Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

A Virtual Volumetric Screen

Author:
Mr 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

Contents

Chapter 1

Introduction

TODO

Chapter 2

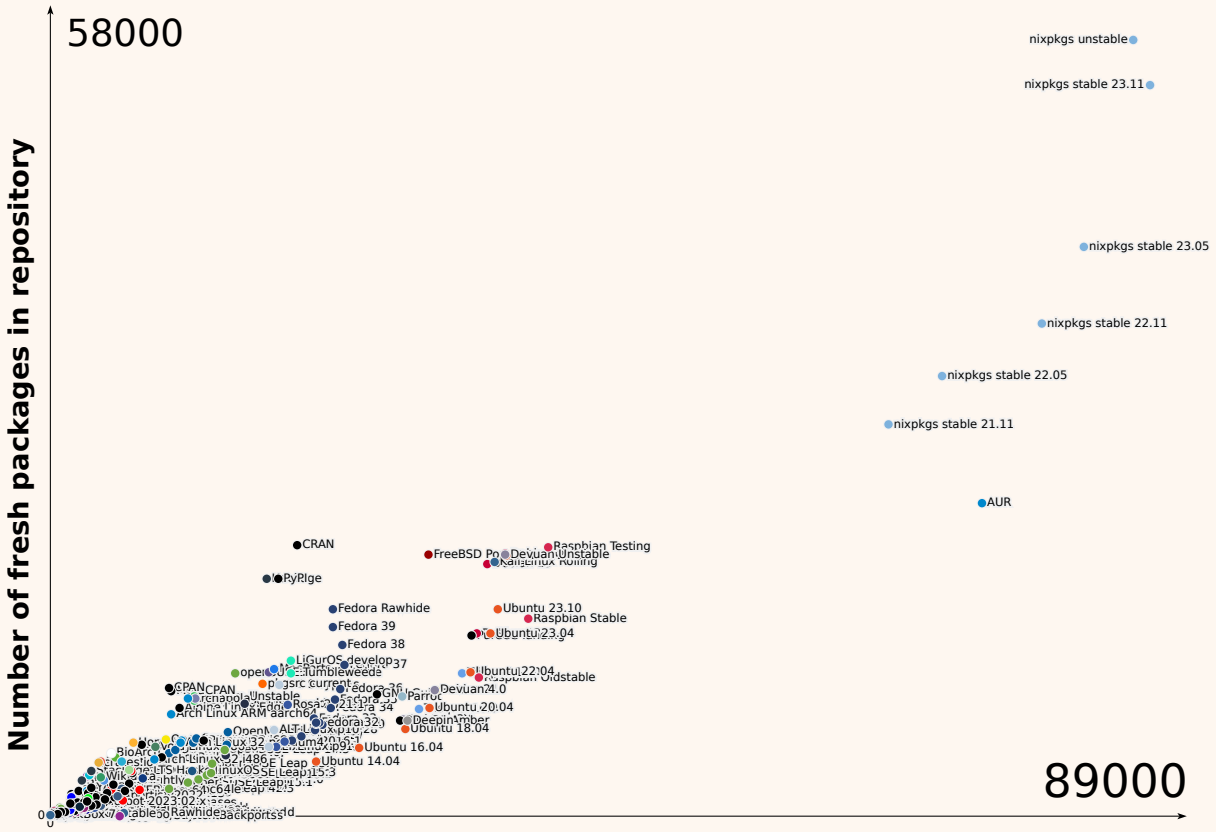
Background

2.1 Nix/NixOS

Figure 2.1.1: the NixOS logo, also know as the Nix Snowflake.

Nix [6] 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 research project Nix [5] is widely used in both industry [1] and academia [4] [15] [3], and its associated public package repository `nixpkgs` [11] as of Jan 2024 has over 80,000 unique packages making it the largest up-to-date package repository in the world as can be seen in Fig 2.1.2. Out of Nix has also grown **NixOS** [7] 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.

Figure 2.1.2: Comparison of all package repositories from Repology.org [16]



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 all the way 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 2.1.3.

Figure 2.1.3: 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 insure 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 non reproducible files and the network [20]. 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 exact same way in the future, regardless of when and where it is used.

These features provide extremely useful for scientific work, CERN uses Nix to package the LHCb Experiment because it allows software to be stable for long periods of time (longer than ever 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 [3].

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 2.1.4.

Figure 2.1.4: Nix Build Loop

1. A hash is computed for the derivation and, using that hash, generate 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 it, 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-realised packages whenever possible and building only what is necessary.

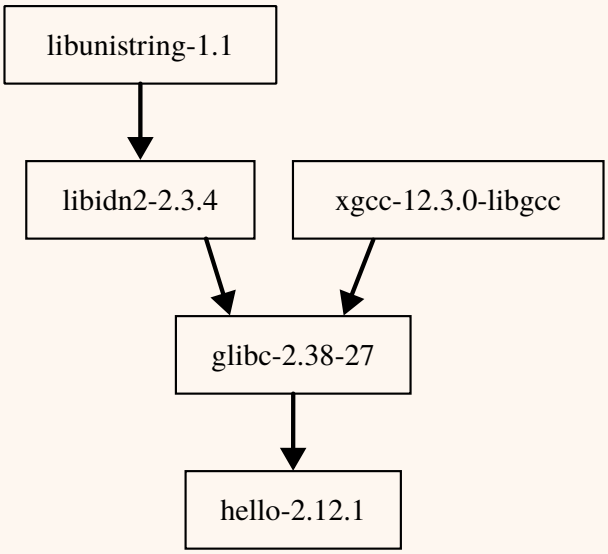
Listing 2.1.1: flake.nix

```
{
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;
# 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";
};
};
}
```

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

  [shell:~]$ TODO get nix depencies
/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
```

Figure 2.1.5: Dependency graph



For an example of making our own nix package. I have create a flake in Fig that builds the basic ”hello” package also available on `nixpkgs`.

Highlighting particular lines in 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 really 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 initial release. The first time I 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 resolves all inputs and then 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 referring 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 really wanted to you could compile different parts of the code with different versions of gcc.

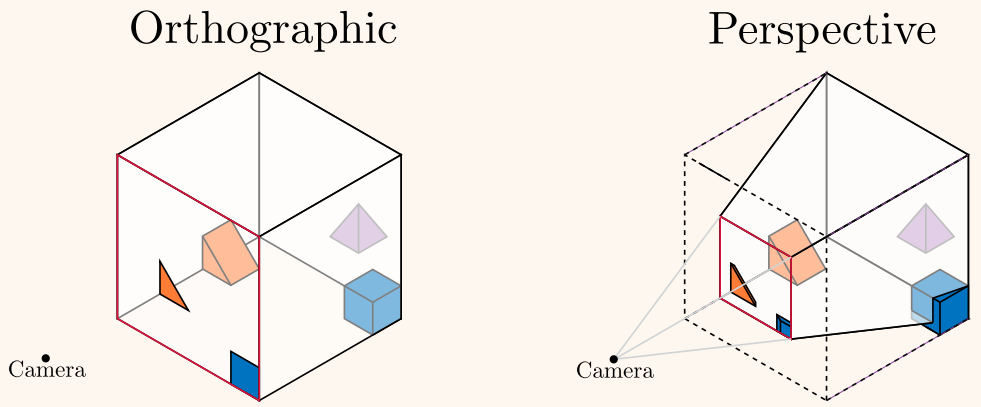
Line 15: Here we are slightly abusing the configure phase to generate a hello.c file. Each phase is essentially run as a bash script. Everything inside `mkDerivation` is happening inside a sandbox and will be discarded once the package is built (techically after we garbage collect).

Line 16: Here we actually build our package

Line 17: In this line we copy our executable we have generate which is currently in the sandbox into the actual package we are producing.

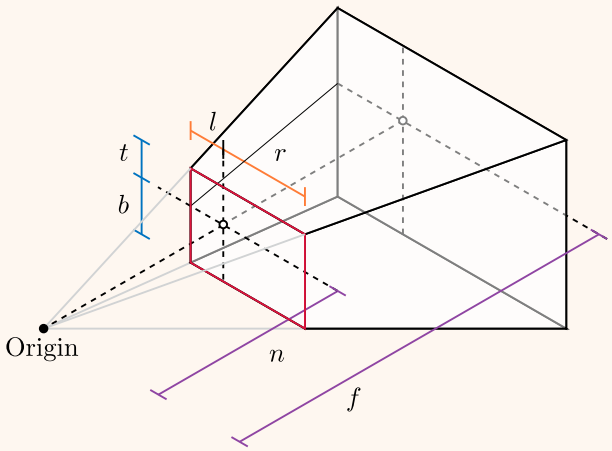
2.2 Perspective Projection

Figure 2.2.1: Orthographic and perspective projections



To represent 3D objects on a 2D surface (our screen) OpenGL support two type 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 accurate and unskewed by perspective (All diagrams in this report are in 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.

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 Fig 2.1 that maps a specified viewing frustum screen-space (with intermediate steps handelled by OpenGL) ToCite . 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.

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

Figure 2.1: frustum perspective matrix

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 bottom often negative and top positive, expressing the extent of the frustum below and above the origin.

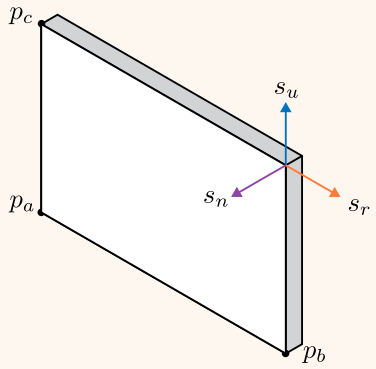
The n and f parameters are scalar values that specify the distances from the origin to the near and far clipping planes along the view direction. Altering the value of n will change the angles of the lines (or vectors) that connect the corners of the near plane to the eye, effectively changing the "field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

If we are able to track the position of a viewers 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 "Generalised Perspective Projection" to calculate f, l, r, b, t, n as the viewers eye moves [14].

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 co-ordinate system of the device we use to track the eyes. These 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



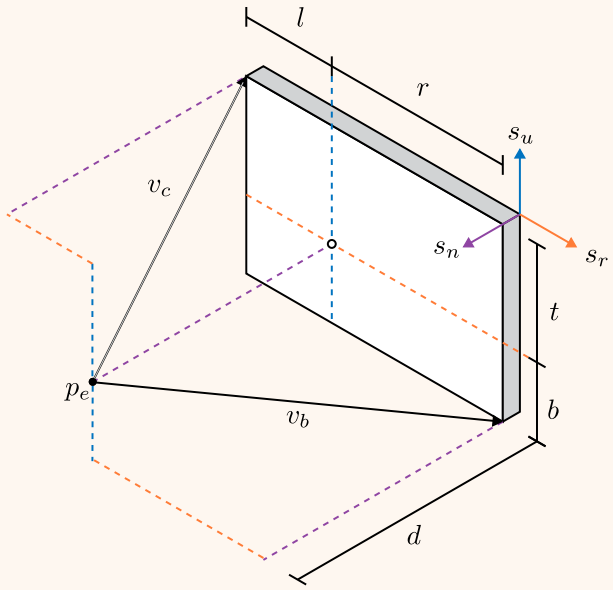
Introducing the viewers eye which we will refer to as p_e. We can draw three vectors v_a, v_b, v_c from the viewers eye p_e to the corners of the screen p_a, p_b, p_c as seen in Fig 2.2.4. In the diagram we also have labelled the components of each of these vectors in the basis of the screen. We can 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 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 any 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:

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

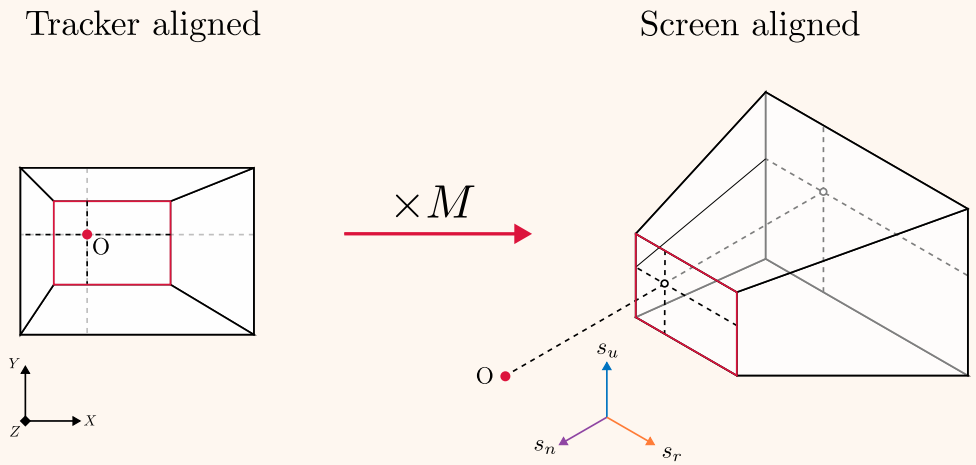
Figure 2.2.4: Screen Intersection with view



We can now generate a projection matrix by calling `frustum` using d as our nearClipping plane distance n with an arbitrary value for the farClipping plane f. 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 remerdy 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:

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

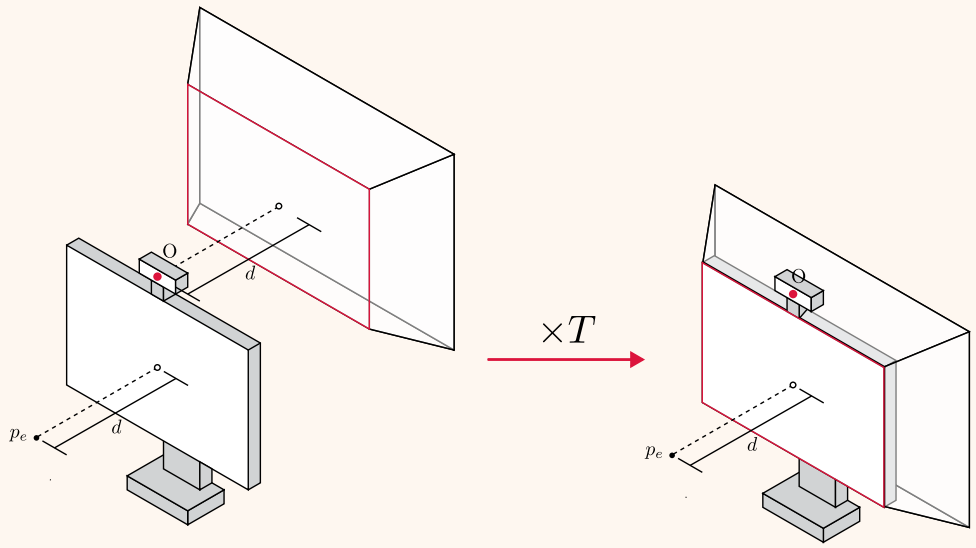
Figure 2.2.5: Moving the frustum from tracker space to screen space



The second problem we have is that we want our projection matrix to move around with the viewers 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 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

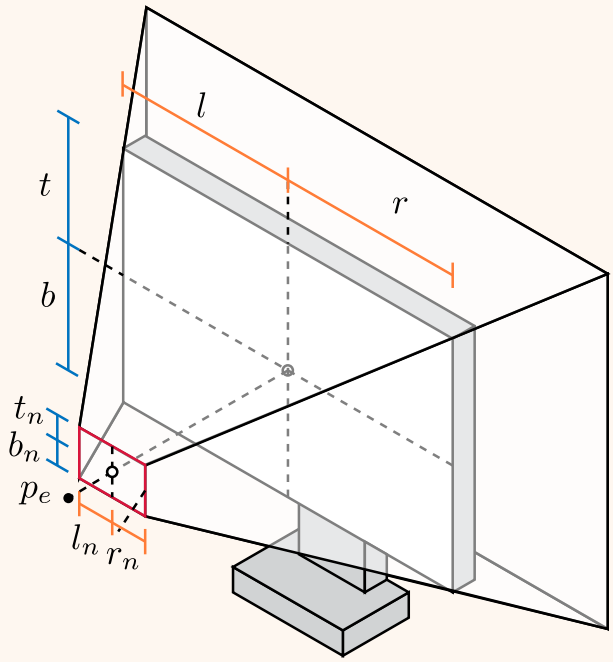


We now have a working method for projecting virtual objects behind our screen onto our screen however it is also possible if we desire to project objects in front of the screen onto the screen as well as long as they lie within the pyramid formed between the edges of the screen and the viewers eye. We can use similar triangles to scale near clipping plane from the plane of the screen to a small distance n from our eye as seen in Fig 2.2.7. 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



Below I have attached some sample code of a function implementing the process we just described.

Listing 2.2.1: 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;
    }
```

Figure 2.2: c++ Sample code for creating the 3D illusion projection

2.3 3D displays

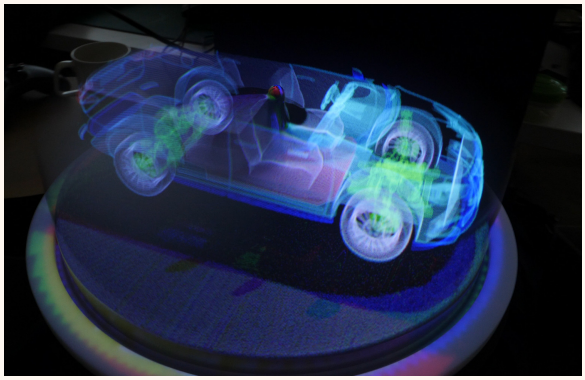
2.3.1 Volumetric Displays

Volumetric displays [9] 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.

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 includes using a rotating mirror [10], emitting screen typically an LED [12], or transparent projector screen [13]. There currently exist commercial products that implement this as can be seen in Fig 2.3.1.

Figure 2.3.1: Two different types of swept volume display



a) The VXR4612 3D Volumetric Display, a projector based persistence of vision display produced by Voxon Photonics.

b) A Volumetric Display / Holographic Signage, a LED based persistence of vision display produced by Brightvox Inc.

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 [19], lasers and phosphorus gas [21], or a transparent laser induced damaged medium that can be projected into [17].

Issues

Expensive: Swept volume displays require extremely high refresh rate projectors, which are expensive and difficult to manufacture or transparent LED screens which are only recently becoming widely available. For example the Voxon VX1 one of the few if only commercial available volumetric displays costs $11,700 USD [18].

High bandwidth requirements: To render objects in real time at 2D display equivalent resolutions while taking a raw voxel stream (as apposed to calculating voxels on hardware from primitive shapes) has an extremely high bandwidth requirement. `60fps` on a $4096 \times 2160 \times 1080$ voxel display with `24 bit` color requires a bandwidth of 1.37×10^3 bits per second/13.7 terabits per second. 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 a based on the historical trend of display bandwidth that a volumetric 4K screen will become feasible in 2060.

Volumetric Screen Simulations

Because of the high cost and bandwidth requirements of volumetric displays, there has been a lot of research into simulating volumetric displays on traditional 2D displays. This is typically done by rendering a 3D scene from multiple perspectives and then combining them into a single image. [8]

Chapter 3

Project Plan

3.0.1 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 of 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 a quite arbitrary package set which has not been packaged for nix yet as far as I was aware. I had experience using open source heading tracking for video games. It thought it might be 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 hardcoding, 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 hardcoded asset paths to ensure compatibility across various systems.

- **October 30th:** I deciced I would attempt to try and package Azure-Kinect-SDK in nix giving myself a 1 week cutoff, 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. Microsofts proprietary `libdepthengine.so` library that is used for sensing depth from the camera was clearly 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 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 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 source ToCite 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 towards 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 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 which 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.

Chapter 4

Evaluation Plan

TODO

Chapter 5

Ethical Issues

TODO

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] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

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

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

[15] 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.

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

[17] S.K. Nayar and V.N. Anand. “3D Display Using Passive Optical Scatterers”. In: *IEEE Computer Magazine* 40.7 (July 2007), pp. 54–63.

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

[19] 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.

[20] 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).

[21] 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.