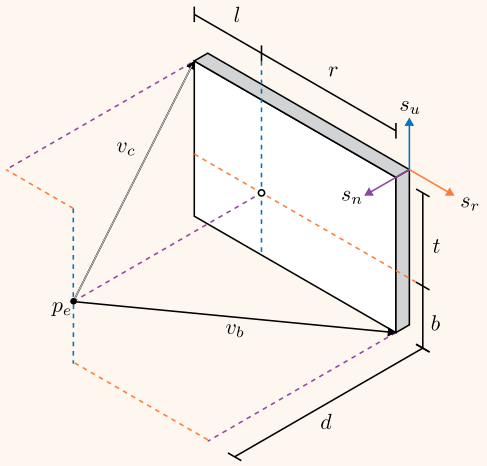
Next, we introduce the viewer's eye which we will refer to as p_e. We can draw 2 vectors v_b, v_c from the viewer's eye p_e to the corners of the screen p_b, p_c as seen in Fig 4.1.4. In the diagram, we also have labeled the components of each of these vectors in the basis of the screen. We can compute 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:

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

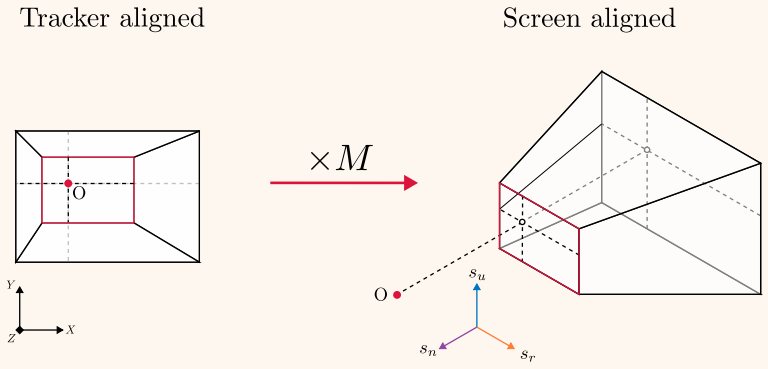
Figure 4.1.4: Screen Intersection with view



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 4.1.5. 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 4.1.5: Rotating the frustum from tracker space alignment into screen space alignment

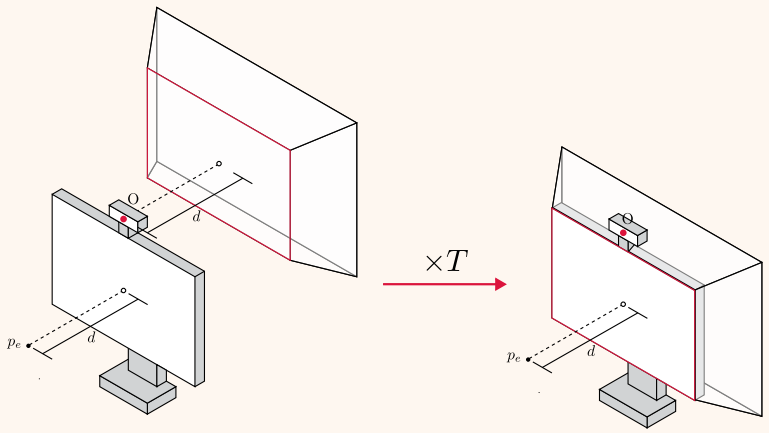


The second problem we have is that we want our projection matrix to move around with the viewer's eye however the mathematics of perspective projection disallow this, with the camera assumed to be at the origin. To translate our viewing frustum to our eye position we must instead translate our eye position (and the whole world) to the origin of our frustum. This can

be done with a translation matrix T as seen in Fig 4.1.6. 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 4.1.6: Translating the viewing frustum to sit inside the screen

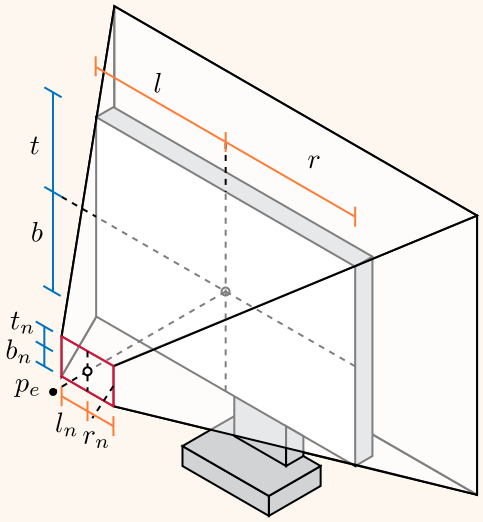


We now have a working method for projecting virtual objects behind our screen onto our screen however it is also possible if we desire to project objects in front of our screen onto the screen as well as long as they lie within the pyramid formed between the edges of the screen and the viewer's eye. 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 4.1.7 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 4.1.7: 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.

4.1.2 Sample code

Below in Listing 4.1.8 we have given an example of a function implementing the process we have just described in C++.

Listing 4.1.8: projection.cpp, Sample code for creating the 3D illusion projection

```
#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);

```

```
21      // Frustum extents (scaled to the near clipping plane)
22      GLfloat l = dot(sr, vc) * n / d;
23      GLfloat r = dot(sr, vb) * n / d;
24      GLfloat b = dot(su, vb) * n / d;
25      GLfloat t = dot(su, vc) * n / d;
26
27      // Create the projection matrix
28      mat4 projMatrix = frustum(l, r, b, t, n, f);
29
30      // Rotate the projection to be aligned with screen basis.
31      mat4 rotMatrix(1.0f);
32      rotMatrix[0] = vec4(sr, 0);
33      rotMatrix[1] = vec4(su, 0);
34      rotMatrix[2] = vec4(sn, 0);
35
36      // Translate the world so the eye is at the origin of the viewing frustum
37      mat4 transMatrix = translate(mat4(1.0f), -eye);
38
39      return projMatrix * rotMatrix * transMatrix;
40  }
```

Chapter 5

Implementation

5.1 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.

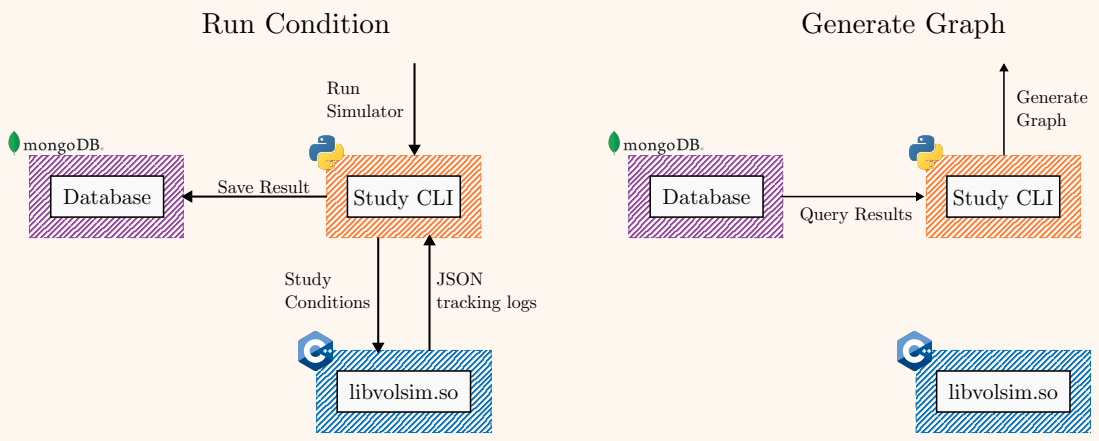
- **Renderer Subsystem**: The rendering system is responsible for displaying the volumetric scene and the virtual objects within it. It uses OpenGL to render the scene based on user positional data from the tracking system, thus creating the illusion of 3D.
- **Tracking Subsystem**: The tracking system 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 5.1.1.

Figure 5.1.1: Two examples of using our system



5.2 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.

5.2.1 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 5.2.1.

Listing 5.2.1: 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'
```

5.2.2 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 5.2.2:

Listing 5.2.2: 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 5.2.3.

Listing 5.2.3: 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 [57] 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) [88] 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 [63], 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 [38] (a lightweight OpenGL utility library) and GLAD [42] (hardware-specific OpenGL drivers) for the programming interface used in rendering 2D and 3D vector graphics with OpenGL [90].

5. **Tracking Models:** We download and configure Dlib [53] and MediaPipe [58], 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 the Appendix.

5.2.3 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 5.2.4.

Listing 5.2.4: 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 5.2.5.

Listing 5.2.5: Terminal

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

5.2.4 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 [62], 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 [8], 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.

5.3 Rendering System

5.3.1 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.

5.3.2 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 [90] 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 [82] or Unreal Engine [26], 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 [35] mathematical library for matrix manipulation and projections, as it was sufficiently fast and simple to use.

5.3.3 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. 5.3.1 and Fig. 5.3.2.

Figure 5.3.1: Right Perspective

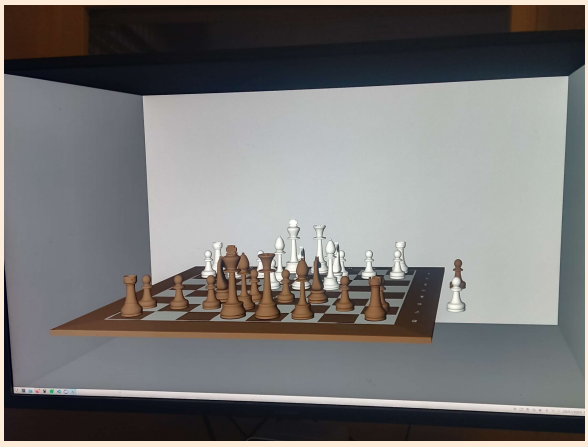
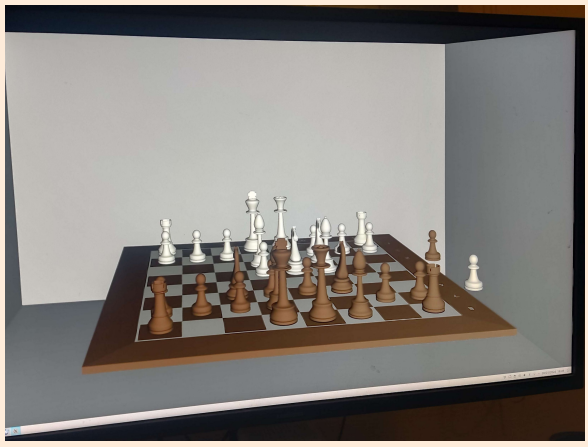


Figure 5.3.2: Left Perspective



5.3.4 Object Loading

Object loading support was added to the rendering system to facilitate the rendering of complex 3D models. We used the tinyobjloader library [80] to load `.obj` object files. We chose

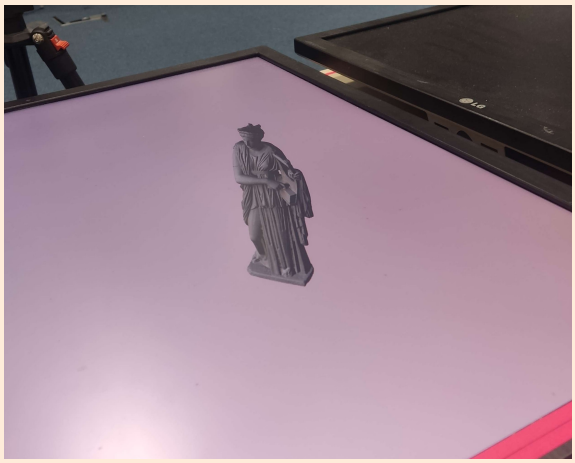
this library over alternatives like Assimp [4] due to its lightweight nature. We constructed our challenge for the user study by modifying primitives such as spheres, cylinders, and cubes loaded as `.obj` files using tinyobjloader to create an interactive task for the user.

5.3.5 Lighting

As shadows enhance the illusion of depth [48], we added a simple lighting model to the scene. We employed the Blinn-Phong [12] 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 [60] in Fig. 6.1.13 and Fig. 5.3.4, respectively.

Figure 5.3.3: Erato

Figure 5.3.4: Erato With Shadows



5.4 Tracking System

5.4.1 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.

5.4.2 Hardware

Figure 5.4.1: Azure Kinect [5]

For this project, we used a Microsoft Azure Kinect camera (Fig 5.4.1). 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.

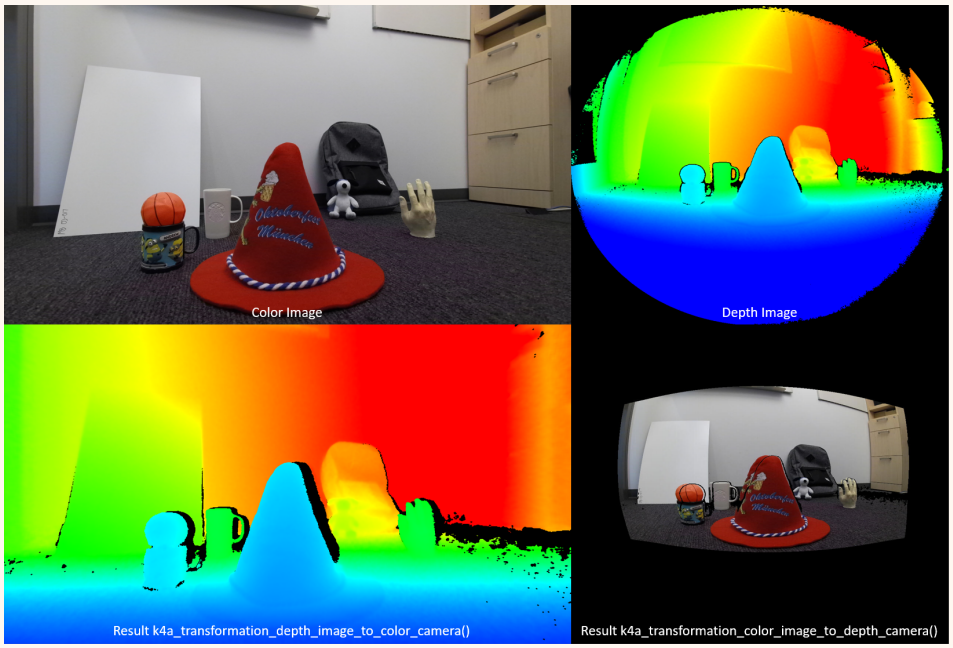
5.4.3 Core Libaries

Azure Kinect SDK (K4A)

We utilize the Azure Kinect SDK (K4A) [63] 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 5.4.2. 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 5.4.2: Different Spaces: Depth and Color Images [84]



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” [49] can occur, as visible in the bottom left image in Fig 5.4.2. 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 [13] 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 [53] 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 [58] 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.

5.4.4 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).

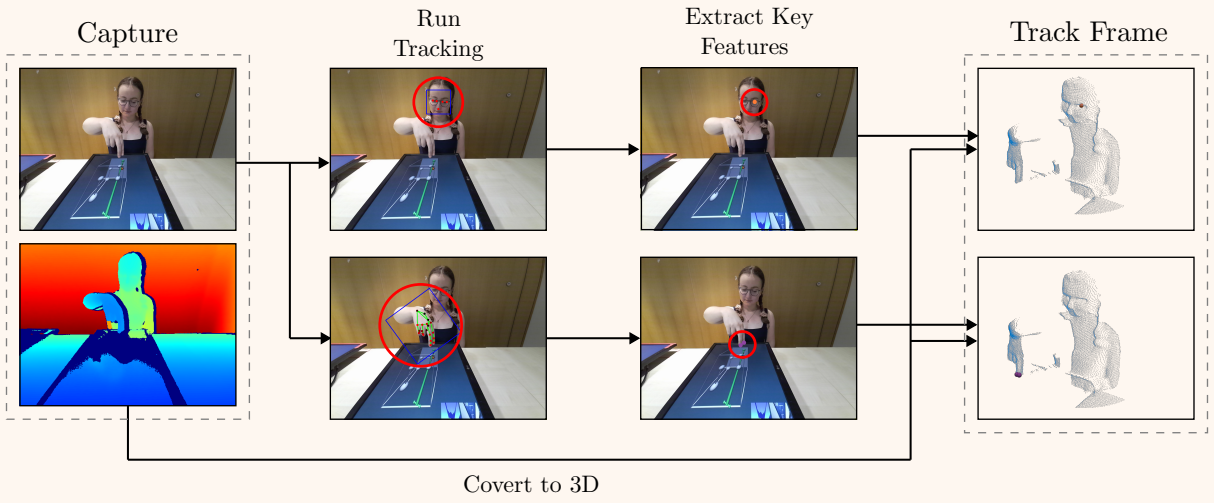
Several different approaches were considered for the tracking system design. Ultimately, we chose to use a method that tracks the positions of the user's face and hands separately using 2D color images. These images are then processed and converted into 3D points.

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 5.4.3: Tracker Overview



As illustrated in Fig 5.4.3, 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.

5.4.5 Tracking Models

Dlib

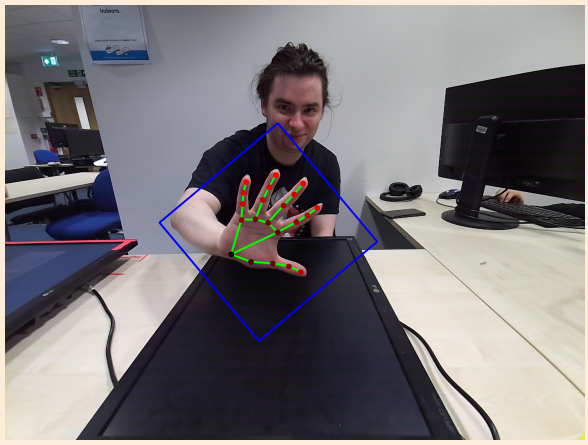
We use Dlib to track the left eye's position through a two-stage process. In the first stage, we utilize a Max-Margin Object Detection model [54, 20] implemented with a convolutional neural network (CNN) [72]. Instead of training our own model, we used a pre-trained model provided by Dlib [21], known 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" [46], trained on the iBUG 300-W face landmark dataset [71]. 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 5.4.4.

Figure 5.4.4: Dlib Face Tracker

Figure 5.4.5: 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 [92]. 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 5.4.5.

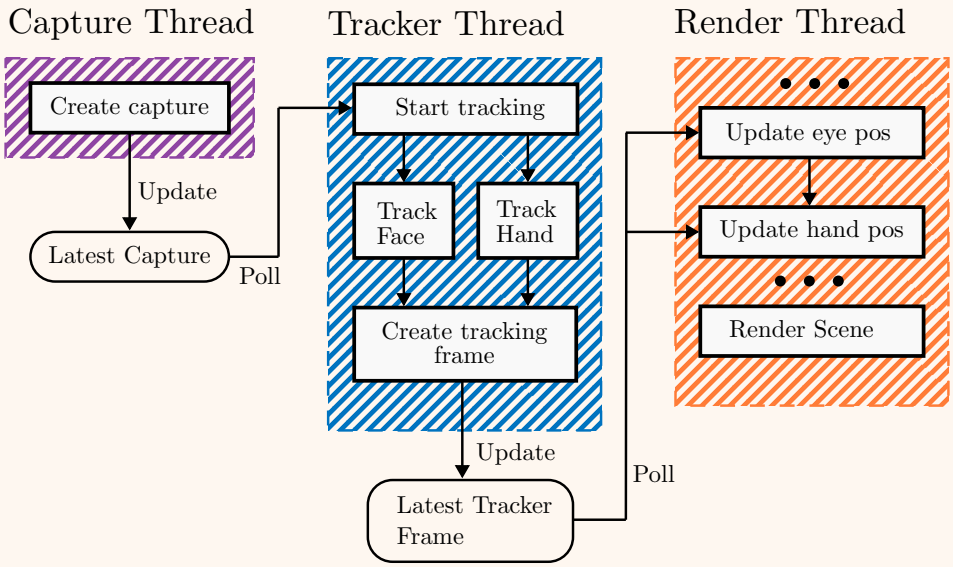
Downscaling

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.

5.4.6 Multithreading

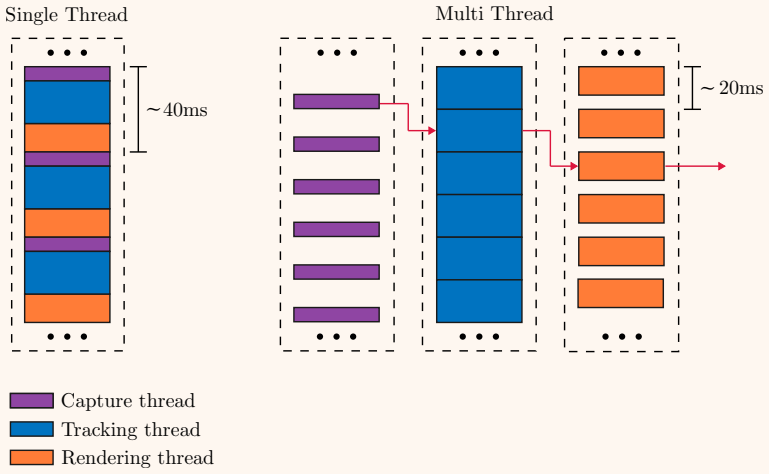
One of the more challenging aspects of this project was making our tracking system performant enough to feel smooth and responsive. To achieve this, we implemented a multi-threaded design, as outlined in Fig 5.4.6. We used separate threads for tracking, capturing, and rendering.

Figure 5.4.6: 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 5.4.7: 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 5.4.7. This also allowed us to run the simulation at a

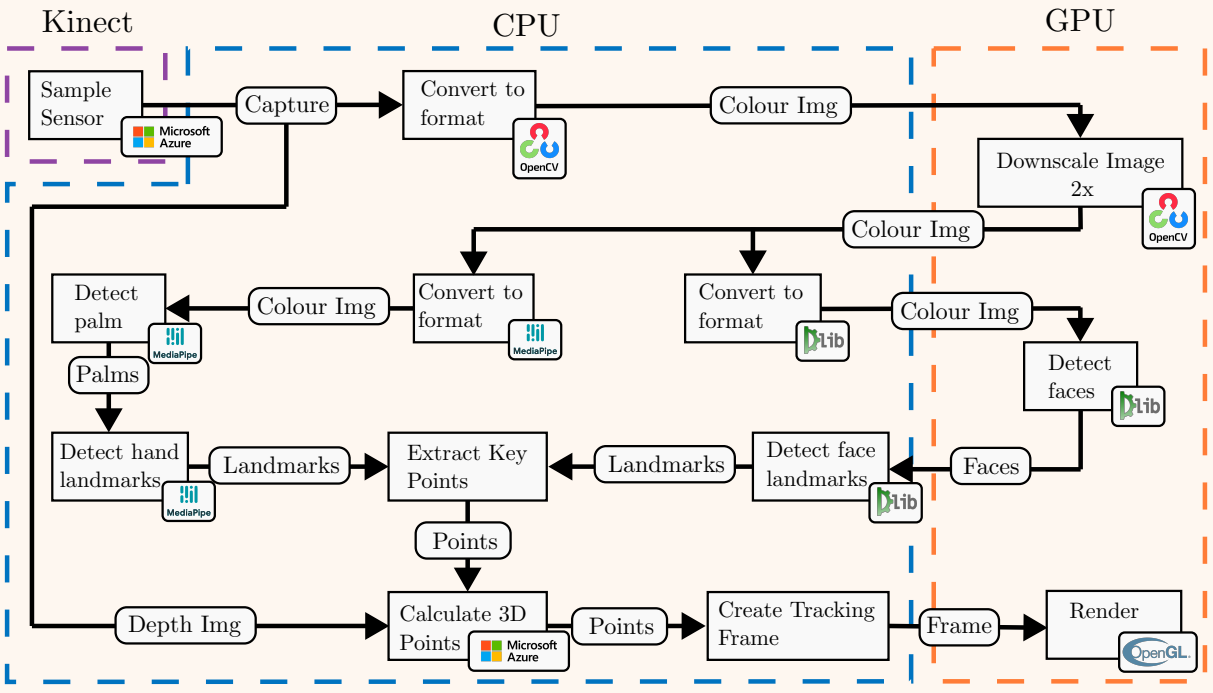
frame rate independent of the tracker. Although this design resulted in the capture thread often being idle, the overall system was light on resources, making this trade-off worthwhile for the significant frame rate improvement it provided.

5.4.7 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 [41] 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 5.4.8, 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 5.4.8: Overall Tracking System Design



5.4.8 Camera Positioning

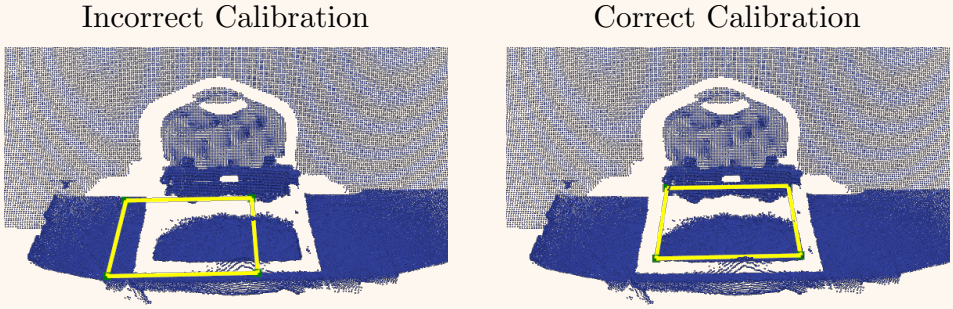
To ensure that the tracking system is calibrated correctly and that the user sees the correct perspective, it is crucial to know the relative position of the camera to the screen. Misalignment can lead to a distorted or incorrect user experience, where objects may appear to be

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 5.4.9.

Figure 5.4.9: Display Calibration



5.5 User Study

5.5.1 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.

5.5.2 Experimental Variables

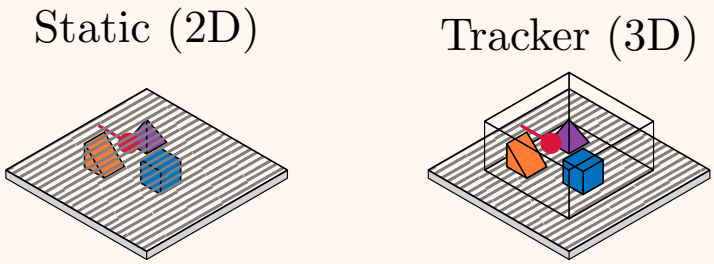
We aimed to test the following two hypotheses:

- **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:

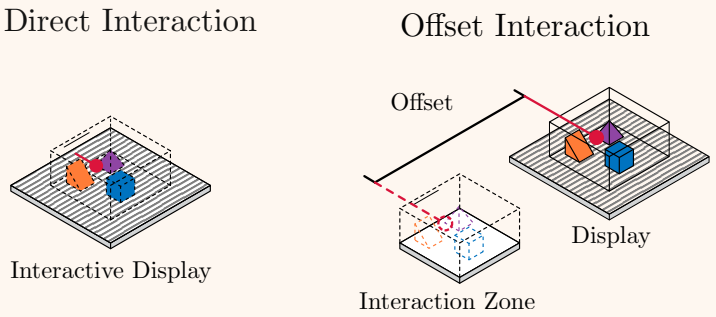
- **Perspective:** `Static` vs. `Tracker`. This variable controls whether the system uses a tracking mechanism to create the illusion of a 3D volumetric display (`Tracker`) or a fixed perspective on a standard monitor (`Static`), as shown in Fig 5.5.1. This tests **H1**.

Figure 5.5.1: 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 5.5.2. This tests **H2**.

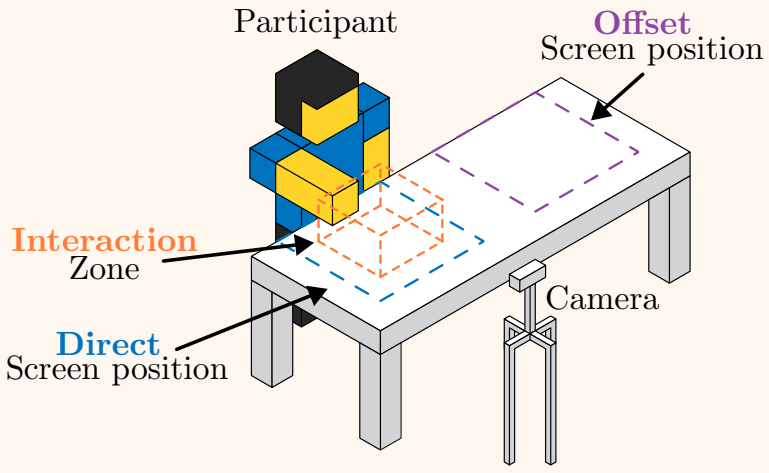
Figure 5.5.2: Direct & Offset Interaction



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 5.5.3. 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 5.5.3: Test Setup



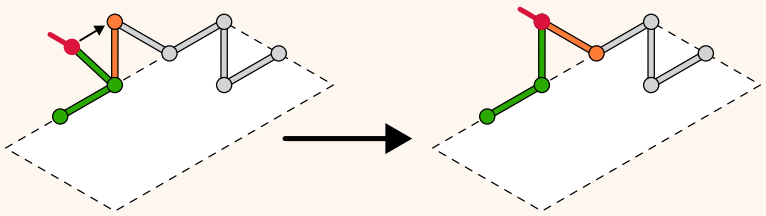
5.5.3 Tasks

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

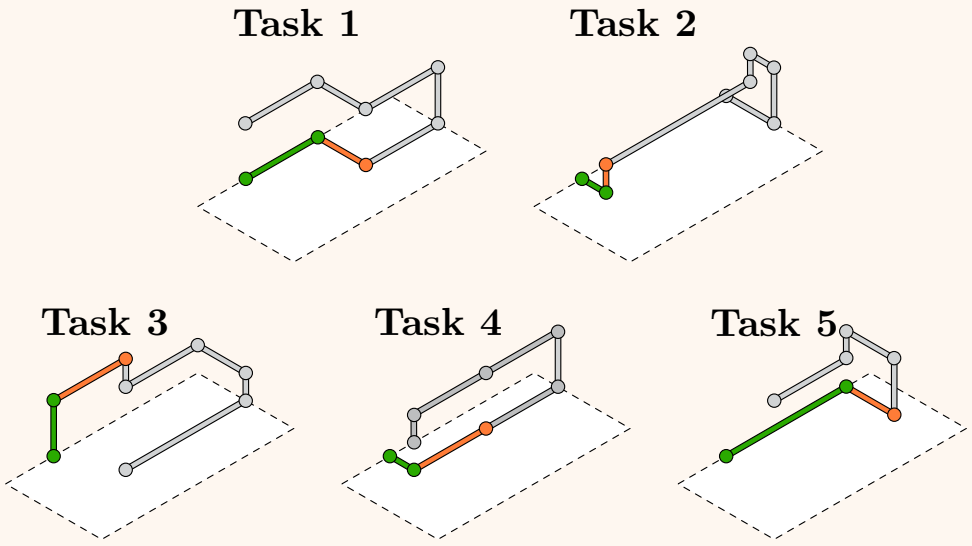
segment, an orange point represents the next segment to be completed, and a red point shows the current position of the participant's hand. Each time a task segment is completed, the colors update accordingly. An example of a completed task can be seen in Fig 5.5.4.

Figure 5.5.4: Completing a Task



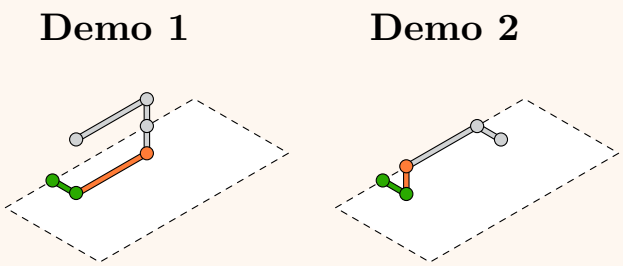
Participants have a time limit of one minute to complete each task. The time at which each point is completed, as well as the position of the hand and eye throughout the task, is recorded. If participants do not complete the task within the time limit, the task is marked as incomplete, and all completed segments are logged. The five different tasks are shown in Fig 5.5.5.

Figure 5.5.5: The Five Tasks



Participants are also given two demonstration tasks to familiarize themselves with the system, as seen in Fig 5.5.6.

Figure 5.5.6: Demo Conditions



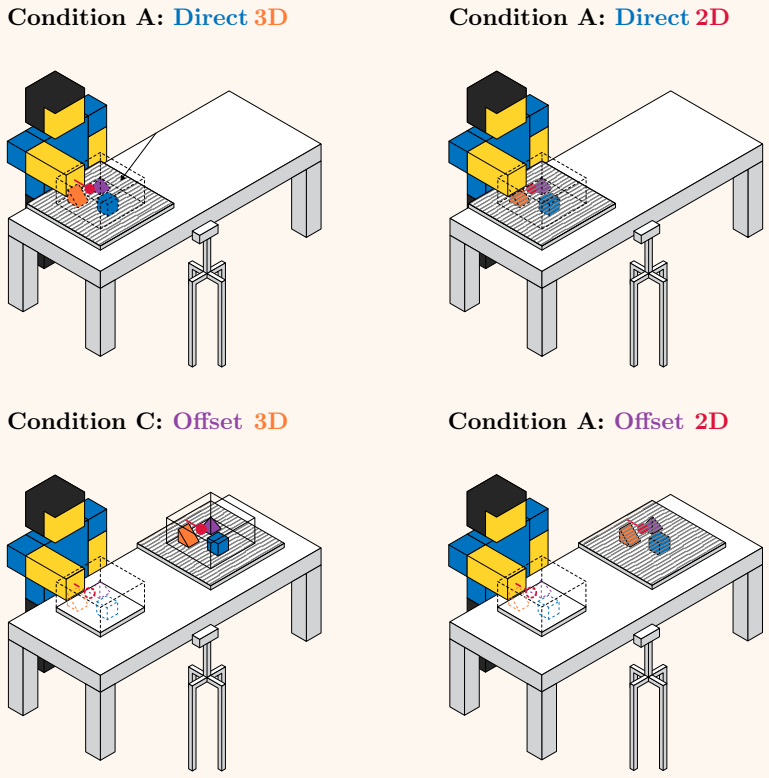
5.5.4 Participants

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 5.5.7.

Figure 5.5.7: Study Conditions



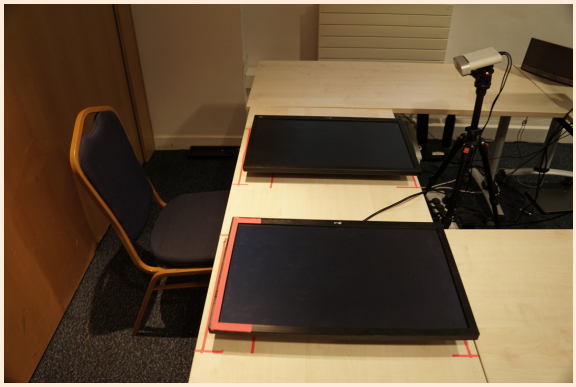
Each condition comprised five tasks, each lasting one minute, which participants were required to complete sequentially. An audio cue signaled the start and end of each task. Participants were allowed to take breaks between conditions. After completing each condition, they filled out a survey regarding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

5.5.5 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 5.5.8, 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 5.5.8: Study: Front View

Figure 5.5.9: 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 5.5.9.
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 5.5.10. 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 5.5.10: Study: Control View

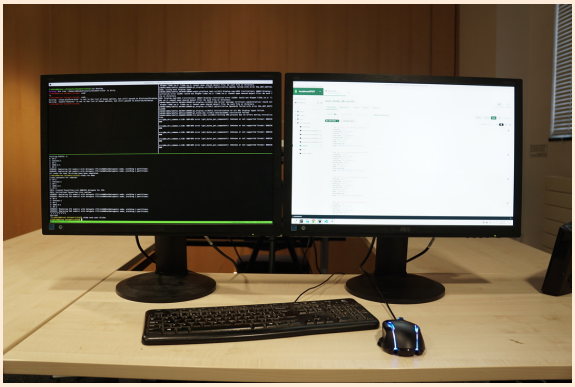
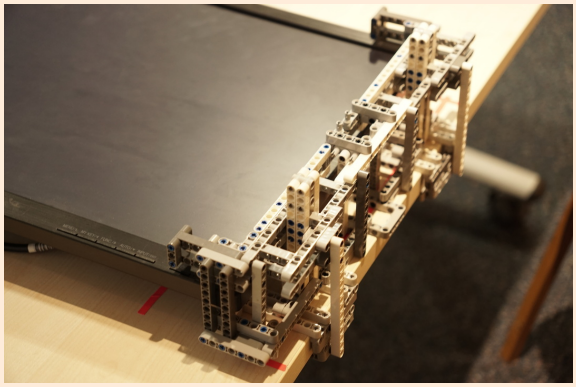


Figure 5.5.11: Study: Calibration Device



We also designed and built a calibration device to realign the displays after each participant completed their tasks, as depicted in Fig 5.5.11. Participants often inadvertently moved the displays during the tasks, and this simple device, made from Lego Technic (A line of Lego inter-

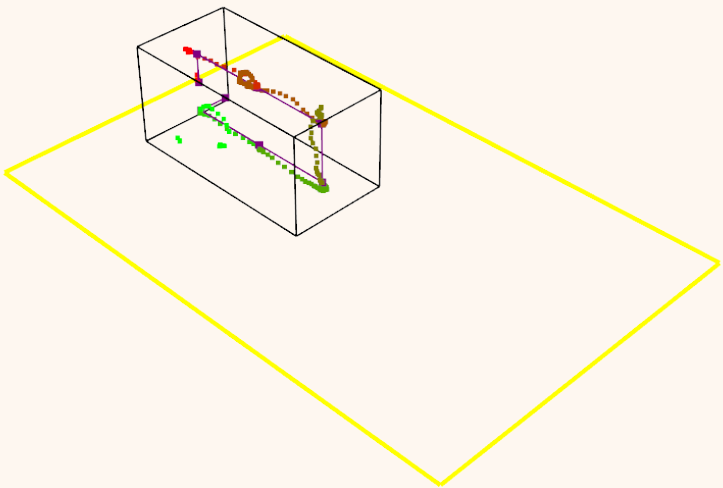
connecting plastic rods and parts, which is useful for creating a variety of mechanical systems), enabled us to easily and accurately reposition the displays to their original alignment.

5.5.6 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 5.5.12, illustrating the trajectory of a user's finger movements during Task 4.

Figure 5.5.12: 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.

5.5.7 Study Implementation

The study was run from a python based CLI using the Click [66] 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 5.5.13.

Listing 5.5.13: Terminal

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

Chapter 6

Evaluation

6.1 Simulator Evaluation

To ensure that our system meets the quality standards necessary for research purposes and accurately simulates a volumetric display, we conducted a comprehensive evaluation using various metrics.

6.1.1 Overall System

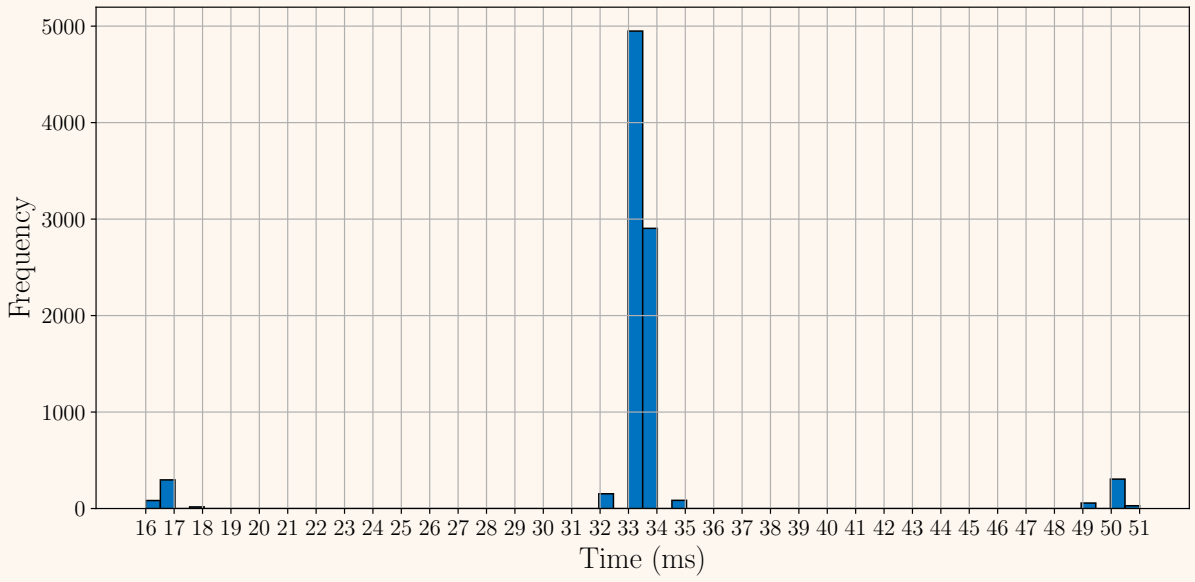
The initial step in our evaluation process involved assessing the performance of the system. TODO don't edit this part

1. Benchmark the system, including GPU and CPU usage using tools such as HTOP and NVIDA-SMI.

6.1.2 Tracking System: Frame Rate, Latency, and Timings

To gain deeper insights into the performance and quality of the tracking system, we conducted several tests. The primary focus was on the inter-update gap, defined as the time interval between consecutive tracker frames being sent to the renderer.

Figure 6.1.1: Tracker Inter-Update Gap (over 5 mins)



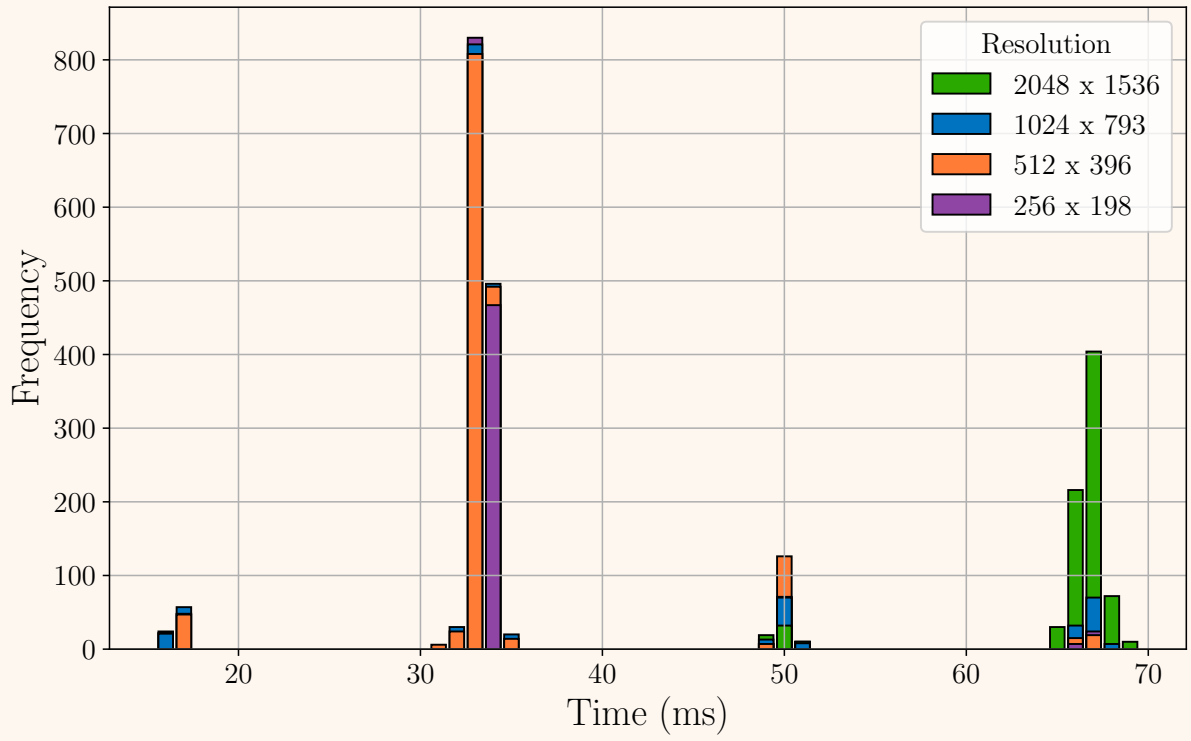
As depicted in Figure 6.1.1, the system's frame rate remained relatively consistent, with 90.83% of frames maintaining a latency between 30 and 35 ms. A peculiar observation was the cluster of latencies at 16-18 ms (4.46%) and 49-51 ms (4.39%). This anomaly is attributed to the frame rate of the renderer, as our system performs lazy conversion of 2D points to 3D only when necessary. Our renderer operates at 60 fps, constrained by our monitors, resulting in frame intervals approximating to 16.666 ms. Although our tracker functions at

30 fps (33.33 ms inter-update gap), there can be an inter-update gap of 16 ms if the tracker is delayed past the initial 33.33 ms, causing an additional delay of approximately 16.66 ms, resulting in a total gap of around 50 ms. However, during this waiting period, another frame is processed and ready for the next cycle, giving the appearance of a higher frame rate than 30 fps.

Downscaling Benchmarks

In exploring the impact of downscaling on performance, we determined that a single instance of downscaling sufficed to meet our performance requirements. Further downscaling did not yield significant performance benefits and adversely affected tracking quality. Figure 6.1.2 illustrates our findings, showing a noticeable performance improvement when reducing the resolution from the source 2048×1536 to 1024×793. Additional reductions in resolution did not provide further speed benefits, as the bottleneck was the 30 fps frame rate of the Kinect camera.

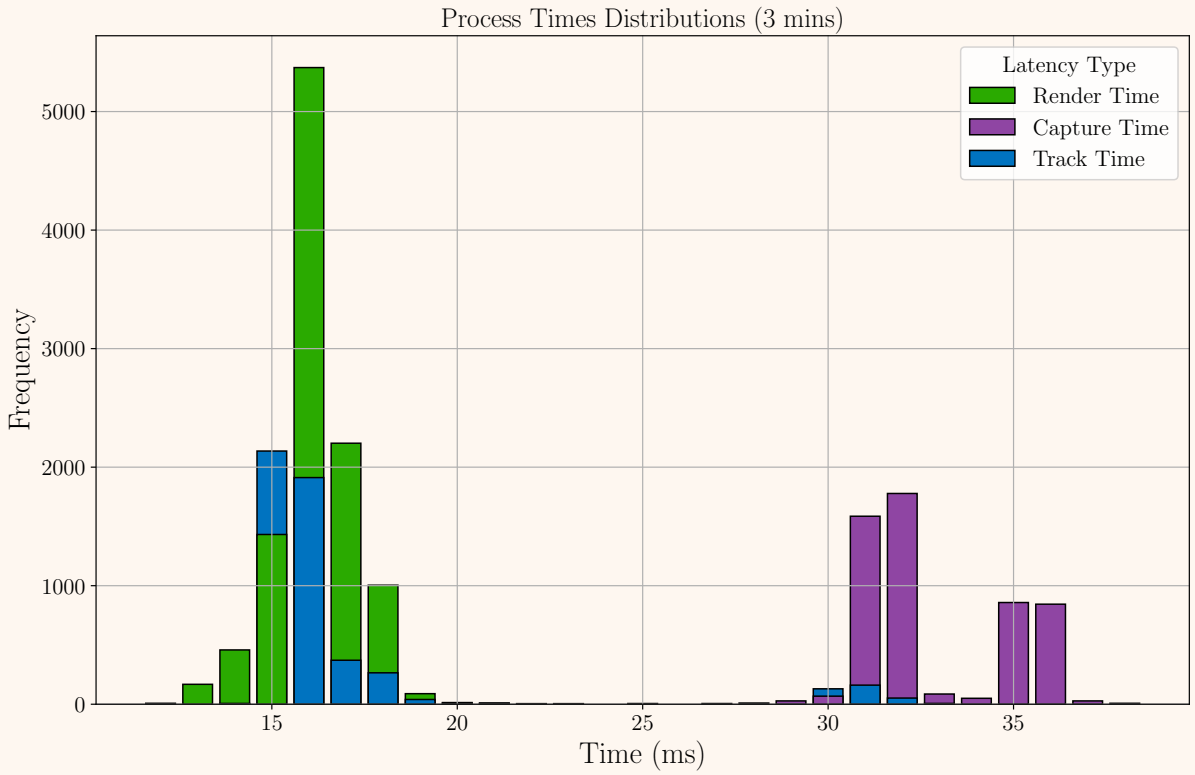
Figure 6.1.2: Comparing Inter-Update Gap By Resolutions (over 1 min)



Timings by Subprocess

Further investigation revealed that among the three concurrent threads (capturing from the Kinect camera, tracking hand and eye movements, and rendering with OpenGL), the primary bottleneck was waiting for the Kinect camera to complete the capture. Figure 6.1.3 highlights that expediting the tracking algorithm offered limited benefit due to this bottleneck.

Figure 6.1.3: Comparing Thread Competition Times



Latency from Camera to Eye

Although precise latency statistics for the Azure Kinect from capture to device are unavailable, estimates suggest a latency of around 90 ms [37]. Considering the time required for OpenGL primitives to display images on the screen, which ranges from 10 to 30 ms depending on the display [25] [18], and combining this with our system's processing time of less than 30 ms, we estimate the total system latency to be approximately 150 ms.

When compared to state-of-the-art systems in similar fields such as VR, our system exhibits higher latency, as shown in Table 6.1.

Table 6.1: Our Photo-to-Photon Latency Vs Common VR Systems reported by OptoFidelity [3]

System	Latency (ms)
`Volumetric Simulator (Ours)`	150 ms
`HTC VIVA XR Elite`	40 ms
`Meta Quest 3`	39 ms
`Meta Quest Pro`	38 ms
`Apple Vision Pro`	11 ms

It is challenging to directly compare our system with other similar systems, such as the Multi-person Fish-Tank Virtual Reality Display [29] [89] or systems used in 3D Display Simulation

Using Head-Tracking with Microsoft Kinect [91], as they do not report their latencies. However, given that both systems utilize Kinect cameras, which are a significant source of latency, we can infer that our system likely has comparable latency to these systems.

6.1.3 Tracking System: Accuracy

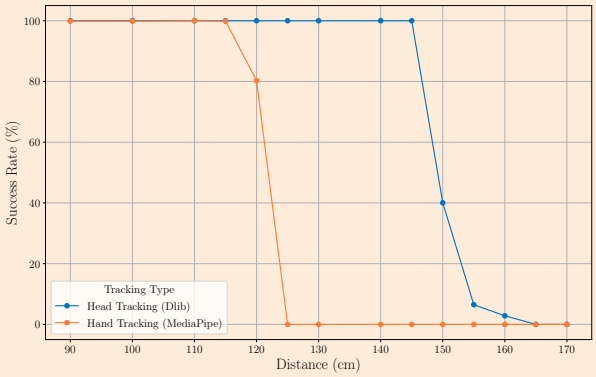
Evaluating the accuracy of our tracking system is crucial to ensure its reliability and effectiveness.

Effective Range

The first aspect of the evaluation focused on the effective tracking range of our system. We placed a participant in a fixed position on a chair and instructed them to slowly wave their hand and move their head. An example of the test setup is shown in Figure 6.1.4. We conducted tests at various distances, recording the percentage of successful captures that detected a face or hand at 30-second intervals. The resolution of the color image used for this test was 1024×793.

Figure 6.1.4: Setup for Distance Testing

Figure 6.1.5: Success Rate at Varying Distances



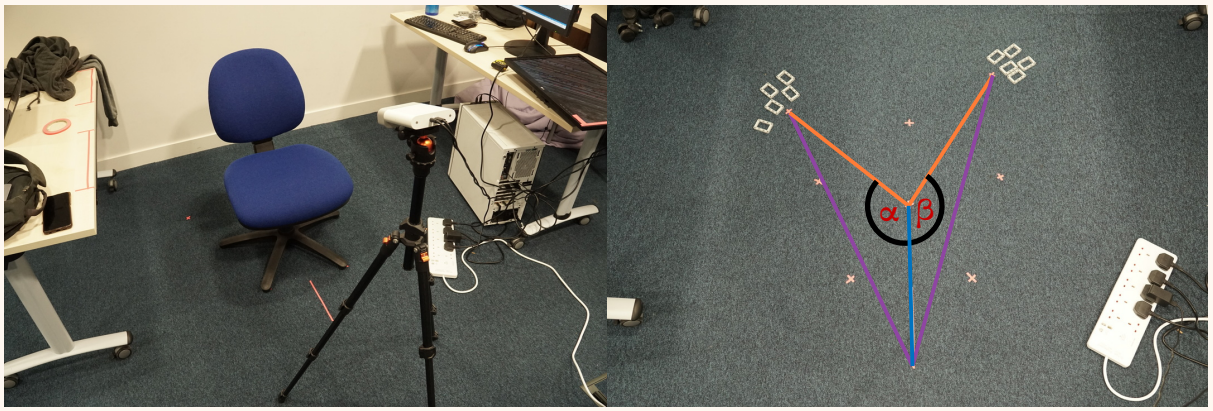
As illustrated in Figure 6.1.5, the success rate for MediaPipe's hand tracking model rapidly decreases after 1 meter, ultimately failing 100% of the time at greater distances. This is likely due to the dataset it was trained on and its primary design for tracking hands using a mobile phone camera [53].

Dlib's head tracking model performed better, with effective tracking up to approximately 1.5 meters. The decline in performance beyond this distance is similarly attributed to the nature of its training data. These findings were considered in the design of our user study, where the camera was positioned to keep the user's hand within 30-70 cm and the head within 1 meter from the camera.

Head Tracking

The effectiveness of the head tracking system is another critical aspect, particularly the range of head positions at which tracking is possible. Specifically, we were interested in the angle at which the model fails to detect a face. We set up a straightforward experiment, shown in Figure 6.1.6, to measure the head angles at which the tracking system ceases to function. Participants were seated on a swivel chair and asked to keep their body still while rotating their head. We noted the position of the midpoint of their feet when tracking failed, and used basic trigonometry to estimate the head angle.

Figure 6.1.6: Angle Setup



This test involved five participants, with results presented in Figure 6.1.7. Although the technique was somewhat rudimentary and thus not highly precise, the angle at which head tracking typically failed ranged between $120°$ and $150°$. This result may seem counterintuitive, as at these angles, the participant is facing away from the camera. However, the face tracking model first detects a face and then maps landmarks to it, even when the face is not directly visible.

Figure 6.1.7: Angle of Failure for Head Tracking

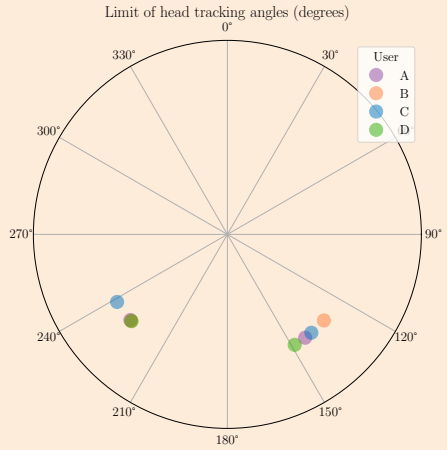
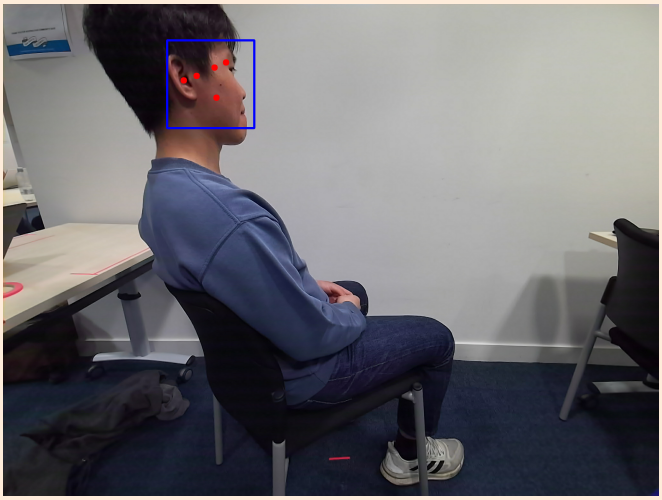


Figure 6.1.8: Incorrect Head Tracking



As shown in Figure 6.1.8, the face tracking model can still detect a face even when it is not visible, such as when detecting the side of the participant's head. The model tends to fail when only hair is visible. Testing on a bald participant could determine whether the system always detects a head regardless of rotation. While this might seem problematic, it is not, as the tracker and display are not visible to the user if their face is not visible. This also allows the system to approximate the position of the eyes, facilitating a smoother transition as the user turns their head back into view, minimizing the need for corrections and reducing noticeable adjustments for the user.

Hand Tracking

During the evaluation of our hand tracking model, we observed that it was significantly less reliable compared to our head tracking model, even when tested on the same images. Based on the metrics recorded during our user study, as illustrated in Fig 6.1.9, the hand tracking model failed for some users up to 10% of the time, whereas the eye tracking model rarely experienced failures. When participants were surveyed about this during our user study they echoed our observations as can be seen in Fig 6.1.10 with a noticeable drop in the rating of hand tracking reliability. This issue was particularly prominent when the model attempted to initially detect the hand. We found that the model was sensitive to specific conditions: it required a well-lit environment, a plain background, and the removal of extraneous objects from the scene. Users wearing t-shirts with intricate patterns experienced a notable decrease in tracking success. To address this, we provided a plain jumper for users to wear if we identified this as a potential problem.

We considered mitigating this issue by employing a segmentation model to separate the hand from the background before tracking. However, we did not have sufficient time to experiment with this solution, and we were concerned that it might significantly degrade the system's latency performance.

Figure 6.1.9: Failure rates during user study

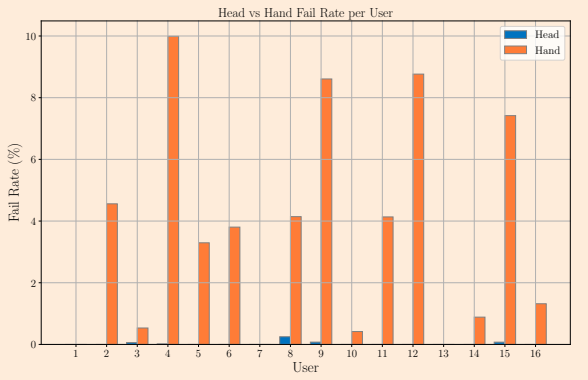
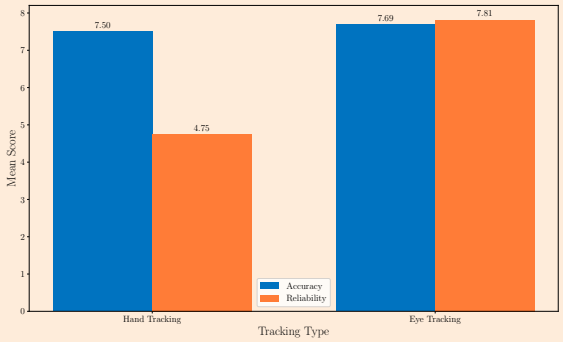


Figure 6.1.10: Accuracy and Reliability of Hand and Eye Tracking Survey



A major challenge of using a depth sampling method is its inability to handle occlusion effectively. Although Mediapipe can predict the position of hand points even when they are not directly visible, our method struggles to sample their 3D positions accurately in such cases. This is problematic because the hand often occludes itself, as demonstrated in Fig 6.1.11, where fingers are occluded by the palm, leading to erroneous depth sampling of the palm instead. Initially, we were concerned that this might also affect glasses wearers. However, we discovered that the depth sampling method, which uses infrared (IR) technology, was able to penetrate through glass.

Figure 6.1.11: Occlusion in Sampling

There are methods to mitigate these issues, such as sampling from points that are not occluded and using the known depth of the hand to infer the depth of the occluded points. This is a complex problem, and we did not have sufficient time to develop a solution that was both accurate and fast. Additionally, we found that the relative depth values estimated by Mediapipe were not very accurate, complicating interaction. We believe that the most effective solution would be to switch to a point cloud-based tracking model [73]. Our solution was to switch to interaction using the middle and index fingers, the two fingers we found to be most reliable.

6.1.4 Renderer

In evaluating our rendering system, we focused primarily on the quality and accuracy of the images it produced.

6.1.5 Quality

To evaluate the quality of our rendering system, we rendered a variety of complex 3D objects, including a chess set, a Minecraft house, and a protein structure. The images produced by our system were of high quality, as shown in Figures 6.1.12, 6.1.13, 6.1.14, and 6.1.15. The system was able to render these objects with high fidelity, accurately representing their 3D structure. The images were sharp and detailed, with precise lighting and shading. The system effectively rendered complex objects with numerous details, such as the protein structure, without any noticeable loss of quality or drop in frame rate. The images produced by our system were comparable to those produced by professional rendering software, demonstrating the high quality of our system.

Figure 6.1.12: Chess Set [17]

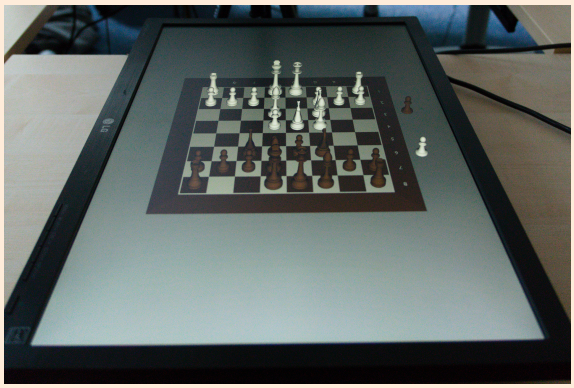


Figure 6.1.13: Erato [60]

Figure 6.1.14: Minecraft House [60]

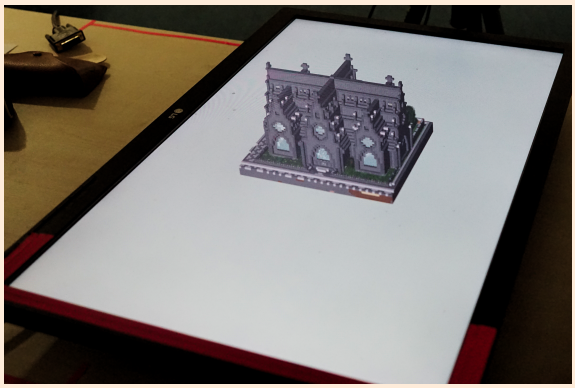
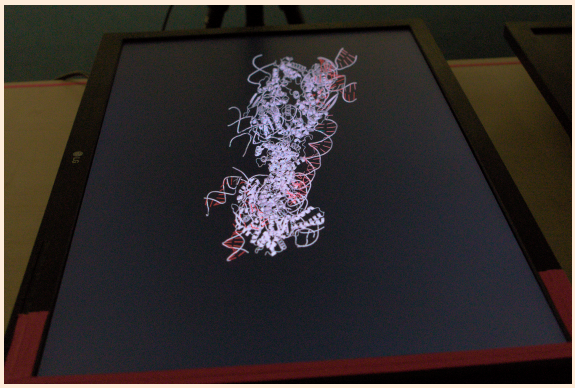


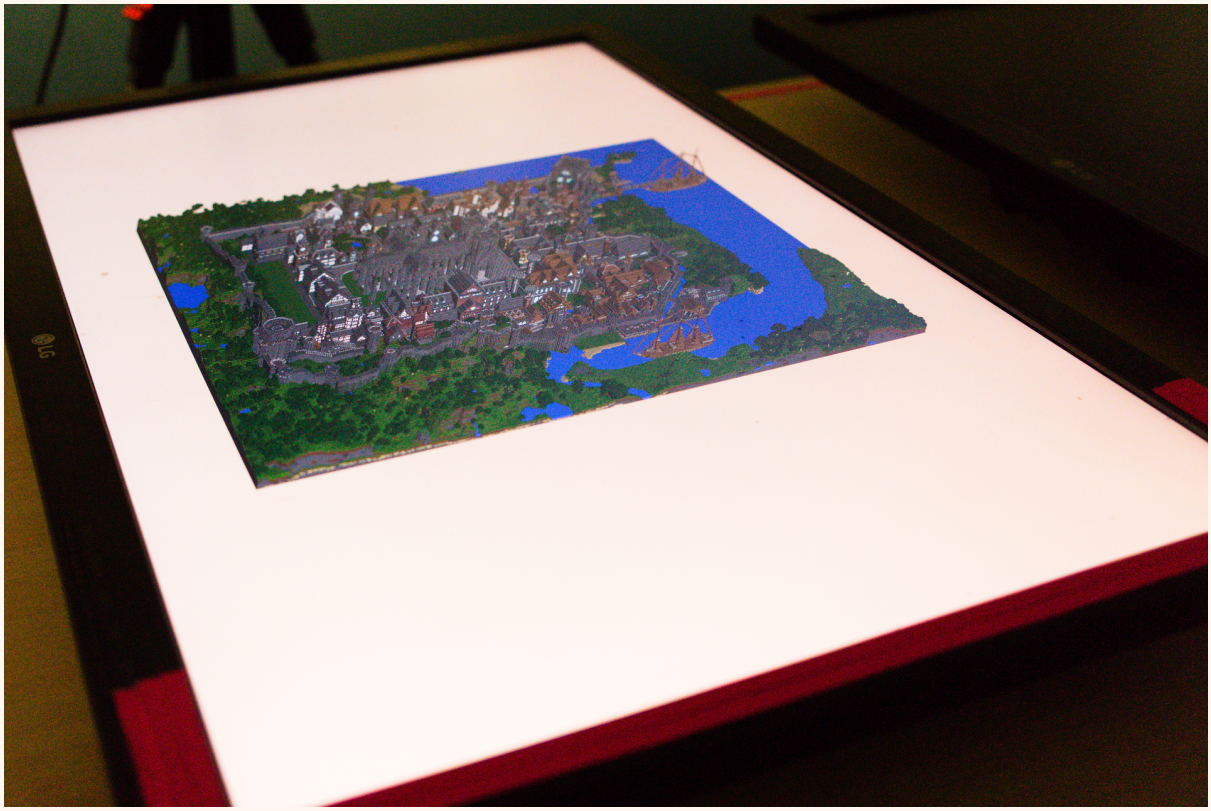
Figure 6.1.15: Retron-Eco1 filament with ADP-ribosylated Effector [7]



We successfully loaded and rendered a 6 million triangle model of Rungholt, a medieval city that was destroyed by a storm surge in the 14th century and recreated in Minecraft, as shown

in Figure 6.1.16. The model was rendered with no frame rate issues, demonstrating the system's ability to handle large and complex models with ease.

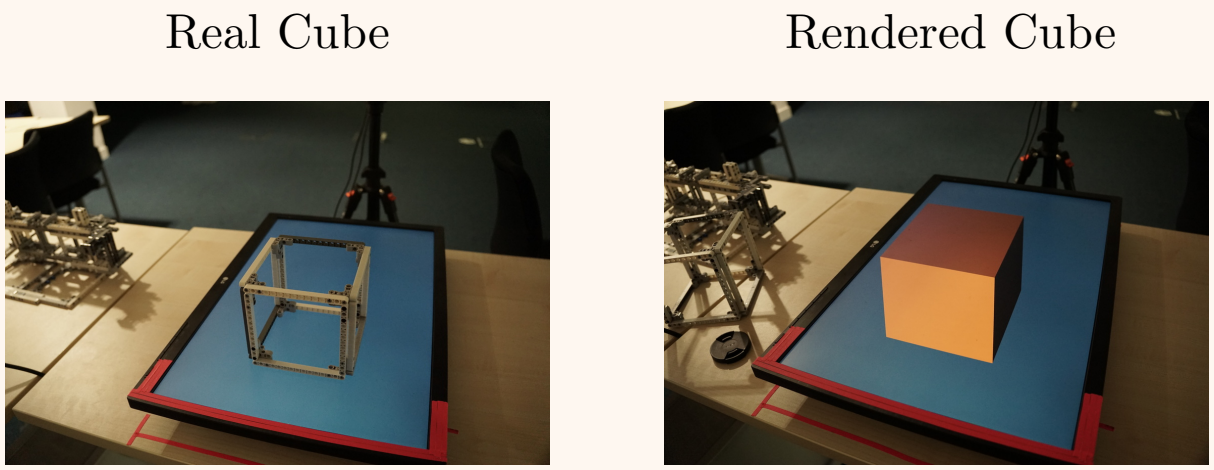
Figure 6.1.16: Rungholt [60]



Accuracy

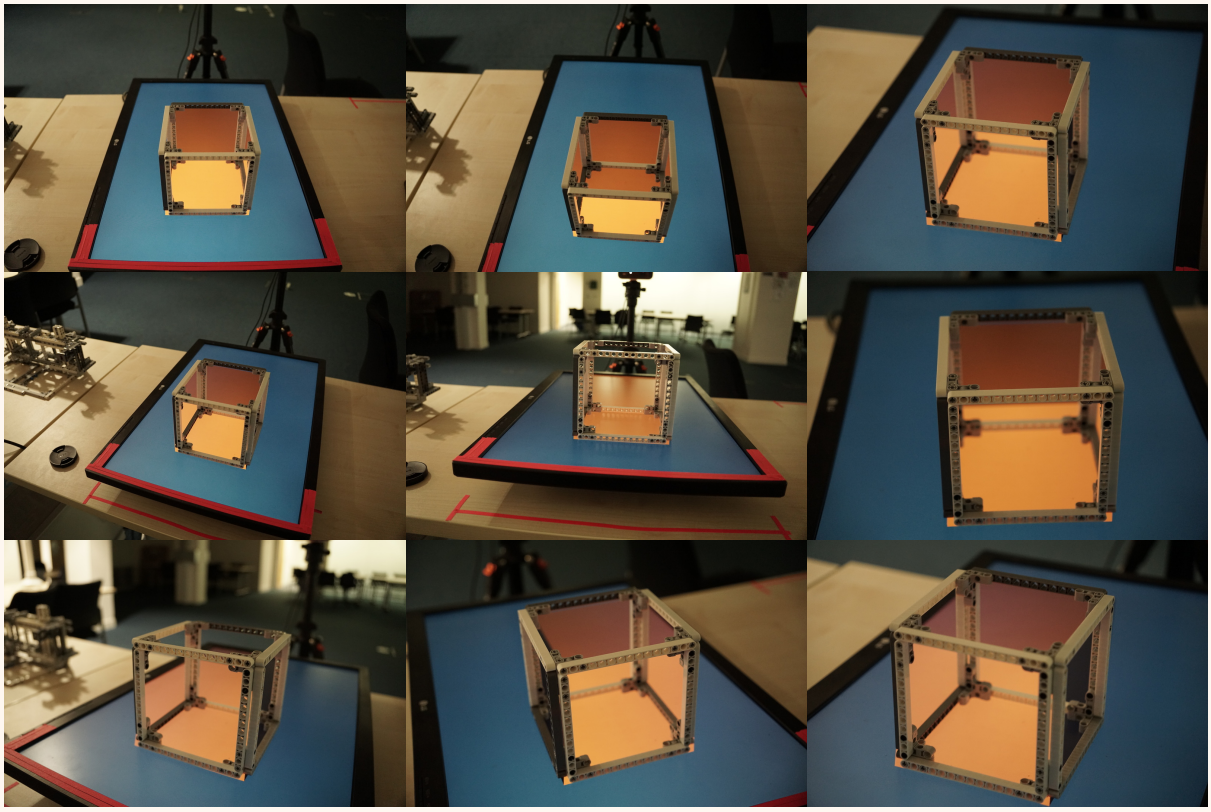
To evaluate the accuracy of our rendering system, we aimed to determine how well it could recreate a real object in a virtual environment. We chose a cube as our test object due to its simplicity and ease of measurement. A physical cube was constructed using Lego, and a corresponding virtual cube was created in our simulator, both with the exact dimensions of 13.5 cm x 13.5 cm x 12 cm, as illustrated in Figure 6.1.17.

Figure 6.1.17: A real and rendered cube



Due to the hollow nature of the physical cube, we were able to compare the two cubes effectively by superimposing them. We tested various perspectives and confirmed that the two cubes were indeed of the same size and shape, visually overlapping completely. Some example perspectives are presented in Figure 6.1.18. It is important to note that slight discrepancies in the images may occur, as the system was tracking the photographer's eye rather than the camera lens, which was positioned below and approximately 10 cm in front of the eye. Adjustments were made to account for this difference. This evaluation demonstrates that our rendering system is accurate and capable of recreating 3D objects from the real world with high precision.

Figure 6.1.18: Superimposed real and rendered cube

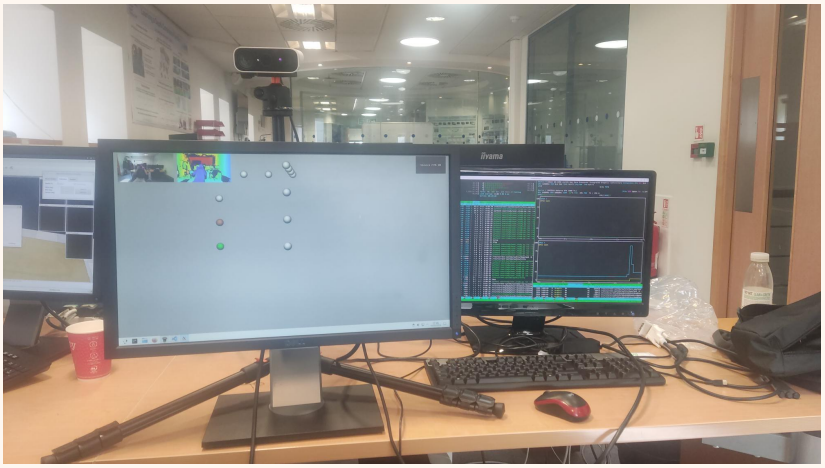


It is worth noting that like all 3D rendering systems based on a 2D display medium, our system is subject to the limitations of dimensions of the screen. This means any object that extends beyond the screen will be clipped, and the user will not be able to see the entire object at once.

6.1.6 Portability

One of the significant advantages of this system is its reproducibility and ease of deployment. It can be set up on a fresh Linux machine from scratch with a single command. The system was developed on a Linux machine (NixOS) using an Intel i5-9600K CPU and an NVIDIA GeForce RTX 2070 Super GPU. For testing purposes, the system was also run on a different Linux machine (NixOS) equipped with an Intel i7-4770K CPU and an NVIDIA GeForce GTX 1080 GPU, as depicted in Fig. 6.1.19. Remarkably, the system executed flawlessly on the first attempt on this different hardware configuration.

Figure 6.1.19: Running on a different machine



We also tested the system on a Windows machine using Windows Subsystem for Linux 2 (WSL2). Although we successfully built the system, the OpenGL renderer did not function as expected. We did not have sufficient time to diagnose the issue thoroughly, but we suspect it is related to WSL GPU passthrough limitations. This issue is likely resolvable with additional troubleshooting.

Currently, the system only supports machines with Intel CPUs and NVIDIA GPUs. However, extending support to AMD CPUs and GPUs should be feasible with further development and fairly trivial modifications to the build system.

6.1.7 Comparison to Other Systems

The field of volumetric displays is relatively niche, but there are few comparable systems available if expand our criteria a little. Table 6.2 provides a comparison of our system to other systems based the various attributes. Our system is unique in that as far as we are aware it is the only system that supports single-camera glasses free eye tracking with integrated hand tracking. Our system is still lacking in some areas, such as multi-user support and stereo rendering, but these are areas that could be addressed in future work but would require the introduction of some form of polarized or shutter glasses.

System	Single Camera	Hand Tracking	Custom Display	Glasses	Stereo	Multi-User
Volumetric Simulator (**ours**)	True	True	False	False	False	False
CoGlobe [94] [93] [28] [29]	False	False	True	True	True	True
OrbeVR [9] [32]	False	False	True	True	True	True
pCubee [75] [76]	Headtracker	False	True	False	False	False
HandheldBall [10]	False	False	True	True	True	True
TeleHuman [52]	False	False	True	False	False	False
HoloDesk [43]	False	True	True	False	False	False
FaceTrack [78]	True	False	False	False	False	False
3DDisplaySimulation [91]	True	False	False	False	False	False

Table 6.2: Comparison of systems based on attributes

TODO

6.2 User Study Evaluation

We conducted our study as outlined in the Implementation section, aiming to investigate the following 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?

6.2.1 Participants

The study took place between the 1st and 5th of June in Room 218 of the Huxley Building at Imperial College London's South Kensington Campus. We were able to engage 16 participants in our study, 14 of whom consented to be photographed, as shown in Fig 6.2.1.

Figure 6.2.1: Participants

The recruitment of participants was primarily facilitated through existing networks, involving colleagues and acquaintances. Attempts to recruit participants through other channels proved less effective.

Consequently, our participant sample was not broadly representative of the wider population. The majority of participants were concentrated around the age of 22, as depicted in Fig 6.2.2, and the group was predominantly male, as illustrated in Fig 6.2.5.

We also surveyed our participants regarding factors we anticipated might influence the study. Notably, 37% of the participants wore glasses during the study (Fig 6.2.3), and almost every participant had prior experience with VR (Virtual Reality) or AR (Augmented Reality), as shown in Fig 6.2.4. Our analysis indicated that these variables did not significantly impact the study results.

Figure 6.2.2: Age

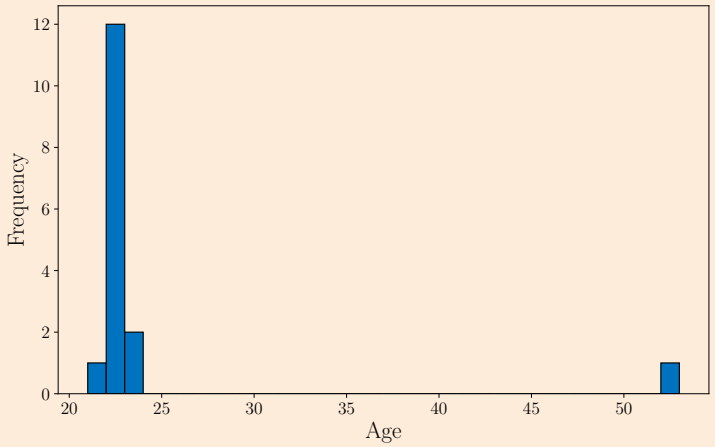


Figure 6.2.3: Glasses

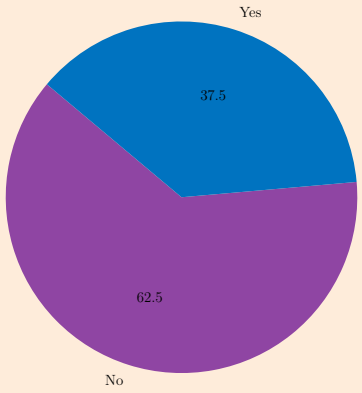


Figure 6.2.4: Used VR/AR

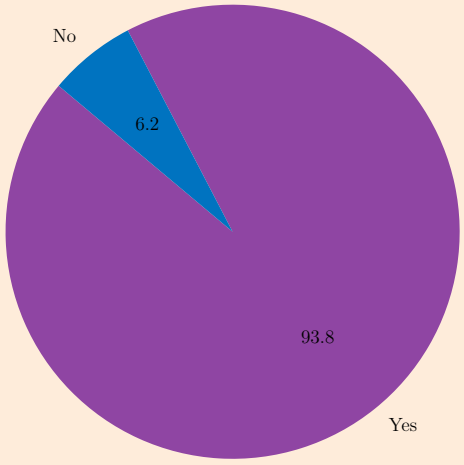
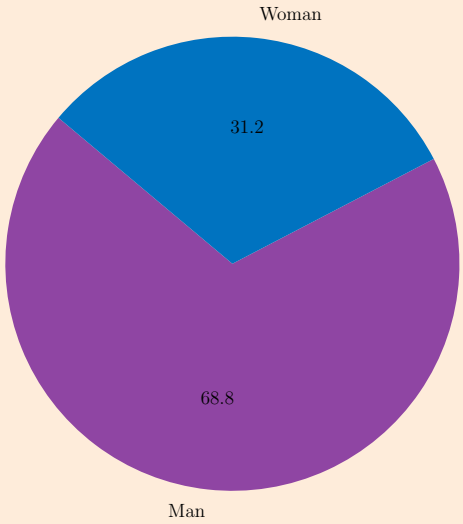


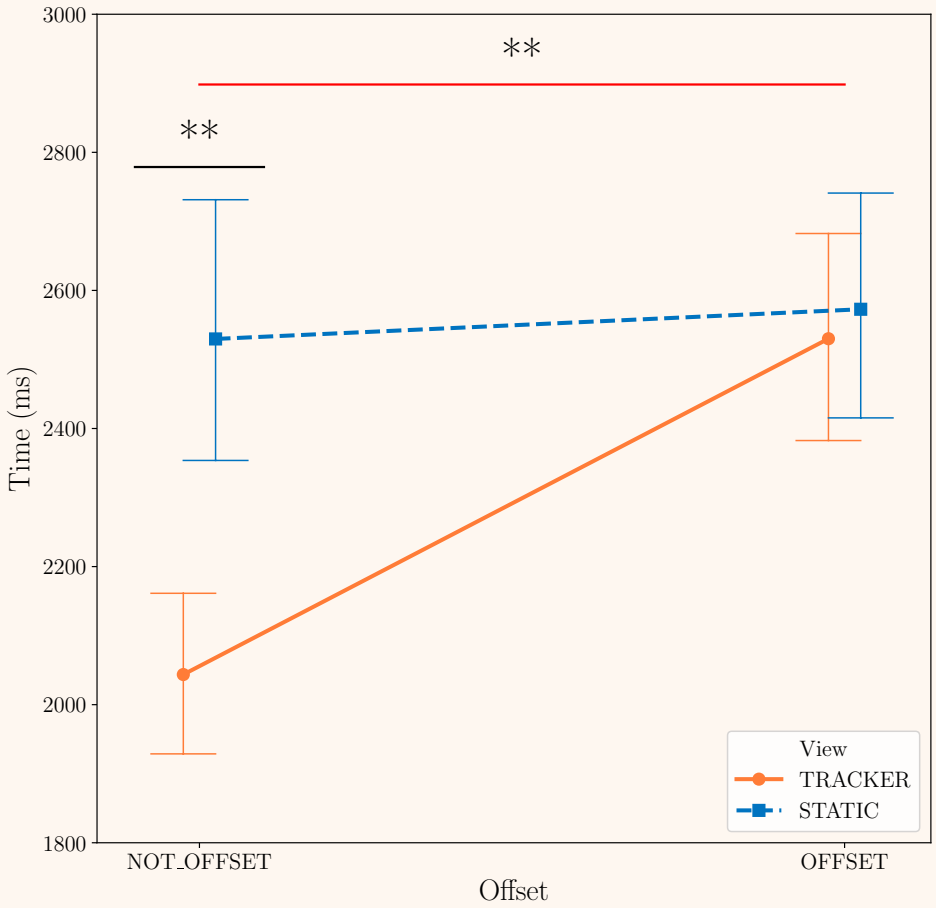
Figure 6.2.5: Gender



Results: Timings

Upon completing the study with all participants, we collected and analyzed the data, focusing particularly on the time taken to complete each segment of the tasks. Our analysis revealed that both the type of view (STATIC, i.e., 2D, versus TRACKER, i.e., 3D) and the presence of an offset (OFFSET versus NO OFFSET) had a statistically significant impact on the time required to complete the segments. We plotted the average segment completion time against these variables in Fig 6.2.6.

Figure 6.2.6: Interaction between View and Offset on Segment Completion Times



To confirm the significance of these results, we conducted a type II ANOVA test, the results of which are displayed in Table 6.3.

The data suggests that participants performed best when using the TRACKER mode (3D view) directly in front of them, completing segments in an average of 2043 ms. Performance deteriorated under other conditions, with average completion times of 2530 ms for STATIC, 2573 ms for STATIC OFFSET, and 2530 ms for TRACKER OFFSET, respectively. This indicates that the advantage of the 3D view is significant only when the display is directly in front of the

Table 6.3: ANOVA Results for Fig 6.2.6

Source	Sum of Squares	df	F	p-value
`C(View)`	$3.363\,186\times10^{7}$	1.0	10.698 212	0.001 092
`C(Offset)`	$3.341\,680\times10^{7}$	1.0	10.629 800	0.001 132
`C(View):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

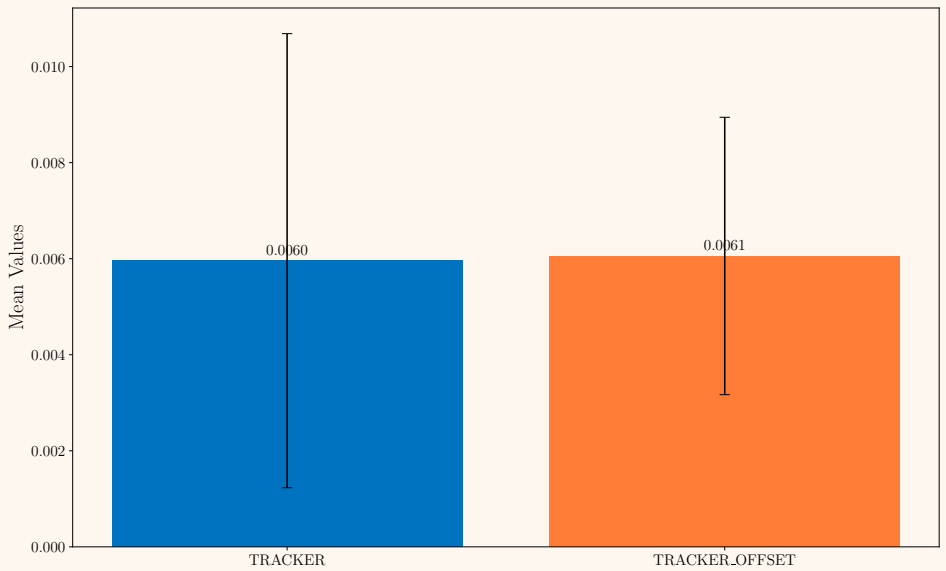
user, and this benefit diminishes when an offset is introduced.

We hypothesize that this phenomenon is due to motion parallax, an effect where closer objects appear to move more relative to farther objects. This effect is crucial for depth perception in the real world and contributes to the effectiveness of 3D displays. However, when the 3D display is offset, the motion parallax effect is reduced, diminishing the advantages of the 3D display.

Interestingly, the data suggests that the offset condition does not significantly impact the time taken to complete the segments, contrary to our expectations. We hypothesized that the offset would significantly affect task completion time, potentially because the user's hand might occlude their view when the display is directly in front. This hypothesis is supported by our observation that participants moved their heads equally, regardless of the offset condition, as shown in Fig 6.2.7. This suggests that participants were still attempting to obtain a better view of the segments even when they were offset.

To further investigate this phenomenon, it would be prudent to conduct additional studies examining multiple offset conditions with varying distances. This approach would enable us to measure more precisely the decline in the effectiveness of the TRACKER mode as the offset distance increases, providing further validation of our motion parallax hypothesis. It would also be insightful to try offsetting the interaction zone rather the display to check if this has an effect.

Figure 6.2.7: Combined Mean Eye Movement Values Per Millisecond and Standard Deviations



As demonstrated in Table 6.4, the t-test results indicate that the offset condition does not have a significant effect on eye or head movement.

Table 6.4: T-Test Results for Fig 6.2.7

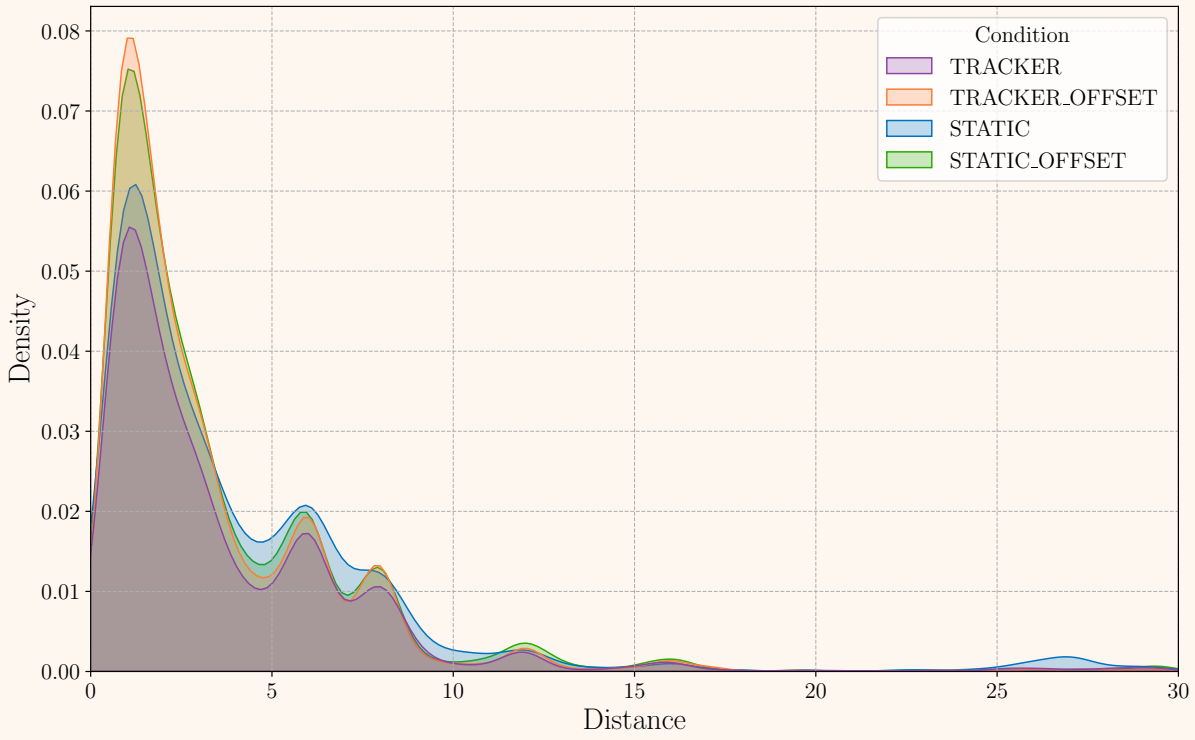
Statistic	Value	p-value
`T-statistic`	-0.4170	0.6767

Results: Hand Precision

Another important metric we investigated was the distance of the user's hand from the end of the segment they were attempting to select over time. This distance represents their error over time, with a value of 0 indicating that the user successfully overlapped their finger with the target, thereby completing the task. The distribution of these distances is illustrated in Fig 6.2.8.

We observed that in both the STATIC OFFSET and TRACKER OFFSET conditions, users exhibited significantly worse precision in selecting the point at the end of the segment. These conditions showed a much larger peak just before reaching a distance of zero, which is the target point that participants were trying to precisely select. Interestingly, the STATIC condition had a much smaller peak near completion, closely resembling the TRACKER condition, but was consistently worse than every other condition throughout the task.

Figure 6.2.8: Distribution of Hand Distances from Segment End



We hypothesize that the significantly worse performance in the OFFSET conditions is due to the lack of motion parallax, which would make it more challenging for users to judge the precise distance and depth between their hand and the end of the segment. This difficulty likely resulted in users spending more time at the end of a segment trying to precisely select the final point. Additionally, the display being further away may have made the task appear smaller and more difficult to gauge accurately.

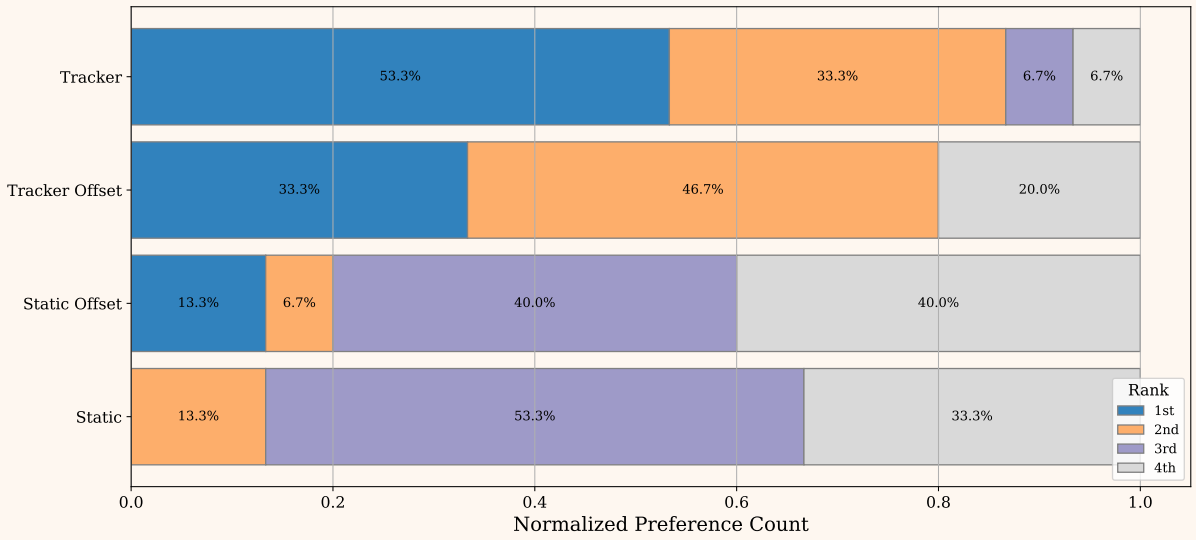
In the STATIC condition, users seemed to struggle more consistently with distance throughout the entire task. This difficulty is suggested by the small pronounced peak between 25-30 cm, which we hypothesize was called by users moving their hand in completely incorrect directions. We believe this may be due to the perspective in the STATIC condition being more challenging than in the STATIC OFFSET condition, as users were positioned above and behind the display rather than above, behind, and to the side. Each segment being oriented along the x, y, or z axis may have led to more overlapping segments in the STATIC condition, complicating the user's ability to discern the correct direction. Although this issue was also presumably present in the TRACKER conditions, users could mitigate it by changing their perspective.

It is noteworthy that the observed peaks in the distance distributions are likely due to the time it takes for users to realize they have completed a segment and need to start moving towards the next point. We are confident in this interpretation, as the peaks align closely with the lengths of the segments (e.g., 2 cm for one segment, 3 cm for eight segments, 6 cm for twelve segments, 8 cm for ten segments, 12 cm for three segments, and 16 cm for one segment).

6.2.2 Survey Results

Another source of our evaluation was a survey conducted after the study. The survey aimed to gather feedback on the users' experiences and preferences regarding the different conditions they encountered. Participants were asked to rank the conditions in order of preference, as shown in Fig 6.2.9.

Figure 6.2.9: Study Condition Rankings



An overwhelming 87% of participants chose either the TRACKER or TRACKER OFFSET condition as their top preference, indicating a strong preference for conditions that involved tracking. Interestingly, no participants ranked the STATIC condition as their first preference, despite it not being significantly slower than the TRACKER OFFSET condition according to the timing data we recorded.

We hypothesize that the strong bias against the STATIC and STATIC OFFSET condition may be due to the perceived naturalness and familiarity of the tracking conditions. Participants may have found these conditions more intuitive, possibly because they provided more visual information, even though this did not necessarily translate into improved task performance.

To analyze the preference rankings statistically, we performed a Friedman test. The results, presented in Table 6.5, show a significant difference in the rankings of the conditions, with a p-value of 0.00162.

These findings underscore the participants' preference for conditions that simulate a more dynamic and interactive experience, which they may have found more engaging or easier to understand.

Table 6.5: Friedman Test Results for Condition Preferences (Fig 6.2.9)

Statistic	Value	p-value
`Friedman Test Statistic`	15.240	0.001 62

Further Analysis

To avoid cluttering this section, graphs displaying the results of our study at a finer granularity can be found in the appendix. These graphs provide a detailed breakdown of participants' responses to different questions related to each task.

If have time add graph of different questions per task from excel

6.2.3 Discussion

The results of our study provide valuable insights into the impact of display and interaction modalities on task performance and user preference in volumetric environments.

Hypothesis Testing

- **H1**: There is a significant difference in task performance between 3D (TRACKER mode) and 2D (STATIC mode) interactions. Participants completed tasks faster and with higher precision in the 3D TRACKER condition when the display was directly in front of them, suggesting that 3D views enhance spatial understanding due to better depth cues like motion parallax. The advantage of 3D views diminishes with an offset.

- **H2**: Task performance varies significantly between direct hand interaction and teleoperation but only when the view was in 3D. Direct hand interaction in the 3D TRACKER condition resulted in superior performance, highlighting the effectiveness of natural hand movements in spatial tasks. The OFFSET conditions introduced challenges, suggesting that indirect manipulation impacts task accuracy.

Precision and Task Complexity

OFFSET conditions significantly hindered precision, suggesting that reduced motion parallax makes it difficult to judge distances. The STATIC condition's poorer performance, indicates challenges with depth perception and segment orientation in 2D views.

User Preferences

There is a strong preference for TRACKER (3D) conditions, highlighting the subjective value of intuitive and interactive interfaces. The significant difference in rankings, as shown by the Friedman test, emphasizes the importance of user experience in technology adoption.

Implications for Future Research and Design

Future research should explore more variations of offset conditions and task complexities. In particular the impact of offsetting the interaction zone rather than the display should be investigated.

Designers of Volumetric Display applications and experiences should prioritize direct hand interaction to enhance user experience and task performance. This is a difficulty for volumetric displays as almost always permeable/tangible so this not always achievable.

Chapter 7

Future Work, Conclusions and Contributions

7.1 Future Work

We have identified several areas for future work that could enhance the capabilities and usability of our volumetric display simulator and further explore the potential of volumetric displays in interactive 3D applications.

7.1.1 Anaglyph 3D

Anaglyph 3D [19] is a technique for displaying 3D images using color-filtered glasses, typically employing red and green filters. Unlike polarized 3D [51], it does not require additional complex hardware. Currently, the 3D effect necessitates closing one eye. Integrating 3D support into the system could enhance the immersive experience by eliminating this limitation.

7.1.2 Multi-User Support

Another potential area for future exploration is multi-user support. Our current tracking system is limited to a single user. Extending it to support multiple users would be a logical progression and relatively straightforward. By utilizing color filter glasses or shutter glasses, it is feasible to render different perspectives to multiple users simultaneously, as suggested in the study ”Two Kinds of Novel Multi-user Immersive Display Systems” [40].

7.1.3 Real-Time Light Detection

Adding an additional camera with a fisheye lens to generate a real-time light map could be a valuable enhancement. This feature would enable the virtual scene to be illuminated by real-world lighting conditions. It would be insightful to investigate whether this addition impacts performance in any significant way, especially concerning the tasks evaluated in our user study.

7.1.4 Generalize CPU/GPU Camera Compatibility

The current project is compatible only with Nvidia GPUs. Expanding compatibility to include AMD GPUs, Intel GPUs, and even CPU-only environments (with expected slower performance) would be beneficial. Furthermore, supporting different depth cameras beyond the Kinect, such as Intel RealSense Depth Cameras [50], is crucial, particularly since the Kinect has been discontinued by Microsoft. Achieving this will require substantial code generalization. Switching cameras to a lower latency system would also be beneficial.

7.1.5 Switch Hand Tracking Model

The hand tracking component of the project is notably the weakest aspect. Transitioning to a more robust hand tracking model is highly recommended. Specifically, adopting a model that utilizes depth images rather than RGB images could significantly improve performance. This would likely be a major undertaking, as off-the-shelf models for this purpose are scarce. Implementing the approach described in the paper ”Accurate, Robust, and Flexible Real-Time Hand Tracking” [73] appears promising.

7.1.6 Further User Study

As indicated in the evaluation section, while the user study provided conclusive results, further investigation is warranted. We are interested in exploring various offset positions to examine the drop-off rate in greater detail. Additionally, adjusting the position of the interaction zone, as opposed to the display position, could yield valuable insights.

7.1.7 Porting to Windows and Mac

Although we have demonstrated that the project can be built with a single command on Linux, extending support to Windows (including WSL) and Mac would be advantageous. We have successfully compiled the project on Windows, but further investigation is required to address WSL-specific issues. We have yet to attempt building it on Mac. This is expected to be more challenging, as porting all GPU-accelerated functionality to Apple Metal [61] may pose significant difficulties, despite Nix's native compatibility with Mac.

7.2 Conclusions and Contributions

The development and evaluation of our Volumetric Display Simulator underscore its potential as a versatile tool for future research in volumetric display technologies. By integrating cost-effective components and reproducible software environments, we have created a system that is both accessible and straightforward, encouraging wider adoption and further development by other researchers.

Our user study demonstrated that the use of head tracking and direct hand interaction significantly enhances task performance in a 3D environment, highlighting the importance of natural interaction modes for effective use of volumetric displays. The findings suggest promising avenues for future research, including the exploration of positional offsets and interaction zone dynamics, which could lead to significant improvements in the usability and functionality of volumetric displays.

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] *Apple Vision Pro Benchmark Test 1: See-Through Latency, Photon-to-Photon*. Feb. 14, 2024. URL: https://www.optofidelity.com/insights/blogs/apple-vision-pro-benchmark-test-1-see-through-latency-photon-to-photon (visited on 06/12/2024).

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

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

[6] *BAE Systems backs local tech company with global defence capability*. BAE Systems | Australia. URL: https://www.baesystems.com/en-aus/article/bae-systems-backs-local-tech-company (visited on 06/15/2024).

[7] RCSB Protein Data Bank. *RCSB PDB - 8QBK: Retron-Eco1 filament with ADP-ribosylated Effector (local map with 1 segment)*. URL: https://www.rcsb.org/structure/8QBK (visited on 06/12/2024).

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

[9] O. R. Belloc et al. "OrbeVR: a handheld convex spherical virtual reality display". In: *ACM SIGGRAPH 2017 Emerging Technologies*. SIGGRAPH '17. Los Angeles, California: Association for Computing Machinery, 2017. ISBN: 9781450350129. DOI: 10.1145/3084822.3091104. URL: https://doi.org/10.1145/3084822.3091104.

[10] Francois Berard and Thibault Louis. "The Object Inside: Assessing 3D Examination with a Spherical Handheld Perspective-Corrected Display". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: Association for Computing Machinery, 2017, pp. 4396–4404. ISBN: 9781450346559. DOI: 10.1145/3025453.3025806. URL: https://doi.org/10.1145/3025453.3025806.

[11] 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.

[12] 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.

[13] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. ” O’Reilly Media, Inc.”, 2008.

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

[15] 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.

[16] 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.

[17] *Chessset wooden chess board with pieces | 3D model*. CGTrader. URL: https://www.cgtrader.com/free-3d-models/sports/toy/chessset (visited on 06/12/2024).

[18] John Cormack. *Latency mitigation strategies (by John Carmack)*. URL: https://danluu.com/latency-mitigation/ (visited on 06/12/2024).

[19] Dorra Dhaou, Saoussen Ben Jabra, and Ezzeddine Zagrouba. “A Review on Anaglyph 3D Image and Video Watermarking”. In: *3D Research* 10.2 (Mar. 2019), p. 13. ISSN: 2092-6731. DOI: 10.1007/s13319-019-0223-1. URL: https://doi.org/10.1007/s13319-019-0223-1.

[20] *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).

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

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

[23] 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.

[24] 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.

[25] Federico Dossena and Andrea Trentini. “OpenLDAT—A system for the measurement of display latency metrics”. In: *Journal of the Society for Information Display* 30.8 (2022), pp. 599–608. DOI: https://doi.org/10.1002/jsid.1104. eprint: https://sid.onlinelibrary.wiley.com/doi/pdf/10.1002/jsid.1104. URL: https://sid.onlinelibrary.wiley.com/doi/abs/10.1002/jsid.1104.

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

[27] European Parliament and Council of the European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council*. of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). OJ L 119, 4.5.2016, p. 1–88, May 4, 2016. URL: https://data.europa.eu/eli/reg/2016/679/oj (visited on 04/13/2023).

[28] 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.

[29] 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.

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

[31] 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.

[32] F. Ferreira et al. "Spheree: A 3D perspective-corrected interactive spherical scalable display". In: (July 2014). DOI: 10.1145/2614066.2614091.

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

[34] 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.

[35] *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).

[36] 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.

[37] Peyman Gholami and Robert Xiao. *AutoDepthNet: High Frame Rate Depth Map Reconstruction using Commodity Depth and RGB Cameras*. 2023. arXiv: 2305.14731 [cs.CV].

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

[39] Xing Gong et al. “Application of a 3D volumetric display for radiation therapy treatment planning I: quality assurance procedures”. en. In: *J Appl Clin Med Phys* 10.3 (July 2009), pp. 96–114.

[40] Dongdong Guan et al. “Two Kinds of Novel Multi-user Immersive Display Systems”. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI ’18. , Montreal QC, Canada, Association for Computing Machinery, 2018, pp. 1–9. ISBN: 9781450356206. DOI: 10.1145/3173574.3174173. URL: https://doi.org/10.1145/3173574.3174173.

[41] *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).

[42] 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).

[43] Otmar Hilliges et al. “HoloDesk: direct 3d interactions with a situated see-through display”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’12. Austin, Texas, USA: Association for Computing Machinery, 2012, pp. 2421–2430. ISBN: 9781450310154. DOI: 10.1145/2207676.2208405. URL: https://doi.org/10.1145/2207676.2208405.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] Leonid Keselman et al. *Intel RealSense Stereoscopic Depth Cameras*. 2017. arXiv: 1705.05548.

[51] Byeong-Cheol Kim. “A Study on the Technique of the 3D Stereoscopic Cinema”. In: *Journal of Korea Multimedia Society* 16 (Aug. 2013). DOI: 10.9717/kmms.2013.16.8.994.

[52] Kibum Kim et al. “TeleHuman: effects of 3d perspective on gaze and pose estimation with a life-size cylindrical telepresence pod”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’12. Austin, Texas, USA: Association for Computing Machinery, 2012, pp. 2531–2540. ISBN: 9781450310154. DOI: 10.1145/2207676.2208640. URL: https://doi.org/10.1145/2207676.2208640.

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

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

[55] Robert Kooima. “Generalized perspective projection”. In: (2009).

[56] 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.

[57] 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.

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

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

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

[61] *Metal Shading Language Specification*. URL: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf (visited on 06/14/2024).

[62] *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).

[63] *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).

[64] Joshua Napoli et al. “Radiation therapy planning using a volumetric 3-D display: PerspectaRAD”. In: (Jan. 2008).

[65] 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.

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

[67] 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.

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

[69] Scott D. Robinson and Patrick J. Green. “3D display applications for defense and security”. In: *Display Technologies and Applications for Defense, Security, and Avionics II*. Ed. by John Tudor Thomas and Andrew Malloy. Vol. 6956. International Society for Optics and Photonics. SPIE, 2008, 69560B. DOI: 10.1117/12.785009. URL: https://doi.org/10.1117/12.785009.

[70] 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.

[71] 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.

[72] 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.

[73] Toby Sharp et al. “Accurate, Robust, and Flexible Real-time Hand Tracking”. In: *CHI ’15 Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. Best of CHI Honorable Mention Award. ACM, Apr. 2015, pp. 3633–3642. ISBN: 978-1-4503-3145-6. URL: https://www.microsoft.com/en-us/research/publication/accurate-robust-and-flexible-real-time-hand-tracking/.

[74] 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.

[75] Ian Stavness, Billy Lam, and Sidney Fels. “pCubee: a perspective-corrected handheld cubic display”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, pp. 1381–1390. ISBN: 9781605589299. DOI: 10.1145/1753326.1753535. URL: https://doi.org/10.1145/1753326.1753535.

[76] Ian Stavness and Florian Vogt. “Cubee: A Cubic 3D Display for Physics-based Interaction”. In: (Jan. 2006). DOI: 10.1145/1179849.1180055.

[77] M. T. Stickland, S. McKay, and T. J. Scanlon. “The development of a three dimensional imaging system and its application in computer aided design workstations”. In: *Mechatronics* 13.5 (2003), pp. 521–532. ISSN: 0957-4158. DOI: https://doi.org/10.1016/S0957-4158(01)00052-6. URL: https://www.sciencedirect.com/science/article/pii/S0957415801000526.

[78] T. Suenaga et al. “Poster: Image-Based 3D Display with Motion Parallax using Face Tracking”. In: *Proceedings of the 2008 IEEE Symposium on 3D User Interfaces*. 3DUI ’08. USA: IEEE Computer Society, 2008, pp. 161–162. ISBN: 9781424420476. DOI: 10.1109/3DUI.2008.4476617. URL: https://doi.org/10.1109/3DUI.2008.4476617.

[79] 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).

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

[81] UK Parliament. *Equality Act 2010*. Publisher: Statute Law Database. URL: https://www.legislation.gov.uk/ukpga/2010/15 (visited on 06/14/2024).

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

[83] 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.

[84] *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).

[85] *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).

[86] *Voxon Photonics Media Kit*. Voxon Photonics. URL: https://voxon.co/community/ (visited on 06/15/2024).

[87] 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.

[88] 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.

[89] 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.

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

[91] 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.

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

[93] Qian Zhou et al. "3DPS: An auto-calibrated three-dimensional perspective-corrected spherical display". In: *2017 IEEE Virtual Reality (VR)*. 2017, pp. 455–456. DOI: 10.1109/VR.2017.7892376.

[94] Qian Zhou et al. "Coglobe: a co-located multi-person FTVR experience". In: *ACM SIGGRAPH 2018 Emerging Technologies*. SIGGRAPH '18. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2018. ISBN: 9781450358101. DOI: 10.1145/3214907.3214914. URL: https://doi.org/10.1145/3214907.3214914.

[95] 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.

Appendix A

Appendix

A.1 VolumetricSim Package

The result of building our simulator is a shared library named `libvolsim.so` shown in Listing A.1.1.

Listing A.1.1: 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
```

A.2 Form A: Overall Digital Survey Form

This form is the primary survey that was given to participants in the user study. It was used to collect data on the participants' demographics, their experience with technology, and their opinions on the volumetric display system.

Volumetric Display User Study

Study Companion Forum

* Required

Participant Info

How much do you agree with the statement on a scale of 1 to 10

1. First Name *

2. Last Name *

3. Gender *

- ◯ Woman
- ◯ Man
- ◯ Non-binary
- ◯ Prefer not to say
- ◯ Other

4. Do you wear glasses *

- ◯ Yes
- ◯ No

5. Have you used VR/AR before? *

- ◯ Yes
- ◯ No
- ◯ Not sure

6. Age *

The value must be a number

7. Handedness (which is your dominant hand) *

- ◯ Left
- ◯ Right
- ◯ Both
- ◯ Other

8. StudyID (this will be generated by Robbie) *

9. Email (if you would like to be updated about this study)

10. I completed the demo

- ◯ Once
- ◯ Twice
- ◯ Three times

Tracker Condition (View does change)

How much do you agree with the statement on a scale of 1 to 10

11. I found completing the tasks for this condition difficult *

1	2	3	4	5	6	7	8	9	10

12. I found this condition frustrating *

1	2	3	4	5	6	7	8	9	10

13. It was unclear in which direction I needed to move my hand next *

1	2	3	4	5	6	7	8	9	10

Static Condition (View does not change)

How much do you agree with the statement on a scale of 1 to 10

14. I found completing the tasks for this condition difficult *

1	2	3	4	5	6	7	8	9	10

15. I found this condition frustrating *

1	2	3	4	5	6	7	8	9	10

16. It was unclear in which direction I needed to move my hand next *

1	2	3	4	5	6	7	8	9	10

Tracker Offset Condition (View does change, hands are offset from view)

How much do you agree with the statement on a scale of 1 to 10

17. I found completing the tasks for this condition difficult *

1	2	3	4	5	6	7	8	9	10

18. I found this condition frustrating *

1	2	3	4	5	6	7	8	9	10

19. It was unclear in which direction I needed to move my hand next *

1	2	3	4	5	6	7	8	9	10

Static Offset Condition (View does not change, hands are offset from view)

How much do you agree with the statement on a scale of 1 to 10

20. I found completing the tasks for this condition difficult *

1	2	3	4	5	6	7	8	9	10

21. I found this condition frustrating *

1	2	3	4	5	6	7	8	9	10

22. It was unclear in which direction I needed to move my hand next *

1	2	3	4	5	6	7	8	9	10

Conclusion

23. The task was able to track my hand **accurately** *

1	2	3	4	5	6	7	8	9	10

24. The task was able to track my hand **reliably** *

1	2	3	4	5	6	7	8	9	10

25. The task was able to track my eye **accurately** *

1	2	3	4	5	6	7	8	9	10

26. The task was able to track my eye **reliably** *

1	2	3	4	5	6	7	8	9	10

27. Rank the conditions in order of prefrence

- **Tracker Condition** (View changes)
- **Static Condition** (View does not change)
- **Tracker Offset Condition** (View changes, offset)
- **Static Offset Condition** (View does not change, offset)

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.

Microsoft Forms

A.3 Form B: Inter-Condition Survey Form

This was the phyiscal survey that was given to participants in the user study. It was used to collect data on the participants' opinions between the different conditions of the volumetric display system. The results were copied into a digital form.

User: _____

Tracker Condition: (View does change)

(Circle one box for each)

I found completing the tasks for this condition difficult

1	2	3	4	5	6	7	8	9	10

I found this condition frustrating

1	2	3	4	5	6	7	8	9	10

It was unclear in which direction I needed to move my hand next

1	2	3	4	5	6	7	8	9	10

Static Condition: (View does not change)

(Circle one box for each)

I found completing the tasks for this condition difficult

1	2	3	4	5	6	7	8	9	10

I found this condition frustrating

1	2	3	4	5	6	7	8	9	10

It was unclear in which direction I needed to move my hand next

1	2	3	4	5	6	7	8	9	10

Tracker Offset Condition: (View does change, hands are offset)

(Circle one box for each)

I found completing the tasks for this condition difficult

1	2	3	4	5	6	7	8	9	10

I found this condition frustrating

1	2	3	4	5	6	7	8	9	10

It was unclear in which direction I needed to move my hand next

1	2	3	4	5	6	7	8	9	10

Static Offset Condition: (View does not change, hands are offset)

(Circle one box for each)

I found completing the tasks for this condition difficult

1	2	3	4	5	6	7	8	9	10

I found this condition frustrating

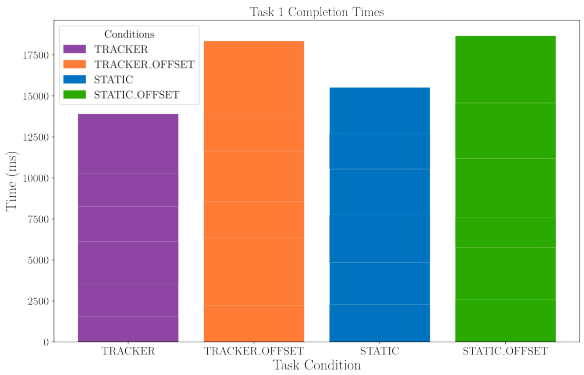
1	2	3	4	5	6	7	8	9	10

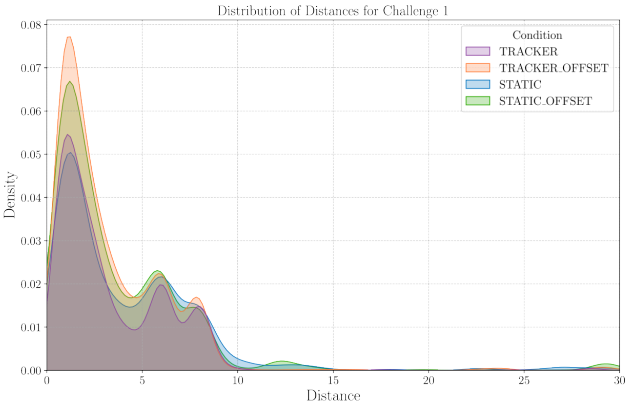
It was unclear in which direction I needed to move my hand next

1	2	3	4	5	6	7	8	9	10

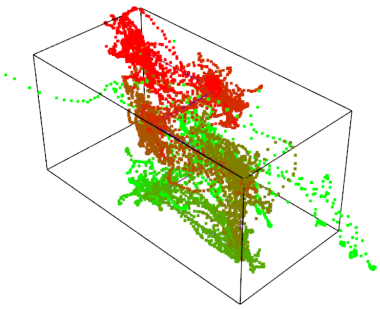
A.4 Full Results

Task 1

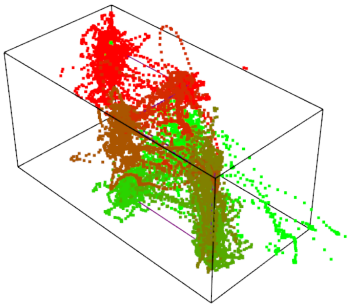




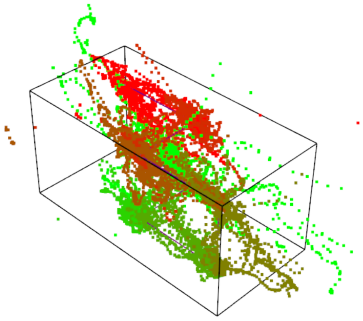
Tracker



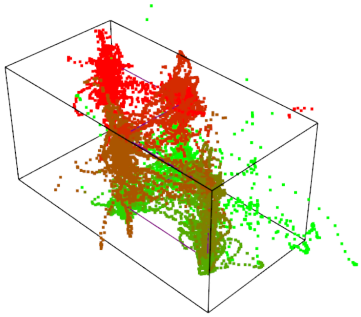
Tracker Offset



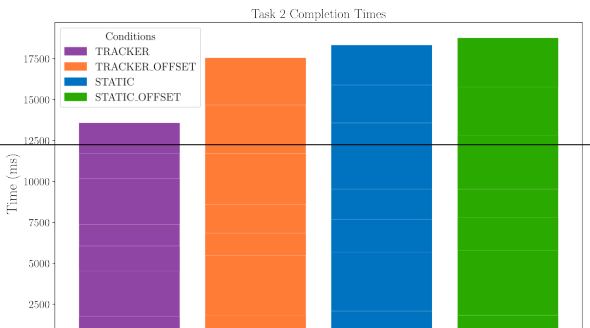
Static

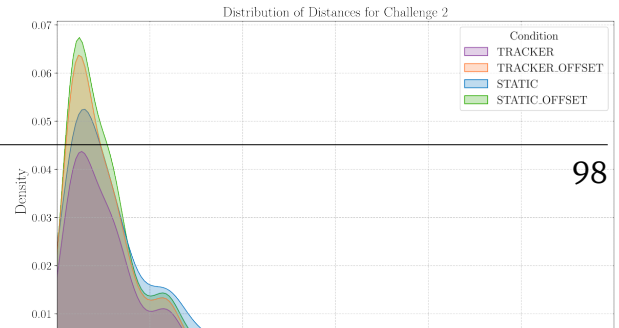


Static Offset

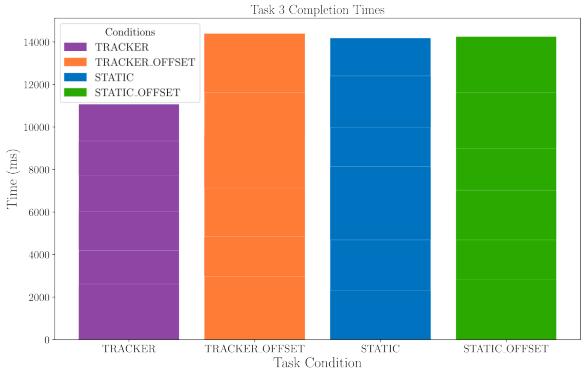


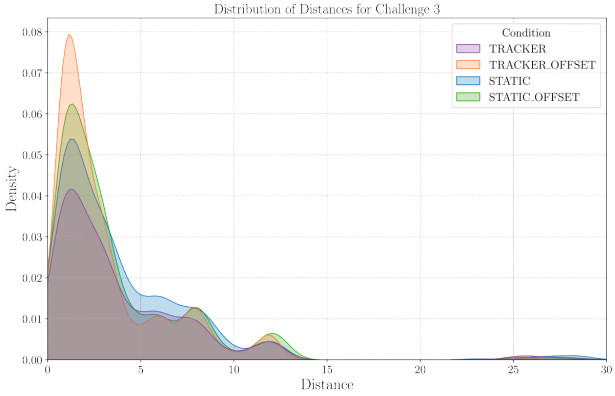
Task 2



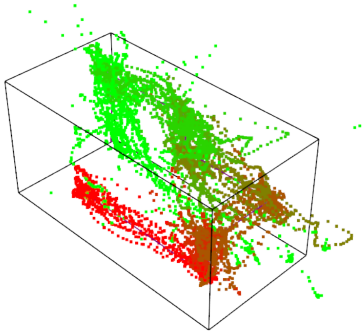


Task 3

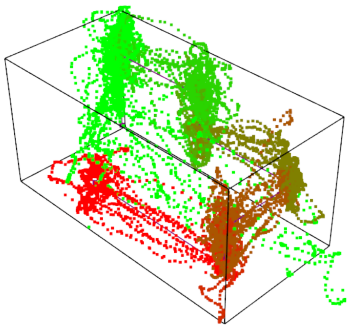




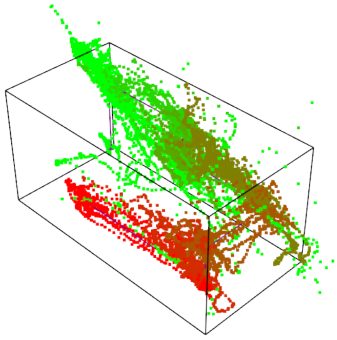
Tracker



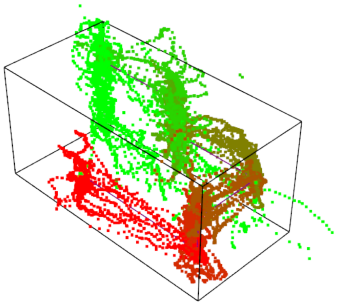
Tracker Offset



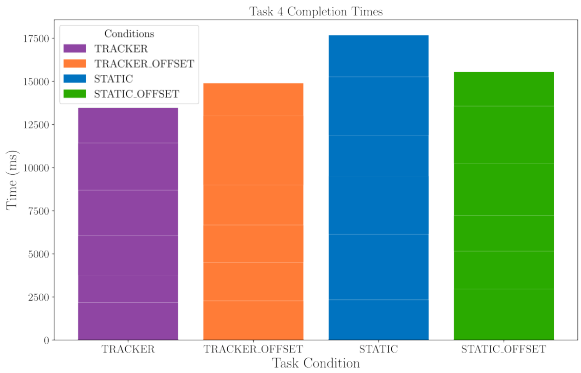
Static

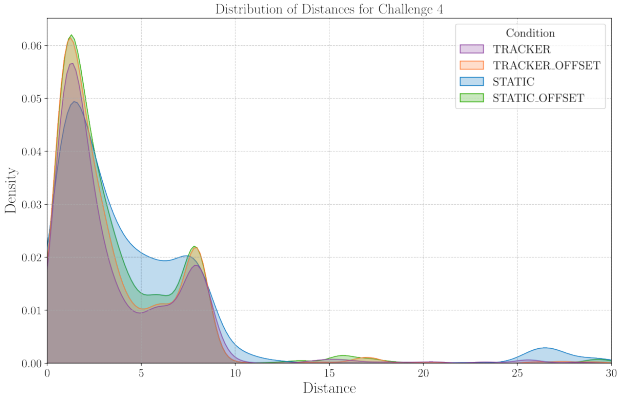


Static Offset

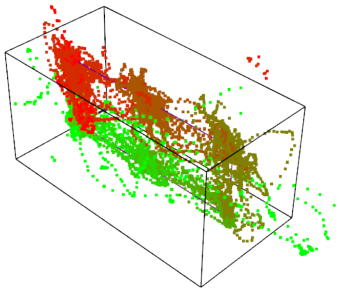


Task 4

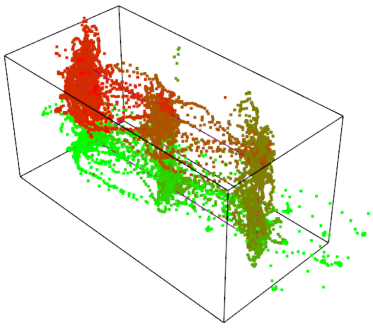




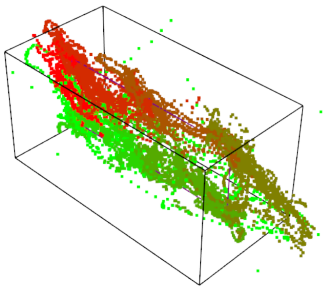
Tracker



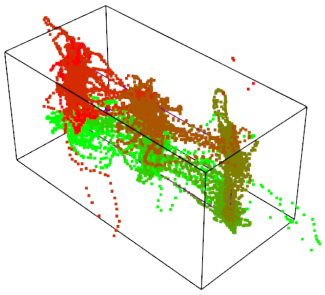
Tracker Offset



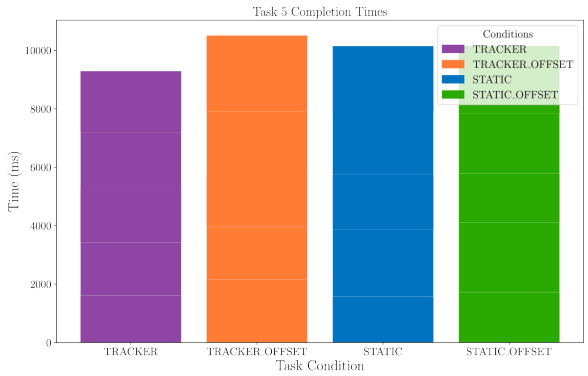
Static

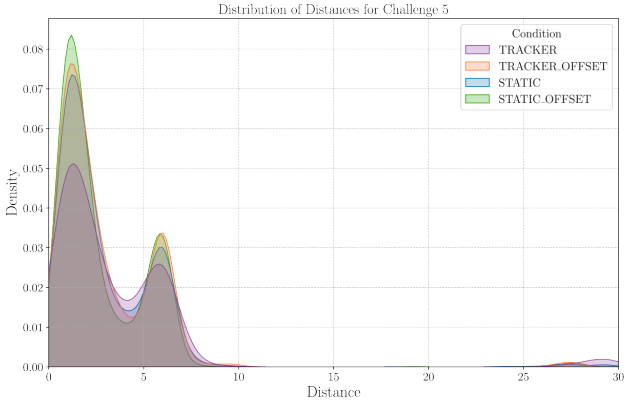


Static Offset

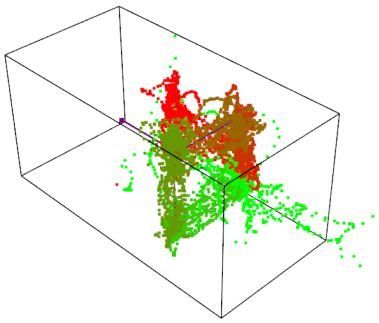


Task 5

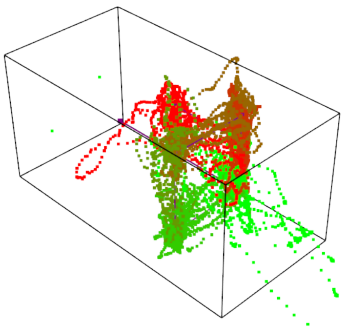




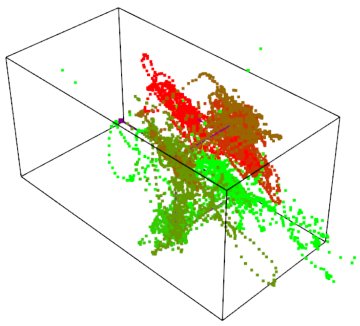
Tracker



Tracker Offset



Static



Static Offset