Rotating and Displaying 4D Objects

Eugene Y. Q. Shen
0277-075

Extended Essay in Mathematics

Sir Winston Churchill Secondary School

May 2015

Word Count: 3325

Abstract

How can objects in four-dimensional space—4D objects—be rotated and displayed on a 2D screen? This essay explores this research question and details how I programmed software using Python 3 to rotate and display regular polygons, polyhedra, and polychora on a computer display using algorithms that I derived from scratch.

I first derive an algorithm for rotating regular polygons about their centres. I then create an algorithm for rotating regular polyhedra about any axis through their centres and an algorithm for displaying 3D objects on a 2D screen. I finally take these 3D algorithms and adjust them to apply in 4D to rotate and display polychora.

I conclude this essay by considering the limitations of my algorithms. Despite these limitations, these algorithms create a fully functional interactive display system that can rotate and display objects in 4D. This answers my research question: 4D objects can be rotated and displayed on a 2D screen using the algorithms I create in this essay.

Word Count: 162

Notation List

I use the following notations in this essay, which I have adapted according to IB's standard system (International Baccalaureate Organization, 73–77):

$P(x_1, x_2, \dots, x_n)$	the point *P* with Cartesian coordinates $x_1, x_2, \dots, x_n$
$P\{r, \theta\}$	the point *P* with polar coordinates r, θ
$P\{r, \theta, \phi\}$	the point *P* with spherical coordinates r, θ, ϕ
$P\{r, \theta, \phi, \omega\}$	the point *P* with hyperspherical coordinates r, θ, ϕ, ω
(AB)	the line containing points *A* and *B*
AB	the length of the line segment with end points *A* and *B*
$\mathbf{a}$	the vector $\mathbf{a}$
$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$	the column vector representation of $\mathbf{a}$
$\overrightarrow{AB}$	the vector represented in magnitude and direction by the directed line segment from *A* to *B*
$\lvert\mathbf{a}\rvert$	the magnitude of $\mathbf{a}$
$\lvert\overrightarrow{AB}\rvert$	the magnitude of $\overrightarrow{AB}$
$\mathbf{a} \cdot \mathbf{b}$	the scalar product of $\mathbf{a}$ and $\mathbf{b}$
$\mathbf{a} \times \mathbf{b}$	the 3D vector product of $\mathbf{a}$ and $\mathbf{b}$
$\times(\mathbf{a}, \mathbf{b}, \mathbf{c})$	the 4D vector product of $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$
$A \in B$	A is an element of set B
$B \backslash A$	The set of elements in set B not in set A

Table of Contents

Introduction

My research question asks: how can 4D objects be rotated and displayed on a 2D screen? The purpose of this essay is to answer this by deriving algorithms that do so.

The website *Hedrondude's Home Page* inspired me with its mesmerizing renders of 4D objects. I wanted to manipulate these objects myself, but could not find any fitting visualization aids, so I built my own display program. I soon found myself applying my existing subject knowledge in trigonometry, polar coordinates, and vector mathematics to an intricate abstract problem: a process worthy of investigation. Since I worked mainly from first principles without much additional research, all my algorithms are comprehensible at a high school level. This was prime material for an extended essay.

I integrated technology with mathematics to create an interactive product. This union of mathematics and visual technology is vital to modern communication, which increasingly relies on computer-generated graphics for conveying data in graphs, simulations, and maps. Specifically, 4D display software can show mathematical beauty to the public and inspire artists to reach new dimensions. Rotation algorithms also have applications in physical modelling, mechanical engineering, and digital media design. Although these applications mainly use more advanced techniques such as rotation matrices, quaternions, and projective geometry, it is still useful to have an elementary interpretation of rotation and display, which this essay gives. My algorithms are limited; however, within their limitations, they give the same results as other techniques.

Python 3

I choose Python 3 and its built-in module Tkinter to program the display system because of their simplicity. Python 3 includes a function $\operatorname{atan2}$ that takes two inputs y and x and outputs a value θ such that $\tan\theta = y/x$ with the output between $-\pi$ and π (Python Software Foundation sec. 9.2.3). I use $\operatorname{atan2}$ and omit all quadrant conversions in this essay because unlike the standard arctangent function, whose range is between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, $\operatorname{atan2}$ outputs angles in the correct quadrant.

2D Rotations

I begin my investigation by rotating *polygons*. A *p-gon* is “a circuit of *p* line-segments $A_1A_2, A_2A_3, \dots, A_pA_1$ joining consecutive pairs of *p* points $A_1, A_2, \dots, A_p$”, and these line-segments and points are respectively *sides* and *vertices* (singular: *vertex*), so that a p-gon is a polygon with *p* vertices (Coxeter 1). A *regular* polygon is a polygon whose sides are of equal length and whose angles, between each consecutive pair of sides, are of equal size. Coxeter notes that all regular polygons have a *centre*, such that all of the polygon’s vertices’ distance to the centre is the same *circumradius* (2). If this centre is the origin $O = O(0,0)$, then polar coordinates are more suited to describe regular polygons than Cartesian coordinates are. If point P has the polar coordinates $P\{r, \theta\}$, then $r = \left|\overrightarrow{OP}\right|$ and θ is the angle that $\overrightarrow{OP}$ makes with the x-axis (Martin 487).

Since the vertices of a regular polygon are the same distance from the centre, polar coordinates represent rotations well. Polar and other non-Cartesian coordinates also represent axes well. However, I represent all vertex data in Cartesian form, which fits more easily with my 3D and 4D rotation algorithms. So, I must derive conversions from non-Cartesian coordinates into Cartesian coordinates. Note on figure 1 that:

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \operatorname{atan2}(y, x)$$

So the conversion formula from Cartesian to polar coordinates is:

$$P(x, y) \xrightarrow{P} P\left\{\sqrt{x^2 + y^2}, \operatorname{atan2}(y, x)\right\} \qquad [3.1]$$

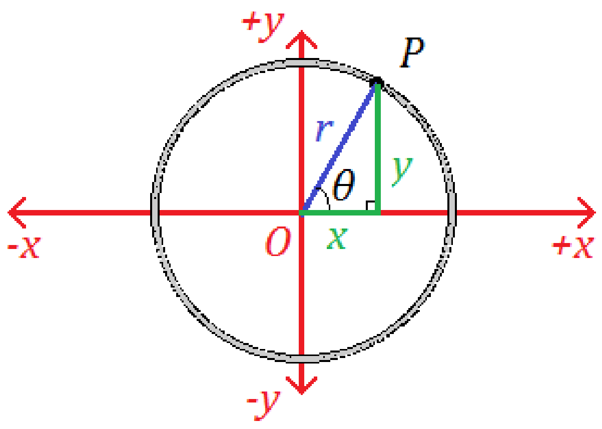

Figure 1. Polar coordinates.

To convert polar coordinates to Cartesian coordinates, solve for x and y:

$$y = x \tan\theta$$

$$r = \sqrt{x^2 + (x\tan\theta)^2} = \sqrt{x^2(1 + \tan^2\theta)} = \sqrt{x^2 \sec^2\theta} = x\sec\theta$$

$$x = r\cos\theta$$

$$y = x\tan\theta = r\cos\theta\tan\theta = r\sin\theta$$

So the conversion formula from polar to Cartesian coordinates is:

$$P\{r,\theta\} \overset{C}{\rightarrow} P(r\cos\theta, r\sin\theta) \qquad [3.2]$$

This displays polygons on the computer screen, which uses Cartesian coordinates. I elaborate more about Python 3 and Tkinter's coordinate system in Appendix D.

Rotations are transformations that fix certain points and preserve the distance between and orientation of all objects rotated (Manning 141). A 2D rotation fixes a *rotation point*. To rotate a polygon, I need only rotate its vertices, since rotations preserve the distance between all points, so the length of each side does not change after rotation. The formula for rotation about the origin is simple, but those for rotations about arbitrary points are not. Therefore, in this essay, I only consider rotations that fix the origin. Note on figure 2 the polar coordinates of P and its image P' after rotation by ω about O:

$$P\{r,\theta\} \overset{R}{\rightarrow} P'\{r,\theta+\omega\} \qquad [3.3]$$

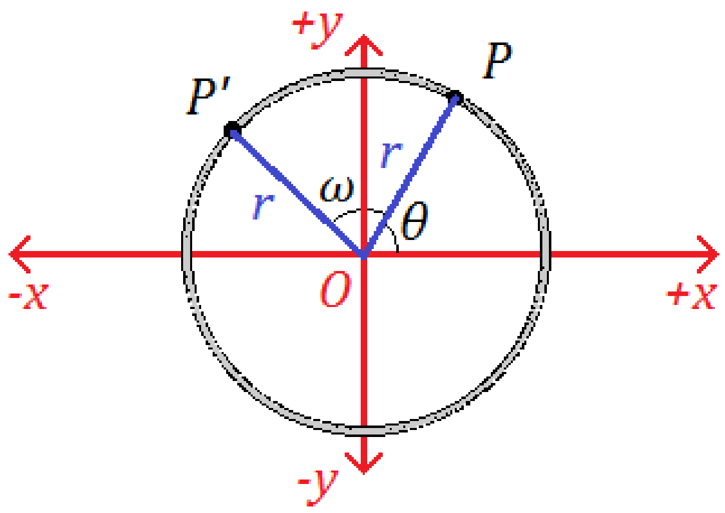

Figure 2. 2D rotation about the origin.

However, I cannot generalize this method to 3D.

Spherical Coordinates

Coxeter defines a *polyhedron* (plural: *polyhedra*) as a finite, connected set of polygons, each of whose sides are shared with only one other polygon (4). Regular polyhedra have a circumradius and a centre, which I set as the origin O (5).

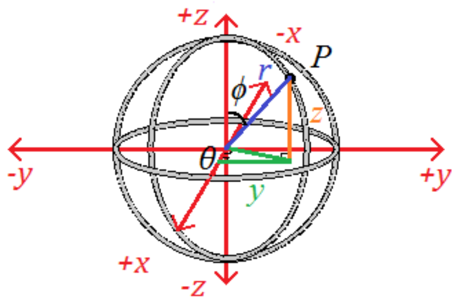

Figure 3. Spherical coordinates.

This circumradius allows spherical coordinates to describe regular polyhedra well. If point P has the spherical coordinates $P\{r, \theta, \phi\}$, then $r = |\overrightarrow{OP}|$, θ is the angle that the orthogonal projection of $\overrightarrow{OP}$ on the xy-plane makes with the x-axis, and ϕ is the angle that $\overrightarrow{OP}$ makes with the z-axis (Weisstein). Weisstein restricts ϕ to:

$$0 \leq \phi \leq \pi \qquad [4.1]$$

To convert Cartesian coordinates to spherical coordinates, note on figure 3 or 4 that:

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\theta = \operatorname{atan2}(y, x)$$

$$\phi = \arccos(z/r)$$

So the conversion formula from Cartesian to spherical coordinates is:

$$P(x, y, z) \overset{S}{\rightarrow} P\left\{\sqrt{x^2 + y^2 + z^2}, \arccos\left(\frac{z}{\sqrt{x^2 + y^2}}\right), \operatorname{atan2}(y, x)\right\} \qquad [4.2]$$

This formula works because atan2 returns θ in the correct quadrant and because the range of Python 3's arccos is the domain of ϕ, as restricted in [4.1].

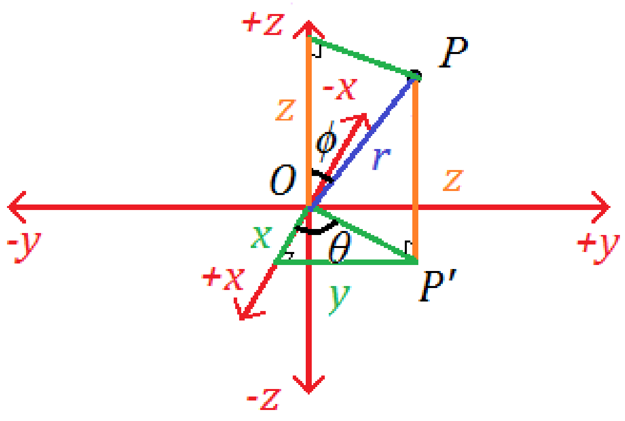

Figure 4. Conversion between Cartesian and spherical coordinates.

To convert spherical coordinates to Cartesian coordinates, let $\overrightarrow{OP'}$ be the orthogonal projection of $\overrightarrow{OP}$ on the xy-plane. Then the z-component of $\overrightarrow{OP'}$ is 0 because the normal equation of the xy-plane is $z = 0$:

$$\overrightarrow{OP} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \overrightarrow{OP'} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

Solve for x, y, and z by noting on figure 4 that:

$$z = r\sin\left(\frac{\pi}{2} - \phi\right) = r\cos\phi$$

$$\overrightarrow{OP'} = r\cos\left(\frac{\pi}{2} - \phi\right) = r\sin\phi$$

$$x = \overrightarrow{OP'}\cos\theta = r\sin\phi\cos\theta$$

$$y = \overrightarrow{OP'}\sin\theta = r\sin\phi\sin\theta$$

So the conversion formula from spherical to Cartesian coordinates is:

$$P\{r, \theta, \phi\} \overset{C}{\rightarrow} P(r\sin\phi\cos\theta, r\sin\phi\sin\theta, r\cos\phi) \qquad [4.3]$$

3D Rotations

To rotate a polyhedron, I need only rotate all its vertices, because a polyhedron is a collection of polygons. To rotate a 3D object about a fixed *rotation axis* by an angle ω, I require for simplicity that the axis pass through the origin. Then, given a point $A\{r_A, \theta_A, \phi_A\}$, the rotation axis is a line (OA), the *axis-line*, with parametric equation:

$$\overrightarrow{OA} = \frac{t}{r_A}\overrightarrow{OA}, t \in \mathbb{R} \qquad [6.1]$$

Consider the rotation of point $P(x, y, z)$ about an axis-line in figure 5. P rotates on a plane that the axis-line is normal to, the *rotation plane*, about the intersection I of the axis-line and the rotation plane. This is a rotation because the distance between points on the same rotation plane does not change after a 2D rotation, and the distance between rotation planes does not change because the axis-line is normal to every rotation plane.

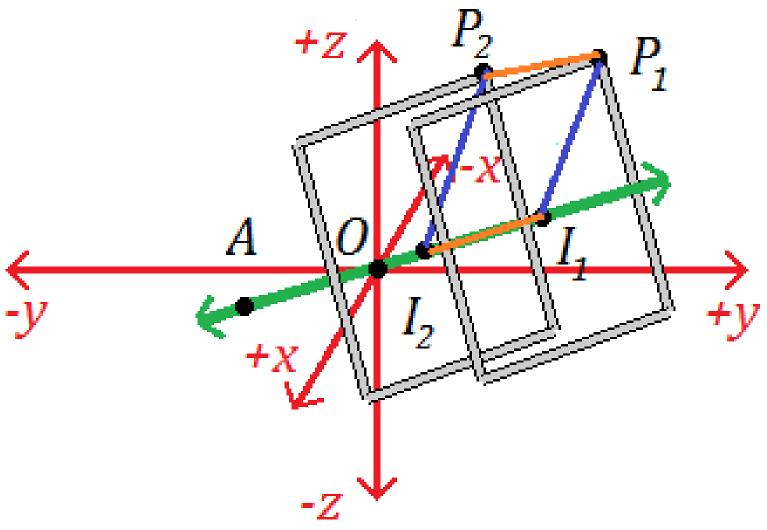

Figure 5. 3D rotation about an axis-line.

Given a point $P(x,y,z)$, the normal equation of its rotation plane is:

$$\overrightarrow{OA} \cdot P = d \tag{6.2}$$

Find I using [6.1] and [6.2]:

$$I = \frac{t}{r_A}\overrightarrow{OA}$$
$$\overrightarrow{OA} \cdot I = \overrightarrow{OA} \cdot \frac{t}{r_A}\overrightarrow{OA} = d$$
$$\frac{t}{r_A}\overrightarrow{OA} \cdot \overrightarrow{OA} = \frac{t}{r_A}\left|\overrightarrow{OA}\right|^2 = \frac{t}{r_A}{r_A}^2 = d$$
$$t = \frac{d}{r_A} \tag{6.3}$$

To apply [3.3] on the rotation plane, set (IP) as the "x-axis" and identify a line (IQ) on the rotation plane perpendicular to $\overrightarrow{IP}$ as the "y-axis". If $\overrightarrow{IQ}$ is on the rotation plane, then it is perpendicular to $\overrightarrow{OA}$, so use the 3D vector product:

$$\overrightarrow{IQ}_{raw} = \overrightarrow{OA} \times \overrightarrow{IP} \tag{6.4}$$

Normalize $\overrightarrow{IQ}_{raw}$ so that one unit on $\overrightarrow{IQ}$ is the same as one unit on $\overrightarrow{IP}$:

$$\overrightarrow{IQ} = \frac{\left|\overrightarrow{IP}\right|}{\left|\overrightarrow{IQ}\right|}\overrightarrow{IQ}_{raw} \tag{6.5}$$

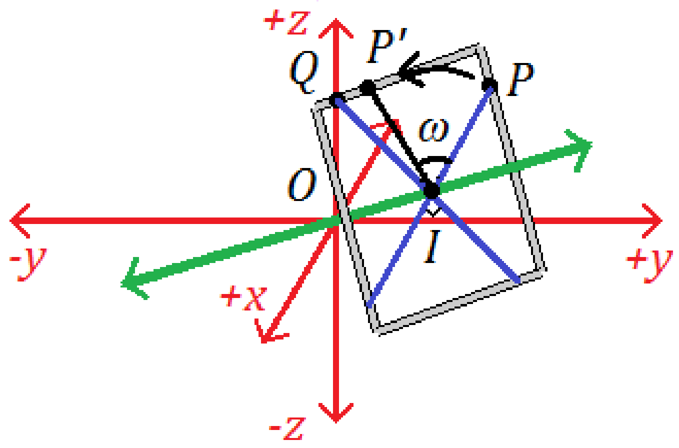

Figure 6. 3D rotation on the rotation plane.

Now apply [3.3] to the rotation plane, as in figure 6, by setting I as the centre of the rotation plane. (IP) is the "x-axis", so the angle of $\overrightarrow{IP}$ is 0 and its magnitude is $\left|\overrightarrow{IP}\right|$:

$$\overrightarrow{IP}\left\{\left|\overrightarrow{IP}\right|, 0\right\} \overset{R}{\rightarrow} \overrightarrow{IP'}\left\{\left|\overrightarrow{IP}\right|, 0+\omega\right\} \qquad [6.6]$$

Convert [6.6] into Cartesian coordinates using [3.1]:

$$\overrightarrow{IP'}\left\{\left|\overrightarrow{IP}\right|, \omega\right\} \overset{C}{\rightarrow} \overrightarrow{IP'}\left(\left|\overrightarrow{IP}\right|\cos\omega, \left|\overrightarrow{IP}\right|\sin\omega\right) \qquad [6.7]$$

[6.7] specifies $\overrightarrow{IP'}$ in terms of $\overrightarrow{IP}$ and $\overrightarrow{IQ}$, which in x- and y-coordinates are:

$$\overrightarrow{IP}\left(\left|\overrightarrow{IP}\right|, 0\right) \overset{R}{\rightarrow} \overrightarrow{IP'}\left(\left|\overrightarrow{IP}\right|\cos\omega\,\overrightarrow{IP}, \left|\overrightarrow{IP}\right|\sin\omega\,\overrightarrow{IQ}\right) \qquad [6.8]$$

Finally, add the coordinates of I to [6.8] because:

$$\overrightarrow{IP} + \overrightarrow{OI} = \overrightarrow{OP}$$

So the general 3D rotation formula is:

$$P(x,y,z) \overset{R}{\rightarrow} P'\left(\left|\overrightarrow{IP}\right|\cos\omega\,\overrightarrow{IP} + \overrightarrow{OI}, \left|\overrightarrow{IP}\right|\sin\omega\,\overrightarrow{IQ} + \overrightarrow{OI}\right) \qquad [6.9]$$

3D Projections

In 2D, all points are on one plane, so I display objects directly on the computer screen. However, in 3D, not all points are on one plane, so I need a *projection algorithm*, an algorithm that displays 3D coordinates on a 2D plane. Consider figure 7 and imagine a *viewing point* V. Its position vector is normal to a plane in space, the *picture plane*, which acts as the computer screen. For analogy, the viewing point is an observer's pupil and the picture plane is the observer's retina. Let the picture plane be a fixed distance f away from V, which corresponds to the fixed width of the observer's eye. The picture plane is further away from the origin than V is, so that if the viewing point moves away from the origin, then objects projected on the picture plane become smaller, as in real life. Extend lines from every point through V and find the intersections of these lines with the picture plane. These lines correspond to rays of light that pass through the observer's pupil and hit the observer's retina. Finally, display these intersections on the computer screen.

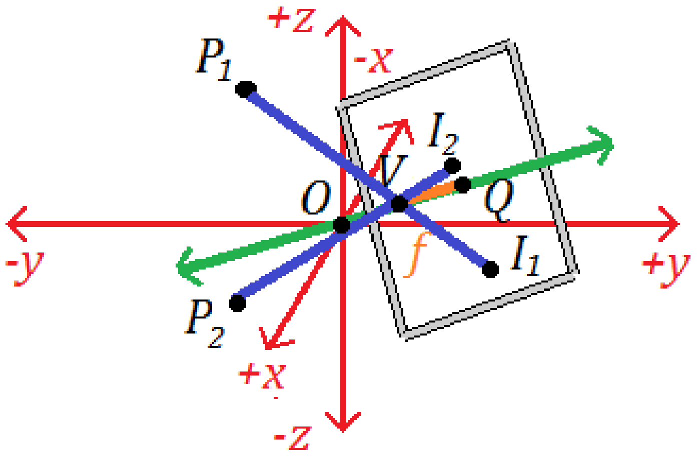

Figure 7. 3D projection on the picture plane.

Given the point V and the picture plane f away from V and normal to $\overrightarrow{OV}$:

$$V = V\{r, \theta, \phi\} \tag{7.1}$$

Let Q be the intersection between $\overrightarrow{OV}$ and the picture plane:

$$Q = \left(1 + \frac{f}{r}\right)\overrightarrow{OV} = Q\{r + f, \theta, \phi\} \tag{7.2}$$

The normal equation of the picture plane is then, by [7.2]:

$$Q \cdot \overrightarrow{OV} = \left(1 + \frac{f}{r}\right)\overrightarrow{OV} \cdot \overrightarrow{OV} = d$$

$$d = \left(1 + \frac{f}{r}\right)\left|\overrightarrow{OV}\right|^2 = (r + f)r \tag{7.3}$$

Given a point to be rotated $P(x, y, z)$, the parametric equation of the line (VP) is:

$$P + \left(\overrightarrow{OV} - P\right)t, t \in \mathbb{R} \qquad [7.4]$$

The intersection I of this line with the picture plane is, by [7.3] and [7.4]:

$$I \cdot \overrightarrow{OV} = d \qquad [7.5]$$

$$\left(P + \left(\overrightarrow{OV} - P\right)t\right) \cdot \overrightarrow{OV} = (r + f)r$$

$$(\overrightarrow{OV} \cdot \overrightarrow{OV} - P \cdot \overrightarrow{OV})t = (r + f)r - P \cdot \overrightarrow{OV}$$

$$t = \frac{(r + f)r - P \cdot \overrightarrow{OV}}{r^2 - P \cdot \overrightarrow{OV}} \qquad [7.6]$$

Now choose two perpendicular vectors $\mathbf{w}$ and $\mathbf{h}$ on the picture plane, so that their coefficients m and n respectively represent the x- and y=coordinates of the computer screen. To restrict $\mathbf{w}$ and $\mathbf{h}$ to allow a deterministic solution, let $\mathbf{w}$ be perpendicular to the z-axis, so that all x-coordinates are strictly horizontal. Since $\mathbf{w}$ is on the picture plane, $\mathbf{w}$ is also perpendicular to $\overrightarrow{OV}$, so apply the 3D vector product:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} r \sin\phi \cos\theta \\ r \sin\phi \sin\theta \\ r \cos\phi \end{bmatrix} = \mathbf{w}_{raw} = \begin{bmatrix} -r \sin\phi \sin\theta \\ r \sin\phi \cos\theta \\ 0 \end{bmatrix}$$

Normalize $\mathbf{w}_{raw}$ so that one unit on $\mathbf{w}$ is one unit:

$$\mathbf{w} = \frac{1}{|\mathbf{w}|}\mathbf{w}_{raw} = \frac{1}{\sqrt{(-r \sin\phi \sin\theta)^2 + (r \sin\phi \cos\theta)^2}}\mathbf{w}_{raw} = \frac{1}{r \sin\phi}\mathbf{w}_{raw}$$

$$\mathbf{w} = \begin{bmatrix} -\sin\theta \\ \cos\theta \\ 0 \end{bmatrix} \qquad [7.7]$$

Now $\mathbf{h}$ is perpendicular to both $\mathbf{w}$ and $\overrightarrow{OV}$, so apply the 3D vector product:

$$\begin{bmatrix} -\sin\theta \\ \cos\theta \\ 0 \end{bmatrix} \times \begin{bmatrix} r\sin\phi\cos\theta \\ r\sin\phi\sin\theta \\ r\cos\phi \end{bmatrix} = \mathbf{h}_{raw} = \begin{bmatrix} r\cos\phi\cos\theta \\ r\cos\phi\sin\theta \\ -r\sin\phi \end{bmatrix}$$

Normalize $\mathbf{h}_{raw}$ so that one unit on $\mathbf{h}$ is one unit:

$$\mathbf{h} = \frac{1}{|\mathbf{h}|}\mathbf{h}_{raw} = \frac{1}{\sqrt{(r\cos\phi\cos\theta)^2 + (r\cos\phi\sin\theta)^2 + (-r\sin\phi)^2}}\mathbf{h}_{raw} = \frac{1}{r}\mathbf{h}_{raw}$$

$$\mathbf{h} = \begin{bmatrix} \cos\phi\cos\theta \\ \cos\phi\sin\theta \\ -\sin\phi \end{bmatrix} \qquad [7.8]$$

Let Q be the centre of the picture plane, and thus also the centre of the computer screen. To find the coefficients m and n of $\mathbf{w}$ and $\mathbf{h}$, express the difference between I and Q in terms of $\mathbf{w}$ and $\mathbf{h}$, by using equations [7.4], [7.5], [7.6], [7.2], [7.7], and [7.8]:

$$I - Q = m\mathbf{w} + n\mathbf{h} \qquad [7.9]$$

$$P + \left(\overrightarrow{OV} - P\right)t - \left(1 + \frac{f}{r_V}\right)\overrightarrow{OV} = m\mathbf{w} + n\mathbf{h}$$

$$(1-t)\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \left(t - 1 - \frac{f}{r}\right)\frac{r}{r}\begin{bmatrix} r\sin\phi\cos\theta \\ r\sin\phi\sin\theta \\ r\cos\phi \end{bmatrix} = m\begin{bmatrix} -\sin\theta \\ \cos\theta \\ 0 \end{bmatrix} + n\begin{bmatrix} \cos\phi\cos\theta \\ \cos\phi\sin\theta \\ -\sin\phi \end{bmatrix}$$

$$(1-t)x + (tr - r - f)\sin\phi\cos\theta = -\mathrm{m}\sin\theta + \mathrm{n}\cos\phi\cos\theta \qquad [7.10]$$

$$(1-t)y + (tr - r - f)\sin\phi\sin\theta = \mathrm{m}\cos\theta + \mathrm{n}\cos\phi\sin\theta \qquad [7.11]$$

$$(1-t)z + (tr - r - f)\cos\phi = -n\sin\phi \qquad [7.12]$$

If $\sin\phi \neq 0$, then by [7.12]:

$$n = \frac{(1-t)z + (tr - r - f)\cos\phi}{-\sin\phi} \qquad [7.13]$$

If $\sin\theta = 0$, then $\cos\theta = \pm 1 = \sec\theta$, and by [7.11]:

$$m\cos\theta = (1-t)y + (tr - r - f)\sin\phi\sin\theta$$

$$m = (1-t)y\cos\theta \qquad [7.14]$$

If $\sin\theta \neq 0$, then by [7.10] and [7.13]:

$$m = \frac{(1-t)x + (tr - r - f)\sin\phi\cos\theta - n\cos\phi\cos\theta}{-\sin\theta} \qquad [7.15]$$

If $\sin\phi = 0$, then $\cos\phi = 1 = \sec\phi$, because $0 \leq \phi \leq \pi$.

If $\sin\theta = 0$, then $\cos\theta = \pm 1 = \sec\theta$, and by [7.10] and [7.11]:

$$m\cos\theta = (1-t)y + (tr - r - f)\sin\phi\sin\theta - n\cos\phi\sin\theta$$

$$m = (1-t)y\cos\theta \qquad [7.16]$$

$$n\cos\phi\cos\theta = (1-t)x + (tr - r - f)\sin\phi\cos\theta + m\sin\theta$$

$$n = (1-t)x\cos\phi\cos\theta \qquad [7.17]$$

If $\sin\theta \neq 0$, then by [7.10]:

$$-\mathrm{m}\sin\theta = (1-t)x + (tr - r - f)\sin\phi\cos\theta - n\cos\phi\cos\theta$$

$$m = \frac{n\cos\phi\cos\theta - (1-t)x}{\sin\theta} \qquad [7.18]$$

By [7.11] and [7.18]:

$$n\cos\phi\sin\theta = (1-t)y + (tr - r - f)\sin\phi\sin\theta - m\cos\theta$$

$$n\cos\phi\frac{\sin^2\theta}{\sin\theta} = (1-t)y - \frac{n\cos\phi\cos\theta - (tr - r - f)x}{\sin\theta}\cos\theta$$

$$n\left(\cos\phi\frac{\sin^2\theta}{\sin\theta} + \cos\phi\frac{\cos^2\theta}{\sin\theta}\right) = (1-t)y + (1-t)x\cot\theta$$

$$n = (1-t)(y + x\cot\theta)\frac{\sin\theta}{\cos\phi} \qquad [7.19]$$

The restrictions in each of these equations ensure that no zero division errors occur.

So the general 3D projection formula is:

$$P(x,y,z) \overset{V}{\rightarrow} P(m,n) \qquad [7.20]$$

Hyperspherical Coordinates

The 4D analogues to 2D polygons and 3D polyhedra are *polychora*, which are a finite, connected set of polyhedra (Bowers). Regular polychora have a centre, which I set as the origin O, and a circumradius, so a 4D analogue of spherical coordinates works well.

In spherical coordinates, both z and r', the magnitude of the orthogonal projection of $\overrightarrow{OP}$ on the xy-plane, depend solely on ϕ. ϕ sets the height z at which the sphere is cut to show a circular cross-section where x and y are found. In 4D, consider figure 8 and imagine an angle ψ that sets the height w at which the 4D sphere is cut to show a spherical cross-section where x, y, and z are found.

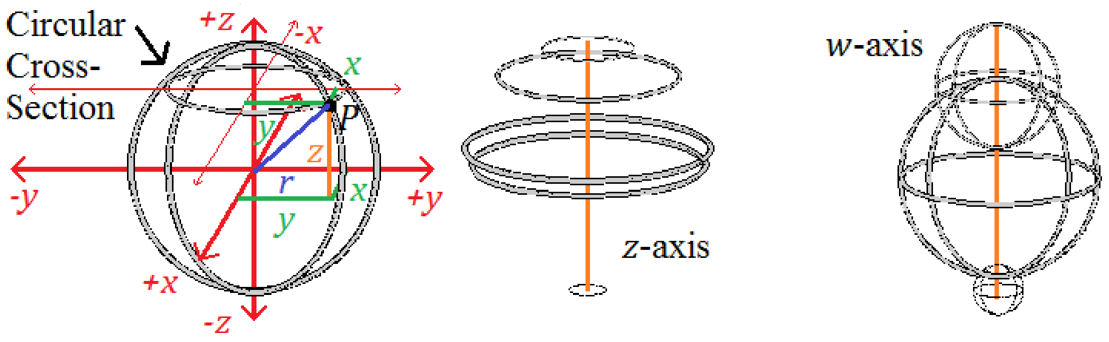

Figure 8. Hyperspherical coordinates.

I call this 4D analogue *hyperspherical coordinates*. If point P has the hyperspherical coordinates $P\{r, \theta, \phi, \psi\}$, then $r = |\overrightarrow{OP}|$, θ is the angle that the orthogonal projection of $\overrightarrow{OP}$ on the xy-plane makes with the x-axis, and ψ is the angle that $\overrightarrow{OP}$ makes with the w-axis. ϕ is the angle that the orthogonal projection of $\overrightarrow{OP}$ on the xyz-hyperplane makes with the z-axis, where a *hyperplane* “consists of the points that we get if we take four planes not points of one plane, all points collinear with any two obtained by this process” (Manning 24). For example, everyday space is a hyperplane, and a spherical cross-section of a 4D sphere is also a hyperplane. I limit θ and ϕ to the xyz-hyperplane because w neither depends on θ nor ϕ, just as Weisstein limits θ to the xy-plane because z does not depend on θ. With similar reasons to why Weisstein restricts ϕ in [4.1], I also restrict ψ:

$$0 \leq \psi \leq \pi \qquad [8.1]$$

To convert hyperspherical coordinates to Cartesian coordinates, use its definition:

$$r = \left|\overrightarrow{OP}\right| = \sqrt{x^2 + y^2 + z^2 + w^2}$$

Consider figure 9. The first diagram shows how to find a spherical cross-section at height w, which has radius r'. The second diagram shows how to find the coordinates of that spherical cross-section. Note that:

$$r' = \sqrt{x^2 + y^2 + z^2}$$

$$\theta = \operatorname{atan2}(y, x)$$

$$\phi = \arccos(z/r')$$

$$\psi = \arccos(w/r)$$

So the conversion formula from Cartesian to hyperspherical coordinates is:

$$P(x, y, z, w) \overset{H}{\rightarrow} P\left\{\sqrt{x^2+y^2+z^2+w^2}, \operatorname{atan2}(y, x),\right.$$

$$\left.\arccos\left(\frac{z}{\sqrt{x^2+y^2+z^2}}\right), \arccos\left(\frac{w}{\sqrt{x^2+y^2+z^2+w^2}}\right)\right\} \qquad [8.2]$$

As before, this formula works because of the restrictions in [4.1] and [8.1].

To convert hyperspherical coordinates to Cartesian coordinates, let $\overrightarrow{OP'}$ be the orthogonal projection of $\overrightarrow{OP}$ on the xyz-hyperplane. Then the w-component of $\overrightarrow{OP'}$ is 0 because the normal equation of the xyz-hyperplane is $w = 0$.

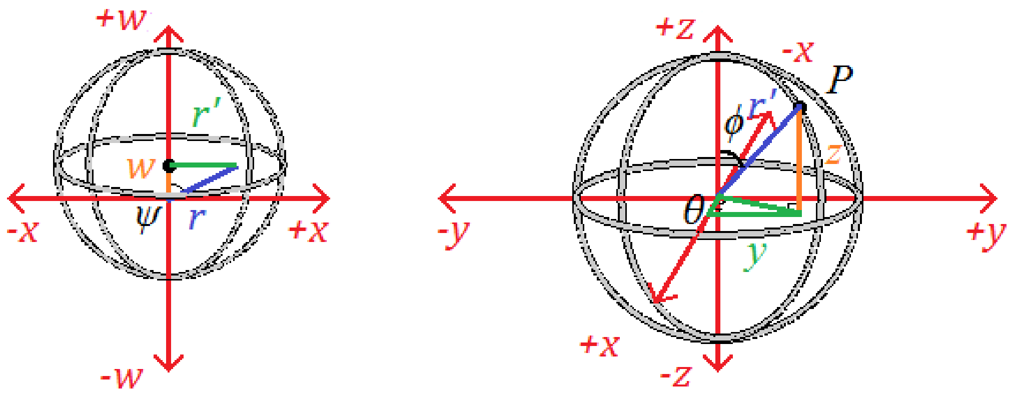

Figure 9. Conversion between Cartesian and hyperspherical coordinates.

Solve for x, y, z, and w by noting on figure 9 that:

$$w = r\sin\left(\frac{\pi}{2} - \psi\right) = r\cos\psi$$

$$r' = \sqrt{x^2+y^2+z^2} = r\cos\left(\frac{\pi}{2} - \psi\right) = r\sin\psi$$

This specifies a sphere in the xyz-hyperplane with radius r', so apply [4.3]:

$$P\{r', \theta, \phi\} \overset{C}{\rightarrow} P(r'\sin\phi\cos\theta, r'\sin\phi\sin\theta, r'\cos\phi)$$

$$x = r'\sin\phi\cos\theta = r\sin\psi\sin\phi\cos\theta$$

$$y = r'\sin\phi\sin\theta = r\sin\psi\sin\phi\sin\theta$$

$$z = r' \cos\phi = r \sin\psi \cos\phi$$

So the conversion formula from hyperspherical to Cartesian coordinates is:

$$P\{r, \theta, \phi, \psi\} \overset{C}{\rightarrow} P(r \sin\psi \sin\phi \cos\theta, r \sin\psi \sin\phi \sin\theta, r \sin\psi \cos\phi, r \cos\psi) \quad [8.3]$$

4D Rotations

To rotate a polychoron, I need only rotate all of its vertices, because a polychoron is a collection of polyhedra. In 3D, an axis-line is normal to a set of rotation planes, as in figure 6, and a rotation rotates each rotation plane about its intersection with the axis-line.

However, in 4D, an axis-line is normal to a set of hyperplanes, so an axis-line cannot determine a unique rotation, and a 4D rotation needs an *axis-plane* (Manning 143). This axis-plane has a set of *absolutely perpendicular planes*, such that every line on the axis-plane is perpendicular to every line on each absolutely perpendicular plane (Manning 80). Each absolutely perpendicular plane rotates about its intersection with the axis-plane, as in figure 10. This is a rotation, as the distance between points on the same absolutely perpendicular plane does not change after a 2D rotation, and the distance between the absolutely perpendicular planes does not change because the axis-line is normal to every absolutely perpendicular plane (Manning 143). These absolutely perpendicular planes are 4D generalizations of the 3D rotation plane, so I call them *rotation planes* as well.

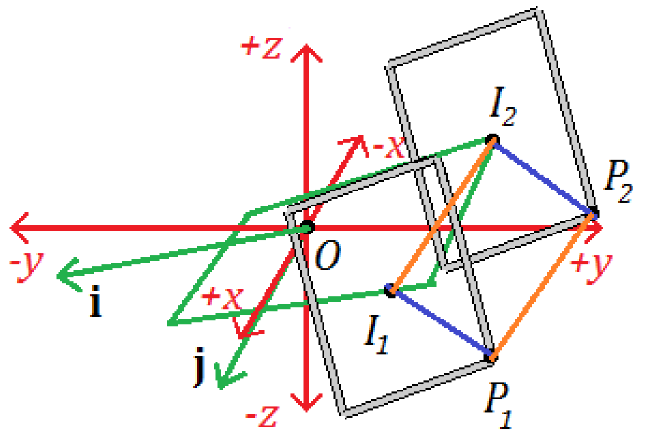

Figure 10. 4D rotation about an axis-plane.

Given a point $P(x, y, z, w)$ to be rotated and two vectors **i** and **j**, let the axis-plane be defined by the origin O and vectors **i** and **j**, and the rotation plane of P be defined by point P and unknown vectors **k** and **l**. Then **i** and **j** must each be perpendicular to **k** and **l** because every line on the axis-plane is perpendicular to every line on the rotation plane of P. Let **i** and **j** be perpendicular unit vectors and choose **k** and **l** such that they are also perpendicular unit vectors. Then:

$$\mathbf{i} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \end{bmatrix}$$

$$\mathbf{j} = \begin{bmatrix} j_1 \\ j_2 \\ j_3 \\ j_4 \end{bmatrix}$$

$$|\mathbf{i}| = |\mathbf{j}| = |\mathbf{k}| = |\mathbf{l}| = 1$$

$$\mathbf{i} \cdot \mathbf{j} = \mathbf{i} \cdot \mathbf{k} = \mathbf{i} \cdot \mathbf{l} = \mathbf{j} \cdot \mathbf{k} = \mathbf{j} \cdot \mathbf{l} = \mathbf{k} \cdot \mathbf{l} = 0 \qquad [9.1]$$

To find **k**, apply the 3D vector product on three components of **i** and **j**, not all of which are zero. If all three components of either **i** or **j** are zero, then the 3D vector product outputs the zero vector, which cannot become a unit vector by multiplication.

If $\sqrt{{i_1}^2 + {i_2}^2 + {i_3}^2} > 0$ and $\sqrt{{j_1}^2 + {j_2}^2 + {j_3}^2} > 0$:

$$\mathbf{k}_{raw} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} \times \begin{bmatrix} j_1 \\ j_2 \\ j_3 \end{bmatrix} = \begin{bmatrix} i_2 j_3 - i_3 j_2 \\ i_3 j_1 - i_1 j_3 \\ i_1 j_2 - i_2 j_1 \end{bmatrix}$$

$$\mathbf{k}_{raw} = \begin{bmatrix} i_2 j_3 - i_3 j_2 \\ i_3 j_1 - i_1 j_3 \\ i_1 j_2 - i_2 j_1 \\ 0 \end{bmatrix}$$

There are three more cases, if $\sqrt{{i_1}^2 + {i_2}^2 + {i_3}^2} = 0$ or $\sqrt{{j_1}^2 + {j_2}^2 + {j_3}^2} = 0$:

If $\sqrt{{i_1}^2 + {i_2}^2 + {i_4}^2} > 0$ and $\sqrt{{j_1}^2 + {j_2}^2 + {j_4}^2} > 0$, $\mathbf{k}_{raw} = \begin{bmatrix} i_2 j_4 - i_4 j_2 \\ i_4 j_1 - i_1 j_4 \\ 0 \\ i_1 j_2 - i_2 j_1 \end{bmatrix}$

If $\sqrt{{i_1}^2 + {i_3}^2 + {i_4}^2} > 0$ and $\sqrt{{j_1}^2 + {j_3}^2 + {j_4}^2} > 0$, $\mathbf{k}_{raw} = \begin{bmatrix} i_3 j_4 - i_4 j_3 \\ 0 \\ i_4 j_1 - i_1 j_4 \\ i_1 j_3 - i_3 j_1 \end{bmatrix}$

If $\sqrt{{i_2}^2 + {i_3}^2 + {i_4}^2} > 0$ and $\sqrt{{j_2}^2 + {j_3}^2 + {j_4}^2} > 0$, $\mathbf{k}_{raw} = \begin{bmatrix} 0 \\ i_3 j_4 - i_4 j_3 \\ i_4 j_2 - i_2 j_4 \\ i_2 j_3 - i_3 j_2 \end{bmatrix}$

Normalize $\mathbf{k}_{raw}$ so that $\mathbf{k}$ is a unit vector:

$$\mathbf{k} = \frac{1}{|\mathbf{k}_{raw}|} \mathbf{k}_{raw}$$

Now with vectors $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$, apply the 4D vector product to determine a vector perpendicular to all three. My derivation of the 4D vector product is in Appendix B.

$$\mathbf{l} = \times(\mathbf{i}, \mathbf{j}, \mathbf{k}) = \begin{bmatrix} i_2(j_3 k_4 - j_4 k_3) - i_3(j_2 k_4 - j_4 k_2) + i_4(j_2 k_3 - j_3 k_2) \\ -i_1(j_3 k_4 - j_4 k_3) + i_3(j_1 k_4 - j_4 k_1) - i_4(j_1 k_3 - j_3 k_1) \\ i_1(j_2 k_4 - j_4 k_2) - i_2(j_1 k_4 - j_4 k_1) + i_4(j_1 k_2 - j_2 k_1) \\ -i_1(j_2 k_3 - j_3 k_2) + i_2(j_1 k_3 - j_3 k_1) - i_3(j_1 k_2 - j_2 k_1) \end{bmatrix}$$

Normalize $\mathbf{l}_{raw}$ so that $\mathbf{l}$ is a unit vector:

$$\mathbf{l} = \frac{1}{|\mathbf{l}_{raw}|}\mathbf{l}_{raw}$$

The axis-plane is the intersection of the two hyperplanes that contain O that $\mathbf{k}$ and $\mathbf{l}$ are normal to. The rotation plane of P is the intersection of the two hyperplanes that contain P that $\mathbf{i}$ and $\mathbf{j}$ are normal to. The equations of these hyperplanes are:

$$P \cdot \mathbf{i} = d_i$$

$$P \cdot \mathbf{j} = d_j$$

$$O \cdot \mathbf{k} = O \cdot \mathbf{l} = 0$$

Then the intersection I between the axis-plane and the rotation plane of P is the intersection of all four hyperplanes, which is the solution to the system of equations:

$$I \cdot \mathbf{i} = I_1 i_1 + I_2 i_2 + I_3 i_3 + I_4 i_4 = d_i$$

$$I \cdot \mathbf{j} = I_1 j_1 + I_2 j_2 + I_3 j_3 + I_4 j_4 = d_j$$

$$I \cdot \mathbf{k} = I_1 k_1 + I_2 k_2 + I_3 k_3 + I_4 k_4 = 0$$

$$I \cdot \mathbf{l} = I_1 l_1 + I_2 l_2 + I_3 l_3 + I_4 l_4 = 0$$

This system can be solved using Gaussian elimination, but I did not derive an algorithm for doing so. This is a major limitation of my 4D rotation algorithm. However, if I only consider the 4D rotations about the six standard axis-planes, namely, the xy-plane, the yz-plane, the xz-plane, the xw-plane, the yw-plane, and the zw-plane, I can derive a 4D rotation formula. Each standard axis-plane has unit vectors $\mathbf{i}$ and $\mathbf{j}$:

$$\mathbf{i} = \begin{bmatrix}\mathbf{i}_1\\\mathbf{i}_2\\\mathbf{i}_3\\\mathbf{i}_4\end{bmatrix}, \mathbf{i}_m = 1, m \in \{1,2,3,4\} \qquad [9.2]$$

$$\mathbf{i}_q = 0 \text{ for all } q \in \{1,2,3,4\}\backslash\{m\}$$

$$\mathbf{j} = \begin{bmatrix} \mathbf{j}_1 \\ \mathbf{j}_2 \\ \mathbf{j}_3 \\ \mathbf{j}_4 \end{bmatrix}, \mathbf{j}_n = 1, n \in \{1,2,3,4\}\backslash\{m\} \qquad [9.3]$$

$$\mathbf{j}_r = 0 \text{ for all } r \in \{1,2,3,4\}\backslash\{n\}$$

By [9.1], $\mathbf{i} \cdot \mathbf{j} = 0$, so $n \neq m$ and $n \in \{1,2,3,4\}\backslash\{m\}$; otherwise, if $n = m$:

$$\mathbf{i} \cdot \mathbf{j} = \mathbf{i}_m \mathbf{j}_m = 1 \neq 0$$

Apply the appropriate 3D vector product to obtain **k**. By [9.1], [9.2], and [9.3]:

$$\mathbf{i} \cdot \mathbf{k} = \mathbf{j} \cdot \mathbf{k} = 0$$

$$\mathbf{i}_m \mathbf{k}_m = \mathbf{j}_n \mathbf{k}_\mathrm{n} = 0$$

$$\mathbf{k}_m = \mathbf{k}_\mathrm{n} = 0$$

k only has three possibly non-zero components, two of which are $\mathbf{k}_m$ and $\mathbf{k}_n$. Therefore, it only has one non-zero component, and since **k** is a unit vector, this is ± 1:

$$\mathbf{k} = \begin{bmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \mathbf{k}_3 \\ \mathbf{k}_4 \end{bmatrix}, \mathbf{k}_o = \pm 1, o \in \{1,2,3,4\}\backslash\{m, n\} \qquad [9.4]$$

$$\mathbf{k}_s = 0 \text{ for all } s \in \{1,2,3,4\}\backslash\{o\}$$

By [9.1], $\mathbf{i} \cdot \mathbf{k} = \mathbf{j} \cdot \mathbf{k} = 0$, so $o \neq m, n$; otherwise, if $o = m$ or $o = n$:

$$\mathbf{i} \cdot \mathbf{k} = \mathbf{i}_m \mathbf{k}_m = 1 \neq 0 \text{ or } \mathbf{j} \cdot \mathbf{k} = \mathbf{j}_n \mathbf{k}_n = \pm 1 \neq 0$$

Finally, apply the 4D vector product to obtain **l**. By [9.1], [9.2], [9.3], and [9.4]:

$$\mathbf{i} \cdot \mathbf{l} = \mathbf{j} \cdot \mathbf{l} = \mathbf{k} \cdot \mathbf{l} = 0$$

$$\mathbf{i}_m \mathbf{l}_m = \mathbf{j}_n \mathbf{l}_\mathrm{n} = \mathbf{k}_o \mathbf{l}_\mathrm{o} = 0$$

$$\mathbf{l}_m = \mathbf{l}_\mathrm{n} = \mathbf{l}_\mathrm{o} = 0 \tag{9.5}$$

But **l** is a unit vector, so its non-zero component must be ± 1:

$$\mathbf{l} = \begin{bmatrix} \mathbf{l}_1 \\ \mathbf{l}_2 \\ \mathbf{l}_3 \\ \mathbf{l}_4 \end{bmatrix}, \mathbf{l}_p = \pm 1, p = \{1,2,3,4\} \backslash \{m, n, o\} \tag{9.6}$$

Although these calculations only apply when [9.2] and [9.3] hold, rotations about the six standard axis-planes are the easiest rotations to comprehend, so these rotations are the most useful. Now find I with equations [9.2] to [9.6]:

$$I \cdot \mathbf{i} = I_m \cdot \mathbf{i}_m = I_m = d_i$$

$$I \cdot \mathbf{j} = I_n \cdot \mathbf{j}_n = I_n = d_j$$

$$I \cdot \mathbf{k} = I_o \cdot \mathbf{k}_o = \pm I_o = 0$$

$$I \cdot \mathbf{l} = I_p \cdot \mathbf{l}_p = \pm I_p = 0$$

$$I = I_m \mathbf{i} + I_n \mathbf{j} \pm I_o \mathbf{k} \pm I_p \mathbf{l} = d_i \mathbf{i} + d_j \mathbf{j} \tag{9.7}$$

[9.7] does not include **j** nor **k**, so for the six standard axis-planes, only **i** and **j** are needed! To apply [3.3] on the rotation plane, set (IP) as the "x-axis" and identify a line (IQ) on the rotation plane perpendicular to $\overrightarrow{IP}$ as the "y-axis". If $\overrightarrow{IQ}$ is on the rotation plane, then it is perpendicular to **i** and **j**, so use the 4D vector product:

$$\overrightarrow{IQ}_{raw} = \times\left(\overrightarrow{IP}, \mathbf{i}, \mathbf{j}\right)$$
$$= \begin{bmatrix} \overrightarrow{IP}_2(i_3 j_4 - i_4 j_3) - \overrightarrow{IP}_3(i_2 j_4 - i_4 j_2) + \overrightarrow{IP}_4(i_2 j_3 - i_3 j_2) \\ -\overrightarrow{IP}_1(i_3 j_4 - i_4 j_3) + \overrightarrow{IP}_3(i_1 j_4 - i_4 j_1) - \overrightarrow{IP}_4(i_1 j_3 - i_3 j_1) \\ \overrightarrow{IP}_1(i_2 j_4 - i_4 j_2) - \overrightarrow{IP}_2(i_1 j_4 - i_4 j_1) + \overrightarrow{IP}_4(i_1 j_2 - i_2 j_1) \\ -\overrightarrow{IP}_1(i_2 j_3 - i_3 j_2) + \overrightarrow{IP}_2(i_1 j_3 - i_3 j_1) - \overrightarrow{IP}_3(i_1 j_2 - i_2 j_1) \end{bmatrix} \tag{9.8}$$

Normalize $\overrightarrow{IQ}_{raw}$ so that one unit on (IQ) is the same as one unit on (IP):

$$\overrightarrow{IQ} = \frac{\left|\overrightarrow{IP}\right|}{\left|\overrightarrow{IQ}\right|}\overrightarrow{IQ}_{raw} \qquad [9.9]$$

Now apply [6.9], the formula for rotating a point on a rotation plane, on $\overrightarrow{IP}$, $\overrightarrow{IQ}$, $\overrightarrow{OI}$, and ω to find the general 4D rotation formula:

$$P(x,y,z,w) \overset{R}{\rightarrow} P'\left(\left|\overrightarrow{IP}\right| \cos\omega\, \overrightarrow{IP} + \overrightarrow{OI}, \left|\overrightarrow{IP}\right| \sin\omega\, \overrightarrow{IQ} + \overrightarrow{OI}\right) \qquad [9.10]$$

4D Projections

It is conceptually easier to extend 3D projections to 4D than it is for rotations.

$$V = V\{r, \theta, \phi, \psi\} \qquad [10.1]$$

Given a distance f and a viewing point V, f specifies the distance of the *picture hyperplane* away from V. The picture hyperplane is the 4D analogue of the 3D picture plane, because in 4D, a vector $\overrightarrow{OV}$ is perpendicular to a hyperplane. To find coordinates on the picture hyperplane, choose three perpendicular vectors **w**, **h**, and **b** on the picture hyperplane, so that the coefficient m of **w** and the coefficient p of **b** represent respectively the x- and y=coordinates of the computer screen. Take **w** and **h** from [7.7] and [7.8]. Since **b** is perpendicular to **w**, **h**, and $\overrightarrow{OV}$, apply the 4D vector product:

$$\times\left(\mathbf{i}, \mathbf{j}, \overrightarrow{OV}\right) = \mathbf{b}_{raw}$$

$$= \begin{bmatrix} r\cos\theta\,(-\sin\phi\cos\omega - 0) - 0 + 0 \\ r\sin\theta\,(-\sin\phi\cos\omega - 0) - 0 + 0 \\ -r\sin\theta\,(\cos\phi\sin\theta\cos\omega - 0) - r\cos\theta\,(\cos\phi\cos\theta\cos\omega - 0) + 0 \\ r\sin\theta\,(\cos\phi\sin\theta\sin\omega\cos\phi + \sin\phi\sin\omega\sin\phi\sin\theta) + r\cos\theta\,(\cos\phi\cos\theta\sin\omega\cos\phi + \sin\phi\sin\omega\sin\phi\cos\theta) - 0 \end{bmatrix}$$

$$\mathbf{b}_{raw} = \begin{bmatrix} -r\cos\omega\sin\phi\cos\theta \\ -r\cos\omega\sin\phi\sin\theta \\ -r\sin^2\theta\,(\cos\phi\cos\omega) - r\cos^2\theta\,(\cos\phi\cos\omega) \\ r\sin^2\theta\,(\sin\omega\cos^2\phi + \sin\omega\sin^2\phi) + r\cos^2\theta\,(\sin\omega\cos^2\phi + \sin\omega\sin^2\phi) \end{bmatrix}$$

Normalize $\mathbf{b}_{raw}$ so that one unit on $\mathbf{b}$ is one unit:

$$\mathbf{b} = \frac{1}{|\mathbf{b}|}\mathbf{b}_{raw} = \frac{1}{r}\mathbf{b}_{raw}$$

$$\mathbf{b} = \begin{bmatrix} -\cos\omega\sin\phi\cos\theta \\ -\cos\omega\sin\phi\sin\theta \\ -\cos\phi\cos\omega \\ \sin\omega \end{bmatrix} \qquad [10.2]$$

Let Q be the centre of the picture hyperplane. To find the coefficients $m,$ n and p of $\mathbf{w}$, $\mathbf{h}$, and $\mathbf{b}$, express the difference between I and Q in terms of $\mathbf{w}$, $\mathbf{h}$, and $\mathbf{b}$, by using equations [7.10], [7.11], [7.12], [10.1], and [10.2]:

$$I - Q = m\mathbf{w} + n\mathbf{h} + p\mathbf{b}$$

$$P + (\overrightarrow{OV} - P)t - \left(1 + \frac{f}{r_V}\right)\overrightarrow{OV} = m\mathbf{w} + n\mathbf{h} + p\mathbf{b}$$

$$(1-t)\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} + \left(t - 1 - \frac{f}{r}\right)\frac{r}{r}\overrightarrow{\begin{bmatrix} r\sin\omega\sin\phi\cos\theta \\ r\sin\omega\sin\phi\sin\theta \\ r\sin\omega\cos\phi \\ r\cos\omega \end{bmatrix}}$$

$$= m\begin{bmatrix} -\sin\theta \\ \cos\theta \\ 0 \\ 0 \end{bmatrix} + n\begin{bmatrix} \cos\phi\cos\theta \\ \cos\phi\sin\theta \\ -\sin\phi \\ 0 \end{bmatrix} + p\begin{bmatrix} -\cos\omega\sin\phi\cos\theta \\ -\cos\omega\sin\phi\sin\theta \\ -\cos\omega\cos\phi \\ \sin\omega \end{bmatrix}$$

$$(1-t)x + (tr - r - f)\sin\omega\sin\phi\cos\theta$$
$$= -\mathrm{m}\sin\theta + \mathrm{n}\cos\phi\cos\theta - p\cos\omega\sin\phi\cos\theta \qquad [10.3]$$

$$(1-t)y + (tr - r - f)\sin\omega\sin\phi\sin\theta \qquad [10.4]$$
$$= \mathrm{m}\cos\theta + \mathrm{n}\cos\phi\sin\theta - p\cos\omega\sin\phi\sin\theta$$

$$(1-t)z + (tr - r - f)\sin\omega\cos\phi = -n\sin\phi - p\cos\omega\cos\phi \qquad [10.5]$$

$$(1-t)w + (tr - r - f)\cos\omega = p\sin\omega \qquad [10.6]$$

Now solve this system of four equations for the three unknowns n, m, and p. Since the full calculation is so cumbersome, I have attached it as Appendix C. However, I have solved the system.

So the general 4D projection formula is:

$$P(x, y, z, w) \overset{V}{\rightarrow} P(m, p) \qquad [10.7]$$

Conclusion

In this extended essay, I described algorithms to rotate and display regular polygons, polyhedra, and polychora. These algorithms answered my research question: I now know how to rotate and display 4D objects, by using the algorithms I created. I extended my existing mathematical knowledge to reach new conclusions with elementary tools. This is important in high school math investigation, for applications are limited in the classroom.

However, my algorithms are limited. They require that rotation axes pass through the origin. They cannot rotate objects about any axis-plane. They involve convoluted casework that will only become more complicated in higher dimensions. They do not help me determine which vertices to connect in a side, and which sides overlap which.

Most importantly, they are slow and cumbersome, especially if I am to extend the 4D vector product to higher dimensions. These limitations are inherent to my algorithms, so I cannot overcome them without developing fundamentally different ones. My algorithms are not superior in any situation, even though there are many reasons to prefer different rotation and projection techniques in different situations. However, a survey of these other techniques would require another extended essay of quite a different character.

Yet, as an exercise in deriving algorithms for abstract mathematical concepts with elementary tools, my essay has succeeded in its purpose. I have successfully applied vector mathematics in a complex situation with no simple solution, and my successful creation of rotation and display algorithms has answered my research question in the affirmative.

Bibliography

Bowers, Jonathan. “Uniform Polychora and Other Four Dimensional Shapes.” *Hedrondude’s Home Page.* Andrew Weimholt, 2014. Web. 30 Oct. 2014.

Coxeter, Harold Scott MacDonald. *Regular Polytopes.* 3rd ed. New York: Dover Publications, 1973. Print.

International Baccalaureate Organization. *Diploma Programme: Mathematics HL Guide.* Cardiff: International Baccalaureate Organization (UK), 2012. Print.

Jesper T. “cross product vector 4d.” *GameDev.net Game Development Community.* GameDev.net LLC, 18 July 2007. Web. 2 Nov. 2014.

Manning, Henry Parker. *Geometry of Four Dimensions.* New York: Dover Publications, 1956. Print.

Martin, David et al. *Mathematics for the International Student: Mathematics HL (Core).* 3rd ed. Adelaide: Haese Mathematics, 2012. Print.

The Python Standard Library, Version 3.4.2. Python Software Foundation, 2014. Web. 2 Oct. 2014.

Weisstein, Eric. “Spherical Coordinates.” *Wolfram MathWorld*. Wolfram Research, n.d. Web. 15 Sept. 2014.

Appendix A: Source Code

This code is single-spaced in a monospace font for ease of reading. I have coloured irrelevant code in grey and code that correspond to equations in this essay in black. The reader may choose to copy and paste this code into a Python file for compiling.

```
"""
Polychoron Visualizer

This script displays the tesseract, pentachoron, and 16-cell.
This script also displays all regular polyhedra and polygons.
These objects can be rotated and viewed from any place in 4D.
This script is the object of Eugene Y. Q. Shen's IB Extended Essay.
"""
import tkinter as tk
import tkinter.ttk as ttk
import math
TITLE = "Polychoron Visualizer"
DESCRIPTION = "\nThis script displays the regular polychora."
pi = math.pi
WIDTH = 600
HEIGHT = 600
BGCOLOUR = "#CCC"
RADIUS = 100    # the circumradius of regular polygons and polyhedra
ZOOM = 40
DISTANCE = 5    # r_v in my projection algorithms
RETINA = 10   # f in my projection algorithms
CHANGE = 10    # 10 pixels change in radius and 0.1r change in distance
DELAY = 28    # 28 ms per pi/24 rotation = 3 rotations every 4 seconds
ROTANGLE = pi/24
EPSILON = 0.00001
def normalize(points, vector=[1]):    # Used in various equations
    """
    Normalizes a vector.

    Takes two vectors as lists and creates a new vector with
    the magnitude of the second and the direction of the first.
    Default length of the second vector is 1.
    """
    norm = math.sqrt(sum([x**2 for x in points]))
    magnitude = math.sqrt(sum([x**2 for x in vector]))
    return [x/norm*magnitude for x in points]

def convert(point, toCartesian):
    """Converts between hyperspherical and Cartesian coordinates."""
    if toCartesian == True:    # Equation [8.3]
        x = point[0]*math.sin(point[3])*math.sin(point[2])*math.cos(point[1])
        y = point[0]*math.sin(point[3])*math.sin(point[2])*math.sin(point[1])
        z = point[0]*math.sin(point[3])*math.cos(point[2])
        w = point[0]*math.cos(point[3])
        return (x, y, z, w)
    elif toCartesian == False:       # Equation [8.2]
        r = math.sqrt(point[3]**2+point[2]**2+point[1]**2+point[0]**2)
        if points[0] >= 0:
            theta = math.acos(point[1]/math.sqrt(point[1]**2+point[0]**2))
        elif points[0] <= 0:
            theta = 2*pi-math.acos(point[1]/math.sqrt(point[1]**2+point[0]**2))
        phi = abs(math.acos(
            point[2]/math.sqrt(point[2]**2+point[1]**2+point[0]**2)))
        omega = abs(math.acos(point[3]/r))
        return (r, theta, phi, omega)

class Main(ttk.Frame):
    """
    GUI class that manages all windows and actions except the canvas.

    Public methods:
    set_status
    close

    Instance variables:
    mousePressed

    Object variables:
    parent
    canvas
    statusLabel
    statusText
    inputText
    """
```

```
def __init__(self, parent):
    """Construct Main class."""
    self.mousePressed = False
    self.parent = parent
    self.parent.title(TITLE)
    self.parent.geometry(
        '{}x{}+{}+{}'.format(
            WIDTH, HEIGHT,    # Centre the application window
            (self.parent.winfo_screenwidth() - WIDTH) // 2,
            (self.parent.winfo_screenheight() - HEIGHT) // 2))
    self.parent.minsize(WIDTH, HEIGHT)
    self._make_menus()
    ttk.Frame.__init__(self, parent)
    self.pack(fill=tk.BOTH, expand=1)
    self._initUI()

def _initUI(self):

    # Initialize GUI placement and bind buttons.

    style = ttk.Style()
    style.configure("TFrame", background=BGCOLOUR)
    style.configure("TLabel", background=BGCOLOUR)
    style.configure("TButton", background=BGCOLOUR)

    titleLabel = ttk.Label(self, text=TITLE)
    titleLabel.pack()
    self.canvas = Canvas(self)
    self.canvas.pack(fill=tk.BOTH, padx=10, expand=1)
    _guiFrame = ttk.Frame(self)
    _guiFrame.columnconfigure(0, weight=1)
    _guiFrame.pack(fill=tk.X, padx=10, pady=10, expand=0)

    # grid _guiFrame widgets: 4 rows, 11 columns
    self.statusText = tk.StringVar()
    self.statusLabel = ttk.Label(
        _guiFrame, textvariable=self.statusText, foreground="red")
    self.statusLabel.grid(row=0, column=0, columnspan=7, sticky=tk.W)
    self.inputText = tk.StringVar()
    self._inputBox = ttk.Entry(_guiFrame, textvariable=self.inputText)
    self._inputBox.bind('<Key-Return>', self.canvas.take_input)
```

```
# No padding on left, 10px padding on right
self._inputBox.grid(row=1, column=0, columnspan=7,
                  padx=(0,10), sticky=tk.E+tk.W)
self._inputBox.focus()

_viewLabel = ttk.Label(_guiFrame, text="Views: ")
_viewLabel.grid(row=2, column=0, columnspan=3, sticky=tk.E)
_xViewBtn = ttk.Button(_guiFrame, text="x", width=2, command=lambda:
                       self.canvas.set_viewaxis([0, pi/2, pi/2]))
_xViewBtn.bind("<Key-Return>", lambda event:
               self.canvas.set_viewaxis([0, pi/2, pi/2]))
_xViewBtn.grid(row=2, column=3)
_yViewBtn = ttk.Button(_guiFrame, text="y", width=2, command=lambda:
                       self.canvas.set_viewaxis([pi/2, pi/2, pi/2]))
_yViewBtn.bind("<Key-Return>", lambda event:
               self.canvas.set_viewaxis([pi/2, pi/2, pi/2]))
_yViewBtn.grid(row=2, column=4)
_zViewBtn = ttk.Button(_guiFrame, text="z", width=2, command=lambda:
                       self.canvas.set_viewaxis([0, 0, pi/2]))
_zViewBtn.bind("<Key-Return>", lambda event:
               self.canvas.set_viewaxis([0, 0, pi/2]))
_zViewBtn.grid(row=2, column=5)
_wViewBtn = ttk.Button(_guiFrame, text="w", width=2, command=lambda:
                       self.canvas.set_viewaxis([0, 0, 0]))
_wViewBtn.bind("<Key-Return>", lambda event:
               self.canvas.set_viewaxis([0, 0, 0]))
_wViewBtn.grid(row=2, column=6, padx=(0,10))

_zoomLabel = ttk.Label(_guiFrame, text="zoom: ")
_zoomLabel.grid(row=3, column=1, sticky=tk.E)
_zUpBtn = ttk.Button(_guiFrame, text="^", width=2, command=lambda:
                     self.canvas.change('zUp'))
_zUpBtn.bind("<Button-1>", lambda event: self._mouse_down('zUp'))
_zUpBtn.bind("<ButtonRelease-1>", self._mouse_up)
_zUpBtn.bind("<Key-Return>", lambda event:
             self.canvas.change('rUp'))
_zUpBtn.grid(row=3, column=2)
_zDownBtn = ttk.Button(_guiFrame, text="v", width=2, command=lambda:
                       self.canvas.change('zDown'))
_zDownBtn.bind("<Button-1>", lambda event: self._mouse_down('zDown'))
_zDownBtn.bind("<ButtonRelease-1>", self._mouse_up)
_zDownBtn.bind("<Key-Return>", lambda event:
               self.canvas.change('zDown'))
_zDownBtn.grid(row=3, column=3)
_distanceLabel = ttk.Label(_guiFrame, text="d: ")
_distanceLabel.grid(row=3, column=4, sticky=tk.E)
_dUpBtn = ttk.Button(_guiFrame, text="^", width=2, command=lambda:
                     self.canvas.change('dUp'))
_dUpBtn.bind("<Button-1>", lambda event: self._mouse_down('dUp'))
_dUpBtn.bind("<ButtonRelease-1>", self._mouse_up)
_dUpBtn.bind("<Key-Return>", lambda event:
             self.canvas.change('dUp'))
_dUpBtn.grid(row=3, column=5)
_dDownBtn = ttk.Button(_guiFrame, text="v", width=2, command=lambda:
                       self.canvas.change('dDown'))
_dDownBtn.bind("<Button-1>", lambda event: self._mouse_down('dDown'))
_dDownBtn.bind("<ButtonRelease-1>", self._mouse_up)
_dDownBtn.bind("<Key-Return>", lambda event:
               self.canvas.change('dDown'))
_dDownBtn.grid(row=3, column=6, padx=(0,10))

_rotateLabel = ttk.Label(_guiFrame, text="Rotate: ")
_rotateLabel.grid(row=0, column=7, rowspan=2, sticky=tk.E)
_leftRotBtn = ttk.Button(_guiFrame, text="<-", width=3,
                         command=lambda: self.canvas.rotate(0))
_leftRotBtn.bind("<Button-1>", lambda event: self._mouse_down(0))
_leftRotBtn.bind("<ButtonRelease-1>", self._mouse_up)
_leftRotBtn.bind("<Key-Return>", lambda event: self.canvas.rotate(0))
_leftRotBtn.grid(row=0, column=8, rowspan=2, sticky=tk.E)
_rightRotBtn = ttk.Button(_guiFrame, text="->", width=3,
                          command=lambda: self.canvas.rotate(1))
_rightRotBtn.bind("<Button-1>", lambda event: self._mouse_down(1))
_rightRotBtn.bind("<ButtonRelease-1>", self._mouse_up)
_rightRotBtn.bind("<Key-Return>", lambda event: self.canvas.rotate(1))
_rightRotBtn.grid(row=0, column=9, rowspan=2, sticky=tk.W)

_xwRotBtn = ttk.Button(_guiFrame, text="xw", width=3,
                       command=lambda: self.canvas.set_rotaxis('xw'))
_xwRotBtn.bind("<Key-Return>",lambda
               event:self.canvas.set_rotaxis('xw'))
_xwRotBtn.grid(row=2, column=7, sticky=tk.E)
_ywRotBtn = ttk.Button(_guiFrame, text="yw", width=3,
                      command=lambda: self.canvas.set_rotaxis('yw'))
    _ywRotBtn.bind("<Key-Return>",lambda
                  event:self.canvas.set_rotaxis('yw'))
    _ywRotBtn.grid(row=2, column=8)
    _zwRotBtn = ttk.Button(_guiFrame, text="zw", width=3,
                            command=lambda: self.canvas.set_rotaxis('zw'))
    _zwRotBtn.bind("<Key-Return>",lambda
                  event:self.canvas.set_rotaxis('zw'))
    _zwRotBtn.grid(row=2, column=9)
    _xyRotBtn = ttk.Button(_guiFrame, text="xy", width=3,
                            command=lambda: self.canvas.set_rotaxis('xy'))
    _xyRotBtn.bind("<Key-Return>",lambda
                  event:self.canvas.set_rotaxis('xy'))
    _xyRotBtn.grid(row=3, column=7, sticky=tk.E)
    _yzRotBtn = ttk.Button(_guiFrame, text="yz", width=3,
                            command=lambda: self.canvas.set_rotaxis('yz'))
    _yzRotBtn.bind("<Key-Return>",lambda
                  event:self.canvas.set_rotaxis('yz'))
    _yzRotBtn.grid(row=3, column=8)
    _xzRotBtn = ttk.Button(_guiFrame, text="xz", width=3,
                            command=lambda: self.canvas.set_rotaxis('xz'))
    _xzRotBtn.bind("<Key-Return>",lambda
                  event:self.canvas.set_rotaxis('xz'))
    _xzRotBtn.grid(row=3, column=9)



def _make_menus(self):
    # Initialize dropdown menus.
    _menuBar = tk.Menu(self.parent)
    self.parent.config(menu=_menuBar)
    _fileMenu = tk.Menu(_menuBar)
    # underline sets position of keyboard shortcut
    _fileMenu.add_command(label="About", underline=0,
                           command=lambda: self._make_popups('About'))
    _fileMenu.add_command(label="Help", underline=0,
                           command=lambda: self._make_popups('Help'))
    _fileMenu.add_command(label="Exit", underline=1,
                           command=self.close)
    _menuBar.add_cascade(label="File", menu=_fileMenu)
```

```
def _make_popups(self, popUpType):

    # Make pop-up windows based on popUpType.

    # Set individual window data
    if popUpType == 'About':
        titleText = "About this application..."
        messageText = TITLE + '\n' + DESCRIPTION
        buttonText = "OK"
        frameWidth = 400
        frameHeight = 200
    elif popUpType == 'Help':
        titleText = "Help"
        messageText = ("Click the buttons to rotate the "
                       "object or change the view.")
        buttonText = "Dismiss"
        frameWidth = 400
        frameHeight = 120

    # Create pop-up window, each with title, message, and close button
    _popUpFrame = tk.Toplevel(self.parent, background=BGCOLOUR)
    _popUpFrame.title(titleText)
    _popUpMessage = tk.Message(_popUpFrame, text=messageText,
                               width=frameWidth, background=BGCOLOUR)
    _popUpMessage.pack()
    _popUpButton = ttk.Button(_popUpFrame, text=buttonText,
                              command=_popUpFrame.destroy)
    _popUpButton.pack()

    # Centre in root window
    _popUpFrame.geometry('{}x{}+{}+{}'.format(
        frameWidth, frameHeight,
        self.parent.winfo_rootx() +
        (self.parent.winfo_width() - frameWidth) // 2,
        self.parent.winfo_rooty() +
        (self.parent.winfo_height() - frameHeight) // 2))

    # Set all focus on the pop up, stop mainloop in main
    _popUpFrame.grab_set()
    _popUpButton.focus()
    self.wait_window(_popUpFrame)
```

```
def _poll(self, button):
    # Handle continuous mouse presses on button.
    # after(t, foo, arg) calls foo(arg) once after t ms and returns an ID.
    # after calls _poll when mouse pressed, which calls _press every t ms.
    if self.mousePressed:
        self._press(button)
        self.after_pollID = self.parent.after(DELAY, self._poll, button)
def _mouse_down(self, button):
    self.mousePressed = True
    self._poll(button)
def _mouse_up(self, event):
    self.mousePressed = False
    self.parent.after_cancel(self.after_pollID)
def _press(self, button):
    if button == 'zUp':
        self.canvas.change('zUp')
    elif button == 'zDown':
        self.canvas.change('zDown')
    elif button == 'dUp':
        self.canvas.change('dUp')
    elif button == 'dDown':
        self.canvas.change('dDown')
    else:
        self.canvas.rotate(button)


def set_status(self, event):
    """Handle and display status changes."""
    if event == "clear":
        self.statusText.set('')
    elif event == "badinput":
        self.statusText.set("Bad input!")
        self.statusLabel.after(1000, self.set_status, 'clear')


def close(self):
    """Close the application."""
    self.parent.destroy()
```

```
class Canvas(tk.Canvas):
    """
    Display class that manages polytope creation, edits, and display.

    Public methods:
    set_viewaxis
    set_rotaxis
    change
    rotate
    take_input

    Object variables:
    parent
    currPolytope
    """
    def __init__(self, parent):
        """Construct Canvas class."""
        self.parent = parent
        tk.Canvas.__init__(self, parent, background="white",
                           relief=tk.GROOVE, borderwidth=5,
                           width=300, height=200)
        self.currPolytope = Polytope([])
        self._reset()
    def take_input(self, event):
        """Take text input from input box."""
        try:
            translatedInput = self._translate(self.parent.inputText.get())
            if translatedInput:
                self.currPolytope = translatedInput
                self._reset()
        except ValueError:
            self.parent.set_status('badinput')
        self._render()
        self.parent.inputText.set('')
    def change(self, change, amount=CHANGE):
        """Change the polytope radius and viewpoint distance."""
        if change == 'zUp':
            self._zoom += amount
        elif change == 'zDown':
            self._zoom -= amount
        elif change == 'dUp':
            self._distance += amount*0.1
        elif change == 'dDown':
            self._distance -= amount*0.1
        self._render()
    def set_viewaxis(self, viewAxis):
        """Change the current view. Takes hyperspherical axis coordinates."""
        # Check that inputs satisfy restrictions
        for n in range(3):
            while viewAxis[n] >= 2*pi:
                viewAxis[n] -= 2*pi
            while viewAxis[n] < 0:
                viewAxis[n] += 2*pi
        if viewAxis[2] > pi:
            viewAxis[2] = 2*pi - viewAxis[2]
        if viewAxis[1] > pi:
            viewAxis[1] = 2*pi - viewAxis[2]
        self._viewAxis = (viewAxis)
        self._render()

    def _view(self, points):    # Equations [10.1] to [10.6] and Appendix C
        """Return a Cartesian double depending on the viewAxis."""
        so = math.sin(self._viewAxis[2])
        co = math.cos(self._viewAxis[2])
        sp = math.sin(self._viewAxis[1])
        cp = math.cos(self._viewAxis[1])
        st = math.sin(self._viewAxis[0])
        ct = math.cos(self._viewAxis[0])
        u = (so*sp*ct, so*sp*st, so*cp, co)    # Equation [10.1]
        w = (-st, ct, 0, 0)    # Equation [7.7]
        h = (cp*ct, cp*st, -sp, 0)    # Equation [7.8]
        b = (-co*sp*ct, -co*sp*st, -co*cp, so)    # Equation [10.2]
        d = self._distance*RADIUS
        f = d + RETINA
        result = []
        for count in range(len(points)):
            x = points[count][0]
            y = points[count][1]
            z = points[count][2]
            w = points[count][3]
            # Equation [7.6]
            t = (f-x*u[0]-y*u[1]-z*u[2]-w*u[3])/(d-x*u[0]-y*u[1]-z*u[2]-w*u[3])
        # Equations from Appendix C
        if abs(self._viewAxis[2]) < EPSILON:
            if abs(self._viewAxis[1] - pi/2) < EPSILON:
                m = (1-t)*-z*sp
                if (abs(self._viewAxis[0]) < EPSILON or
                    abs(self._viewAxis[0] - pi) < EPSILON):
                    n = (1-t)*y*ct
                    p = (1-t)*(-x*co*sp*ct)
                else:
                    n = (1-t)*(x-y*ct/st)*-st
                    p = ((1-t)*y-n*ct)/(-co*sp*st)
            else:
                if (abs(self._viewAxis[0]) < EPSILON or
                    abs(self._viewAxis[0] - pi) < EPSILON):
                    n = (1-t)*y*ct
                    m = (1-t)*(x-z*sp/cp*ct)*cp/ct
                else:
                    n = (1-t)*(x-y*ct/st)*(st/(ct*ct-st*st))
                    m = ((1-t)*(y-z*sp/cp*st)+n*ct)*cp/st
                p = ((1-t)*z+m*sp)/(-co*cp)

        else:
            p = ((1-t)*w+(d*t-f)*co)/so
            if abs(self._viewAxis[1]) < EPSILON:
                if (abs(self._viewAxis[0]) < EPSILON or
                    abs(self._viewAxis[0] - pi) < EPSILON):
                    n = (1-t)*y*ct
                    m = (1-t)*x*cp*ct
                else:
                    n = (1-t)*(x-y*ct/st)*-st
                    m = ((1-t)*y-n*ct)/(cp*st)
            else:
                m = -((1-t)*z+(d*t-f)*so*sp+p*co*cp)/sp
                if (abs(self._viewAxis[0]) < EPSILON or
                    abs(self._viewAxis[0] - pi) < EPSILON):
                    n = (1-t)*y*ct
                else:
                    n = -((1-t)*x+(d*t-f)*so*sp*ct-m*cp*ct+p*co*sp*ct)/st
        result.append((n*self._zoom, p*self._zoom))
    return result
```

```
def set_rotaxis(self, rotAxis):
    """Change the current rotation axis-plane."""
    if rotAxis == "xw":
        self.currPolytope.set_rotaxis((1,0,0,0),(0,0,0,1))
    elif rotAxis == 'yw':
        self.currPolytope.set_rotaxis((0,1,0,0),(0,0,0,1))
    elif rotAxis == 'zw':
        self.currPolytope.set_rotaxis((0,0,1,0),(0,0,0,1))
    elif rotAxis == 'xy':
        self.currPolytope.set_rotaxis((1,0,0,0),(0,1,0,0))
    elif rotAxis == 'yz':
        self.currPolytope.set_rotaxis((0,1,0,0),(0,0,1,0))
    elif rotAxis == 'xz':
        self.currPolytope.set_rotaxis((1,0,0,0),(0,0,1,0))
    self._render()

def rotate(self, direction, rotAngle=ROTANGLE):
    """Rotate polytope on button press by ROTANGLE radians."""
    if direction == 0:
        self.currPolytope.rotate(rotAngle)
    elif direction == 1:
        self.currPolytope.rotate(-rotAngle)
    self._render()

def _translate(self, entry):

    # Translate text input to return a Polytope object.

    # Clear canvas, return empty polytope for _render to process
    if entry == ('' or "clear" or "reset"):
        return Polytope([])

    # Exit application
    if entry == "quit" or entry == "exit" or entry == "close":
        self.parent.close()

    # Set axis of projection
    elif entry.startswith("v"):
        self.set_viewaxis([float(num) for num in entry[1:].split(',')])

    # Pentachoron coordinates, side-length 4r
    elif entry == "{3,3,3}":
        r = 2*RADIUS
        return Polytope(([(r/math.sqrt(10),r/math.sqrt(6),r/math.sqrt(3),r),
                          (r/math.sqrt(10),r/math.sqrt(6),r/math.sqrt(3),-r),

  (r/math.sqrt(10),r/math.sqrt(6),-2*r/math.sqrt(3),0),
                          (r/math.sqrt(10),-r*math.sqrt(3/2),0,0),
                          (-2*r*math.sqrt(2/5),0,0,0)],
                         ((0,1),(0,2),(0,3),(0,4),(1,2),
                          (1,3),(1,4),(2,3),(2,4),(3,4))))

    # Tesseract coordinates, side-length 2r
    elif entry == "{4,3,3}":
        r = RADIUS
        return Polytope(([(r,r,r,r),(r,r,r,-r),(r,r,-r,r),(r,r,-r,-r),
                        (r,-r,r,r),(r,-r,r,-r),(r,-r,-r,r),(r,-r,-r,-r),
                        (-r,r,r,r),(-r,r,r,-r),(-r,r,-r,r),(-r,r,-r,-r),
                        (-r,-r,r,r),(-r,-r,r,-r),(-r,-r,-r,r),(-r,-r,-r,-r)],
                       ((0,1),(1,3),(3,2),(2,0),(12,13),(13,15),(15,14),(14,12),
                        (4,5),(5,7),(7,6),(6,4),(8,9),(9,11),(11,10),(10,8),
                        (0,4),(1,5),(2,6),(3,7),(8,12),(9,13),(10,14),(11,15),
                        (0,8),(1,9),(2,10),(3,11),(4,12),(5,13),(6,14),(7,15))))

    # Hexadecachoron coordinates, side-length 4r
    elif entry == "{3,3,4}":
        r = 2*RADIUS
        return Polytope(([(r,0,0,0),(-r,0,0,0),(0,r,0,0),(0,-r,0,0),
                          (0,0,r,0),(0,0,-r,0),(0,0,0,r),(0,0,0,-r)],
                         ((0,2),(0,3),(0,4),(0,5),(0,6),(0,7),
                          (1,2),(1,3),(1,4),(1,5),(1,6),(1,7),
                          (2,4),(2,5),(2,6),(2,7),(3,4),(3,5),
                          (3,6),(3,7),(4,6),(4,7),(5,6),(5,7))))

    # Schlafli symbol: {p/d}
    elif entry.startswith("{") and entry.endswith("}"):
        if ',' in entry:
            return Polytope(self._schlafli3D(entry[1:-1]))
        else:
            return Polytope(self._schlafli2D(entry[1:-1]))
    else:
        raise ValueError
```

```
def _schlafli2D(self, entry):
    # Take 2D Schlafli symbol and return its hyperspherical coordinates.
    num = entry.split('/')
    p = int(num[0])
    d = 1
    if len(num) > 1:
         d = int(num[1])
    rs = [RADIUS]*p
    thetas = [(2*k*d*pi/p+pi/2) for k in range(p)]
    phis = [pi/2]*p
    omegas = [pi/2]*p
    edges = [(k-1,k) for k in range(p)]

    points = []
    for n in range(len(rs)):
       points.append(convert((rs[n], thetas[n], phis[n], omegas[n]),True))
    return points, edges


def _schlafli3D(self, entry):
    # Take 3D Schlafli symbol, return hyperspherical coordinates and edges.
    # No support for star polyhedra.
    num = entry.split(',')
    p = int(num[0])
    q = int(num[1])
    r = RADIUS
    ap = pi*(1-2/p)
    aq = 2*pi/q

    # Coordinates of the north pole
    rs = [r]
    thetas = [0]
    phis = [0]
    edges = []

    # Coordinates of the first ring
    rs += [r]*q
    thetas += [k*aq for k in range(q)]
    firstPhi = 2*math.acos(math.sqrt((1-math.cos(ap))/(1-math.cos(aq))))
    phis += [firstPhi]*q
```

```
# Edges of the first ring
edges += [(0,k) for k in range(1,q+1)]
if p == 3:
    edges += [(k,k+1) for k in range(1,q)]
    edges += [(q,1)]

if round(firstPhi,4) <= round(math.pi/2,4):    # Catching float errors
    if round(firstPhi,4) < round(math.pi/2,4):
        # Coordinates of the last ring, excludes octahedron
        rs += [r]*q
        thetas += [(k+1/2)*aq for k in range(q)]
        phis += [pi-firstPhi]*q

    # Coordinates of the south pole
    rs += [r]
    thetas += [0]
    phis += [pi]

    # Edges of the last ring
    n = len(rs)
    edges += [(n-1,n-q+k) for k in range(-1,q-1)]
    if p == 3:    # Icosahedron edges
        edges += [(n-q+k-1,n-q+k) for k in range(0,q-1)]
        edges += [(n-2,q+1)]

    if p == 5:
        # Coordinates of the middle dodecahedron rings
        s = math.sqrt(2*r**2*(1-math.cos(firstPhi)))
        t = math.sqrt(2*s**2*(1-math.cos(ap)))
        secondPhi = math.acos(1-(t/r)**2/2)
        r2 = r*math.sin(secondPhi)
        a = math.acos(1-(t/r2)**2/2)/2
        b = math.acos(1-(s/r2)**2/2)
        rs += [r]*2*q
        thetas += [k*aq+a for k in range(q)]
        thetas += [k*aq+a+b for k in range(q)]
        phis += [secondPhi]*2*q
        rs += [r]*2*q
        thetas += [(k+1/2)*aq+a for k in range(q)]
        thetas += [(k+1/2)*aq+a+b for k in range(q)]
        phis += [pi-secondPhi]*2*q
```

```
            # Edges of the middle dodecahedron rings
            edgeTypes = ((1,7,0), (1,9,1), (4,10,0), (4,12,1),
                        (8,3,0), (14,3,0), (8,8,1), (11,3,0))
            for edge in edgeTypes:
                edges += [(k%3+edge[0], k+edge[0]+edge[1])
                          for k in range(0+edge[2],3+edge[2])]

        elif n > 2*q:
            # Edges of the middle nondodecahedron rings
            edges += [(k,q+k) for k in range(1,q+1)]
            edges += [(k+1,q+k) for k in range(1,q)]
            edges += [(1,2*q)]

    omegas = [pi/2]*len(rs)
    points = []
    for n in range(len(rs)):
        points.append(convert((rs[n], thetas[n], phis[n], omegas[n]),True))
    return points, edges

def _render(self):
    # Clear the canvas, centre, and display Polytope object.
    self.delete(tk.ALL)
    if self.currPolytope.get_edges():    # Equation [15.3]
        w = self.winfo_width()//2
        h = self.winfo_height()//2
        points = [(-point[0]+w, point[1]+h) for point in
                  self._view(self.currPolytope.get_points())]
        edges = self.currPolytope.get_edges()
        for edge in edges:
            self.create_line(points[edge[0]], points[edge[1]],
                             fill='#000', width=5)
            self.create_line(points[edge[0]], points[edge[1]],
                             fill='#CCC', width=3)

def _reset(self):
    self.delete(tk.ALL)
    self._zoom = ZOOM
    self._distance = DISTANCE
    self.set_rotaxis('xw')
    self.set_viewaxis([0, 0, pi/2])
```

```
class Polytope():
    """

    Drawing class that stores polytope coordinates and manages rotations.

    Public methods:
    get_points
    get_edges
    set_rotaxis
    rotate

    """


    def __init__(self, data):
        """Construct Polytope class."""
        if data:
            self._number = len(data[0])
            self._points = data[0]
            self._edges = data[1]
        elif not data:
            self._number = 0
            self._points = []
            self._edges = []


    def get_points(self):
        """Return a list of points of the polytope."""
        return self._points


    def get_edges(self):
        """Return a list of edges of the polytope."""
        return self._edges


    def set_rotaxis(self, i, j):
        self._axis_i = i
        self._axis_j = j
    def rotate(self, rotAngle):    # Equations [9.7] to [9.10]
        """
        Rotate the current polytope by rotAngle about the plane
        defined by i and j, where i and j are quadruples.
        """
        i = self._axis_i
        j = self._axis_j
        cos = math.cos(rotAngle)
        sin = math.sin(rotAngle)

        n = 0
        for n in range(self._number):
            p = self._points[n]
            r = sum([p[t]*i[t] for t in range(4)])
            s = sum([p[t]*j[t] for t in range(4)])
            I = [i[t]*r+j[t]*s for t in range(4)]    # Equation [9.7]
            ip = [p[t]-I[t] for t in range(4)]
            if ip != [0,0,0,0]:    # If I = P, then there is no rotation
                # Equation [9.8]
                iq = [(ip[1]*(i[2]*j[3]-i[3]*j[2]) - ip[2]*(i[1]*j[3]-i[3]*j[1]) +
                       ip[3]*(i[1]*j[2]-i[2]*j[1])),
                      (-ip[0]*(i[2]*j[3]-i[3]*j[2]) + ip[2]*(i[0]*j[3]-i[3]*j[0]) -
                       ip[3]*(i[0]*j[2]-i[2]*j[0])),
                      (ip[0]*(i[1]*j[3]-i[3]*j[1]) - ip[1]*(i[0]*j[3]-i[3]*j[0]) +
                       ip[3]*(i[0]*j[1]-i[1]*j[0])),
                      (-ip[0]*(i[1]*j[2]-i[2]*j[1]) + ip[1]*(i[0]*j[2]-i[2]*j[0]) -
                       ip[2]*(i[0]*j[1]-i[1]*j[0]))]
                iq = normalize(iq, ip)    # Equation [9.9]
                # Equation [9.10]
                self._points[n] = [ip[t]*cos + iq[t]*sin + I[t] for t in range(4)]

root = tk.Tk()
main = Main(root)
root.mainloop()
```

Appendix B: 4D Vector Product Derivation

I adapt this from the IB Math HL 3D vector product derivation (Martin 423).

Given vectors $\mathbf{i} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \end{bmatrix}$, $\mathbf{j} = \begin{bmatrix} j_1 \\ j_2 \\ j_3 \\ j_4 \end{bmatrix}$, and $\mathbf{k} = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix}$, to find a vector $\mathbf{l} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$

so that $\mathbf{i} \cdot \mathbf{l} = \mathbf{j} \cdot \mathbf{l} = \mathbf{k} \cdot \mathbf{l} = 0$, it must be that:

$$\mathbf{i} \cdot \mathbf{l} = i_1x + i_2y + i_3z + i_4w = 0 \qquad [13.1]$$

$$\mathbf{j} \cdot \mathbf{l} = j_1x + j_2y + j_3z + j_4w = 0 \qquad [13.2]$$

$$\mathbf{k} \cdot \mathbf{l} = k_1x + k_2y + k_3z + k_4w = 0 \qquad [13.3]$$

Multiply and subtract:

$$j_1(i_1x + i_2y + i_3z + i_4w) - i_1(j_1x + j_2y + j_3z + j_4w) = 0$$

$$(j_1i_2 - i_1j_2)y + (j_1i_3 - i_1j_3)z + (j_1i_4 - i_1j_4)w = 0 \qquad [13.4]$$

$$k_1(j_1x + j_2y + j_3z + j_4w) - j_1(k_1x + k_2y + k_3z + k_4w) = 0$$

$$(k_1j_2 - j_1k_2)y + (k_1j_3 - j_1k_3)z + (j_1k_4 - k_1j_4)w = 0 \qquad [13.5]$$

$$k_1(i_1x + i_2y + i_3z + i_4w) - i_1(k_1x + k_2y + k_3z + k_4w) = 0$$

$$(k_1i_2 - i_1k_2)y + (k_1i_3 - i_1k_3)z + (k_1i_4 - i_1k_4)w = 0 \qquad [13.6]$$

Multiply and subtract again, if j_1 or i_1 and $(k_1j_2 - j_1k_2)$ or $(j_1i_2 - i_1j_2)$ are not 0, by using [13.4] and [13.5]:

$$(k_1j_2 - j_1k_2)\big((j_1i_2 - i_1j_2)y + (j_1i_3 - i_1j_3)z + (j_1i_4 - i_1j_4)w\big)$$
$$- (j_1i_2 - i_1j_2)\big((k_1j_2 - j_1k_2)y + (k_1j_3 - j_1k_3)z + (j_1k_4 - k_1j_4)w\big) = 0$$

$$(k_1j_2 - j_1k_2)\big((j_1i_3 - i_1j_3)z + (j_1i_4 - i_1j_4)w\big)$$
$$= (j_1i_2 - i_1j_2)\big((k_1j_3 - j_1k_3)z + (j_1k_4 - k_1j_4)w\big)$$
$$\big((k_1j_2 - j_1k_2)(j_1i_3 - i_1j_3) - (j_1i_2 - i_1j_2)(k_1j_3 - j_1k_3)\big)z$$
$$= \big((j_1i_2 - i_1j_2)(j_1k_4 - k_1j_4) - (k_1j_2 - j_1k_2)(j_1i_4 - i_1j_4)\big)w$$

Express z and w parametrically, for some $t \in \mathbb{R}, t \neq 0$:

$$w = \big((i_3j_1 - i_1j_3)(j_2k_1 - j_1k_2) + (i_1j_2 - i_2j_1)(j_3k_1 - j_1k_3)\big)t$$
$$z = \big((i_2j_1 - i_1j_2)(j_1k_4 - j_4k_1) + (i_4j_1 - i_1j_4)(j_1k_2 - j_2k_1)\big)t$$

Or:

$$w = \big((i_4k_1 - i_1k_4)(j_2k_1 - j_1k_2) + (i_1k_2 - i_2k_1)(j_3k_1 - j_1k_3)\big)t \qquad [13.7]$$
$$z = \big((i_2k_1 - i_1k_2)(j_3k_1 - j_1k_3) + (i_1k_2 - k_1i_2)(j_2k_1 - j_1k_2)\big)t \qquad [13.8]$$

Now substitute [13.7] and [13.8] into [13.4]:

$$(j_1i_2 - i_1j_2)y + (j_1i_3 - i_1j_3)\big((j_1i_2 - i_1j_2)(j_1k_4 - k_1j_4) - (k_1j_2 - j_1k_2)(j_1i_4 - i_1j_4)\big)t$$
$$+ (j_1i_4 - i_1j_4)\big((k_1j_2 - j_1k_2)(j_1i_3 - i_1j_3) - (j_1i_2 - i_1j_2)(k_1j_3 - j_1k_3)\big)t$$
$$= 0$$

$$(j_1i_2 - i_1j_2)y = -(j_1i_3 - i_1j_3)(j_1i_2 - i_1j_2)(j_1k_4 - k_1j_4)t$$
$$+ (j_1i_3 - i_1j_3)(k_1j_2 - j_1k_2)(j_1i_4 - i_1j_4)t$$
$$- (j_1i_4 - i_1j_4)(k_1j_2 - j_1k_2)(j_1i_3 + i_1j_3)t$$
$$+ (j_1i_4 - i_1j_4)(j_1i_2 - i_1j_2)(k_1j_3 - j_1k_3)t$$

$$y = \big((i_4j_1 - i_1j_4)(j_3k_1 - j_1k_3) + (i_1j_3 - i_3j_1)(j_1k_4 - j_4k_1)\big)t \qquad [13.9]$$

Finally, substitute [13.7], [13.8], and [13.9] into [13.1]:

$$i_1x + i_2\big((j_1i_4 - i_1j_4)(k_1j_3 - j_1k_3) - (j_1i_3 - i_1j_3)(j_1k_4 - k_1j_4)\big)t$$
$$+ i_3\big((j_1i_2 - i_1j_2)(j_1k_4 - k_1j_4) - (k_1j_2 - j_1k_2)(j_1i_4 - i_1j_4)\big)t$$
$$+ i_4\big((k_1j_2 - j_1k_2)(j_1i_3 - i_1j_3) - (j_1i_2 - i_1j_2)(k_1j_3 - j_1k_3)\big)t = 0$$

$$i_1x = -i_2(j_1i_4 - i_1j_4)(k_1j_3 - j_1k_3)t + i_2(j_1i_3 - i_1j_3)(j_1k_4 - k_1j_4)t$$
$$- i_3(j_1i_2 - i_1j_2)(j_1k_4 - k_1j_4)t + i_3(k_1j_2 - j_1k_2)(j_1i_4 - i_1j_4)t$$
$$- i_4(k_1j_2 - j_1k_2)(j_1i_3 - i_1j_3)t + i_4(j_1i_2 - i_1j_2)(k_1j_3 - j_1k_3)t$$

$$i_1x = -i_2j_1i_4k_1j_3t + i_2j_1i_4j_1k_3t + i_2i_1j_4k_1j_3t - i_2i_1j_4j_1k_3t$$
$$+ i_2j_1i_3j_1k_4t - i_2j_1i_3k_1j_4t - i_2i_1j_3j_1k_4t + i_2i_1j_3k_1j_4t$$
$$- i_3j_1i_2j_1k_4t + i_3j_1i_2k_1j_4t + i_3i_1j_2j_1k_4t - i_3i_1j_2k_1j_4t$$
$$+ i_3k_1j_2j_1i_4t - i_3k_1j_2i_1j_4t - i_3j_1k_2j_1i_4t + i_3j_1k_2i_1j_4t$$
$$- i_4k_1j_2j_1i_3t + i_4k_1j_2i_1j_3t + i_4j_1k_2j_1i_3t - i_4j_1k_2i_1j_3t$$
$$+ i_4j_1i_2k_1j_3t - i_4j_1i_2j_1k_3t - i_4i_1j_2k_1j_3t + i_4i_1j_2j_1k_3t$$

The coloured expressions cancel:

$$i_1x = +i_2i_1j_4k_1j_3t - i_2i_1j_4j_1k_3t - i_2i_1j_3j_1k_4t + i_2i_1j_3k_1j_4t$$
$$+ i_3i_1j_2j_1k_4t - i_3i_1j_2k_1j_4t - i_3k_1j_2i_1j_4t + i_3j_1k_2i_1j_4t$$
$$+ i_4k_1j_2i_1j_3t - i_4j_1k_2i_1j_3t - i_4i_1j_2k_1j_3t + i_4i_1j_2j_1k_3t$$

$$x = \big((i_2j_4 - i_4j_2)\,(j_3k_1 - j_1k_3) + (i_3j_2 - i_2j_3)(j_1k_4 - j_4k_1)$$
$$+ (i_4j_3 - i_3j_4)(j_2k_1 - j_1k_2)\big)t \qquad [13.10]$$

Since I have not done any divisions, this equation always holds true. However, this calculation does not apply if j_1 and i_1 or $(k_1 j_2 - j_1 k_2)$ and $(j_1 i_2 - i_1 j_2)$ are 0, in which case I must use [13.4] and [13.6] or [13.5] and [13.6]. I could now repeat the calculation twice and combine the results of [13.7], [13.8], [13.9], and [13.10]. However, my equations became far too cumbersome; instead, I used the 4D vector product from “cross product vector 4d” (Jesper T):

$$\mathbf{l} = \times(\mathbf{i}, \mathbf{j}, \mathbf{k}) = \begin{bmatrix} i_2(j_3k_4 - j_4k_3) - i_3(j_2k_4 - j_4k_2) + i_4(j_2k_3 - j_3k_2) \\ -i_1(j_3k_4 - j_4k_3) + i_3(j_1k_4 - j_4k_1) - i_4(j_1k_3 - j_3k_1) \\ i_1(j_2k_4 - j_4k_2) - i_2(j_1k_4 - j_4k_1) + i_4(j_1k_2 - j_2k_1) \\ -i_1(j_2k_3 - j_3k_2) + i_2(j_1k_3 - j_3k_1) - i_3(j_1k_2 - j_2k_1) \end{bmatrix} \quad [13.11]$$

[13.7], [13.8], [13.9], and [13.10] all satisfy [13.11], and so [13.11] is indeed the general equation for the 4D vector product, as described by Jesper T.

Appendix C: 4D Projection Equations Solutions

I solve for n, m, and p in [10.3], [10.4], [10.5], and [10.6]. I number the casework for ease of reading; there are eight cases in all.

$$(1-t)x + (tr - r - f)\sin\omega\sin\phi\cos\theta = -\mathrm{m}\sin\theta + \mathrm{n}\cos\phi\cos\theta - p\cos\omega\sin\phi\cos\theta \quad [10.3]$$

$$(1-t)y + (tr - r - f)\sin\omega\sin\phi\sin\theta = \mathrm{m}\cos\theta + \mathrm{n}\cos\phi\sin\theta - p\cos\omega\sin\phi\sin\theta \quad [10.4]$$

$$(1-t)z + (tr - r - f)\sin\omega\cos\phi = -n\sin\phi - p\cos\omega\cos\phi \quad [10.5]$$

$$(1-t)w + (tr - r - f)\cos\omega = p\sin\omega \quad [10.6]$$

1. If $\sin\omega \neq 0$, then by [10.6]:

$$p = \frac{(1-t)w + (dt - f)\cos\omega}{\sin\omega}$$

1.1. If $\sin\phi \neq 0$, then by [10.5]:

$$m = \frac{(1-t)z + (dt - f)\sin\omega\cos\phi + p\cos\omega\cos\phi}{-\sin\phi}$$

1.1.1. If $\sin\theta \neq 0$, then by [10.3]:

$$n = \frac{(1-t)x + (dt - f)\sin\omega\sin\phi\cos\theta - m\cos\phi\cos\theta + p\cos\omega\sin\phi\cos\theta}{-\sin\theta}$$

1.1.2. If $\sin\theta = 0$, then $\cos\theta = \pm 1 = \sec\theta$, and by [10.4]:

$$n\cos\theta = (1-t)y + (dt - f)\sin\omega\sin\phi\sin\theta - m\cos\phi\sin\theta + p\cos\omega\sin\phi\sin\theta$$

$$n = (1-t)y\cos\theta$$

1.2. If $\sin\phi = 0$, then $\cos\phi = \pm 1 = \sec\phi$.

1.2.1. If $\sin\theta \neq 0$, then by [10.3] and [10.4]:

$$m\cos\phi\sin\theta = (1-t)y + (dt-f)\sin\omega\sin\phi\sin\theta - n\cos\theta + p\cos\omega\sin\phi\sin\theta$$

$$m = \frac{(1-t)y - n\cos\theta}{\cos\phi\sin\theta}$$

$$-n\sin\theta = (1-t)x + (dt-f)\sin\omega\sin\phi\cos\theta - m\cos\phi\cos\theta + p\cos\omega\sin\phi\cos\theta$$

$$-n\sin\theta = (1-t)x - \frac{(1-t)y - n\cos\theta}{\cos\phi\sin\theta}\cos\phi\cos\theta$$

$$n\left(-\sin\theta - \frac{\cos^2\theta}{\sin\theta}\right) = (1-t)x - (1-t)y\frac{\cos\theta}{\sin\theta}$$

$$n = (1-t)\left(x - y\frac{\cos\theta}{\sin\theta}\right)(-\sin\theta)$$

1.2.2. If $\sin\theta = 0$, then $\cos\theta = \pm 1 = \sec\theta$, and by [10.3] and [10.4]:

$$m\cos\phi\cos\theta = (1-t)x + (dt-f)\sin\omega\sin\phi\cos\theta + n\sin\theta + p\cos\omega\sin\phi\cos\theta$$

$$m = (1-t)x\cos\phi\cos\theta$$

$$n\cos\theta = (1-t)y + (dt-f)\sin\omega\sin\phi\sin\theta - m\cos\phi\sin\theta + p\cos\omega\sin\phi\sin\theta$$

$$n = (1-t)y\cos\theta$$

2. If $\sin\omega = 0$, then $\cos\omega = \pm 1 = \sec\omega$.

2.1. If $\cos\phi \neq 0$, then by [10.5]:

$$-p\cos\omega\cos\phi = (1-t)z + (dt-f)\sin\omega\cos\phi + m\sin\phi$$

$$p = \frac{(1-t)z + m\sin\phi}{-\cos\omega\cos\phi}$$

2.1.1. If $\sin\theta \neq 0$, then by [10.3] and [10.4]:

$$m\cos\phi\sin\theta = (1-t)y + (dt-f)\sin\omega\sin\phi\sin\theta + n\cos\theta + p\cos\omega\sin\phi\sin\theta$$

$$m\cos\phi\sin\theta = (1-t)y + n\cos\theta + \frac{(1-t)z + m\sin\phi}{-\cos\omega\cos\phi}\cos\omega\sin\phi\sin\theta$$

$$m\left(\cos\phi\sin\theta + \frac{\sin^2\phi}{\cos\phi}\sin\theta\right) = (1-t)y + n\cos\theta - (1-t)z\frac{\sin\phi}{\cos\phi}\sin\theta$$

$$m = \left((1-t)\left(y - z\frac{\sin\phi}{\cos\phi}\sin\theta\right) + n\cos\theta\right)\frac{\cos\phi}{\sin\theta}$$

$$-n\sin\theta = (1-t)x + (dt-f)\sin\omega\sin\phi\cos\theta - m\cos\phi\cos\theta + p\cos\omega\sin\phi\cos\theta$$

$$-n\sin\theta = (1-t)x - m\cos\phi\cos\theta + \frac{(1-t)z + m\sin\phi}{-\cos\omega\cos\phi}\cos\omega\sin\phi\cos\theta$$

$$-n\sin\theta = (1-t)\left(x - z\frac{\sin\phi}{\cos\phi}\cos\theta\right) - m\left(\cos\phi\cos\theta + \frac{\sin^2\phi}{\cos\phi}\cos\theta\right)$$

$$-n\sin\theta = (1-t)\left(x - z\frac{\sin\phi}{\cos\phi}\cos\theta\right)$$
$$-\left((1-t)\left(y - z\frac{\sin\phi}{\cos\phi}\sin\theta\right) + n\cos\theta\right)\left(\cos^2\phi\frac{\cos\theta}{\sin\theta} + \sin^2\phi\frac{\cos\theta}{\sin\theta}\right)$$

$$n\left(-\sin\theta + \frac{\cos^2\theta}{\sin\theta}\right) = (1-t)\left(x - z\frac{\sin\phi}{\cos\phi}\cos\theta - y\frac{\cos\theta}{\sin\theta} + z\frac{\sin\phi}{\cos\phi}\cos\theta\right)$$

$$n = (1-t)\left(x - y\frac{\cos\theta}{\sin\theta}\right)\left(\frac{\sin\theta}{\cos^2\theta - \sin^2\theta}\right)$$

2.1.2. If $\sin\theta = 0$, then $\cos\theta = \pm 1 = \sec\theta$, and by [10.3] and [10.4]:

$$m\cos\phi\cos\theta = (1-t)x + (dt-f)\sin\omega\sin\phi\cos\theta + n\sin\theta + p\cos\omega\sin\phi\cos\theta$$

$$m\cos\phi\cos\theta = (1-t)x + \frac{(1-t)z + m\sin\phi}{-\cos\omega\cos\phi}\cos\omega\sin\phi\cos\theta$$

$$m\left(\cos\phi\cos\theta + \frac{\sin^2\phi}{\cos\phi}\cos\theta\right) = (1-t)\left(x - z\frac{\sin\phi}{\cos\phi}\cos\theta\right)$$

$$m = (1-t)\left(x - z\frac{\sin\phi}{\cos\phi}\cos\theta\right)\left(\frac{\cos\phi}{\cos\theta}\right)$$

$$n\cos\theta = (1-t)y + (dt-f)\sin\omega\sin\phi\sin\theta - m\cos\phi\sin\theta + p\cos\omega\sin\phi\sin\theta$$

$$n = (1-t)y\cos\theta$$

2.2. If $\cos\phi = 0$, then $\sin\phi = \pm 1 = \csc\phi$, and by [10.5]:

$$-m\sin\phi = (1-t)z + (dt-f)\sin\omega\cos\phi + p\cos\omega\cos\phi$$

$$m = (1-t)(-z\sin\phi)$$

2.2.1. If $\sin\theta \neq 0$, then by [10.3] and [10.4]:

$$-p\cos\omega\sin\phi\sin\theta = (1-t)y + (dt-f)\sin\omega\sin\phi\sin\theta - n\cos\theta - m\cos\phi\sin\theta$$

$$p = \frac{(1-t)y - n\cos\theta}{-\cos\omega\sin\phi\sin\theta}$$

$$-n\sin\theta = (1-t)x + (dt-f)\sin\omega\sin\phi\cos\theta - m\cos\phi\cos\theta + p\cos\omega\sin\phi\cos\theta$$

$$-n\sin\theta = (1-t)x + \frac{(1-t)y - n\cos\theta}{-\cos\omega\sin\phi\sin\theta}\cos\omega\sin\phi\cos\theta$$

$$n\left(-\sin\theta - \frac{\cos^2\theta}{\sin\theta}\right) = (1-t)\left(x - y\frac{\cos\theta}{\sin\theta}\right)$$

$$n = (1-t)\left(x - y\frac{\cos\theta}{\sin\theta}\right)(-\sin\theta)$$

2.2.2. Finally, if $\sin\theta = 0$, then $\cos\theta = \pm 1 = \sec\theta$, and by [10.3] and [10.4]:

$$n\cos\theta = (1-t)y + (dt-f)\sin\omega\sin\phi\sin\theta - m\cos\phi\sin\theta + p\cos\omega\sin\phi\sin\theta$$

$$n = (1-t)y\cos\theta$$

$$-p\cos\omega\sin\phi\cos\theta = (1-t)x + (dt-f)\sin\omega\sin\phi\cos\theta + n\sin\theta - m\cos\phi\cos\theta$$

$$p = (1-t)(-x\cos\omega\sin\phi\cos\theta)$$

Appendix D: Tkinter Canvas Coordinates

Tkinter's *canvas* displays objects on the computer screen. The canvas uses a 2D Cartesian coordinate system with the origin in the top-left corner and y-values increasing towards the bottom. This contrasts with the standard 2D Cartesian coordinate system, which has the origin in the centre and y-values increasing towards the top. If I have the standard 2D Cartesian coordinates of an object, to display the object on the canvas, I must both reflect the object about the x-axis and translate the result to the centre of the canvas. If w and h are respectively the width and the height of the canvas, then the formula that properly displays standard 2D Cartesian coordinates in canvas coordinates is:

$$P(x,y) \to P'\left(x+\frac{w}{2}, -y+\frac{h}{2}\right) \qquad [15.1]$$

So given (x,y), I actually make Tkinter render $\left(x+\frac{w}{2}, -y+\frac{h}{2}\right)$ on its canvas.

When I project coordinates onto the picture plane, because the picture plane is behind the viewing point, the signs of the images on the picture plane are actually flipped. This is similar to how images are flipped on the human retina. As with how the human mind must flip these images in visual processing, I must also flip these coordinates:

$$P(x,y) \to P'(-x,-y) \qquad [15.2]$$

The final result, upon combining [15.1] and [15.2], is:

$$P(x,y) \to P'\left(-x+\frac{w}{2}, y+\frac{h}{2}\right) \qquad [15.3]$$