Visualizing the Fourth Dimension with Virtual Reality

David Dumas Brandon Reichman
University of Illinois at Chicago

Mathematical Computing Laboratory

UIC

Summary

We created a virtual reality program that allows the user to view and rotate projections of four-dimensional polyhedra and surfaces in four-dimensional space. We wanted the user to be able to interact with simple objects, such as a cube, in a way that feels natural, making the idea of four-dimensional geometry accessible to a wide audience of virtual reality users.

To make this program, we used the development environment and 3D graphics engine *Unity* [2] and the *Oculus* virtual reality system including the *Rift* headset and real time hand position tracking *Touch* controllers. The program itself was written in the C# programming language.

The program was created in the Fall semester of 2017 as a semester project with faculty advisor David Dumas in the Mathematical Computing Laboratory, and was funded by a grant from the UIC College of Liberal Arts and Sciences Undergraduate Research Initiative (LASURI).

Motivation

One can not normally manipulate four-dimensional shapes, as we are stuck inside of a three-dimensional world. However, there are ways to use three-dimensional space to look at four-dimensional objects. For example, when looking at a shadow of a three-dimensional object, one is looking at a two-dimensional projection. This idea can be extended to the fourth dimension, so the shadow of a four-dimensional object is three-dimensional.

Virtual reality gives us the capability to calculate and draw these three-dimensional projections and to put them inside of a three dimensional virtual environment. This gives the user the ability to view these projections from any orientation and directly interact with the objects.

4D Objects

5-Cell: The 5-cell is the four-dimensional analogue to a tetrahedron. This object consists of 5 three-dimensional tetrahedron *cells*, 10 two-dimensional triangle faces, 10 edges, and 5 vertices.

8-Cell: The 8-cell is the four-dimensional analogue of a cube. It is also called a *hypercube* or a *tesseract*. This object consists of 8 three-dimensional cube cells, 24 two-dimensional square faces, 32 edges, and 16 vertices.

RP^2: This is the *real projective plane*, a non-orientable surface that is closed, meaning is has no edges. In a sense, it is analogous to the Möbius strip, but with no edges. It is impossible to embed this object in three-dimensional space without self-intersections, however it can be embedded in four-dimensional space. Our program contains one such embedding, which is constructed using polynomial equations.

RP^2 Cut Open: The program can also display another RP^2 mesh which been cut open along a Möbius strip, leaving a disk behind. In viewing this object and comparing it to the full **RP**^{2} mesh, it is a fun exercise to convince yourself that the missing part is really a Möbius strip.

Flat Torus: This is a torus embedded in four-dimensional space using the following function: $(s,t) \longmapsto (\cos s, \sin s, \cos t, \sin t)$. This embedding has two evident families of circles on the torus which are perpendicular at every point. This mesh in four-dimensional space is most compatible with the intrinsic geometry of the torus, which is Euclidean. This torus in four-dimensional space is flat, whereas the usual circular torus of revolution in three-dimensional space is curved. This torus in four-dimensions looks the same at every point. It doesn't have an "inner" and "outer" part that look different, like a torus in three-dimensions does.

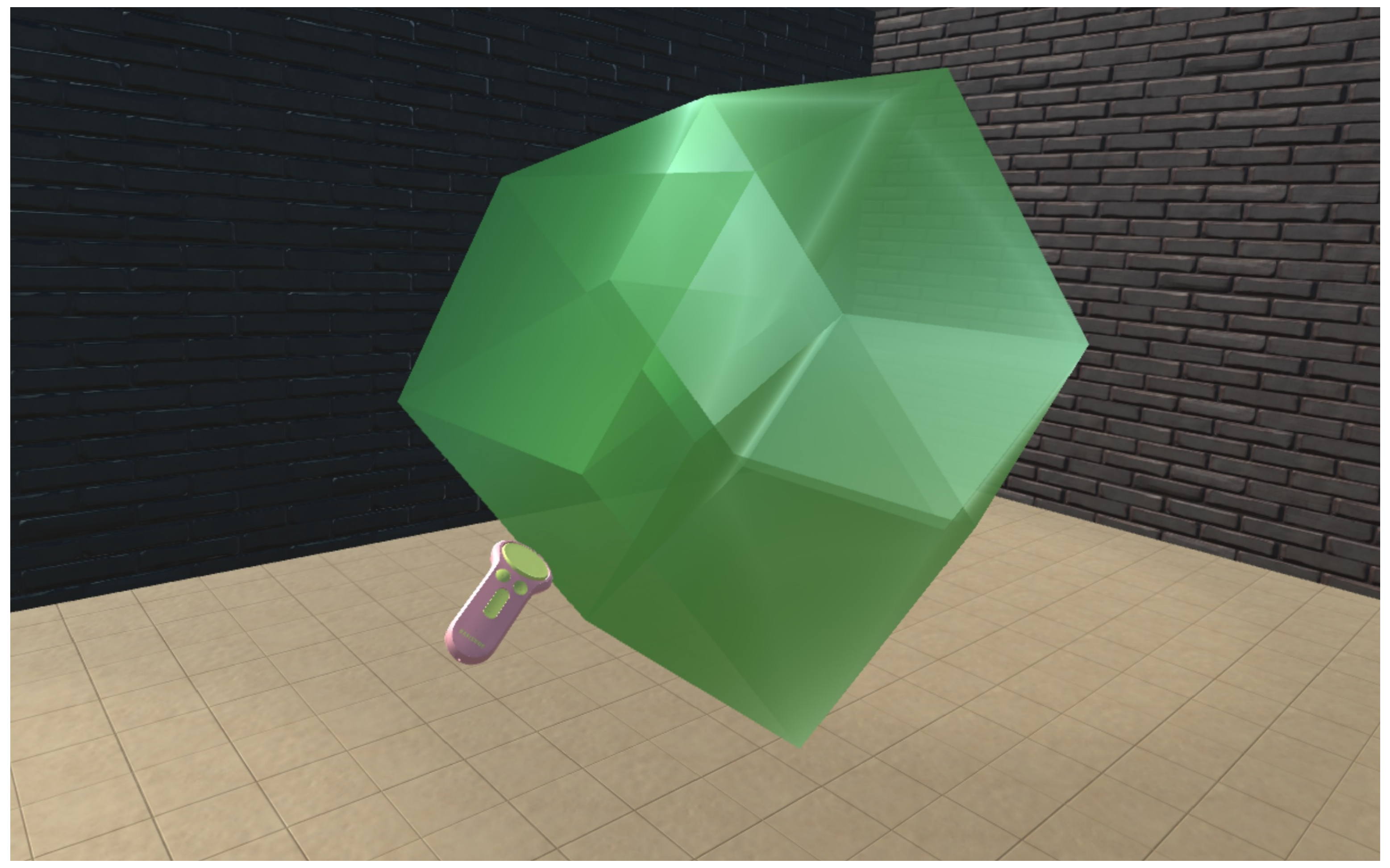
Rotations

Rotations of objects can be represented by matrices. In four dimensions, there is enough space for two simultaneous, independent rotations. For example, one can rotate a four-dimensional object on the XY-plane by α degrees and on the ZW-plane by β degrees, with X, Y, and Z referring to the three directions in three-dimensional space and W being the fourth direction in four-dimensional space. To do this rotation, one would apply the following matrix.

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & \cos\beta & -\sin\beta \\ 0 & 0 & \sin\beta & \cos\beta \end{pmatrix}$$

Our program allows the user to control these two independent rotations by turning their right hand α degrees and turning their left hand β degrees.

It is easy to rotate objects in three dimensions in VR by allowing the user to grab and turn the object with a hand controller. Developing a natural way to control four-dimensional rotations using hand controllers has been one of the main challenges in this project. Our approach is to have the user rotate their hands as if they are turning two "dials" that control the two independent rotation planes.



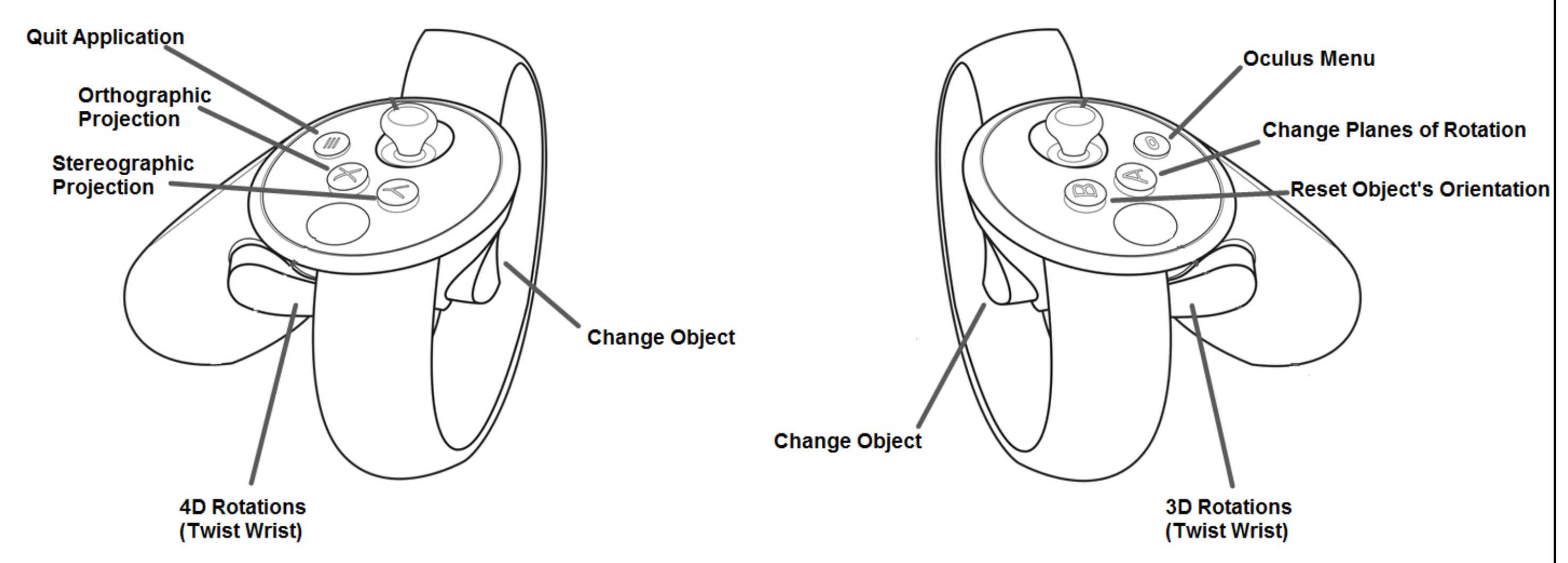
Shown above is an *orthographic* projection of a hypercube which has had various four-dimensional rotations applied to it. Our program allows for *composite* rotations, meaning the user can preform successive rotations in different combinations of planes.

4D Mesh Representations

Selecting data structures and file formats for representing four-dimensional objects was another important aspect of the project. To make the program more versatile, we wanted it to accept arbitrary objects stored in a data file. For three-dimensional objects, there already exist many file formats that represent these objects as triangular meshes, such as STL, OBJ, 3DS, and BLEND files. Since we are only considering two-dimensional triangular meshes in four-dimensional space, compared to these existing formats we only needed to add an extra coordinate to each vertex of each triangle. We learned that the OBJ mesh file format (Wavefront Object format) allows for a fourth coordinate to be added, and that these extra data are ignored by most programs when rendering the object. OBJ files are also text-based, so it is relatively easy to parse through for the information we needed.

Our program interprets these 4D OBJ files using a custom parser to extract the extra vertex coordinate data. It then stores the fourth coordinate in an unused part of the the object's texture data within the Unity 3D engine. In this way, the fourth coordinate is automatically passed to a custom *shader* we developed. The shader unpacks the extra coordinate from the texure variables, applies the necessary four-dimensional rotations, and projects the object back to 3D space. This allows the use of 4D meshes with no changes to the Unity graphics engine.

Controls


Quit Application
Orthographic Projection
Stereographic Projection
Change Object
4D Rotations (Twist Wrist)
Oculus Menu
Change Planes of Rotation
Reset Object's Orientation
Change Object
3D Rotations (Twist Wrist)

Oculus Touch controller diagrams adapted from [1].

Challenges

Creating a virtual reality program that allows one to interact with four-dimensional shapes presented a number of challenges.

We had to customize Unity's existing shader to allow for four-dimensional isometries to be applied to our objects.

Since we did not have any four-dimensional meshes to start with, we also had to create our own OBJ files.

Unity's engine isn't expecting four-dimensional objects to be rendered within it, and some rendering artifacts remain as a result of our shader-based approach. The most significant of these regards lighting: At present, the shading and highlights on the surfaces do not move realistically as the object is rotated.

Future Objectives

While we made significant progress during the semester, there are still aspects of the program we would like to continue developing and improving on. These include:

- Creating more four-dimensional OBJ files to be viewed in the program
- Creating a user-friendly way to import new objects into the program
- Improving the user interface to allow for more natural grabbing and turning to specify 3D rotations
- Adding support for flexible objects that the user can interact with

Some problems we did not have time to solve, such as the lighting issue. In future development, we hope to be able to recalculate the vectors normal to the object faces within the shader. This would result in more realistic lighting and shading.

References

[1] *Oculus (Developer documentation)*. `https://developer.oculus.com/`. Accessed December 1, 2017.

[2] *Unity (version 2017.1.1f1)*. `https://unity3d.com`. Accessed December 1, 2017.