3D Projection and Rendering

Maths

gzn364

Abstract

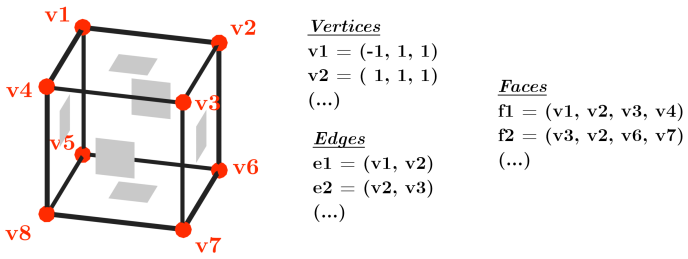
In this investigation, I explore how I can use matrices and vectors to project 3D objects onto 2D surfaces, in order to visualize and render virtual 3D models. I create a Python program within Blender to test these methods.

Contents

1 Introduction

Three-dimensional objects can be modelled mathematically using vertices (points in virtual space), which are connected with polygons to form solid surfaces. This type of 3D modelling is widely used in engineering, visual effects, and other types of design. Computers are used to store and process the 3D data, as models quickly become very complicated - constituting millions of vertices.

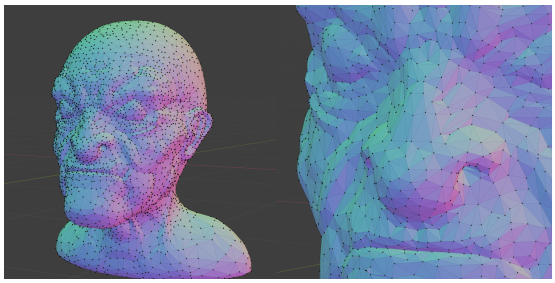
For example, a cube can be modelled as its 8 vertices, 12 edges, and 6 faces:



Humans perceive the world in two dimensions. Light bounces off the surfaces of physical objects and is focused onto the back of the eye, forming a 2D image – a projection of 3D reality - from which the brain infers 3D structure. Therefore, in order for humans to interact with and manipulate 3D models, the object data needs to be presented in two dimensions (on a screen or a sheet of paper, for example), such that we can infer the model's 3D structure. This can be achieved using projection - mapping 3D points onto a 2D plane.

I first became interested in 3D projection through using Blender 3D [2] (See Figure 1). Blender is a 3D modelling and animation program - it allows the user to create and modify 3D objects, and to render them. Rendering involves adding virtual cameras and lighting to a scene of 3D objects, and simulating how that light should interact with the scene - creating an image.

Figure 1: A 3D model I previously made in Blender. $\approx 60,000$ vertices



Mesh geometry (simplified), in Blender

Rendered (Cycles)

3D Print (physical)

I realised that the way the software displayed and rendered 3D objects was very similar mathematically to how the eye (or a camera) produces a 2D image of the world. The images in Figure 1 are *all* 2D projections of the same object! (one from a real camera, the others virtual). This concept was fascinating to me; I decided to further explore how 3D objects can be represented in 2D using projection and rendering.

2 Blender

Blender has several in-built render engines - programs that project the object data into 2D coordinates, and simulate lighting within the scene. (The rendered image in Figure 1 was rendered using Blender's Cycles render engine.) Also, Blender is open-source, meaning that anyone can view and configure the program's source code, or run Python scripts within the program.

I therefore set out to create a rendering engine (a program for displaying and rendering 3D models) within Blender. This would allow me to implement and test my mathematical projection methods, without having to program any interface for positioning test objects. I have included some of the code behind this render engine in Appendix C.

2.1 3D Objects

In Blender, a virtual scene is separated into distinct objects, each with its own type (mesh, camera, light, etc.) and its own properties. Each object has an associated transformation: position, rotation, and scale. These are applied to the object at a local *origin* point.

Mesh objects are objects with geometry; each is associated with a mesh. A mesh consists of a list of vertices - coordinates in a local space, relative to the object origin. The mesh also stores a list of edges (two connected vertices in the vertex list) and faces (multiple connected edges). The world position coordinates of the vertices are transformed according to the object's position, rotation, and scale, around the origin. The world position of each vertex may therefore differ from its local position.

Other object types - cameras and lighting, for example - have no geometry, and only affect how the mesh is rendered. A camera is an object that determines the viewing angle and position. It also determines the projection method used; the main two are perspective and orthographic.

I decided to replicate these two projection methods in my engine, starting with orthographic - the simpler of the two. But first, I needed to define how the object data would be stored and transformed (during projection). I discovered that matrices would be extremely useful for describing transformations, especially rotation.

3 Matrices

A matrix is a rectangular array of numbers, arranged in a grid of rows and columns. A matrix of R rows and C columns is known as an RxC matrix.

An example of a 2x4 matrix:

$$\begin{bmatrix} 5 & 6 & 3 & 3 \\ 3 & 1 & 2 & 7 \end{bmatrix}.$$

Matrices can be added and subtracted from each other, as well as multiplied by scalars (see Appendix A). Matrices can also be multiplied by other matrices. A definition for matrix multiplication can be derived from linear transformations. Translation, rotation, and scaling, can thus all be described using matrices.

After I learnt this way of deriving matrix transformations in a video series by Grant Sanderson [1], I was able to derive a lot of other useful maths for my project.

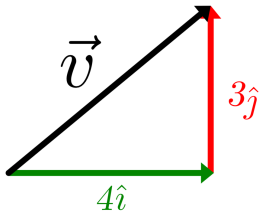
3.1 Linear Transformation

Let $\vec{v}$ be the 2-dimensional vector with x and y components 4 and 3,

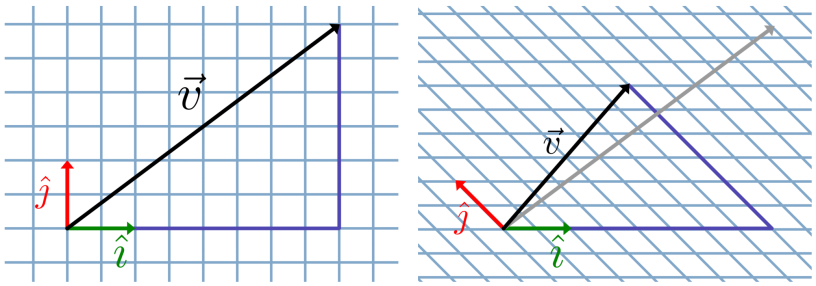
$$\vec{v} = \begin{pmatrix} 4 \\ 3 \end{pmatrix}.$$

This vector can also be written as a sum of the 2D basis vectors, $\hat{\imath}$ and $\hat{\jmath}$:

$$\vec{v} = 4\hat{\imath} + 3\hat{\jmath}$$



Therefore, by altering the basis vectors, $\hat{\imath}$ and $\hat{\jmath}$, we can change the direction and magnitude of $\vec{v}$.



All vectors within this 2D vector space can be written in terms the basis vectors. Therefore changing the basis vectors transforms the vector space. This operation is known as a *linear transformation*, and has the following properties:

1. All straight lines remain straight (linear).
2. The position of the origin is unchanged.

Consider how $\vec{v}$ is transformed when the basis vectors are changed to:

$$\hat{\imath} = \begin{pmatrix} \hat{\imath}_x \\ \hat{\imath}_y \end{pmatrix}, \quad \hat{\jmath} = \begin{pmatrix} \hat{\jmath}_x \\ \hat{\jmath}_y \end{pmatrix}.$$

Then,

$$\begin{aligned}\vec{v} &= 4\hat{\imath} + 3\hat{\jmath} \\ &= 4\begin{pmatrix}\hat{\imath}_x \\ \hat{\imath}_y\end{pmatrix} + 3\begin{pmatrix}\hat{\jmath}_x \\ \hat{\jmath}_y\end{pmatrix} \\ &= \begin{pmatrix}4\hat{\imath}_x + 3\hat{\jmath}_x \\ 4\hat{\imath}_y + 3\hat{\jmath}_y\end{pmatrix}\end{aligned}$$

Or, more generally,

$$\vec{v} = \begin{pmatrix}\vec{v}_x\hat{\imath}_x + \vec{v}_y\hat{\jmath}_x \\ \vec{v}_x\hat{\imath}_y + \vec{v}_y\hat{\jmath}_y\end{pmatrix}$$

Because the basis vectors are sufficient to describe the transformation, they are grouped together in a 2x2 *transformation matrix*, M.

$$M = \begin{bmatrix}\hat{\imath}_x & \hat{\jmath}_x \\ \hat{\imath}_y & \hat{\jmath}_y\end{bmatrix}$$

Also, the vector, $\vec{v}$ can be described using a 2x1 matrix, V:

$$V = \begin{bmatrix}\vec{v}_x \\ \vec{v}_y\end{bmatrix}$$

If R is the resultant vector in 2x1 matrix form,

$$R = \begin{bmatrix}R_x \\ R_y\end{bmatrix} = \begin{bmatrix}\vec{v}_x\hat{\imath}_x + \vec{v}_y\hat{\jmath}_x \\ \vec{v}_x\hat{\imath}_y + \vec{v}_y\hat{\jmath}_y\end{bmatrix}.$$

Then, if we define the transformation of M on V as the multiplication of the two matrices,

$$R = M(V) = \begin{bmatrix}\hat{\imath}_x & \hat{\jmath}_x \\ \hat{\imath}_y & \hat{\jmath}_y\end{bmatrix}\begin{bmatrix}\vec{v}_x \\ \vec{v}_y\end{bmatrix} = \begin{bmatrix}\vec{v}_x\hat{\imath}_x + \vec{v}_y\hat{\jmath}_x \\ \vec{v}_x\hat{\imath}_y + \vec{v}_y\hat{\jmath}_y\end{bmatrix}$$

Notice, in the resultant matrix, the element at row r and column c is equal to the dot product of the r^{th} row of M and the c^{th} column of V. (For example, the x component in the result is the sum of the x component of each basis vector multiplied by its corresponding coefficient in $\vec{v}$).

This is how matrix multiplication is defined for any dimension of matrix. However, clearly the number of columns in M (length of each row) must be equal to the number of rows (length of each column) in V. Otherwise, it would not be possible to perform the dot product of a row in M and a column in V.

For example, it is possible to multiply a 4x5 matrix by a 5x2 matrix, but it is *not* possible to multiply a 5x2 matrix by a 4x5 matrix.

Also, the number of rows in M and the columns of rows in V determine the dimensions of the resultant matrix; multiplying a 4x5 matrix by a 5x2 matrix results in a 4x2 matrix.

These matrix transformations can be performed in vector spaces of any dimension, including in 3 dimensions.

Let $\vec{v}$ now be a *3*-dimensional vector,

$$\begin{aligned}\vec{v} &= \begin{pmatrix}a \\ b \\ c\end{pmatrix}. \\ &= a\hat{\imath} + b\hat{\jmath} + c\hat{k} \\ &= \begin{pmatrix}a\hat{\imath}_x + b\hat{\jmath}_x + c\hat{k}_x \\ a\hat{\imath}_y + b\hat{\jmath}_y + c\hat{k}_y \\ a\hat{\imath}_z + b\hat{\jmath}_z + c\hat{k}_z\end{pmatrix}\end{aligned}$$

Let V be the vector $\vec{v}$ in 3x1 matrix form, M the transformation matrix, and R the resultant vector. Then,

$$R = \begin{bmatrix} a\hat{\imath}_x + b\hat{\jmath}_x + c\hat{k}_x \\ a\hat{\imath}_y + b\hat{\jmath}_y + c\hat{k}_y \\ a\hat{\imath}_z + b\hat{\jmath}_z + c\hat{k}_z \end{bmatrix} = \begin{bmatrix} \hat{\imath}_x & \hat{\jmath}_x & \hat{k}_x \\ \hat{\imath}_y & \hat{\jmath}_y & \hat{k}_y \\ \hat{\imath}_z & \hat{\jmath}_z & \hat{k}_z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = M(V)$$

For example, the matrix, $\begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, performs a scale by 3 along the x axis.

See Appendix A for more properties of matrix multiplication.

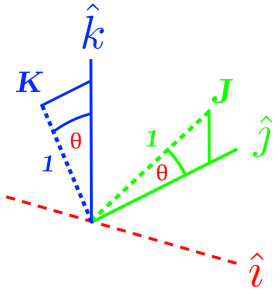
3.2 Rotation Matrices

If 3D vectors are stored as 3x1 matrices in my program, I will be able to transform them by multiplying by transformation matrices. For example, I will need to transform each vertex by the local transformation of the parent object.

Scaling along the x, y, and z axis individually can easily be achieved with a transformation matrix, as I demonstrated above. A 3x3 transformation matrix cannot perform translation because a linear transformation cannot move the origin (although transformation matrices *can* be derived for homogeneous coordinates). However, translation can be achieved by adding the point vector and the translation vector.

Rotation around the origin can be achieved using a 3x3 transformation matrix. Lets first consider how the elementary rotations (around a single axis) will affect the basis vectors.

For a rotation of θ anti-clockwise around the x ($\hat{\imath}$) axis, the $\hat{\jmath}$ and $\hat{k}$ basis vectors will be affected:



Let I, J, and K be the new basis vectors. $\hat{\imath}$ is unchanged, as are $\hat{\jmath}_x$ and $\hat{k}_x$ (both 0). Therefore,

$$I = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

Now, the modulus of the basis vectors is 1. Therefore,

$$\begin{aligned}\cos\theta &= \frac{J_y}{1}\\ J_y &= \cos\theta\\ J_z &= \sin\theta\\ J &= \begin{pmatrix}0\\ \cos\theta\\ \sin\theta\end{pmatrix}\end{aligned}$$

$$\begin{aligned}K_y &= -\sin\theta\\ K_z &= \cos\theta\\ K &= \begin{pmatrix}0\\ -\sin\theta\\ \cos\theta\end{pmatrix}\end{aligned}$$

Hence, the transformation matrix, $R_x(\theta)$, for a rotation of θ anti-clockwise around the x axis, is given as:

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

Similarly, I determined the rotation matrices for elementary rotations around the y and z axis.

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

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

The rotation matrix for a rotation of α in the x axis, β in the y axis, and γ in the z axis, will be the three elementary rotation matrices multiplied together.

$$\begin{aligned}R_{xyz}(\alpha,\beta,\gamma) &= R_z(\gamma)R_y(\beta)R_x(\alpha)\\ &= \begin{bmatrix}\cos\gamma & -\sin\gamma & 0\\ \sin\gamma & \cos\gamma & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos\beta & 0 & \sin\beta\\ 0 & 1 & 0\\ -\sin\beta & 0 & \cos\beta\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\alpha & -\sin\alpha\\ 0 & \sin\alpha & \cos\alpha\end{bmatrix}\\ &= \begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos\beta & \sin\alpha\sin\beta & \cos\alpha\sin\beta\\ 0 & \cos\alpha & -\sin\alpha\\ -\sin\beta & \sin\alpha\cos\beta & \cos\alpha\cos\beta\end{bmatrix}\\ &= \begin{bmatrix}\cos\beta\cos\gamma & \sin\alpha\sin\beta\cos\gamma-\cos\alpha\sin\gamma & \cos\alpha\sin\beta\cos\gamma+\sin\alpha\sin\gamma\\ \cos\beta\sin\gamma & \sin\alpha\sin\beta\sin\gamma+\cos\alpha\cos\gamma & \cos\alpha\sin\beta\sin\gamma-\sin\alpha\cos\gamma\\ -\sin\beta & \sin\alpha\cos\beta & \cos\alpha\cos\beta\end{bmatrix}\end{aligned}$$

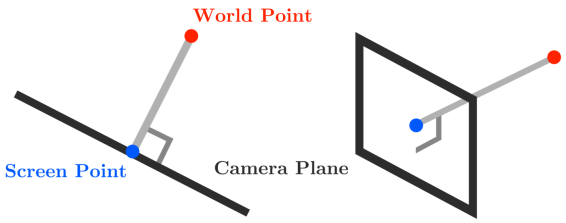
The order in which the rotation is applied is very important. Matrix multiplication is not commutative, so the order 'XYZ' will give a completely different overall rotation matrix to 'ZYX'.

Note, programming the enormous rotation matrix above wasn't necessary; I programmed the elementary rotations, as well as matrix multiplication functionality. This allowed me to multiply them together in any order.

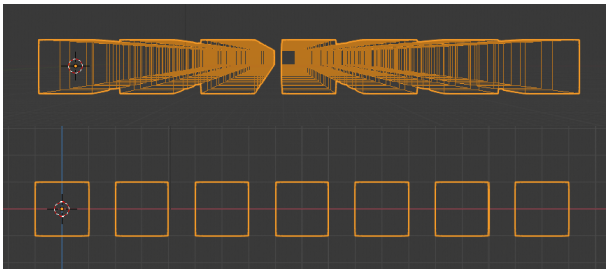
Armed with the ability to rotate, I was ready to tackle orthographic projection.

4 Orthographic Camera

An orthographic camera is one which projects world points onto its plane orthogonally. That is, each world point is projected onto the camera plane such that the line between the point and the projected point is perpendicular to the plane, and parallel to the normal of the plane.



Orthographic cameras are useful tools for 3D design, as unlike perspective cameras, object size does not diminish with distance from the camera (see below).



Perspective (top) vs orthographic (bottom) view of same scene in Blender 3D

My first attempt to create a functioning orthographic camera involved solving a simultaneous equation - the intersection of a plane in vector form and a line in vector form. I later devised a simpler method that uses matrix transformations, but I have included the original method in Appendix B. Although the method is more convoluted, it helps to understand what orthographic projection is.

4.1 Matrix Method

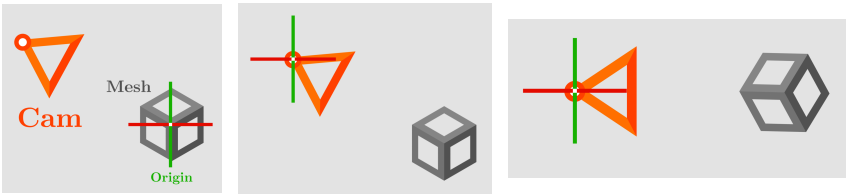
Firstly, each world point, P, needs to be translated such that the camera is at the origin, thereby allowing rotation around the camera. This can be achieved by subtracting the camera position from the point position.

Now that the camera is at the origin, P can be rotated by the inverse of the camera's rotation. I stored the camera rotation as an 'XYZ' order rotation. Therefore, the inverse rotation would be -1 multiplied by each component, and applied in reverse order: 'ZYX'. The overall rotation matrix, R, is:

$$R = R_x(-\theta_x)R_y(-\theta_y)R_z(-\theta_z),$$
where the camera global rotation is $(\theta_x, \theta_y, \theta_z)$.

If the camera plane were subjected to these transformations, it would now be facing downwards (the rotation being 0,0,0) - facing the $-z$ direction. The rotation is applied to all world points, so their positions relative to the camera have been preserved. The normal to the camera is now the z axis, so applying a scaling matrix of 0 along the z axis projects the points onto the camera plane (see Figure 2)

Figure 2: Three stages of my orthographic algorithm, shown in 2D.



Camera in scene | Camera to origin | Reverse rotation

The world points of each vertex also have to be found by applying the parent object transformation.

$$P_{\text{world}} = M_{\text{scale}} \cdot M_{\text{rot}} \cdot (P_{\text{local}} + L),$$

where M_{scale} and M_{rot} are the scaling and rotation matrices for the parent object, and L is the parent object location.

Then, the projected point, P_{proj}:

$$\begin{aligned} P_{\text{proj}} &= R(P_{\text{world}} - L_{\text{cam}}) \\ &= [R_x(-\theta_x)R_y(-\theta_y)R_z(-\theta_z)] \cdot [M_{\text{scale}} \cdot M_{\text{rot}} \cdot (P_{\text{local}} + L) - L_{\text{cam}}] \end{aligned}$$

Note, that this gives the projected point before scaling in the z axis. In the program I do not perform a scale, I simply take the x and y components to be the 2D screen coordinates.

I implemented this method in my Blender render engine. For each edge in the scene, I projected the vertices onto the camera plane and converted these scene coordinates to pixel coordinates, on the render image (using the camera's orthographic scale setting). I then drew a line on the image, connecting the projected vertices. This created a wireframe orthographic effect; it worked perfectly! My rendering engine replicated Blender's built-in orthographic rendering processes perfectly. Figure 3 shows a test comparing rendered images of a teapot object.

Figure 3: A test of my orthographic rendering engine.

Blender Viewport | Blender Cycles | My orthographic engine

At this point, I had already written over 1000 lines of code, so I decided to continue my exploration without updating the program.

4.2 Homogeneous Coordinates

It is often more useful to represent positions in homogeneous coordinates. This involves adding another dimension, w, and multiplying each Cartesian component by its value.

$$\begin{aligned} (x, y, z) &\implies (wx, wy, wz, w) \\ (x, y, z, w) &\implies (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}) \end{aligned}$$

If w is set to 1, homogeneous coordinates can be used to perform translation by matrix multiplication. The homogeneous coordinates of a point (x, y, z) would be $(x, y, z, 1)$. We can therefore achieve a translation of (T_x, T_y, T_z) by multiplying as follows:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{bmatrix}$$

This means, by using homogeneous coordinates, the orthographic projection of a given camera could be calculated as a single matrix multiplication.

Recall the formula I ended up with:

$$P_{\text{proj}} = [R_x(-\theta_x)R_y(-\theta_y)R_z(-\theta_z)] \cdot [M_{\text{scale}} \cdot M_{\text{rot}} \cdot (P_{\text{local}} + L) - L_{\text{cam}}]$$

Now, I'll re-write assuming homogeneous coordinates are used:

$$\begin{aligned} P_{\text{proj}} &= [R_x(-\theta_x)R_y(-\theta_y)R_z(-\theta_z)] \cdot [T_{-L_{\text{cam}}} \cdot M_{\text{scale}} \cdot M_{\text{rot}} \cdot T_L \cdot P_{\text{local}}] \\ &= R_x(-\theta_x) \cdot R_y(-\theta_y) \cdot R_z(-\theta_z) \cdot T_{-L_{\text{cam}}} \cdot P_{\text{world}} \\ &= M(P_{\text{world}}) \\ \text{where,} \quad M &= R_x(-\theta_x) \cdot R_y(-\theta_y) \cdot R_z(-\theta_z) \cdot T_{-L_{\text{cam}}} \end{aligned}$$

Apparently, using a bounding box (left, right, top, bottom, etc) to represent the orthographic clipping panes (what the camera can see), a much simpler matrix multiplication for orthographic projection, M, can be derived [3]:

$$M = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

I find this extremely interesting, but I have no idea how or why it works. This will definitely be an area for further study.

5 Conclusion

The purpose of my investigation was to explore exactly how 3D objects can be projected onto 2D planes. Using rotation matrices, I devised a method for orthographic projection. I implemented this in a wireframe Blender rendering engine - the program created projected images accurately (compared to the inbuilt rendering engines), suggesting that the mathematics I used was correct.

I discovered that the projections I have studied can be simplified to single matrix multiplications, using homogeneous coordinates. However, I failed to understand how where the simplest of these formulas were derived from.

An extension to my project would therefore be to further investigate the use of homogeneous coordinates for geometric projections.

Also, I haven't been able to address properly perspective projections. This is unfortunate, as my interest was initially sparked by the parallels between projection of the real world, and projections of virtual objects - perspective projections are found throughout nature.

However, I *have* gained a lot of insight into 3D projection by studying orthographic projections.

References

[1] Grant Sanderson. *Linear transformations and matrices — Essence of linear algebra, chapter 3*. 3Blue1Brown, Youtube. Aug. 2016. URL: https://www.youtube.com/watch?v=kYB8IZa5AuE&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&index=3.

[2] The Blender Foundation. *Blender 2.82*. Feb. 2020. URL: https://www.blender.org/.

[3] C.W. Brown. *8.2 - Orthographic Projections*. Learn WebGL. 2015. URL: http://learnwebgl.brown37.net/08_projections/projections_ortho.html.

[4] S. H. Ahn. *Homogeneous Coordinates*. 2005. URL: http://www.songho.ca/math/homogeneous/homogeneous.html.

[5] Khan Academy. *Linear Algebra*. Online lesson series. 2020. URL: https://www.khanacademy.org/math/linear-algebra.

[6] Jay Abramson. *College Algebra*. OpenStax, 2015. ISBN: 978-1-938168-38-3. URL: https://openstax.org/details/books/college-algebra.

Appendices

A More Matrix Properties

A.1 Addition

Matrices of the same dimensions can be added together (or subtracted from each other). The result is a matrix of the same dimensions, where each element is the sum of the corresponding elements in each matrix. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 7 & 3 & 5 \end{bmatrix} + \begin{bmatrix} 8 & 3 & 9 \\ 5 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 9 & 5 & 12 \\ 12 & 4 & 7 \end{bmatrix}$$

Matrices can also be multiplied by scalars - by multiplying each element in the matrix by the scalar.

$$3 \begin{bmatrix} 1 & 2 & 3 \\ 7 & 3 & 5 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 9 \\ 21 & 9 & 15 \end{bmatrix}$$

These definitions are consistent with the ideas of adding vectors together by adding the corresponding components, and of scaling vectors by multiplying each component by the scalar.

A.2 Matrix Multiplication

What are the properties of matrix multiplication? Let:

- A be the matrix for a 2D rotation of 90° clockwise.
- B be the matrix for a 2D scaling of -2 in the x axis.
- C be the matrix form of some 2D vector.

A.2.1 Associativity

We apply the two transformations to C; B first, then A. The resulting matrix, R is:

$$R = A(B(C))$$

The overall transformation here would be AB. This is evident when you consider that A and B each represent changes in basis vectors. The basis vectors of $B(C)$ are the columns of B. After applying A

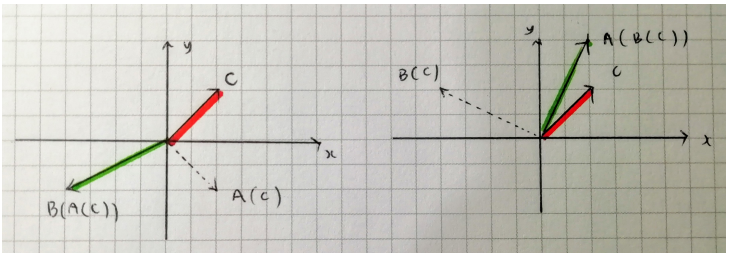
to $B(C)$, these basis vectors will also be transformed by A. Hence, the basis vectors of $A(B(C))$ would be the columns of $A(B)$, and the overall transformation matrix would be $A(B)$. Therefore,

$$\boxed{A(BC) = (AB)C}$$

Matrix multiplication is associative.

A.2.2 Commutativity

Now, compare the result when B is applied first to that when A is applied first.



The resulting vector in each case is different. Therefore,

$$AB(C) \neq BA(C)$$

$$\boxed{AB \neq BA}$$

Matrix multiplication is *not* commutative.

A.2.3 Identity

The identity matrix in three dimensions, I, is the transformation matrix that maps every point, P onto itself (no transformation), such that:

$$I(P) = P.$$

For such a transformation, the basis vectors will be unchanged:

$$\hat{\imath} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \hat{\jmath} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \hat{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

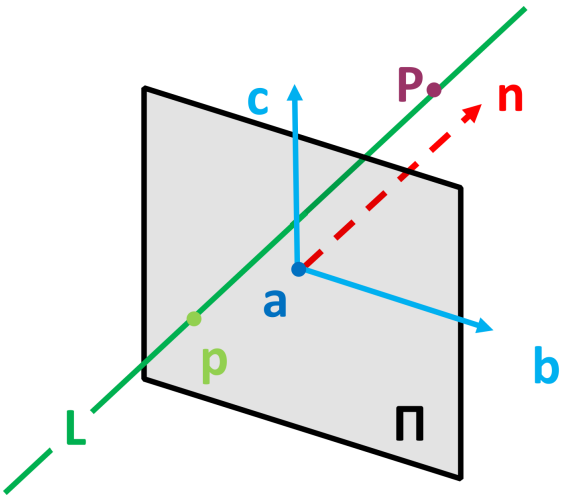
$$\therefore \quad I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The identity matrix can be derived in any dimensions using the fact that the basis vectors remain unchanged.

B Alternative Orthographic Projection

Orthogonal projection is where points in 3-dimensional space are projected onto a 2D plane (a screen) along a line parallel to the normal of that plane.

Consider the following diagram:



The projection plane, Π, can be described as:

$$\Pi : r = a + \lambda b + \mu c,$$

where a is the location of the plane, and b and c are perpendicular vectors in the plane. This parametric form will be most useful in this case, as it will allow simple translation of points on the plane into pixel coordinates on a screen.

Also shown in the diagram above, P denotes the 3D point to be projected, and L a line intersecting point P, and which is parallel to n: the normal vector of plane Π:

$$L : r = P + \gamma n$$

Point p is thus the projection of point P onto the plane – the intersection of L with the plane. I will now attempt to find p as a function of P (and the parameters of the plane, a, b, c, and n).

Where L intersects plane Π:

$$\begin{aligned} r_L &= r_\Pi \\ a + \lambda b + \mu c &= P + \gamma n \\ \lambda b + \mu c - \gamma n &= P - a \end{aligned}$$

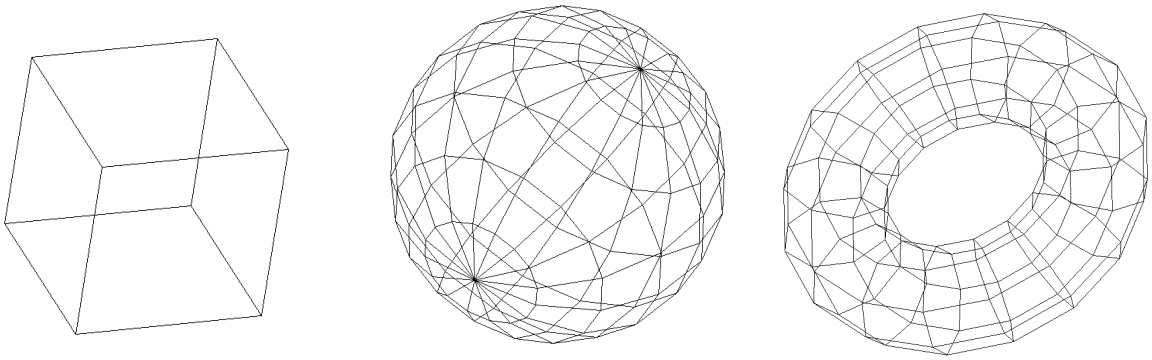
This gives a system of linear equations which can be solved for each project point:

λ	μ	γ	
b_x	c_x	$-n_x$	$P_x - a_x$
b_y	c_y	$-n_y$	$P_y - a_y$
b_z	c_z	$-n_z$	$P_z - a_z$

Although for a given point this projection can be calculated by hand, in the program it is solved by an inbuilt function for linear algebra. The objects are stored as a list of points, and a list of edges each described by the two vertices they connect (faces are also stored, but I was only interested in the objects' wireframes).

The program then sets up the system of equations above for each of the object's points, and solves for the point's screen position. Lines are then drawn on the output image, connecting the screen points that correspond to each edge's vertices.

Below are several screenshots of meshes for which I rendered orthographic projections using the above method.



C Python Program

I created a rendering engine within Blender, to implement my orthographic projection algorithm. In this appendix, I have included a small amount of the most important code - most is omitted.

C.1 objtypes.py

This script contains the classes for my scene objects types. Upon rendering, I converted all objects to this format. **mathematics** is an external script I wrote, containing definitions of Matrix, and Vector3 (which inherits from Matrix).

```
from mathematics import *

class Object3D:
    '''
    Parent object class.
    Rotation must be XYZ order.
    '''
    def __init__(self, name, location = Vector3.zero, rotation = Vector3.zero, scale =
    Vector3.unit):

        self.name = name
        self.location = location
        self.rotation = rotation
        self.scale = scale

    def translate(self, tA):
        self.location += tA
    def rotate(self, rA):
        self.rotation += rA
    def scale(self,v):
        self.scale = self.scale.straight(v)

    def __str__(self):
        return "{} ({})\nLocation : {}\nRotation : {}\nScale : {}\n_______".format(self.
    name,type(self),self.location,self.rotation,self.scale)


class MeshObject(Object3D):
    '''
    Mesh object, inheriting from Object3D
    '''
    def __init__(self,name, location, rotation, scale, mesh, **kwargs):
        super().__init__(name, location, rotation, scale, **kwargs)
        self.mesh = mesh

class Mesh:
    def __init__(self, vertices, edges, faces):
        self.vertices = vertices
        self.edges = edges
        self.faces = faces

    def vertex_local_to_global(self, vCoord, parentObj):
```

```
        '''Converts vertex to global'''
        scale = Vector3.Transform.scaleXYZ(
            *tuple(parentObj.scale)
        )
        rot = Vector3.Transform.rotXYZ(
            *tuple(parentObj.rotation)
        )

        out = (rot * scale * vCoord) + parentObj.location
        return out


class Orthographic(Object3D):
    def __init__(self, name, location, rotation, scale, orthoscale, **kwargs):

        super().__init__(name, location, rotation, scale, **kwargs)

        self.orthoscale = orthoscale

    def project(self, vertex):
        '''
        Project vertex position onto camera plane.
        Returns 2D position of projected point.
        '''
        v = copy(vertex)

        rot = Vector3.Transform.rotZYX(
            -self.rotation.z,
            -self.rotation.y,
            -self.rotation.x
        )

        v = rot * (v - self.location)

        return Vector2 (v.x, v.y,)

    def getpixelscale(self, resolution):
        '''returns pixel length of 1 scene unit'''

        return max(resolution) /  self.orthoscale
```

C.2 Vector3

Vector3 was one of the most important classes, as it held most transformation and position information. Vector3 was defined in **mathematics**, I have included it on its own.

```
class Vector3(Matrix):

    def __init__(self, x = 0, y = 0, z = 0):
        super().__init__([[x, y, z]])
    def __str__(self):
        return "v({}, {}, {})".format(self.x, self.y, self.z)
    def __iter__(self):
        yield self.x
        yield self.y
        yield self.z
    @property
    def x(self):
        return self.data[0][0]
    @x.setter
    def x(self, value):
        self.data[0][0] = value
        return

    @property
    def y(self):
        return self.data[0][1]
    @y.setter
    def y(self, value):
        self.data[0][1] = value
        return
```

```
26
27      @property
28      def z(self):
29          return self.data[0][2]
30      @z.setter
31      def z(self, value):
32          self.data[0][2] = value
33          return
34
35      def ToMatrix(self):
36          return Matrix(self.data)
37      def cross(self, v):
38          pass
39      def dot(self, v):
40          if not isinstance(v, Vector3):
41              raise TypeError("Cannot calculate dot product of Vector3 and {}".format(type
    (v)))
42
43          a = self[0]
44          b= v[0]
45
46          return NumberArray.Dot(a, b)
47      def straight(self, v):
48          if not isinstance(v, Vector3):
49              raise TypeError("Cannot calculate straight product of Vector3 and {}".format
    (type(v)))
50
51          out = copy(self)
52          out.x *= v.x
53          out.y *= v.y
54          out.z *= v.z
55
56          return out
57
58      @property
59      def magnitude(self):
60          return sqrt((self.x ** 2)+(self.y ** 2)+(self.z ** 2))
61      @magnitude.setter
62      def magnitude(self, value):
63          scale = value / magnitude
64          #self *= scale
65          return
66
67      #standard vectors
68      @property
69      def unit(self):
70          return Vector3(1,1,1)
71      @property
72      def zero(self):
73          return Vector3(self)
74      @property
75      def left(self):
76          return Vector3(-1,0,0)
77      @property
78      def right(self):
79          return Vector3(1,0,0)
80      @property
81      def forward(self):
82          return Vector3(0,1,0)
83      @property
84      def backward(self):
85          return Vector3(0,-1,0)
86      @property
87      def up(self):
88          return Vector3(0,0,1)
89      @property
90      def down(self):
91          return Vector3(0,0,-1)
92
93      @staticmethod
94      def FromMatrix(matrix):
95          out = Vector3()
96          out.data = matrix.data
```

```
97          return out
98
99      class Transform:
100         @staticmethod
101         def rotX(t):
102             '''
103             Elementary rotation of theta (t)
104             around the x axis (negative clockwise)
105             '''
106             return Matrix.FromRows(
107                 [
108                     [1, 0, 0],
109                     [0, cos(t), -sin(t)],
110                     [0, sin(t), cos(t)]
111                 ]
112             )
113         @staticmethod
114         def rotY(t):
115             '''
116             Elementary rotation of theta (t)
117             around the y axis (negative clockwise)
118             '''
119             return Matrix.FromRows(
120                 [
121                     [cos(t), 0, sin(t)],
122                     [0, 1, 0],
123                     [-sin(t), 0, cos(t)]
124                 ]
125             )
126         @staticmethod
127         def rotZ(t):
128             '''
129             Elementary rotation of theta (t)
130             around the z axis (positive clockwise)
131             '''
132             return Matrix.FromRows(
133                 [
134                     [cos(t), -sin(t), 0],
135                     [sin(t), cos(t), 0],
136                     [0, 0, 1]
137                 ]
138             )
139
140         @staticmethod
141         def rotXYZ(x,y,z):
142             '''
143             Euler rotation of x,y,z around respective axis.
144             Order: XYZ
145             '''
146             Rx = Vector3.Transform.rotX(x)
147             Ry = Vector3.Transform.rotY(y)
148             Rz = Vector3.Transform.rotZ(z)
149
150             return Rz * Ry * Rx
151
152         @staticmethod
153         def rotZYX(z,y,x):
154             '''
155             Euler rotation of x,y,z around respective axis.
156             Order: ZYX
157             '''
158             Rx = Vector3.Transform.rotX(x)
159             Ry = Vector3.Transform.rotY(y)
160             Rz = Vector3.Transform.rotZ(z)
161
162             return Rx * Ry * Rz
163         @staticmethod
164         def scaleXYZ(x,y,z):
165             '''
166             Performs a scale of x, y, and z in the respective axis.
167             '''
168             return Matrix.FromRows(
169                 [
```

```
                    [x, 0, 0],
                    [0, y, 0],
                    [0, 0, z],
                ]
            )
```

C.3 RenderiaEngine

RenderiaEngine is a class within the main Blender script that defines the render engine, and what happens when a render is called. Again, I have included it on its own.

```
class RenderiaEngine(bpy.types.RenderEngine):
    # These three members are used by blender to set up the
    # RenderEngine; define its internal name, visible name and capabilities.
    bl_idname = "RENDERIA"
    bl_label = "Renderia"
    bl_use_preview = False

    # Init is called whenever a new render engine instance is created. Multiple
    # instances may exist at the same time, for example for a viewport and final
    # render.
    def __init__(self):
        self.scene_data = None
        self.draw_data = None

    # This is the method called by Blender for both final renders (F12) and
    # small preview for materials, world and lights.
    def render(self, depsgraph):

        scene = depsgraph.scene
        settings = scene.renderiasettings

        scale = scene.render.resolution_percentage / 100.0
        self.size_x = int(scene.render.resolution_x * scale)
        self.size_y = int(scene.render.resolution_y * scale)
        self.is_rendering = True
        centre = Vector2(
            int(self.size_x / 2),
            int(self.size_y / 2)
            )


        # has to be called to update the frame on exporting animations
        scene.frame_set(scene.frame_current)

        #convert to renderia format
        objects = bpyconvert.ConvertObjects(scene)
        camera = bpyconvert.ConvertCamera(scene.camera)

        pixelscale = camera.getpixelscale((self.size_x, self.size_y,))
        # Fill the render result with a flat color. The framebuffer is
        # defined as a list of pixels, each pixel itself being a list of
        # R,G,B,A values.
        bgcolor = settings.bgcolour
        linecolor = settings.wirecolour

        pixel_count = self.size_x * self.size_y
        rect = [bgcolor] * pixel_count


        #calculate total vertices
        #this is to accurately update render progress
        totalEdges = 0
        for obj in objects:
            if isinstance(obj, objtypes.MeshObject):
                totalEdges += len(obj.mesh.edges)


        progress = 0
        for obj in objects:
            if isinstance(obj, objtypes.MeshObject):
```

```
                mesh = obj.mesh
                for edge in mesh.edges:
                    self.update_progress(progress / totalEdges)

                    #get start and end vertices (Vector3)
                    v1, v2 = mesh.vertices[edge[0]], mesh.vertices[edge[1]]
                    #get global vertex positions
                    v1 = mesh.vertex_local_to_global(v1, obj)
                    v2 = mesh.vertex_local_to_global(v2, obj)

                    #project vertex points (now Vector2)
                    p1 = camera.project(v1) * pixelscale + centre
                    p2 = camera.project(v2) * pixelscale + centre

                    #draw line between points

                    for x,y in bresenham.bresenham(*p1.ToInt(), *p2.ToInt()):
                        if (0 <= x <= self.size_x) and (0 <= y <= self.size_y):
                            try:
                                rect[y * self.size_x + x] = linecolor
                            except:
                                continue

                    progress += 1


        # Here we write the pixel values to the RenderResult
        result = self.begin_result(0, 0, self.size_x, self.size_y)
        layer = result.layers[0].passes["Combined"]
        layer.rect = rect
        self.end_result(result)
```