THE GEOMETRIC PIPELINE

Notes for an Undergraduate Course in Computer Graphics

University of Minho
António Ramires

V0.55
2021-03-04

1 Introduction

The geometric pipeline deals with all the operations required to determine the screen coordinates of each and every vertex that defines the triangles which make up a 3D model. This document covers the maths behind this multi stage process. It starts by introducing some ground material, such as points and vectors, to establish the bases for the core of the chapter: geometric transformations.

Then it describes the camera, with its intrinsic and extrinsic parameters, as well as the operations regarding the placement of the camera and the required projection to obtain a 2D image. To conclude, an overview of the geometric pipeline is presented, and details about the depth buffer are provided.

2 Coordinate Systems

As far as we are concerned we will consider a 3D coordinate system as a set of three axes and an origin. Furthermore, unless stated otherwise, we will assume that all pairs of axis are perpendicular, as in common Cartesian coordinate systems.

There are two types of coordinate systems: left and right handed. The main difference has to do with the relative orientation of the Z axis. When curling our fingers from the X axis towards the Y axis, the thumb provides the direction of the Z axis. If we use the right hand to determine the direction of the Z axis then we have a right hand system, otherwise we have a left hand system.

In the computer graphics community it is common to use a right hand system where X points to the right, Y points upwards, and Z has its direction defined by the right hand rule. A left hand system has the Z axis pointing in the opposite direction. Both systems are shown in Figure 2.1.

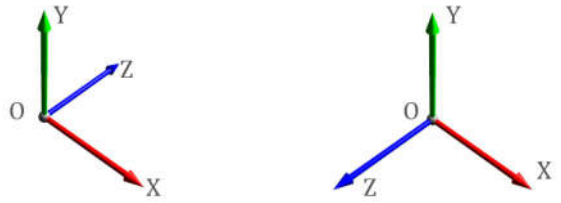

Figure 2.1 - Left and Right handed systems

To define points and vectors in a coordinate system, we're going to used axis aligned unit vectors, or versors, as shown in Figure 2.2.

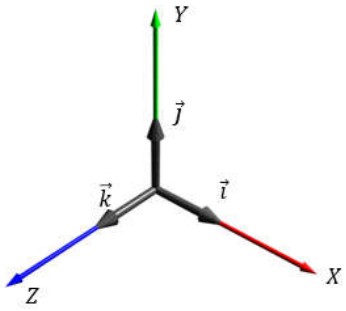

Figure 2.2 - Versors

3 Vectors

Vectors are entities with two properties: direction and magnitude. Figure 3.1 shows a set of vectors.

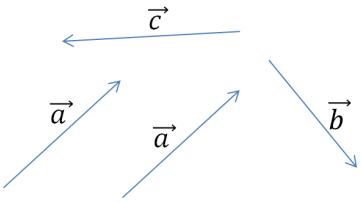

Figure 3.1 - Vectors

Vectors are commonly represented as arrows, and their starting point is not relevant to the vector definition. Vectors are not tied to a location; they are "free" to move around as long as they preserve their direction and magnitude. Vector $\vec{a}$ is represented twice in Figure 3.1 to stress this idea.

Some basic operations on vectors include addition, subtraction and scaling.

3.1 Basic Vector operations

Consider vectors $\vec{b}$ and $\vec{c}$, represented in Figure 3.2.

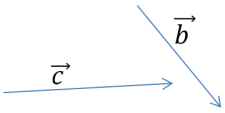

Figure 3.2 - Two vectors

Vectors can be added together to create a new vector (eq. 1).

$$\vec{a} = \vec{b} + \vec{c} = \vec{c} + \vec{b} \tag{1}$$

Graphically this operation is presented in Figure 3.3. The addition is represented graphically by placing the tail of one vector on the tip of the other vector, and then drawing a vector from the tail of the first vector to the tip of the second vector.

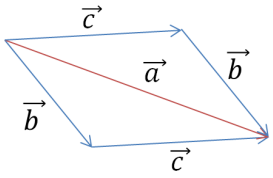

Figure 3.3 - Vector sum

This is known as the parallelogram rule. As can be seen from Figure 3.3Figure 2.1, vector addition is commutative.

Scaling is another simple operation. Scaling changes magnitude, and, when the scale factor is negative, it can reverse the direction of a vector.

$$\vec{a} = k\vec{b}, k \in \mathbb{R} \quad (2)$$

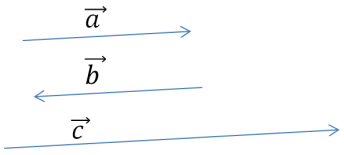

Figure 3.4 - Vector scaling

In Figure 3.4, we have $\vec{b} = -\vec{a}$ and $\vec{c} = 2\vec{a}$.

Vector subtraction can be viewed as vector addition with negative scaling, i.e.,

$$\vec{a} = \vec{b} - \vec{c} = \vec{b} + (-\vec{c}) \quad (3)$$

Graphically, we can place the tip of vector $\vec{c}$ on the tip of vector $\vec{b}$, and then draw a vector from the tail of $\vec{b}$ to the tail of $\vec{c}$. A simpler way of performing vector subtraction is to consider the vector $-\vec{c}$ and proceed as we did for vector sum. Figure 3.5 shows the two approaches. The end result, vector $\vec{a}$, is the same as expected.

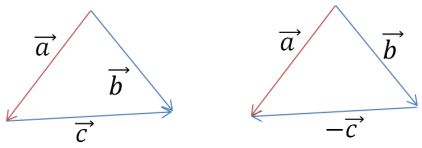

Figure 3.5 - Vector subtraction: left: $\vec{a} = \vec{b} - \vec{c}$; right: $\vec{a} = \vec{b} + (-\vec{c})$

A set of vectors with addition and scalar multiplication operations is called a vector space.

All vectors in a coordinate system can be defined based on the versors of a coordinate system. An example is shown in Figure 3.6. Vector $\vec{v}$ can be decomposed in its horizontal and vertical components as $\vec{v} = \vec{\imath} + 2\vec{\jmath}$. We will use the notation $\vec{v} = (v_x, v_y, v_z)$.

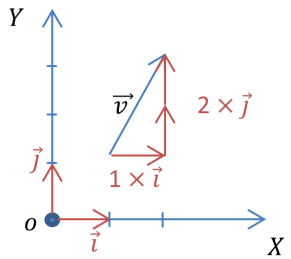

Figure 3.6 – Vector in a coordinate system

Considering a vector $\vec{a} = a_x\vec{\imath} + a_y\vec{\jmath} + a_z\vec{k}$, vector length is defined as:

$$|\vec{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2} \tag{4}$$

A vector whose length is one is called a unit vector. Creating a unit length vector for an arbitrary direction $\vec{b}$ is called normalization. To normalize a vector first compute its length and then divide each of its components by the length.

$$\overrightarrow{b_N} = \frac{\vec{b}}{|\vec{b}|} = \left(\frac{b_x}{|\vec{b}|}, \frac{b_y}{|\vec{b}|}, \frac{b_z}{|\vec{b}|}\right) \tag{5}$$

3.2 Projection

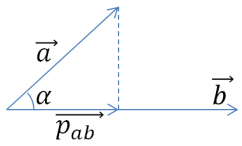

Figure 3.7 – Vector projection

The vector $\overrightarrow{p_{ab}}$ in Figure 3.7 is called the projection of $\vec{a}$ onto $\vec{b}$. The norm of $\overrightarrow{p_{ab}}$ is given by:

$$|\overrightarrow{p_{ab}}| = |\vec{a}|\cos(\alpha) \tag{6}$$

And the vector itself can be defined as a multiple of the normalized vector $\vec{b}$.

$$\overrightarrow{p_{ab}} = |\vec{a}|\cos(\alpha) \times \frac{\vec{b}}{|\vec{b}|} \tag{7}$$

3.3 Dot Product

The dot product, a.k.a. inner product, is an operation between two vectors that results in a scalar. It is commonly represented with a dot and is defined geometrically as:

$$\vec{a} \cdot \vec{b} = \cos(\alpha)|\vec{a}||\vec{b}| \tag{8}$$

The angle α is the smallest angle formed by the two vectors.

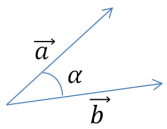

Figure 3.8 – Angle between two vectors

The sign of the dot product is determined by the cosine, hence:

$$\begin{cases} \vec{a} \cdot \vec{b} < 0 \quad if \quad \frac{\Pi}{2} < \alpha \leq \Pi \\ \vec{a} \cdot \vec{b} = 0 \quad if \quad \alpha = \frac{\Pi}{2} \\ \vec{a} \cdot \vec{b} > 0 \quad if \quad 0 \leq \alpha < \frac{\Pi}{2} \end{cases} \tag{9}$$

From eq. 6) and 8) we get:

$$\vec{a} \cdot \vec{b} = \cos(\alpha)\,|\vec{a}||\vec{b}| = |\overrightarrow{p_{ab}}\,||\vec{b}| \tag{10}$$

Assuming that $\vec{b}$ is a unit vector, we get that the dot product is the length of the projected vector. The projected vector can be defined as:

$$\overrightarrow{p_{ab}} = \frac{\vec{a}\cdot\vec{b}}{|\vec{b}|}\vec{b} \tag{11}$$

It is straight forward that

$$(k\vec{a}) \cdot \vec{b} = \cos(\alpha)\,|k\vec{a}|\left|\vec{b}\right| = \mathrm{k}\cos(\alpha)\,|\vec{a}|\left|\vec{b}\right| = k(\vec{a} \cdot \vec{b}) \tag{12}$$

Let's consider now that we want to compute projection of a vector $\vec{v} = \vec{a} + \vec{b}$ onto vector $\vec{u}$, as shown in Figure 3.9

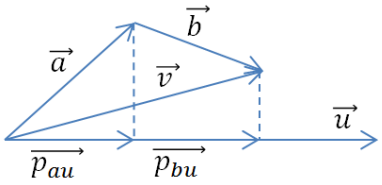

Figure 3.9 – Vector sum projection

As can be seen from Figure 3.9, $\overrightarrow{p_{au}} + \overrightarrow{p_{bu}} = \overrightarrow{p_{vu}}$. Therefore,

$$\overrightarrow{p_{vu}} = \overrightarrow{p_{au}} + \overrightarrow{p_{bu}} \Leftrightarrow \frac{(\vec{a}+\vec{b})\cdot\vec{u}}{|\vec{u}|}\vec{u} = \frac{\vec{a}\cdot\vec{u}}{|\vec{u}|}\vec{u} + \frac{\vec{b}\cdot\vec{u}}{|\vec{u}|}\vec{u} \Leftrightarrow \left(\vec{a} + \vec{b}\right)\cdot\vec{u} = \vec{a}\cdot\vec{u} + \vec{b}\cdot\vec{u} \tag{13}$$

Hence, dot product satisfies the distributive law.

Considering two vectors $\vec{u} = u_x\vec{\imath} + u_y\vec{\jmath} + u_z\vec{k}$ and $\vec{v} = v_x\vec{\imath} + v_y\vec{\jmath} + v_z\vec{k}$, we can write:

$$\vec{u} \cdot \vec{v} = (u_x\vec{\imath} + u_y\vec{\jmath} + u_z\vec{k}) \cdot (v_x\vec{\imath} + v_y\vec{\jmath} + v_z\vec{k}) \tag{14}$$

Applying the distributive law we get:

$$\begin{aligned}\vec{u} \cdot \vec{v} = \quad & u_x v_x(\vec{\imath}\cdot\vec{\imath}) + u_x v_y(\vec{\imath}\cdot\vec{\jmath}) + u_x v_z(\vec{\imath}\cdot\vec{k}) + \\ & u_y v_x(\vec{\jmath}\cdot\vec{\imath}) + u_y v_y(\vec{\jmath}\cdot\vec{\jmath}) + u_y v_z(\vec{\jmath}\cdot\vec{k}) + \\ & u_z v_x(\vec{k}\cdot\vec{\imath}) + u_z v_y(\vec{k}\cdot\vec{\jmath}) + u_z v_z(\vec{k}\cdot\vec{k})\end{aligned} \tag{15}$$

We know, from the dot product definition (see eq. 8) that considering two axis versors $\vec{a}$ and $\vec{b}$:

$$\begin{cases}\vec{a}\cdot\vec{a} = 1 \\ \vec{a}\cdot\vec{b} = 0 \quad if \;\; \vec{a} \neq \vec{b}\end{cases} \tag{16}$$

Therefore eq. 15) simplifies to

$$\vec{u} \cdot \vec{v} = u_x v_x + u_y v_y + u_z v_z \tag{17}$$

Eq. 17), the algebraic definition of the cross product, provides an extremely simple way to compute the dot product. As a consequence, the dot product provides a very easy, and computationally inexpensive way of computing the cosine of two unit vectors:

$$\cos(\alpha) = u_x v_x + u_y v_y + u_z v_z \tag{18}$$

3.4 Cross Product

The cross product is an operation that given two vectors provides a vector which is perpendicular to the plane defined by the two input vectors. The direction of the cross product vector is given by the right hand rule.

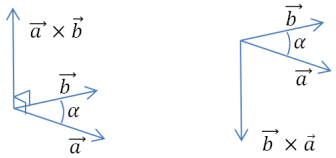

Figure 3.10 – Cross product

The cross product geometric definition is:

$$\vec{a} \times \vec{b} = |\vec{a}||\vec{b}| \sin(\alpha)\, \vec{n} \tag{19}$$

Where $\vec{n}$ is a unit vector perpendicular to the plane defined by vectors $\vec{a}$ and $\vec{b}$, and α is the smallest angle between the two vectors. The length of the cross product vector is given by

$$|\vec{a} \times \vec{b}| = |\vec{a}||\vec{b}| \sin(\alpha) \tag{20}$$

The length is the area of the parallelogram formed by vectors $\vec{a}$ and $\vec{b}$. Considering the parallelogram in Figure 3.11, the area is the length of the base ($|\vec{a}|$) times the height ($|\vec{b}|\sin(\alpha)$).

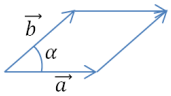

Figure 3.11 – Parallelogram formed by two vectors.

It is straight forward that

$$(k\vec{a}) \times \vec{b} = |k\vec{a}||\vec{b}| \sin(\alpha)\, \vec{n} = k|\vec{a}||\vec{b}| \sin(\alpha)\, \vec{n} = k(\vec{a} \times \vec{b}) \tag{21}$$

When applying the cross product to the versors we get:

$$\begin{cases} \vec{\imath} \times \vec{\imath} = \vec{\jmath} \times \vec{\jmath} = \vec{k} \times \vec{k} = 0 \\ \vec{\imath} \times \vec{\jmath} = \vec{k} \qquad \vec{\jmath} \times \vec{\imath} = -\vec{k} \\ \vec{\imath} \times \vec{k} = -\vec{\jmath} \qquad \vec{k} \times \vec{\imath} = \vec{\jmath} \\ \vec{\jmath} \times \vec{k} = \vec{\imath} \qquad \vec{k} \times \vec{\jmath} = -\vec{\imath} \end{cases} \tag{22}$$

Cross product is distributive over addition. In order to prove this we have to achieve some intermediate results first. Consider a vector $\vec{c}$ and a perpendicular plane P. Now consider an arbitrary vector $\vec{a}$ not parallel to $\vec{c}$. We can compute the projection of $\vec{a}$ onto P as $\overrightarrow{a'}$ (see Figure 3.12).

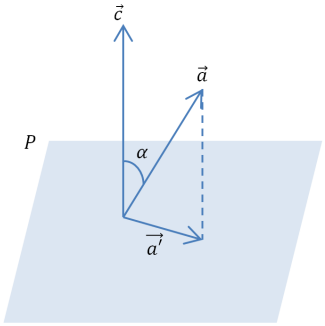

Figure 3.12 – Projecting $\vec{a}$ onto P

We know that

$$\left|\overrightarrow{a'}\right| = \sin(\alpha)\,|\vec{a}| \tag{23}$$

Hence,

$$|\vec{a}\times\vec{c}| = |\vec{a}||\vec{c}|\sin(\alpha) = \left|\overrightarrow{a'}\right||\vec{c}| = \left|\overrightarrow{a'}\right||\vec{c}|\sin(90) = \left|\overrightarrow{a'}\times\vec{c}\right| \tag{24}$$

We also know that both cross products point in the same direction (see Figure 3.13), therefore,

$$\vec{a}\times\vec{c} = \overrightarrow{a'}\times\vec{c} \tag{25}$$

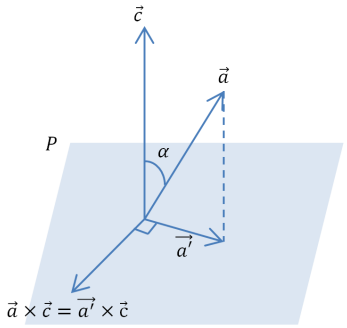

Figure 3.13 - $\vec{a}\times\vec{c} = \overrightarrow{a'}\times\vec{c}$

Let's add another vector $\vec{b}$, also not parallel to $\vec{c}$, and its projection onto P as $\overrightarrow{b'}$. Figure 3.14 shows that the projection of the sum equals the sum of the projections (eq. 120).

$$\left(\vec{a}+\vec{b}\right)' = \overrightarrow{a'}+\overrightarrow{b'} \tag{26}$$

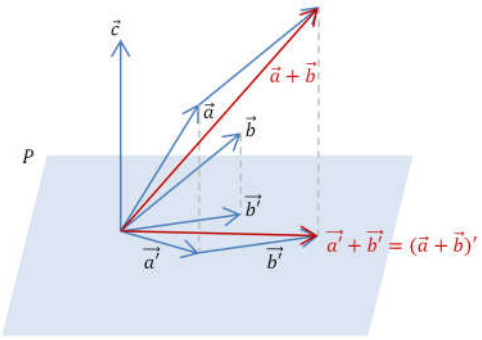

Figure 3.14 - $(\vec{a}+\vec{b})' = \vec{a'}+\vec{b'}$

Figure 3.15 shows a view from the top (vector $\vec{c}$ is perpendicular to the page). Assuming $|\vec{c}| = 1$, the figure shows the resulting cross products with vector $\vec{c}$ up to scale, showing that:

$$(\vec{a'}+\vec{b'}) \times \vec{c} = \vec{a'} \times \vec{c} + \vec{b'} \times \vec{c} \tag{27}$$

Using the result in eq. 25) on the left hand side of eq. 27) we get

$$(\vec{a'}+\vec{b'}) \times \vec{c} = (\vec{a}+\vec{b})' \times \vec{c} = (\vec{a}+\vec{b}) \times \vec{c} \tag{28}$$

And using the result in eq. 25) on the right hand side of eq. 27) we get

$$\vec{a'} \times \vec{c} + \vec{b'} \times \vec{c} = \vec{a} \times \vec{c} + \vec{b} \times \vec{c} \tag{29}$$

Finally, putting eq. 28) and 29) together we get the result we we're looking for: the cross product is distributive.

$$(\vec{a}+\vec{b}) \times \vec{c} = \vec{a} \times \vec{c} + \vec{b} \times \vec{c} \tag{30}$$

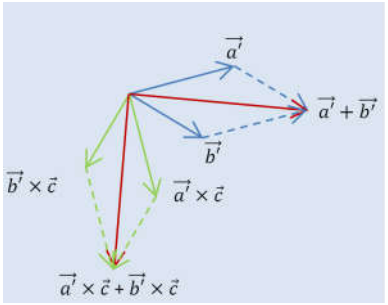

Figure 3.15 – $(\vec{a'}+\vec{b'}) \times \vec{c} = \vec{a'} \times \vec{c} + \vec{b'} \times \vec{c}$

Being distributive allows us to deduce a component formula from the geometric formula presented in eq. 19). Let $\vec{a} = a_x\vec{\imath} + a_y\vec{\jmath} + a_z\vec{k}$ and $\vec{b} = b_x\vec{\imath} + b_y\vec{\jmath} + b_z\vec{k}$, then

$$a \times b = (a_x\vec{\imath} + a_y\vec{\jmath} + a_z\vec{k}) \times (b_x\vec{\imath} + b_y\vec{\jmath} + b_z\vec{k}) \tag{31}$$

$$a \times b = a_x b_x(\vec{\imath} \times \vec{\imath}) + a_x b_y(\vec{\imath} \times \vec{\jmath}) + a_x b_z(\vec{\imath} \times \vec{k}) + a_y b_x(\vec{\jmath} \times \vec{\imath}) + a_y b_y(\vec{\jmath} \times \vec{\jmath}) + a_y b_z(\vec{\jmath} \times \vec{k}) + a_z b_x(\vec{k} \times \vec{\imath}) + a_z b_y(\vec{k} \times \vec{\jmath}) + a_z b_z(\vec{k} \times \vec{k}) \tag{32}$$

Using the results in eq. 22) we get:

$$a \times b = +a_x b_y \vec{k} - a_x b_z \vec{j} - a_y b_x \vec{k} + a_y b_z \vec{i} + a_z b_x \vec{j} - a_z b_y \vec{i} \tag{33}$$

Which can be written as

$$a \times b = (a_y b_z - a_z b_y)\vec{i} + (a_z b_x - a_x b_z)\vec{j} + (a_x b_y - a_y b_x)\vec{k} \tag{34}$$

To compute the components of the cross product vector $\vec{v} = \vec{a} \times \vec{b}$ we can use eq. 35)

$$\begin{aligned} v_x &= a_y b_z - a_z b_y \\ v_y &= a_z b_x - a_x b_z \\ v_z &= a_x b_y - a_y b_x \end{aligned} \tag{35}$$

The rule of Sarrus for computing determinants can be helpful to memorize this formula:

$$\begin{matrix} i & j & k & i & j & k \\ a_x & a_y & a_z & a_x & a_y & a_z \\ b_x & b_y & b_z & b_x & b_y & b_z \end{matrix}$$

For each line, multiply the items in it. Then add all the resulting terms in blue lines and subtract the terms in red lines.

4 Points and Vectors

A point defines a location. New points can be obtained by adding vectors to them (eq. 36). Geometrically this operation is a translation of the original point.

$$p' = p + \vec{v} \tag{36}$$

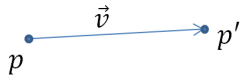

Figure 4.1 - Points and vectors

We say that p was *translated* by $\vec{v}$.

As a consequence of eq. 36) and Figure 4.1, we can find all points p' in the line passing through p and with direction $\vec{v}$ can be found with eq. 37).

$$p' = p + k\vec{v} \tag{37}$$

Vectors can also be defined based on points (eq. 38).

$$\vec{v} = p' - p \tag{38}$$

Note that, as opposed to vectors, it doesn't make sense to talk about point addition or point multiplication by a scalar using the definitions of sum and multiplication given above. However, consider the following:

$$p = p_1 + \vec{u} \tag{39}$$

$$\vec{u} = \alpha\vec{v} \tag{40}$$

$$\vec{v} = p_2 - p_1 \tag{41}$$

Combining eqs. 39), 40) and 41)we can write

$$p = p_1 + \alpha(p_2 - p_1) \tag{42}$$

When $0 \leq \alpha \leq 1$, point p is in the line segment between $p_1$and p_2. The geometric interpretation of this expression, when $0 \leq \alpha \leq 1$, is presented in Figure 4.2.

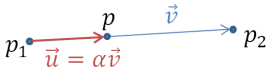

Figure 4.2 - Geometric interpretation of eq. 42)

We can reorganize eq. 42) as follows:

$$p = (1 - \alpha)p_1 + \alpha p_2 \tag{43}$$

Eq. 43) shows a linear combination of points called *affine combination*.

A point p is an affine combination of points $p_1, \ldots, p_n$when

$$p = \alpha_1 p_1 + \cdots + \alpha_n p_n \tag{44}$$

where the scalars $\alpha_1, \ldots, \alpha_n$ are such that

$$\alpha_i \geq 0 \ \wedge \ \alpha_1 + \cdots + \alpha_n = 1$$

Geometrically speaking, if we consider the smallest convex polygon enclosing vertices $p_1, \ldots, p_n$, i.e. the convex hull of the set $\{p_1, \ldots, p_n\}$, then p is inside the convex hull. An example is presented in Figure 4.3. Note that point p_5 is not one of the enclosing polygon's vertices because the polygon must be convex.

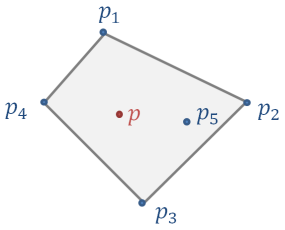

Figure 4.3 - Convex hull and affine combination

5 Homogeneous Coordinates

For simplicity's sake, sometimes 2D graphics will be used to present concepts or definitions, but generalization to 3D is straightforward.

Consider point p in Figure 5.1. How can we describe it, based on the information we have about the coordinate system, and the operations discussed above regarding points and vectors?

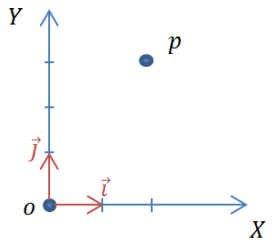

Figure 5.1 - Point in space

The coordinate system has an origin o and two versors: $\vec{\imath}, \vec{j}$. Hence, point p could be written as:

$$p = o + 2\vec{\imath} + 3\vec{j} \tag{45}$$

Note that, without the origin in eq. 45), the right side of the equation would be a vector (vector plus vector = vector), hence, p could not represent a point. It is the introduction of the origin that makes the right side of the equation a point.

Eq. 45) can be written in matrix form as show in eq. 46).

$$p = [\vec{\imath} \quad \vec{j} \quad o] \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \tag{46}$$

Hence, to write the coordinates of a 2D point, three values are required. Note that, commonly, the origin is implicitly assumed. However, in here we're going to need it later so we'll keep it as part of the point definition.

So, what is the result when two points are subtracted to provide a vector? Assume that $p_2 = (2,3,1)$ and $p_1 = (1,2,1)$

$$v = p_2 - p_1 = (2,3,1) - (1,2,1) = (1,1,0) \tag{47}$$

Vectors have the last coordinate as zero, whereas points have their last coordinate as 1. For 2D points, this triplet of values is called the homogeneous coordinates. Given a point with homogeneous coordinates (x, y, w), the Cartesian coordinates are computed as $(\frac{x}{w}, \frac{y}{w})$.

Consider now a point with homogeneous coordinates $(4,6,2)$. The Cartesian coordinates are $(2,3)$. Point p_2 in eq. 47) has the same Cartesian coordinates. In fact, there is an infinite number of points in homogeneous coordinates whose Cartesian coordinates are the same.

For our purposes, given a point with Cartesian coordinates $(x,\ y)$, to obtain the homogeneous coordinates we select the point which has $w = 1$, i.e., the point with homogeneous coordinates $(x,\ y,\ 1)$.

Geometrically this can be interpreted as shown in Figure 5.2. Note that the extra axis has been labelled with W to ease the later transition to 3D. Points p and q are in the plane $w = 1$. Points q and r have the same Cartesian coordinates. To find the Cartesian coordinates of r we project it onto the plane $w = 1$ along the line that connects r to the origin, resulting in point q.

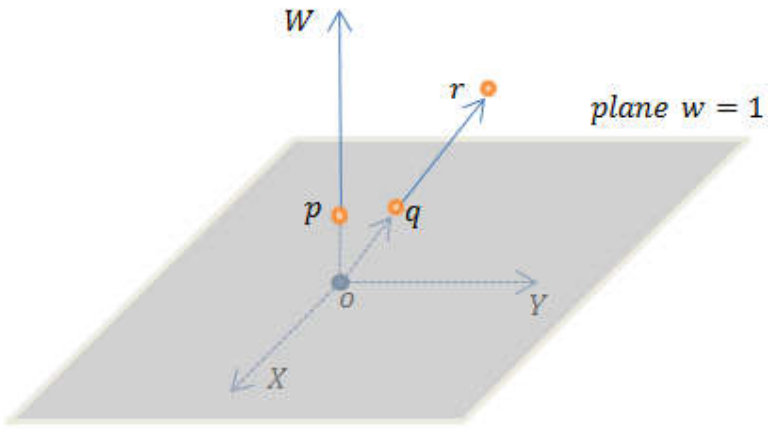

Figure 5.2 - Graphical representation of homogeneous coordinates

The coordinates of the points in Figure 5.2 are as shown in Table 5.1 :

Point	Homogeneous	Cartesian
p	$(0,0,1)$	$(0,0)$
q	$(0.5,1,1)$	$(0.5, 1)$
r	$(1,2,2)$	$(0.5, 1)$

Table 5.1 - Homogeneous and Cartesian coordinates

Graphically, a vector is always drawn as a direction in the plane $w = 0$, and can be referred to as a point in infinity, i.e. the vector would intersect plane $w = 1$ at infinity. Hence, points and vectors are different in homogeneous coordinates. A 2D vector is represent as a triplet where the last element is always 0. 2D points are triplets (x, y, w) where $w \neq 0$.

When the context is clear, points vs. vectors, it is common to drop the extra coordinate, and just use the Cartesian coordinates.

So, why do we need homogeneous coordinates? This will become clear when geometric transformations are presented.

6 Affine Geometric Transformations

Affine geometric transformations preserve straight lines, and distance ratios of points lying in a straight line. Preserving straight lines means that if three points are in a straight line they will continue to be in a straight line after being transformed. Their relative distances to each other will also be preserved. Angles and lengths are not necessarily preserved but parallel lines will remain parallel.

Some of the most common transformations are translation, rotation and scaling. These are used extensively to compose the objects in a 3D scene, and to “place” the camera.

6.1 Translation

Translation moves a point, or a set of points, in a given direction, and length. This will be useful to move objects around, either to build a 3D scene, "place" the camera, or perform animations. Applying a translation to all the points that define a 3D object translates the object itself.

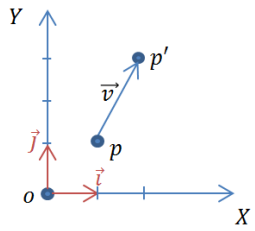

Figure 6.1 - Translation

Graphically, in Figure 6.1, point p is translated using vector $\vec{v}$. Mathematically this can be expressed as in eq. 48)

$$p' = p + \vec{v} \tag{48}$$

Intuition tells us that

$$p' = (p_x + v_x,\ p_y + v_y,\ 1) \tag{49}$$

Checking out if eq. 49) is really what we get is simple. First, decomposing into the coordinate system elements, the versors and the origin, we can write

$$p' = p + \vec{v} = p_x\vec{\imath} + p_y\vec{\jmath} + o + v_x\vec{\imath} + v_y\vec{\jmath} \tag{50}$$

Grouping the last three terms as o'

$$o' = o + v_x\vec{\imath} + v_y\vec{\jmath} \tag{51}$$

We get

$$p' = p_x\vec{\imath} + p_y\vec{\jmath} + o' \tag{52}$$

Rewriting the previous equation in matrix form we get

$$p' = [\vec{\imath} \quad \vec{\jmath} \quad o'] \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \tag{53}$$

Considering that

$$\vec{\imath} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \vec{\jmath} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, o = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{54}$$

Then o' can be written as

$$o' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ 1 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ 1 \end{bmatrix} \tag{55}$$

And

$$p' = \begin{bmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_y \\ 1 + 0 \end{bmatrix} \tag{56}$$

As expected (see eq.49), eq. 56) states that when a point p with homogeneous coordinates $(p_x, p_y, 1)$ is translated with vector $\vec{v}$ with coordinates $(v_x, v_y, 0)$ the resulting point has coordinates $(p_x + v_x, p_y + v_y, 1)$.

An important result is that a 3x3 matrix can be used to translate a Cartesian 2D point. The general form of the matrix is as presented in eq. 56).

Let's see what happens when we apply the same matrix to a vector $\vec{a} = (a_x, a_y, 0)$.

$$\vec{a}' = \begin{bmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ 0 \end{bmatrix} = \begin{bmatrix} a_x \\ a_y \\ 0 \end{bmatrix} \tag{57}$$

As shown in eq. 57) $\vec{a}' = \vec{a}$. This makes perfect sense since vectors are "free", i.e. they are not attached to a location as points are. Vectors are characterized by a direction and length, and none of these properties is affected by translation, hence, applying a translation to a vector results in the same vector.

Moving up from 2D to 3D, the general form of a translation matrix representing a translation by a vector $\vec{v}$ is

$$T = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{58}$$

Given a matrix T, point p in homogeneous coordinates is translated as

$$p' = Tp \tag{59}$$

If we use $\vec{v}$ to build translation matrix T then using $-\vec{v}$ we get the inverse translation T^{-1}.

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -v_x \\ 0 & 1 & 0 & -v_y \\ 0 & 0 & 1 & -v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{60}$$

As expected, the following equation is true

$$TT^{-1} = I \tag{61}$$

6.2 Scale

Scaling is required to deal with models built with different units. For instance, we may want to assemble a scene where one model was built with centimeters as units, whereas another was built using meters. To put these two models together in a scene one of them must be scaled, so that the units match.

Scales are also useful when one wants to use the same model in different situations. For instance a scene may depict a real car and a play car. Both may use the same car model, but scaling will be required to combine these two.

Scaling implies multiplying the coordinates of each point that defines the model by a certain amount. A scale can affect the axis differently, using different scale factors for each, for instance to make a model thinner, without affecting its height. Nevertheless, uniform scales, where the same scale factor is used for all axes, are the most common.

2D Scaling, in Cartesian coordinates can be defined as:

$$p' = (s_x, s_y) \times p = (s_x \times p_x, s_y \times p_y) \tag{62}$$

Geometrically this can be represented as in Figure 6.2 where point p has been *scaled* uniformly by 3 units in all axes. If a line is drawn from the origin passing through point p, point p' will be in the that line, thrice the distance from the origin.

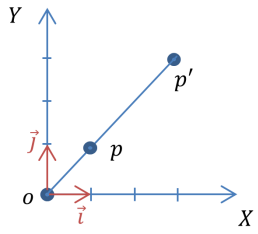

Figure 6.2 - Scaling

The matrix form for this operation is simply a diagonal matrix with the scaling coefficients for each axis along the diagonal.

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{63}$$

Therefore to *scale* a point we can write

$$p' = Sp = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x \times p_x \\ s_y \times p_y \\ 1 \end{bmatrix} \tag{64}$$

Note that, with matrix S no scale factor is applied to the w component of the homogeneous coordinates. To see why this is so, let's suppose that we defined matrix S as follows

$$\mathrm{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \tag{65}$$

where $s_z \neq 1$. We know that $s_z \neq 0$ otherwise the result wouldn't be a point, but what if $s_z = 2$?

$$p' = Sp = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x \times p_x \\ s_y \times p_y \\ 2 \end{bmatrix} \tag{66}$$

Point $p' = (s_x \times p_x, s_y \times p_y, 2)$ in homogeneous coordinates, and

$$\mathrm{p}' = (\frac{\mathrm{s_x} \times \mathrm{p_x}}{2}, \frac{\mathrm{s_y} \times \mathrm{p_y}}{2}) \tag{67}$$

in Cartesian coordinates, which is probably not what we wanted.

In 3D the matrix S is defined as

$$\mathrm{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{68}$$

The inverse operation can be performed with matrix S^{-1}

$$S^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{69}$$

6.3 Rotation about the Main Axes

Rotation is another very useful transformation. Models often require a rotation to be displayed appropriately in a 3D scene.

First, let's consider a simple 2D case where a point is lying on the X axis with Cartesian coordinates $(p_x, 0)$. We want to apply a rotation, centred on the origin, of an angle α (Figure 6.3).

Rotations are performed counter clockwise. This is a situation where actually thinking in 3D makes it easier to see why. In 3D, rotations are performed about a direction called the rotation axis. So, a rotation in the XY plane is performed about the Z axis. The orientation of the rotation can be found using the right hand rule (as we are using right hand coordinate systems). The thumb points along the direction of the rotation axis (Z), and the curling of the fingers will provide the positive rotation direction. So if the thumb points along the Z axis, the fingers will curl from X to Y, hence, it's counter clockwise.

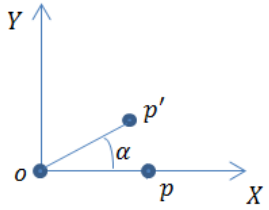

Figure 6.3 - Rotation

Using trigonometry, if $p = (p_x, 0)$ then

$$p' = (p_x \times \cos(\alpha),\ p_x \times \sin(\alpha)) \tag{70}$$

Now let's consider an arbitrary rotation, as shown in Figure 6.4. This time we want to rotate point p' by an angle β and get p''.

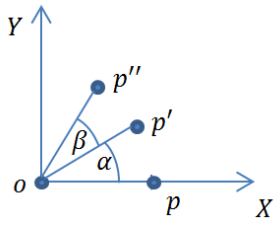

Figure 6.4 – Arbitrary 2D rotation

Again using trigonometry we have

$$p'' = (p_x \times \cos(\alpha + \beta), p_x \times \sin(\alpha + \beta)) \tag{71}$$

We (should) know that

$$\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) \tag{72}$$

$$\sin(\alpha + \beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta) \tag{73}$$

Hence, we can rewrite the X component of p'' in eq. 71) as follows

$$p_x'' = p_x \times (\cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)) \tag{74}$$

$$p_x'' = p_x \times \cos(\alpha)\cos(\beta) - p_x \times \sin(\alpha)\sin(\beta) \tag{75}$$

Using the result in eq. 70) we get

$$p_x'' = p_x' \times \cos(\beta) - p_y' \times \sin(\beta) \tag{76}$$

Solving for p_y'' we get

$$p_y'' = p_x \times \sin(\alpha + \beta) \tag{77}$$

Using eq. 73) we get

$$p_y'' = p_x \times (\sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)) \tag{78}$$

$$p_y'' = p_x \times \sin(\alpha)\cos(\beta) + p_x \times \cos(\alpha)\sin(\beta) \tag{79}$$

Combining with eq. 70) gives us

$$p_y'' = p_x' \times \sin(\beta) + p_y' \times \cos(\beta) \tag{80}$$

Hence, from eq. 76) and eq. 80) we have

$$p'' = (p'_x \times \cos(\beta) - p'_y \times \sin(\beta), \quad p'_x \times \sin(\beta) + p'_y \times \cos(\beta)) \tag{81}$$

In matrix form, and using homogeneous coordinates we get

$$p'' = \begin{bmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p'_x \\ p'_y \\ 1 \end{bmatrix} \tag{82}$$

In 3D, rotation is far more complex since the rotation axis can be arbitrary. First let's consider rotations about each of the main axes: X, Y, and Z.

Although we could find the rotation matrix as we did in the 2D case, we're going to take a shortcut and use the knowledge we got from the 2D case to get there faster.

Rotation around the Z axis is similar to the 2D rotation. We know that the Z and W components will not be affected by the rotation. Therefore the respective rows are as in an identity matrix:

$$\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{83}$$

We also know that the both Z and W components have no part on the new x and y values. Hence, we get only four unknown elements:

$$\begin{bmatrix} ? & ? & 0 & 0 \\ ? & ? & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{84}$$

The remaining items, those that contribute to the new x and y components are similar to the 2D case. The 3D matrix for rotation around the Z axis is presented in eq. 85).

$$\mathrm{R}_{Z,\beta} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) & 0 & 0 \\ \sin(\beta) & \cos(\beta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{85}$$

The same reasoning provides the matrices for rotations around the X and Y axis:

$$\mathrm{R}_{\mathrm{X},\beta} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) & 0 \\ 0 & \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{86}$$

$$\mathrm{R}_{Y,\beta} = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{87}$$

Note that the terms with the sin function get a sign swap in the rotation around the Y axis. This can easily be confirmed trying to rotate a few sample points.

The inverse rotation matrix around each axis can be easily computed by negating the angle. For instance, consider the rotation matrix around the Z axis with an angle α. The inverse rotation is

$$R_{z,\alpha}^{-1} = \begin{bmatrix} \cos(-\alpha) & -\sin(-\alpha) & 0 & 0 \\ \sin(-\alpha) & \cos(-\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{88}$$

To get the last matrix in equation 88) we can use the following trigonometric relations:

$$\begin{aligned} \cos(-\alpha) &= \cos(\alpha) \\ \sin(-\alpha) &= -\sin(\alpha) \end{aligned} \tag{89}$$

6.4 Rotation about an Arbitrary Axis

Consider that, instead of a main axis, we want to perform a rotation about an arbitrary direction defined by $\vec{v}$.

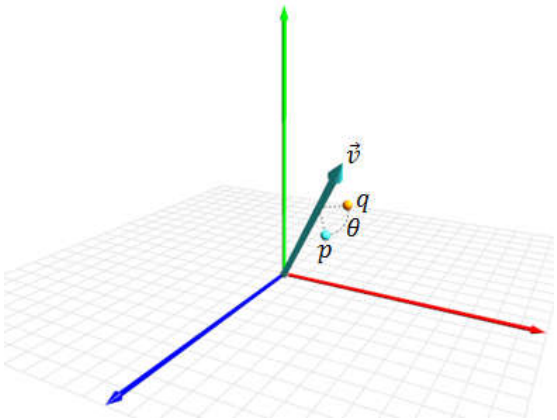

Figure 6.5 – Rotation about an arbitrary axis

Consider point p in Figure 6.5. Point q can be obtained rotating p about $\vec{v}$ by an angle θ. Therefore, we want a rotation matrix such that

$$q = R_{v,\theta} p \tag{90}$$

To obtain a matrix to perform a point 3D rotation of an angle θ about an arbitrary axis $\vec{v}$, based on what we know so far, we can use a three step process:

1. Rotate vector $\vec{v}$ so that it aligns with the Z (or X or Y) axis.
2. Rotate the point around the selected axis by θ.
3. Undo the rotations in step 1.

Unless $\vec{v}$ is already coinciding with an axis, step 1 itself requires two phases. Let's consider the general case where $\vec{v}$ does not coincide with any axis, as show in Figure 6.6.

First we perform a rotation so that $\vec{v}$ ends up in the YZ or XZ plane. Going for the first option, the YZ plane, we need to perform a rotation about the Y axis. To determine the matrix we need the cosine and sine of the angle β (see Figure 6.6).

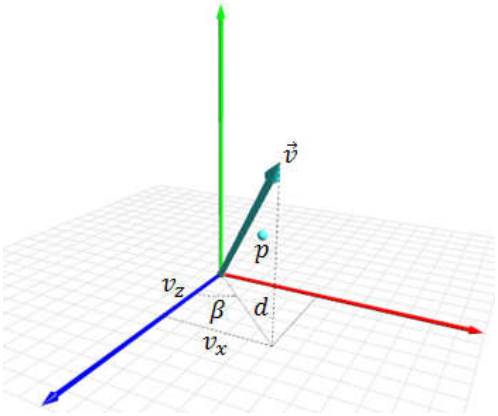

Figure 6.6 – Data required to rotate $\vec{v}$ to put it in the YZ plane

Let's consider that $\vec{v} = (v_x, v_y, v_z, 0)$. The projection of $\vec{v}$ on the XZ plane is $\overrightarrow{v_{xz}} = (v_x, 0, v_y, 0)$ and $|v_{xz}| = d = \sqrt{v_x^2 + v_z^2}$.

From Figure 6.6 we know that:

$$\begin{aligned} \cos(\beta) &= \frac{v_z}{d} \\ \sin(\beta) &= \frac{v_x}{d} \end{aligned} \tag{91}$$

However, notice that based on the right-hand rule the rotations are performed counter clockwise, and the rotation in Figure 6.6 is clockwise (the trigonometric relations in eq. 89) are going to be useful). Hence, we need to consider the rotation angle as being – β. Therefore,

$$\begin{aligned} \cos(-\beta) &= \frac{v_z}{d} \\ \sin(-\beta) &= -\frac{v_x}{d} \end{aligned} \tag{92}$$

The required matrix, R_y, and its inverse (required in step 3) can be found in eq. 93).

$$\mathrm{R}_Y = \begin{bmatrix} \frac{\mathrm{v_z}}{\mathrm{d}} & 0 & -\frac{\mathrm{v_x}}{\mathrm{d}} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{\mathrm{v_x}}{\mathrm{d}} & 0 & \frac{v_z}{\mathrm{d}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \mathrm{R}_Y^{-1} = \begin{bmatrix} \frac{\mathrm{v_z}}{\mathrm{d}} & 0 & \frac{\mathrm{v_x}}{\mathrm{d}} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{\mathrm{v_x}}{\mathrm{d}} & 0 & \frac{\mathrm{v_z}}{\mathrm{d}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{93}$$

After applying this transformation, eq. 94) , the rotated vector $\overrightarrow{v'}$ is in the YZ plane, as shown in Figure 6.7.

$$\overrightarrow{v'} = R_y \vec{v} \tag{94}$$

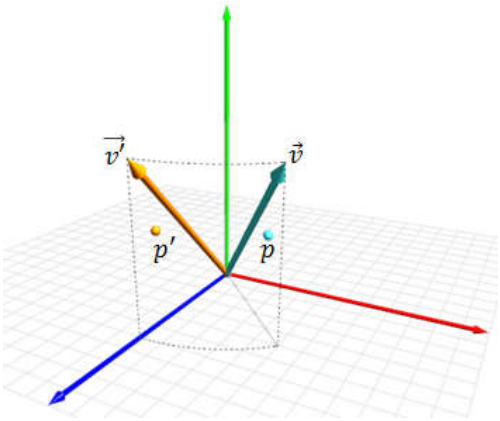

Figure 6.7 - Vector aligned with the YZ plane

Next, a rotation around the X axis is required to align the rotation axis with the Z axis. The rotation angle is α, see Figure 6.8, and again we can easily compute the cosine and sine of the required rotation. The cosine is $d/|v|$ and the sine is $v_y/|v|$.

Assuming $\vec{v}$ is normalized will save those divisions. Note that there is no loss of generalization when assuming a normalized vector, as the vector is only providing a direction and its magnitude is not relevant for the final result.

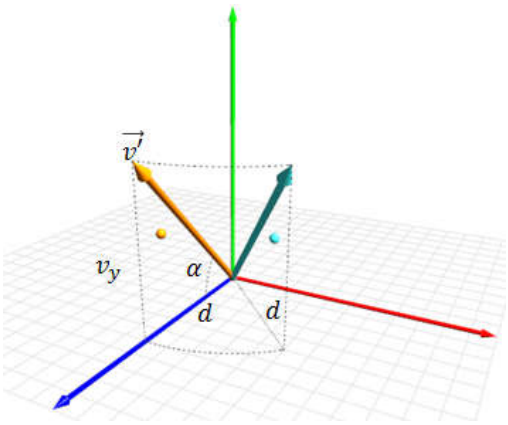

Figure 6.8 – Data required for the rotation to align $\vec{v'}$ with the Z axis

The required matrix and its inverse are:

$$R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & d & -v_y & 0 \\ 0 & v_y & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad R_X^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & d & v_y & 0 \\ 0 & -v_y & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{95}$$

After the transformations in eq. 95) we get a vector $\vec{v''}$ that is aligned with the Z axis, as required. So far we get point p'' from eq. 97). This terminates step 1 of the process.

$$\vec{v''} = R_X \vec{v'} = R_X R_Y \vec{v} \tag{96}$$

$$p'' = R_X p' = R_X R_Y p \tag{97}$$

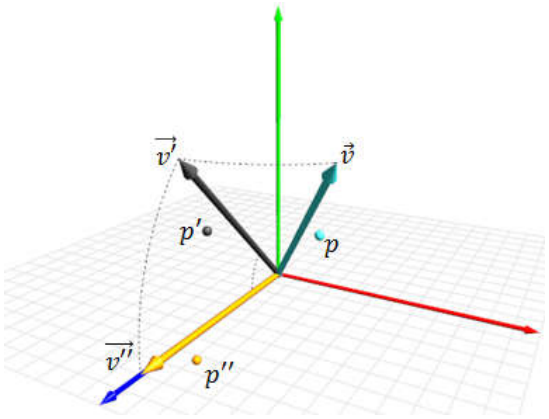

Figure 6.9 – Aligning $\vec{v}$ with the Z axis

Moving on to step 2, a rotation around the Z axis, by an angle θ is performed with the matrix from eq. 85) as shown in Figure 6.10.

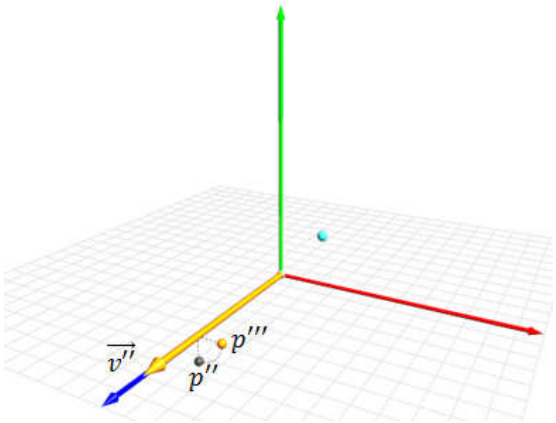

Figure 6.10 – End of step 2

The point p''' from Figure 6.10 is obtained with eq. 98).

$$p''' = R_{Z,\theta}p'' = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} p'' \tag{98}$$

The last step consists in applying, in reverse order, the inverse matrices computed in eqs. 93) and 95).

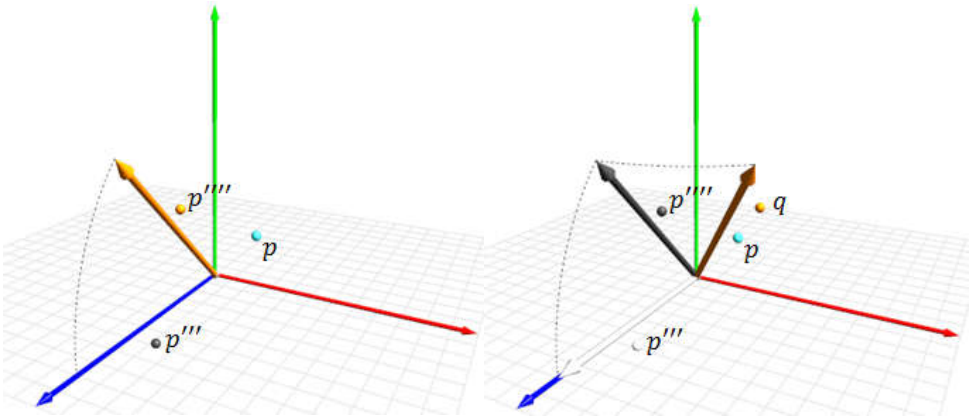

Figure 6.11 – Final rotation step. Left: undoing the rotation on the X axis; right: undoing the rotation on the Y axis

In Figure 6.11, point p'''' is obtained with eq. 99) where R_x^{-1} is as defined in eq. 95).

$$p'''' = R_X^{-1} p''' \tag{99}$$

And point q, the end result, is obtained as shown in eq. 100).

$$q = R_Y^{-1} p'''' \tag{100}$$

Putting all the matrices together, the final rotation matrix for the rotation of angle θ about a vector $\vec{v}$ can be written as follows:

$$R_{v,\theta} = R_Y^{-1} \times R_X^{-1} \times R_{Z,\theta} \times R_X \times R_{yY} \tag{101}$$

Replacing the computed matrices we get:

$$R_{v,\theta} = \begin{bmatrix} \frac{v_z}{d} & 0 & \frac{v_x}{d} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{v_x}{d} & 0 & \frac{v_z}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & d & v_y & 0 \\ 0 & -v_y & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & d & -v_y & 0 \\ 0 & v_y & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{v_z}{d} & 0 & -\frac{v_x}{d} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{v_x}{d} & 0 & \frac{v_z}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{102}$$

Where

- $\vec{v} = (v_x, v_y, v_z)$
- $d = \sqrt{v_x^2 + v_z^2}$
- $c = \cos(\theta)$
- $s = \sin(\theta)$

These matrices, (eq. 102), can be multiplied together resulting in a single matrix to perform a 3D rotation about an arbitrary axis $\vec{v}$ with an angle θ. The resulting matrix $R_{v,\theta}$ is presented in eq. 103), replacing v_x, v_y, v_z with x, y, z, respectively, to provide a less cluttered version.

$$\begin{bmatrix} x^2 + (1 - x^2)\cos(\theta) & xy(1 - \cos(\theta)) - z \times sin(\theta) & xz(1 - \cos(\theta)) + y \times \sin(\theta) & 0 \\ xy(1 - \cos(\theta)) + z \times sin(\theta) & y^2 + (1 - y^2)\cos(\theta) & yz(1 - \cos(\theta)) - x \times \sin(\theta) & 0 \\ xz(1 - \cos(\theta)) - y \times \sin(\theta) & yz(1 - \cos(\theta)) + x \times \sin(\theta) & z^2 + (1 - z^2)\cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{103}$$

To compute point q as the rotation of point p about axis $\vec{v}$ by an angle θ we write:

$$q = R_{v,\theta} \times p \tag{104}$$

6.5 Summary

The basic geometric transformations, translation, scale, and rotation, have all been defined using matrices. This provides a unified view that will prove useful when considering transformation composition. With this approach we can transform vectors and points using the same matrices. Furthermore, all matrices have a common structure. A translation matrix has the following format:

$$M = \left[\begin{array}{ccc|c} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

Rotation and scale matrices have a similar format:

$$M = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

These matrices have clearly identified blocks as follows:

$$M = \left[\begin{array}{ccc|c} & \text{RS} & & \text{T} \\ & \text{(3x3)} & & \text{(3x1)} \\ \hline & 0\ \text{(1x3)} & & 1 \end{array}\right]$$

We shall take advantage of the block format notation latter.

7 Matrices and Coordinate Systems

So far we've explored the basic geometric transformations and seen how these can be expressed with matrices. In this section we're going to see how the matrices are directly related to coordinate system specification. Furthermore, we'll see two different ways to interpret a geometric transformation.

To define a point $p(p_x, p_y, p_z)$ in a coordinate system we started from the origin and added $p_x\vec{\imath}$, $p_y\vec{\jmath}$ and $p_z\vec{k}$, where $(\vec{\imath}, \vec{\jmath}, \vec{k})$ are the versors of the coordinate system. This is depicted in Figure 7.1 (note that in the figure we added the z component before the y component, which is equivalent since vector addition is commutative).

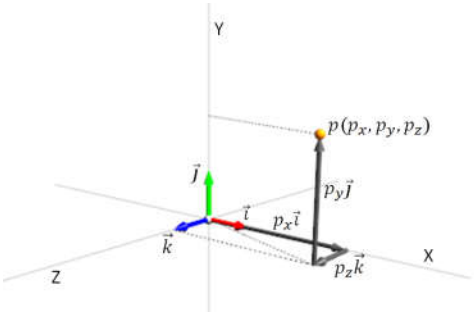

Figure 7.1- Point in coordinate system

Mathematically, we can write:

$$p = x\vec{\imath} + y\vec{\jmath} + z\vec{k} + o = \begin{bmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} & o \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{105}$$

Looking at eq. 105) we can see that the first three columns of the matrix are vectors and represent the versors of the coordinate system, and the last column is a point representing the origin.

The matrix above is the identity matrix because we haven't performed any geometric transformations yet. This is the initial system, our *global* or *world* coordinate system, upon where the geometric transformations will take place.

Let's review the translation operation. In order to translate a point p by a vector $\vec{v}$ we could built a translation matrix T and write:

$$p' = Tp \tag{106}$$

Where

$$T = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \vec{\imath'} & \vec{\jmath'} & \vec{k'} & o' \end{bmatrix} \tag{107}$$

With

$$\vec{\imath'} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \vec{\jmath'} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \vec{k'} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad o' = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} \tag{108}$$

If we look at the columns of the matrix T we can represent a new coordinate system by interpreting each of the first three columns as a versor and the last column as the origin. This new coordinate system, commonly called *local* or *model* coordinate system, can be drawn relative to the world coordinate system.

Matrix T preserves the versors, i.e. $\vec{\imath'} = \vec{\imath}$, $\vec{\jmath'} = \vec{\jmath}$, and $\vec{k'} = \vec{k}$. The origin o' gets a new location:

$$o' = o + \vec{v} \tag{109}$$

For instance, consider point p and a translation defined by a vector $\vec{v}$ resulting in point p' as represented in Figure 7.2.

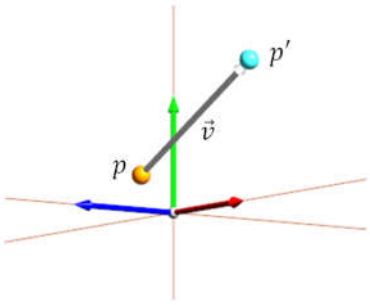

Figure 7.2 – Point translation

Another way to picture this geometric transformation is as a translation of the coordinate system itself as shown in Figure 7.3.

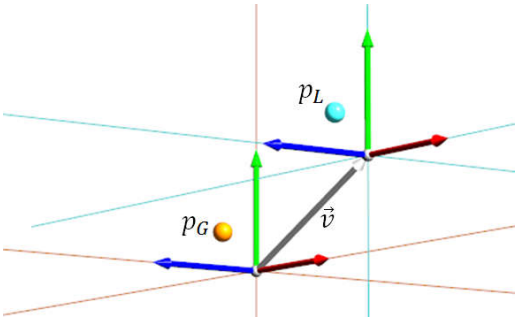

Figure 7.3 – Coordinate system translation

In Figure 7.3 we can see the world coordinate system (bottom left) and the resulting coordinate system (top right) after being translated by vector $\vec{v}$. If we consider a point (p_G) in the global coordinate system and translated it with vector $\vec{v}$ we obtain the same location as when we draw the point (P_L), with its original coordinates, relative to the translated, or local, coordinate system.

This provides us with two equivalent ways of interpreting a geometric transformation. We can interpret a translation as moving a point within a coordinate system (as in Figure 7.2), or we can interpret it as being a coordinate system change (as in Figure 7.3). This duality will prove to be a useful tool when we examine geometric transformation composition.

This duality also allows us to deduce the rotation matrix about a main axis with a different, perhaps more intuitive, approach. Consider for instance the rotation about the Z axis by an angle α.

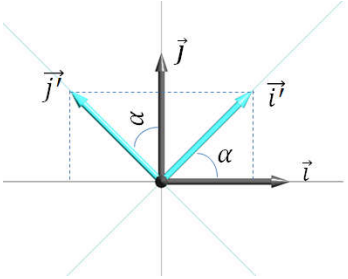

Figure 7.4 – Rotation around the Z axis

In Figure 7.4 we can see both the global (grey) and local (cyan) coordinate systems (the Z axis is perpendicular to the paper). We can write the versors of the local system $(\vec{\imath'},\ \vec{\jmath'}, \vec{k'})$ as a function of the versors of the global coordinate system $(\vec{\imath}, \vec{\jmath}, \vec{k})$. The versor associated with the Z axis remains unchanged, hence,

$$\vec{k'} = 0\vec{\imath} + 0\vec{\jmath} + \vec{k} \tag{110}$$

Both $\vec{\imath'}$ and $\vec{\jmath'}$ can be computed using some trigonometry:

$$\begin{aligned} \vec{\imath'} &= \cos(\alpha)\,\vec{\imath} + \sin(\alpha)\,\vec{\jmath} + 0\vec{k} \\ \vec{\jmath'} &= -\sin(\alpha)\,\vec{\imath} + \cos(\alpha)\,\vec{\jmath} + 0\vec{k} \end{aligned} \tag{111}$$

Considering that the matrix for this transformation will have in its columns the new versors and the origin (which remains unaltered by the rotation) we can define the matrix as being

$$R_{z,\alpha} = \begin{bmatrix} \vec{\imath'} & \vec{\jmath'} & \vec{k'} & o \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{112}$$

The result in eq. 112) matches exactly the matrix that we got in eq. 85).

We can also work the other way around: given a matrix we can draw the coordinate system transformation it represents. For instance let us consider the following matrix M representing a composite geometric transformation:

$$M = \begin{bmatrix} 0.707 & -0.707 & 0 & 2 \\ 0.707 & 0.707 & 0 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{113}$$

The matrix represents a transformed local coordinate system, where the new origin is at $(2,1,-1)$, as defined by the fourth column. The new Z axis remains with the same direction. The first and second columns, representing the X and Y axes, provide the values that define the new versors, relative to original global coordinate system. Figure 7.5 show the original global coordinate system (dark grey), and the local coordinate system (cyan).

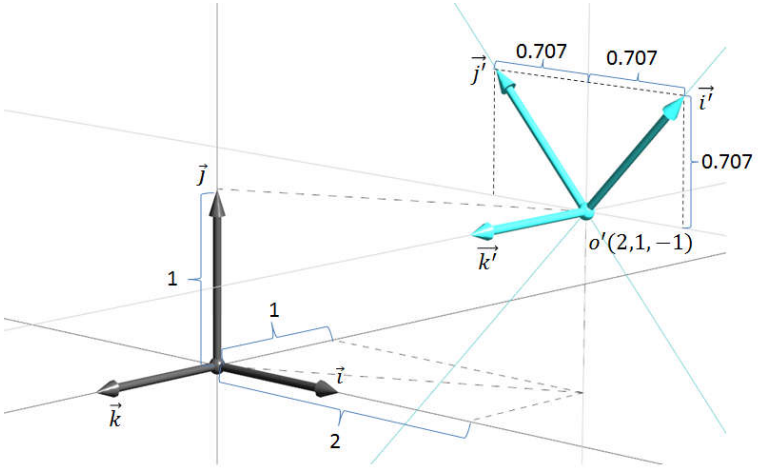

Figure 7.5 – Coordinate system after a geometric transformation

So, in essence, from a matrix we can draw the transformed coordinate system.

7.1 Composition of Geometric Transformations

Geometric transformations can be composed and the resulting transformation is also a 4x4 matrix with the same block format as the individual matrices.

Consider vector $\vec{v} = \vec{a} + \vec{b}$. We can describe a translation by vector $\vec{a}$ with a matrix T_a and a translation by vector $\vec{b}$ with a matrix T_b as defined in eq. 114).

$$T_a = \begin{bmatrix} 1 & 0 & 0 & a_x \\ 0 & 1 & 0 & a_y \\ 0 & 0 & 1 & a_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_b = \begin{bmatrix} 1 & 0 & 0 & b_x \\ 0 & 1 & 0 & b_y \\ 0 & 0 & 1 & b_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{114}$$

Suppose we first apply the translation defined by $\vec{a}$. Then, point p will be transformed as:

$$p' = T_a p \tag{115}$$

Next we apply the transformation defined by $\vec{b}$ to point p'.

$$p'' = T_b p' = T_b T_a p \tag{116}$$

This is equivalent to transforming p by $\vec{v}$, as shown in Figure 7.6.

$$p'' = T_v p \tag{117}$$

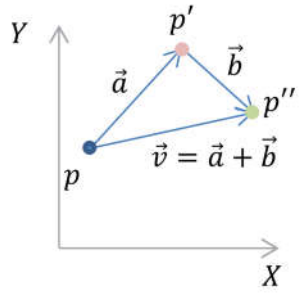

Figure 7.6 – Composite translation

The composition of the two translations can be read in two different, but equivalent, ways. Reading the right hand side of eq. 116) from right to left we have point p, then a translation by $\vec{a}$, and finally a translation by $\vec{b}$. All these translations occur in the same coordinate system, see Figure 7.6.

The expression can also be read from left to right, in which case we focus on the coordinate systems, and represent the point with its original coordinates in the new coordinate system, as seen in Figure 7.7.

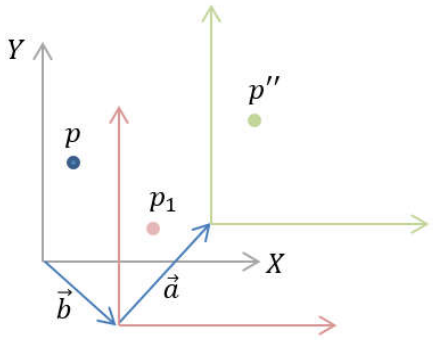

Figure 7.7 – Translation of coordinate systems

Since vector addition is commutative, the order of the operations is not relevant when considering only translations.

$$p'' = T_a T_b p = T_b T_a p \tag{118}$$

However, this is not the general case. Consider for instance that $p'' = T_v R_{45,Z} p$. First, we're going to read from left to right, i.e., we're going to focus on the coordinate system transformations, see Figure 7.8. The first operation is a translation defined by a vector $\vec{v}$, the second operation is a rotation of 45 degrees about the Z axis. The original coordinate system is represented in grey in Figure 7.8. After the translation we get the red coordinate system. The final system, after the rotation is represented in green.

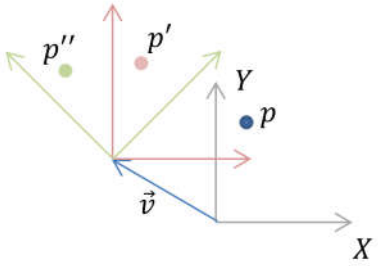

Figure 7.8 – Translation followed by a rotation of the coordinate system

In Figure 7.9, the point is being transformed in a fixed coordinate system. So $p' = R_{45,Z} p$, and $p'' = T_v p'$. As expected, if both figures, Figure 7.8 and Figure 7.9, where superimposed, point p'' would be in the same plane.

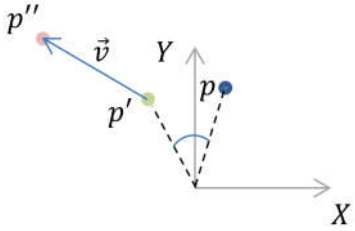

Figure 7.9 – Rotation followed by a translation of a point in a fixed coordinate system

Now consider that we switched the order of operations, so $q'' = R_{45,Z}T_v p$.

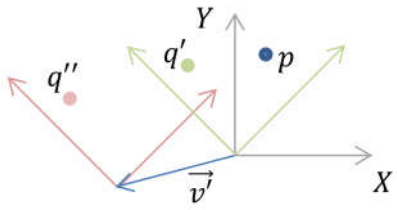

Figure 7.10 – Switching the order of operations

Comparing Figure 7.10 with Figure 7.9, it is clear that $p'' \neq q''$. The above example shows that, in the general case, the composition of geometric transformations is not commutative.

Compositions containing a non-uniform scale require some care when combined with rotations. The rotation matrix presented in sections 6.3 and 6.4 implicitly assumes that the versors are all the same length, something which is not true when considering a previous non-uniform scale. So performing a rotation in a coordinate system that has been previously non-uniformly scaled does not provide the results we would expect from a rotation of the coordinate system. Not only may we be surprised by the shape of the coordinate system, but the magnitude of the rotated versors is not the magnitude of the original versors.

Consider for instance a non-uniform scale provided by matrix S and a 45 degrees rotation about the Z axis provided by matrix R.

$$S = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R = \begin{bmatrix} \cos(45) & -\sin(45) & 0 & 0 \\ \sin(45) & \cos(45) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{119}$$

If we compute $M = SR$ we get:

$$M = \begin{bmatrix} 3 \times \cos(45) & -3 \times \sin(45) & 0 & 0 \\ \sin(45) & \cos(45) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{120}$$

Drawing the coordinate system defined by M we get the result presented in Figure 7.11, where the original versors are depicted in blue, and the transformed versors (from the columns of matrix M) are colored in red. As the figure shows, the new X and Y axis are no longer perpendicular.

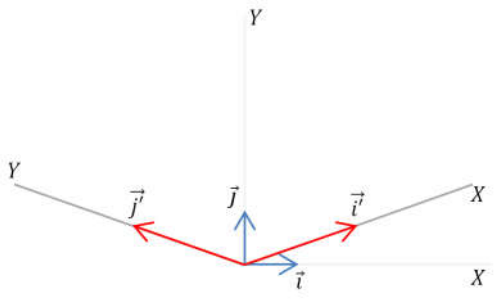

Figure 7.11 – Rotation after non-uniform scale

Combining scales with translations does not raise the same issue as translations do not alter the versors of the coordinate system.

A particularly useful case is placing an object in a chosen position, with a particular orientation. Suppose a model of a snowman has been defined such that without transformations it looks as in Figure 7.12. Now consider that we want to place the snowman as in Figure 7.13. In this particular case T is a translation matrix that translates the snowmen 2 units to the right and 2 unit along the negative Z axis. The rotation is performed around the Y axis, with an angle of -30 degrees.

To achieve this we can apply two basic geometric transformations: a translation and a rotation.

In this case a points for the new snowman, p', could be computed as the result of applying the transformations to a point of the untransformed snowman, p:

$$p' = Mp = TRp \tag{121}$$

Considering matrix block notation we get

$$M = TR = \begin{bmatrix} I & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r & t \\ 0 & 1 \end{bmatrix} \tag{122}$$

In general, when we want to displace an object by a vector and then rotate it, the resulting matrix composition will have the above format.

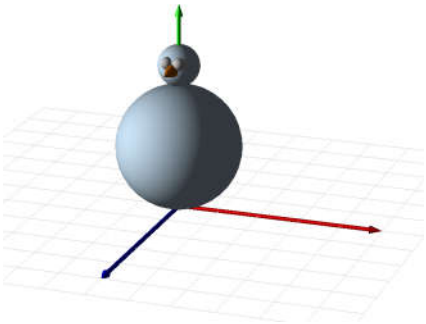

Figure 7.12 – A snowman without any geometric transformations

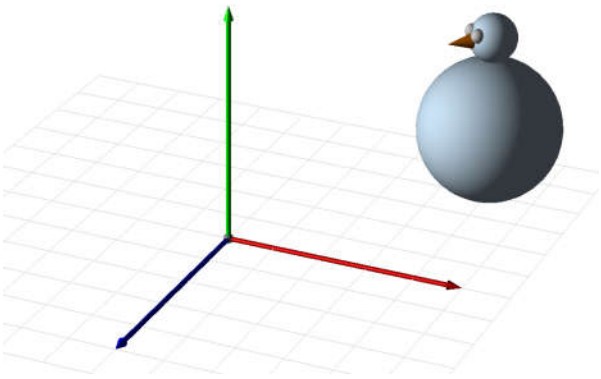

Figure 7.13 – The snowman with geometric transformations applied

7.2 Inverse of an Arbitrary Geometric Transformation

Consider an arbitrary affine geometric composition such as presented in eq.123).

$$M = M_1 \times M_2 \times \ldots \times M_n \tag{123}$$

The inverse of such a transformation can be computed as the multiplication of the inverses of the matrices on the right side of the equation, in reverse order, i.e.,

$$M^{-1} = (M_1 \times M_2 \times \ldots \times M_n)^{-1} = M_n^{-1} \times \ldots \times M_2^{-1} \times M_1^{-1} \tag{124}$$

Another approach is to find the inverse based on the blocks that make up a matrix representing a geometric transformation. We know that

$$M = \begin{bmatrix} RS & T \\ 0 & 1 \end{bmatrix} \tag{125}$$

Multiplying two of these matrices provides a result as follows.

$$M_1 \times M_2 = \begin{bmatrix} RS_1 & T_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} RS_2 & T_2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} RS_1 \times RS_2 & RS_1 \times T_2 + T_1 \\ 0 & 1 \end{bmatrix} \tag{126}$$

If $M_1 \times M_2 = I$ then

$$\begin{cases} RS_1 \times RS_2 = I_3 \\ RS_1 \times T_2 + T_1 = 0 \end{cases} \Leftrightarrow \begin{matrix} RS_2 = RS_1^{-1} \\ T_2 = -RS_1^{-1} \times T_1 \end{matrix} \tag{127}$$

Hence, the inverse of matrix M (eq. 125) can be written as:

$$M^{-1} = \begin{bmatrix} RS^{-1} & -RS^{-1} \times T \\ 0 & 1 \end{bmatrix} \tag{128}$$

The inverse of the 3x3 submatrix RS can be computed by any of the traditional algebraic matrix inversion methods. However, when no scales are applied there is a great shortcut. As mentioned before each column of submatrix RS represents a versor of the coordinate system. Having this in mind, consider what happens when we multiply submatrix RS and its transpose RS^T.

$$RS^T \times RS = \begin{bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{bmatrix} \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} = \begin{bmatrix} \vec{i} \cdot \vec{i} & \vec{i} \cdot \vec{j} & \vec{i} \cdot \vec{k} \\ \vec{j} \cdot \vec{i} & \vec{j} \cdot \vec{j} & \vec{j} \cdot \vec{k} \\ \vec{k} \cdot \vec{i} & \vec{k} \cdot \vec{j} & \vec{k} \cdot \vec{k} \end{bmatrix} \tag{129}$$

From the dot product definition (section 3.3) we know that

$$\vec{a} \cdot \vec{b} = \cos(\alpha) |\vec{a}| |\vec{b}| \tag{130}$$

Since we assumed no scales are present, all the lengths will be one and the dot product formula simplifies to the $\cos(\alpha)$. Hence, as the axes are perpendicular to each other, all elements apart from those in the diagonal will be zero. The diagonal will be filled with 1's.

$$\vec{a} \cdot \vec{b} = 0 \; if \; \vec{a} \neq \vec{b}$$
$$\vec{a} \cdot \vec{a} = 1 \tag{131}$$

Therefore, when only rotations are used the transpose is the inverse.

$$RS^T = RS^{-1} \tag{132}$$

In this particular case the inverse of the 4x4 matrix M is

$$M^{-1} = \begin{bmatrix} RS^T & -RS^T \times T \\ 0 & 1 \end{bmatrix} \tag{133}$$

8 Camera

Matrices let us specify coordinate system changes that allow us to move, scale and orient our 3D models to compose a scene in global space. The camera is our view to this assembled world. Through the camera we are able to see the 3D world we build with geometric primitives, such as triangles, and geometric transformations.

The most simple camera model is the pinhole camera, where the camera can be though as a closed light proof box, with a tiny hole (the pinhole) in the center of the front side. Light enters through the hole and hits the opposite side of the box where an inverted image is formed (see Figure 8.1).

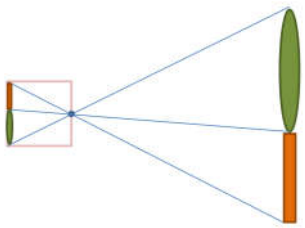

Figure 8.1 – Pinhole camera producing an inverted image

In computer graphics we render in a rectangular window, so our hole is rectangular. The viewing volume provided by this camera is limited by a pyramid.

It is common to add a couple of planes limiting the viewing volume of the camera: the near and far planes. The main goal of these planes is to have a finite volume that can easily be manipulated to allow us to project the 3D scenario in a 2D window. The viewing volume becomes a truncated pyramid, i.e. a frustum (see Figure 8.2). Only objects inside the view frustum will be visible in the final rendering.

A camera has a set of parameters that defines its behavior, namely a set of extrinsic parameters to define its physical location and orientation, and a set of parameters that define intrinsic features, such as field of view. In a pinhole camera, the field of view can be seen as the angle α. The intrinsic parameters help to define the projection from the 3D world to the 2D window.

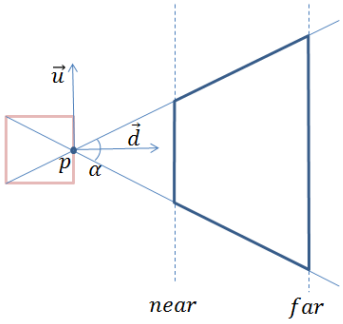

Figure 8.2 - Camera parameters

Figure 8.2 depicts a pinhole camera with its parameters. The extrinsic parameters are:

- p – camera position
- $\vec{d}$ – looking direction
- $\vec{u}$ – the up vector

The up vector determines the tilt of the camera, for instance an up vector $\vec{u} = (0,1,0)$produces the equivalent of the landscape mode and this is the default in the general case. When the up vector is rotated, the camera tilts accordingly.

The intrinsic parameters are:

- α – the vertical viewing angle, or field of view (fov)
- $near$ – the distance to the near plane
- far – the distance to the far plane

In computer graphics we do not need to consider the image plane as the back of the box. Since this is a mathematical model we can choose where we want the image plane. It is more convenient to select the image plane in front of the position of the camera, as the final image won't be inverted.

Rendering is performed in a 2D rectangular portion of a window of arbitrary dimensions. This is called screen space. The final goal of the projection operation is to transform the points in our world according to the camera position to convert their coordinates into screen space coordinates.

The camera defines a new space called the camera space. In this space, the camera is at the origin, looking long the negative Z axis. To transform the points from the world space to camera space see section 8.1.

Section 8.2 deals with the transformation from camera space to window space. This process is commonly performed with an intermediate step. The camera coordinates are first transformed to an intermediate space, called the clip space, from which they are finally transformed into window coordinates. The reason for this intermediate space will become clear as we move on.

8.1 "Placing" the Camera

The extrinsic parameters provide us the camera position, its viewing direction and the up vector. Based on these parameters we can transform all the vertices from global space to camera space.

The first step is to compute a coordinate system for the camera based on the extrinsic parameters. The position establishes the origin of the coordinate system. The viewing direction matches the negative Z axis, and if the up vector matches the Y axis, normalizing these vectors should provide us with the $\vec{k}$ and $\vec{j}$ versors. To get the remaining versor we use the cross product operation: $\vec{\imath} = \vec{k} \times \vec{j}$.

When the up vector is only an approximation to the Y axis of the camera coordinate system (as it is often the case when we use `gluLookAt`) then we must do some extra work to find the versors.

Consider the parameters provided in the `gluLookAt` function:

- p – camera position
- l– a point that the camera is aiming
- $\overrightarrow{up}$ - an approximation to the up vector.

From p and l we can get the viewing direction as $\vec{d} = l - p$.

The viewing direction continues to match the negative Z direction, hence,

$$\vec{k} = -\vec{d}/|\vec{d}| \tag{134}$$

To compute the versor for the X axis we use both $\vec{k}$ and the approximation to the Y axis, the up vector.

$$\vec{\imath} = \overrightarrow{up} \times \vec{k} \,/|\overrightarrow{up} \times \vec{k}| \tag{135}$$

Finally, the versor for the Y axis can be computed as $\vec{j} = \vec{k} \times \vec{\imath}$. Note that, as $\vec{\imath}$ and $\vec{k}$ are normalized and orthogonal, there is no need to normalize vector $\vec{j}$ as the result of the cross product is already unit length.

Once we have the three versors we could build a matrix to "place" the camera. Matrix M, in eq. (136), would place the camera in global space, as depicted in Figure 8.3.

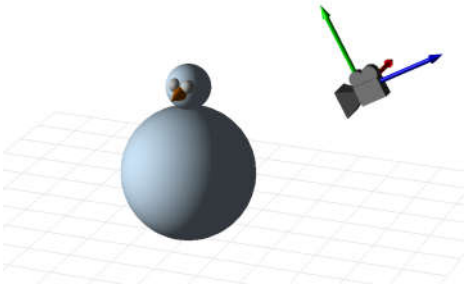

Figure 8.3 – Camera in global space.

$$M = \begin{bmatrix} \vec{\imath} & \vec{j} & \vec{k} & p \end{bmatrix} \tag{136}$$

Using block matrix notation we can write

$$M = \begin{bmatrix} R & P_3 \\ 0 & 1 \end{bmatrix} \tag{137}$$

Where

$$\begin{cases} P_3 = [p_x \quad p_y \quad p_z] \\ R = \begin{bmatrix} i_x & j_x & k_x \\ i_y & j_y & k_y \\ i_z & j_z & k_z \end{bmatrix} \end{cases} \tag{138}$$

In fact what we really want is to move all objects to camera space, as this is the space where projection (see section 8.2) will occur. So, instead of placing the camera in global space, we want to do exactly the opposite, we want to place all objects (which are in global space) in camera space. To better understand this concept consider the following example: A camera is at a distance *k* from an object (Figure 8.4 a).

It is equivalent, in terms of the final rendered picture, to either move the camera closer to the object (Figure 8.4 c), or the object closer to the image (Figure 8.4 b). The transformations to achieve Figure 8.4 b (move the ball) is the inverse of the transformation to achieve Figure 8.4 c (move the camera).

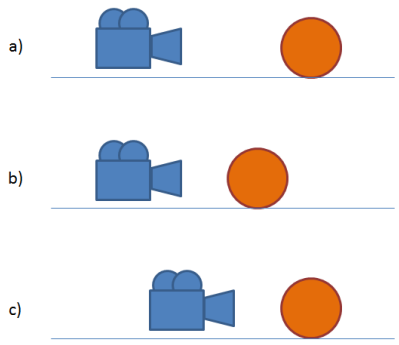

Figure 8.4 – Relative position between camera and an object.

Consider the coordinate system centered in the camera shown in Figure 8.3. This coordinate system defines a space that will be referred as *camera space* from now on. The camera is at the origin of this space, and as mentioned before the looking direction is along the negative Z axis. Figure 8.5 shows the same snowman as in Figure 8.3, but this time in camera space.

Figure 8.5 – Snowman in camera space.

Since we want to perform the inverse operation of placing the camera, the matrix we want is actually M^{-1}.

$$M^{-1} = \begin{bmatrix} R & P_3 \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} R^T & -R^T \times P_3 \\ 0 & 1 \end{bmatrix} \tag{139}$$

8.2 Projecting the 3D World to a 2D window

This section deals with the transformation from camera space to clip space (section 8.2.1), and the conversion from clip space to screen coordinates (section 8.2.2). All the computations and results are obtained using only the intrinsic camera parameters, namely the field of view, and the near and far distances.

8.2.1 From Camera Space to Clip Space

In camera space the origin is the position of the camera. The camera is "looking" into the negative Z axis. The goal of the conversion from camera space to clip space is to transform the viewing volume in camera space to a cube in clip space where all coordinates vary between -1 and 1. We shall assume a symmetric frustum, i.e., points with $z = 0$ will appear in the middle of the frustum.

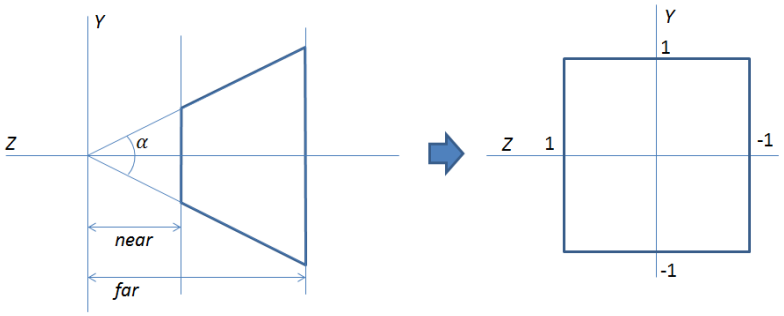

Figure 8.6 - From Camera Space to Clip Space

The conversion between camera space and clip space will be performed with a 4x4 matrix, thus keeping the geometric transformation pipeline uniform, i.e., all operations are performed with 4x4 matrices.

The approach taken in here is to fill up the values in the matrix step by step.

First, consider a point $p(p_x, p_y, p_z, 1)$ in the view frustum. To obtain X and Y coordinates we need to project p to the image plane. For now, let's assume that the image plane is at a distance d from the camera. The projected point in Figure 8.7 is p'.

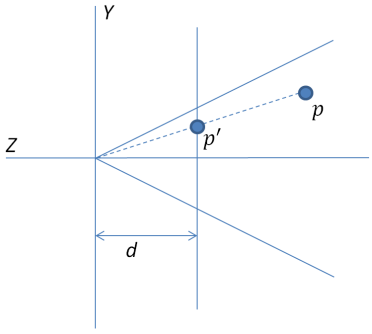

Figure 8.7 – Projecting p on the image plane

The relation between the y coordinate of p and p' can be derived based on the ratio of similar triangles (eq.140).

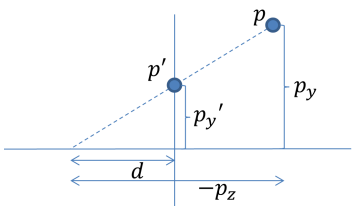

Figure 8.8- Similar triangles

$$\frac{p_y'}{d} = \frac{p_y}{-p_z} \tag{140}$$

Therefore,

$$p_y' = p_y \frac{d}{-p_z} \tag{141}$$

A similar ratio is obtained for the x component. Hence, the coordinates of p' are $\left(p_x \frac{d}{-p_z}, p_y \frac{d}{-p_z}, -d, 1\right)$ or, multiplying all coordinates by $-p_z$, $(p_x d,\ p_y d,\ p_z d,\ -p_z)$.

We need a matrix for this transformation, i.e. $p' = Mp$. The following matrix will do the trick:

$$M = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{142}$$

This provides us with the first values of our matrix. Now let us try to convert some points in camera space to clip space to find the remaining values of matrix M.

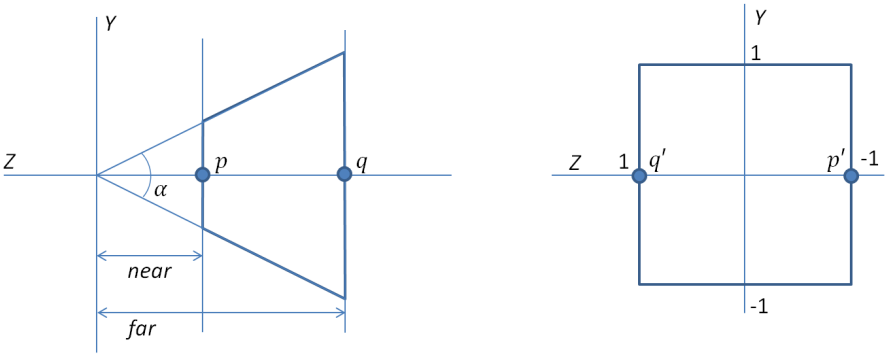

Figure 8.9 - From camera space to clip space (step 1)

From

Figure 8.9 we know that

$$\begin{aligned} p &= (0, 0, -near, 1) \quad & p' &= (0, 0, -1, 1) \\ q &= (0, 0, -far, 1) \quad & q' &= (0, 0, 1, 1) \end{aligned} \tag{143}$$

Using the matrix in eq. 142), only the third row is relevant, and within the row only the last two values are used. So we could try a matrix such as the one in eq. 144) and then try to find the value of a and b:

$$M = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{144}$$

We know that

$$\begin{aligned} p' &= Mp \\ q' &= Mq \end{aligned} \tag{145}$$

Using M from eq. (144), we get

$$p' = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -near \\ 1 \end{bmatrix} \quad q' = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -far \\ 1 \end{bmatrix} \tag{146}$$

Therefore, from eq. 146) we know that

$$\begin{aligned} p' &= (0, 0, -a \times near + b,\ near) \\ q' &= (0, 0, -a \times far + b,\ far) \end{aligned} \tag{147}$$

Or

$$\begin{aligned} p' &= (0, 0, \tfrac{-a \times near + b}{near},\ 1) \\ q' &= (0, 0, \tfrac{-a \times far + b}{far},\ 1) \end{aligned} \tag{148}$$

Combining eq. 143) and eq. 148) we get the following system of equations

$$\begin{cases} \frac{-a \times near + b}{near} = -1 \\ \frac{-a \times far + b}{far} = 1 \end{cases} \tag{149}$$

Solving the system we get

$$\begin{cases} \frac{-a \times near + b}{near} = -1 \\ \frac{-a \times far + b}{far} = 1 \end{cases} \Leftrightarrow \begin{cases} -a \times near + b = -near \\ -a \times far + b = far \end{cases} \Leftrightarrow \begin{cases} b = near(a-1) \\ -a \times far + near(a-1) = far \end{cases}$$

$$\Leftrightarrow \begin{cases} b = near(a-1) \\ -a \times far + near \times a = far + near \end{cases} \Leftrightarrow \begin{cases} b = near(a-1) \\ a(near - far) = far + near \end{cases}$$

$$\Leftrightarrow \begin{cases} b = near(a-1) \\ a = \frac{far + near}{near - far} \end{cases} \Leftrightarrow \begin{cases} b = near\left(\frac{far + near}{near - far} - 1\right) \\ a = -\frac{far + near}{far - near} \end{cases}$$

$$\Leftrightarrow \begin{cases} b = near\left(\frac{far + near}{near - far} - \frac{near - far}{near - far}\right) \\ a = -\frac{far + near}{far - near} \end{cases} \Leftrightarrow \begin{cases} b = near\left(\frac{2far}{near - far}\right) \\ a = -\frac{far + near}{far - near} \end{cases}$$

$$\Leftrightarrow \begin{cases} b = -\frac{2 \times near \times far}{far - near} \\ a = -\frac{far + near}{far - near} \end{cases}$$

So our matrix looks as follows:

$$M = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \times near \times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (150)$$

All that's left is to find a value for d. In order to achieve this, consider point p in camera space and point p' in clip space in Figure 8.10.

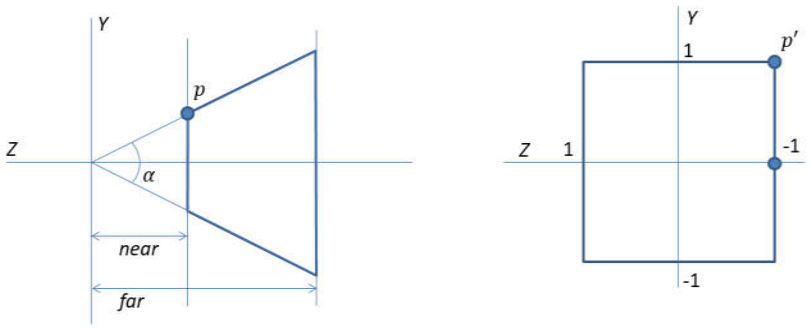

Figure 8.10 – From camera space to clip space (step 2)

The coordinates of the points are:

$$p = (0,\ tg\left(\frac{\alpha}{2}\right) near, -near,\ 1)$$
$$p' = (0, 1, -1, 1) \quad (151)$$

Using matrix M we get

$$p' = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ tg\left(\frac{\alpha}{2}\right) near \\ -near \\ 1 \end{bmatrix} \tag{152}$$

Hence,

$$p' = \begin{bmatrix} 0 \\ d \times tg\left(\frac{\alpha}{2}\right) near \\ \dots \\ near \end{bmatrix} \tag{153}$$

Which is equivalent to

$$p' = \begin{bmatrix} 0 \\ d \times tg\left(\frac{\alpha}{2}\right) \\ \dots \\ 1 \end{bmatrix} \tag{154}$$

Putting together 151) and 154) we get:

$$1 = tg\left(\frac{\alpha}{2}\right) \times d \tag{155}$$

$$d = cotg\left(\frac{\alpha}{2}\right) \tag{156}$$

The same reasoning can be applied to a point $q = \left(tg\left(\frac{\alpha}{2}\right) near, 0, -near, 1\right)$ which should be transformed to $q' = (1, 0, -1, 1)$. Therefore our matrix M looks like:

$$M = \begin{bmatrix} cotg\left(\frac{\alpha}{2}\right) & 0 & 0 & 0 \\ 0 & cotg\left(\frac{\alpha}{2}\right) & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{157}$$

If the frustum base is not square then we need to multiply the x component by a ratio r as defined in eq.158)

$$r = \frac{width}{height} \tag{158}$$

$$M = \begin{bmatrix} cotg\left(\frac{\alpha}{2}\right) \times ratio & 0 & 0 & 0 \\ 0 & cotg\left(\frac{\alpha}{2}\right) & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{159}$$

Hence, in order to transform a point p from camera space to a point p'in clip space we apply matrix M as:

$$p' = Mp \tag{160}$$

After this transformation, the W component is typically different from one. Hence, the coordinates are then divided by the W component providing Normalized Device Coordinates (NDC). In NDC all coordinates for points that in camera space are inside the view frustum have values between -1 and 1. The final 3D coordinates in clip space for point p_c, are:

$$p_c = (\frac{p'_x}{p'w}, \frac{p'_y}{p'_w}, \frac{p'_z}{p'_w}) \tag{161}$$

8.2.2 From Clip Space to Screen Coordinates

The rendering goes to a rectangular region of the window created by an OpenGL application. Although this area is commonly set as the whole window, this is a particular case. The rectangular area is called the viewport. See Figure 8.11 for a generic viewport. The blue line is presented only to show the boundaries of the viewport.

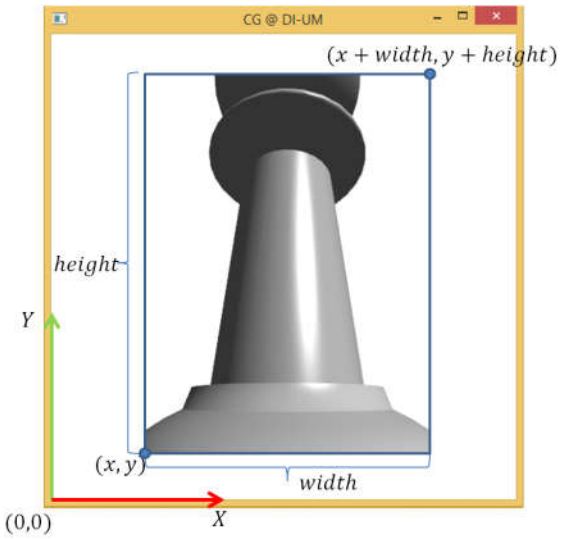

Figure 8.11 – Window coordinates

The viewport is defined by its initial (x, y) coordinates, plus $width$ and $height$.

The transformation from NDC to window coordinates is a simple mapping such that the $(p_{c.x}, p_{c.y})$ values in NDC are translated and scaled according to the viewport definition:

$$\begin{aligned}[-1,1] \rightarrow [x, x + width] \\ [-1,1] \rightarrow [y, y + height]\end{aligned} \tag{162}$$

This mapping can be achieved as follows:

$$\begin{aligned}x_{sc} = \frac{width}{2} \times p_{c.x} + x + \frac{width}{2} \\ y_{sc} = \frac{height}{2} \times p_{c.y} + y + \frac{height}{2}\end{aligned} \tag{163}$$

9 The Geometric Pipeline

We've explored a bunch of operations so far. This section aims at linking them together in a pipeline where the inputs are the vertices of the triangles as they are created analytically or loaded from a file containing a model, and the outputs are the screen coordinates of the point.

The first stage is to assemble our scene. 3D models, defined in local space, are composed in a scene through geometric transformations. For this purpose the most popular transformations are translations, rotations, and scales. Eq. 164) translates this idea recurring to regular expressions, where a point in local space p_L is transformed with zero or more transformations to get to global space, p_G.

$$p_G = M\, p_L, \quad M = [TRS]* \tag{164}$$

The camera defines a new space, where the camera is located at the origin looking in the negative Z direction. All objects must therefore be transformed into this space which is defined by the camera's extrinsic parameters: position, view direction, and up vector. This is also achieved with geometric transformations, namely a rotation (defined based on the view direction and up vector) and a translation (defined by the camera's position).

$$p_C = R_C\, T_C\, p_G \tag{165}$$

Figure 9.1 illustrates this section of the geometric pipeline.

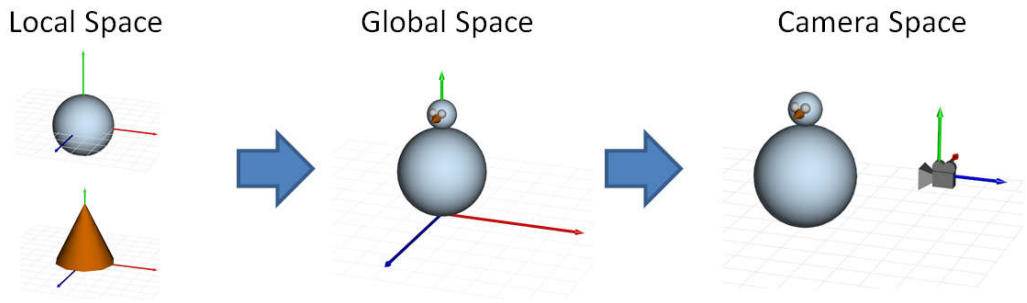

Figure 9.1 – From local space to camera space

Once in camera space, all potentially visible objects are inside a frustum defined by the camera intrinsic parameters, namely the field of view, and the distances to the near and far planes. This frustum is transformed into a cube with dimension 2, moving us from camera space to clip space (see Figure 9.2).

For perspective projections with symmetric frusta, this is achieved with the matrix defined in eq.159). In here we shall call that matrix P, for projection.

$$p_{Clip} = P\, p_C \tag{166}$$

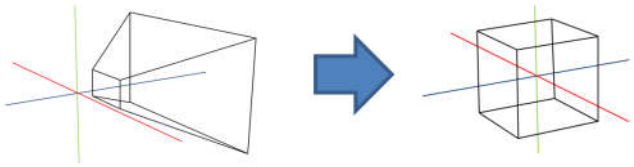

Figure 9.2 – from camera space to clip space

In general, points in clip space (p_x, p_y, p_z, p_w) have the W coordinate different from one, so homogeneous divide is required, providing us with points in Normalized Device Coordinates:

$$p_{NDC} = (\frac{p_x}{p_w}, \frac{p_y}{p_w}, \frac{p_z}{p_w}) \tag{167}$$

Finally, using eq.163), we get the screen coordinates of our points as output of the geometric pipeline.

10 Depth Buffer

Consider that we draw an object 5 units away from the camera and then ask to draw another object at 10 units from the camera and partially occluded by the first object.

To properly solve this situation we must resolve, pixel by pixel, which pixels from the second object are visible from the camera. The depth buffer, or Z buffer, is an auxiliary buffer that allows OpenGL to deal with these occlusions. It is a buffer with the same width and height as the framebuffer, and stores the depth of each pixel as seen from the camera.

Initially the depth buffer is filled with the largest possible depth. When a pixel is to be drawn at location (x, y) in the viewport, its depth is compared with the depth stored in the Z buffer at the same (x, y) location. If the pixel depth is shorter than the stored depth then the pixel may be safely drawn and its depth is stored in the Z buffer, otherwise the pixel is discarded and the Z Buffer is not altered.

Therefore, it is important that the Z buffer has enough precision to distinguish between nearby depths to ensure correct drawing ordering.

Let's see what happens to the Z component of a point in camera coordinates when it is transformed to clip space assuming a symmetric frustum for a perspective projection.

Consider a point $p = (0,0,z_e,1)$ in camera coordinates. The point p' in clip space can be computed as:

$$p' = ProjMatrix \times p = \begin{bmatrix} cotg\left(\frac{\alpha}{2}\right) \times ratio & 0 & 0 & 0 \\ 0 & cotg\left(\frac{\alpha}{2}\right) & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2\times near\times far}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ z_e \\ 1 \end{bmatrix} \quad (168)$$

$$= \begin{bmatrix} 0 \\ 0 \\ -\frac{far+near}{far-near} \times z_e - \frac{2\times near\times far}{far-near} \\ -z_e \end{bmatrix} \quad (169)$$

We are only interested in the Z component of p', after perspective division:

$$z_w = \frac{-\frac{far+near}{far-near}\times z_e - \frac{2\times near\times far}{far-near}}{-z_e} \quad (170)$$

$$z_w = \frac{far+near}{far-near} + \frac{2\times near\times far}{far-near} \times \frac{1}{z_e} \quad (171)$$

The depth values are stored as integers with b bits. Considering $s = 2^b - 1$, the stored value is computed as:

$$z'_w = s \times (z_w \times 0.5 + 0.5) \quad (172)$$

Note that z_w is a value between -1 and 1, so we had to change the range to $[0,1]$. The variable z'_w is stored as an integer, and since the right side of equation 172) is a real number, it makes sense

to consider rounding to the nearest integer. What this implies is that the stored depth represents a depth interval. For instance when $z'_w = 0$ we mean the interval $[0,0.5[$ and when when $z'_w = 1$ we mean the interval $[0{,}5,1.5[$, and so on. The last interval is $[s - 0.5, s]$.

So far we've covered the steps from getting a depth value in camera coordinates to a depth value in a range between 0 and 1. The first thing that is clear is that this relation is not linear. Plotting it we get something like:

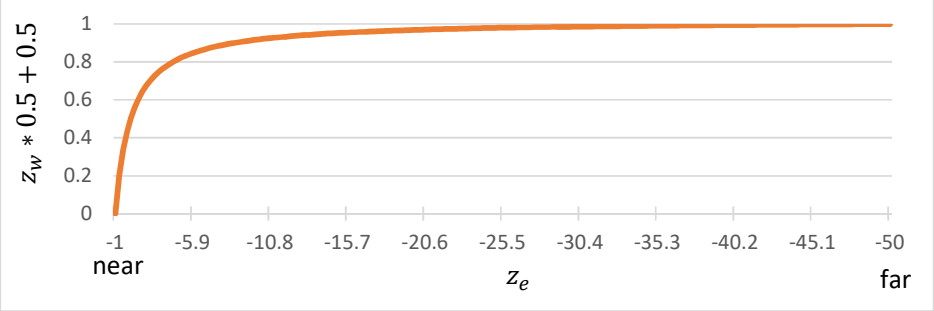

Figure 3 – Non-linear relation between Z-buffer stored value camera space z value

From the graph we can see that when z_e is larger we might get into depth precision problems. The Z buffer provides far more precision for objects near the camera than for objects far away. Note that this is not on purpose, rather it results directly from the derivation of the projection matrix.

To better grasp this issue of depth resolution lets rewrite equation 172) so that we have z_e written as a function of z'_w.

$$z'_w = s \times \left(\frac{-\frac{far+near}{far-near} \times z_e - \frac{2 \times near \times far}{far-near}}{-z_e} \times 0.5 + 0.5 \right) \tag{173}$$

Dividing everything by s and then subtracting both sides by 0.5 we get

$$\frac{z'_w}{s} - 0.5 = \frac{-\frac{far+near}{far-near} \times z_e - \frac{2 \times near \times far}{far-near}}{-z_e} \times 0.5 \tag{174}$$

Splitting the right side fraction provides

$$\frac{z'_w}{s} - 0.5 = 0.5 \times \frac{far+near}{far-near} + \frac{near \times far}{far-near} \times \frac{1}{z_e} \tag{175}$$

Moving the terms without z_e to the left side

$$\frac{z'_w}{s} - 0.5 - 0.5 \times \frac{far+near}{far-near} = \frac{near \times far}{far-near} \times \frac{1}{z_e} \tag{176}$$

Isolating z_e

$$z_e = \frac{near \times far}{\left(\frac{z'_w}{s} - 0.5 - 0.5 \times \frac{far+near}{far-near} \right)(far-near)} \tag{177}$$

$$z_e = \frac{near \times far}{\left(\frac{z'_w}{s} \times (far-near) - 0.5 \times (far-near) - 0.5 \times (far+near) \right)} \tag{178}$$

$$z_e = \frac{near \times far}{\frac{z'_w}{s} \times (far - near) - far} \quad (179)$$

If $z'_w = 0$, and using eq. 179) we should have $z_e = -near$

$$z'_w = 0 \Longrightarrow z_e = \frac{near \times far}{-far} = -near \quad (180)$$

On the other hand if $z'_w = s$, then $z_e = -far$.

$$z'_w = s \Longrightarrow z_e = \frac{near \times far}{(far - near) - far} = -far \quad (181)$$

This helps to verify equation 179).

As mentioned before, since the depth is represented with integers, it makes sense to consider rounding to the closest integer as a solution.

Equation 179) allows us to examine the precision we get with different z_w values as well as different bit depths. Assume that the depth is stored with 16 bits, $near = 1$ and $far = 1000$.

When we have a 16 bit depth we get $s = 2^{16} - 1 = 65535$. To get the depth interval values represented by the first and last depth values we need to compute z_e for $z'_w = 0.5$ and $z'_w = 65534.5$.

When $z'_w = 0.5$ we get

$$z'_w = 0{,}5 \Longrightarrow z_e = \frac{near \times far}{\frac{0.5}{s} \times (far - near) - far} = -1.000007622 \quad (182)$$

What this implies is that for values of z_e in $[-1.000007622, -1]$ the z_w value stored on the z buffer is 0. Considering our units to be meters the precision is superior to a 100th of a millimetre which is quite good.

Now let's take a look at the other end of the spectrum. If $z'_w = s - 0.5 = 65534.5$, i.e., the beginning of the last depth interval

$$z'_w = s - 0.5 \Longrightarrow z_e = \frac{near \times far}{\frac{s-0.5}{s} \times (far - near) - far} = -992{,}4357722 \quad (183)$$

This implies that for values of z_e in $[-1000, -992{,}4357722]$ the z_w value stored on the z buffer is 65535. This is almost an 8 unit gap, so we might get into trouble with objects at these distance. For instance, the store depth is the same for an object with $z_e = -994$ or $z_e = -998$. Depending on the order these objects are drawn we will get different results because the z comparison will not be able to distinguish between the two.

Let's push it a little bit further and consider $near = 0{,}01$

$$z'_w = 0.5 \Longrightarrow z_e = -0.010000076 \quad (184)$$

$$z'_w = s - 0.5 \Longrightarrow z_e = -567.2331641 \quad (185)$$

This implies that for values of z_e in $[-1000, -567.2331641]$ the z_w value stored on the z buffer is 65535. Now this is almost half of the whole depth interval we are visualizing and will most likely cause unexpected results when rendering a scene.

Now let's increase the bit depth a little bit. Assuming 24 bits, $ear = 1$, $far = 1000$

$$z'_w = 0.5 \Longrightarrow z_e = -1.000000030 \quad (186)$$

$$z'_w = s - 0.5 \Longrightarrow z_e = -999.970228364 \quad (187)$$

Again, setting $near = 0{,}01$ provides worse results, but far better than with 16 bit depth buffer.

$$z'_w = 0.5 \Longrightarrow z_e = -0{,}010000001 \quad (188)$$

$$z'_w = s - 0.5 \Longrightarrow z_e = -997.028652606 \quad (189)$$