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

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

Volumetric Displays are a new and exciting technology that has the potential to revolutionize the way we interact with computers. They are a type of 3D display that can be viewed from any angle without the need for special glasses by multiple people simultaneously. [11] These displays differ from a virtual reality experience 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 consensus on what the best way to build a volumetric display is and as we cover in the background section many approaches are being attempted by research groups both academic and industrial.

Figure 1.1.1: A volumetric display created at Columbia University using passive optical scattering [24]

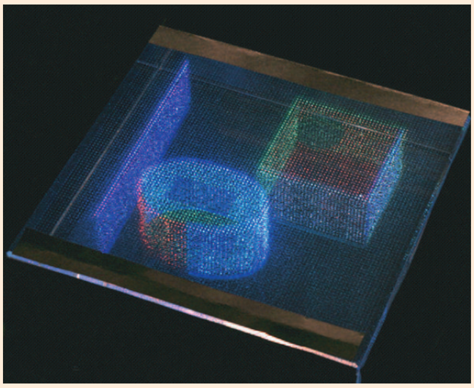
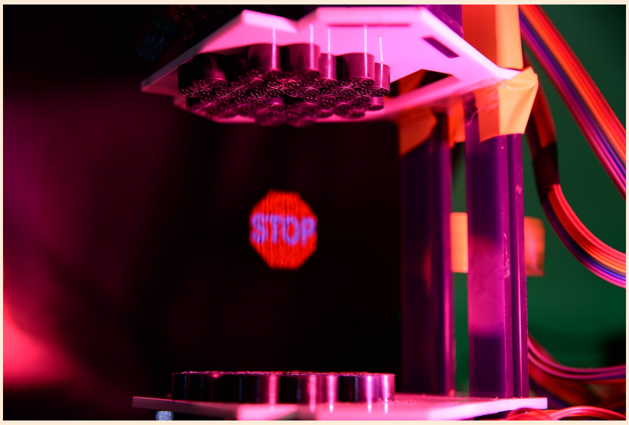


Figure 1.1.2: A volumetric display created at Bristol University using acoustic trapping [14]



It is difficult to conduct human-computer interaction (HCI) research into volumetric displays because these devices are not widely available, expensive to manufacture, and have high bandwidth requirements. This makes it difficult to conduct user studies and experiments. People have created virtual simulations of volumetric displays to try and solve this problem (see background), but these solutions are often complicated and expensive to replicate.

1.2 Objectives

With this project, we aim to provide a cheap, multi-platform, lightweight and simple platform for simulating volumetric displays. We hope that this will enable researchers to conduct HCI research into volumetric displays without the need for expensive hardware. 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 package our platform in the nix package manager [7] which allows it to be easily run on any platform and hardware that supports nix by running a single line of code `sudo nix run github:RobbieBuxton/VolumetricSim`.

- **Lightweight**: By using simple rending algorithms in OpenGL [26] to render a volumetric display our software is computationally cheap to run compared to rendering/games engines that might be used typically for HCI research like Unity.
- **Cheap**: By relying on only a depth camera and standard monitor our software requires minimal hardware to run, making research conducted on our platform cheaper and easy to run.
- **Simple**: We have designed our software to be as simple as possible by taking advantage of the nix package manager to handle all the dependencies and by depending on external libraries like `dlib` [20] to handle more complicated tasks like face detection. This makes it easy to replicate and modify.
- **Reproducible:** By building with nix we can guarantee that any experiments conducted using the simulator 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 simulation platform. We will conduct a user study to compare 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 (See evaluation).

Chapter 2

Background

2.1 Nix/NixOS

2.1.1 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] [22] [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 [23]. 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 2.1.1.

Figure 2.1.1: Nix Store Path

`/nix/store/sbldylj3clbkc0aqvjjzfa6slp4zdvlj-hello-2.12.1`

Prefix	Hash part	Package name

Source files, like tarballs and patches are downloaded and stored in the store directory to ensure all required inputs are always available. As different dependencies result in a different hash and therefore location in the store directory you can have multiple versions or variants of the same package installed while also at the same time avoiding "DLL hell" by making it impossible to accidentally point at the wrong package. Another important result is that upgrading or uninstalling a package cannot ever break other applications. Nix builds packages in a sandbox to ensure packages are built the same way on every machine by restricting access to nonreproducible files and the network [29]. A package can be pinned (and should be) to nix release meaning that once it builds and works today it will continue to work the same way in the future, regardless of when and where it is used.

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 ever long long-term support operating systems) and it means that as the software is reproducible; all the experiments are completely reproducible as all bugs present in the original version stay to 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 populated by CI systems. You can also specific custom caches. The basic process for building nix packages can be seen in Fig 2.1.2.

Figure 2.1.2: Nix Build Loop

1. A hash is computed for the derivation and, using that hash, generates a nix store path, e.g `/nix/store/sbldylj3clbkc0aqvjjzfa6slp4zdvlj-hello-2.12.1`.
2. With the store path in hand, check if the derivation has already been built. First, checks 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, use that, if not continue.
3. Next it checks if the store path exists in a configured binary cache, this is by default `cache.nixos.org`. If it does download from the cache and use that if not continue.
4. Use Nix to 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. The colour green.

2.1.2 Example of a Nix package

To give an example of making a Nix package. We have created a flake in Listing 2.1.1 that builds the classic example package ”hello” also available on `nixpkgs`.

Listing 2.1.1: flake.nix

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

```
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 my `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 the line 2). What nix does is resolve all inputs, and then it calls 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 the package as it 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 and c compiler and version of that compiler we liked. If you wanted to you could compile different parts of the code with different versions of GCC.
- **Line 15:** Here we are slightly abusing the configure phase to generate a hello.c file. Each phase is essentially run as a bash script. Everything inside `mkDerivation` is happening inside a sandbox and 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.

Below we have given some examples of how to run and investigate our hello package in Listing 2.1.2.

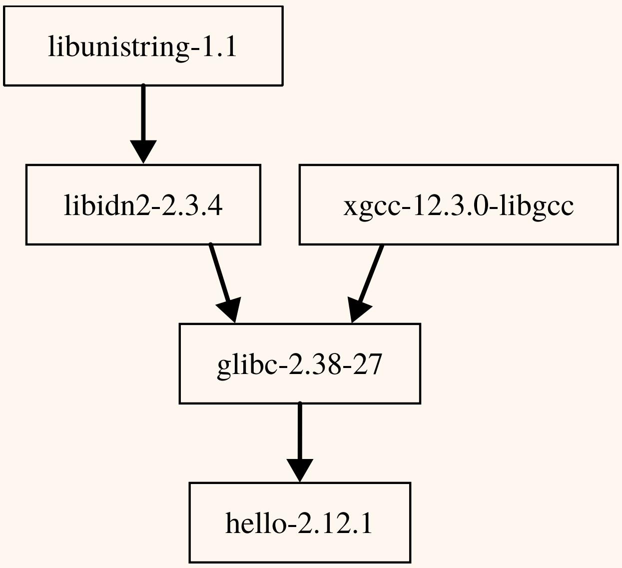
Listing 2.1.2: Terminal

```
[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
```

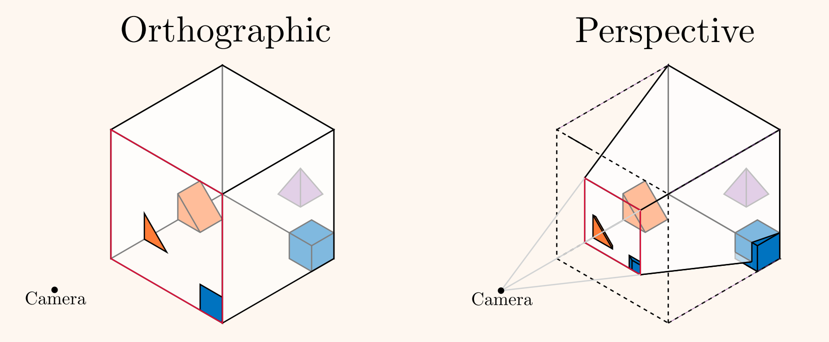
In Fig 2.1.3 we can see the package dependency graph of our `hello` package. We are dependent on 4 packages `libunistring`, `libidn2`, `xgcc`, `glibc` all of which nix have installed and configured separately the rest of the non-nix system.

Figure 2.1.3: Dependency graph



2.2 Perspective Projection

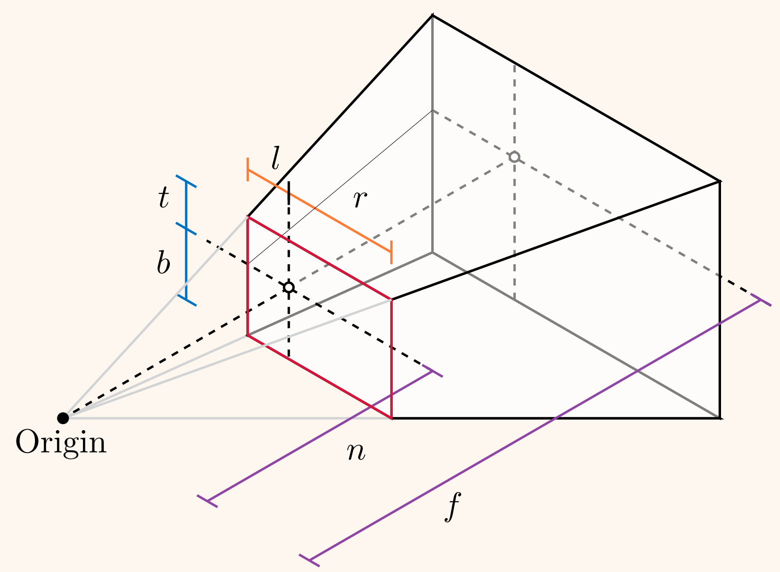
Figure 2.2.1: Orthographic and perspective projections



To represent 3D objects on a 2D surface (our screen) OpenGL supports two types of projections: perspective and orthographic as seen in Fig 2.2.1. 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 (All diagrams in this report are from the orthographic perspective). Unlike orthographic projection, perspective projection simulates the way the human eye perceives the world, with objects appearing smaller as they are farther from the viewpoint as the projection lines converge at a vanishing point. To create the illusion of 3D in this project we must use a perspective projection.

2.2.1 Generating the perspective projection

Figure 2.2.2: Using frustum to generate a perspective projection



OpenGL provides the `frustum` function as seen in Fig 2.2.2 which can be used to construct a perspective matrix.

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

This maps a specified viewing frustum screen-space (with intermediate steps handled by OpenGL) [16]. The viewing frustum is specified by six parameters: f, l, r, b, t, n which represent left, right, bottom, top, near, and far. 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 left typically being a negative value and right a positive value, 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.

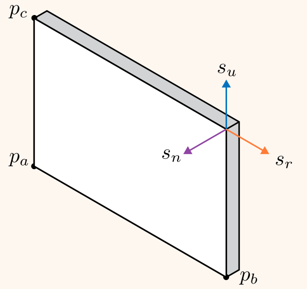
The n and f parameters are scalar values that specify the distances from the origin to the near and far clipping planes along the view direction. Altering the value of n will change the angles of the lines (or vectors) that connect the corners of the near plane to the eye, effectively changing the ”field of view”. Changing the value f affects the range of depth that is captured within the scene.

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 `frustum`. 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 moves [21].

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 are in tracker space, the coordinate system of the device we use to track the eyes. This point can be used to generate an orthonormal basis of the screen of s_r, s_u and s_n which represents the directions up, right and normal to the screen respectively as seen in Fig 2.2.3. 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 2.2.3: Defining a screen in 3D space



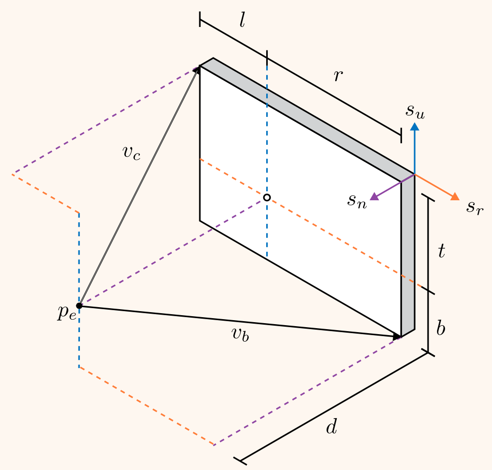
Introducing the viewer's eye which we will refer to as p_e. We can draw three vectors v_a, v_b, v_c from the viewers eye p_e to the corners of the screen p_a, p_b, p_c as seen in Fig 2.2.4. In the diagram, we also have labeled the components of each of these vectors in the basis of the screen. We can 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 `frustum` we must first find the point where a line drawn perpendicular to the plane of the screen that passes through p_e strikes the plane of 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 its component of v_a, v_b, v_c in the screen basis vector s_n, however, as s_n is in the opposite direction we must invert it. 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:

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

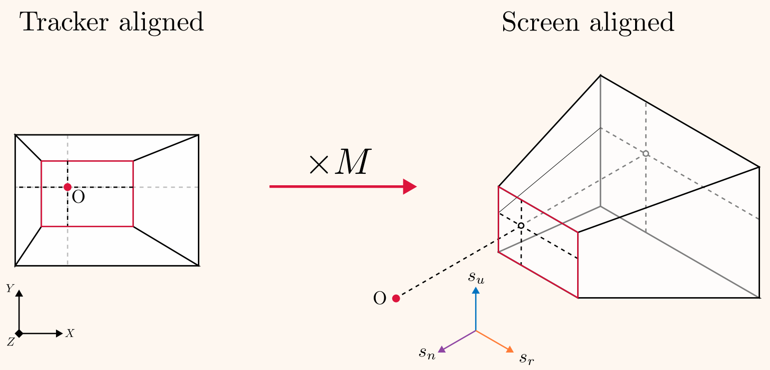
Figure 2.2.4: Screen Intersection with view



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. Furthermore, we have now successfully generated our viewing frustum but we still have two problems. Firstly our frustum has been defined in tracker space so is pointed in the direction of our camera not the normal of our screen. We can remedy this problem by using 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 2.2.5. M is defined as follows:

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

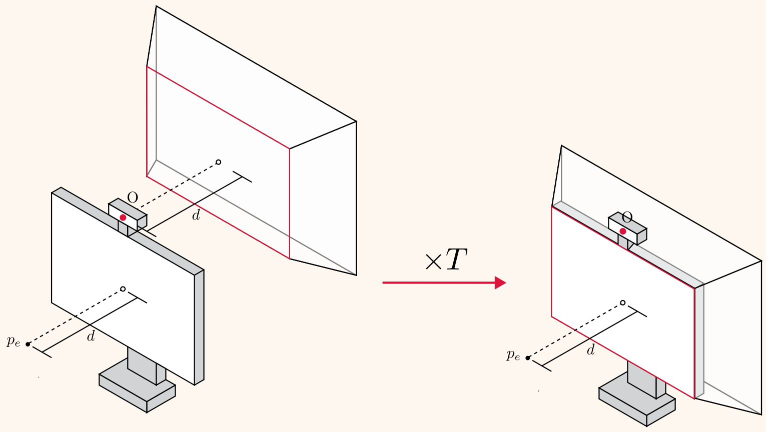
Figure 2.2.5: Moving the frustum from tracker space to screen space



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 forever trapped 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 apex/origin of our frustum. This can be done with a translation matrix T as seen in Fig 2.2.6. T can be generated with the OpenGL function `translate` where we want to offset it by the vector from our Origin to the viewers 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 2.2.6: Translating the viewing frustum to sit inside the screen

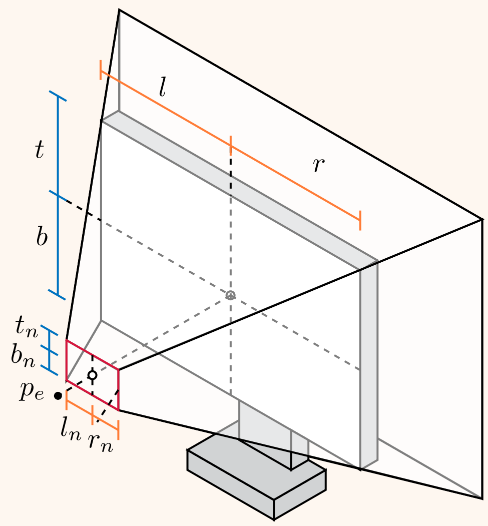


We now have a working method for projecting virtual objects behind our screen onto our screen however it is also possible if we desire to project objects in front of the screen onto the screen as well as long as they lie within the pyramid formed between the edges of the screen and the viewer's eye. We can use similar triangles to scale the near-clipping plane from the plane of the screen to a small distance n from our eye as seen in Fig 2.2.7. Furthermore, we now have 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 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 2.2.7: Extending the near plane to not clip out objects in front of the screen



Following these steps, we can create an accurate projective providing the perspective we would expect to see if there was a scene in front and behind our screen.

2.2.2 Sample code

Below we have attached some sample code of a function implementing the process we just described which is self-explanatory.

Listing 2.2.1: 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);

    // 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;
}
```

2.3 Volumetric displays

Volumetric displays [11] are a promising technology that offers a captivating three-dimensional viewing experience. By emitting light for each voxel, or volume element, in a 3D space, these displays transcend the limitations of traditional 2D planes, providing a truly immersive 3D effect. This innovative approach enables the accurate representation of virtual 3D objects, including focal depth, motion parallax, and vergence, which 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.

2.3.1 Swept Volume Displays

Swept volume displays represent one category of volumetric displays. They employ a moving 2D display to create a 3D image. This is achieved by moving the 2D display through a 3D space and emitting light from the display at each point. Common techniques for achieving this include using a rotating mirror [12], emitting screen typically an LED [15], or transparent projector screen [19]. There currently exist commercial products that implement this as can be seen in Fig 2.3.1 and Fig 2.3.2.

Figure 2.3.1: The VXR4612 3D Volumetric Display, a projector-based persistence of vision display produced by Voxon Photonics. [31]

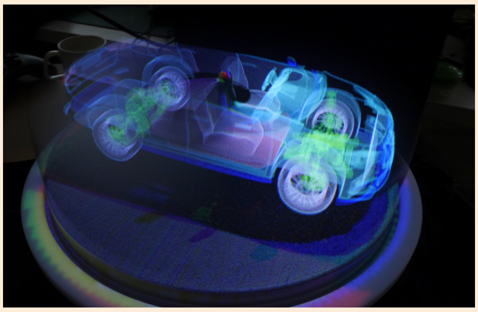
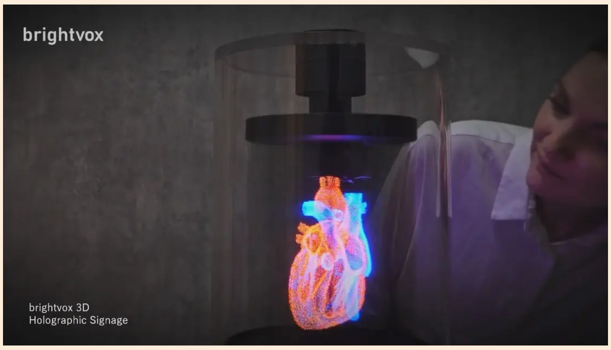


Figure 2.3.2: A Volumetric Display / Holographic Signage, an LED-based persistence of vision display produced by Brightvox Inc. [3]



2.3.2 Static Volume Displays

Static volume displays are another category of volumetric displays. They employ a static 3D display to create a 3D image. This is achieved by emitting light 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 [24]. There has also been some research into quantum dot-based displays [18].

2.3.3 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].

2.3.4 Issues

Volumetric displays often require custom/cutting-edge hardware (e.g. extremely high refresh rate projectors, transparent micro LEDs) which makes them expensive, difficult to manufacture 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 commercially 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.

2.3.5 Volumetric Screen Simulations

Because of these issues, there has been some research into simulating volumetric displays. One commonly used method is by using 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 the image on the displays is adjusted accordingly so that there appears to be a 3D image 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 the user 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 has been to take advantage of 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 3

Project Plan

3.1 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.

3.2 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 render 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 render takes the live colour image stream from the Kinect compresses it about 4 times on the GPU then uses a CNN in `dlib` to detect a face. With that face, a 5-point facial landmark detector is run which gives up two points (the left and right side of the eye) we can use 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 **??**.

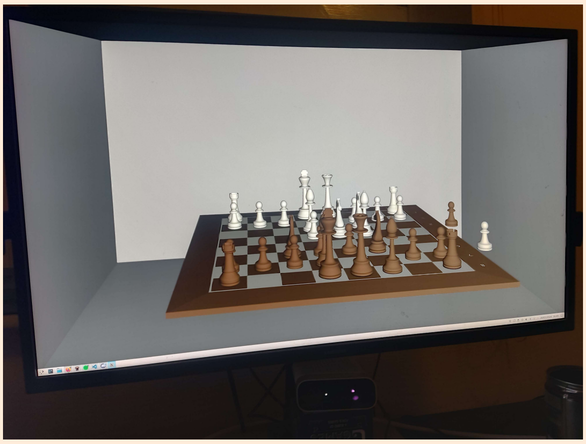


Figure 3.2.1: View from the left

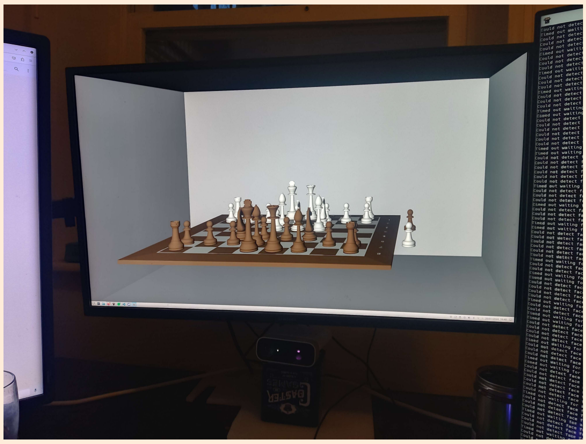


Figure 3.2.2: View from the right

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 be tested) be able to be run from any x86-intel machine 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 3.2.1.

Listing 3.2.1: Terminal

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

Nix will download and configure all 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. (I also haven't tested this on another machine yet, so there is a high chance this doesn't work yet)

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 on 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. 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.

3.3 Future Milestones

3.3.1 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 will be interested to find out if he thinks it is better to track 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. Not yet completely decided on what this will be, but I am leaning toward a chess game currently against an AI opponent. 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 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 if it looks correct. I am still not sure if it is correctly calibrated, i.e. if the screen is defined correctly based on the camera and the offset is correctly measured. I plan 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 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.

3.3.2 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 tasks to be on the 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 function as 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.

- **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. It was also good to support different depth cameras other than the Kinect 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 replayed later. This would allow experiments to be reproduced without having to use the device and would allow people to externally verify experiments. This would require a lot of refactorings and would probably take a week or two.

Chapter 4

Evaluation Plan

4.1 Demonstrated Functionality

4.1.1 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). 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.

4.1.2 Renderer

The render 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 compare our simulator.

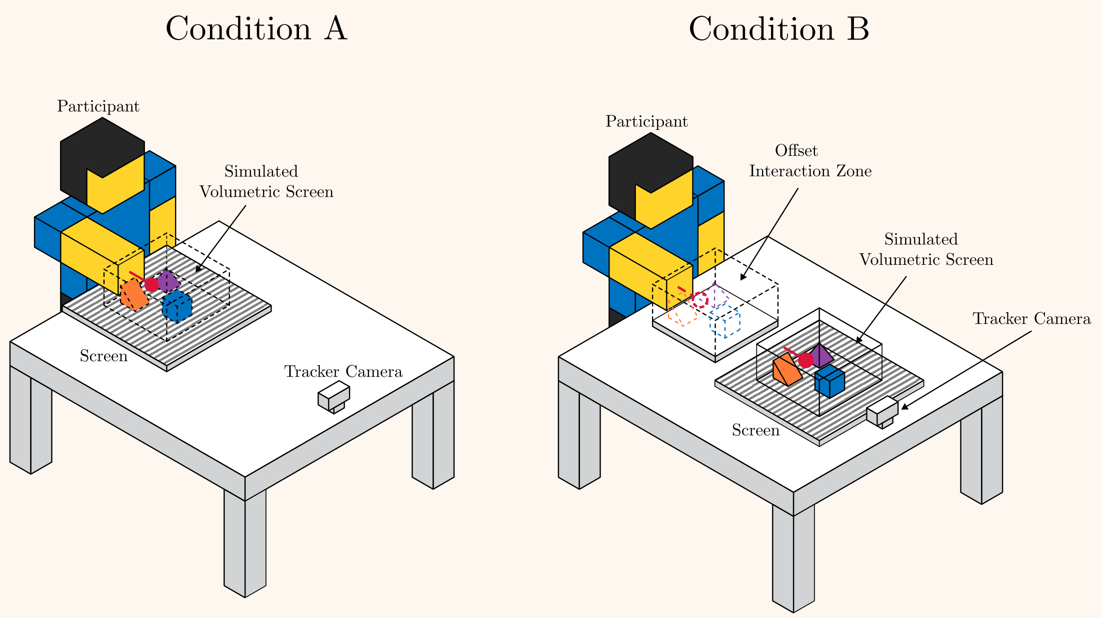
4.1.3 Reproducibility

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).

4.2 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 4.2.1: Teleoperation User Study



We plan to run a user study with two conditions as can be seen in 4.2.1. In the first, condition A, we plan to have the participant interact in a virtual task with their hands directly interacting with the objects where they percieve 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.

4.3 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, Simple, 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 5

Ethical Issues

5.1 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.

5.2 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.

5.3 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 not be more dangerous than other existing technologies.

5.4 Copywrite Limitations

5.4.1 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.

5.4.2 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] Davis E. King. "Dlib-ml: A Machine Learning Toolkit". In: *J. Mach. Learn. Res.* 10 (Dec. 2009), pp. 1755–1758. ISSN: 1532-4435.

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

[22] 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.

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

[24] 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.

[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.