Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

A Virtual Volumetric Screen

Author:
Robert Searby Buxton

Supervisor:
Dr Nicole Salomons

January 1, 1980

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

Abstract

TODO

Acknowledgments

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

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

Contents

Chapter 1

Introduction

1.1 Motivations

A volumetric display is a type of graphical display device that can provide a visual representation of objects natively in 3D. They can be viewed from any angle without the need for special visual apparatus by multiple people simultaneously [11]. These displays differ from more traditional virtual reality devices in that they are not immersive, but rather they are a window into a virtual world (See Fig 1.1.1 and Fig 1.1.2). There is no real consensus on what exactly constitutes a volumetric display, let alone the best way to build one. As we cover in the background section there are currently many different approaches being attempted by research groups both academic and industrial to create these displays.

Figure 1.1.1: One of Columbia University's passive optical scattering based volumetric displays [23]

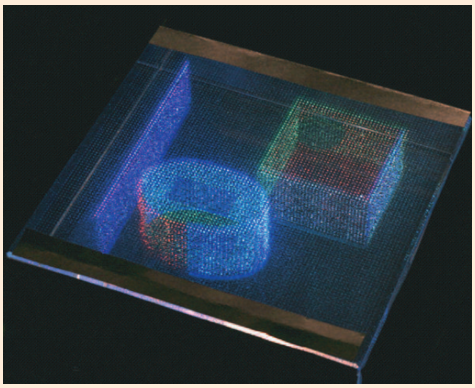
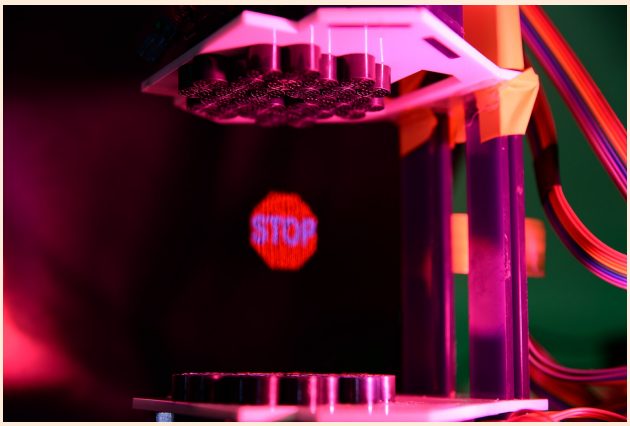


Figure 1.1.2: One of Bristol University's acoustic trapping based volumetric displays [14]



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

1.2 Objectives

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

1.2.1 Volumetric Simulator

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

- **Multi-platform**: We have packaged our platform in the Nix package manager [7] which allows it to be easily run and ported to any platform and hardware that Nix supports.
- **Lightweight**: We use simple rending algorithms in OpenGL [26] to create our virtual volumetric display allowing our software to be computationally cheap to run compared to more fully-fledged rendering/games engines that might be used typically for HCI research like Unity.
- **Cheap**: By relying on only a generic depth camera and a standard display our software requires minimal hardware to run, making research conducted on our platform cheap and easy to run.
- **Reproducible:** By building with Nix we can guarantee that any experiments conducted using this platform will be completely reproducible. (See background)

1.2.2 User experiment

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

Chapter 2

Ethical Discussion

2.1 User Study

1. Got ethical approval from ethics committee.

Chapter 3

Background

3.1 Nix/NixOS

3.1.1 Introduction to Nix

Nix [7] 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 [6] is widely used in both industry [1] and academia [5] [21] [4], and its associated public package repository `nixpkgs` [13] as of Jan 2024 has over 80,000 unique packages making it the largest up-to-date package repository in the world [22]. Out of Nix has also grown **NixOS** [8] a Linux distribution that is conceived and defined as a deterministic and reproducible entity that is declared functionally and is built using the **Nix** package manager.

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

Figure 3.1.1: Nix Store Path

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

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

Nix builds packages in a sandbox to ensure they are built exactly the same way on every machine by restricting access to nonreproducible files, OS features (like time and date), and the network [29]. 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 [4].

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

Figure 3.1.2: Nix Build Loop

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

3.1.2 Example of a Nix package

To give an example of what a Nix package might look like. We have created a flake (one method of defining a package) in Listing 3.1.1 that builds a version of the classic example package ”hello”.

Listing 3.1.1: flake.nix

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

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

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

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

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

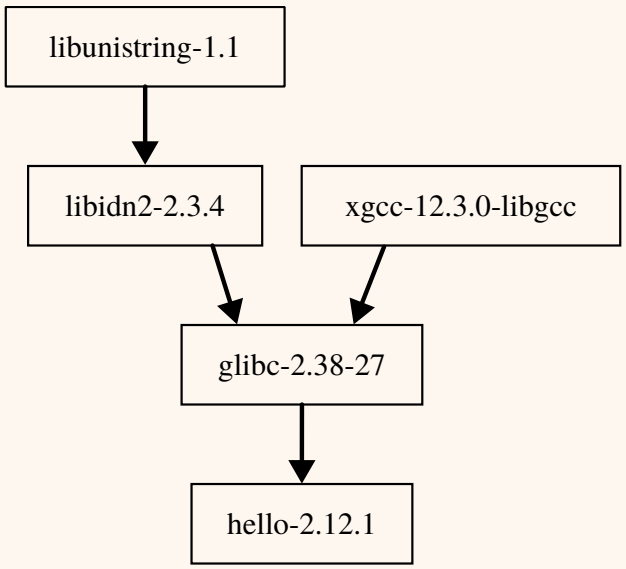
Below we have given some examples of how to run and investigate our hello package in Listing 3.1.2.

Listing 3.1.2: Terminal

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

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

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

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

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

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

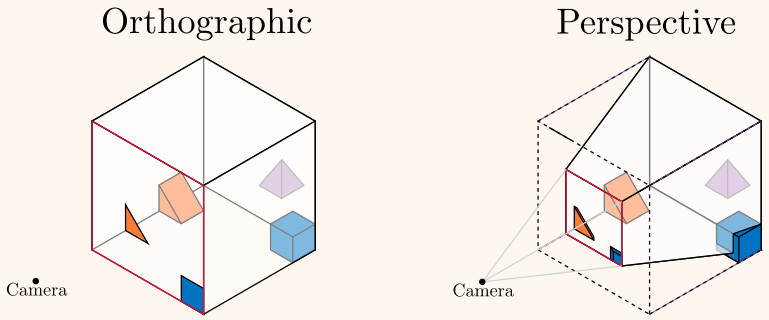
In Fig 3.1.3 we can see the package dependency graph of our `hello` package. We are only dependent on 4 packages `libunistring`, `libidn2`, `xgcc`, `glibc` all of which Nix have installed and configured separately the rest of the non-nix system (assuming we are not on NixOS).

Figure 3.1.3: Dependency graph



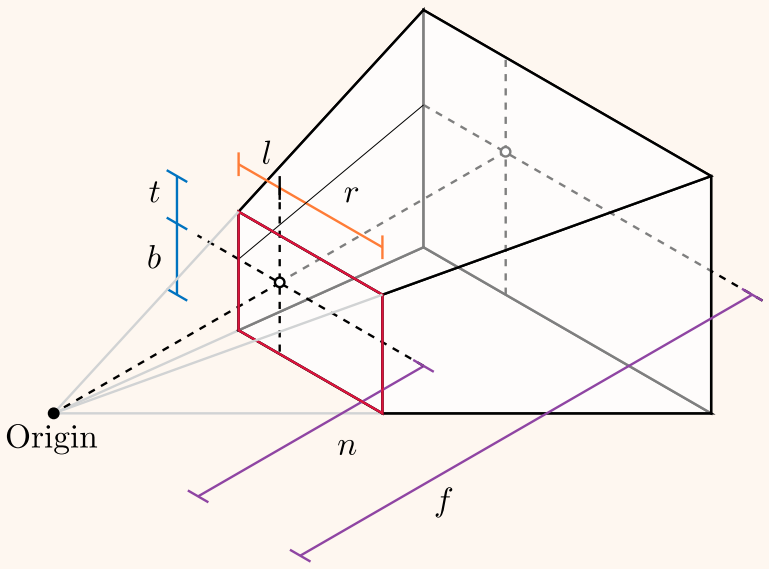
3.2 Perspective Projection

Figure 3.2.1: Orthographic and perspective projections



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

Figure 3.2.2: Using frustum to generate a perspective projection



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

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

This maps a specified viewing frustum to screen space (with intermediate steps handled by OpenGL) [16]. 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.

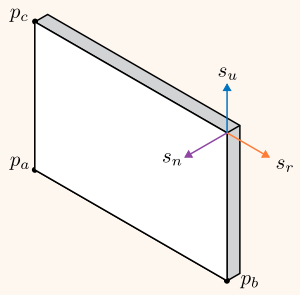
If we can track the position of a viewer's eye in real time then we can create the illusion of a 3D scene behind and in front of a display using this `frustum` function. This can be done fairly trivially following Robert Kooima's method he sets out in "Generalized Perspective Projection" to calculate f, l, r, b, t, n as the viewer's eye changes position [20].

3.2.1 Generating the perspective projection

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

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

Figure 3.2.3: Defining a screen in 3D space



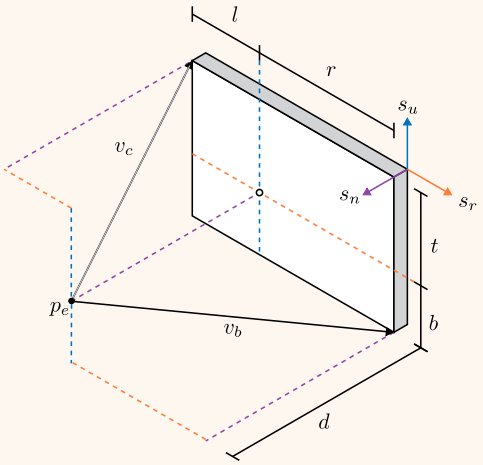
Next, we introduce the viewer's eye which we will refer to as p_e. We can draw 2 vectors v_b, v_c from the viewer's eye p_e to the corners of the screen p_b, p_c as seen in Fig 3.2.4. In the diagram, we also have labeled the components of each of these vectors in the basis of the screen. We can compute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

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

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

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

Figure 3.2.4: Screen Intersection with view

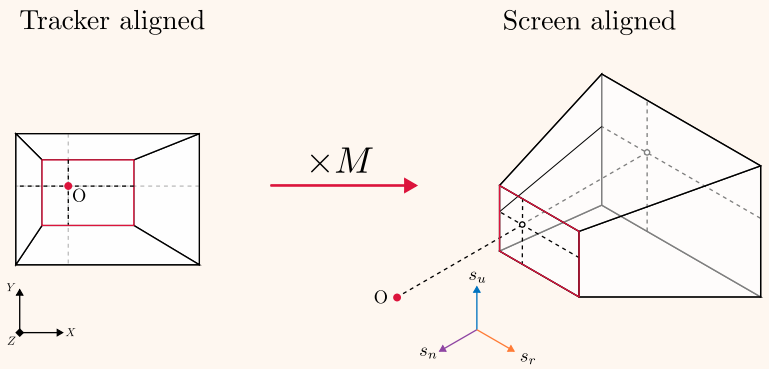


We can now generate a projection matrix by calling `frustum` using d as our near-clipping plane distance n with an arbitrary value for the far-clipping plane f depending on our required scene

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

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

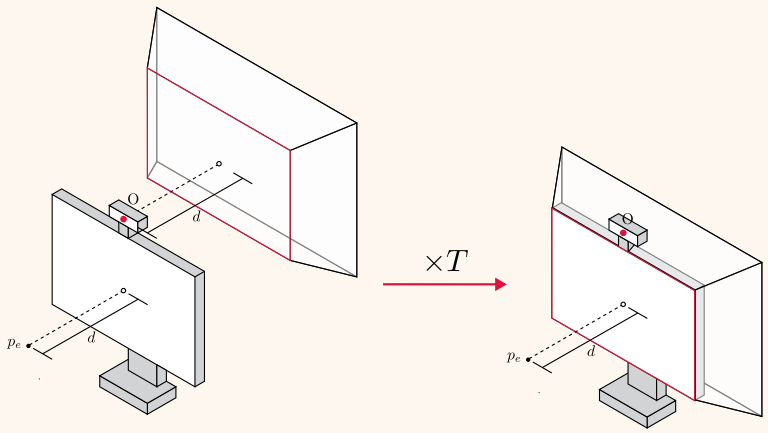
Figure 3.2.5: Rotating the frustum from tracker space alignment into screen space alignment



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

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

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

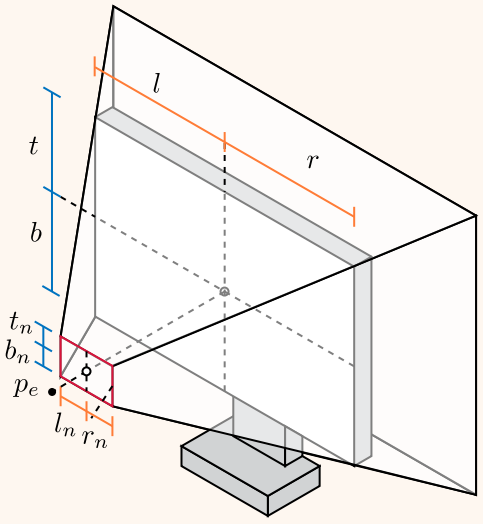


We now have a working method for projecting virtual objects behind our screen onto our screen however it is also possible if we desire to project objects in front of our screen onto the screen as well as long as they lie within the pyramid formed between the edges of the screen and the viewer's eye.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

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

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

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



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

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

3.2.2 Sample code

Below in Listing 3.2.1 we have given an example of a function implementing the process we have just described in C++.

Listing 3.2.1: projection.cpp, Sample code for creating the 3D illusion projection

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

using namespace glm;

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

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

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

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

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

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

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

    return projMatrix * rotMatrix * transMatrix;
}
```

3.3 Volumetric displays

Volumetric displays [11] provide a three-dimensional viewing experience by emitting light from each voxel, or volume element, in a 3D space. This approach enables the accurate representation of virtual 3D objects while providing accurate focal depth, motion parallax, and vergence. Vergence refers to the rotation of a viewer's eye to fixate on the same point they are focusing on. Moreover, volumetric displays allow multiple users to view the same display from different angles, providing unique perspectives of the same object simultaneously.

3.3.1 Swept Volume Displays

Swept volume displays are one prominent category of volumetric displays. They employ a moving 2D display to create a 3D image through the persistence of vision effects. This is achieved by moving the 2D display through a 3D space at high speeds while emitting light from the display where it reaches the position of each corresponding voxel. Common techniques for achieving this include using a rotating mirror [12], an emitting screen, typically an LED-based [15], or a transparent projector screen [19]. There currently exist commercial products that implement this technique as can be seen in Fig 3.3.1 and Fig 3.3.2.

Figure 3.3.1: The VXR4612 3D Volumetric Display, a projector-based persistence of vision display produced by Voxon Photonics. [31]

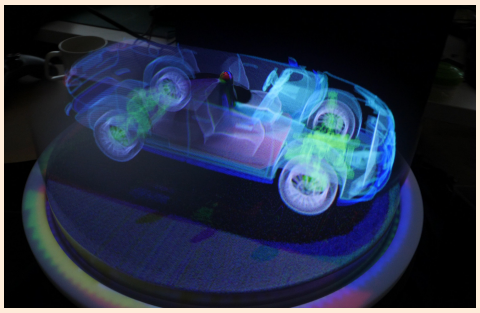


Figure 3.3.2: A Volumetric Display / Holographic Signage, an LED-based persistence of vision display produced by Brightvox Inc. [3]

3.3.2 Static Volume Displays

Static volume displays are another category. They employ a static transparent medium that when interacted with creates a 3D image. The result is that light is emitted from the display at each point in a 3D space. Techniques for achieving this range from using a 3D array of LEDs [27], lasers and phosphorus gas [32], or a transparent laser-induced damaged medium that can be projected into [23]. There has been research into photon-activated dye [24] and even quantum dot-based displays [18].

3.3.3 Trapped Particle Displays

Acoustic Trapping Displays displays are a relatively new category of volumetric displays. They employ a 3D array of particles that are suspended in air using acoustic levitation. [14] [17] 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 [28]. The advantage of this sort of display is that space that is not being used to display an object is empty and can be passed through. This is in contrast to swept volume displays/most static volume displays where the space not being used to display an object is filled with the display's hardware.

3.3.4 Issues

Volumetric displays often require custom/cutting-edge hardware (e.g. extremely high refresh rate projectors, transparent micro LEDs, complex laser systems) which makes them expensive, difficult to manufacture and calibrate and not widely available. For example, the Voxon VX1, one of the few if only commercially available volumetric displays costs, $11,700 USD [25] 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 [2] 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 [35] but this still provides a major issue.

3.3.5 Volumetric Screen Simulations

Because of these issues, there has been some research into simulating volumetric displays. One commonly used method is the so-called fish tank virtual reality (FTVR) display [33] which has been commonly used to simulate volumetric displays, [10], [34]. 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 [30]). 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 [9].

Chapter 4

Implementation

4.1 Overall System

4.1.1 Introduction

- The build system is responsible for compiling the simulator and the rendering system, as well as running the user study.
- Complex build system that requires the compilation of C++, CUDA, and Python code.
- Needs to download and manage large ML models and object files to render.
- Needs to be easily portable to allow for the user study to be conducted on a variety of systems.
- From prior experience, we know that the build system can be a significant bottleneck in the development process.
- Chose to use nix as the build system as it is a declarative build system that allows for the easy management of dependencies.
- Allows overlays to modify packages globally.

4.1.2 Overview

1. Draw a diagram of the build system
2. The build system is split into two parts: the simulator, and the user study.
3. Show how the simulator is compiled as a shared library and called from the user study.
4. Show how models and object files are downloaded and managed.

4.1.3 Dev Environments

1. Use nix-shell to create a development environment that contains all the dependencies required to build the system.
2. Dev shell for building and coding the simulator
3. Dev shell for building and coding the user study
4. Dev shell for launching and managing local mongodb database

4.1.4 Extra work

1. If something is not already packaged in nixpkgs, you have to package it yourself.

Azure Kinect Package

1. The Azure Kinect SDK is not packaged in nixpkgs.
2. Microsoft provides an out of date Ubuntu package.
3. We had to port this package to nix.
4. Was a pain.

Dlib package

1. Dlib was packaged in nixpkgs but cuda support had been added wrong.
2. Was able to fix this locally (explain nice overlays for nix).
3. Decided to be a good citizen and submit a PR to nixpkgs, to fix it for everyone. This took much longer than expected as ended up having to fix a lot of other things in the package that weren't a problem for me but were for everyone else.

MediaPipe

1. MediaPipe was not packaged in nixpkgs.
2. MediaPipe is built with Bazel which is supported in nixpkgs but is annoying to work with.
3. To use MediaPipe in the system, we had to package it in nixpkgs and convert it to being a shared library by wrapping it in a c interface.
4. This was a pain.

4.2 Rendering System

4.2.1 Introduction

1. Rendering system is responsible for rendering the 3D models and the environment.

4.2.2 OpenGL

1. OpenGL is a cross-platform graphics API that is used to render 2D and 3D graphics
2. Used GLM mathematical library for matrix manipulation and projects.

4.2.3 Object Loading

1. Object loading support was added to the rendering system to allow for the rendering of complex 3D models.
2. We construct our challenge by warping spheres, cylinders, and cubes.
3. Used tinyobjloader to load .obj files

4.2.4 Lighting

1. We are using a simple lighting model (Blinn-Phong) that uses ambient, diffuse, and specular lighting.

4.2.5 Perspective

1. Take eye position from the tracker and render the scene from that perspective.

4.3 Tracking System

4.3.1 Introduction

1. Explain what the tracking system is.
2. Explain why we need a tracking system.
3. Explain what the tracking system is used for.
4. Requirements (i.e needs to be high-resolution, low latency, high framerate, etc.)

To be able to accurately produce a virtual volumetric screen, we need to be able to track the user's face and hands. This is so we can render the correct perspective of the screen to the user. Our tracking system had to fit the following requirements:

- **High resolution:** To be able to accurately track the user's face and hands smoothly.
- **High framerate:** To be able to track the user's face and hands smoothly.
- **Low latency:** To be able to track the user's face and hands in near real-time.

4.3.2 Hardware

Figure 4.3.1: Azure Kinect

Get Image from here https://deviestore.com/product/microsoft-azure-kinect/ source microsoft for the image.

For this project we used a Microsoft Azure Kinect camera ToCite Fig 4.3.1. The Azure Kinect camera has two sensors, a depth sensor (IR Sensor) and a colour sensor. For this project we have configured the camera to capture images at its widest field of view $90° \times 74.3°$, with the exposure time 12.8 ms and it's highest framerate 30fps. To be able to use these settings we had to compromise on the resolution of the images running the RGB camera at 2048×1536 rather than it's maximum resolution of 4096×3072 and the depth camera at 2×2 binned with a resolution of 512×512 rather than it's maximum unbinned resolution of 1024×1024. Because we use the depth camera in wide field of view mode $120° \times 120°$ rather than $75° \times 65°$ we

compromised on the maximum operating range of the camera only being able to track hands up to 2.88m away rather than 5.46m.

4.3.3 Core Libaries

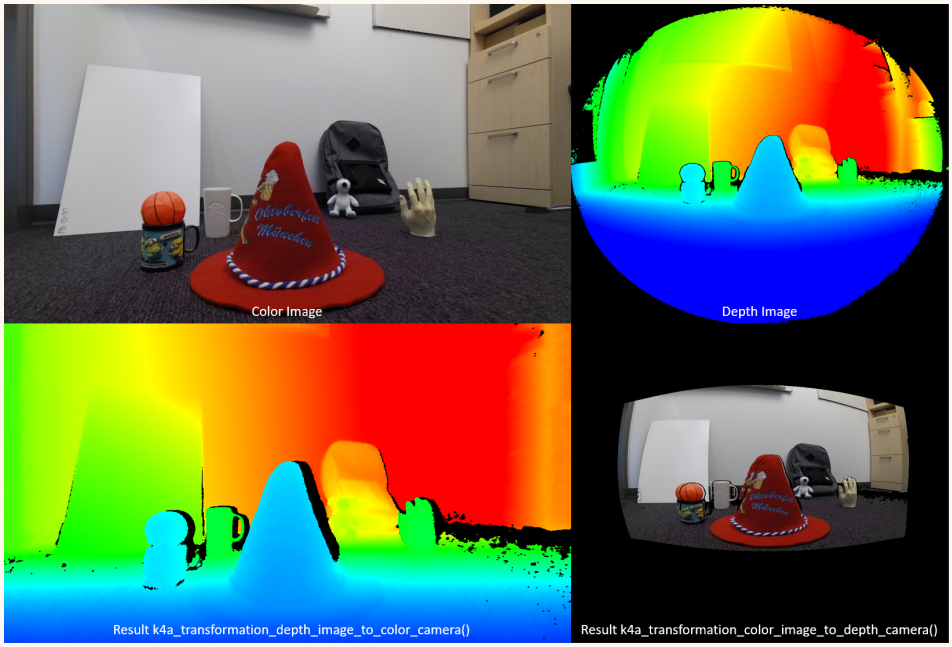
Figure 4.3.2: Core Libraries used for tracking

Cite libraries for the images

Azure Kinect SDK (K4A)

We use the Azure Kinect SDK (K4A) library to retrieve Captures from the Kinect and the handle the ”space” transformations you need to calculate the positions of points in 3D.

Figure 4.3.3: Different spaces changing Depth and Colour Images



The camera when polled with K4A returns a ”Capture” which is a struct containing a colour image, depth image and an IR image. We don’t use the IR image in our project so we ignore it. It is worth noting that the depth image is in a different space to the colour image as can

be seen in Fig 4.3.3. The depth image is in the ”depth space” and the colour image is in the ”colour space”. To use the two images in conjunction we have to convert them into the same space.

Using a ”calibration” that is generated at the start of the program k4a lets you convert between the 4 difference ”spaces”, ”Depth 2D”, ”Depth 3D”, ”Colour 2D”, ”Colour 3D”. There are interesting performance implications of using different spaces for different tasks. For example going from ”Depth 2D” to ”Depth 3D” is significantly faster than going from ”Colour 2D” to ”Colour 3D”.

Another interesting side effect of converting between spaces if because the IR and colour cameras are physically offset and diffract differently you get ”depth shadows” ToCite as can be seen in the Bottom Left image in Fig 4.3.3. Which can make it difficult when trying to track thin objects like fingers as you have a high probability of getting an invalid depth.

OpenCV

We use OpenCV to handle the images we get from the Azure Kinect SDK. OpenCV is a library that provides a comprehensive set of functions for image processing and computer vision. We use OpenCV to convert the images we get from the Azure Kinect SDK into a format that can be more easily used by Dlib and MediaPipe. We make use of OpenCV's GPU accelerated functions like to down scaling. To increase the performance of our tracking system. We also use OpenCV for debugging to render the images to the screen.

Dlib

We use Dlib to track the user's face. Dlib is a modern C++ toolkit containing machine learning algorithms and tools for creating complex software in C++ to solve real-world problems. We use Dlib's face tracking model ToCite to track the user's face. Dlib's face tracking model is a deep learning model that can track the user's face in real-time. We use Dlib's GPU accelerated functions to increase the performance of our tracking system. We initially used Dlib's CPU face tracking functionality but found that it was bottlenecking our system. We then switched to Dlib's GPU functions and found that it significantly increased the performance of our tracking system. Fill out more

MediaPipe

We use MediaPipe to track the user's hands. MediaPipe is a cross-platform framework for building multi-modal applied machine learning pipelines. We use MediaPipe's hand tracking model ToCite on colour images to track the user's hands. MediaPipe's hand tracking model is a deep learning model that can track the user's hands in real-time. Fill out more

4.3.4 Overall Tracking System Design

1. Diagram of how the system works.
2. Picture diagram of capture to tracking frame.

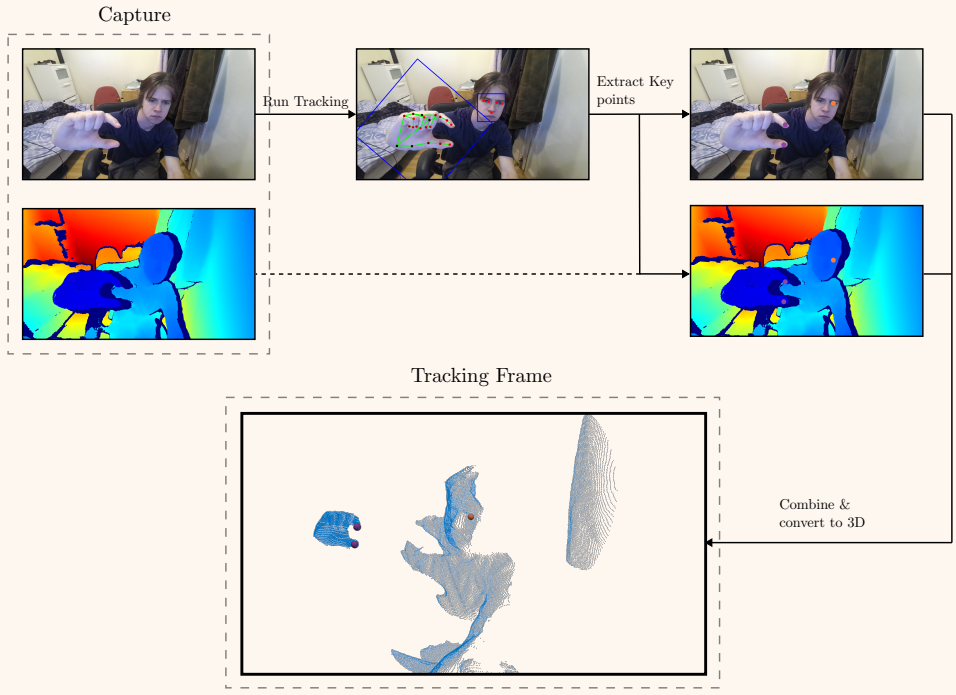
3. Mention how we track the hands and face seperately so two capture frames.

The purpose of the tracking system in brief strokes is to convert the Captures provided by the Kinect Camera into 3D point representing useful information from the render like the position of the users eye and fingers. The current system just returns the position of the left eye and index and thumb tip on the closest hand to the camera. The system runs as the same speed of as camera (30fps) and feels smooth.

There were a number of different approaches we could have taken but in the end we decided to use a system that tracked the position of the users face and hands separately in 2D which we then converted into 3D. The main other approach we could have taken was to track the face and hands in 3D directly. In the end the main reason we decided against this approach was because although the rendering system was a key part of this project it was not the main focus. The main focus was the user study and the tracking system was just a means to an end. The ecosystem for tracking in 2D is much more mature (thanks in part to mobile phones) than in 3D and we were able to leverage a lot of existing work (Such as the previously described libraries Dlib, MediaPipe, and OpenCV). Another reason was that we were concerned about the performance of tracking in 3D. We were worried that tracking in 3D would be too slow to be able to track the users face and hands in real-time, however is suspect this probably would have been fine. There are a number of downsides that arise from tracking in 2D. For example, as we will talk about a bit more in the evaluation section, it is difficult to find the accurate positions of objects in 3D space that are occluded for example fingers behind other fingers. Because fingers are fairly small objects we also have to sample a vague area where we think it is and pick the closet point as if you only sample the predicted point you can miss the finger.

update image to have two tracking frames and track hand and eye separately and show what happens in tracking fails

Figure 4.3.4: Tracker overview



As can be seen in Fig 4.3.4 the basic flow of the tracking system is as follows.

1. Retrieve the Capture from the Kinect Camera.
2. Run hand and head tracking on the colour image.
3. Extract the key 2D points from the hands (thumb and index tip) and face (left eye).
4. Use the depth image to convert the 2D points into 3D points and put these in a ”tracking frame” to be sent to the rendered.

While it might seem strange at first as can be seen in the Tracking Frame in Fig 4.3.4 we actually keep a separate instance for the left eye and the thumb and index tips. This is because there is no guarantee that the tracker will be able to detect both the face and hands at the same time. For example if the user is holding their hand in front of their face the tracker will only be able to detect the hand. If this happens our system will just reuse the last known position which is a good approximation of faces position.

4.3.5 Tracking Models

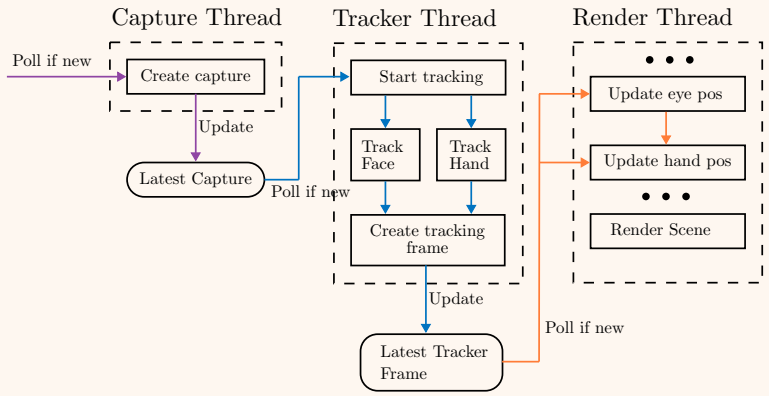
1. Dlib and MediaPipe are used to track the face and hands respectively.
2. Dlib is used to track the face. 5 point model with a cnn. cite correctly.
3. MediaPipe is used to track the hands. 21 point model with a cnn. cite correctly.
4. mention running on downscaled images to increase performance.

5. mention switching to tracking middle finger and index and more stable when camera looking down.
6. Mention model only finds position of surface of finger

4.3.6 Multithreading

1. Seperate thread for tracking capturing and rendering
2. Increased framerate but did not change latency.
3. More wasteful in terms of resources but there is there are more than enough resources.
4. Talk about polling architecture and how we cache frames so we don't do redundant work.

Figure 4.3.5: Multi-threaded design

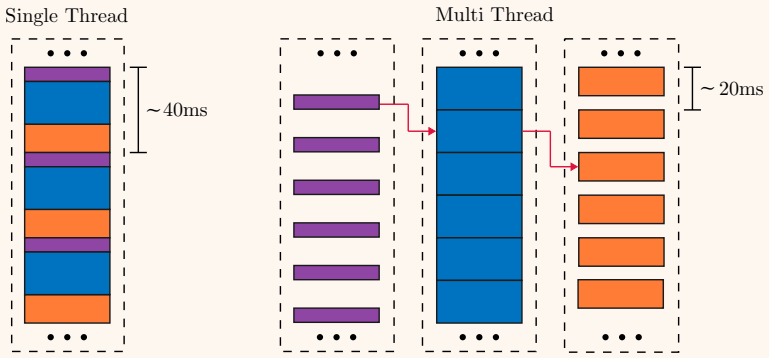


One of the more challenging aspects of this project was to make our tracking system run fast enough that it felt smooth. To achieve this we had to make use of multi-threaded design as can be seen in Fig 4.3.5. We used a separate thread for tracking, capturing, and rendering. This is a slightly wasteful use of resources as we effectively waste a thread waiting for captures however as the whole simulation is surprisingly light on resources this was not a problem it was worth it for the framerate gain it provided.

The purpose of this design choice is to make sure there is always a threading running the tracking models as these are by far the computationally most expensive part of the system. Using a multithreaded design does not reduce the latency (which we talk about more in evaluation) of the system but rather increases the throughput which in our case is the framerate of the application. As we are using a camera that runs at 30fps we need to be able to process an image every $\frac{100\text{ms}}{30\text{ms}} = 33\text{ms}$. As can be seen in Fig 4.3.6 by switching to a multi-threaded design we can increase the framerate of our whole tracking system to be the same as time required to run our tracking modules on our already retrieved captures. This also gives us the advantage of being able to run the simulation at an independent framerate to the tracker.

TO DO ADD KEY, also show how long each individual part takes. Also show that rendering happens multiple times a second

Figure 4.3.6: Single vs Multi threaded design

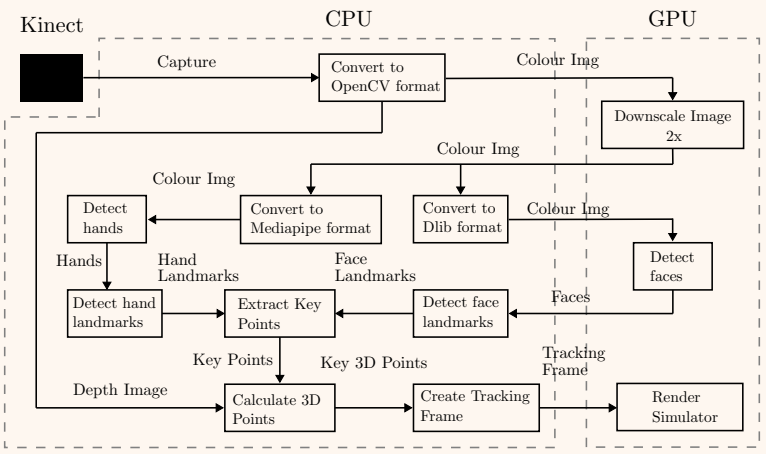


4.3.7 GPU Acceleration

Another method we use to increase the performance of our tracking system is to use GPU acceleration. Both Dlib and MediaPipe support GPU acceleration. We only used GPU acceleration in Dlib as we found that it significantly increased the performance of our tracking system. get actual stats for dlib gpu stats. We did not use GPU acceleration in MediaPipe as the CPU speed was already sufficient and the purported speedup of 12.27ms with GPU acceleration vs 17.12ms to cite https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker with cpu only didn't seem worth the effort of enabling CUDA in mediapipe after it already provided difficult to use (See build systems for more information) . We did not test the performance of MediaPipe with GPU acceleration as we did not have the time.

As you can see in Fig 4.3.7 we only used GPU acceleration for 3 parts of our tracking system. We down scale our colour images using OpenCV's GPU accelerated pyramid down function as this is a highly parallel task and benefits from the acceleration. We also run the dlib cnn for detecting the face on the GPU. Lastly we run our rendering system on the GPU.

Figure 4.3.7: Overall Tracking System Design



1. Dlib and MediaPipe both support GPU acceleration. Only did it in DLIB. Get benchmarks from Mediapipe and DLIB
2. downscaling to increase performance has the downside of reducing precision/resolution but it was negligible.

4.3.8 Camera Positioning

1. Camera is measured in 3D space and orientation is recorded.
2. Screen is measured in 3D space.
3. Camera starts off orientedby 30 degrees down????? need to double check (this is false)
4. Iterative process of guessing where screen is adjusting

4.4 User Study

4.4.1 Introduction

1. The user study was conducted to evaluate the effectiveness of the system in improving the spatial reasoning skills of the participants.

To validate the usefulness of our system we decided to conduct a Within-Subjects User Study. The user study was designed to show the capacity of the system for being used in research. The study was designed to test the effectiveness of users using volumetric displays under different conditions.

4.4.2 Experimental Variables

1. Independent Variables:
   (a) 3D Perspective (On/Off)
   (b) Interaction Offset (On/Off)
   (c) So 4 conditions
2. Dependent Variables:
   (a) Time taken to complete task
   (b) Number of subtasks completed
   (c) Eye and hand positions
3. Control Variables:
   (a) The five tasks are the same in each condition.
   (b) Position of participant
   (c) Position of the tracking camera
   (d) Position of the zone of interaction
   (e) Size of the display.
4. Confounding Variables:
   (a) The participants may have different levels of experience with VR/Volumetric.
   (b) Left-handed vs right-handed might make a difference for the tasks.
   (c) Wearing glasses might make a difference for head tracking.
5.

For this study we wanted to evaluate the difference in performance of participants in interacting with volumetric screens with their hands.
Our two independent variables were:

- **3D Perspective**: (On/Off). This controls if the system is able to use the eye tracking system to create the illusion of a 3D volumetric display as can be seen in Fig TODO .
- **Interaction Offset**: (On/Off). This controls if display is directly in front of the participant or if it is offset by a fixed amount as can be seen in Fig TODO .

Giving us a total of 4 conditions to test.
The first condition we wanted to test was if there was any noticeable drop in performance when using the system in 2D (i.e with eye tracking disabled but still using hand tracking). The second condition was if there was a performance difference if participants "teleoperated" (controlled it with a fixed offset) the simulator vs using their hands directly. Combining these two conditions gives us the four conditions we tested as can be seen in Fig **??**.

4.4.3 Tasks

1. Trace out points like a buzzwire game.
2. Draw Diagram of how the task works
3. Draw Isometric view of the tasks
4. Tasks are designed to be annoying if not in 3D.
5. Timeout of 1 minute.

In each of the 4 conditions the participant must complete the same 5 tasks. The tasks are designed to be simple but more difficult to complete if not in 3D. To complete a task a participant must trace the path between the points with their index and middle finger in the order the simulator presents to them. A green point is completed and an orange point represents the next point ot be completed as can be seen in Fig TODO .

The participants have a timeout of 1 minute to complete the task. If they do not complete the task in the time limit, the task is marked as incomplete. The time each point is completed is recorded as well as the position of the hand and eye throughout the minute. The 5 different tasks are shown in Fig TODO .

4.4.4 Study Implementation

1. We run the study from a python based CLI (Click).
2. We compile the simulator as a shared library and call it from the python CLI using a C-FFI.
3. We receive the results and logs from python in json format and store them in a mongoDB database.
4. This data is then used to generate the results.

4.4.5 Participants

1. We record age, gender, if they are left or right handed, and if they have any experience with VR, and if they wear glasses.
2. The participants were given a random order of the four conditions.
3. They complete the 5 tasks in each
4. Fill out survey about each condition.
5. Fill out survey about the system as a whole at the end.
6. Ran the study in Huxley building.

4.4.6 Study Results

1. Use ANOVA test?
2. ????

Chapter 5

Evaluation

5.1 User Study Evaluation

1. Look at the results.

5.2 Simulator Evaluation

1. Measure Latency (Don't forget to mention the latency of the camera as well (exposure time))
2. Measure Accuracy (How often tracker loses track of hand)
3. In for example do tasks and see how many times it loses tracking and why.
4. Limitations of the simulator
5. It works on other machines first run, (different machine different gpu). Verify it works on WSL?
   (a) Occlusion (show a picture of occlusion with fingers behind the palm of the hand)
   (b) Lighting
   (c) Hand size

Chapter 6

Conclusions and Future Work

Chapter 7

Project Plan (To be removed)

7.1 Progress Log

- **October 10th:** At the project's inception, I made several considerations. I chose OpenGL over more extensive systems like Unity, prioritizing its lightweight nature, which is crucial in a project where frame rate and latency significantly impact user experience. My prior experience with OpenGL served as a valuable risk mitigation factor. To streamline development and avoid the hassle of building on multiple machines, I opted for Nix, a choice rooted partly in my familiarity with the software and my prior negative experience trying to build OpenGL applications on different machines. Nix's capability to ensure consistent builds with a single command across various machines appealed to me. As a proactive risk mitigation measure, I decided I should verify the feasibility/difficulty of building OpenGL within the Nix environment. My supervisor Nicole Simmons had access to Azure Kinects which would be a good camera option however they have quite an arbitrary package set that has not been packaged for Nix yet as far as I was aware. I had experience using open-source heading tracking for video games. I thought it might be a good idea to investigate using a variation of aruco marker tracking in OpenCV with a webcam.

- **October 13th:** I attempted to implement an aruco paper marker tracker using a Logitech C270 HD webcam, which proved unsuccessful. However, I investigated the neural net tracking approach showed promise but did not work that well suggesting that upgrading to a higher-end webcam might yield better results.

- **October 19th:** I achieved a basic version of OpenGL building with Nix on my machine, initially utilizing some unsightly hard coding, which I successfully refactored out.

- **October 22nd:** I established an automated development environment using Nix, which loads useful packages for building and debugging while configuring Visual Studio Code's intellisense to recognize the necessary libraries, simplifying development.

- **October 23rd:** I upgraded to a more recent version of OpenGL and eliminated hard-coded asset paths to ensure compatibility across various systems.

- **October 30th:** I decided I would attempt to try and package Azure-Kinect-SDK in Nix giving myself a 1-week cut-off, if I could not get it working by then I would switch to a different solution or ditch the idea of using Nix altogether.

- **November 01st:** Progress continued as I managed to hack together an initial version of the Azure-Kinect-SDK on NixOS. However, reliability in the build process remained a concern. I had to fork the Azure-Kinect-SDK repository because it has officially been dropped by Microsoft and there are CMAKE build bugs on Linux.

- **November 05th:** I managed to create a reliable build process by patching the RPATHs for the produced binaries using `patchelf`. Microsoft's proprietary `libdepthengine.so` library that is used for sensing depth from the camera was minimally patched to work on Ubuntu (the only "officially" supported Unix-based OS). To get it to work in Nix required patching out leftover redundant Windows DLL-based paradigms which was very annoying.

- **November 06th:** I identified the need to address Git submodules' issues and build libk4a and k4atools separately. The Azure Kinect SDK used git submodules that don't work well with Nix.
- **November 07th:** I resolved the GitHub submodules issue by employing Git fetch instead of including the flake in the repository. Additionally, I engaged in a productive meeting with my supervisor, Nicole, discussing the physical design of the "real world" versus the matrix, including a physical space mirroring "the matrix" dimensions, marked with Aruco cubes at each corner.
- **November 08th:** I split the Azure SDK flake into its dedicated GitHub repository, enabling it to build from there. Additionally, I created a separate package, libk4a, which k4a-tools (formerly known as k4aviewer) now utilizes. Although k4a-tools were built successfully, I encountered challenges with libk4a due to missing dependencies.
- **November 13th:** I addressed the issue with libk4a not functioning by adding a specific dependency (udev) and refining code dependencies. I now stood theoretically prepared to commence development. I noted that the Kinect offered a better field of view below than above for depth sensing.
- **November 23rd:** The day brought hours of troubleshooting a perplexing issue with the translation matrix, ultimately resolving it. Progress continued as I focused on familiarizing myself with OpenGL and working on achieving the room perspective.
- **November 25th:** I successfully simulated the virtual "room" behind the screen and embarked on exploring head tracking.
- **December 15th:** A Kinect Class was created, and basic frame reading functionality was implemented, paving the way for further exploration into eye tracking using OpenCV.
- **December 16th:** I drew inspiration from a student at Cambridge master's thesis [34] and devised a plan for obtaining eye position using a series of steps, including acquiring a raw RGB image from the Kinect, employing OpenCV for eye tracking, projecting onto depth data, and scaling for use as the viewer's position.
- **December 17th:** I achieved basic eye tracking using OpenCV and `dlib`, albeit with reduced speed due to CPU utilization. Further refinement and refactoring were required, drawing from insights gleaned from a relevant paper.
- **December 19th:** I successfully integrated CUDA support in OpenCV through my Nix build, despite encountering significant challenges. The next step was to create a pipeline for image compression to expedite recognition. Additionally, I made a GitHub issues page for maintaining a useful to-do list and performed an upgrade to the new NixOS 23.11 release while addressing issues related to building OpenCV with CUDA.
- **December 20th:** I accelerated `dlib`'s performance significantly by utilizing CUDA image compression, acknowledging the trade-off between speed and resolution/accuracy. I contemplated the possibility of utilizing Mediapipe for facial recognition using GPU functions and considered optimizing face detection by selecting a smaller window based on prior head position.

- **December 29th:** After returning from a holiday, I resumed work on head tracking, resolving issues with camera coordinate alignment and making refinements to the code. I also addressed a concern related to paths being relative to the source directory rather than the `/nix/store/`.
- **December 30th:** I resolved the local path issue and improved the readability of the Nix flake. My focus shifted toward model loading, currently utilizing TinyObjLoader.
- **December 31st:** I successfully achieved basic model loading, albeit with some complexities. I now loaded the Cornell box as the default scene, although textures remained a work in progress. Additionally, I streamlined the process of automatically downloading TinyObjLoader from GitHub.
- **January 02:** Progress continued with multithreading functionality and basic refactoring. My attention turned to 3D glasses, evaluating options like polarized 3D and Anaglyph 3D. I also explored the reasons behind the slow face detection, attributing it primarily to the `dlib` face detector's performance. By enabling CUDA and AVX support, I managed to switch `dlib` to use a CNN that utilized the GPU, significantly improving performance. I had to fix a bug in the Nix package for `dlib` to enable GPU support because it wasn't actually using it.
- **January 03:** I faced challenges related to camera coordinates and perspective, leading to adjustments and continued exploration of rendering techniques. I recognized the importance of rendering to assess the 3D effect accurately and contemplated the possibility of establishing a proper test rig.
- **January 04:** The day brought further challenges in dealing with perspective issues, prompting a reversion to the original version of perspective. Additionally, I commenced work on loading textures onto models.
- **January 05:** Progress continued with the implementation of Blinn-Phong lighting and the start of downloading polyheavy models. A critical bug affecting perspective was identified and resolved.
- **January 06:** I achieved full model loading with face-by-face materials. I decided to use a chess set as a demonstration, as it offered an interesting perspective with its chequerboard pattern. The concept extended to the possibility of implementing hand interaction for playing chess against an AI opponent like Stockfish, which held significant potential as a captivating feature.
- **January 09:** In a significant development, I upstreamed my local fixes to the Dlib Nix package in a PR to nixpkgs, making them accessible to others.
- **January 10:** Started work on interim report.

7.2 Current Status

At the time of writing the interim report, I have achieved a working prototype that can display virtual 3D scenes to the viewer. The viewer can move their head around and see the scene

from different perspectives. The renderer has been written from the ground up in OpenGL and is capable of rendering models loaded from OBJ files using the `TinyObjLoader` library. The viewer's eye position is tracked using a Microsoft Azure Kinect. The renderer takes the live colour image stream from the Kinect, compresses it about 4 times on the GPU using CUDA then uses a CNN in `dlib` to detect a face. With that face, a 5-point facial landmark detector is run which gives out two points we care about (the left and right side of the eye) we use these to get an approximation for the position of the pupil. We then use the Kinect to map this point in camera space to the depth camera to get the 3D coordinates of the viewer's eye. We then render the appropriate scene as can be seen in Fig 7.2.1 and Fig 7.2.2. The demo currently runs at 60fps and is convincing if you close one eye.

Figure 7.2.1: View from the left

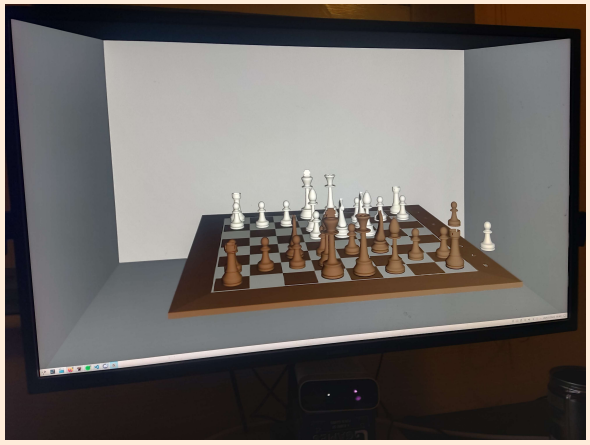
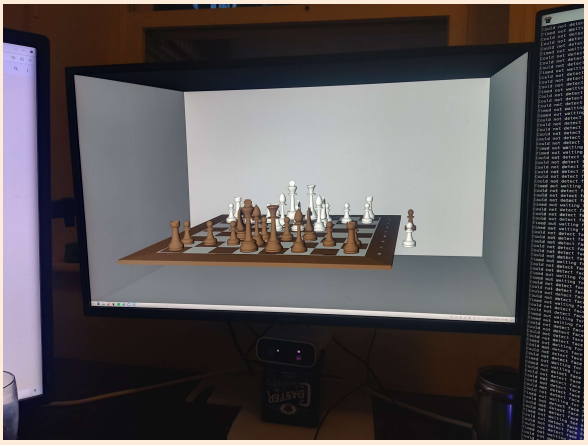


Figure 7.2.2: View from the right



The repository is publically available at `https://github.com/RobbieBuxton/VolumetricSim` and is slightly over 1500 lines of C++ and Nix code. The repository should (but not yet tested) be able to be run from any x86-intel machine (to be extended later) with an Nvidia GPU and an Azure Kinect. All you need to do is install the Nix package manager which I recommend downloading from here `https://zero-to-nix.com/concepts/nix-installer` and run the shell command from Listing 7.2.1.

Listing 7.2.1: Terminal

```
[shell:~]$ sudo nix run github:RobbieBuxton/VolumetricSim
```

Nix will download and configure all the dependencies, it will probably take a while on the first run because we use CUDA and the Nix community currently doesn't publically cache unfree software.

As a side effect of this project, I have also created a repository of a package that will automatically build and run the Azure Kinect SDK in Nix. This is available at `https://github.com/RobbieBuxton/k4a-nix`.

This is a key milestone as represents the baseline functionality required to start the more interesting additions that will comprise the novel part of this project. This has de-risked the project as if some features turn out to be unfeasible it should not be difficult to pivot to a different approach while still having a significant deliverable.

7.3 Future Milestones

7.3.1 Key milestones

- **Hand Tracking:** The next key milestone is to implement hand tracking. This will allow the user to interact with the virtual scene. I have not yet got a concrete idea about the best methods to do this but plan to organize a meeting with Andrew Davison to discuss this, as he is an expert in this field and hopefully should be able to point me in the right direction. I would be interested to find out if he thinks it would be better to track the hand in the point cloud or first on the color camera and map to the point cloud. Furthermore, I predict this task will be the most difficult part of the project and could easily take a month or more.

- **Interactive Challenge:** To test the device we will need to create an interactive challenge. I am not yet completely decided on what this will be, but I am leaning toward a chess game against an AI opponent currently. The user will be able to move the pieces with their hands and the AI will move its pieces using a chess engine. This will be a good test of the device as it will require the user to interact with the virtual scene and will be a good demonstration of the device's capabilities. I predict this task a week or two. This task is also completely dependent on the hand-tracking challenge.

- **Shadows:** Shadows are a key part of giving the illusion of depth. I plan to implement shadows using ray tracing or shadow mapping. I have implemented ray tracing before in a previous project and it worked well. However, I am not sure if it will be fast enough to run in real time. I will need to do some research into shadow mapping to see if it is a better option. I predict this task will take a week or less.

- **Calibration mode:** Currently the device is difficult to debug because the only method you have is based on if it looks visually correct. I am still not sure if it is correctly calibrated exactly, i.e. if the screen is defined correctly based on the camera and the offset is correctly measured. I plan to create a mode to verify if the simulator is calibrated correctly. Furthermore, I still need to research to figure out the best way to do this. I am aware of a few papers that claim to tackle this problem. I predict this task will take a week or less.

- **Run User Study:** The final milestone is to run a user study to evaluate the device. I plan to run a user study with 10-20 participants. I will ask them to complete a questionnaire about their experience with the device and ask them to complete the interactive challenge. Furthermore, I predict this task will take a week or less. This will require ethics approval which I need to apply for after this interim report deadline.

7.3.2 Optional milestones

- **Adapt to be compliant with OpenXR:** OpenXR is an open standard for virtual reality and augmented reality. It is supported by all the major players in the industry including Microsoft, Valve, Oculus, Google, and many more. It would be a good idea to adapt the project to be compliant with this standard, so it can load into any OpenXR compatible application. I am currently not sure how difficult this is or if it is even feasible.
- **Run on a Nvidia Jetson:** The Nvidia Jetson is a small embedded computer with a GPU. It would be interesting to see if the project could be adapted to run on this device. This would allow all the GPU-heavy machine learning tasks to be computed on this device, and you could plug the device into a monitor and have a portable volumetric display. The Nvidia jetson is already packaged in Nix, so it would be a function of fixing bugs from the new hardware and optimizing the code to run on a smaller GPU than my desktop which is am currently testing on. This would probably take a week or two. I already own an Nvidia Jetson.
- **Anaglyph 3D:** Anaglyph 3D is a method of displaying 3D images using filters typically red and green color filters and does not require special hardware. The 3D effect currently requires 1 eye to be closed so adding 3D support would make it a more immersive experience. I predict this task will take a day or two as I just need to duplicate the perspective per eye.
- **Realtime light detection:** Taking inspiration from what I have learned from advanced graphics this term, it might be interesting to add another camera, a fish eye lens and use that to generate a real-time light map. This would allow the virtual scene to be lit by real-world lighting. I have already talked to Prof Abhijeet Ghosh about this idea, and he thinks it is feasible. However, this is going in a slightly different direction with the project. I predict this task will take a week or two.
- **Improve compatibility:** Currently the project only works on Nvidia GPUs. It would be good to improve compatibility to work on AMD GPUs and Intel GPUs and also run without a GPU (albeit slowly). It would also be good to support different depth cameras other than the Kinect (Like Intels Intellisense) as this has been discontinued by Microsoft. This would require a lot of refactorings and would probably take a week or two.
- **Tracker Logging:** It would be good to log the tracker data to a file so that it can be reused later. This would allow experiments to be reproduced without having to use the tracker device again and would allow people to externally verify experiments. This would require a lot of refactorings and would probably take a week or two.

Chapter 8

Evaluation Plan (To be removed)

8.1 Demonstrated Functionality

8.1.1 Eye Tracker

One of the key components of our volumetric display simulator is the eye tracker. It is responsible for tracking the position and orientation of the user's head. This is used to render the volumetric display from the correct perspective. To properly evaluate our simulator we must first evaluate the quality of the eye tracker. The key metrics to track would be at different camera input resolutions (we can vary the resolution by pyramiding down) and the frame rate the tracker can run (bounded by Azure Kinect's maximum fps of 30) where the tracker thinks the eye is. What percentage of the time can it detect an eye during an example input, and what are the maximum orientations of a face that it can detect an eye? We can also compare the accuracy of the eye tracker to other eye trackers.

8.1.2 3D Position accuracy

Once we have evaluated the eye tracker we can then evaluate the accuracy of the eye position. We can do this by comparing the predicted eye position from our system to the actual measured eye position from the origin of our tracker camera. This is more a measurement of calibration than the eye tracker itself as we assume the tracker is already functioning correctly.

8.1.3 Renderer

The renderer is responsible for rendering the volumetric display from the correct perspective. To properly evaluate our simulator we prove that the rendered scene is accurate. We can do this by comparing the rendered scene to the real scene we are trying to simulate. We can do this by scanning a real scene with a depth camera and then rendering the same scene with our simulator and comparing the two from the same perspective. There are currently no open-source volumetric display simulators to which we can realistically compare our simulator.

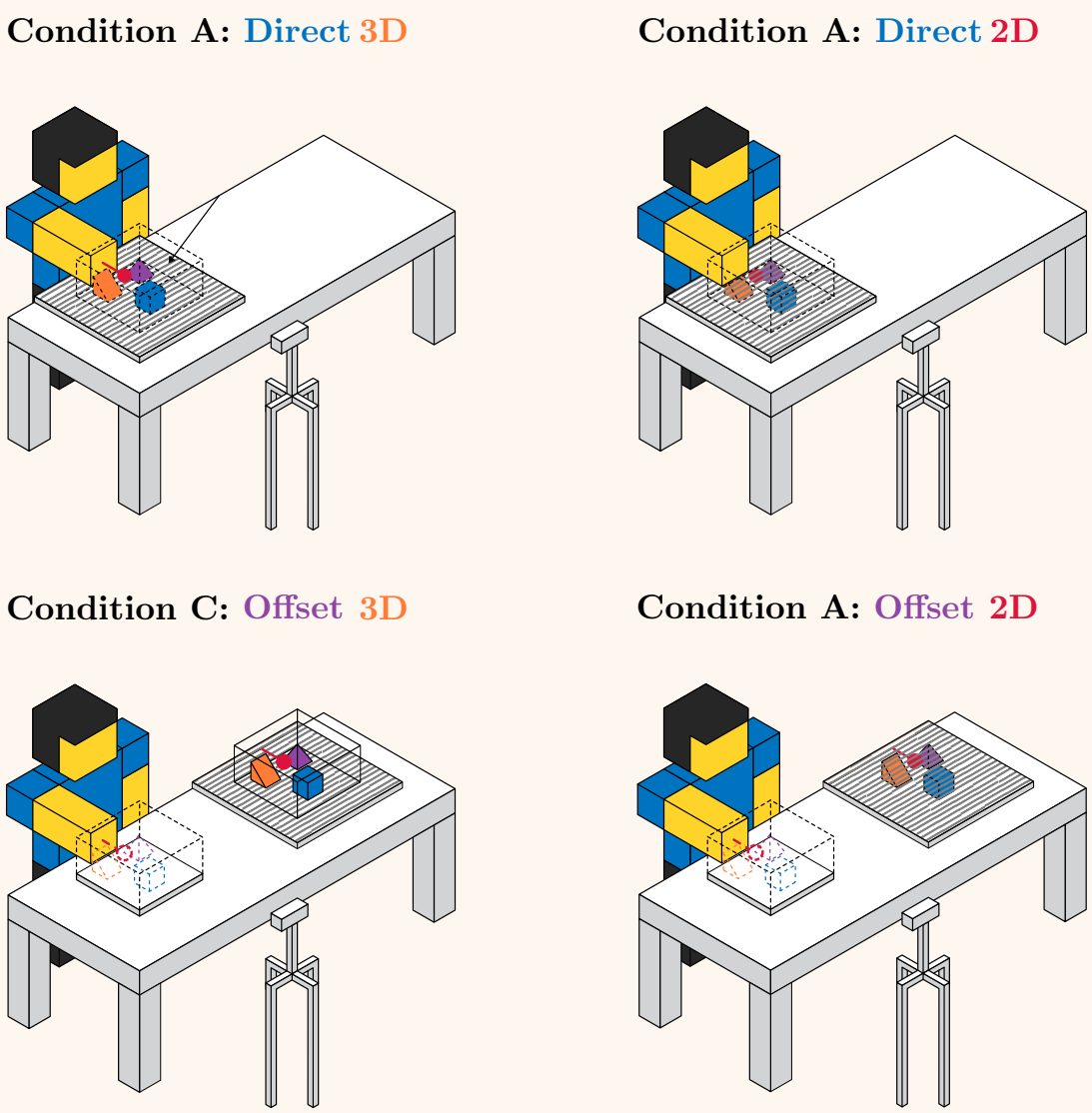
8.1.4 Reproducibility

The purpose of building this project with Nix was to provide a reproducible platform for conducting HCI research into volumetric displays. We need to show that we can easily build this project on a variety of platforms and that the results of any experiments conducted on one platform can be completely reproduced on another platform (i.e. by recording the output of the tracker camera and re-using it on a different machine).

8.2 Success Criteria: User Study

If the simulator can be effectively used to conduct novel HCI research into volumetric displays then we will know we have succeeded.

Figure 8.2.1: Teleoperation User Study



We plan to run a user study with two conditions as can be seen in 8.2.1. In the first, condition A, we plan to have the participant take part in a virtual task with their hands directly interacting with the objects where they perceive they are. In the second, condition B, we plan to have the participant interact with the scene in a second offset interaction zone while they perceive their interactions on a separate display. This will test the ease at which a user can interact with a volumetric display via teleoperation. We will measure the time taken to complete the task and the number of errors made. We will also ask the participants to fill out a questionnaire to measure their subjective experience. We will then compare the results of the two conditions to see if there is a significant difference in the time taken to complete the task and the number of errors made. We will also compare the results of the questionnaire to see if there is a significant difference in the subjective experience of the two conditions.

8.3 Novel Contributions

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

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

Chapter 9

Ethical Issues (To be removed)

9.1 Human participants

We will be running a user study to evaluate our simulator, so we will need to get approval from Imperial College London's Science, Engineering and Technology Research Ethics Committee (**SETREC**). This process can take some time, so we will need to start this process after the submission of this interim report to ensure we have approval by the spring term when the user study is expected to take place.

9.2 Data collection

We will be collecting data from the user study, so we will need to ensure that we comply with the General Data Protection Regulation (**GDPR**). We will need to ensure that we have a data protection impact assessment (**DPIA**) and that we have a data management plan (**DMP**). Furthermore, we will also need to ensure that we have consent from the participants to collect their data and reuse it.

9.3 Military applications

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

9.4 Copywrite Limitations

9.4.1 Open Source

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

9.4.2 Proprietary

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

Bibliography

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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).

[30] 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.

[31] *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).

[32] 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.

[33] 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.

[34] 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.

[35] 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.