Spring 2022

Fourth-Dimensional Education in Virtual Reality

Jesse P. Hamlin-Navias
Bard College, jh1024@bard.edu

Fourth-Dimensional Education in Virtual Reality

Senior Project Submitted to
The Division of Science, Math, and Computing
of Bard College

by

Jesse Panther Hamlin-Navias

Annandale-on-Hudson, New York
May 2022

Abstract

This project was driven by an interest in mathematics, visualization, and the budding field of virtual reality. The project aimed to create virtual reality software to allow users to interact and play with three-dimensional representations of four-dimensional objects. The chosen representation was a perspective projection. Much like three-dimensional shapes cast two-dimensional shadows, four-dimensional shapes cast three-dimensional shadows. Users of the software developed in this project could interact and experiment with these three-dimensional shadows using hand controlled inputs. The hypothesis put forward is that virtual reality is currently the best medium to teach three-dimensional and four-dimensional geometry.

for the Bois

Acknowledgements

I'd like to thank Keith O'Hara for his personableness and willingness to laugh through my years at college. Bubble Sort is the best sort. Thanks to my team of 'study buddies' and testers: Benicio, Marty, Alex, Aja, and Ro. Thank you Emmanuel for cheering me on. Thank you to my D&D party for keeping me sane (Vulburg is technically alive). Thanks to Justin P Barnett and his fantastic online Open XR tutorials, as well as his amazing Discord community that solved a number of my technical issues. Jacob, I'm unsure how two human beings snap into place so nicely together, thank you for your friendship since childhood, and for being interested in every outlandish mathematical fantasy I've ever had. Thank you BrierMae, you bloomed later in my life than either of us expected, and I look forward to the future. And most especially, thank you Katri. Before I met you, if I had to wish for a partner, I wouldn't have wished for you because I didn't realize someone as wonderful as you was an option.

Table of Contents

Introduction

A few things separate mathematics and works of fiction, and the largest thing that separates the two is presentation. Exploring an alternative world brings joy through learning, not only about the imagined, but the real. In a fantastical setting we begin to understand our own world. Mathematical spaces with fundamentally different logic generate fascinating stories. Flatland by Edwin Abbott is the most classic example that merges the two, employing geometry and dimensionality for social satire. However, mathematics has always had a more difficult onramp to explaining and sharing these fictions with the general public. The mental task of visualizing these logical fantasies is not easy. This difficulty of conceptualization has been lessened by new technologies, which have brought new mediums and therefore new presentations. Recent works of mathematical fiction have appeared in the form of video games. The game Hyperbolica takes place three-dimensionally in hyperbolic space and for a short stint in spherical space, both totally non-euclidean. The game Manifold Garden, as its name suggests, arises in a manifold, and to be specific: a series of fourth-dimensional toruses. Even the very popular game Portal plays in the realm of mathematics. It is very much a game of physics, but it is also a game about giving the player the ability to locally break euclidean's geometry, and its main dilemma lies in the player being able to conceptualize this new power. These games challenge players not just to reimagine what's possible, like in magical worlds, but to reimagine how to do even the most basic of things. Players rebuild their mental formations about how things work from the ground up.

I have been interested in the visualization and conceptualization of higher dimensional spaces since childhood. Growing up I watched videos of the strange and alien pulsations of

higher dimensional shapes. This and my comprehensive introduction to M.C. Escher at a young age left me fascinated by the spatially impossible. I grappled with the art of visualization, and the question naturally arose: can you truly know a fourth-dimensional cube? Is the human capable of conceiving fourth-dimensional spaces and shapes so innately and internally that they could interact and manipulate those shapes as easily as they do with three-dimensional shapes? We are certainly familiar with the dimension directly below us, two-dimensional space, as we can imagine an infinitely flat plane with ease, so why not the dimension right above us? Are we psychologically barred from ever being able to fully understand, only being able to conceive bits and pieces at time, or are we able to put all those pieces together into a whole?

I am not a psychologist, and my interest and skill does not extend to spacial phycological research. Aligned with this fact, my hypothesis is not whether or not we are capable of complete fourth-dimensional conception. Rather, my hypothesis is that the way to answer that question is with Virtual Reality [VR]. More specifically, my hypothesis is that certain mathematical lessons can be taught best using VR. VR is a new medium, it is totally unique from mediums that came before it. It is comparable to video games, movies, and theater. But it must be respected as its own medium because of the remarkable differences between it and its predecessors. In that light, it is worthwhile to ask the question: what is VR best used for? Not just what is made more fun and exciting, but what new things are made easier and possible.

Lessons in geometry, particularly lessons in three-dimensional and higher seem ripe for improvement through VR. The reason for this is the required levels of stacking abstractions to teach geometry. Whenever we teach something, we must abstract it into a concept, and then abstract it into a medium of communication. Any teacher will know the limitations of verbal

communication. Visuals, especially in geometry, can often be more effective. However, the visual medium of choice in classrooms, for very practical reasons, is the black/white-board. The problem in teaching three-dimensional geometry through that medium is that the board is two dimensional. If one wishes to communicate depth on a board, one must come up with and then describe an abstract representation of depth. This extra layer of abstraction, on top of the two already present (concept and medium), muddles things. One should not underestimate the confusing power of stacking abstractions. The field of philosophy spends much of its energy in large part managing and evaluating which abstractions are best fit to describe reality/experience, because of how useful and difficult it is to stack abstractions. Any attempt to draw on a board a fourth-dimensional cube just using vertices and edges will immediately reveal the unsatisfactory nature of the medium. In contrast, the medium of VR is naturally three-dimensional. No additional abstractions for three-dimensional geometric communication are needed. The sensation of depth bridges the gap. One will note that communicating three-dimensional geometry on a board is not that difficult, and can be managed with practice. The question is begged then; if boards can communicate one dimension higher without much fuss, can VR do the same?

VR has interested me as a new medium for human expression. My interest in VR exceeds far past games, and is dwarfed under the shadow of the true stakes of VR. I'm interested in the possibility and risks of building a world that is coextensive with digital reality. My expectations for VR are, although high, not excessive. I do not believe VR will replace physical reality, that it will not become the new axle that our world spins around. Augmented reality on the other hand does hint, and perhaps threaten, at that sort of future. Because of the connection between the

development of VR and augmented reality, I believe it important that VR be developed with ethics in mind. I know augmented reality technologies are beyond my scope at the time of this project, as they are too early in development for me to be of much use. My time for now is better spent experimenting with VR, and so that is the path I chose.

The marketing of VR headsets positions VR to be limitless. Put them on *anywhere*. Transport yourself to *any* space. Become *fully* immersed. VR wishes to completely emulate reality, and then go further. This escapism is an appeal to the possibilities outside the bounds of the physical world, circulated for the benefit of selling VR. In this imagining of VR, "freedom is highly valued, but rather than creatively engaging with the contingent limits of freedom, [virtual environments] propose we surround ourselves with freedom as a commodity we produce *as if* gods" (Hillis, 1999, p. xxxiii). This fantasy of freedom, of escaping from reality, requires detaching from physicality, of leaving, maybe forever, the physical world behind. This escapism is a falsehood, propelled to sell VR units. VR, like any other medium, has its limitations and weaknesses that cannot be made up for by more usage of the medium. In particular, the software developed in my project points to the limits of VR's self proposed pure freedom. By giving the VR user a new freedom, interacting with the three-dimensional shadows of fourth-dimensional shapes, the user will also encounter insurmountable limitations. We cannot ever inhabit four-dimensional spaces in VR, and so that is a freedom that cannot exist for us. But we can experience parts of it at a time, just not its totality. We can, in a very literal sense, only gain glimpses of this higher mathematical world. We become aware of our limitations through the freedom to see what's impossible.

In this project, I built software for the Quest 2, a VR headset, in which users can interact and play with three-dimensional objects and their two-dimensional shadows, as well as play with three-dimensional shadows of fourth-dimensional objects. The focus of this project was the construction of software. Therefore, the purpose of this paper is to give an in-depth explanation of how all of the software was built, and why it was developed that way. It is my hope that this paper will act as a useful guide for future projects using Unity and building for VR.

Literature Review

What follows is a philosophically driven exploration of the VR field. This research eventually led me to my project's focus. I was particularly interested in the notions being built about VR. How we interact with and conceptualize VR is important because it guides how we develop hardware and software for VR as we move into the future. And as we move into the future, it's of utmost importance that we empower individuals to take control and shape VR to their own needs. The terminology used for this subject has morphed over the years, leaving definitions and verbal standards murky. Word choice is complex, because any consistent use of vocabulary in this proposal would be making a claim about how ‘real’ VR is. For the purposes of this review, the word reality will never be used without a specific descriptor, in an attempt to take the fewest sides. Digital reality will refer to spaces and experiences provided through VR headsets. Virtual is intentionally avoided here, but when it is used, it's used as a label for the technologies that create immersive digital spaces and experiences, but it does not itself refer to those spaces and experiences. Physical reality will refer to experiences found while not using virtual reality technology.

Space and Reality

A tension exists within the literature of the virtual, an ongoing debate on the ‘realness’ of VR. Two questions hang above those who develop theories and methods for the new technology; How much is physical space a consideration in creating digital realities? Is digital reality considered real? The hardware and software matches this shifting paradigm, with the separation between physical and digital being bridged (Saker, 2020) (Alkemade 2017) (Augmented Reality Based Framework 2021) (Fernandes 2016).

The belief that VR is free from the physical and is 'virtual' is a widely held public understanding of VR (Hillis 1999) (Boellstorff 2015), and is mirrored in the literature (Choe, 2019)(Agić, 2020)(Fagernäs, 2021). Asserted is that there is a physical world, and that VR is separate from that reality (Choe, 2019)(Agić 2020)(Fagernäs, 2021). This separation allows development of improving new software, but puts a lower priority on the development of hardware.

The second belief is of VR as a method to represent, improve, or serve the physical world rather than creating a real world within itself (Interaction Design for Multi-User Virtual Reality Systems, 2021)(Augmented Reality Based Framework, 2021)(Alkemade, 2017)(Fernandes, 2016). The applied field of digital reality, often interested in military, industry, and factory production, sees VR as distinctly physical, but virtual rather than 'real' (Alkemade, 2017)(Interaction Design for Multi-User Virtual Reality Systems, 2021)(Augmented Reality Based Framework, 2021). This approach often challenges and improves VR hardware and its design, but runs the risk of creating uninventive software when extending reality instead of reinventing it.

The third and final common view understands VR as a valid reality and inherently physical. Often touted by psychologists, philosophers, and anthropologists, academics of this persuasion argue that VR is just another form reality takes, no less real or authentic than physical reality (Banakou, 2021)(Boellstorff, 2015)(Saker, 2020)(Fagernäs, 2021). This perspective produces wild new software, but can fall short by creating software that isn't particularly applicable to everyday physical reality (Saker, 2020)(Fernandes, 2016).

Immersion and Presence

Echoed through papers is the pursuit of the ethereal ‘presence’, the user's feeling of ‘being there’ (Saker, 2020)(Choe, 2019)(Interaction Design for Multi-User Virtual Reality Systems, 2021)(Agić, 2020)(Fernandes, 2016)(Fagernäs, 2021). Immersion is something different, the quality of technology to mirror physical experience. Higher fidelity, faster frames, these are things that increase immersion (Saker, 2020)(Fernandes, 2016). But the content of the experience itself, the perhaps undefinable quality of realness, these constitute presence (Saker, 2020)(Fagernäs, 2021)(Agić, 2020).

After exceeding a lower bound, increasing fidelity doesn’t improve immersion by much. After VR reaches 6 degrees of freedom (the term for the technology that allows a user to rotate and move their body however they like), there are no more degrees of freedom to be met (Saker, 2020). But reducing fidelity below that lower bound drastically negatively affects immersion. Presence on the other hand, requires work and iteration to achieve (Fernandes, 2016)(Fagernäs, 2021). It is improved through what might seem like minor perceptual obstacles that users encounter. Those trying to create presences rely on UI that feel easy, natural, and intuitive (Augmented Reality Based Framework, 2021)(Fernandes, 2016)(Interaction Design for Multi-User Virtual Reality Systems, 2021).

User Input Methods

A common type of experiment in the field aims to measure the efficiency, accuracy, and user reviews of different VR user input methods (Choe, 2019)(Alkemade, 2017)(Interaction Design for Multi-User Virtual Reality Systems, 2021). This approach assumes a self contained, limited approach to UI, in which the goal of experimentation is to find the best generalized input

method. Choe (2019) measured two main methods of user input while using VR headsets. Each method used the gaze of the user as the cursor, but one required the user to click a button to confirm where they were looking as their input, while the other selected what the user was looking at as their input after a few seconds of looking at one spot. The study then compared the methods and found that the button method was faster but less precise. In Alkemade (2017), the time required to complete tasks in a mock computer aided design software was measured, with a combination of two different input methods and two output methods. The first combination was traditional mouse and computer screen, the second was hand tracking technology and a computer screen, the third was hand tracking with VR. The paper found that the hand tracking and computer screen combination performed much worse than the other two, while traditional methods versus hand tracking plus VR performed essentially the same. Approaches to the development of VR UI like these allows comparisons of different input methods. Attempts to determine which input technology is best are methodologies that believe there should or will be a limited few technologies by which users will interact with VR. However, this approach is not the full picture because it assumes efficiency and accuracy are the metrics to measure by, when there are other possible metrics possible, such as expressivity, creativity, or comfortability (Fernandes, 2016)(Agić 2020, Fagernäs, 2021).

UI in VR is obsessed with visual stimuli, and physical movement (Alkemade, 2017)(Choe, 2019)(Agić 2020)(Fernandes, 2016). This focus makes sense for the medium, but leaves other elements out of consideration, like smell, or touch, or even sound (Hillis, 1999). However, participants of studies will bring up these elements themselves, which leads to research talking about the elements, without explicitly testing them (Alkemade, 2017)(Fernandes, 2016).

Conclusion

This research gives insight into the conceptual stakes of VR. What is reality, and what limitations do we bring by defining that reality? The level of immersion and presence of VR is determined in part by the UI of VR. A higher quality of immersion and presence shapes how we consider the reality of VR. By providing VR experiences that challenge a user's fundamental assumptions about reality, we can give those users a place to start questioning the inner workings of how they perceive reality, and therefore can engage and place VR in their conception of reality where it is most useful to them.

Technical Background

Quest VR:

VR can mean many things. Often, when talking about VR, people mean a HMD, or head mounted display. VR HMDs have three branches of their hardware. The computation/rendering, the display, and the tracking. Computation includes standard computer processors as well as specialized built in algorithms to convert tracking data into spatial locations, and rendering (turning computer memory into data ready for display). Computation can either happen out of the headset requiring the headset be tethered to the computer by a data cable, or untethered from a computer, having the processing in the headset which lowers the maximum computational power of the headset but increases the movement freedom of the user. The Oculus Quest 2 has a Adreno 650 graphics card built in, which is an untethered VR solution.

The display includes any hardware element of VR that takes rendered data and turns it into sensory experiences for humans. This does not just include visual screens, but in the case of the Quest 2 also sound. The visual displays of HMDs are highly specialized, often with two screens, one for each eye. However the Quest 2 has one larger screen that is separated visually by the two lenses within the head mount. The sound system of the Quest 2 includes two small speakers that rest half an inch from each ear.

Tracking includes any hardware that gathers data about what a user is doing physically. The fidelity of tracking is measured in degrees of freedom. A degree of freedom is a motion of the user that the system can keep track of. This includes moving forward and back, left and right,

up and down, and the rotations: pitch, yaw, and roll. The Quest 2 has six degrees of freedom. The Quest 2 has three types of sensors:

> "Linear acceleration and rotational velocity data from [inertial measurement units] (IMUs) in the headset and controllers are integrated to track the orientation and position of [the headset and controllers] with low latency. Image data from cameras in the headset helps generate a 3D map of the room, pinpointing landmarks like the corners of furniture or the patterns on your floor. These landmarks are observed repeatedly, which enables [Quest's hardware and software] to compensate for drift (a common challenge with IMUs, where even tiny measurement discrepancies build up over time, resulting in inaccurate location tracking). Infrared light emitting diodes (LEDs) in the controllers are detected by the headset cameras, letting the system bound the controller position drift caused by integrating multiple IMUs" (Hesh, 2019).

Basics of linear algebra:

My project requires me to manipulate points and data in two-dimensions, three dimensions, and four dimensions. One of the most common ways of creating and manipulating points in spaces is using tools of linear algebra. A reason I chose linear algebra is because of how easy it is to convert algorithms for manipulating points in one space to an algorithm manipulating points in a higher dimensional space. Of interest for this project in linear algebra are vectors and matrices. Vectors can be used to denote specific points in a coordinate space, and can be represented as an array of numbers. A coordinate space has an origin, and a vector for

each dimension of the space. These basis vectors can be thought of as the axis of space. In three-dimensional space these would be the x, y and z axis.

For example, a representation of a point in three-dimension space is as such:

$$\begin{bmatrix} 3 & 0 & -2 \end{bmatrix}$$

Where the first number is the x coordinate, the second is the y coordinate, and the final is the z coordinate of the point. If we want to represent a fourth dimensional point, we simply add an extra column to our vector:

$$\begin{bmatrix} 4 & 3 & 0 & -2 \end{bmatrix}$$

This is now a fourth dimension point, the first number being the newly added w coordinate. Vectors can have values added, subtracted, multiplied, and divided from them.

$$\begin{bmatrix} 4 & 3 & 0 & -2 \end{bmatrix} * \text{-}0.5 = \begin{bmatrix} 4 * -0.5 & 3 * -0.5 & 0 * -0.5 & -2 * -0.5 \end{bmatrix} = \begin{bmatrix} -2 & -1.5 & 0 & 1 \end{bmatrix}$$

Vectors can be added and subtracted from each other. To do so, just combine each pair of coordinates appropriately.

$$\begin{bmatrix} 3 & 0 & -2 \end{bmatrix} + \begin{bmatrix} -1 & 4 & 0 \end{bmatrix} = \begin{bmatrix} 3-1 & 0+4 & -2+0 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \end{bmatrix}$$

A vector in coordinate space can be thought of as a point, but it can also be visualized as an arrow pointing from the origin of that space to the given point. That arrow has direction and magnitude (or length). When we add two vectors together, vectors *a* and *b*, you can think of starting at the origin, placing *a*'s arrow there. Then, we follow *a* to the point it points to, which we will call *pa*. We place vector *b* on *pa*, and follow it to its point, *pb*. Adding and subtracting

vectors can represent translating a point through space. This allows us to go between using a vector to represent a point, and using a vector to represent a translation transformation.

Matrices are the second half of this toolkit, and are incredibly useful in representing transformations. These transformations come in the form of matrix multiplication. By combining matrices through multiplication, we can represent a point as a vector and a transformation as a matrix. The result is the point after the transformation. I used matrices to perform rotational transformations. Matrices are rectangular arrays of numbers, and are represented thusly:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

To transform vectors via matrices, we first need to know how to combine matrices via multiplication. And to know how to multiply matrices, we must learn how to find dot products. Matrix multiplication can be thought of using a series of dot products. A dot product takes two equally sized lists of numbers, and combines them into a single number. The dot product of two three dimensional vectors is found like so:

$$\begin{bmatrix} y0 & y1 & y2 \end{bmatrix} * \begin{bmatrix} x0 \\ x1 \\ x2 \end{bmatrix} = \begin{bmatrix} x0 * y0 + x1 * y1 + x2 * y2 \end{bmatrix}$$

A dot product then, is pairing numbers from the list of numbers, multiplying each of those pairs, and then adding all those multiplied numbers together. Note that in matrix multiplication, vectors can be treated like a matrix with either only one row or one column. So this dot-product example is actually an example of multiplying a 1x3 matrix by a 3x1 matrix, and the product is a 1x1

matrix. Notice also that the first matrix uses its row to find a dot product, and the second matrix uses its column. This is actually a rule of matrix multiplication, and follows into larger matrices:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} * \begin{bmatrix} 6 & 9 \\ 7 & 10 \\ 8 & 11 \end{bmatrix} = \begin{bmatrix} 0*6+1*7+2*8 & 0*9+1*10+2*11 \\ 3*6+4*7+5*8 & 3*9+4*10+5*11 \end{bmatrix} = \begin{bmatrix} 23 & 32 \\ 86 & 122 \end{bmatrix}$$

We can see that the row 0 column 0 of the product matrix is made by finding the dot product of row 0 of the first matrix with column 0 of the second matrix. Row 0 column 1 of the product matrix is made by finding the dot product of row 0 of the first matrix with column 1 of the second matrix. This pattern continues for any size matrix multiplication. Note that because of the fact that the first matrix uses its rows when multiplying, and the second matrix uses its columns, matrix multiplication is not commutative. That is to say, the order of multiplication with matrices matters. If we were to swap the order of multiplication in the example directly above, we would get a 3x3 product matrix, rather than a 2x2. For this reason, to multiply matrices together the first matrix must have as many columns as the second matrix has rows. The produced matrix will have the same number of columns as the first matrix, and as many rows as the second matrix.

Multiplying a vector by a matrix represents taking a point, the vector, and putting it through some transformation, the matrix. In this way matrices come to represent transformation. My project uses a limited number of transformations, the first of which are rotations, represented by rotational matrices.

Rotation matrices

For every possible rotation in a space, there is a matrix that represents it. The 3 three-dimensional rotational matrices look as follows:

$$\mathrm{R}_{yz}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix} \qquad R_{xz}(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

$$R_{xy}(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Where θ is the angle of transformation desired. A rotation is moving a point on a plane such that it maintains a constant distance from some given center point. The three basic planes of three-dimensional space are the xy plane, the yz plane, and the xz plane. Multiplying a vector by one of these matrices rotates the point on that plane, maintaining constant distance from the origin. If we want to rotate a point, *p*, around a different center point (other than the origin) a combination of translations and rotation is required. Let's call this new center point *c*. First, subtract *c* from *p*. This is equivalent to translating both *p* and *c* such that *c* lies on the origin. Then multiply *p* by the rotation matrix, rotating it around the origin. Finally, add *c* back to *p*. We can now rotate any point around any center point.

However, there are an infinite number of planes one could construct intersecting the origin, which means there are also an infinite number of possible rotations around the origin. We can construct any of these rotations by multiplying the three rotation matrices together, and then inputting the three θ values using a independent value for each. This plus our translation trick

above means we can rotate any point in three-dimensional space around any other point by any rotation we wish.

Projection matrices

The other type of matrix transformation I used was projection matrix. A projection matrix takes a point in one space and transforms it into a point in a different space. I projected the vertices of my three-dimensional shapes onto a two-dimensional plane, the ground, to create 'shadows'. I created two projection types, isometric and perspective. Perspective projection is the main focus of my project. My perspective projection uses a point light source and a plane, in this case the ground. Any point in three-dimensional space below the light and above the plane gets mapped onto the plane, based on the light and the points location. This resembles closely how shadows act. Like rotations matrices, projection matrices always assume the center of the projection, in this case the light source, is at the origin. To address this, the same sort of translation, matrix transformation, and then reverse translation algorithm is needed. I won't show the projection matrix here because explaining it is a bit complex and outside the scope of this project. However there is a very nice mathematical simplification of the translation-projection-reverse translation which I will show here. Given a light source at $[x_l, y_l, z_l]$, a point at $[x, y, z]$, and assuming the plane to project to is the basic xz plane that intersects the origin, the vector $[x_p, y_p, z_p]$ is the point $[x, y, z]$ after a perspective projection:

$$x_p = x_1 - ((x - x_1) / ((y - y_1)/y_1))$$

$$y_p = 0$$

$$z_p = z_1 - ((z - z_1) / ((y - y_1)/y_1))$$

My other projection, isometric, is incredibly simple. Given a vector [x, y, z], the isometric projection of that vector onto the basic xz plane that intersects the origin is the vector [x, 0, z]. This is equivalent to my above perspective projection if the light point had a y value of infinity.

Mathematics of higher dimensional spaces:

Dimensions are a quality of mathematical spaces. A dimension is a direction an object in space could be translated without changing its coordinates in any other dimension. Better put, the dimension of a space is the minimum number of coordinates needed to specify a point in that space. In standard Euclidean spaces, this works out to mean the maximum number of co-orthogonal lines one can put in the space. A one-dimensional space is a line, in which objects are points or collections of points on that line. They can be translated along that line. Adding a direction of movement perpendicular to the one-dimensional space creates a two-dimensional infinite plane. Adding another orthogonal direction creates a three-dimensional space, an infinite cube. We can continue adding orthogonal directions above three. This is difficult to picture, but if you let go of visualizing it and stick to the logical descriptions and implications, higher dimensional spaces are surprisingly easy to work with. The math and logic is not the confusing part, it’s the conceptualization and application.

For example, what does a fourth dimensional object look like? That's a hard question to answer visually. But if we instead focus on mathematical definitions, we get a working logical model that skips the need for visualization. A point is the simplest geometrical object we can

work with. A point is zero-dimensional. Two points define, or bound, a line, which is one-dimensional. Three or more lines bound a polygon (a triangle is the simplest case), which is two-dimensional. Four or more polygons bound a polyhedron (a tetrahedron), which is three-dimensional. So what bounds a four-dimensional object? What is the face, or outermost surface? Five or more polyhedrons of course. Again, this is hard to visualize, but the logic was simple, and continues in this way through every higher dimension. We will return to the visualization later in my project.

In three-dimensional space, there are three basic translations. A basic translation moves an object along the vector of one of the space's basic vectors, or axes. In four dimensional space, there are four axes, so there are four basic translations. In three-dimensional space, there are three basic rotations, as we covered earlier. Recall that a rotation needs a plane, and a plane can be defined by two vectors. A basic plane is defined by two axes. In three-dimensional space, there are three possible combinations of two of the three axes. In four-dimensional space, there are six possible combinations of two of the four axes. This means in four-dimensional space, there are six basic rotations around the origin. I won't cover the transformation matrices here, I will cover them later, and they can be found in my code.

The perspective projection of a three-dimensional object into two-dimensional space looks like a shadow, or a two dimensional object because it has width and length, but no height. The perspective projection of a four-dimensional object into three-dimensional space looks like a three-dimensional object.

Unity:

Unity is a game development software, and is one of the most widely used pieces of software of its kind. Unity is responsible for speeding up development times by organizing and maintaining game data hierarchies, providing large C# libraries for game making, compiling and rendering scenes, and building projects to multiple platforms. It also has many tools to build software for different platforms, and has a very high degree of customizability. It has two main modes of development, two-dimensional software and three-dimensional software. My project used the 3D development platform, and so I will only be covering that, although many things that are true of 3D Unity are also true of 2D Unity.

Pieces of software built in Unity are broken up into scenes. Each scene can be filled with objects. These objects can be things that appear visually in the software, like grass, or can be rules that the software is effected by, like a day night cycle, can be a combination of both, like a ball that can be thrown and bounces off of other things, or can be an empty placeholder. These different objects are organized in the hierarchy window. Each object can have children objects, hence the name hierarchy window. These objects can have multiple children, although an object can only have one parent object. This allows users to search for objects via code very quickly, and makes Unity's built in methods easy to use.

Each object has a list of components shown in the inspector window. This is where things like controls over its appearance, as well as its behavior would appear. Each object has a transform component, which is its location, rotation, and scale in the world. These attributes of the transform are stored as three Vector3's, an object type built into unity that represents three-dimensional vectors. The rotations of objects are actually stored as quaternions, but can be

obtained as Euler angles, which can be represented as a three-dimensional vector. Transform component contains an object's parent and its children. A transform might not matter for an object like a day night cycle controller, but for most objects it is quite useful. Another common type of component is a mesh. This is a series of triangles that can be rendered to create shapes in three-dimensional space. A material is another kind of component that determines those triangles' appearance, as in their color, interaction with light sources, and any textures. Rigidbody components apply forces, such as gravity to objects. Colliders allow objects to physically interact, such as bumping into each other, or for determining when objects ‘overlap’. The final most common type of component is a Script. This is a piece of code, which I write in C#, that the object inherits. A script can do almost anything you can think of, including recreating the properties of other types of components. Scripts can affect objects they are not components of, but it is best practice to have scripts only affect their own object, and the object's children. Components can be added to objects, and turned off and on, called disabling and enabling, for testing purposes.

Due to the object hierarchy-component structure of Unity, it is a very object oriented platform. Objects contain code, and as such, it is most often efficient to have objects with properties that do things, rather than systems that create objects and then manipulate them. This can mean that procedural generation, creating data algorithmically, in Unity can be a bit confusing at first, as procedural generation is easier to conceptualize in the structure of functional programming. This posed a challenge to me in my coding. Although three-dimensional object to two-dimensional projection was easy to translate into object oriented programming, four-dimensional to three-dimensional is not as object oriented. This is due to the

fact that Unity cannot create four-dimensional objects, and as such, the representations of four-dimensional objects had to sit somewhere between an object and a function.

Another important part of Unity is the Project window. This contains the assets a developer has created or downloaded. These assets can be used again and again inside the scene. Examples of common assets are materials and scripts. Instead of customizing each object's material individually, it is much faster to create a material in the Project window, and then drag and drop that material onto each object that should look the same. Scripts are also held in the Project window. This way, a developer doesn't have to copy and paste code from one object into another, but instead can write the script once, and then add it to each object that should inherit those properties. The Project window is also where downloaded assets, called packages, are accessible. These assets might not be used over and over again, but instead can add extra functionality to a Unity project, and the Project window is the way a developer can look through the scripts included in packages.

Unity renders the scene currently being worked on in the scene view and the game view windows. The game view window shows the software through the perspective of the camera object in the scene. This is useful for testing to make sure users' experiences will be as intended. The scene view shows the software through any view the developer wishes, like an extra camera that has no bearing on the software. This is useful for creating and positioning objects, dragging and dropping assets, as well as testing from a more birds eye view.

Scripts in Unity can be written in a number of languages, the most common of which is C#. Unity provides a large specialized library for use in scenes called the Unity Engine. All Unity C# scripts derive from the MonoBehavior class. From these we have access to the two

most common specialized methods in Unity. Monobehaviour.Start is called when a script is enabled on an object. Enabling happens upon adding a script to an object. Monobehaviour.Update() is called every frame, making it the source of all in-game events. Note that while editing before playing, Update is called continuously, making testing very easy. When the play button is pressed, all Scripts are reloaded, meaning all Start functions are called. This means that pressing play starts the developers software off fresh each time, even though in editing many things might have been changed in the scene.

Open XR:

Open XR is an open source package for Unity that connects Unity with VR, allowing projects to be run and rendered on headsets. The XR Interaction Toolkit changes the input method of Unity projects to the controls of VR headset, allowing for motion tracking and controller inputs to be used in game. Together, they make developing software for multiple different VR brands at the same time possible. I'm not going to go in depth on my explanation here because of the ever evolving nature of Open XR. It is very likely that any detailed account of how Open XR works I could make, would in the near future be outdated. The basics that will likely remain the same are this: upon downloading Open XR properly and adjusting your Unity's project setting correctly, you can run your project on a number of different VR HMDs with full motion tracking. Open XR comes with a number of helpful additional components to allow developers to quickly create grabbable objects and teleportation among other things. XR Interaction Toolkit allows developers to access headset and controller positions, as well as customize controller inputs. Open XR is still in development, and connecting VR headsets to

computers and to Unity is not easy, even with Open XR. It is however made a possibility, for without Open XR, it would likely have been infeasible for me to attempt this project.

Methods

Design Process:

The first things I needed to develop were three-dimensional objects and their two-dimensional projections. The reason for this is that programming four-dimensional objects is difficult. The code for four-dimensional objects is not substantially more complex than three-dimensional ones, but the testing is so much more difficult. This is because four-dimensional objects cannot be seen, and so cannot be tested in their native mathematical spaces. All that can be rendered are projections or slices of them. Starting with four-dimensional objects and their three-dimensional projections would be like trying to develop three-dimensional objects and their shadows without being allowed to look at and visually test your three-dimensional objects.

The benefit of linear algebra is that to mathematically move from a lower dimension to the next higher one, all that is needed is to add an extra row and/or column to your vectors and matrices. A bit more work is needed when coding the transition to a higher dimension, but the amount of work is still quite small. Copying and pasting made up the majority of my work in the transition. Coding three dimensional shapes and their two dimensional projections in Unity took me a semester of work because of the complexity of managing the shape and its shadow. This might seem like a lot, but recall that all the coding I was doing was really in preparation for a higher dimension. Everything was as modular as I could conceptualize it (I was still learning Unity as I went, and so a higher degree of modularity is still quite possible). The time spent developing the software in a general manner paid off, I was able to code the four-dimensional objects and their three-dimensional shadows in only two days. This method section will follow

the same pattern, explaining the code for three-dimensional to two dimensional projection first, and then expanding to the four-dimensional code.

Matrices:

To implement linear algebra in C#, we must first build a matrix class to handle matrix multiplication and manipulation:

```
public class Matrix
{
    public static Vector3 matmul(Vector3[] r, Vector3 v3)

    public static Vector3[] matmul(Vector3 v3, Vector3[] r)

    public static Vector3[] matmul(Vector3[] a, Vector3[] b)
}
```

We overloaded a matrix multiplication method, `matmul()`, which takes two matrices, multiplies them and returns the product. Matrices are not represented by a special type of object, but instead are arrays of `Vector3`s. The `Vector3` class is included in the Unity library, and represents three-dimensional vectors. They take three floating point numbers to construct, which correspond to x, y, and z. This shortcut is allowable since all matrix multiplication we are going to be working with for now are with these simple vectors, or matrices with columns of length 3. Later we will introduce fourth-dimensional vectors and matrices. The `Matrix` class is able to multiply, in the order shown above, a 3x3 matrix by a three-dimensional vector, a three-dimensional vector by a 3x3 matrix, and a 3x3 matrix by a 3x3 matrix.

Next we implement rotation matrices. All three rotation matrices are combined into one single rotation method:

```
public static Vector3[] rotate3d(float xa, float ya, float za)
```

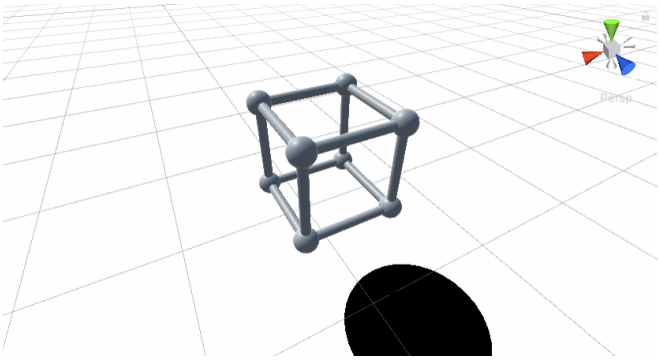
The three inputs are the basic rotation angles on the yz plane, xz plane, and xy plane. This method uses the rotation matrices from the technical background. However, `rotate3d()` does not create three rotation matrices and then multiplies them together. Instead, the matrices are already multiplied together, only needing the angle values to complete the matrix. This buys a small amount of processing speed, increasing efficiency. Finally we need to combine points with our rotation matrix::

```
public static Vector3 rotate_point(Vector3 p, Vector3[] rotate)
{ return matmul(rotate, p); }
```

rotate_point takes a point and a rotation matrix, and returns the rotated point. It's just a prettier version of matmul, which makes later code easier to read.

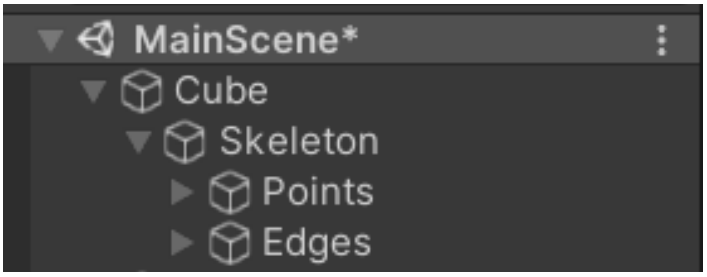
Three-Dimensional Shape Construction and Memory:

With these methods complete, we can now start building and interacting with three-dimensional shapes in a Unity scene. We'll use a cube as the example, since that is the easiest to build due to its very simple coordinates. The completed cube looks like this:



Complex shapes in the software are constructed out of spheres as the vertices and cylinders as the edges. It's important that the components of this cube are organized correctly so that the code

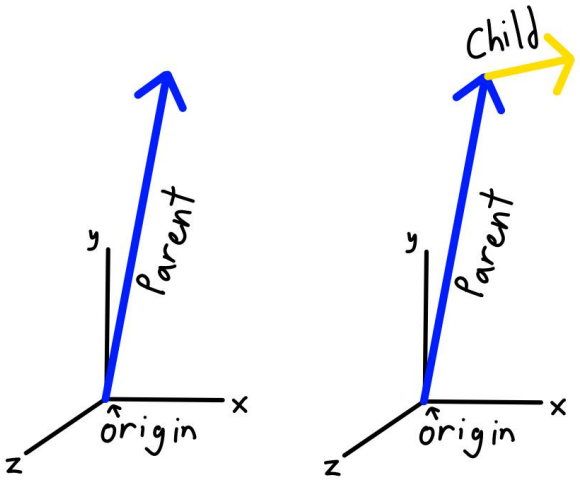
written for it is efficient and understandable, and also so that positioning in the world is easy. I organized my objects as such:



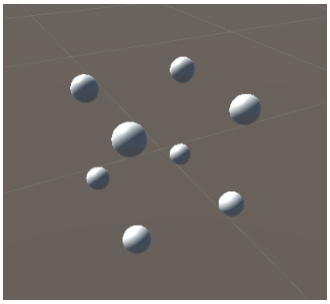
Here we see a portion of the hierarchy window. At the very top is the current scene, which holds all the assets and objects. Below that is the Cube object, which is empty, and has the Skeleton as a child. An empty object is an object with just a transform component and may also have scripts. An object being empty does not mean that object has no children. The Skeleton is also an empty object. The Skeleton has the children objects Points and Edges. Points and Edges are empty objects which hold the spheres and cylinders seen in the above picture. Having the Cube parent be separate from the skeleton parent is important. The Cube object will eventually also have a Projection child object, which will hold the shadow. Organizing like this means that the Cube holds both its three-dimensional parts as well as its two-dimensional parts, but that we are also able to computationally distinguish between objects that are part of the three-dimensional skeleton, and the objects that are part of its shadow.

Each game object has a transform component. The transform holds a `Vector3` called position, which determines its location in the scene. An object with no parent object has an origin based vector (scenes are not objects, and so are not parent objects, even though they appear in the hierarchy window). Remember that vectors are actually arrows with an angle and a magnitude, or length. An origin based vector's arrow starts at the origin and extends outward.

However, the location vector of an object with a parent works differently. An object with a parent is called a child object. A child object's location vector starts at the location of its parent object and extends outwards. It works this way so that when you move a parent object, all its children objects move too:



The position of Cube's transform is not important for now. However, Skeleton's transform position is, because we will treat the center of Skeleton as the center point of all translations and rotations of our three-dimensional shape. Therefore, when building our shape, we will make sure Skeleton's position vector is (0, 0, 0), i.e. at the world's origin. Then, we will create the vertices of the cube as if the cube was centered at the origin:



Now the skeleton object can be moved anywhere and the points will follow correctly. For now we are not going to put cylinders in the Edges object because the edges are going to use a whole different system to rotate.

We'll build a Shape3D script to be attached to the Cube object. The Shape3D script will manage the rotation transformation of the Cube as well as later the projection of the three-dimensional object into two-dimensions. Shape3D manages its memory carefully using load(), Vector3 center, Vector3[] vertices, and Vertex_Store(). load() retrieves data about all relevant objects in the scene, and stores them as global variables in Shape3D. load() only needs to be called once, as the variables it loads are all pointers that point to data that does change. center represents the center point of our shape. The array vertices[] will be used to store a copy of the shape's sphere positions, but as if the shape was centered at the origin. Storing the points at the origin has the benefit of not having to do the full translate, rotate, translate pattern. Instead we can just rotate and then translate by the center vector. The other benefit is that we have a static version of the shape to rely on, that we can derive rotations from without changing, lowering the chance of multiple rotations leading to problems. Vertex_Store() is a public method that updates vertices[] to the current state of our shape's spheres, but again, as if the shape was centered at the origin. Shape3D's Start() function will look as such:

```
load_objects();
center = skeleton.position;
vertices = new Vector3[points.childCount];
Vertex_Store();
```

points is the transform of the Points game object. Our script now always starts with all the gameobjects it needs, and an extra copy of our shape's points to manipulate.

Three-Dimensional Shape Transformations:

The two types of transformations that will be usable in the software will be translations and rotations. We'll implement translations first:

```
public void Translate(Vector3 translation)
{
    center = translation;

    for (int i = 0; i < points.childCount; i++)
    {
        Transform sphere = points.GetChild(i);

        sphere.position = vertices[i] + center;
    }
}
```

A host of terminology must be explained here: `points` is a global transform generated by the load() function. It is the transform of the Points game object, which holds all the spheres. `points.childCount` returns the number of children `points` has, ie. the number of spheres contained in Points. `points.GetChild(i)` returns the transform of the i'th sphere contained in Points. Finally, `.position` is the `Vector3` of the transform that holds an object's position. `Translate()` changes the `center` of our shape, and then iterates through all the spheres and updates their position by adding the stored `Vector3` from `vertices[]` to the new `center`.

The method `Rotate()` in Shape3D we implement will be very similar, adding extra steps to rotate our points by a matrix:

```
public void Rotate(float xRotation, float yRotation, float zRotation)
{
    Vector3[] rotation = Matrix.rotate3d(xRotation, yRotation, zRotation);

    for (int i = 0; i < points.childCount; i++)
    {
        Transform sphere = points.GetChild(i);

        sphere.position = Matrix.rotate_point(vertices[i], rotation) + center;
    }
}
```

`Rotate()` iterates through all the points of an object, and rotates each one, given three rotation angles. Remember that the vectors in `vertices[]` are centered around the origin, so after finding the rotated vertice, we need to translate that point by the vector of `center`.

Notice that both `Translate()` and `Rotate()` use `vertices[]` to transform from, so calling either multiple times in a row will not add transformations together. To do that, one must call these transformations, and then `Vertex_Store()`. After calling `Vertex_Store()`, following transformations will be additive. This fine control of the shape's memory will become useful later when we add user inputs.

Three-Dimensional Edge Updating:

Our shape's vertices can be created, stored, modified, and updated, but it lacks edges. We need to position cylinders such that they connect all of our spheres in a way that represents the shape we want to represent. Shape3D contains a public integer array called `temp_edges[]` that can be edited from the inspector. `temp_edges[]` holds the adjacency list for the points of the Cube. It retains the data of which spheres in Cube should be connected by a cylinder. The array

is structured in pairs of indices, for example `temp_edges[0]` and `temp_edges[1]` are the indices of two points in the Cube that are adjacent. The integer value of `temp_edges[x]` refers to a child object of Points, using `Points.GetChild(temp_edges[x])`. The edge that should be connecting points can be gotten with `Edges.GetChild(i)`, while the indices of the two points are contained in `temp_edges[i * 2]` and `temp_edges[i * 2 + 1]`. Although an array of tuples would have been cleaner, I chose an array of integers because it was easiest to use in the Unity inspector window to build the objects.

The system for rotating the Cube's spheres and rotating the cylinders must be different because of how linear algebra intersects with Unity's transform components. When rotating a vertice with `rotate_point()`, the sphere that visually represents one of the Cube's vertices is moved, but the sphere itself is not actually rotated. If the sphere was colored black on the top and white on bottom, after `rotate()` was called, the sphere would be in a different place, and the Cube would have seemed to rotate, but the sphere itself would still be black on top and white on bottom. This might seem a strange way to represent complex three-dimensional objects, but this is all in preparation for fourth-dimensional shapes. There are easier and faster ways to code three-dimensional shapes, their transformations, and their shadows, but those same easy ways do not exist for higher dimensions.

To orient our edges correctly, we need to scale, rotate, and position our cylinders based on the pair of points they connect. In Matrix, we will implement a method called `update_edge()`:

```
public static void update_edge(Transform point1, Transform point2,
                                                  Transform edge)
{
    Vector3 point1v = point1.position;
```

```
        Vector3 point2v = point2.position;

        edge.localScale = new Vector3(edge.localScale.x,
  (Vector3.Distance(point1v, point2v) / 2f) * (1f / edge.parent.parent.localScale.x),
                                      edge.localScale.z);

        edge.position = point2v;

        edge.LookAt(point1);

        edge.Rotate(90f, 0, 0);

        edge.position = (point1v + point2v) / 2f;

    }
```

Given a pair of spheres and a cylinder, `Matrix.update_edge()` scales, rotates, and positions the cylinder to connect the two spheres. The scaling lengthens or shortens the height of the cylinder without changing its diameter. This scaling is unnecessary for three-dimensional objects, but it will become important for four-dimensional objects. In Unity, a cylinder's center point is halfway between its two circular faces, in the middle of its diameter. One can calculate the two points which sit on each of a cylinder's circular faces in the center of those circles. However, using that to determine a rotation is incredibly complex because of the limited forms rotations can be entered into Unity. The function `transform.LookAt(transform)` is used in `update_edge()` to rotate the cylinder correctly. `LookAt()` rotates an object around its center to face another object. Because it rotates around an object's center, the cylinder must first be moved to one of the sphere's locations. Then `LookAt()` can be called in respect to the other sphere's location. Afterwards, rotating the cylinder ninety degrees in the x axis is required because of what Unity considers a cylinder 'facing' a point. Finally, `update_edge()` moves the cylinder to the midpoint between the two spheres. After the three steps of scaling, rotating, and positioning, the cylinder will connect the two spheres in the scene.

We'll build a public method in Shape3D called `Update_Edges()` that will position all our edges correctly. `Update_Edges()` will iterate through all the cylinders in our Cube, using `temp_edges[]` to find the points to connect to, and `update_edge()` to orient the cylinder. We'll add `Update_Edges()` to the end of `Translate()` and `Rotate()` so that after we transform a shape the edges connect the new points.

Three-Dimensional Vertice Perspective Projection:

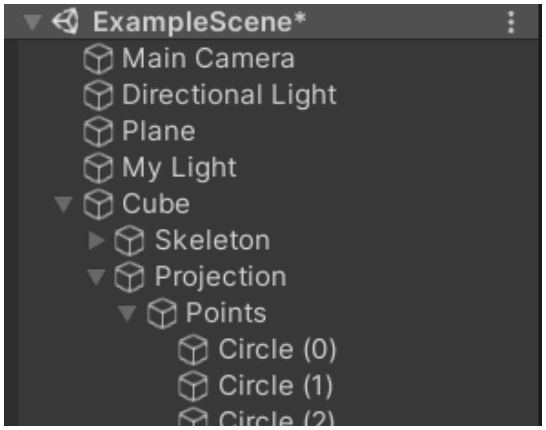
With three-dimensional shapes and their transformations in place, we can now implement perspective projections. We'll use the same approach we used before, calculating the vertices of our desired two-dimensional shape mathematically, and then later implementing edges using `temp_edges[]` and logic derived from our vertice's positions. A perspective projection requires a light source, so in our scene we'll add a small sphere object, name it My Light (Light is a special term in Unity so we want to specify this one is ours.), and move it up above our Cube. Well add a white plane back into our scene to act as the floor. We'll build a rudimentary projection method in Matrix:

```
public static float[] y_3d_2d(Vector3 l, Vector3 p)
{
    float den = (p.y - l.y) / l.y;

    float new_x = l.x - ((p.x - l.x) / den);

    float new_z = l.z - ((p.z - l.z) / den);

    return new float[] {new_x, new_z};
}
```

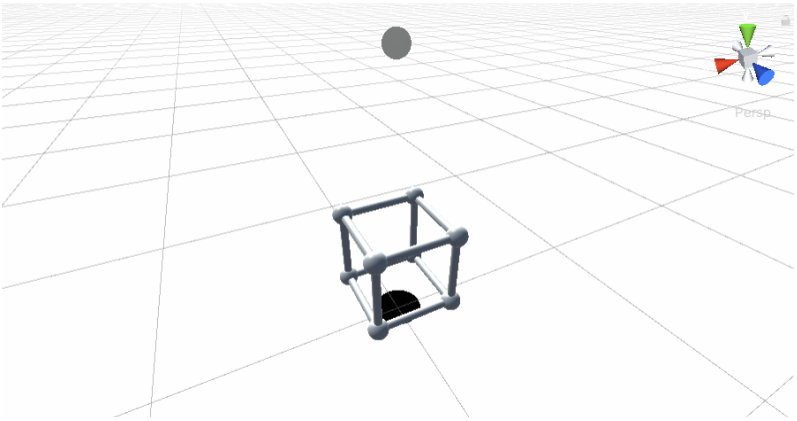
This computation is derived from the textbook Interactive Computer Graphics (Angel, 250). Given the position of our light, `l`, and the position of the point we want to project, `p`, `y_3d_2d()` returns an array that has the x and z coordinates of the point after projection. We can assume the

y value of the point after projection will always be 0, so there's no need to return it in this function. These equations were taken from Interactive Computer Graphics (Angel, 250)

We'll now create our shadow points in our Unity scene. Cube gains a child called Projection. Projection will contain the children Points and Edges, much like Skeleton. Well fill Projection>Points with an equal number of flat circles as Skeleton>Points has. The Hierarchy window will look like this:



And our scene will now look like this:

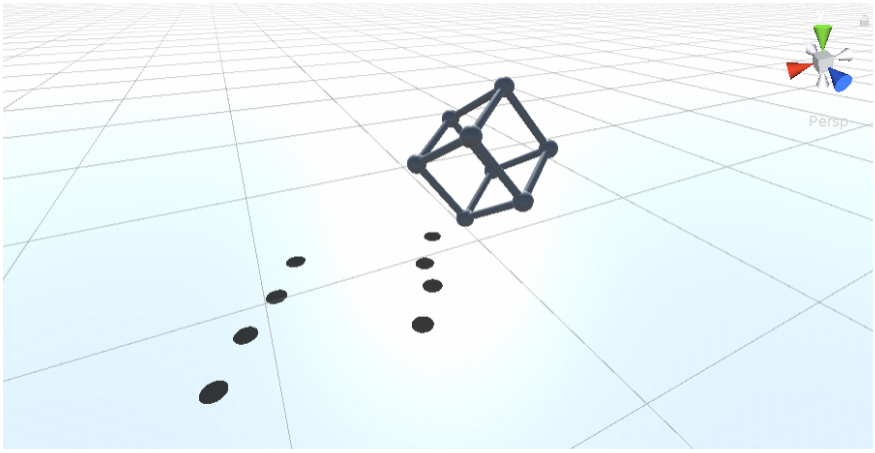


Here we see the light floating above the cube, and the shadow circles stacked ontop of eachother sitting below the cube. Important to note is that My Light, as well as the spheres and cylinders in the skeleton are set to cast and receive no shadows. All objects in Unity automatically cast

shadows onto other objects, and display shadows that would be cast on them. This project's software is building its own system to cast shadows, and as such, we want to disable Unity's pre-built shadow renderer. Now we will write in Shape3D a function to move our shadow vertices based on the position of our three-dimensional vertices and the light's position:

```
void Shadow_Update()
{
    for (int i = 0; i < points.childCount; i++)
    {
        Transform sphere = points.GetChild(i);

        Transform shadow = shadow_points.GetChild(i);

        float[] sv = Matrix.y_3d_2d(my_light, sphere.position);

        shadow.position = new Vector3(sv[0], 0.001f, sv[2]);
    }
}
```

`shadow_points` is a new transform added to the `load()` function. It is the transform component of the Projection>Points object. Notice that the y position of the shadow is set to 0.001 instead of 0. This is because we need the shadow to be just above our plane so that they don't intersect and render improperly. We'll call `Shadow_Update()` at the end of `Translate()` and `Rotate()`. Our scene can now look like this:



We're going to add some scaling so that spheres with a taller y value have shadows that are larger, and spheres with shorter y values have smaller shadows. This will simulate how real shadows act, but without stretching the shadows into ovals, and with a maximum and minimum size. To do this, we need to calculate the lowest sphere of the Cube and the highest sphere of the Cube. In Shape3D, `minmaxp()` will set two global variables, `minp` and `maxp` as the smallest and largest y values among all spheres in Cube. `minmaxp()` will be called at the beginning of `Shadow_Update()`. We'll modify `y_3d_2d()` to take a minimum value and a maximum value. We'll calculate the scaling as such:

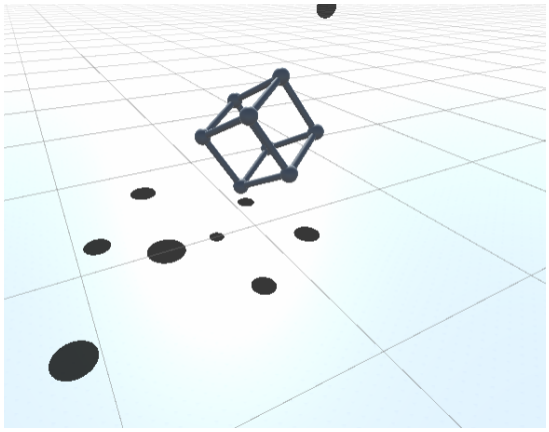
```
float scaling = Mathf.Clamp((p.y - min_y) / (max_y - min_y), 0f, 1f);
```

and we'll return `new float[] {new_x, scaling, new_z}`. In `Shadow_Update()`, we'll change the scale of the shadow circle based on the scaling returned by `y_3d_2d()`. We're going to use this min-max scaling system for extra features later.

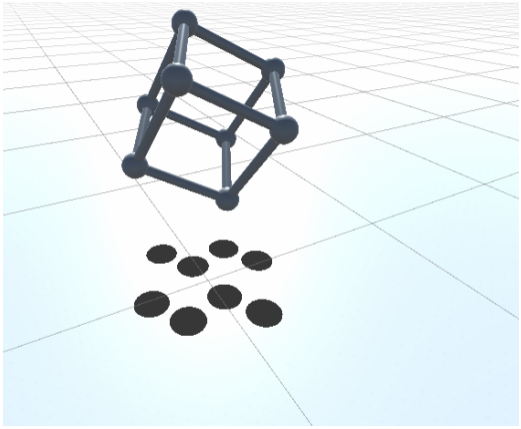
The shadow simulating code leads to a few bugs; if My Light is below one of the vertices, what happens? The shadow of the vertice appears on the plane in a completely wrong place. If My Light is below a sphere, we don't want that sphere to show up on the plane, because the light is supposed to be shining down onto the plane. Additionally, what happens if one of Cube's spheres is moved below the plane? The final edge case is what happens if My Light is moved below the plane? To account for all of these, some additional if-statements are written into `y_3d_2d()`. The end behavior is that if a sphere is above My Light or below the plane, its scaling is set to -1. `Shadow_Update()` handles a scaling of -1 to mean place the shadow below the plane, hiding it from view. If My Light is placed below the plane, y_3d_2d returns `new float[]`

`{p.x, 0.5f, p.z}`. All points then will keep their x and z, and all of them will be scaled halfway. This is isometric projection, in which a point's y value does not matter.

Perspective Projection:

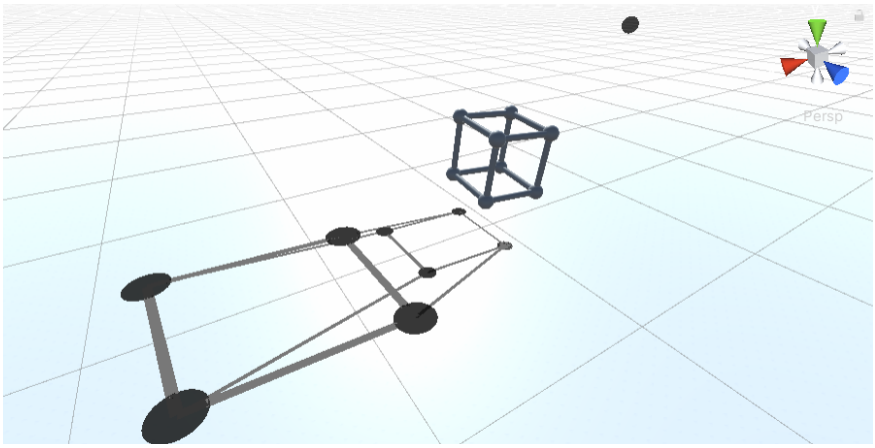


Isometric Projection:



Three-Dimensional Edge Projection:

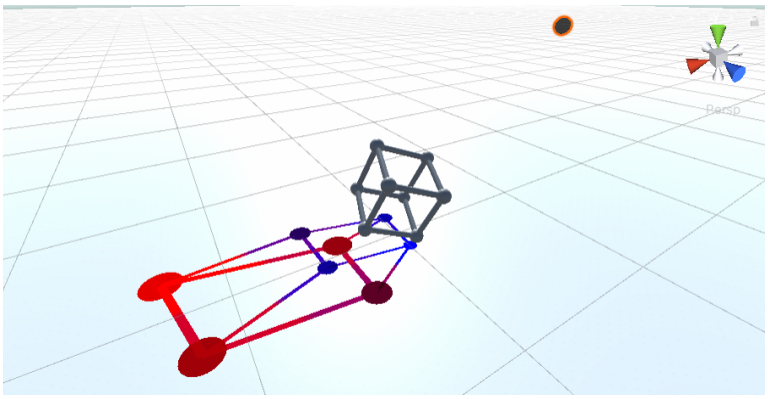
Now we'll add in the shadows of the edges. We'll fill Projection>Edges with plane objects, which are two dimensional rectangles that can be scaled and rotated as needed. In Matrix, the method `update_shadow_edge()` works very similarly to `update_edge()`. The scaling is slightly different, and no additional rotation is needed after using `LookAt()`, but everything else is the same. Given two circles and a plane, `update_shadow_edge()` will scale, rotate, and position the plane such that it connects the two given circles. In Shape3D, `Shadow_Update()` will loop through each plane in Projection>Edges, and using `temp_edges[]`, call `update_shadow_edge()`, orienting each plane.



Because of how `update_shadow_edge()` works, planes connecting to a circle which is below the plane will also be placed below the plane. This means that when a sphere is above My Light, ie. its shadow below the plane, all edges connecting to it will not have their shadows visible to a user.

Coloring the Shadows:

Two-dimensional shadows are not complex to look at. Our brains are hardwired to process, interpret, and abstract them. However, three-dimensional shadows of four dimensional objects are not so easy to look at. To help aid in this, my project colors its shadows to exploit additional functions of sight. The final product of this colorization looks as such:



Shadow vertices are colored based on their corresponding sphere's y value. The larger the original y value, the more red the shadow's point will be. The lower the original y value, the more blue the shadow's point will be. The value of these colors are in respect to the highest and lowest vertices in the three-dimensional shape. If the shape is moved up and down, that won't change any of the shadow's colors. Only rotating the shape will change that.

To implement this, we will need a more robust rendering pipeline than the one that Unity comes pre-built with. A rendering pipeline is what takes the information in a scene and turns it into visuals on a computer screen. My software makes use of Unity's Universal Rendering Pipeline [URP]. The reason for using this more complex pipeline is because it allows us to dynamically generate gradients of color, which we will need to color the shadow's edges. URP can be downloaded as a package in Unity's package manager, and Unity has a built-in tool to convert our old materials to ones URP can render.

In `Shadow_Update()`, well add two extra lines inside the for loop that updates the shadow points:

```
var shadowrend = (shadow.gameObject).GetComponent<Renderer>();

shadowrend.material.SetColor("_BaseColor",
                    new Color(sv[1], 0f, 1f - sv[1], 0.8f));
```

This retrieves the renderer of our shadow point, and then in the next line sets the color of the point's material. The color is set in RGBA, or red green blue alpha values. We use the scaling value from our `y_3d_2d()` method to determine our red and blue value, leaving green at 0. The alpha value of the shadow points is 0.8, which makes the circle slightly transparent. Creating gradients on the shadow edges between the points is more complex. We'll add the following lines into the for loop that updates all the shadow edges:

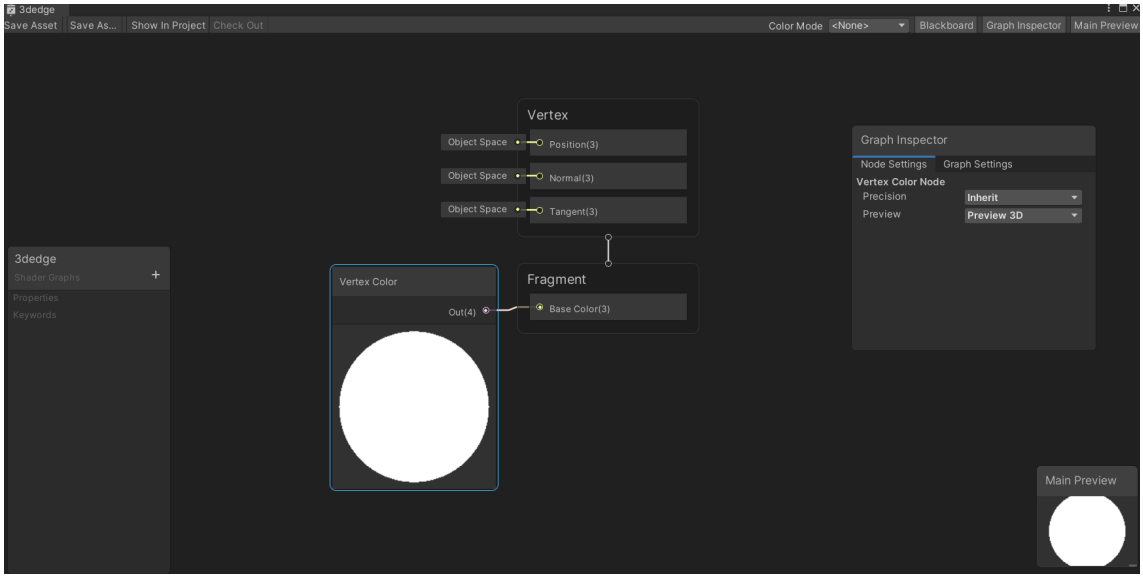
```
Mesh mesh = shadow_edges.GetChild(i).gameObject.
                                    GetComponent<MeshFilter>().mesh;

Vector3[] vertices = mesh.vertices;

Color[] colors = new Color[vertices.Length];

var p1c = shadow_points.GetChild(temp_edges[i * 2]).
                     GetComponent<Renderer>().material.color;

var p2c = shadow_points.GetChild(temp_edges[i * 2 + 1]).
                         GetComponent<Renderer>().material.color;

float minz = vertices[0].z;

float maxz = vertices[vertices.Length - 1].z;

for (int j = 0; j < vertices.Length; j++)
{
   colors[j] = Color.Lerp(p2c, p1c,
                                    (vertices[j].z-minz)/(maxz-minz));
}
mesh.colors = colors;
```

The first line creates a new Mesh object called `mesh`. A mesh is what Unity renders. It is a collection of two-dimensional triangles in three-dimensional space. These triangles can be colored, textured, and made to react to light and shadows in different ways. The mesh created here will become the mesh of the cylinder. The second line creates an array of `mesh`'s vertices, the points of all of the mesh's triangles. For each vertex in the mesh, we're going to calculate a color, so the third line creates an array of colors, with the same number of indices as the mesh has vertices. The fourth and fifth line retrieve the color of the two circles that the edge connects.

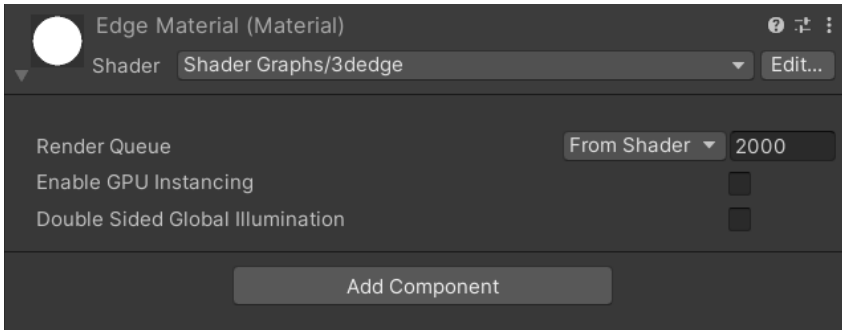
We're now ready to calculate the colors in array Color. For each vertex, we are going to interpolate a color between the second point's color and the first point's color, based on the z value of the vertex. Interpolate means to create new values between two other values, where the new value would fall on the continuous line between the two values. We use the z value to interpolate because `Matrix.update_edge()` scales and rotates the edges such that each rectangle's

local z is the axis which connects one point to another. We calculate a number between 0 and 1, inclusive, based on the vertex's z value and use that to interpolate. `Color.Lerp(Color a, Color b, float t)` returns an a color interpolated from the fraction `t` between color `a` and color `b`. If `t` = 0.5, then the returned color is halfway between `a` and `b`. A `t` value of 0 returns the first color, while a `t` value of 1 returns the second color.

Finally, we set the color array of our mesh equal to the new color array we just filled. However, this is not enough to make the edges appear as gradients. We need to make a new shader. A shader calculates the amount of light, dark, and color to render. Part of URP is its Shader Graph asset. Shader Graph is a node based system to create custom shaders. I will not explain shader graph in depth because I did not have to use it in depth. After creating a new shader called 3Dedge, we'll open it and add a new vertex color node and connect that vertex color node to the shader's Base Color fragment:



We'll create a new material for the shadow edges, and we'll set the shader of that material to be our 3Dedge shader:



With all these pieces in place, the color value of our mesh's vertices will be sent to the shader, which our edges are using, resulting in gradient edges.

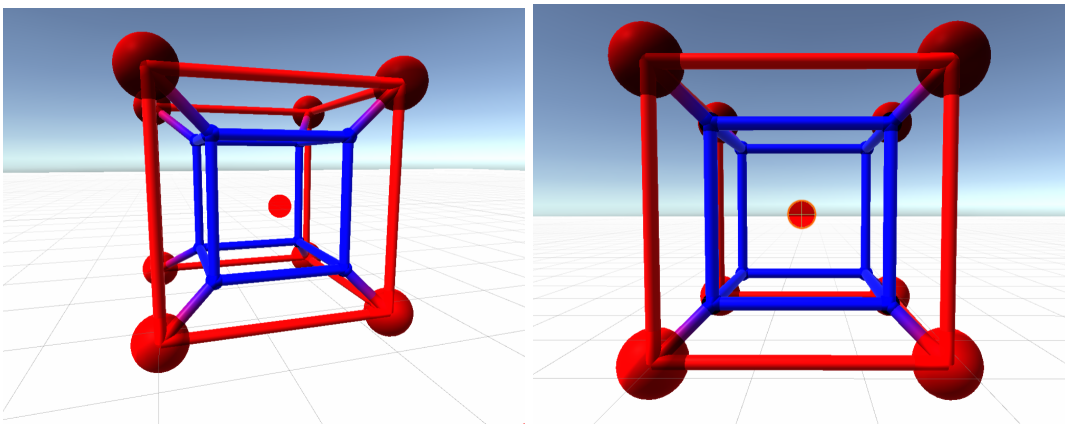
Fourth-Dimensional Matrix:

Implementing four-dimensional shapes and their three-dimensional shadows is relatively easy given all the code built for three-dimensional shapes and their two-dimensional shadows. The following is a list of the additional methods in the Matrix class that will be needed, and brief explanations.

Additional `matmul()` methods are needed to multiply a 4x4 matrix by a fourth-dimensional vector, as well as a 4x4 matrix by another 4x4 matrix. Six new methods are needed to generate the six possible basic rotation matrices (each of which rotate on a basic plane in four-dimensional space). No combined rotation matrix method is included, due to the high complexity of hand coding out six 4x4 matrices multiplied together. Additional versions of `rotate_point()` are built to take a fourth-dimensional point and a four-dimensional rotation matrix. Finally, `w_4d_3d()` is implemented, which closely resembles `y_3d_2d()`. The main difference is that `w_4d_3d()` uses max and min values calculated using the w coordinates of points, and calculates a y value for the shadow.

Fourth-Dimensional Shapes:

The Shape4D class also closely mirrors Shape3D. The most noticeable difference is the storage of the original fourth-dimensional vertex coordinate. Unity does not have a fourth-dimensional space to place objects in, so all fourth-dimensional objects we wish to project must be stored purely as data. Shape4D has two `Vector4` arrays; `save_points[]` and `points[]`. `save_points[]` acts like `vertices[]` in Shape3D, it stores a version of our fourth-dimensional vertices as if the shape was centered around the origin, x:0, y:0, z:0, w:0. `points[]` acts like the in-scene shape does for three-dimensional shapes, holding the position of the shape's vertices around a center vector, which does not have to be the origin. `Update_Edges()` and `Shadow_Update()` use `points[]` to calculate, while `Translate()` and `Rotate()` use `save_points[]`. Other noticeable differences are `Shape4D.load()`, which loads a few less game objects than `Shape3D.load()` because `points[]` acts as the four-dimensional shape, and we don't need to manage four-dimensional edges the user will never see. Another difference is that `Shape4D.Update_Edges()` updates the three-dimensional shadow's edges, not the four-dimensional digital object's shadow. The three-dimensional shadow of a four-dimensional cube in my software appears like so:



My Light:

A small amount of code is needed to manage the light source of the perspective projections. My Light has a script component called MyLight, which has a public float called `w`. `w` is used to pretend that My Light's position is a `Vector4` rather than a `Vector3`.

In MyLight `Update()`, if the position of My Light changes, we loop through every shape in 3D Shapes and every shape in 4D Shapes, calling their shadow functions so that their shadows change as the light moves.

Finally, My Light has an XR Grab Interactable component added to it. This allows a user to grab the light by pointing at it with their controller and pressing the grip button. The light then moves with their controller until they let it go, and allows it to extend the light out from themselves in a straight line.

User Inputs:

Because of the strangeness of interacting with four-dimensional objects, and the specificity of how unser interaction works in this software, Open XR was not used for the user's interactions with three-dimensional objects or four dimensional objects. Instead, a custom script called Controller is used. Two small capsule objects were added to the scene, called Left Capsule and Right Capsule, and Controller is added as a component to both. Controller has a private `ActionBasedController` named `controller`. The `ActionBasedController` object is inherited from the XR Interaction Toolkit, and is the game object of the Quest 2 controllers.

The `Update()` method of Controller sets the position and rotation of the capsule to be the same as the `controller`'s position and rotation. This means in the software, the user has a capsule at the location of both of their hands. `Update()` also manages the logic of the possible

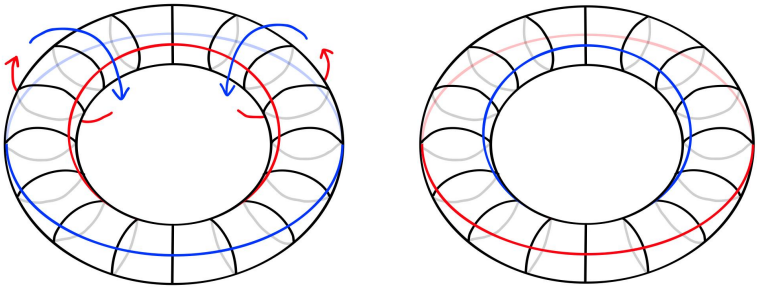
inputs of the software, the trigger button, grab button, and primary buttons of both controllers. Both the grab button and the trigger button have three methods in the controller related to their use: a beginning of input method, a continuous input method, and an end of input method. The trigger button has the `Grab()`, `Grabbing()`, and `Release()` functions. The grab button has the `Select()`, `Selecting()`, and `Unselect()` functions. `Update()` does not allow the trigger input and grab input from one controller to be used at the same time, so a user may only successfully use one of those buttons at a time. The primary buttons of both controllers follow a similar pattern, with if statements to check which controller the Controller script is attached to.

`Grab()` and `Select()` iterate through the three-dimensional shapes and four-dimensional shapes active in the scene, although `Select()` does not iterate through the three-dimensional shapes. Each shape's center is compared to the position of the controller. The first shape to be found that is close enough to the controller is set as Controller's global transform: `grabbed_shape`/`selected_shape`. For three-dimensional shapes, 'close enough' is calculated by multiplying the public range float of Controller, the scale of the shape, and a new public float added to Shape3D called `grab_range`. For fourth-dimensional shapes, a new method in Shape4D is added called `boundingbox()`. That method returns the minimum and maximum x, y, and z of all of the shadow spheres. These coordinates can be used to calculate a rectangular prism that fully encloses the shadow. If the controller is within that prism, the shadow is considered grabbed. At successfully grabbing or selecting an object, `Grab()` and `Select()` change the color of the capsule to blue or green respectively, while if no shape is found in range, the capsule is colored red. `Grab()` and `Select()` store the `position`, `position_offset` from the shape's center, and `starting_rotation` of the controller at the frame their corresponding button is pressed. After

`Grab()` or `Select()` is called, `Update()` ensures that `Grabbing()` or `Selecting()` is called every frame after, and that `Grab()` or `Select()` is not called again until the button is released.

`Grabbing()` calls the shape's `Translate()` and `Rotate()` function, based on the Controller's position, difference in position and difference in rotation from when the grab or select button was first pressed. `Grabbing()` translates three-dimensional shapes and four-dimensional shapes to wherever the controller is plus a `position_offset` vector so that when a user grabs a shape, it does not snap to the controller's location. This `position_offset` vector is rotated every frame based on the difference between the `starting_rotation` and current rotation of the controller, which makes rotating feel more natural. `Grabbing()` rotates a three-dimensional shape based on this rotational difference. `Grabbing()` rotates four-dimensional objects in three of the six four-dimensional rotations. The three rotations it rotates on are the three in which the w coordinate of the points do not change. These three rotations appear very similarly to the three basic rotations of three-dimensional space.

`Selecting()` can only interact with the three-dimensional shadows of four-dimensional objects. `Selecting()` rotates the selected fourth-dimensional object in the remaining three basic rotations in fourth-dimensional space. However, it does use the controller's rotation to calculate these rotations. Instead, it uses the controller's difference between its `starting_position` and its current position. This works visually because each of the final three fourth-dimensional rotations resemble a strange curling like-motion, each moving along the three different axes of three-dimensional space. This curling motion, most understandable with a fourth-dimensional cube, is like if you took a torus, and twisted it so the inside edge that makes up its 'donut hole' became its outermost edge, and vice-versa:



Moving the controller along the three axes then rotates these imaginary donuts in all three directions at once. Finally, Release() and Unselect() set the color of the controller to white, and set many of the variable's used for previous logic back to null.

The primary buttons use similar logic to the trigger and grab buttons. The primary button on the right hand changes the user from being able to see and interact with three-dimensional shapes to four, or vice-versa. The left controller's primary button completely resets the scene, moving all objects and data back to their original states.

Results

Overall, the software works as intended. I was surprised by the engagement testers found with the project, and the general eagerness of people who had heard of my project to try it. There definitely exists an interest in mathematical entertainment. The software itself also surprised me in how natural it felt to use, especially the fourth dimensional rotations. However, although intuitive, the abstract meaning of the three-dimensional shadows of four-dimensional shapes does not seem to bridge the gap as far as I had hoped. The program does not build a deep intuition of these higher dimensional shapes, although it certainly engages and educates about the subject, even if in the traditionally conceptually abstract way.

Discussion

Hardware:

Connecting VR headsets to computers is not easy to learn. To reasonably test my software on the Quest 2, I needed in a few seconds to be able to run a new version of the software on my headset. This requires the tethered connection, the alternative being building a version of the project to an online location to then download onto the headset. The time that would take would add up quickly, greatly slowing down development and killing and creative momentum. The main reason for the complexity of connecting a headset to a computer is the lack of standardization, learning resources, and consistency. Once learned, the process is not too difficult, but even decoding the terminology was a learning process on its own.

The computer hardware side caused its own difficulties. During the project, I had to purchase a new laptop, in part because my older laptop's graphics card was not of high enough quality. However, my new laptop could not connect to my HMD. My laptop actually has two different graphics cards, one of which was basic and much weaker than the modern graphics card I had bought my laptop for. The HMD was attempting to use this weaker graphics card, and so could not run. The roundabout solution was to disable the weaker driver. This didn't at first work. However after a 30 minute frustration break, upon returning to my computer the HMD was suddenly connected with my computer again. The randomness and un-explainability of this fix points to the fickle and uncertain nature that VR currently rests in. This project turned out to be a lot more 'I.T.' work than originally expected.

Software:

Open XR and XR Interaction Toolkit lacks clear API documentation with understandable examples. The API documentation is comprehensive and detailed, but its organization is not easy to parse and lacks approachable human explanation like that of the Unity API documentation. A good number of online tutorials are made by developers, but they soon become out of date, slowly culling the total number of useful references.

This project required me to learn a lot of new subjects at once. It was difficult to properly plan the structure of my code, because I lacked the full knowledge of what I was coding towards, and how the different required systems of my project would best interact. As a result of this, I had to refactor large sections of my code four or so times. This was not too difficult in itself, but it did soak up time. For example, an unfortunate fact about C# and Unity is that neither has a matrix library. I was able to find some C# matrix libraries online, but they were difficult to work with, as their API documentation was written poorly or just outdated, and the effort needed to get them to work seemed greater than the effort of just developing my own small matrix library for the few operations I needed. So after trying two libraries and failing, I refactored my code into my own matrix class.

Future Development:

Testing my software has been promising. Engagement with the project is overall positive. The software generally keeps a user's attention for ten minutes at a time. Improvements could be made. A problem that exists is the user inputs of rotations. My matrix multiplication system to generate rotations works perfectly well, and when called in code to update a shape’s rotation, appears visually pleasing and understandable. However, generating rotations from user

movement inputs requires converting the hand held controller's rotations into matrix multiplication. The rotations of these controllers are accessible as quaternions or euler angles. My code plugs these euler angles directly into the rotation matrix angle inputs. This solution is not mathematically sound, and the interpolation of the rotations has large gaps at certain angles, and is chaotic at other angles. A conversion from quaternions or euler angles to rotation matrices is needed. I found I did not have the time to learn and implement this, and that this was rarely brought up as a complaint by those who tested my software. However, the strangeness of the rotation interaction limits a user's potential for understanding what is happening when they rotate a fourth-dimensional shape.

A menu and tutorial would also benefit the program greatly. Menus help users navigate the application and understand its purpose. A tutorial is desperately needed if this project is to be ever used outside the context of demonstrations. In its current state, my verbal explanations are what ferry users into understanding what they are interacting with. An appreciation of the tools the software gives is hard to reach without a conceptual background. In that light, a text or audio tutorial, along with possibly some short puzzles in which users must match a fourth-dimensional shape's shadow to a target shadow would help users more deeply engage, value, and understand their experience.

More user inputs to manipulate shapes would allow users to explore the fourth-dimension in ever greater freedom. The ability to translate a fourth-dimensional shape or the light source along the w axis would help illuminate the shadow-like nature of the three-dimensional projections of four-dimensional objects. The ability to scale shapes would allow more users to

comfortably interface. And a more robust resetting system, with an undo button and light reset would help orient many users.

The most interesting expansion I can think of for this project is the addition of user created shapes. It is certainly possible for a user to set the position of new vertices and then connect edges from a new vertex to other vertices. To do so in four dimensions requires the user to also set the w coordinate of their vertex, likely using the thumb stick on a controller. This functionality gives an entirely new window into higher dimension, to the extent that I'm unsure if any other medium or piece of software has done something like it before. I believe the potential for users to gain a sense of how higher dimensions works would greatly increase with this feature.

Conclusion:

My software works well with verbal instruction to guide users. Geometric education in VR looks to have potential, with the promise of those learning directing their own education through playfulness rather than rigid instruction. However, this project has not provided clear intuition into how four-dimensional spaces work, but rather an approachable starting place.

Appendix

‘Jesse Explains’ by Marty Graham:
https://vimeo.com/705986547

```
45          return holder;
46      }
47
48      public static Vector4[] matmul(Vector4[] a, Vector4[] b)
49      {
50          Vector4[] holder = new Vector4[4];
51
52          for (int i = 0; i < 4; i++)
53          {
54              for (int j = 0; j < 4; j++)
55              {
56                  holder[i][j] = a[0][j] * b[i][0] + a[1][j] * b[i][1] + a[2][j]
                      * b[i][2] + +a[3][j] * b[i][3];
57              }
58          }
59
60          return holder;
61      }
62
63      //Generates a 3D rotation matrix around the x axis given an angle
64      public static Vector3[] rotate_x(float angle)
65      {
66          return new Vector3[] { new Vector3(1f, 0f, 0f),
67                                 new Vector3(0f, Mathf.Cos(angle), Mathf.Sin
                      (angle)),
68                                 new Vector3(0f, -1f * Mathf.Sin(angle),
                      Mathf.Cos(angle)) };
69      }
70
71      //Generates a 3D rotation matrix around the y axis given an angle
72      public static Vector3[] rotate_y(float angle)
73      {
74          return new Vector3[] { new Vector3(Mathf.Cos(angle), 0f, -1f *
                Mathf.Sin(angle)),
75                                 new Vector3(0f, 1f, 0f),
76                                 new Vector3(Mathf.Sin(angle), 0f, Mathf.Cos
                      (angle)) };
77      }
78
79      //Generates a 3D rotation matrix around the z axis given an angle
80      public static Vector3[] rotate_z(float angle)
81      {
82          return new Vector3[] { new Vector3(Mathf.Cos(angle), Mathf.Sin(angle),
                0f),
83                                 new Vector3(-1f * Mathf.Sin(angle), Mathf.Cos
                      (angle), 0f),
84                                 new Vector3(0f, 0f, 1f) };
85      }
86
```

```
87      //Generates a 3D rotation matrix given an x, y, and z angle
88      public static Vector3[] rotate3d(float xa, float ya, float za)
89      {
90          float cosx = Mathf.Cos(-xa);
91          float cosy = Mathf.Cos(-ya);
92          float cosz = Mathf.Cos(-za);
93          float sinx = Mathf.Sin(-xa);
94          float siny = Mathf.Sin(-ya);
95          float sinz = Mathf.Sin(-za);
96          return new Vector3[] { new Vector3(cosy*cosz, sinx*siny*cosz-cosx*sinz,
               cosx*siny*cosz+sinx*sinz),
97                                 new Vector3(cosy*sinz, sinx*siny*sinz+cosx*cosz,
                       cosx*siny*sinz-sinx*cosz),
98                                 new Vector3(-siny,     sinx*cosy,
                       cosx*cosy) };
99      }
100
101     //Generates a 4D rotation matrix on the xy plane, given an angle
102     public static Vector4[] rotate_xy(float angle)
103     {
104         return new Vector4[] { new Vector4(1f, 0f, 0f, 0f),
105                                new Vector4(0f, 1f, 0f, 0f),
106                                new Vector4(0f, 0f, Mathf.Cos(angle), Mathf.Sin
                    (angle)),
107                                new Vector4(0f, 0f, -1f * Mathf.Sin(angle),
                    Mathf.Cos(angle))    };
108     }
109
110     //Generates a 4D rotation matrix on the xz plane, given an angle
111     public static Vector4[] rotate_xz(float angle)
112     {
113         return new Vector4[] { new Vector4(1f, 0f, 0f, 0f),
114                                new Vector4(0f, Mathf.Cos(angle), 0f, Mathf.Sin
                    (angle)),
115                                new Vector4(0f, 0f, 1f, 0f),
116                                new Vector4(0f, -1f * Mathf.Sin(angle), 0f,
                    Mathf.Cos(angle))    };
117     }
118
119     //Generates a 4D rotation matrix on the xw plane, given an angle
120     public static Vector4[] rotate_xw(float angle)
121     {
122         return new Vector4[] { new Vector4(1f, 0f, 0f, 0f),
123                                new Vector4(0f, Mathf.Cos(angle), Mathf.Sin
                    (angle), 0f),
124                                new Vector4(0f, -1f * Mathf.Sin(angle),
                    Mathf.Cos(angle), 0f),
125                                new Vector4(0f, 0f, 0f, 1f)    };
126     }
```

```
127
128     //Generates a 4D rotation matrix on the yz plane, given an angle
129     public static Vector4[] rotate_yz(float angle)
130     {
131         return new Vector4[] { new Vector4(Mathf.Cos(angle), 0f, 0f, Mathf.Sin
            (angle)),
132                                new Vector4(0f, 1f, 0f, 0f),
133                                new Vector4(0f, 0f, 1f, 0f),
134                                new Vector4(-1f * Mathf.Sin(angle), 0f, 0f,
                       Mathf.Cos(angle))};
135     }
136
137     //Generates a 4D rotation matrix on the yw plane, given an angle
138     public static Vector4[] rotate_yw(float angle)
139     {
140         return new Vector4[] { new Vector4(Mathf.Cos(angle), 0f, Mathf.Sin
            (angle), 0f),
141                                new Vector4(0f, 1f, 0f, 0f),
142                                new Vector4(-1f * Mathf.Sin(angle), 0f,
                       Mathf.Cos(angle), 0f),
143                                new Vector4(0f, 0f, 0f, 1f)  };
144     }
145
146     //Generates a 4D rotation matrix on the zw plane, given an angle
147     public static Vector4[] rotate_zw(float angle)
148     {
149         return new Vector4[] { new Vector4(Mathf.Cos(angle), Mathf.Sin(angle),
            0f, 0f),
150                                new Vector4(-1f * Mathf.Sin(angle), Mathf.Cos
                       (angle), 0f, 0f),
151                                new Vector4(0f, 0f, 1f, 0f),
152                                new Vector4(0f, 0f, 0f, 1f)  };
153     }
154
155     //Rotates a 3D point, given a point and a 3D rotation matrix.
156     //Rotates around the origin, but can be given an alternative center point
157     public static Vector3 rotate_point(Vector3 p, Vector3[] rotate)
158     { return matmul(rotate, p); }
159
160     public static Vector3 rotate_point(Vector3 p, Vector3 c, Vector3[] rotate)
161     { return (matmul(rotate, p - c)) + c; }
162
163     //Rotates a 4D point, given a point and a 4D rotation matrix.
164     //Rotates around the origin, but can be given an alternative center point
165     public static Vector4 rotate_point(Vector4 p, Vector4[] rotate)
166     { return matmul(rotate, p); }
167
168     public static Vector4 rotate_point(Vector4 p, Vector4 c, Vector4[] rotate)
169     { return (matmul(rotate, p - c)) + c; }
```

```
170
171     //Given a 3D light l and a 3D point p, returns the 2D point
172     //of the perspective projection of p via l onto the ground plane.
173     //Also returns a scaling factor based on the points fractional hight
174     //in respect to the lowest and highest point of a shape
175     public static float[] y_3d_2d(Vector3 l, Vector3 p, float min_y, float
          max_y)
176     {
177         //Is the light above the plane
178         if (l.y > 0f)
179         {
180             //Is the light above the point and is the point above the plane
181             if (l.y > p.y & p.y > 0)
182             {
183                 float den = (p.y - l.y) / l.y;
184                 float new_x = l.x - ((p.x - l.x) / den);
185                 float new_z = l.z - ((p.z - l.z) / den);
186                 float scaling = Mathf.Clamp((p.y - min_y) / (max_y - min_y),
                      0f, 1f);
187                 return new float[] {new_x, scaling, new_z};
188             }
189             //Scalings of -1 are interpreted as points not to be shown
190             else { return new float[] { 0f, -1f, 0f }; }
191         }
192         //Isomentric projection
193         else { return new float[] { p.x, 0.5f, p.z }; }
194
195     }
196
197     //Given a 4D light l and a 4D point p, returns the 3D point
198     //of the perspective projection of p via l onto our 3D space.
199     //Also returns a scaling factor based on the points fractional w coordinate
200     //in respect to the lowest and highest w coordinate of a shape
201     public static float[] w_4d_3d(Vector4 l, Vector4 p, float min_w, float
          max_w)
202     {
203         //Is the light above our 3D space
204         if (l.w > 0f)
205         {
206             //Is the light above the point
207             if (l.w > p.w)
208             {
209                 float den = (p.w - l.w) / l.w;
210                 float new_x = l.x - ((p.x - l.x) / den);
211                 float new_y = l.y - ((p.y - l.y) / den);
212                 float new_z = l.z - ((p.z - l.z) / den);
213                 float scaling = Mathf.Clamp((p.w - min_w) / (max_w - min_w),
                      0f, 1f);
214                 return new float[] { new_x, new_y, new_z, scaling };
```

```
215             }
216             //Hides the point below the ground plane of the 3D space
217             else { return new float[] { 0f, -1f, 0f, 1f }; }
218         }
219         //Isometric Projection
220         else { return new float[] { p.x, p.y, p.z, 0.5f }; }
221
222     }
223
224     //Orients a cylinder such that it connects two points in space
225     public static void update_edge(Transform point1, Transform point2,
          Transform edge)
226     {
227         Vector3 point1v = point1.position;
228         Vector3 point2v = point2.position;
229
230         //Scales the cylinder to be the magnitude between the two points
231         edge.localScale = new Vector3(edge.localScale.x,
232                             (Vector3.Distance(point1v, point2v) / 2f) * (1f /
                        edge.parent.parent.localScale.x),
233                             edge.localScale.z);
234
235         //Rotates the cylinder to be the angle between the two points
236         edge.position = point2v;
237         edge.LookAt(point1);
238         edge.Rotate(90f, 0, 0);
239
240         //Positions the cylinder to be between the two points
241         edge.position = (point1v + point2v) / 2f;
242     }
243
244     //Orients a plane such that it connects two points in space
245     public static void update_shadow_edge(Transform point1, Transform point2,
          Transform edge)
246     {
247         Vector3 point1v = point1.position;
248         Vector3 point2v = point2.position;
249         float scale = point1.localScale.x * 4f;
250
251         //Scales the plane to be the magnitude between the two points
252         edge.localScale = new Vector3(0.005f * scale, 0.0003f,
253             (Vector3.Distance(point1v, point2v) / 2f) * (1f /
                  edge.parent.parent.localScale.x) * 0.2f);
254
255         //Rotates the plane to be the angle between the two points
256         edge.position = point2v;
257         edge.LookAt(point1, Vector3.zero);
258
259         //Positions the plan to be between the two points
```

```
260             edge.position = (point1v + point2v) / 2f;
261         }
262     }
263
```

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Shape3D : MonoBehaviour
6  {
7      [Range(0.001f, 4f)]
8      public float obj_scale = 1f; //A fraction to rescale the 3D shape by
9      [Range(0.001f, 10f)]
10     public float grab_range = 1f; //A fraction to fine tune grab interaction
         range
11
12     public Vector3 center = Vector3.zero; //The center points of the shape
13     [SerializeField] private int[] temp_edges; //Holds the adjacency
         information of shape's points. Inputed manuly
14
15     private Vector3[] vertices; //Holds a copy of shape's vertex's positions as
          if shape was at the origin
16     private float minp; //The largest y coordinate of all the vertices
17     private float maxp; //The smalles y coordinate of all the vertices
18
19     //Objects in the scene Shape3D needs to run:
20     private Transform mylight;
21     private Transform skeleton;
22     private Transform points;
23     private Transform edges;
24     private Transform projection;
25     private Transform shadow_points;
26     private Transform shadow_edges;
27     private Transform my_light;
28
29     // Start is called before the first frame update
30     void Start()
31     {
32         //Loads data, stores points in vertices, calcultes minp and maxp,
33         // and one time orients 3D edges and 2D shadow
34         load_objects();
35         center = skeleton.position;
36         vertices = new Vector3[points.childCount];
37         Vertex_Store();
38         minmaxp();
39         Update_Edges();
40         Shadow_Update();
41     }
42
43     // OnValidate is called when the script is loaded or when a value is
         changed in the inspector
44     void OnValidate()
45     {
```

```
46          //Scales object when inspector obj_scale is changed
47          this.gameObject.transform.GetChild(0).localScale = new Vector3
              (obj_scale, obj_scale, obj_scale);
48      }
49
50      //Loads the memory locations of all objects Shape3D needs
51      private void load_objects()
52      {
53          skeleton = this.gameObject.transform.GetChild(0);
54          points = skeleton.GetChild(0);
55          edges = skeleton.GetChild(1);
56          projection = this.gameObject.transform.GetChild(1);
57          shadow_points = projection.GetChild(0);
58          shadow_edges = projection.GetChild(1);
59          my_light = GameObject.Find("My Light").transform;
60      }
61
62      //Stores the positions of skeleton's points in vertices[]
63      // as if skeleton was centered at the origin
64      public void Vertex_Store()
65      {
66          for (int i = 0; i < points.childCount; i++)
67          {
68              vertices[i] = points.GetChild(i).position - center;
69          }
70      }
71
72      //Sets minp and maxp based on skeleton's points
73      private void minmaxp()
74      {
75          int max = 0;
76          int min = 0;
77          for (int i = 0; i < points.childCount; i++)
78          {
79              if (points.GetChild(i).position.y > points.GetChild
                  (max).position.y)
80              { max = i; }
81              else if (points.GetChild(i).position.y < points.GetChild
                  (min).position.y)
82              { min = i; }
83          }
84          maxp = points.GetChild(max).position.y;
85          minp = points.GetChild(min).position.y;
86      }
87
88      //Rotates the points of skeleton, using vertices[] current rotation
89      //Updates shape's edges and shadow afterwards
90      public void Rotate(float xRotation, float yRotation, float zRotation)
91      {
```

```
92          Vector3[] rotation = Matrix.rotate3d(xRotation, yRotation, zRotation);
93          for (int i = 0; i < points.childCount; i++)
94          {
95              Transform sphere = points.GetChild(i);
96              sphere.position = Matrix.rotate_point(vertices[i], rotation) +
                  center;
97          }
98          Update_Edges();
99          Shadow_Update();
100     }
101
102     //Changes the position of skeleton, using vertices[] current rotation
103     //Updates shape's edges and shadow afterwads
104     public void Translate(Vector3 translation)
105     {
106         center = translation;
107         for (int i = 0; i < points.childCount; i++)
108         {
109             Transform sphere = points.GetChild(i);
110             sphere.position = vertices[i] + center;
111         }
112         Update_Edges();
113         Shadow_Update();
114     }
115
116     //Iterates through all cylinders and orients them properly
117     public void Update_Edges()
118     {
119         for (int i = 0; i < edges.childCount; i++)
120             Matrix.update_edge(points.GetChild(temp_edges[i * 2]),
                  points.GetChild(temp_edges[i * 2 + 1]), edges.GetChild(i));
121     }
122
123     //Orients and colors the objects of shadow based on skeleton's vertices and
          my light
124     public void Shadow_Update()
125     {
126         minmaxp();
127         //for each vertex of skeleton
128         for (int i = 0; i < points.childCount; i++)
129         {
130             //Set the position of the vertex's shadow
131             Transform sphere = points.GetChild(i);
132             Transform shadow = shadow_points.GetChild(i);
133             float[] sv = Matrix.y_3d_2d(my_light.position, sphere.position,
                  minp, maxp);
134             shadow.position = new Vector3(sv[0], Mathf.Clamp(sv[1], -1f,
                  0.001f), sv[2]);
135
```

```
136             //Scale the shadow based on the origional vertex's lost y value
137             float scale = sphere.localScale.x * (obj_scale + sv[1] / 2f);
138             shadow.localScale = new Vector3(scale, 0.0001f, scale);
139 
140             //Change the color of the shadow based on the origional vertex's
                  lost y value
141             var shadowrend = (shadow.gameObject).GetComponent<Renderer>();
142             shadowrend.material.SetColor("_BaseColor", new Color(sv[1], 0f, 1f
                  - sv[1], 0.8f));
143         }
144         //for each edge of skeleton
145         for (int i = 0; i < edges.childCount; i++)
146         {
147             //Orient the shadow of edge so that it connects its two shadow
                  vertices
148             Matrix.update_shadow_edge(shadow_points.GetChild(temp_edges[i *
                  2]), shadow_points.GetChild(temp_edges[i * 2 + 1]),
                  shadow_edges.GetChild(i));
149 
150             //Create a new mesh and array of vertices for the edge shadow
151             Mesh mesh = shadow_edges.GetChild
                  (i).gameObject.GetComponent<MeshFilter>().mesh;
152             Vector3[] vertices = mesh.vertices;
153 
154             //Create new color array where the colors will be stored.
155             Color[] colors = new Color[vertices.Length];
156 
157             //Store the colors of the two vertices of this edge
158             var p1c = shadow_points.GetChild(temp_edges[i *
                  2]).GetComponent<Renderer>().material.color;
159             var p2c = shadow_points.GetChild(temp_edges[i * 2 +
                  1]).GetComponent<Renderer>().material.color;
160 
161             //Calculate the color of each vertex
162             float minz = vertices[0].z;
163             float maxz = vertices[vertices.Length - 1].z;
164             for (int j = 0; j < vertices.Length; j++)
165             {
166                 colors[j] = Color.Lerp(p2c, p1c, (vertices[j].z -minz)/(maxz-
                      minz));
167             }
168 
169             // assign the array of colors to the Mesh.
170             mesh.colors = colors;
171         }
172     }
173 }
```

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Shape4D : MonoBehaviour
6  {
7      [Range(0.001f, 2f)]
8      public float obj_scale = 1f; //Scale of the shape
9
10     public Vector4 center = new Vector4(0f, 0f, 0f, 0f); //4D center of the
         shape
11     public Vector4[] save_points; //Copy of the shape's vertices, but centered
         at 4D origin
12     [SerializeField] private Vector4[] points; //The current vertice positions
13     public int[] temp_edges; //Adjacency list of the vertices in the shape
14
15     private float minp; //The largest w coordinate of all the vertices
16     private float maxp; //The smallest w coordinate of all the vertices
17
18     //Objects in the scene Shape4D needs
19     private Transform projection;
20     private Transform shadow_points;
21     private Transform shadow_edges;
22     private Transform my_light;
23
24     // Start is called before the first frame update
25     void Start()
26     {
27         //Loads data, stores points in points, calcultes minp and maxp,
28         // and one time orients 3D shadow
29         load_objects();
30         points = new Vector4[save_points.Length];
31         for (int i = 0; i < save_points.Length; i++)
32         { save_points[i] = save_points[i] * obj_scale; }
33         for (int i = 0; i < save_points.Length; i++)
34         { points[i] = save_points[i] + center; }
35         minmaxp();
36         Shadow_Update();
37         Update_Edges();
38     }
39
40     // OnValidate is called when the script is loaded or when a value is
         changed in the inspector
41     void OnValidate()
42     {
43         //Scales object when inspector obj_scale is changed
44         /*for (int i = 0; i < save_points.Length; i++)
45         {
46             save_points[i] = (save_points[i] / last_obj_scale) * obj_scale;
```

```
47          }
48          last_obj_scale = obj_scale;*/
49      }
50
51      //Loads the memory locations of all objects Shape4D needs
52      public void load_objects()
53      {
54          projection = this.gameObject.transform.GetChild(0);
55          shadow_points = projection.GetChild(0);
56          shadow_edges = projection.GetChild(1);
57          my_light = GameObject.Find("My Light").transform;
58      }
59
60      //Stores the positions of points[] vectors in save_points[]
61      // as if points[] was centered at the origin
62      public void Vertex_Store()
63      {
64          for (int i = 0; i < save_points.Length; i++)
65          {
66              save_points[i] = points[i] - center;
67          }
68      }
69
70      //Sets minp and maxp based on points[] vectors
71      private void minmaxp()
72      {
73          int max = 0;
74          int min = 0;
75          for (int i = 0; i < points.Length; i++)
76          {
77              if (points[i].w > points[max].w)
78              { max = i; }
79              else if (points[i].w < points[min].w)
80              { min = i; }
81          }
82          maxp = points[max].w;
83          minp = points[min].w;
84      }
85
86      //Returns the smallest right rectangular prism
87      // that has a face parallel with the ground plane
88      // that the shadow of the shape could fit into
89      public Vector3[] boundingbox()
90      {
91          Vector3 def = shadow_points.GetChild(0).position;
92          Vector3 max = def;
93          Vector3 min = def;
94
95          for (int i = 1; i < shadow_points.childCount; i++)
```

```
96          {
97              Vector3 shadow = shadow_points.GetChild(i).position;
98
99              if (shadow.x > max.x)
100             { max.x = shadow.x; }
101             else if (shadow.x < min.x)
102             { min.x = shadow.x; }
103
104             if (shadow.y > max.y)
105             { max.y = shadow.y; }
106             else if (shadow.y < min.y)
107             { min.y = shadow.y; }
108
109             if (shadow.z > max.z)
110             { max.z = shadow.z; }
111             else if (shadow.z < min.z)
112             { min.z = shadow.z; }
113         }
114         return new Vector3[] { min, max };
115     }
116
117     //Rotates the vertices of points[], using save_points[] current rotation
118     //Updates shape's shadow afterwards
119     public void Rotate(float xyRotation, float xzRotation, float xwRotation,
          float yzRotation, float ywRotation, float zwRotation)
120     {
121         Vector4[] xyrot = Matrix.rotate_xy(xyRotation);
122         Vector4[] xzrot = Matrix.rotate_xz(xzRotation);
123         Vector4[] xwrot = Matrix.rotate_xw(xwRotation);
124         Vector4[] yzrot = Matrix.rotate_yz(yzRotation);
125         Vector4[] ywrot = Matrix.rotate_yw(ywRotation);
126         Vector4[] zwrot = Matrix.rotate_zw(zwRotation);
127         Vector4[] rotation = Matrix.matmul(Matrix.matmul(Matrix.matmul
              (Matrix.matmul(Matrix.matmul(xyrot, xzrot), xwrot), yzrot), ywrot),
              zwrot);
128         for (int i = 0; i < save_points.Length; i++)
129         {
130             points[i] = Matrix.rotate_point(save_points[i], rotation) + center;
131         }
132         Shadow_Update();
133         Update_Edges();
134     }
135
136     //Changes the position of the shape, using save_points[] current rotation
137     //Updates shape's shadow afterwards
138     public void Translate(Vector4 translation)
139     {
140         center = translation;
141         for (int i = 0; i < save_points.Length; i++)
```

```
142         {
143             points[i] = save_points[i] + center;
144         }
145         Shadow_Update();
146         Update_Edges();
147     }
148 
149     //Iterates through all the shadow's cylinders and orients them properly
150     public void Update_Edges()
151     {
152         for (int i = 0; i < shadow_edges.childCount; i++)
153         {
154             Matrix.update_edge(shadow_points.GetChild(temp_edges[i * 2]),
                  shadow_points.GetChild(temp_edges[i * 2 + 1]),
                  shadow_edges.GetChild(i));
155         }
156     }
157 
158     //Orients and colors the objects of shadow based on points[] vertices and
        my light
159     public void Shadow_Update()
160     {
161         minmaxp();
162         //for each vertex in points
163         for (int i = 0; i < save_points.Length; i++)
164         {
165             //Set the position of the vertex's shadow
166             Transform shadow = shadow_points.GetChild(i);
167             Vector4 sphere = points[i];
168             Vector4 light4d = new Vector4(my_light.position.x,
                  my_light.position.y, my_light.position.z,
                  my_light.gameObject.GetComponent<MyLight>().w);
169             float[] shadow_v3 = Matrix.w_4d_3d(light4d, sphere, minp, maxp);
170             shadow.position = new Vector3(shadow_v3[0], shadow_v3[1], shadow_v3
                  [2]);
171 
172             //Scale the shadow based on the origional vertex's lost y value
173             float scale = (obj_scale * 0.2f) + (shadow_v3[3] / 12f);
174             shadow.localScale = new Vector3(scale, scale, scale);
175 
176             //Change the color of the shadow based on the origional vertex's
                  lost y value
177             var shadowrend = (shadow.gameObject).GetComponent<Renderer>();
178             shadowrend.material.SetColor("_BaseColor", new Color(shadow_v3[3],
                  0f, 1f - shadow_v3[3], 0.8f));
179         }
180         //for each edge of skeleton
181         for (int i = 0; i < shadow_edges.childCount; i++)
182         {
```

```
183             //Create a new mesh and array of vertices for the edge shadow
184             Mesh mesh = shadow_edges.GetChild
                  (i).gameObject.GetComponent<MeshFilter>().mesh;
185             Vector3[] vertices = mesh.vertices;
186
187             //Create new color array where the colors will be stored.
188             Color[] colors = new Color[vertices.Length];
189
190             //Store the colors of the two vertices of this edge
191             var p1c = shadow_points.GetChild(temp_edges[i *
                  2]).GetComponent<Renderer>().material.color;
192             var p2c = shadow_points.GetChild(temp_edges[i * 2 +
                  1]).GetComponent<Renderer>().material.color;
193
194             //Calculate the color of each vertex
195             float miny = vertices[0].y;
196             float maxy = vertices[vertices.Length - 1].y;
197             for (int j = 0; j < vertices.Length; j++)
198             {
199                 colors[j] = Color.Lerp(p2c, p1c, (vertices[j].y - miny) / (maxy
                      - miny));
200             }
201
202             // assign the array of colors to the Mesh.
203             mesh.colors = colors;
204         }
205     }
206
207 }
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyLight : MonoBehaviour
{
    public float w = 10f; //A w value to emulate the light being a 4D points
    private float old_w; //The w of light last frame
    private Vector3 old_position; //The position of the light last frame

    // Start is called before the first frame update
    void Start()
    {
        old_position = this.gameObject.transform.position;
        old_w = w;
    }

    // Update is called once per frame
    void Update()
    {
        //Set the lights color based on its w value
        var lightrend = this.GetComponent<Renderer>();
        lightrend.material.SetColor("_BaseColor", new Color(1f - (w / 20f), 0f,
          w / 20f, 0.8f));

        //If the light changes position or w, iterate through all shapes
        Transform Shapes3d = GameObject.Find("3D Shapes").transform;
        Transform Shapes4d = GameObject.Find("4D Shapes").transform;
        if (this.gameObject.transform.position != old_position || w != old_w)
        {
            //Recalculate the shadow of each active 3D shape
            for (int i = 0; i < Shapes3d.childCount; i++)
            {
                if (Shapes3d.GetChild(i).gameObject.activeSelf)
                {
                    Shape3D projection = Shapes3d.GetChild
                    (i).GetComponent<Shape3D>();
                    projection.Shadow_Update();
                }
            }
            //Recalculate the shadow of each active 4D shape
            for (int i = 0; i < Shapes4d.childCount; i++)
            {
                if (Shapes4d.GetChild(i).gameObject.activeSelf)
                {
                    Shape4D projection = Shapes4d.GetChild
                    (i).GetComponent<Shape4D>();
                    projection.Shadow_Update();
                    projection.Update_Edges();
```

```
47                  }
48              }
49          }
50
51          //Set the old_ variables equal to their corresponding variables
52          old_position = this.gameObject.transform.position;
53          old_w = w;
54      }
55  }
```

```
1  using UnityEngine;
2  using UnityEngine.SceneManagement;
3  using UnityEngine.InputSystem;
4  using UnityEngine.XR;
5  using UnityEngine.XR.Interaction.Toolkit;
6  using UnityEngine.XR.Interaction.Toolkit.Inputs;
7
8  public class Controller : MonoBehaviour
9  {
10     [SerializeField] private ActionBasedController controller; //Game Object of
          controller
11     [SerializeField] private InputActionManager inputActionManager; //Input
         database of controller
12     private InputAction Primary_Button; //Primary Button memory location
13     [SerializeField] private Transform t_controller; //Controller's transform
14
15     private Vector3 position; //Position of controller
16     private Vector3 rotation; //rotation of controller
17
18     [Range(0.5f, 1f)]
19     public float range = 0.5f; //Adjustable Grab range of controller
20     [Space]
21     private Transform grabbed_shape; //3D or 4D shape that is grabbed
22     private Transform selected_shape; //4D shape that is selected
23
24     private Vector3 starting_position_offset; //Difference between initial grab
          position and center of shape
25     private Vector3 starting_position; //Initial grab position of controller
26     private Vector3 starting_rotation; //Initial rotation of controller
27     private bool isgrabbing;
28     private bool isselecting;
29     private int last_primary_button = 0;
30     private Shape3D shape3d_script; //Script of currently grabbed or selected
         shape
31     private Shape4D shape4d_script; //Script of currently grabbed or selected
         shape
32
33     // Start is called before the first frame update
34     void Start()
35     {
36         //Adjusts primary button based on which controller this script is
             attached to.
37         t_controller = controller.gameObject.transform;
38         if (controller.gameObject.name == "RightHand Controller")
39         {
40             Primary_Button = inputActionManager.actionAssets[0].FindActionMap
                 ("XRI RightHand").FindAction("Primary Button");
41         }
42         else if (controller.gameObject.name == "LeftHand Controller")
```

```
43          {
44              Primary_Button = inputActionManager.actionAssets[0].FindActionMap
                  ("XRI LeftHand").FindAction("Primary Button");
45          }
46          isgrabbing = false;
47          isselecting = false;
48      }
49
50      // Update is called once per frame
51      void Update()
52      {
53          //Updates position and rotation of controller
54          position = t_controller.position;
55          rotation = t_controller.rotation.eulerAngles;
56
57          //Orients capsule to controllers position and rotation
58          this.gameObject.transform.position = position;
59          this.gameObject.transform.rotation = t_controller.rotation;
60
61          //Stores if the controller is grabbing or selecting
62          float activate = controller.activateAction.action.ReadValue<float>();
63          float select = controller.selectAction.action.ReadValue<float>();
64
65          //Logic for grabbing input -> Initial grab, grabbing, release
66          if (activate > 0f && !isgrabbing && !isselecting)
67          {
68              isgrabbing = true;
69              Grab();
70          }
71          else if (activate > 0f && isgrabbing)
72          {
73              Grabbing();
74          }
75          else if (isgrabbing && activate == 0f)
76          {
77              isgrabbing = false;
78              Release();
79          }
80          //Logic for selecting input -> Initial selection, selecting, release
81          if (select > 0f && !isselecting && !isgrabbing)
82          {
83              isselecting = true;
84              Select();
85          }
86          else if (select > 0f && isselecting)
87          {
88              Selecting();
89          }
90          else if (isselecting && select == 0f)
```

```
91          {
92              isselecting = false;
93              Unselect();
94          }
95          //Logic for primary buttons of either controller
96          if (Primary_Button.ReadValue<float>() == 1 && last_primary_button == 0)
97          {
98              last_primary_button = 1;
99              if (controller.gameObject.name == "RightHand Controller")
100             {
101                 Change_Mode();
102             }
103         }
104         else if ((Primary_Button.ReadValue<float>() == 0 && last_primary_button
                == 1))
105         {
106             //Reset_Scene() is only called when button is released so infinite
                  resets do not occur
107             last_primary_button = 0;
108             if (controller.gameObject.name == "LeftHand Controller")
109             {
110                 Reset_Scene();
111             }
112         }
113     }
114
115     //Called the first frame the trigger button is pushed
116     private void Grab()
117     {
118         var rend = this.gameObject.GetComponent<Renderer>();
119         //If this controller currently has no grabbed shape
120         if (grabbed_shape == null)
121         {
122             //Iterate through the 3D shapes
123             Transform shapes3d = GameObject.Find("3D Shapes").transform;
124             for (int i = 0; i < shapes3d.childCount; i++)
125             {
126                 //If the indexed shape is active in the scene
127                 if (shapes3d.GetChild(i).gameObject.activeSelf)
128                 {
129                     Shape3D script3d = shapes3d.GetChild
                        (i).GetComponent<Shape3D>();
130                     Vector3 center = script3d.center;
131                     //If the controller is close enough to the shape
132                     if ((position - center).magnitude <= (range *
                        script3d.obj_scale * script3d.grab_range))
133                     {
134                         //Grab the shape
135                         grabbed_shape = script3d.gameObject.transform;
```

```
136                         shape3d_script = script3d;
137                         //Change the color of the capsule to blue
138                         rend.material.SetColor("_BaseColor", new Color(0f, 0f,
                        1f, 1f));
139                         //Store the current position offset from the shapes
                        center,
140                         // and the controllers rotation
141                         starting_position_offset = center - position;
142                         starting_rotation = rotation;
143                         //Stop interating through the 3D shapes
144                         break;
145                     }
146                 }
147                 //If not, color the capsule red
148                 rend.material.SetColor("_BaseColor", new Color(1f, 0f, 0f,
                    1f));
149             }
150         }
151         //If this controller currently has no grabbed shape
152         if (grabbed_shape == null)
153         {
154             //Iterate through the 4D shapes
155             Transform shapes4d = GameObject.Find("4D Shapes").transform;
156             for (int i = 0; i < shapes4d.childCount; i++)
157             {
158                 //If the indexed shape is active in the scene
159                 if (shapes4d.GetChild(i).gameObject.activeSelf)
160                 {
161                     Shape4D script4d = shapes4d.GetChild
                        (i).GetComponent<Shape4D>();
162                     Vector4 center = script4d.center;
163                     Transform shadow_points = shapes4d.GetChild(i).GetChild
                        (0).GetChild(0);
164                     //Calculate the bounding box of the shadow of the shape
165                     Vector3[] boundingbox = script4d.boundingbox();
166                     Vector3 min = boundingbox[0];
167                     Vector3 max = boundingbox[1];
168                     //If the controller is in the bounding box of the shape
169                     if ((position.x >= min.x && position.y >= min.y &&
                        position.z >= min.z)
170                         && (position.x <= max.x && position.y <= max.y &&
                        position.z <= max.z))
171                     {
172                         //Grab the shape
173                         grabbed_shape = script4d.gameObject.transform;
174                         shape4d_script = script4d;
175                         //Store the current position offset from the shapes
                        center,
176                         // and the controllers rotation
```

```
177                         Vector3 temp_center = center;
178                         starting_position_offset = temp_center - position;
179                         starting_rotation = rotation;
180                         //Change the color of the capsule to blue
181                         rend.material.SetColor("_BaseColor", new Color(0f, 0f,
                        1f, 1f));
182                         //Stop iterating through 4D shapes
183                         break;
184                     }
185                 }
186                 //If not, color the capsule red
187                 rend.material.SetColor("_BaseColor", new Color(1f, 0f, 0f,
                      1f));
188             }
189         }
190     }
191
192     //Called every frame that the trigger button is being pressed
193     private void Grabbing()
194     {
195         //If the grabbed shape is 3D
196         if (shape3d_script != null)
197         {
198             //Rotate the offset of the initial positional offset by change in
                  rotation of the shape
199             Vector3[] pos_rotation = Matrix.rotate3d((rotation.x -
                  starting_rotation.x) / 60f,
200                                                      (rotation.y -
                        starting_rotation.y) / 60f,
201                                                      (rotation.z -
                        starting_rotation.z) / 60f);
202             Vector3 position_offset = Matrix.rotate_point
                  (starting_position_offset, pos_rotation);
203             //Change the center of the shape to the controller's position + the
                   initial positional offset
204             shape3d_script.Translate(position + position_offset);
205             //Rotate the shape by the difference between the controllers
                  rotation and the initial contorller rotation
206             shape3d_script.Rotate((rotation.x - starting_rotation.x) / 60f,
207                                   (rotation.y - starting_rotation.y) / 60f,
208                                   (rotation.z - starting_rotation.z) / 60f);
209         }
210         //If the brabbed shape is 4D
211         else if (shape4d_script != null)
212         {
213             //Rotate the offset of the initial positional offset by change in
                  rotation of the shape
214             Vector3[] pos_rotation = Matrix.rotate3d((rotation.x -
                  starting_rotation.x) / 60f,
```

```
215                                                     (rotation.y -
                    starting_rotation.y) / 60f,
216                                                     (rotation.z -
                    starting_rotation.z) / 60f);
217             Vector3 position_offset = Matrix.rotate_point
                  (starting_position_offset, pos_rotation);
218             //Change the center of the shape to the controller's position + the
                   initial positional offset
219             Vector3 temp_total = position + position_offset;
220             shape4d_script.Translate(new Vector4(temp_total.x, temp_total.y,
                  temp_total.z, shape4d_script.center.w));
221             //Rotate the shape in 3 of the 6 fourth-dimensional rotations,
222             //(which resemble our 3 three-dimensional rotations)
223             //by the difference between the controllers rotation and the
                  initial contorller rotation
224             shape4d_script.Rotate(0f, 0f,
225                                   (rotation.x - starting_rotation.x) / 60f,
226                                   0f,
227                                   -(rotation.y - starting_rotation.y) / 60f,
228                                   (rotation.z - starting_rotation.z) / 60f);
229         }
230     }
231
232     //Called when the trigger button is released
233     private void Release()
234     {
235         //Set the color of the capsule to white
236         var rend = this.gameObject.GetComponent<Renderer>();
237         rend.material.SetColor("_BaseColor", new Color(1f, 1f, 1f, 1f));
238
239         //Update the storage of the shape's vertices
240         if (shape3d_script != null)
241         { shape3d_script.Vertex_Store(); }
242         if (shape4d_script != null)
243         { shape4d_script.Vertex_Store(); }
244
245         //Let go of shapes
246         grabbed_shape = null;
247         shape3d_script = null;
248         shape4d_script = null;
249     }
250
251     //Called the first frame the grab button is pushed
252     private void Select()
253     {
254         var rend = this.gameObject.GetComponent<Renderer>();
255         //If this controller currently has no selected shape
256         if (selected_shape == null)
257         {
```

```
258             //Iterate through the 4D shapes
259             Transform shapes4d = GameObject.Find("4D Shapes").transform;
260             for (int i = 0; i < shapes4d.childCount; i++)
261             {
262                 //If the indexed shape is active in the scene
263                 if (shapes4d.GetChild(i).gameObject.activeSelf)
264                 {
265                     Shape4D script4d = shapes4d.GetChild
                        (i).GetComponent<Shape4D>();
266                     Vector4 center = script4d.center;
267                     Transform shadow_points = shapes4d.GetChild(i).GetChild
                        (0).GetChild(0);
268                     //Calculate the bounding box of the shadow of the shape
269                     Vector3[] boundingbox = script4d.boundingbox();
270                     Vector3 min = boundingbox[0];
271                     Vector3 max = boundingbox[1];
272                     //If the controller is in the bounding box of the shape
273                     if ((position.x >= min.x && position.y >= min.y &&
                        position.z >= min.z)
274                         && (position.x <= max.x && position.y <= max.y &&
                        position.z <= max.z))
275                     {
276                         //Select the shape
277                         selected_shape = script4d.gameObject.transform;
278                         shape4d_script = script4d;
279                         //Store the initial position of the controller
280                         starting_position = position;
281                         //Change the color of the capsule to green
282                         rend.material.SetColor("_BaseColor", new Color(0f, 1f,
                        0f, 1f));
283                         //Stop iterating through 4D shapes
284                         break;
285                     }
286                 }
287                 //If not, change the color of the capsule to Red
288                 rend.material.SetColor("_BaseColor", new Color(1f, 0f, 0f,
                      1f));
289             }
290         }
291     }
292
293     //Called every frame that the grab button is being pressed
294     private void Selecting()
295     {
296         //If there is a 4D shape selected
297         if (shape4d_script != null)
298         {
299             //Rotate the shape in 3 of the 6 fourth-dimensional rotations,
300             //which do not resemble any of our three-dimensional rotations
```

```
            //by the difference between the controllers intial position and its current position
            Vector3 position_diff = (position - starting_position) * shape4d_script.obj_scale * 4f;
            shape4d_script.Rotate(position_diff.z, position_diff.y, 0f, position_diff.x, 0f, 0f);
        }
    }

    //Called when the grab button is released
    private void Unselect()
    {
        //Change the color of the capsule to white
        var rend = this.gameObject.GetComponent<Renderer>();
        rend.material.SetColor("_BaseColor", new Color(1f, 1f, 1f, 1f));

        //Update the storage of the shape's vertices
        if (shape4d_script != null)
        { shape4d_script.Vertex_Store(); }

        //Let go of shapes
        selected_shape = null;
        shape4d_script = null;
    }

    //Called when the right controller's primary button is first pressed
    private void Change_Mode()
    {
        //Let go of all shapes
        grabbed_shape = null;
        selected_shape = null;
        shape3d_script = null;
        shape4d_script = null;

        //Determine if 3D objects must be deactivate or activated
        Transform shapes3d = GameObject.Find("3D Shapes").transform;
        bool active3ds = shapes3d.GetChild(0).gameObject.activeSelf;

        //Swap the activation mode of 3D shapes and 4D shapes
        for (int i = 0; i < shapes3d.childCount; i++)
        {
            shapes3d.GetChild(i).gameObject.SetActive(!active3ds);
        }

        Transform shapes4d = GameObject.Find("4D Shapes").transform;
        for (int i = 0; i < shapes4d.childCount; i++)
        {
            shapes4d.GetChild(i).gameObject.SetActive(active3ds);
        }
```

```
347     }
348
349     //Called when the left controller's primary button is released
350     private void Reset_Scene()
351     {
352         //Reset the software
353         SceneManager.LoadScene("MainScene", LoadSceneMode.Single);
354     }
355 }
```

Bibliography

Abbott, Edwin Abbott, 1838-1926. Flatland: a Romance of Many Dimensions. New York :Dover Publications, 19531952.

Agić, Ana, Ella Murseli, Lidija Mandić, and Lea Skorin - Kapov. "The Impact of Different Navigation Speeds on Cybersickness and Stress Level in VR." *Journal of Graphic Engineering & Design (JGED)* 11, no. 1 (January 2020): 5–11. https://doi.org/10.24867/JGED-2020-1-005.

Alkemade, Remi & Verbeek, Fons & Lukosch, Stephan. (2017). On the Efficiency of a VR Hand Gesture Based Interface for 3D Object Manipulations in Conceptual Design. *International Journal of Human-Computer Interaction.* 33. 10.1080/10447318.2017.1296074.

Alves Fernandes, Luís Miguel, Gonçalo Cruz Matos, Diogo Azevedo, Ricardo Rodrigues Nunes, Hugo Paredes, Leonel Morgado, Luís Filipe Barbosa, et al. "Exploring Educational Immersive Videogames: An Empirical Study with a 3D Multimodal Interaction Prototype." *Behaviour & Information Technology* 35, no. 11 (November 2016): 907–18. https://doi.org/10.1080/0144929X.2016.1232754.

Angel, Edward, and Dave Shreiner. *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL*. 6th ed., Addison-Wesley Publishing Company, 2011.

"Augmented Reality (AR) Based Framework for Supporting Human Workers in Flexible Manufacturing." *Procedia CIRP* 96 (January 1, 2021): 301–6. https://doi.org/10.1016/j.procir.2021.01.091.

Banakou, D., & Slater, M. (2014). Body ownership causes illusory self-attribution of speaking and influences subsequent real speaking. *Proceedings of the National Academy of Sciences of the United States of America, 111*(49), 17678-17683. Retrieved March 25, 2021, from http://www.jstor.org/stable/43278738

Boellstorff, Tom. *Coming of Age in Second Life: An Anthropologist Explores the Virtually Human*. Princeton: Princeton University Press, 2015.

Fagernäs, Simon, William Hamilton, Nicolas Espinoza, Alexander Miloff, Per Carlbring, and Philip Lindner. "What Do Users Think about Virtual Reality Relaxation Applications? A Mixed Methods Study of Online User Reviews Using Natural Language Processing." *Internet Interventions* 24 (April 1, 2021): 100370. https://doi.org/10.1016/j.invent.2021.100370.

Hesch, Joel, Anna Kozminski, Oskar Linde *Powered by AI: Oculus Insight*. https://ai.facebook.com/blog/powered-by-ai-oculus-insight/. Accessed 4 May 2022.

Hillis, Ken. *Digital Sensations: Space, Identity, and Embodiment in Virtual Reality*. Electronic Mediations, v. 1. Minneapolis: University of Minnesota Press, 1999.

Hyperbolic, for PC, CodeParade, 2022.

"Interaction Design for Multi-User Virtual Reality Systems: An Automotive Case Study." *Procedia CIRP* 93 (January 1, 2020): 1259–64. https://doi.org/10.1016/j.procir.2020.04.036.

Manifold Garden, for PC, William Chyr Studio, 2020

Mungyeong Choe, Yeongcheol Choi, Jaehyun Park, and Hyun K. Kim. "Comparison of Gaze Cursor Input Methods for Virtual Reality Devices." *International Journal of Human-Computer Interaction* 35, no. 7 (July 2019): 620–29. https://doi.org/10.1080/10447318.2018.1484054.

Portal, for PC, Valve, 2007

Saker, Michael, and Jordan Frith. "Coextensive Space: Virtual Reality and the Developing Relationship between the Body, the Digital and Physical Space." *Media, Culture & Society* 42, no. 7–8 (October 1, 2020): 1427–42. https://doi.org/10.1177/0163443720932498.