Rendering assignment: 4D geometry

Ashley Smith

April 14, 2020

Contents

1 Background description (10%)

For this assignment, I have chosen to learn about and investigate the fourth dimension with respect to graphics. Most games are typically set in a two or three dimensional world and the fourth dimension is rarely discussed as part of a game's core design, if discussed at all. I aim to create an interactive visualisation of four-dimensional geometry in an effort to educate both myself and others how the fourth dimension may be applicable to graphics used in video games or other media outlets.

1.1 2D to 3D: Why do dimensions matter?

A game's graphics, character controls, world concepts and designs and even a game's genre are affected by the number of dimensions a game has – this can be seen in Hopoo Games' Risk of Rain (2013) and its sequel Risk of Rain 2 (Hopoo Games, 2019) shown in figure 1. In two-dimensional games, characters are generally confined to two axes and are therefore considered to be top-down or side-on; Risk of Rain is a platformer and makes use of a side-on perspective. Because of this, enemies line up and then stack on top of each other to attack the player, who's only choices of evasion are to run away or jump over the enemies. Risk of Rain 2 however is 3D, meaning that the enemies attempt to surround the player and the player can escape in multiple directions while also retaining their option to try and jump over enemies.

Figure 1: A 3D sequel to the 2D game *Risk of Rain* (Hopoo Games & Community, 2019)

It could be said that the transition to 3D was good for Risk of Rain (2013), however, there are many features in the first game that couldn't be brought into the game. Some player abilities that shot in straight lines to hit multiple enemies wouldn't be as effective if the enemies weren't lined up or stacked on top of one another; ladders and other platforming elements that took advantage of the space on a player's screen would look awkward and cramped in an otherwise-spacious 3D environment, meaning that the player is mainly stuck running around on the ground; and finally, while the player can see more things in front of them without the constraint of one's monitor size, enemies can easily spawn or wander behind the player and perform surprise (or sometimes *unfair*) attacks and ending the player's game. Despite these shortcomings, Risk of Rain 2 (2019) is considered a well-received improvement to the original (Hopoo Games & Community, 2019).

1.2 3D to 4D: What will be achieved from this project?

If the third dimension can create new opportunities and conceptualise new gameplay and new mechanics, it does make one wonder about the possibilities that the fourth dimension may bring to gaming. With virtual reality and other technologies becoming more accessible and accepted as avenues of entertainment, the fourth dimension may not seem so experimental. In this project, I wrote my own 4D geometry viewer in an effort to lessen the gap between the third and fourth dimensions. While this project doesn't feature anything new, it is hoped that the development of this project is just as insightful and interesting as the end result and that the approaches used to build the application will be of use to future developers wanting to pioneer the fourth dimension.

Video games are just as much about the feel of the world as they are about the look — being able to perceive the world from different positions and angles and interact with it in various ways both contribute to how well a world has been crafted. I feel that this project benefits from user interaction in a similar way and will help users understand the fourth dimension better.

Instead of just displaying a 4D object, I wanted to allow the user to be in control of how the object is looked at and how it is transformed. These interactions are made with both keyboard shortcuts and an interactive menu. Users can move and rotate the camera to examine the 4D object, while a menu can be used to apply various translations, rotations and scales to the object. The combination of each of these transformations can be seen in a 5x5 matrix and their effects are applied to the 4D object and therefore forming the connection between how an object is transformed and how it appears — something which may be confusing in the context of the fourth dimension.

1.3 4D to 2D: The challenges of visualising 4D

Visualising the fourth dimension in simulations and games isn't simple. 2D games can be seen as simplified 3D worlds, especially when art and rendering techniques are used to try and mimic the 3D world as seen in classic arcade games like After Burner (Sega AM2, 1987) as shown in figure 2. Humans are three-dimensional beings and as such we feel familiar with worlds that appear to be 3D even when looking at them through a two-dimensional display. When a 3D game is rendered, matrix mathematics is employed to translate coordinates in world space to screen space and therefore render a 3D world onto the screen. 4D objects and worlds are incomprehensible to beings who can only perceive the third dimension, meaning that there is always going to be a sense of esoterism when working with anything beyond three dimensions. Moreover, 4D geometry needs to be translated into 2D geometry in order to be rendered on screen, creating an additional barrier to visualisation.

Figure 2: 2D graphics mimicking the 3D world in After Burner (Awesome Daily Staff, 2016)

Firstly, in order to see a 4D object, a 4D object needs to be created. The tesseract is a 4D hypercube and is suitable as the main polytope for this project. For this project, I want to perform projections and other transformations using GPU shaders like you would for an ordinary game. This is in contrast to the approach Hollasch (1991) and many other article writers chose; they transform the vertices in `C++` and then give those to the GPU instead. Secondly, in order to view the fourth dimension, a 4D camera can be used to view the fourth dimension from various positions and angles. Next, a projection matrix is used to project vertices into the third

dimension, where it is then perceived by a separate, 3D camera and then finally projected to 2D for rendering on screen. Getting all of these steps correct is difficult as they cannot be worked on independently and tested easily — it is the sum of these steps that achieve even the simplest result.

2 Standout elements (10%)

My largest achievement with this project was that I actually managed to render a tesseract, a 4D hypercube. Given a set of 4D vertices and a list of indices, any poly in the 4th dimension and below can be rendered in this project. Changing the 3D and 4D cameras alter the perspective of the shape which is handled by the GPU as it would be in a normal video game. This was extremely difficult at first because of homogenous coordinates.

OpenGL interprets coordinates to be homogenous when rendering (Woo, Neider, Davis, & Shreiner, 1999) meaning that a vector $v = (5, 4, 3, 2)$ is divided by it's final component w such that $v' = (\frac{5}{2}, \frac{4}{2}, \frac{3}{2}, \frac{2}{2})$. Figure 3 illustrates how the 4th component of a 4D vector is used to scale vertices to give the deceiving appearance of a tesseract when it is actually just two interconnected cubes with a difference in scaling. Overcoming this required a deep investigation of the 4D to 3D projection matrix and why it wasn't eliminating the 4th component of vertex coordinates.

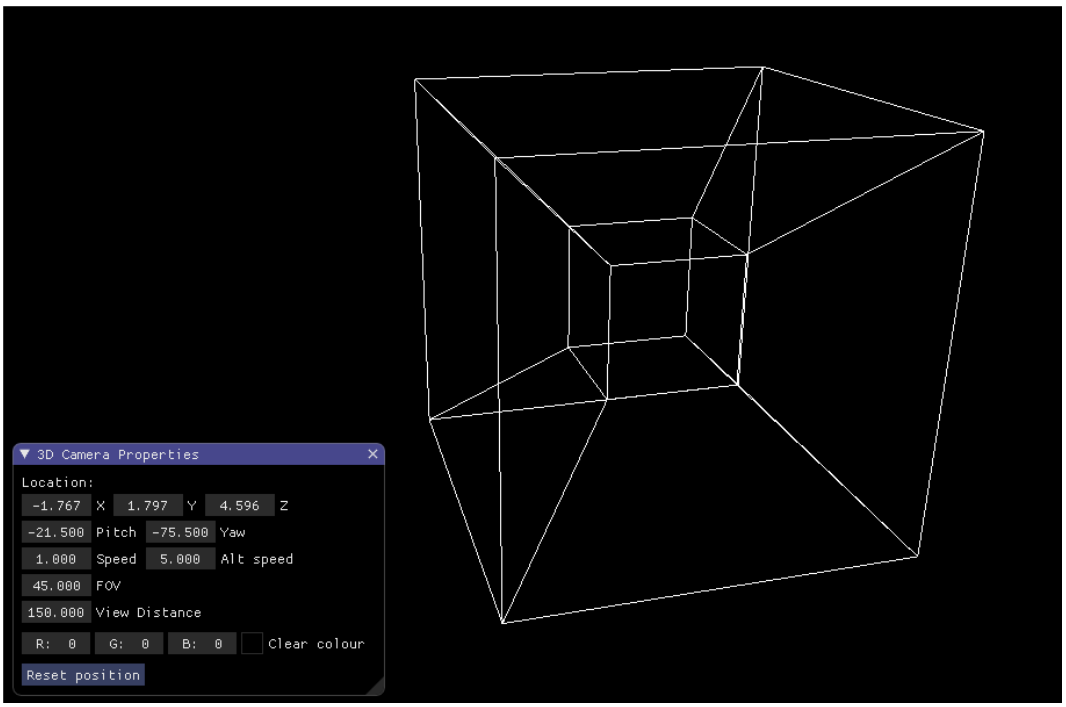


Figure 3: Homogenous coordinates creating the appearance of a tesseract

Another feature I'm proud of is the summary window. For each polytope there is a set of information that describes the number of vertices, edges, faces and cells of not only the 4D shape but also each variant of the same polytope such as a line, a square and a cube — while this information is trivial, being able to compare the geometry between each shape does help create an understanding of what a higher dimension means. Furthermore, clicking the name of each shape scales the rendered polytope such that a cube would get flattened into a square, for instance. I hope that small details like this inspire interest in the user to learn more about the relationships between dimensions.

Finally, I'm very proud of the fact that I created this application from scratch while maintaining cross-platform support. Until this project, I had never set up a 3D application from scratch and so learning and creating my own abstractions for use in the graphics pipeline was very fulfilling. The CMake build tool (Cedilnik, Hoffman, King, Martin, & Neundorf, 2000) allows a developer to generate an appropriate `C` or `C++` project file such as `.sln` for compilation on various platforms, while the use of GLFW (The GLFW Development Team, 2016), OpenGL (Khronos Group, 1992), GLM (G-Truc, 2005) and ImGui (Cornut, 2014) meant that there were no dependencies on any given operating system and therefore any desktop should be able to compile it. This is a personal victory as it means that anyone interested in seeing and experimenting with 4D geometry can compile my program and take inspiration from the source code.

3 Design (30%)

The bulk of the project is split between three core areas:

- **The viewer:** The main scene containing interactive interfaces, polytopes, cameras and all the logic necessary for communication between them.
- **The polytopes:** The generation and usage of various polytopes such as simplices and hypercubes as well as the common interface used to polymorphically interact with them.
- **The pipeline:** How a set of coordinates are given to the GPU and when and how they are transformed by GLSL shader code for on-screen rendering.

3.1 The viewer

3.1.1 Menubar

When the application is first opened, the `Viewer` is the first thing the user is presented with. The `Viewer` class is defined in `Viewer.h` and acts a scene capable of handling input and being updated and rendering every frame. At the top of the screen there is a menu bar with two sub-menus: the first allows you to change which polytope is being rendered and the second allows you to close and re-open various windows. I tried to avoid hiding functionality from the user for such a simple project, so this this is sufficient for a menu bar.

3.1.2 Camera windows

There are two camera windows, one that is for the 3D camera and another for the 4D camera. Both cameras are required for viewing the fourth dimension but the 3D camera is the one that the user will be most familiar with. The 3D camera can be controlled using the keys shown in table 1. The 4D camera cannot be piloted in the same way as it simply wouldn't make sense.

Keyboard shortcut	**Description**
`W, A, S, D`	Move forwards, left, backwards and right respectively
`Space, C`	Move upwards and downwards in relation to the camera
`Right mouse button`	Look around the scene while held

Table 1: 3D camera controls

While moving the 3D camera acts as you'd expect, moving and turning the 4D camera can appear to twist and deform the object without necessarily moving it. While this project doesn't include any theories about the meaning of the fourth dimension, one way to think about it is looking at the same object but from a different point in time — it would not move, but simply changing the time of observation can impact an object one is looking at.

3.1.3 Polytope information window

As mentioned in section 2, another part of the viewer is the information window which shows a table of different shapes belonging to the selected polytope. Clicking on the name of a shape scales the rendered object so that it resembles the same shape — multiplying the y and z components of cube vertices would give you a line segment for instance, as only x would be able to vary. These transformations are applied smoothly for the users pleasure. Figure 4 shows this window displaying the information for hypercubes up to the 4th dimension.

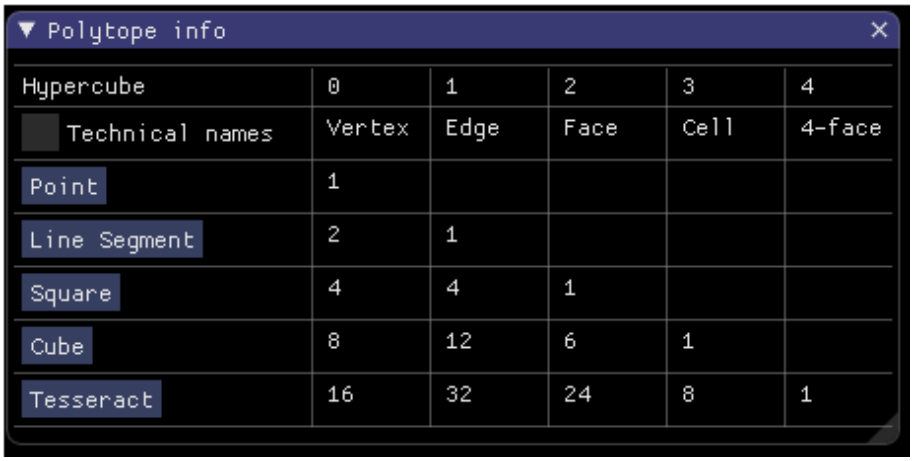


Figure 4: Polytope information window detailing different hypercubes

3.1.4 Transformation window

The final window is the transformation window shown in figure 5. In this window, the user can specify different types of transformations to be applied to the object — the product of all these transformations can be seen in the matrix at the top of the window. This is the exact matrix that is sent to the GPU along with the view and projection matrices from the cameras. There are multiple tabs in the transformation window to modify each transformation separately. While they are mainly self-explanatory and just involve dragging values to increase or decrease them, there are some notable features of some transformations.

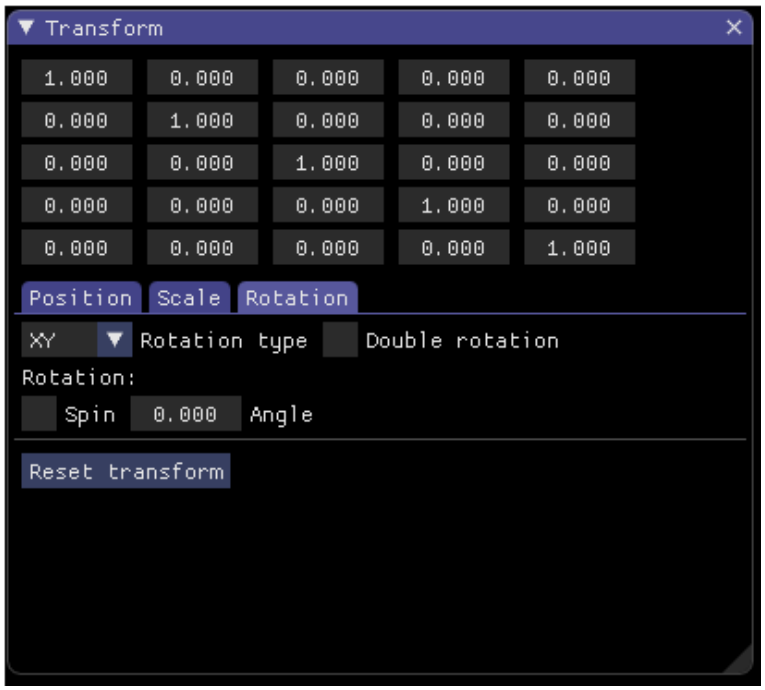


Figure 5: The transformation window on the rotation section

When configuring the rotation matrix, the user can specify whether they want to manually set the angle or whether they want the object to automatically spin. The user then chooses a plane to rotate — in 3D, if you want to rotate around the y axis, then the xz plane is what needs to change. In 4D, rather than rotating around an axis, a plane is used instead and so the user can specify which pair of axis they wish to manipulate. Finally, in 4D it is possible to perform 2 rotations with the same time with separate planes (such as xz and yw), therefore in this project the user can perform the infamous '4D double rotation', along with applying independent, automatic spins on each rotation.

When translating the matrix in 4D, the object doesn't move as expected. While translating in 3D simply moves the object to a different location, moving in 4D distorts the object in a way that seems incorrect. However, by performing 4D camera movement and a 4D translation, one can verify that these are indeed correct and that it is our expectations that are incorrect.

3.2 The polytopes

There are three polytopes for this project: The *simplex*, the *hypercube* and the *hyperoctahedron*. The construction and final appearance of these polytopes can be seen in appendix A. The next polytope I would include would be the *120 cell*, but with 600 vertices and 1200 edges, constructing a hyperdodecahedron would be very time consuming and its visualisation may be confusing with so many lines drawn — while rendering solid shapes instead of wireframes would make this more attractive to implement, this would be at the cost of detail. Each polytope only handles it's construction and storage of its information as the rest is handled by a common base class defined in the file `Polytope.h`.

This common class stores `vectors` of the objects position and scale as well as some `floats` representing rotation information. Every frame, matrices representing these transformations are recalculated and then combined to form the final transformation matrix that can be seen in the transformation window discussed in section 2. The values that are tweaked with that interface are these variables inside `Polytope`, meaning that each shape's transformations are stored separately from one another.

When a polytope derived from the base class is constructed using its specialised constructor, it is pointing `vertices` and `indices` to the addresses of arrays containing `floats` and `unsigned integers` respectively. When these arrays are set up, `updateVertices` is called to inform `Polytope` how many values have been stored in and how they should be rendered. This means that the `Polytope` base class doesn't know what it's rendering, but rather *how* to render it.

3.3 The pipeline

3.3.1 View matrices

Each frame, the `viewer` handles the users input and then calls `updateTransform` on the polytope, requesting that its transformation matrix should be recalculated in preparation for rendering. When it's time to render the object, view matrices are retrieved from both the 3D and 4D cameras; the 3D camera simply uses `glm::lookAt` to create its view matrix, however the 4D camera's view matrix had to be calculate by hand as is the case with a lot of the other 4D functionality. Consider a simple translation in 3D space:

$$\begin{pmatrix} 1 & 0 & 0 & i \\ 0 & 1 & 0 & j \\ 0 & 0 & 1 & k \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+i \\ y+j \\ z+k \\ 1 \end{pmatrix}$$

Despite being a 3D transformation, the use of a 4x4 matrix and homogenous coordinates is required to produce a translation. Every 4D operation including the transformations in section 3.1.4 were implemented with 5D (as in 5x5) matrices. Since this use-case is so uncommon, most graphics libraries including GLM do not provide 5D vectors, matrices or the operations to build and manipulate them. It is because of this that I could not rely on GLM and had to write my own matrix implementations of the construction and multiplication of these 5x5 matrices. The 5x5 4D view matrix $\mathbf{M}$ can be found by crossing the vectors *up*, $forward$ (found by $target - position$) and *over*; *over* is required for the same reason that *up* is necessary in 3D view matrices and each dimensional axis requires a new 'up' (GSBicalho, 2017, `CameraND.h`).

$$\mathbf{M} = \left\{ \begin{pmatrix} 1 & 0 & 0 & 0 & q_x \\ 0 & 1 & 0 & 0 & r_x \\ 0 & 0 & 1 & 0 & s_x \\ 0 & 0 & 0 & 1 & t_x \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \middle| \quad \begin{aligned} q &= \mid cross3(up, over, t) \mid \\ r &= \mid cross3(over, t, q) \mid \\ s &= \mid cross3(t, q, r) \mid \\ t &= \mid target - position \mid \end{aligned} \right\}$$

Note that the use of 4D vectors enables the usage of the triple-cross-product. In 3D, two vectors are inserted and cross-multiplied in a 3x3 matrix where elements in the final column is taken as the output, meaning that in a 4x4 matrix, 3 4D vectors can be multiplied in the same way:

$$cross3(a,b,c) = \begin{vmatrix} a_x & b_x & c_x & \mathbf{i} \\ a_y & b_y & c_y & \mathbf{j} \\ a_z & b_z & c_z & \mathbf{k} \\ a_w & b_w & c_w & \mathbf{l} \end{vmatrix}$$

3.3.2 Projection matrices

With the model matrix being created via user-defined transformations and the view matrices being calculated by the cameras, the only matrices remaining are the ones that handle projecting from a higher dimension to a lower dimension. While the 3D to 2D projection was achieved trivially with the help of GLM's `glm::perspective` function, projecting 4D to 3D is a more complex process.

There isn't necessarily a correct solution for 4D to 3D projection as we cannot comprehend what the 4th dimension should look like; it's easy to determine whether the perspective used in art is unrealistic because we can compare it to real life, but we cannot compare what we see in 3D to what we expect to see in 4D. Schloss (2016, 2:42) created an insightful video showing how a tesseract can be projected to 3D by projecting its shadow into the third dimension, a technique known as a stereographic projection. Schloss proposed the following projection matrix $\mathbf{P}$ where l_w is the location of the light source along the 4D axis and p is the point to be projected:

$$\mathbf{P} = \begin{pmatrix} \frac{1}{l_w - p_w} & 0 & 0 & 0 \\ 0 & \frac{1}{l_w - p_w} & 0 & 0 \\ 0 & 0 & \frac{1}{l_w - p_w} & 0 \end{pmatrix}$$

Since w is the vertex of the 4D object to be projected, this matrix needs to be calculated for every vertex in the shape, therefore this calculation needs to happen after the vertices have been transformed by both transformation and view matrices. As discussed in section 1.3, this project is going to use shaders to take advantage of the GPU, meaning that the construction of the projection matrix $\mathbf{P}$ along with the ability to multiply 5x5 matrices and 4D vectors needed be implemented with GLSL inside the vertex shader — this can be seen in appendix B. Everything is now taken into consideration and rendering a 4D object is the same as it would be for any 3D object: binding buffers, sending vertex and index data to the GPU, passing uniforms to the shaders and finally drawing the elements based on the information stored in the `Polytope` class.

4 Implementation (10%)

Development began with the foundations of the program — creating a window using GLFW and creating the core loop of the program to create opportunities for event handling and object updating and rendering. The first milestone was to render a cube centered at $(0, 0, 0, 0)$ on screen and then turn it into a tesseract. As described in section 2, simply rendering a cube with 4D coordinates didn't work even though it gave off the impression that it did. In order to incorporate a 4D camera the 4D coordinates needed to be interpreted as such and not just as the scaling factor for homogenous coordinates.

Despite the tesseract not being rendered in the intended way, there was a 3D object being rendered on screen and so work was put into the 3D camera. I learned that while setting up the camera, OpenGL's clipping space was actually left-handed where the z-axis extends towards the camera and not into the horizon when the x axis extends to the right. It was very difficult to get my bearings when the world is completely black and the tesseract is symmetrical and I continuously thought about whether camera was facing the wrong direction, my matrix multiplication function was incorrect, my usage of row-major or column-major matrices was incorrect or if I was cross-multiplying my vectors the wrong way round. This simple misunderstanding depleted a lot of hours of development time.

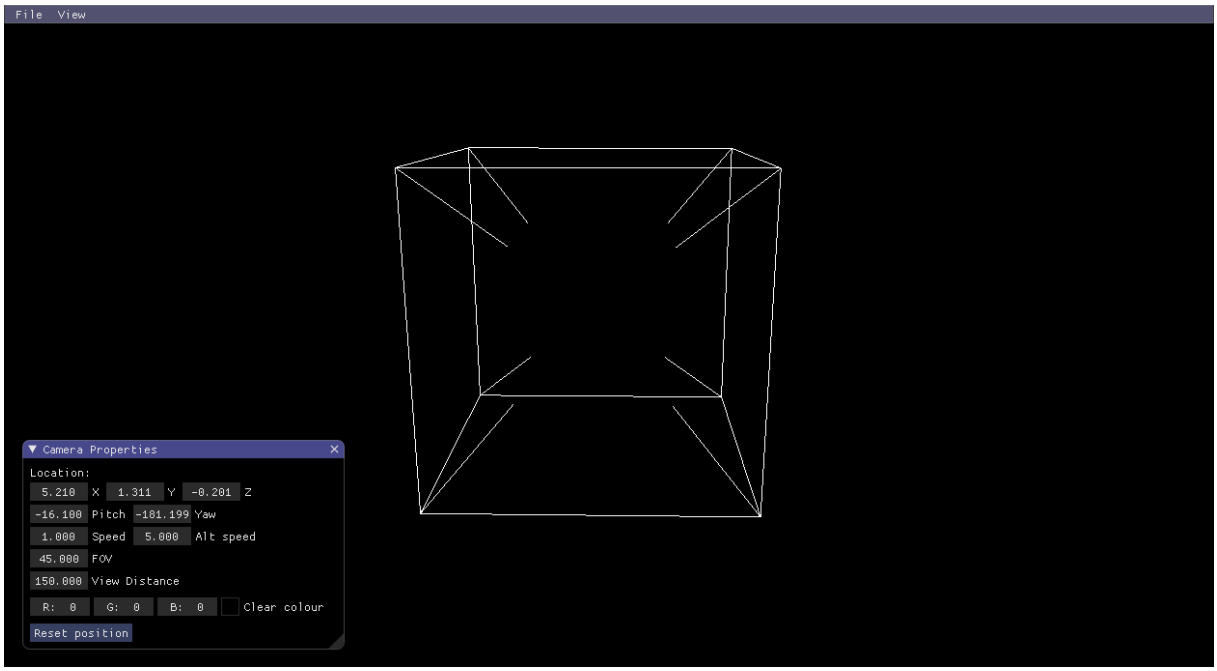


Figure 6: The inner cube totally disappears 'into the fourth dimension'

After the 3D camera was fixed and could be piloted as expected, it was time to get the 4D camera functioning correctly. The 4D camera's initial implementation made the tesseract completely disappear indicating that there were numerous issues occurring — I had no idea what was the 'correct' way to do this and I had to perform a lot of experimentation. I had no idea whether the use of OpenGL or GLM meant that information was inapplicable, or whether my vertices were somehow constructed or buffered incorrectly. After shifting the w components of the tesseract from $w \in \{-1, 1\}$ to $w \in \{0, 1\}$, I finally had something rendering on screen. Figure 6 shows the perplexing issue of how non-zero w components of the tesseract's vertices disappeared.

Since this was my first glimpse into the effects of rendering non-homogenous 4D coordinates I found this result exciting despite it being a bug. At first I suspected the bug to be related to the interaction between the near and far planes of the 4D camera and the fourth dimension, but it was actually my projection. Projection matrices eliminate a component of a vector by being sized differently; multiplying a 4x3 matrix with vector of length 4 would produce coordinates containing 3 elements instead. I initially had my projection set up as a 4x4 matrix and accepted the output to be a 3D coordinate written in a homogenous way and so I scaled the entire vector by its w component. After altering my projection matrix produce `vec3`s, the tesseract was rendered although it was skewed due to no longer being centered around $(0, 0, 0, 0)$. This was fixed by reverting the w component in the tesseract's vertices such that $w \in \{-1, 1\}$.

After rendering the initial tesseract, it was time to allow the user to manipulate the tesseract's transformation matrix as well as introduce the different polytopes shown in appendix A. The `simplex`, `hyperoctahedron` along with the existing `hypercube` were the three polytopes I settled on as the 4D shapes of other polytopes would require a project of similar magnitude to this one to generate. The `Polytope` class was then introduced to give the three chosen polytopes a base class to override as discussed in section 3.2 so that each polytope had an opportunity to construct itself and fill out the information window. I believe this was the most appropriate way of including new shapes as the only thing that separates how a cube and an octahedron are rendered is the geometry itself. This class would be more than capable of handling further 4D geometry in the future if shapes like the 120-cell were included.

With that, I had achieved all the objectives I set out to meet for this project as shown in figure **??**. If I had managed to fix the issues with the 4D view and projection matrices sooner, I would have investigated constructive solid geometry (CSG) as part of the project. CSG allows you to add and subtract meshes from other meshes to create conjoined shapes and holes; this project would have then been able to deform the 4D object such that for any value of w the shape would render a 3D slice of the 4D object. This couldn't be done without CSG as it requires the discovery of vertices between buffered vertices in order to create new shapes, something that is far beyond the scope of this project and would require substantially more time to implement.

Bibliography

Awesome Daily Staff. (2016). The 20 most awesome arcade games and arcades of all time. Retrieved April 14, 2020, from https://theawesomedaily.com/20-most-awesome-arcade-games-and-arcades-of-all-time/

Cedilnik, A., Hoffman, B., King, B., Martin, K., & Neundorf, A. (2000). Cmake (Version 3.17.0). Retrieved April 12, 2020, from https://cmake.org/

Cornut, O. (2014). Immediate mode GUI (ImGui) (Version 1.75). Retrieved April 12, 2020, from https://github.com/ocurnut/imgui

G-Truc. (2005). OpenGL Mathematics Library (GLM) (Version 0.9.9.7). Retrieved April 13, 2020, from https://glm.g-truc.net/0.9.7/index.html

GSBicalho. (2017). TrueNgine. Retrieved April 13, 2020, from https://github.com/GSBicalho/TrueNgine

Hollasch, S. R. (1991). Four-space visualization of 4d objects. *MS Dissertation, Arizona State University.*

Hopoo Games. (2013). Risk of Rain.

Hopoo Games. (2019). Risk of Rain 2.

Hopoo Games, & Community. (2019). Store page and reviews for the game Risk of Rain 2. Retrieved April 11, 2020, from https://store.steampowered.com/app/632360/Risk_of_Rain_2/

Khronos Group. (1992). Open Graphics Library (OpenGL) (Version 4.6). Retrieved April 12, 2020, from https://opengl.org/

Schloss, J. (2016). Understanding 4D – The Tesseract. Retrieved April 13, 2020, from https://www.youtube.com/watch?v=iGO12Z5Lw8s

Sega AM2. (1987). After Burner.

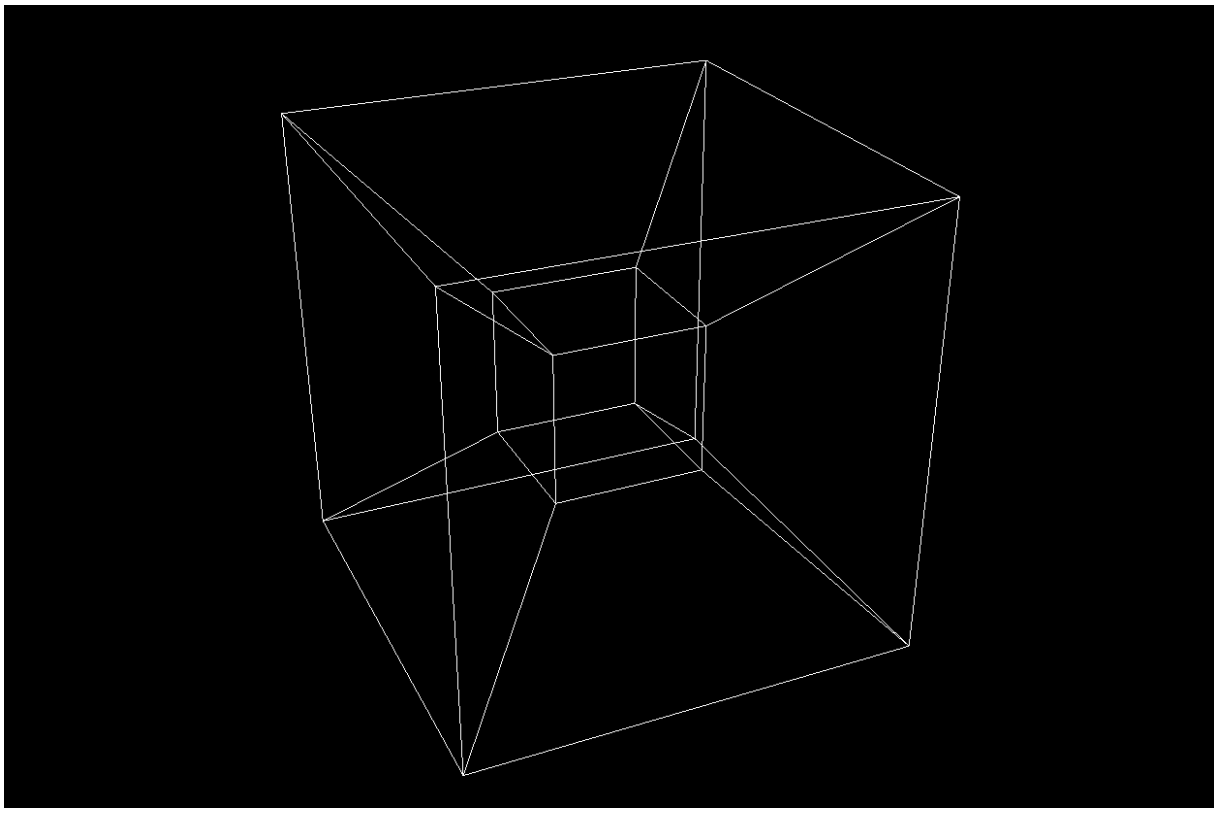
The GLFW Development Team. (2016). Graphics Library Framework (GLFW) (Version 3.3.2). Retrieved April 12, 2020, from https://www.glfw.org/

Woo, M., Neider, J., Davis, T., & Shreiner, D. (1999). *Opengl programming guide: The official guide to learning opengl, version 1.2.* Addison-Wesley Longman Publishing Co., Inc.

Appendices

Appendix A Polytopes

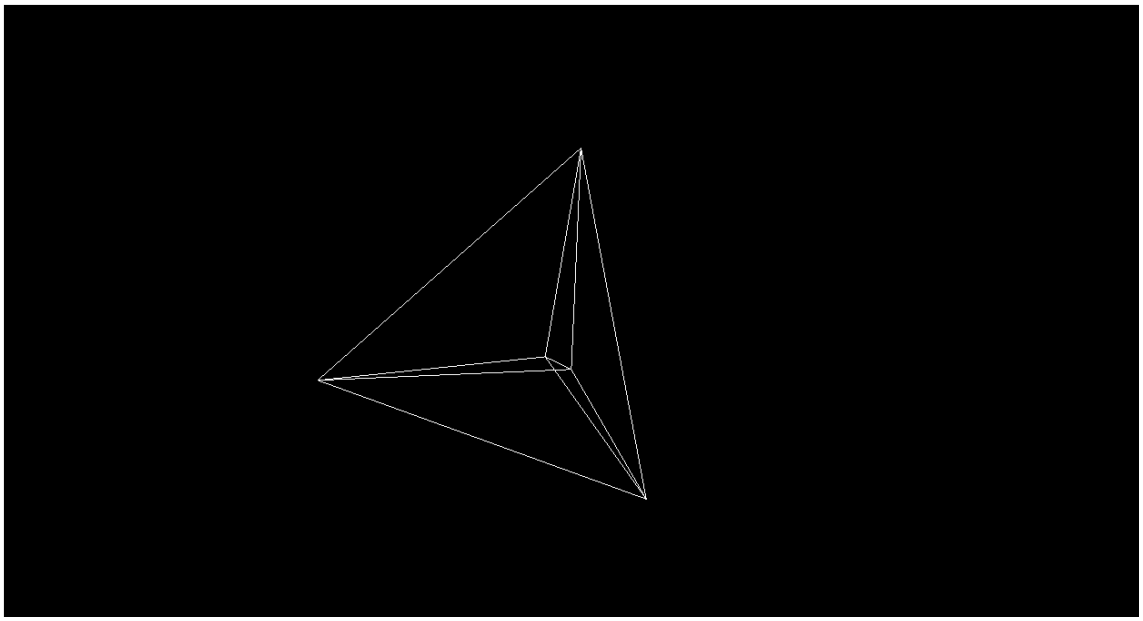
Hypercube



```
1  // Constructor
2  App::Hypercube::Hypercube() : Polytope() {
3
4    // Generate vertices
5    vertices = new float[2 * 2 * 2 * 2 * 4];
6    unsigned int vertexCount = 0;
7
8    // Create a tesseract
9    for (int w = -1; w < 2; w += 2) {
10     for (int z = -1; z < 2; z += 2) {
11       for (int y = -1; y < 2; y += 2) {
12         for (int x = -1; x < 2; x += 2) {
13
14           // x, y, z, w
15           vertices[vertexCount++] = (float)x;
16           vertices[vertexCount++] = (float)y;
17           vertices[vertexCount++] = (float)z;
18           vertices[vertexCount++] = (float)w;
19         }
20       }
21     }
22   }
23
```

```
24    // Generate indices (lines)
25    indices = new unsigned int[32 * 2];
26    unsigned int indexCount = 0;
27
28    // Connect all tesseract vertices
29    for (int w = 0; w < 2; ++w) {
30
31      // Find which cube this vertex belongs to (inner or outer)
32      const unsigned int origin = w * 2 * 2 * 2;
33
34      // Top face
35      indices[indexCount++] = origin;
36      indices[indexCount++] = origin + 1;
37      indices[indexCount++] = origin + 1;
38      indices[indexCount++] = origin + 3;
39      indices[indexCount++] = origin + 3;
40      indices[indexCount++] = origin + 2;
41      indices[indexCount++] = origin + 2;
42      indices[indexCount++] = origin;
43
44      // Left face
45      indices[indexCount++] = origin;
46      indices[indexCount++] = origin + 4;
47      indices[indexCount++] = origin + 4;
48      indices[indexCount++] = origin + 6;
49      indices[indexCount++] = origin + 6;
50      indices[indexCount++] = origin + 2;
51
52      // Right face
53      indices[indexCount++] = origin + 1;
54      indices[indexCount++] = origin + 5;
55      indices[indexCount++] = origin + 5;
56      indices[indexCount++] = origin + 7;
57      indices[indexCount++] = origin + 7;
58      indices[indexCount++] = origin + 3;
59
60      // Bottom face
61      indices[indexCount++] = origin + 4;
62      indices[indexCount++] = origin + 5;
63      indices[indexCount++] = origin + 6;
64      indices[indexCount++] = origin + 7;
65    }
66
67    // Identify and connect vertices in both inner and outer cubes
68    for (int z = 0; z < 2; ++z) {
69      for (int y = 0; y < 2; ++y) {
70        for (int x = 0; x < 2; ++x) {
71          const unsigned int innerCube = x + (y * 2) + (z * 2 * 2);
72          const unsigned int outerCube = innerCube + 2 * 2 * 2;
73          indices[indexCount++] = innerCube;
74          indices[indexCount++] = outerCube;
75        }
76      }
77    }
78
79    // Push generated vertices to these buffers
80    updateVertices(GL_LINES, vertexCount, indexCount);
81  }
```

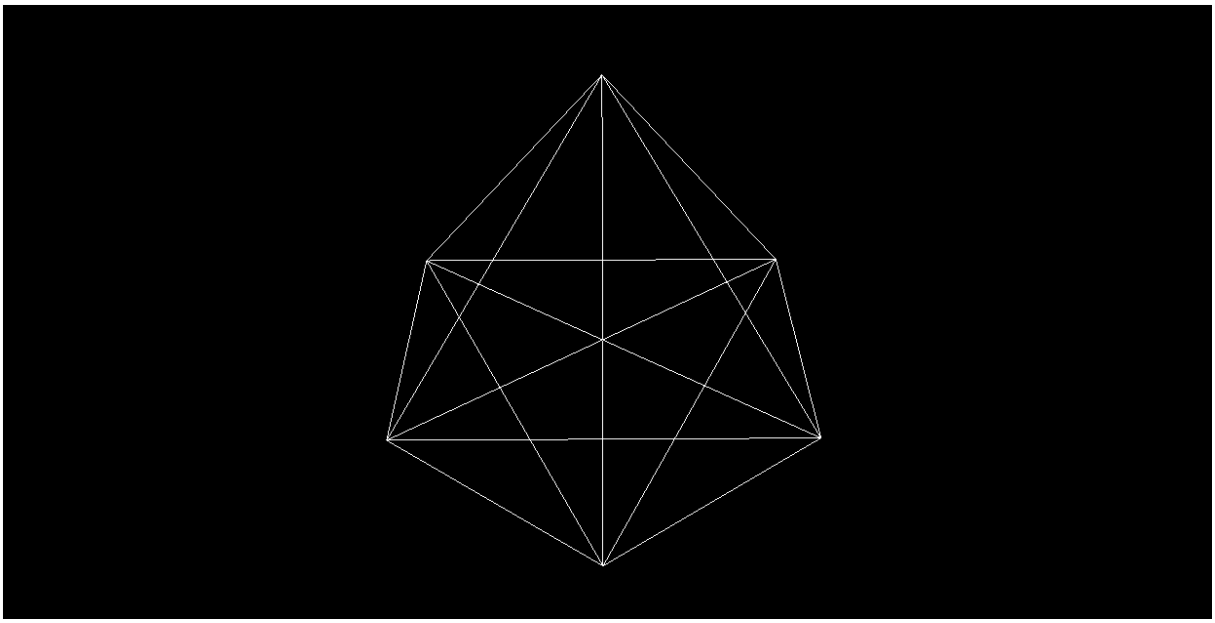
Simplex



```
1  // Constructor
2  App::Simplex::Simplex() : Polytope() {
3
4    // Precalculate values
5    const float root3 = sqrt(3);
6    const float root6 = sqrt(6);
7    const float root10 = sqrt(10);
8
9    // Generate vertices
10   vertices = new float[5 * 4];
11   unsigned int vertexCount = 0;
12
13   vertices[vertexCount++] = 1.f / root10;
14   vertices[vertexCount++] = 1.f / root6;
15   vertices[vertexCount++] = 1.f / root3;
16   vertices[vertexCount++] = -1.f;
17
18   vertices[vertexCount++] = 1.f / root10;
19   vertices[vertexCount++] = 1.f / root6;
20   vertices[vertexCount++] = 1.f / root3;
21   vertices[vertexCount++] = 1.f;
22
23   vertices[vertexCount++] = 1.f / root10;
24   vertices[vertexCount++] = 1.f / root6;
25   vertices[vertexCount++] = (-2.f) / root3;
26   vertices[vertexCount++] = 0.f;
27
28   vertices[vertexCount++] = 1.f / root10;
29   vertices[vertexCount++] = -sqrt(3.f / 2.f);
30   vertices[vertexCount++] = 0.f;
31   vertices[vertexCount++] = 0.f;
32
33   vertices[vertexCount++] = (-2.f) * sqrt(2.f / 5.f);
34   vertices[vertexCount++] = 0.f;
35   vertices[vertexCount++] = 0.f;
36   vertices[vertexCount++] = 0.f;
```

```
37
38      // Generate indices
39      indices = new unsigned int[10 * 2];
40      unsigned int indexCount = 0;
41
42      indices[indexCount++] = 0;
43      indices[indexCount++] = 1;
44      indices[indexCount++] = 0;
45      indices[indexCount++] = 2;
46      indices[indexCount++] = 0;
47      indices[indexCount++] = 3;
48      indices[indexCount++] = 0;
49      indices[indexCount++] = 4;
50
51      indices[indexCount++] = 1;
52      indices[indexCount++] = 2;
53      indices[indexCount++] = 1;
54      indices[indexCount++] = 3;
55      indices[indexCount++] = 1;
56      indices[indexCount++] = 4;
57
58      indices[indexCount++] = 2;
59      indices[indexCount++] = 3;
60      indices[indexCount++] = 2;
61      indices[indexCount++] = 4;
62
63      indices[indexCount++] = 3;
64      indices[indexCount++] = 4;
65
66      // Push generated vertices to these buffers
67      updateVertices(GL_LINES, vertexCount, indexCount);
68  }
```

Hyperoctahedron



```
// Constructor
App::Hyperoctahedron::Hyperoctahedron() {

  // Generate vertices
  vertices = new float[8 * 4];
  unsigned int vertexCount = 0;

  // Create a tetracross
  for (unsigned int j = 0; j < 4; ++j) {
    for (int val = -1; val < 2; val += 2) {
      for (unsigned int i = 0; i < 4; ++i) {
        vertices[vertexCount++] = i == j ? (float)val : 0.f;
      }
    }
  }

  // Generate indices
  indices = new unsigned int[24 * 2];
  unsigned int indexCount = 0;

  // Connect tetracross vertices (ignoring opposite pairs)
  for (unsigned int j = 0; j < 8; ++j) {
    for (unsigned int i = j + 1 + ((j + 1) % 2); i < 8; ++i) {
      indices[indexCount++] = j;
      indices[indexCount++] = i;
    }
  }

  // Push generated vertices to these buffers
  updateVertices(GL_LINES, vertexCount, indexCount);
}
```

Appendix B Vertex shader source code (GLSL)

```
#version 330 core
layout (location = 0) in vec4 aPos;
uniform float transform[25];
uniform float view4D[25];
uniform float distance;
uniform mat4 view3D;
uniform mat4 projTo2D;

// Used to multiply a 5x5 matrix by a 5D vector
void mult(out float ret[5], in float a[25], in float b[5]) {
  float result[5] = float[5](0, 0, 0, 0, 0);
  for (int j = 0; j < 5; j++) {
    for (int i = 0; i < 5; i++) {
        result[j] += a[j + (i * 5)] * b[i];
    }
  }
  ret = float[5](0, 0, 0, 0, 0);
  for (int i = 0; i < 5; i++) {
      ret[i] = result[i];
  }
}

// Entry point
void main() {

  // Generate 5D homogenous coordinate
  float pos[5] = float[5](aPos.x, aPos.y, aPos.z, aPos.w, 1);

  // Transform vertex
  mult(pos, transform, pos);

  // View vertex from the 4D camera
  mult(pos, view4D, pos);

  // Project from 4D to 3D
  vec4 p = vec4(pos[0], pos[1], pos[2], pos[3]) / pos[4];
  mat3x4 to3D = mat3x4(
      vec4(1 / (distance - p.w), 0.f, 0.f, 0.f),
      vec4(0.f, 1 / (distance - p.w), 0.f, 0.f),
      vec4(0.f, 0.f, 1 / (distance - p.w), 0.f));
  vec3 p3D = p * to3D;

  // View with the 3D camera and project to 2D
  gl_Position = (projTo2D * view3D * vec4(p3D, 1));
}
```