Generalizations Of n-Space Geometric Projections and Transformations

Skerries Community College

Adam Kelly, Sam Enright

Stand Number: **2514**

January 2018

Comments

Comments

Summary

Both members of this group have a strong interest in the mathematics of higher dimensions. We are fascinated by geometric space in more complex forms than the typical two or three dimensional forms that were established by the ancient Greeks (most notably Euclid). When exploring topics such as these, we found that it helps to have visualization techniques to build up intuitions. This in turn then helps when dealing with the more rigorous mathematics of shapes and points in these higher dimensions. In the course of our research into higher dimensional space, we noticed an absence of visualization techniques, above that of four dimensional space.

With this in mind, we looked at current research papers on higher dimensional geometry and visualization, and saw a distinct lack of coherence. Most papers described techniques and formulae that were not generalized to an arbitrary number of dimensions. For example, many papers gave solutions for how to perform a rotation in four dimensions, though didn't generalize this concept to n-dimensions. Furthermore, papers varied widely, and many of the approaches detailed were incompatible with each other. One paper we found described rotation using techniques from algebraic geometry, which cannot be used with one describing translations using linear algebra, without great inefficiency.

We set out with the goal of developing a clear, implementable method of visualizing shapes in n dimensions. Our project aims were to generate formulae that would work in any specified number of dimensions. To visualize shapes in n-dimensions, we needed the transformations of translation, scaling around the origin and an arbitrary point, and rotation about the origin and an arbitrary plane. We also determined in our research that we would need the methods of perspective and orthographic projection to fulfill this goal.

When we first investigated transformations, we found many of the solutions to involve matrices and vectors. The matrices represented the transformations and the vectors the points. We came across two categories of transformations - affine and linear.

Linear transformations can, by definition, be represented using square matrices of size n - the number of dimensions being considered. 1.6.4. However, we found that translation was a different kind of transformation, namely an affine transformation. The corresponding matrix made use of homogeneous coordinates (see section 1.6.6). Homogeneous coordinates involve adding an additional element with the value of '1', to the vector representing the point, and adding an additional row and column to the matrix, with all values filled in as '0', except for bottom right value of the matrix, filled in as '1'. This inauspicious addition has the effect of adding an additional dimension to the transformation, which allows the translation transformation to be easily represented using the $(n+1) \times (n+1)$ matrix. See equation 2.2.

The use of homogeneous coordinates let us easily generalize the translation transformation into n-dimensions. For the sake of consistency, we used homogeneous coordinates to generalize all of the transformations in this report.

The scaling transformation is, by definition, equivalent to vector multiplication. We generalized this by finding the Diagonal Matrix (see section 1.6.4). The diagonal matrix has a vector along its diagonal, with all other elements as 0. The resulting matrix is equivalent to multiplying by that vector, meeting the definition of scaling.

The scaling described was scaling around the origin. To scale around an arbitrary point, the point was translated to the origin, and the same translation was applied to all other points being scaled. Then, after the scaling, the translation was applied in reverse.

Generalizing rotations was a more complex process. The concept of a rotation in higher dimensions is not well defined. We found that a number of research papers concluded different results ([Ban90], [Hol91], [McC03], [Han]). We saw the approach detailed by [Hol91] for the case of four dimensional space, could be generalized quite well into n dimensional space.

The approach was looking at the two dimensional planes, made by each of the axes of the coordinate system. Rotating these individually, and then composing these rotations allowed any rotation to be made up. This rotation method had a clear and simple implementation, but the number of planes that could be rotated increased greatly in higher dimensions, with $\binom{n}{2}$ (n choose 2, a combinatorics function) planes. This proved to be a challenge when developing the user interaction for the visualization software (see section 4.3).

To generalize the rotations, we used a method where a set of vectors is used to specify the space being rotated around. The first vector is translated to the origin, and the other vectors relative to that. Then the vectors are rotated to align with the axes, and the final rotation applied. The rotation to align the vectors with the axes, and the translations, are then reversed, completing the process. See listing 1.

After we had finished working on the generalizations of scaling, translation and rotation, we started work on the projections. A projection is a transformation between two spaces, where the resulting space has fewer dimensions then the initial space.

After researching types of projections, we found that two types of projections would be most effective: perspective projections and orthographic projections.

Perspective projections are similar to how we see the world. Parallel lines appear to 'meet' at a point, and sizes are not preserved, giving a perception of depth. The space has to be moved first to the view you want, then each vertex of the shapes in the space has to have the projection applied to it.

The other category of projection we looked at was orthographic projection. This type of projection is noted for its applications in technical drawing, and is characterized by its preservation of length and parallel lines.

We found that a projection that transforms an n dimensional space to an $n-1$ dimensional space can be applied recursively until the number of dimensions is 2. This is the approach we took when formalizing our method.

Using a perspective projection from three dimensions to two requires the use of a clipping volume (a shape where everything outside doesn't get rendered), to remove unwanted parts of the scene. We noticed that we could simplify the projection matrices for higher dimensions by only clipping the last 3D $\rightarrow$ 2D projection. These methods led us to generalize the projections to go from n dimensions to two. See section 3.2

After we had general matrices/formulae for the transformations and projections, we moved onto the last part of our project - writing a piece of software that would allow the visualization of an n dimensional shape.

We looked at methods of viewing shapes in 3D. One such method was wireframing, where the vertices and edges of an object are shown. Wireframing, while losing some surface information, shows all edges of an object, and can also show the relationships between them. In higher dimensions, wireframes allow you to see how the shape moves, without being obstructed by the surface of the object. A more in-depth exploration is in section 1.4.

There were a number of considerations we had when writing the software, all of which are explored in chapter 5. Some of these are the representations of the vertices and edges in the software, how to apply the various projections and transformations repeatedly and efficiently, and how to let the user interact with the software, given the vast number of parameters to the rotations.

Our finished software is available at `https://adamisntdead.github.io/higher-shapes`.

Throughout our project, we encountered few topics which were beyond the initial scope of our research, but became important parts of our project. One of these was calculating the coordinates and edges of the platonic solids in n dimensions. The platonic solids are the equivalent of 'regular shapes'. We needed to be able to calculate these so that we could show them in our visualization software.

We also considered 3D printing the projections of the objects, which is explored in section 5.2.

Lastly, we looked briefly at displaying curved surfaces using wireframes in n dimensions. Through the use of the advice given by mathematician and author Cliff Stoll (See Appendix B) and research, we were able to visualize a Klein bottle [Weic], Mübius strip, and a Clifford Torus using our software. This is done through parametric equations. Currently, our software can display parametric surfaces in n-dimensions.

Acknowledgements

We would like to thank our teachers, Ms. Corbett and Ms. Sullivan, for their feedback and support throughout the duration of our project. We would like to acknowledge Steve Hollasch, author of [Hol91], for quickly providing feedback on our software, and his encouragement. We would like to thank Cliff Stoll for sharing his knowledge on the construction of a Klein Bottle in four space, along with the Euler characteristic. We would like to thank Fiachra Knox, professor of mathematics in Simon Fraser University, for taking time to read our final report.

Lastly, we would like to thank our parents, for their continual patience, enthusiasm and financial support.

Contents

List of Figures

Chapter 1

Introduction

It is not obvious how to visualize shapes in more then three dimensions. This lack of ability has historically been a source of inspiration for writers, painters, and other artists. The idea was first thoroughly explored in literature in 'Flatland: A Romance of Many Dimensions' by the theologian Edwin Abbott [Abb84]. In the book, a race of people who exist on a 2D plane and scoff at the idea of a third direction past $up - down$ and $left - right$ challenges the reader's assumptions about the nature of dimensions.

The approach detailed in this report is one of visualizing the structure of shapes, and the positions of their vertices. The visualization technique used is wireframing, which involves looking at the vertices and edges of a shape and constructing the shape from there. The reasoning for this is detailed in the report.

This method can be applied to any dimensional space. This differs from more specific approaches, as it doesn't require the presence of special relations between the dimensions (such as coloring the model according to the position in a higher dimension).

1.1 Background To Our Research

Trying to visualize three dimensional space is complex. The conversion of a 3D model onto a 2D image on a screen is only possible as the result of much research and study. The primary methods of visualization used are focusing (blurring the background to give an illusion of depth), parallax (moving the background at different speeds compared to the foreground) and anaglyph (giving different colors to each eye, producing a stereoscopic effect). These methods are effective because the person looking at the images is familiar with the structure and the traits of what they are looking at. This, along with experience, helps the person viewing it extract all of the necessary information, to gain an accurate understanding of what is being presented, and to reconstruct the 3D scene.

Paring this back, it shows the reason why imagining more then three dimensions is hard. When rendering n dimensions, you are losing the ability to exactly represent $n - 2$ dimensions, having to instead compress them into a two dimensional space. The projection from n dimensions down to two is looking at a projection of a projection of a projection and so on. Thus the information loss is significant. This, along with a lack of intuition about the workings and nuances of these projections, make it hard to visualize.

Seeing visualizations of these higher dimensional shapes builds an intuition, through the use of motion and visual cues. This, to a large extent, is the focus of our research and this report.

1.2 History

The historical background of the idea of 'Four Dimensions' is explored in [Ban], the source for the majority of information presented in this section.

The first reference to a space having a *number of dimensions* dates back to Aristotle (384 - 322 B.C.) who wrote "The line has magnitude in one way, the plane in two ways, and the solid in three ways. Beyond this, there is no other magnitude because three are all.", and "There is no transfer into another kind, like the transfer from length to area and from area to a solid".

Ptolemy gave a proof in the 2nd century that there is not more then three distances in his book *On Distance*. Ptolemy wrote this so "because of the necessity that distances should be defined, and that the distances defined should be taken along perpendicular lines, and because it is possible to take only three lines that are mutually

perpendicular, two by which the plane is defined and a third measuring depth; so that if there were any other distance after the third it would be entirely without measure and without definition."

In the third century, Diophantus, an algebraist, used terms such as *cube-cube* to denote a cube times a cube. The algebraists, used to algebra linking to geometry, used powers greater then 3, but saw them as 'unreal', and avoided them.

By the late fifteenth century, higher powers were no longer avoided, but they were allowed only for a purely numerical use. Higher dimensions were seen as unnatural, perhaps most vehemently by John Wallis (1616-1703) when he described objects of a power higher then three as a "Monster in Nature, less possible than a Chimaera or Centaure." He is also quoted as saying "Length, Breadth and Thickness, take up the whole of Space. Nor can Fansie imagine how there should be a Fourth Local Dimension beyond these Three."

Writers in the eighteenth century began to consider the possibility of representing mechanics as a geometry of four dimensions, considering time to be the fourth dimension. This idea was written about in an article from 1754.

In the early part of the nineteenth century, the intellectual paradigm began to shift yet further away from mathematics purely being a representation of that which physically exists. As a consequence, more and more mathematicians began to study higher dimensional geometry, which was still in its infancy. In this exploration of the fundamentals of geometry, the very assumptions on which the great Greek mathematician Euclid built his geometry, which had been viewed with an almost Biblical degree of unequivocal support for over a millennium, began to be questioned.

The fifth postulate, or what would now be called an axiom, of Euclidean geometry states that through one point, only one parallel line can be drawn to a given line. This postulate began to be questioned by mathematicians, who wondered what conclusions they would be led to assuming a geometric space to which this statement didn't apply. When you assume that there are no parallel lines to a given line through a point, the resulting mathematics is that of elliptic geometry - the geometry that describes the surface of the earth. When you assume the opposite - that there are infinitely many lines parallel to a given line through a point - the resulting mathematics is that of hyperbolic geometry. This latter form of geometry, considered to be the more abstract of the two, was spearheaded by the mathematicians János Bolyai and Nikolai Ivanovich Lobachevsky in the early 19th century. For how abstract this mathematics is, it has still found modern applications in physics, including determining the shape of the universe.

This new wave of mathematics was met with a great degree of cynicism by those who insisted mathematics ought be more firmly rooted in what is "real". The mathematician Charles Dodgson (who is most famous for his writings under the pseudonym of Lewis Carroll) was one of the most vocal critics. One theory, predominately espoused by the literary scholar Melanie Bayley, states that Dodgson's *Alice's Adventures in Wonderland* was written as a satire on this Victorian trend of abstraction in mathematics. Bayley cites the cumbersome and complex nature of almost all of Dodgson's other writings as evidence of this. Her point is perhaps no better illustrated than in an anecdote involving Queen Victoria who, after asking Dodgson to dedicate his next book to her after liking *Alice in Wonderland* so much, had *An Elementary Treatise on Determinants* written in her name.

Toward the end of the 19th century, the number of books, memoirs and papers on the geometry of four or more dimensions had increased dramatically. Arthur Cayley (1821-1895) and William K. Clifford (1843-1879, who the Clifford torus is named for) both wrote large amounts on the subject. An area of interest had been uncovered, though it was lacking in popularity.

Some still argue that higher dimensional geometry is not necessary, and that geometry is intended for the study of concrete objects, and since we live in a three-dimensional world, we will never come across higher-dimensional shapes. Higher dimensional geometry, on the contrary, is intrinsically linked to science and is therefore directly applicable to mathematical physics. The basis of String Theory (more generally M-Theory) for example, is in higher dimensions. According to this theory, the laws of physics are emergent from the way in which additional spacial dimensions past our three are folded up on themselves, in what is called the Calabi-Yao manifold. This form of dimensionality is far beyond the scope of this project, though is nonetheless fascinating.

The notion that time is the fourth dimension became most concretely established in the early 20th century, with Albert Einstein's development of the Special and General Theories of Relativity. The current view amongst physicists is that our universe exists on a 4D "spacetime" - a curving geometry in which space and time are inextricably linked. In this 4D space, mass produces curvature, and this curvature determines the movement of that mass. To this day General Relativity remains the most elegant and complete physical model of gravity.

None of these mathematicians have, or will ever be able to, change the inescapable fact that we will not be able to see what higher dimensional objects would 'look' like. However, using visualization tools to reason about the shapes, our conceptual understanding of higher dimensional shapes can be increased. It must also be acknowledged that applications in physics are often found posthumously to those who developed the underlying mathematics, and that the most practical applications of higher dimensional generalizations may not be found for centuries.

1.3 Existing Work and Literature

Some of the first work done on visualizing / understanding higher dimensional space was, as previously stated, through literature and analogy, of two dimensional creatures trying to understand three dimensional space. ([Abb84])

Wireframes have been considered to view the structure of four-space shapes [Nol67]. Noll's approach consisted of generating pictures with a plotter and then transferring each drawing onto film, and variating these images rapidly to produce an animated movie. The movies that were produced offered a great degree of insight into four dimensional objects, but the lack of interaction was a downside, limited by the technology of the time (1967).

[McC03] details adding four dimensional rendering to OpenGL (A common graphics programming environment). The paper details the process of adding an extension to the library. The paper is written in a style which approaches it completely from a computer graphics point of view, and more specifically from the pardigrams which are needed in the OpenGL library.

[Zho92] presents methods of visualizing four dimensional space. It explores visual phenomena and its relations to higher dimensional space. This thesis also explores the applications of visualizing higher dimensional geometry.

1.4 Overview Of This Project

This project aims to generalize the transformations and projections used in normal two and three dimensional computer graphics, to visualize shapes and data in an n-dimensional space through wireframing. The finished tool, the visualization software, takes a list of vertices and a list of edges (a list of connections between the vertices), and produces images. These images are in two dimensions (displayed on a screen), and are 2D projections of the nD space. These images are interactive, can be rotated, and can have their shape and style of projection changed.

The limitations of this software are the same of any wireframe viewer, in that it does not include the surface data (what in 3D would be the faces, in 4D the cells, and so on). There is an information loss, but it is a very fast method, which doesn't involve the time it would take to, for example, raytrace the scene (the method used to create photorealistic 3D renders). It also allows the visualization to show the connections and relationships within the shape or data.

Another benefit, over other approaches to visualizing higher dimensional space, is the display of curves. This, while briefly touched upon, is largely out of the scope of this project, but is detailed in the 'Further Research' section of this report.

Lastly, this visualization technique can be used to create models which can be 3D printed, and seen in actual 3D space. This also is briefly touched upon in this report.

1.5 Contents Of This Report

Chapter 2 covers transformations that are needed in the visualization software. It includes transformations in 2D, along with their generalized nD counterparts, along with matrix representations of all of them. In 2.2, the generalized matrices are also generalized to arbitrary points, axes and vice versa (see 2.2).

This section also gives formal definitions for each of the transformations, to show the correctness of the generalizations.

Chapter 3 covers the process of visualizing three dimensional space, the rendering process and projections. It also covers the generalized projections which are needed for the visualization software.

Chapter 4 details the visualization software, and a guide to the implementation. It also includes images of the finished wireframes and software.

Chapter 5 covers all of the work done outside of the original project, such as generating the coordinates of the Platonic solids in n dimensions, creating 3D printable models of nD objects, and visualizing curves and functions.

Finally, Chapter 6 provides a conclusion to this research, including a description of the challenges that were faced during the project, as well as an outline of areas for further research and expansion relating to the work that can be found in this report.

1.6 Initial Definitions and Notation

This section is included to consolidate the notation used in this report to the reader. It is not intended to teach the concepts presented in this section, but this report should be accessible with a small amount of background knowledge in computer graphics, programming, and/or linear algebra.

1.6.1 Vectors

A Euclidean vector (usually referred to simply as a vector) is a geometric object that has magnitude (length) and direction.

A vector is frequently represented by a line segment, from point A to point B, and denoted $\overrightarrow{AB}$

Vectors also have a number of operations (the ones used in this report are outlined below).

This report uses coordinates quite heavily. The coordinates used are Cartesian coordinates.

In the Cartesian coordinate system, a vector can be represented by the coordinates of its initial and ending point. For example, the points $A = (0, 0, 1)$ and $B = (0, 1, 0)$ specify the vector $\overrightarrow{AB}$. The vectors in this report's initial point is the origin, $O = (0, 0, 0)$, thus a vector from the origin to A needs only A's coordinates.

In this report, a vector $v = (0, 0, 1)$ is a vector $\overrightarrow{OV}$, where V is the point $(0, 0, 1)$ in Cartesian coordinates.

The addition of vectors is simple using coordinates. To add to vectors, $v = (a, b, c)$ and $t = (d, e, f)$, you add corresponding coordinates, thus $v + t = (a + d, b + e, c + f)$.

The examples given so far have needed 3 coordinates to represent the vector. This is because it is in Euclidean 3-space. The vectors in this space are part of the vector space $\mathbb{R}^3$. This needs 3 real coordinates to specify a vector, and vector addition and scalar multiplication (multiplying all coordinates by a number). For n-dimensional Euclidean space, the vector space is $\mathbb{R}^n$, represented by a list of n real numbers.

1.6.2 Vector Basis

A vector space a set closed under vector addition and scalar multiplication. Every vector space has a vector basis. A vector basis of a vector space V is defined as a subset $v_1, \ldots, v_n$ of vectors in V that are are linearly independent (they cannot be made from adding other vectors in the basis), and span V, meaning $v_1, \ldots, v_n$ is a vector basis of V if every $v \in V$ can be written uniquely as

$$v = a_1 v_1 + a_2 v_2 + \cdots + a_n v_n$$

where $a_1, \ldots, a_n$ are scalars.

In $\mathbb{R}^n$, there are n vectors in the vector basis. An example of a vector basis of $\mathbb{R}^n$, called the standard basis, is the set of vectors

$$\{(1, 0, 0, \ldots, 0, 0), (0, 1, 0, \ldots, 0, 0), (0, 0, 1, \ldots, 0, 0), \ldots, (0, 0, 0 \ldots, 1, 0)(0, 0, 0 \ldots, 0, 1)\}$$

1.6.3 Subspaces Of Vector Spaces

[Weig] gives the following definition of subspaces:

Let $\mathbf{V}$ be a real vector space, then $\mathbf{W}$ is a real subspace of $\mathbf{V}$ if $\mathbf{W} \subset \mathbf{V}$ and for every $w_1, w_2 \in \mathbf{W}$ and $t \in \mathbb{R}$, $w_1 + w_2 \in W$, and $tw_1 \in W$.

1.6.4 Matrices

A matrix (plural matrices) is an array of numbers / expressions arranged in rows and columns. A matrix with 3 rows and 4 columns would be a 3×4 matrix. When a matrix has the same number of rows and columns, its called a square matrix.

Matrices are used to represent linear transformations. Any linear transformation can be represented by a matrix and vice versa.

[Weid] gives the following definition for the transformation given by a matrix.

The transformation given by the system of equations

$$\begin{aligned} x'_1 &= a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ x'_2 &= a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ &\vdots \\ x'_m &= a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n \end{aligned}$$

is represented as a matrix equation by

$$\begin{pmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_m \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

This also defines matrix multiplication by a vector (in the above, the matrix is being multiplied by the vector on the right, to give the vector on the left). Matrix multiplication is denoted by either the symbol $\cdot$ or $\times$

There are a few common matrices which come up quite often. The first is the identity matrix, which corresponds with the identity linear transformation. This matrix doesn't change the vector it is being multiplied with. It is given the name I or Id, and has the form

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

Another matrix which is used in this report is the diagonal matrix. The diagonal matrix that has non zero elements only in the diagonal running from the upper left to the lower right. The diagonal can be represented as a vector, for example, the vector $v = (1, 2, 3)$ would have the diagonal matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

This matrix, when multiplied by a vector, would give the same result as multiplying by the vector v.

1.6.5 Matrix Operations

There are two matrix operations which are used in this report. The first is Matrix Multiplication, or the multiplication of two matrices together to yield a new matrix.

The product, C of two matrices, A and B is $C_{i,k} = A_{i,j}B_{j,k}$, where j is summed over all possible values of i and k.

Matrix multiplication is associative and distributive, but not commutative, so order matters when multiplying matrices. Another note is when multiplying a vector v by matrices M_1, M_2 and M_3,

$$((vM_1)M_2)M_3 = v(M_1M_2M_3)$$

This is part of the reason of why matrices are used heavily in computer graphics. Instead of applying (multiplying) the matrices by the vector one by one, the matrices can be multiplied together. This is more efficient, as instead of multiplying all the individual matrices by the vector, only one matrix needs to be used.

GPU's (Graphics Processing Units) can also be used to multiply matrices together very quickly.

The second matrix operation used in this report is the matrix inverse. The matrix inverse corresponds to 'undoing' operation done by a matrix. The inverse of a matrix M is M^{-1}.

Formally $MM^{-1} = I$. The calculation of matrix inverses can be found at [Weie]

1.6.6 Homogeneous Coordinates

Homogeneous coordinates are defined formally as coordinates (x_1, x_2, x_3) of a finite point (x, y), for which

$$\frac{x_1}{x_3} = x$$

$$\frac{x_2}{x_3} = y$$

In an arbitrary amount of dimensions, n, the homogeneous point $p = (p_1, p_2, \ldots, p_n, p_{n+1})$ correspond to the point $(x_1, x_2, \ldots, x_n)$ in Cartesian Coordinates, where

$$\begin{aligned}
\frac{p_1}{p_{n+1}} &= x_1 \\
\frac{p_2}{p_{n+1}} &= x_2 \\
&\vdots \\
\frac{p_n}{p_{n+1}} &= x_n
\end{aligned}$$

Homogeneous coordinates are used over standard Cartesian coordinates for a number of reasons. Formulae and involving homogeneous coordinates are often simpler then the Cartesian coordinate counterpart. They also allow affine transformations (see [Weia]) to be easily represented by a matrix.

When using homogeneous coordinates with matrices, the point p is used as a vector, with $p_1, p_2, \ldots, p_{n+1}$ used as elements of the vector.

A point using Cartesian coordinates can easily be turned into a point using homogeneous coordinates by adding a 1 to the end, thus meeting the definition. This is what is done for the visualization program.

All transformations in this report use homogeneous coordinates.

1.6.7 Terms For Higher Dimensional Space

There are a number of terms used in this report to indicate the amount of dimensions that are in the Euclidean Space being considered. The terms *nD*, *n-Space*, and *n-Dimensional* are used interchangeably to denote the Euclidean Space has n dimensions.

When referencing matrices and vectors, the notation $\mathbb{R}^n$ is used to denote the Real Vector Space, denoted by n real numbers, which corresponds to the n-Dimensional Euclidean Space.

The term *Polytope* is used as a generalized version of polygon and polyhedron in n Dimensions.

The term *Higher Dimensional* indicates that the amount of dimensions being considered is greater then 3. With this, the term *Hyper* is also used alongside another term to indicate that it is in Higher Dimensional Euclidean space.

The term *Space* is used as an abbreviation for Euclidean space or Real Vector Space, depending on the context.

The term *Platonic Solids* refers to shapes in n dimensions which each face is a congruent, regular polygon. See section 5.1.

Chapter 2

Transformations in Higher Dimensional Geometry

In Euclidean geometry, a transformation is a function that changes a set of points. All of the transformations studied in this report are linear transformations (when using homogeneous coordinates), and thus can be written using matrices.

Definition: A transformation is a bijective (reversible) function $f : \mathbb{R}^n \to \mathbb{R}^n$.

The transformations which are studied in this report are *translation*, *scaling* and *rotation*.

2.1 Transformations in 2D

2.1.1 Translation

A translation is a geometric transformation that moves every point of a figure or a space by the same distance in a given direction. This is the same as the addition of a constant vector.

Definition: If v and p are arbitrary, fixed vectors, the translation operator T_v will work as $T_v \cdot \mathbf{p} = \mathbf{p} + \mathbf{v}$

Using homogeneous coordinates, the operation is very simple to represent as a matrix. In two dimensions, for the homogeneous vector v, the translation matrix is: ([see Vin05, section 1.10.3])

$$T_v = \begin{pmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{pmatrix}$$

Multiplying by the 2D homogeneous vector p,

$$T_v \cdot \mathbf{p} = \begin{pmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + v_x \\ p_y + v_y \\ 1 \end{pmatrix} = \mathbf{p} + \mathbf{v}$$

Which meets the definition.

2.1.2 Scaling

There are two types of scaling, uniform scaling and non-uniform scaling.

Definition: Uniform scaling is a geometric transformation that increases or diminishes objects by a scale factor that it the same in all directions. The resulting object is geometrically similar to the original. When the scale factor is 1, the objects are congruent.

Definition: Non-uniform scaling the scaling where at least one of the scaling factors is different from the others. The resulting shape is not similar to the original.

Definition: If v is a fixed vector, then the scaling operator S_v will work as $S_v \cdot \mathbf{p} = \mathbf{vp}$

To scale an object in 2D by a homogeneous vector v, each point p needs to be multiplied by the 2D scaling matrix ([see Vin05, section 1.10.1])

$$S_v = \begin{pmatrix} v_x & 0 & 0 \\ 0 & v_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Giving the result

$$S_v \cdot \mathbf{p} = \begin{pmatrix} v_x & 0 & 0 \\ 0 & v_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} = \begin{pmatrix} v_x p_x \\ v_y p_y \\ 1 \end{pmatrix}$$

Which meets the definition. The scaling is also uniform if and only if $v_x = v_y$.

2.1.3 Rotation

Definition: A rotation is a transformation which keeps a point fixed and keeps all other points the same distance from that point.

In two dimensions, there is only one angle needed to specify a rotation about the origin. The rotation is rotating the plane (or the point relative to the plane) anticlockwise by angle θ. For this project, θ will be specified in radians.

Looking at rotation from a matrix perspective, as we have the other transformations, the matrix must preserve the orientation of the vector space. This implies that the determinant of the matrix must be one. The distance from the point being rotated around must be preserved also, which implies that the matrix must also be orthogonal. This gives a definition for the rotation matrix:

Definition: A rotation matrix is a square matrix R with real entries such that $R^T = R^{-1}$ and $detR = 1$

Looking at the 2D rotation matrix R_θ ([see Vin05, section 1.10.4]),

$$R_\theta = \begin{pmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

Multiplying by a point p gives

$$R_\theta \cdot \mathbf{p} = \begin{pmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} = \begin{pmatrix} p_x cos\theta - p_y sin\theta \\ p_x sin\theta + p_y cos\theta \\ 1 \end{pmatrix}$$

This matrix corresponds with rotating the xy plane.

2.2 Transformations in Higher Dimensions

2.2.1 Translation

Translations can be easily generalized into n-dimensions. The use of homogeneous coordinates means that for an n-dimensional transformation, its matrix representation will be an $(n+1) \times (n+1)$ matrix.

Let v be the n-dimensional homogeneous vector $v = (v_1, v_2, \ldots, v_n, 1)$. The n-dimensional transformation matrix T_v is

$$T_v = \begin{pmatrix} 1 & 0 & \ldots & 0 & v_1 \\ 0 & 1 & \ldots & 0 & v_2 \\ & & \ddots & & \vdots \\ 0 & 0 & \ldots & 1 & v_n \\ 0 & 0 & \ldots & 0 & 1 \end{pmatrix} \tag{2.2}$$

It's easy to see that this meets the definition of the translation operator, as multiplying by the homogeneous vector p

$$T_v \cdot \mathbf{p} = \begin{pmatrix} 1 & 0 & \dots & 0 & v_1 \\ 0 & 1 & \dots & 0 & v_2 \\ & & \ddots & & \vdots \\ 0 & 0 & \dots & 1 & v_n \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \\ 1 \end{pmatrix} = \begin{pmatrix} p_1 + v_1 \\ p_2 + v_2 \\ \vdots \\ p_n + v_n \\ 1 \end{pmatrix} = \mathbf{p} + \mathbf{v}$$

An interesting point for the implementation of this matrix, is that it is the augmented matrix of the $n \times (n+1)$ identity matrix and the vector v, thus $T_v = (I|v)$.

2.2.2 Scaling

Scaling can also easily be generalized into n-dimensions. The matrix S_v becomes

$$S_v = \begin{pmatrix} v_1 & 0 & \dots & 0 & 0 \\ 0 & v_2 & \dots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & \dots & v_n & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix} \tag{2.3}$$

Multiplying by a homogeneous vector p

$$S_v \cdot \mathbf{p} = \begin{pmatrix} v_1 & 0 & \dots & 0 & 0 \\ 0 & v_2 & \dots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & \dots & v_n & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \\ 1 \end{pmatrix} = \begin{pmatrix} v_1 p_1 \\ v_2 p_2 \\ \vdots \\ v_n p_n \\ 1 \end{pmatrix}$$

As before, the scaling is uniform iff $v_1 = v_2 = \dots = v_n$.

Interesting notes for the implementation is that uniform scaling of a vector is equivalent to the scalar multiplication of a vector. For non-uniform scaling, the diagonal of the scaling matrix is the numbers (from the vector v) $v_1, v_2 \dots$ and so on.

Equation 2.3 describes scaling about the origin. To scale around a homogeneous point a, the translation matrix from a to the origin T_{-a} is calculated, along with the translation from the origin to a, T_a. The point to be scaled is transformed using T_{-a}, then scaled by S_v, then translated again by T_a.

As multiple transformation matrices are used, they can be multiplied together to get one matrix which preforms all of this (in order, because, as stated before, matrix multiplication is not commutative). Multiplying these together, you get

$$T_{-a} S_v T_a = \begin{pmatrix} v_1 & 0 & \dots & 0 & a_1 v_1 - a_1 \\ 0 & v_2 & \dots & 0 & a_2 v_2 - a_2 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & v_n & a_n v_n - a_n \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix} \tag{2.4}$$

2.2.3 Rotation

Rotation in three dimensions (and higher) is not commutative, unlike that in two dimensions. Rotations also need three angles to specify any rotation (about the axis, the rotation can be done with one angle, around an arbitrary segment). This can be done through the composition of 2D rotations on each of the 3 axes.

In general, in n dimensional space, a rotation will be around an $n-2$ dimensional hyperplane. Rotating from hyperplanes at the origin, to specify any rotation, $\binom{n}{n-2}$ angles are needed. Again this is not the case when using an arbitrary hyperplane.

For example, in 2 dimensions you need 1 angle, in 3 dimensions you need 3 angles, in 4 dimensions you need 6 angles, in 5 you need 10 and in 6 you need 15. This approach is generalized from [see Hol91, Chapter 2.2]

As mentioned before, the vector space $\mathbb{R}^n$ is specified by n basis vectors. These are used as the axis.

Labeling the standard basis vectors $\{X_1, X_2, X_3, \ldots, X_n\}$, any standard plane ($\mathbb{R}^2$) can be specified by 2 of these vectors. (To rotate around an $\mathbb{R}^{n-2}$ space, you need that many vectors to specify the space) In 2.1, this rotation, using the notation just described, is a rotation of the plane $\{X_1, X_2\}$ by θ, in the direction of X_2. In [DB94], there is a description of a rotation matrix, for rotating an $\mathbb{R}^2$ subspace, specified by 2 axes (standard basis vectors), by θ:

$$R_{a,b}(\theta) = \left[\begin{array}{c|l} & r_{a,a} = \cos(\theta) \\ & r_{b,b} = \cos(\theta) \\ & r_{a,b} = -\sin(\theta) \\ r_{i,j} & r_{b,a} = \sin(\theta) \\ & r_{j,j} = 1, \quad j \neq a, \quad j \neq b \\ & r_{i,j} = 0, \quad \text{elsewhere} \end{array}\right] \tag{2.5}$$

With this matrix, the complete rotation matrix can be made by multiplying 2.5 together, changing a and b to the indexes of the axes which specify the $\mathbb{R}^2$ subspace we are rotating.

To rotate around an arbitrary $\mathbb{R}^{n-2}$ subspace, let this subspace be specified by the set vectors $V = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{n-2})$, where $\mathbf{v}_m$ is a vector. let $\mathbf{v}_{m,n}$ represent the nth coordinate of the mth vector in V .

First, the vector $\mathbf{v}_1$ must be translated to the origin, with the matrix $T_{-\mathbf{v}_1}$. This matrix is then applied to all vectors. The other vectors are then rotated to align with the plane made from the nth and $n-1$th basis vector. To make $\mathbf{v}_{m,n} = 0$, the matrix $R_{n,n-1}(atan2(v_{m,n}, v_{m,n-1}))$ is used. (Here $atan2(x, y)$ is used instead of $atan(\frac{x}{y})$ as it gives the principle value of the arctan of $\frac{x}{y}$) .

These matrices which are orienting what is being rotated around are accumulated by multiplying the original $T_{-\mathbf{v}_1}$ by each of the rotation matrices. Finally, to calculate the actual rotation of θ, this accumulated matrix, M is multiplied by the rotation matrix $R_{n-1,n}(\theta)$. Finally, the inverse of the orientation matrices are applied, giving the result

$$M \cdot R_{n-1,n}(\theta) \cdot M^{-1}$$

This procedure is implemented in Listing 1, using Javascript and the math library *mathjs* by Jos De Jong.

```
const math = require("mathjs");

/**
 * @param {Number} theta - Rotation angle in radians
 * @param {Int} a - Index of the first axis specifying the rotation plane
 * @param {Int} b - Index of the second axis specifying the rotation plane
 * @param {Int} n - Number of dimensions
 */
function rotate(theta, a, b, n) {
  return math
    .eye(n + 1) // Uses Homogeneous Coordinates
    .subset(math.index(a - 1, a - 1), math.cos(theta))
    .subset(math.index(a - 1, b - 1), -math.sin(theta))
    .subset(math.index(b - 1, a - 1), math.sin(theta))
    .subset(math.index(b - 1, b - 1), math.cos(theta));
}

/**
 * @param {Number} theta - Rotation angle in radians
 * @param {Array} V - Array of vectors specifying the R^n-2 subspace
 * @param {Int} n - Number of dimensions
 */
function rotateAround(theta, V, n) {
  let matrix; // Accumulate The Matrices

  matrix = translate(math.multiply(V[0], -1), n);
  V = V.map(v => math.multiply(matrix, v));

  let currentMatrix;
  for (let i = 1; i < V.length - 1; i++) {
    for (let j = n - 1; j > i; j++) {
      // Make V_i,j 0
      currentMatrix = rotate(math.atan2(V[(i, j)]), j, j - 1, n);
      V = V.map(v => math.multiply(currentMatrix, v));
      matrix = math.multiply(matrix, currentMatrix);
    }
  }

  return math.multiply(
    math.multiply(matrix, rotate(theta, n - 1, n, n)),
    math.inv(matrix)
  );
}
```

Listing 1: Calculating the n dimensional rotation matrix by angle θ around the $\mathbb{R}^{n-2}$ subspace formed by the array of vectors V

Chapter 3

Visualizing n-Dimensions

To visualize n-space, you need to have a visualization technique that takes into account the position the space is being viewed from (the viewpoint), the direction being faced, and the orientation of the space from the viewpoint.

3.1 Visualizing Three Dimensions

First, visualization in three dimensions will be examined, then these techniques will be generalized to visualizing n-dimensions.

3.1.1 Perspective Projection

The first thing that is required for this form of projection is the viewpoint. The viewpoint, providing no orientation itself, can be specified by a point in 3D space. This point is also called the *from-point*.

Now the direction being faced needs to be specified. This can be specified as a point in the scene in which the 'camera' is facing, or as a vector from the viewpoint towards the direction being faced. Using a point, as opposed to a vector specifying the direction, is that it allows the rendering to 'track' that point as it moves through the space. This point is called the *to-point*.

The last thing needed to specify is the orientation. So far, the from and to points have specified the position and the direction, but has not specified what angle the space is being viewed from. The orientation can be thought of as making sure the final render is the right way up.

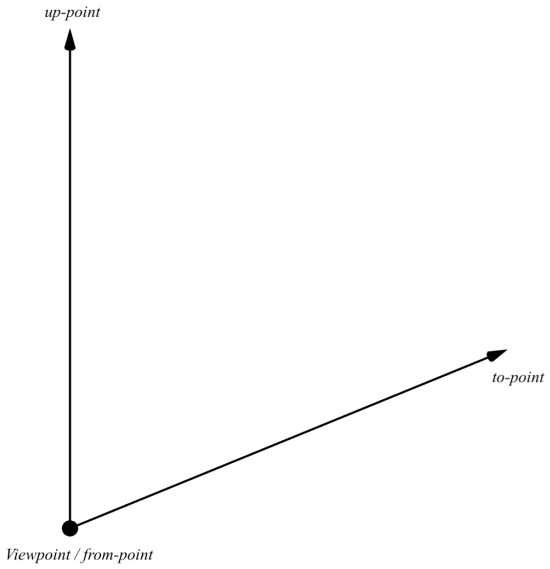


Figure 3.1: The *up-point*, *to-point*, and the *from-point* in 3D

This orientation vector is called the *up-vector*, and must not be parallel to the *to-vector*, as if they were parallel, there would be no orientation provided. See figure 3.1.

Objects are projected onto the screen based on a field of view, analogous to a cameras zoom and focus length, and by specifying a *clipping volume*. Objects outside the clipping volume cannot be seen.

The parameters are:

- Field of view: The total vertical angle of view
- Aspect: The aspect ratio of the screen (ratio of the width to the height)
- The near plane's z position
- The far plane's z position

The clipping volume is a truncated pyramid, with the near and far planes specifying the top and bottom of the pyramid.

See figure 3.2 for a diagram of these parameters.

To render the three dimensional space, the parameters are used to project the scene onto a two-dimensional rectangle, and the x and y coordinates are used then to display the image on the screen.

First, the 'camera' must be moved to the position specified by the from, to and up points. This is done by moving every point in the space.

The matrix used to preform this transformation is the equivalent of a rotation matrix. It is a linear transformation, so will not be specified by homogeneous coordinates, but can be used by appending a 0 to the ends of each row and column, and setting the bottom right corner to 1.

This matrix M, which will make the 'camera look at' the *to* point, from the *from*, with an orientation of *up*, is as follows:

$$\begin{aligned} M_3 &= to - from \\ M_2 &= up \times M_3 \\ M_1 &= M_2 \times M_3 \end{aligned} \tag{3.1}$$

Here the notation M_2 refers the the 2nd column of the matrix M, and is used as a vector for the purpose of this calculation. Here, $\times$ is the Cross Product (see [Weib]). The matrix is given in this form as the result of the calculation, in the general case, is large and doesn't provide any additional insight.

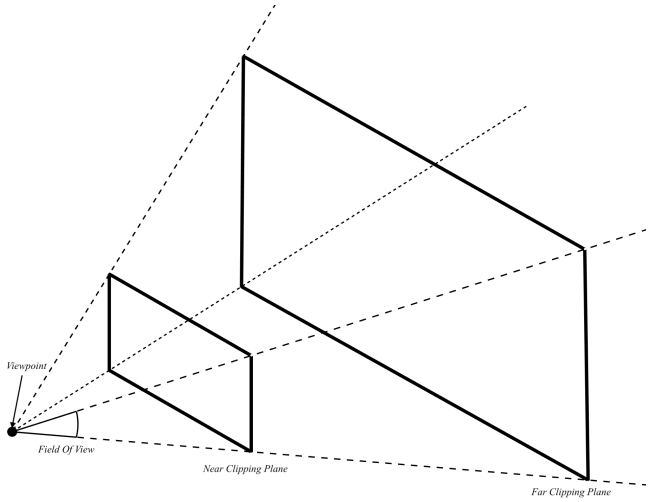


Figure 3.2: Perspective Projection Parameters

The perspective projection matrix is given at [Ope], using homogeneous coordinates:

$$\text{perspective(fovy, aspect, zNear, zFar)} = \begin{pmatrix} \frac{\frac{1}{\tan(\frac{\text{fovy}}{2})}}{\text{aspect}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\text{fovy}}{2})} & 0 & 0 \\ 0 & 0 & \frac{\text{zNear+zFar}}{\text{zNear-zFar}} & -1 \\ 0 & 0 & -2 \times \frac{\text{zNear} \times \text{zFar}}{\text{zNear-zFar}} & 0 \end{pmatrix}$$

What this matrix does is move the vertices around to the truncated pyramid in front of the viewpoint between the near and far clipping planes widening along the field of view angle (fovy). This also moves things that are farther away closer to the center by division by the distance coordinate. The whole projection matrix, P is as follows:

$$P = T_{-\text{from}} \cdot M_{\text{to, from, up}} \cdot \text{perspective}_{\text{(fovy, aspect, zNear, zFar)}}$$

The 'look at matrix' M has been extended to homogeneous coordinates for this.

To project the vectors in the space, the vectors must be homogeneous. Then, after multiplying by the matrix calculated above, P, the vector must normalized into a non homogeneous coordinate, then the z coordinate must be dropped.

The normalization is done as follows:

$$\text{normalize}(v) = \begin{pmatrix} \frac{v_0}{v_{n-1}} \\ \frac{v_1}{v_n - 1} \\ \vdots \\ \frac{v_{n-2}}{v_{n-1}} \end{pmatrix} \tag{3.2}$$

Making the whole process of projecting a vector v

$$normalize(\begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} \cdot P)$$

Which gives a 2D vector, using homogeneous coordinates. This vector can be turned into a standard $\mathbb{R}^2$ vector by normalizing again.

An interesting observation is that most of these processes are not very specialized to three dimensions, which makes generalizing them quite easy.

3.1.2 Orthographic Projection

Orthographic projection is the other category of projection that is considered in this project. Orthographic projection preserves parallel lines and doesn't add in 'depth'.

The simplest type of orthographic projection in 3D is onto the plane $z = 0$, with the matrix (in homogeneous coordinates)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In a more general case, the orthographic projection has 6 parameters, which define the clipping volume, *left, right, bottom, top, near, far*. These form a box with the minimum corner at $(left, bottom, -near)$ and the maximum corner at $(right, top, -far)$.

The orthographic projection matrix can be made up from two transformations, a scaling by the vector $(\frac{2}{right-left}, \frac{2}{top-bottom}, \frac{-2}{far-near})$ and a translation by the vector $(-\frac{right+left}{right-left}, -\frac{top+bottom}{top-bottom}, -\frac{far+near}{far-near})$

$$P = ST$$

Making the process of projecting a vector v

$$\text{normalize}(v \cdot P)$$

Which again gives a 2D vector, using homogeneous coordinates. This vector can also be turned into a standard $\mathbb{R}^2$ vector by normalizing again.

3.2 Visualizing n Dimensions

The projections which have just been shown move every point in the n dimensional space into an $n-1$ dimensional subspace. This gives a formal, general definition for a projection:
Definition: A projection is a transformation from $\mathbb{R}^n \to \mathbb{R}^m$, where $n > m$.

3.2.1 Perspective Projection

In three dimensions, the first thing that was done was to 'move the camera' to look at a point of interest. This is analogous in n dimensions, but does have some intricacies to take into consideration.

The cross product was used in the original calculation of M (see equation 3.1). Its use was to calculate a vector in 3D perpendicular to the 2 vectors given.

In n dimensions, the analog calculates a vector, mutually perpendicular to $n-1$ vectors. This is the same as calculating the normal vector (see appendices C.1).

To orient the 'camera', in n dimensions, there is another $n-2$ vectors needed, along with the *to* and *from* vectors needed. The orientation vectors are $B = b_1, \ldots, b_{n-2}$, which are all linearly independent, giving our generalized matrix M the form:

$$\begin{aligned} M_n &= to - from \\ M_{n-1} &= b_1 \times \cdots \times b_{n-2} \times M_n \\ M_{n-2} &= b_2 \times \cdots \times b_{n-2} \times M_{n-1} \times M_n \\ &\vdots \\ M_1 &= M_2 \times \cdots \times M_n \end{aligned} \tag{3.3}$$

The perspective projection matrix is somewhat simpler. When projecting from $\mathbb{R}^n \to \mathbb{R}^2$, only the field of view angle (*fovy*) needs to be taken into account. This gives P the form

$$\text{perspective}_n(\text{fovy}) = \begin{pmatrix} \frac{1}{\tan(\frac{fovy}{2})} & 0 & \dots & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fovy}{2})} & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix} \tag{3.4}$$

Notice that there is no need for a clipping volume at this point, as that will be done in the last projection from 3D to 2D.

The full projection matrix $P : \mathbb{R}^n \to \mathbb{R}^3$ is

$$P = T_{-from} \cdot M_{to,from,\mathbf{B}} \cdot \text{perspective}_n(\text{fovy}) \cdot \text{perspective}_{n-1}(\text{fovy}) \cdot \dots \cdot \text{perspective}_3(\text{fovy}) \tag{3.5}$$

Then to project a vector v normalize is applied $n-2$ times

$$normalize(normalize(\dots(v \cdot P))) \tag{3.6}$$

Which turns a vector from $\mathbb{R}^n \to \mathbb{R}^2$.

3.2.2 Orthographic Projection

In 3D, the orthographic matrix takes two vectors, specifying the minimum and maximum point of a box, and then moves the center of the box to the origin, and scales it to have each side with length 2.

The higher dimensional analog of this is two vectors specifying the minimum and maximum points of a hyperbox. The center of this box is found with the midpoint formula for the two vectors, and then it is scaled.

Let two vectors, $m = (m_1, \dots, m_n)$ and $k = (k_1, \dots, k_n)$ represent the minimum and maximum points of the box. The midpoint of these vectors is $\frac{m+k}{2}$. And to scale so each edge is of length 2, the scaling vector is $(\frac{2}{k_1-m_1}, \dots, \frac{2}{k_n-m_n})$. This gives the orthographic projection matrix P

$$P = S_{(\frac{2}{k_1-m_1},\dots,\frac{2}{k_n-m_n})} \cdot T_{-\frac{m+k}{2}} \tag{3.7}$$

Then again to project a vector v normalize is applied $n-2$ times

$$normalize(normalize(\dots(v \cdot P))) \tag{3.8}$$

Which again turns a vector from $\mathbb{R}^n \to \mathbb{R}^2$.

Chapter 4

Displaying Wireframes Of n Dimensional Objects

When designing and writing the visualization software, there was a number of considerations and problems which we had to solve. Many of these related to the logistics and sheer amount of data/parameters that comes with dealing with higher dimensional objects and meshes (representations of objects made up of simpler shapes).

The purpose of our software is to visualize higher dimensional objects using wireframes. It was around this central goal that this software was designed and built.

The choice of programming and graphics technology was made early on. The programming language *Javascript* was chosen, as it can be used cross platform and the code written is quite readable and high level. Initially, the programming languages C and Rust were considered, but the static typing combined with the fact that the software had to support an arbitrary amount of elements in the arrays (see 4.1) meant the pros, speed and type safety, were outweighed by the cons of the greatly increased prototyping time, along with the speed of modern computers. Despite this, during the testing phase it was discovered that the software showed a large amount of slowdown as the number of dimensions went above 13. This would suggest that further optimization would be required. Still, our software fulfills its purpose quite well, and the design is quite extendable.

The choice of C♯ and Unity was also considered, but the lack of support for a fully featured math and linear algebra library meant that this technology was not feasible for our use case.

After the choice of programming language, the choice of graphics technology came down to two categories. One option was that of using a simple 2D graphics library, where the rendering would be based on directly drawing lines on the screen. The advantages of this were performance and the level of agnosticism it would provide to our implementation methods about the underlying technology. The other option was that of using a 3D graphics library. This would make the rendering process about creating projections down to 2D and then using this to make meshes. The advantages of this approach were that there is a wide range of preexisting tools and libraries to work with. After considering possible uses outside the direct visualization software, we realized that using a 3D graphics library would allow the projection down to 3D, and from that models could be made, which could in turn be 3D printed.

The choice of graphics library was *Three.js*, a library now supported by Google.

4.1 Storing The Object Data

The wireframing approach to visualization requires two things to visualize an object, its vertices and its edges. The points in $\mathbb{R}^n$ are represented as a vector containing $n + 1$ elements. The simplest way of representing an ordered list of elements in Javascript is an array. These vertices were then themselves stored in another array.

The other component needed is the edges. Edges are a line between two vertices, thus the representation of an edge didn't need to include the vertices' coordinates itself, but a reference to that. This would avoid repetition. There is no pointers in javascript, so we used the indices of the array of vertices (the position in the array of a vertex) to refer to them. Thus an array $[2, 3]$ would represent an edge between the 3rd and 4th vertex in the array of vertices (The positions in an array start from 0).

After some research into the math libraries in Javascript, the library *mathjs* by Jos de Jong stood out as the most fully featured. Adam was also familiar with the library, having contributed to its development before,

which made it a natural choice for the visualization software.

Mathjs has in-build support for matrices along with all of the matrix and vector operations we needed (multiplication, inverse, cross product), which simplified our development process.

4.2 Projecting The Object Into 2 Dimensions

With the vertices represented as arrays, it became easy to apply various transformations. *Mathjs* supports arrays as vectors by default, so there was no conversion process necessary. A small geometry library was written to support the generalized transformations we had developed. This library (*js/geometry.js* in the source code) contains all of the matrices and formulae detailed in chapters 2 and 3.

To apply a matrix to the set of vectors was simple, but it meant that the original, n dimensional coordinates had to be stored alongside their 2D projected counterparts. After each application of a transformation, the vector had to be projected again to 2D. The projection matrices from equations 3.5 and 3.7 were precomputed to improve the performance of the software.

The rendering process of our software is:

1. Object input (vertices and edges)
2. Calculate transformation and projection matrices
3. Apply transformations
4. Project to 2D coordinates
5. Create *three.js* object from coordinates and edges
6. Display object
7. Return to 2.

4.3 User Interaction

As with any piece of software, the user interaction is one of the most important parts. It was decided from the start that the rotation of the n dimensional objects was the most essential part of the visualization.

Initially, various prototypes were built with different methods of interaction. Input via mouse position, sliders, and typing in amounts was explored, but all of the methods were either lacking in intuition or weren't extendable to an arbitrary amount of dimensions. A reason for this is the large amount of possible rotation planes, $\binom{n}{2}$, which have to be considered.

It was settled that the rotation had to by automatic, so a combinations function was used to rotate each 2D plane in the nD space by an angle θ, set by the user through the "Rotation Speed" input.

It was also planned to let the user choose whether to use an orthographic or perspective projection. It was decided that a toggle would be used, so the user could change the projection method without resetting the current rotation.

Lastly, when considering different input methods of shapes, we thought it would be useful to have a list of shapes, applicable to any dimension, which could show off the softwares capabilities. This is gone into detail in sections 5.1 and 5.3.

The vertices and edges of the object can be manually inputed (or the parametric equation, again see section 5.3), but the full list of shapes which are already available are

- n-dimensional hypercube
- n-dimensional simplex
- n-dimensional orthoplex
- Hecatonicosachoron (4D)
- Hexacosichoron (4D)

- Icositetrachoron (4D)
- Hexadecachoron (4D)
- Klein bottle (4D)
- Möbius strip (3D)
- Sphere (3D)
- Cliffords torus (4D)

4.4 Example Wireframe Images and Software Screenshots

Here is a collection of screenshots of the finished software and the wireframes generated by it. See Figure 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6.

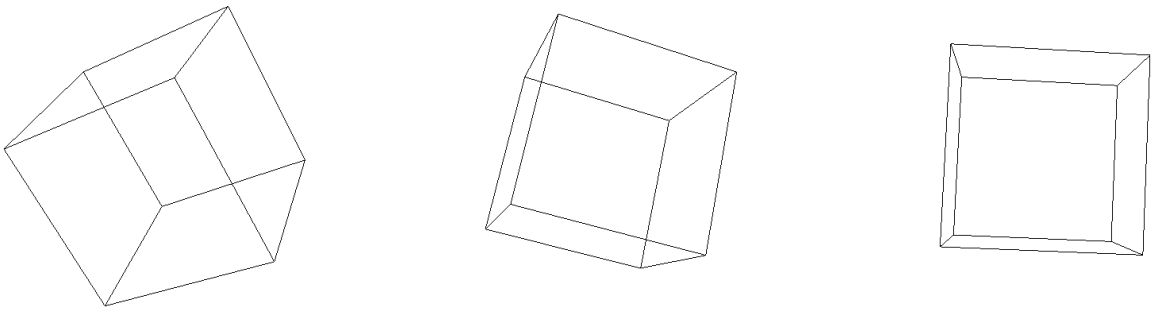


Figure 4.1: Wireframe Of A 3D Cube

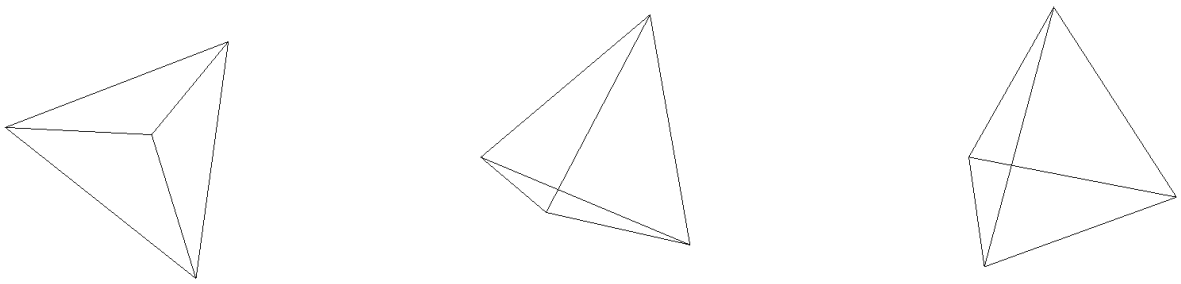


Figure 4.2: Wireframe Of A 3D Tetrahedron

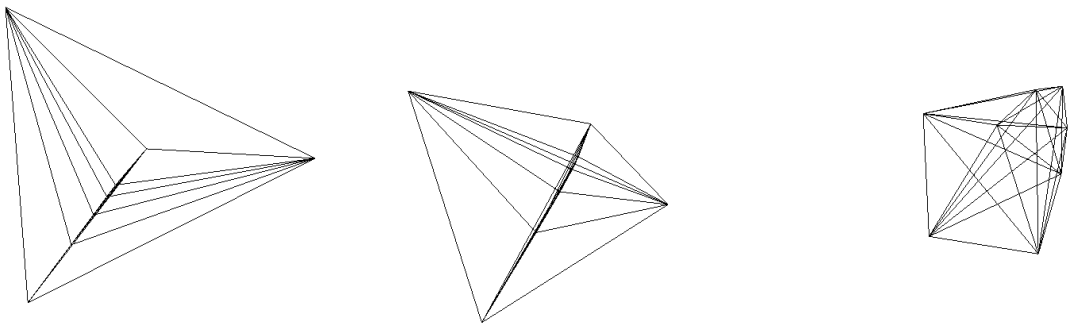


Figure 4.3: Wireframe Of A 7D Simplex

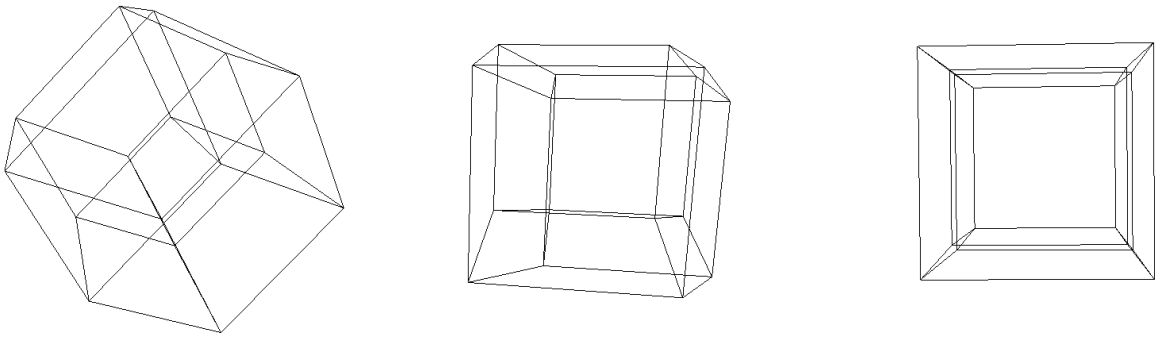


Figure 4.4: Wireframe Of A 4D Hypercube

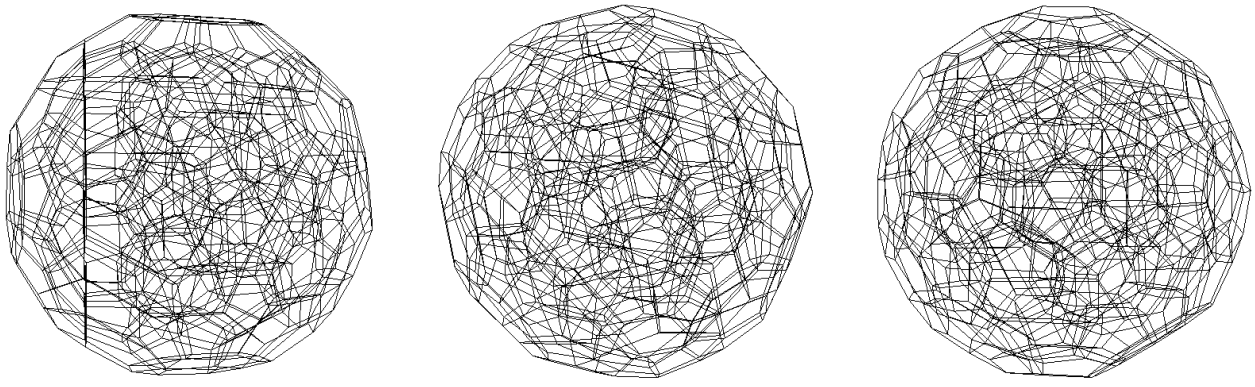


Figure 4.5: Wireframe Of A Hecatonicosachoron (120-Cell, 4D Platonic Solid)

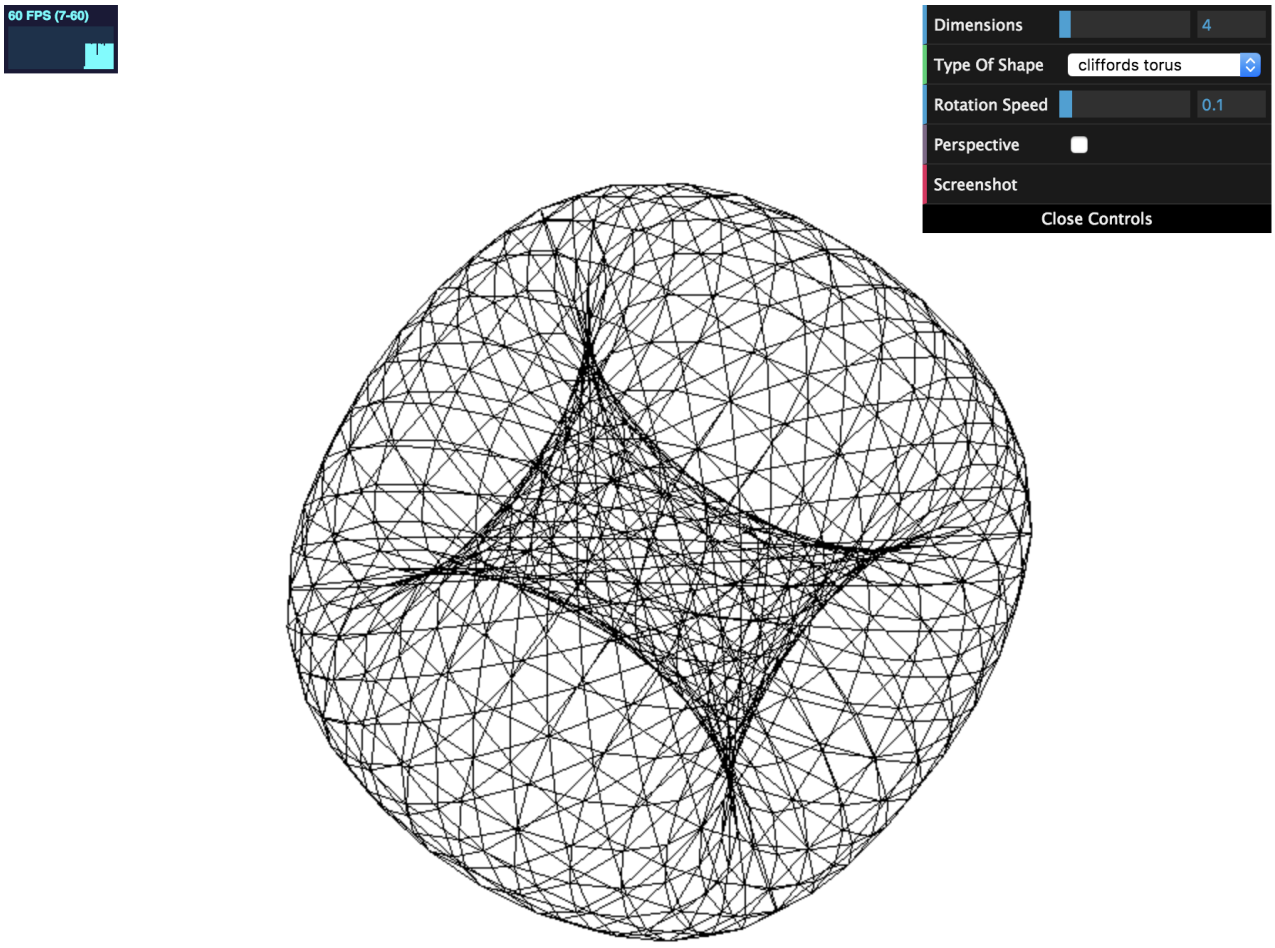


Figure 4.6: Our Software's User Interface Displaying A 4D Cliffords Torus

Chapter 5

Extensions Of Our Initial Project

5.1 Calculating The Vertices And Edges Of Regular Polytopes

There are three types of regular polytope which exists in any number n dimensions. These are:

- *Simplices*, this includes the equilateral triangle, the regular tetrahedron.
- *Hypercube*, this includes the square and cube.
- *Orthoplex*, this includes the square and the regular octahedron

There are other shapes which do not extend into n dimensions. In two dimensions, there are infinitely many regular polygons with an arbitrary number of sides. In three dimensions, the regular convex shapes (known as platonic solids) include the dodecahedron (twenty vertices) and the icosahedron (twelve vertices). In four dimensions, there is also the polyoctahedron (twenty four vertices), the polydodecahedron (six hundred vertices) and the polytetrahedron (one hundred and twenty vertices).

5.1.1 Regular 2D Polygons

The approach taken with these algorithms is to calculate the position of each vertex and the indexes of the vertices which are the endpoints for each edge. There is a well known algorithm for calculating the vertices of a regular n-gon. In a regular, convex n-gon, each interior angle has a measure of

$$\frac{(n-2)\pi}{n}$$

radians. Starting from the point $P_0 = (0, 1)$, it can be derived that the i^{th} point will be $P_i = (sin\frac{2\pi i}{n}, cos\frac{2\pi i}{n})$. It is clear that the distance from the origin to P_i must be one, as $\sqrt{cos^2\theta + sin^2\theta} = 1$.

The edges of the n-gon can also be calculated quite simply. Each point connects to the next point, with the last point connected to the first. Each point P_i is connected to $P_{(i+1) \mod n}$. The modulus is used to connect P_n to P_0. This procedure is implemented in javascript in Listing 2.

5.1.2 Regular Simplices

The simplex (plural simplices), is the generalized notion of triangles and tetrahedrons to arbitrary dimensions. The *n-simplex* is an n-dimensional polytope which is the convex hull of its $n + 1$ vertices.

The 2-simplex is called a triangle, a 3-simplex is a tetrahedron and a 4-simplex is a 5-cell.

There is an easy way to form / calculate the points of a regular simplex. This is called the standard or probability simplex. It a simplex formed from the $n + 1$ standard unit vectors. Formally:

$$\{x \in \mathbb{R}^{n+1} : x_0 + \cdots + x_n = 1, x_i \geq 0, i = 0, \ldots, n\}$$

This however leaves the shape embedded in $(n + 1)$-dimensional space. It could be rotated, but there is a different algorithm which can generate them in n-dimensional space. Using the first $n - 1$ points from the formula above, every coordinate of the last point is $\frac{-1}{1+\sqrt{1+n}}$.

The way its calculated in Listing 3 is each row of an $n \times n$ identity matrix is used as a point. Then, a n-space vector filled with ones is multiplied by $\frac{-1}{1+\sqrt{1+n}}$, giving us all of our coordinates.

```
/**
 * Makes a point into a homogeneous point
 *
 * @param {Array} p - The point
 */
function homogeneous(p) {
  return p.concat(1);
}

/**
 * @param {Int} n - Number of Sides
 */
function polygon(n) {
  let vertices = [];

  for (let i = 0; i < n; i++) {
    vertices.push([
      Math.sin(i / n * 2 * Math.PI),
      Math.cos(i / n * 2 * Math.PI)
    ]);
  }

  let edges = [];

  for (let j = 0; j < n; j++) {
    edges.push([j, (j + 1) % n]);
  }

  return [vertices.map(homogeneous), edges];
```

Listing 2: Calculate The Vertices And Edges Of An n Sided Polygon (2D)

The edges between the vertices form a complete graph. Thus the edges can be calculated by calculating all combinations of two vertices.

5.1.3 Regular Hypercubes

The hypercube is the generalized notion of squares and cubes into arbitrary dimensions. The hypercube is a convex figure which consists of groups of opposite parallel line segments. The hypercube is also referred to as the n-cube or measure polytope. It is also a case of an n-orthotope, the generalized notion of a rectangle.

The hypercube in n-dimensional space has 2^n vertices, $2^{n-1}n$ edges, and the longest diagonal is $\sqrt{n}$.

There is also a simple algorithm to calculate the coordinates of a unit hypercube. All of the points of the unit hypercube can be given as all sign permutations of the coordinates $(\pm\frac{1}{2}, \pm\frac{1}{2}, \ldots, \pm\frac{1}{2})$.

The edges of a unit hypercube are of unit length, thus you can generate all combinations of 2 vertices, and then remove those that are not the correct length.

This is implemented in Listing 4, again using lodash.

5.1.4 Regular Orthoplexes

The orthoplex is the generalized notion of squares and octohedrons into arbitrary dimensions. It is also called the n-orthoplex. It has $2n$ vertices and $2(n-1)n$ edges.

The vertices of an orthoplex can be given as all permutations of $(\pm 1, 0, 0, \ldots, 0)$. The edges can be calculated in a similar way to that of a hypercube, but checking that the Euclidean distance between the points is $\sqrt{2}$.

This is implemented in Listing 5.

5.2 3D Printing Projections Of n-Dimensional Objects

An interesting idea that removes the need to project from 3D to 2D is that of 3D printing the wireframe projections of the objects. To be 3D printed, the object's mesh has to be watertight, and should be able to support itself during the printing process. Using a similar process as that of visualizing the objects, the object's vertices get projected into 3D, and then small spheres are added to a mesh at these points. Then, using the data given for the edges, each pair of vertices are used as start and endpoints for cylinders, which again get added to the mesh. Finally, the mesh is merged together to give one final mesh which can be printed.

The spheres are used for two reasons, to highlight the vertices, and to join the cylinders (edges) and ensure they are watertight. When the vertices are being projected, a perspective projection is needed, especially for shapes such as the hypercube, otherwise the mesh has many intersections which degrade the 3D models visual impact.

The mesh created then has to be taken into a 3D printing program, to fix any errors that occur during the export of the mesh and to send to the 3D printer.

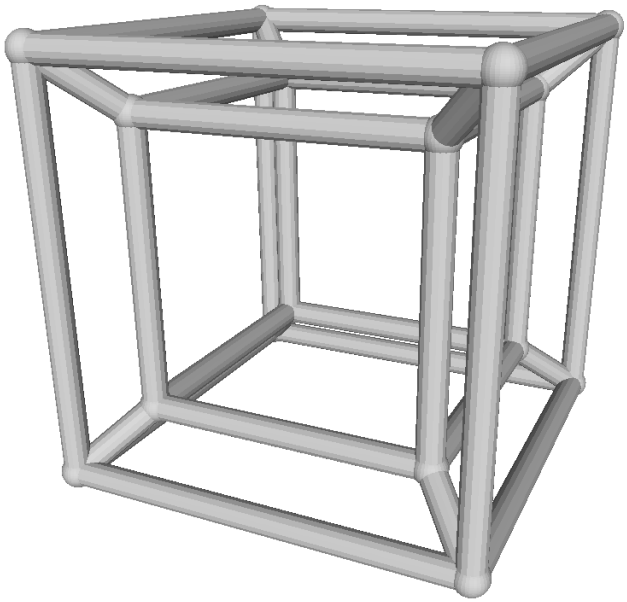


Figure 5.1: Mesh Of A 4 Dimensional Hypercube, projected into 3 Dimensions and prepared for printing.

5.3 Displaying Curves and Graphing Functions Of n Variables

When writing the software, we thought that a strong application and extension of our project would be the display of curved objects, surfaces and with that, the graphs of functions with many variables. This would allow the visualization software to show the user their data in a more intuitive way then numbers on a page.

A well known and common shape is that of a Klein bottle ([Weic]). A Klein bottle is similar to the Möbius strip in that in $\setminus^4$, it has only one side. While seeking a way to visualize these objects using out software, we reached out to mathematician, physicist and author Cliff Stoll, asking about the triangulation of a Klein bottle (turning it into a mesh). While unable to directly give us a method, he did however describe in detail the topological construction of a Klein bottle, and the relation of the vertices to the Euler Characteristic. This gave a large amount of insight which enabled us to solve our problem.

Whilst searching for the usual methods of writing a 3D function plotter, the topic of parametric equations and surfaces came up.

A parametric equation is a group of quantities as functions of one or more independent variables called parameters. ([Weif]).

Parametric equations can be used to express the coordinates of curves or surfaces.

Using parametric equations, we wrote a program to take parametric equations and the amount of complexity in the resulting mesh, and return the vertices and edges which could in turn be displayed using our visualization software.

5.3.1 Klein Bottle

The first curved object that was studied was the Klein bottle. [Fer17] gives the following parametric equation (called a parametrization) for the Klein bottle

$$\begin{aligned} x &= (1 + \cos v)\cos u \\ y &= (1 + \cos v)\sin u \\ z &= \sin v \cos \frac{u}{2} \\ w &= \sin v \sin \frac{u}{2} \end{aligned} \tag{5.1}$$

Where w is the fourth dimensional coordinate, $0 \leq u \leq 2\pi$ and $0 \leq v \leq 2\pi$. The resulting mesh is shown in figure 5.2.

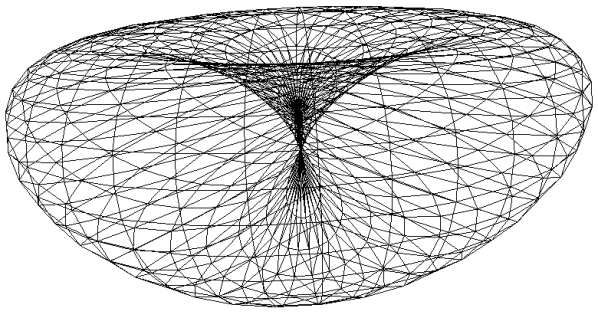


Figure 5.2: Mesh Of A Klein Bottle, Generated From Equation 5.1

5.3.2 Möbius Strip

The Möbius strip is a three dimensional shape with one edge and one side. It can be constructed by taking a long, narrow piece of paper, and twisting it once at the end and joining like a cylinder. It's parametrization is

$$\begin{aligned} x &= (1 + \frac{v}{2}\cos \frac{u}{2})\cos u \\ y &= (1 + \frac{v}{2}\cos \frac{u}{2})\sin u \\ z &= \frac{v}{2}\sin \frac{u}{2} \end{aligned} \tag{5.2}$$

Where $0 \leq u \leq 2\pi$ and $-1 \leq v \leq 1$. The resulting mesh is shown in figure 5.3

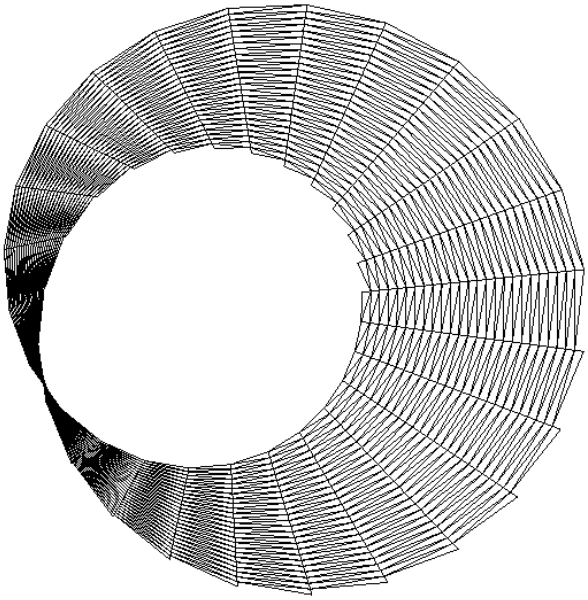


Figure 5.3: Mesh Of A Möbius Strip, Generated From Equation 5.2

5.3.3 Clifford Torus

The Clifford torus is the surface generated by the translation of a circle along another circle, where the two circles are in directly orthogonal planes in $\mathbb{R}^4$. The Clifford torus is the four dimensional analogue of the torus. In Cartesian coordinates, this shape's equation is enlightening due to the reliance of the equation of a circle. $x^2 + y^2 = 1$, $z^2 + w^2 = 1$. Here, w is the fourth dimensional coordinate.

The parametrization of this shape is

$$\begin{aligned} x &= \cos u \\ y &= \sin u \\ z &= \cos v \\ w &= \sin v \end{aligned} \tag{5.3}$$

Where $0 \leq u \leq 2\pi$ and $0 \leq v \leq 2\pi$. The resulting mesh is shown in figure 5.4

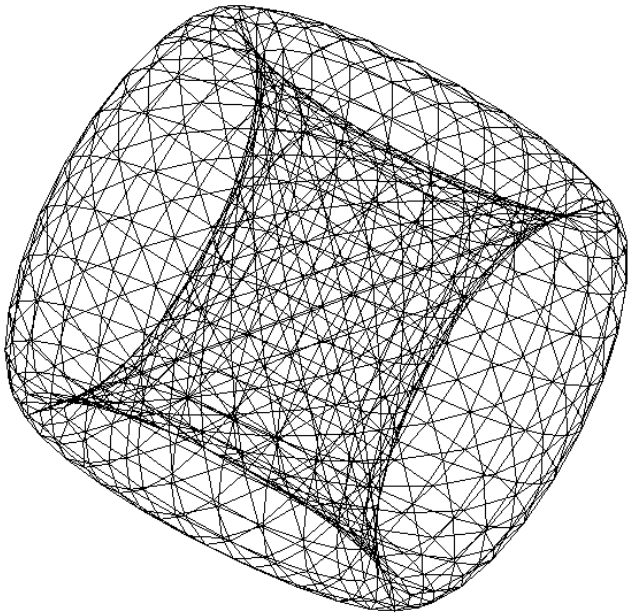


Figure 5.4: Mesh Of A Clifford Torus, Generated From Equation 5.3

```javascript
const _ = require("lodash");

/**
 * Generates all possible combinations of the elements of an array,
 * Returns an array of arrays containing 2 elements.
 *
 * @param {Array} arr - Array to generate combinations of
 */
function combinations(arr) {
  let results = [];

  for (let i = 0; i < arr.length - 1; i++) {
    for (let j = i + 1; j < arr.length; j++) {
      results.push([arr[i], arr[j]]);
    }
  }

  return results;
}

/**
 * @param {Int} n - Number of dimensions
 */
function simplex(n) {
  const s = -1 / (1 + Math.sqrt(1 + n));

  let vertices = [];

  for (let i = 0; i <= n; i++) {
    // Loop once for every vertex, i is the index of the vertex
    let vertex = [];

    if (i === n) {
      for (let j = 0; j < n; j++) {
        vertex.push(s);
      }
    } else {
      for (let j = 0; j < n; j++) {
        // Loop for each coordinate.

        // Check if the coordinate index is the same as the vertex index
        // if so, it's one, otherwise its 0
        vertex.push(j === i ? 1 : 0);
      }
    }
    vertices.push(vertex);
  }

  const edges = combinations(_.range(vertices.length));

  return [vertices.map(homogeneous), edges];
```

Listing 3: Calculate The Vertices And Edges Of A Regular n-Simplex (n-D)

```
function distance(a, b) {
  var sum = 0
  var n
  for (n = 0; n < a.length; n++) {
    sum += Math.pow(a[n] - b[n], 2)
  }
  return Math.sqrt(sum)
}


/**
 * @param {Int} n - Number of dimensions
 */
function hypercube(n) {
  let vertices = [];
  for (let i = 0; i < Math.pow(2, n); i++) {
    let binaryString = i.toString(2); // Get i in binary

    if (binaryString.length < n) {
      // Pad with 0's
      binaryString =
        new Array(n - binaryString.length + 1).join("0") + binaryString;
    }

    vertices.push(
      binaryString.split("").map(x => (x === "0" ? -1 / 2 : 1 / 2))
    );
  }

  const edges = combinations(_.range(vertices.length)).filter(
    x => distance(vertices[x[0]], vertices[x[1]]) === 1
  );

  return [vertices.map(homogeneous), edges];
}
```

Listing 4: Calculate The Vertices And Edges Of A Regular n-Dimensional Hypercube

```javascript
/**
 * @param {Int} n - Number of dimensions
 */
function orthoplex(n) {
  let vertices = [];

  for (let i = 0; i < n; i++) {
    // Loop once for every vertex, i is the index of the vertex
    let vertex1 = [];
    let vertex2 = [];

    for (let j = 0; j < n; j++) {
      vertex1.push(j === i ? 1 : 0);
      vertex2.push(j === i ? -1 : 0);
    }
    vertices.push(vertex1);
    vertices.push(vertex2);
  }

  const edges = combinations(_.range(vertices.length)).filter(
    x => distance(vertices[x[0]], vertices[x[1]]) === Math.sqrt(2)
  );

  return [vertices.map(homogeneous), edges];
}
```

Listing 5: Calculate The Vertices And Edges Of A Regular n-Dimensional Orthoplex

Chapter 6

Conclusion

The previous chapters explored the visualization technique of wireframing in n-dimensions, using generalizations of the standard projections and transformations used in two and three dimensional computer graphics. This technique has allowed the rapid display of higher dimensional objects.

6.1 Research Conclusions

The applications of this research are numerous. The visualization of higher dimensional shapes is applicable to a wide range of problems, such as visualizing n dimensional spline curves, which can be shown through the wireframing method.

One of the most significant applications of these visualization techniques is that of gaining an intuition to how the shapes are structured, and how higher dimensional geometry works. Seeing the shapes visualized aids in the understanding of their structure and the data that they represent.

The methods which we established to visualize the higher dimensional shapes are general and most can be reused for other visualization techniques, if that was researched further.

6.2 Areas For Further Research And Expansion

There is clearly a lot of room for expansion in the field of visualizing higher dimensional geometry.

The concept of visualizing curves in n dimensions has, to a great extent, remained unexplored. The uses are numerous, such as visualizing differential equations of several variables. The visualization of curved surfaces in n dimensions would allow shapes from the realm of algebraic topology to be easily visualized and explored with just a parametric equation.

Another visualization method which could be explored is that of n dimensional RGB (Red Green Blue) fields, where the n-space is divided into blocks like pixels, and each assigned a color. This method could provide different perspectives, which can have the same uses as detailed in the research conclusions.

Bibliography

[Abb84] E. A. Abbott. *Flatland - A Romance of Many Dimensions*. Barnes and Noble Books, Inc., 1884.

[Ban] Thomas F. Banchoff. *History of Thought: Four Dimensional Geometry*. URL: http://www.math.brown.edu/~banchoff/STG/ma8/papers/anogelo/hist4dim.html.

[Ban90] Thomas F Banchoff. *Beyond the third dimension: geometry, computer graphics, and higher dimensions*. Scientific Amer. Libr. New York, NY: Scientific American Library, 1990. ISBN: 9780716750253.

[DB94] Kirk L. Duffin and William A. Barrett. "Spiders: A New User Interface for Rotation and Visualization of N-dimensional Point Sets". In: *Proceedings of the Conference on Visualization '94*. VIS '94. Washinton, D.C.: IEEE Computer Society Press, 1994, pp. 205–211. ISBN: 0-7803-2521-4. URL: http://dl.acm.org/citation.cfm?id=951087.951126.

[Fer17] Robert Ferreol. *Klein bottle*. https://www.mathcurve.com/surfaces.gb/klein/klein.shtml. 2017.

[Han] Andrew J. Hanson. "Geometry for N-Dimensional Graphics". In: ().

[Hol91] Steven Richard Hollasch. "Four-Space Visualization of 4D Objects". MA thesis. Arizona State University, Aug. 1991.

[McC03] Josh McCoy. "High Dimensional Rendering in OpenGL". In: (2003).

[Nol67] A. Michael Noll. "A Computer Technique for Displaying N-dimensional Hyperobjects". In: *Commun. ACM* 10.8 (Aug. 1967), pp. 469–473. ISSN: 0001-0782. DOI: 10.1145/363534.363544. URL: http://doi.acm.org/10.1145/363534.363544.

[Ope] OpenGL. *Perspective Projections: Beyond 3D*. https://ef.gy/linear-algebra:perspective-projections.

[Vin05] John Vince. *Geometry for Computer Graphics: Formulae, Examples and Proofs*. Springer London, 2005.

[Weia] Eric W. Weisstein. *Affine Transformation. From MathWorld—A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/AffineTransformation.html.

[Weib] Eric W. Weisstein. *Cross Product. From MathWorld—A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/CrossProduct.html.

[Weic] Eric W. Weisstein. *Klein Bottle. From MathWorld—A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/KleinBottle.html.

[Weid] Eric W. Weisstein. *Matrix. From MathWorld—A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/Matrix.html.

[Weie] Eric W. Weisstein. *Matrix Inverse. From MathWorld—A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/MatrixInverse.html.

[Weif] Eric W. Weisstein. *Parametric Equations. From MathWorld—A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/ParametricEquations.html.

[Weig] Eric W. Weisstein. *Subspace. From MathWorld—A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/Subspace.html.

[Zho92] Jianhua Zhou. "Visualization of four-dimensional space and its applications". In: (1992).

Appendix A

Source Code For The Visualization Software

Our software is all written in javascript, using the libraries three.js, dat gui, mathjs, and lodash.

The source code is available at `https://github.com/adamisntdead/higher-shapes`, and the finished software is available at `https://adamisntdead.github.io/higher-shapes/`

Appendix B

Correspondences

Correspondence With Cliff Stoll about the triangulation of a Klien Bottle.

Vertices Of Klein Bottles In Four Dimensions

Cliff Stoll - Chief Bottle Washer Sun, Dec 31, 2017 at 1:58 AM
To: Adam Kelly

You touch upon a very interesting mathematical concept.

Triangular meshes lead directly into the mathematical Euler Characteristic.

Start with the simplest mesh possible: a rectangle.

Label the left edge as A, with an arrow pointing up
Label the right edge as A, with an arrow pointing down

Now, if you join A to A, with their arrows matching, you'll get a Mobius loop.

OK, start all over.
New rectangle (second one)

label the top edge as B, with an arrow pointing to the right
label the bottom edge as B, with an arrow pointing to the right.

Now, if you join B to B with the arrows matching, you'll get a cylinder.

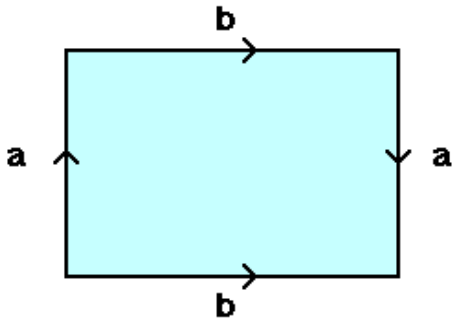
OK, start all over.
New rectangle (3rd one)

Label the left edge as A, with an arrow pointing up
Label the right edge as A, with an arrow pointing up

label the top edge as B, with an arrow pointing to the right
label the bottom edge as B, with an arrow pointing to the right.

Now, join A to A with matching arrows -- you get a cylinder
Join B to B with the arrows matching, the cylinder turns into a Torus.

OK, start all over.
New rectangle (4th one)



Label the left edge as A, with an arrow pointing up
Label the right edge as A, with an arrow pointing down

label the top edge as B, with an arrow pointing to the right
label the bottom edge as B, with an arrow pointing to the right.

Now, join B to B with matching arrows -- you get a cylinder
Join A to A with the arrows matching --- You'll get a Klein bottle!
(But you can't do this in our universe without self-intersection)

OK, so what's next?
Get the last rectangle.
Label its corners (clockwise from the top left) : A, B, C, D (capital letters for corners)
Notice that corner A is the same point as corner D because you've looped it around when making the cylinder.
Notice that corner B is the same point as corner C for the same reason.

Now notice that corner A is the same point as corner C, since you connected those points when making the Kleinbot
And, of course, Corner D is the same point as corner B, for the same reason.
Corners A, B, C, and D are the same point in a Klein bottle.

(substitute the word "Vertex" for "corner"...)

Let's calculate Euler's constant now:

Break up the last rectangle into triangles.
You'll draw one line from the top left to the bottom right.

OK, let's draw a diagonal line from corner A to corner C (so this line goes from a point, clear around the whole Klein bottle, and back to the originating point) Call this diagonal line "c" (lower case c)

That diagonal line (c) separates the rectangle into 2 triangles.

There's one point (A/B/C/D are just one point)

There are 2 triangles visible - two faces.

There are, in all, 3 edges

The Euler characteristic is:
Vertices - edges + faces
1 - 3 + 2
which equal zero.
The Euler characteristic of a Klein bottle is zero.

This is a hint of the connections between topology, geometry, and 3-D printing.

Cheers,
-Cliff
[Quoted text hidden]

Appendix C

Calculating Normal Vectors In $\mathbb{R}^n$

In R^n, where $n > 2$, and with basis vectors $x^1, \ldots, x^n$, let there be $n-1$ linearly independent vectors $V = \{v^1, \ldots, v^{n-1}\}$, the normal vector n of these vectors is

$$n = v_1 \times v_2 \times \cdots \times v_{n-1} = \begin{vmatrix} v_1^1 & v_1^2 & \ldots & v_1^{n-1} & x^1 \\ v_2^1 & v_1^2 & \ldots & v_2^{n-1} & x^2 \\ & & \vdots & & \\ v_n^1 & v_n^2 & \ldots & v_n^{n-1} & x^n \end{vmatrix} \tag{C.1}$$

Where v_b^a is the bth element of the ath vector in V.