Intro3D

Tommy Petersen
tp@ai-agents.com

November 14, 2009

Abstract

This article presents methods on how to make a simple “flyable” 3D world from scratch using only basic skills from mathematics and programming.

1 Introduction

If you want to make a 3D world from scratch, then this article may be of use for you. It is assumed that you have some basic skills within mathematics and programming. From mathematics, you should know of simple geometry and matrix computation. From programming, you should have some basic knowledge of graphics programming in some programming language. Essentially, you should be able to draw a point on a graphics screen using some palette of colors.

Relying mostly on your mathematical skills, this article will present you with some methods, that you can use in order to make a simple 3D world consisting of lines and points, in which a user can “fly” around as he or she wishes.

2 Projection of a point

Every point in the 3D world can be described by three coordinates, but the user sees the 3D world on a 2D screen. This section presents a way of projecting any given 3D point to a 2D point in such a way, that an entire 2D image of projected points will look like 3D.

2.1 Camera

The user is modelled by a camera which is capable of being positioned anywhere and in any orientation in the 3D world. The camera has an eyepoint and a projection area, which are positioned in such a way that the eyepoint is the apex of an infinitely large pyramid containing the projection area as a cut through it. The camera is illustrated in figure 1. The idea is, that the camera is only able to see the points

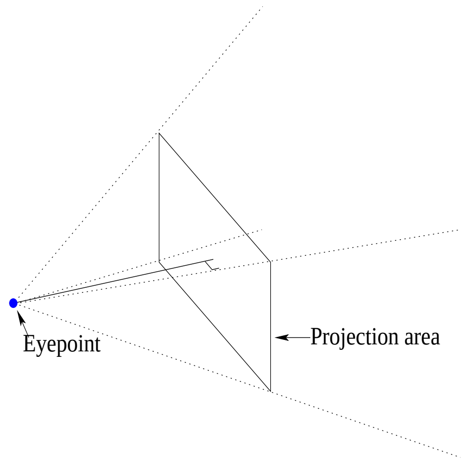

Figure 1: An illustration of the camera.

contained in the pyramid excluding the eyepoint. Every such point is projected onto the projection area which can readily be mapped onto a physical screen, since it is a rectangular 2D area.

2.2 Defining the projection point

So how is the projection of a point defined? Well, take any point inside the pyramid, and draw the line through that point and the eyepoint of the camera. This line will intersect the projection area in a single point, which is the projection point. On figure 2, the projection point is marked with a cross. If two different 3D points

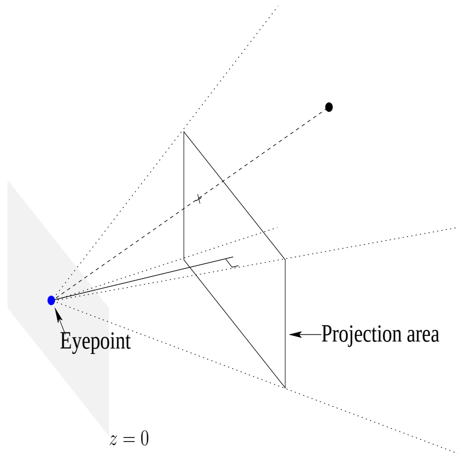

Figure 2: Illustrating the projection of a point.

map to the same projection point, then the 3D point closest to the plane $z = 0$ is the one which is actually mapped. This can be significant, if for instance the two 3D points have different colors and if the projection point gets the color of the chosen 3D point.

2.3 Computing the projection point's coordinates

But what about the coordinates? We need a way to compute the 2D coordinates of the projected point given the camera and the 3D point. In order to do this, we first define the camera's local coordinate system as the three mutually perpendicular unit vectors having the eyepoint as origo as illustrated on figure 3. Secondly, also on figure 3, we define the number P as the distance between the eyepoint and the projection area. Given the coordinate (x, y, z) of the 3D point in the camera's local coordinate system, we want to map it to some 2D coordinate (x', y') in the projection area.

What characterizes the points in the projection area? Well, if we see them as 3D points with respect to the camera's local coordinate system, they all have their third coordinate equal to P. So we want the projection point to have the third coordinate equal to P. This will be the case if we multiply the 3D point's z-coordinate by $\frac{P}{z}$. We must also remember the x- and y-coordinates of the 3D point. Since the projection follows a line we can also multiply the x- and y-coordinates

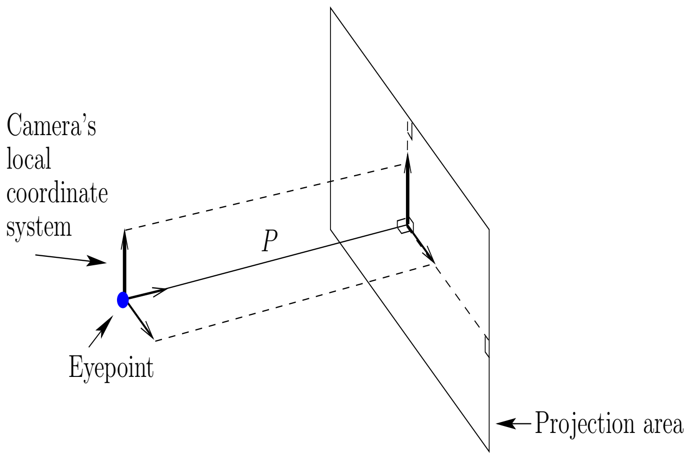

Figure 3: The camera's local coordinate system and the distance P between the eyepoint and the projection area.

by $\frac{P}{z}$. Hence the projection point is

$$(x \cdot \frac{P}{z},\ y \cdot \frac{P}{z})$$

2.4 The z-buffer datastructure

The projection area must be mapped to the physical screen in order for the user to see the graphics on the screen. The coordinates for each of the projection points in the projection area are real valued and must be mapped to integer screen coordinates.

A way to do the map is to use the so called z-buffer datastructure which contains two two dimensional integer arrays. The arrays each have the same number of rows and columns as the respective numbers of rows and columns in the screen plus an extra one. The first array, the "color" array, has a color value in each entry, so it contains the actual graphics image. The second array, the "depth" array, has a real valued depth value for each entry. It is initialized with the maximum real value. The depth array is used in order to keep track of the depth of the closest 3D point which hitherto has been projected to the screen coordinate as given by the entry's array indexes. See figure 4

For example, assume that some "new" 3D point is projected onto a projection point, which is then mapped to a z-buffer entry already "occupied" by some other 3D point. Also assume, that the new 3D point has an even smaller depth than is already in the depth array at the z-buffer entry. Then the color and the depth array are both updated with the values (color respectively depth) of the new 3D point.

As mentioned previously, the projection area consists of real valued coordinates. Furthermore, let X denote the width, and Y denote the height of the projection area. Then the coordinates have x coordinate between $-\frac{X}{2}$ and $+\frac{X}{2}$, and y coordinate between $-\frac{Y}{2}$ and $+\frac{Y}{2}$. Since the z-buffer consists of integer valued coordinates having x coordinate between 0 and the width of the screen, and y coordinate between 0 and the height of the screen, it follows, that the projection point must be

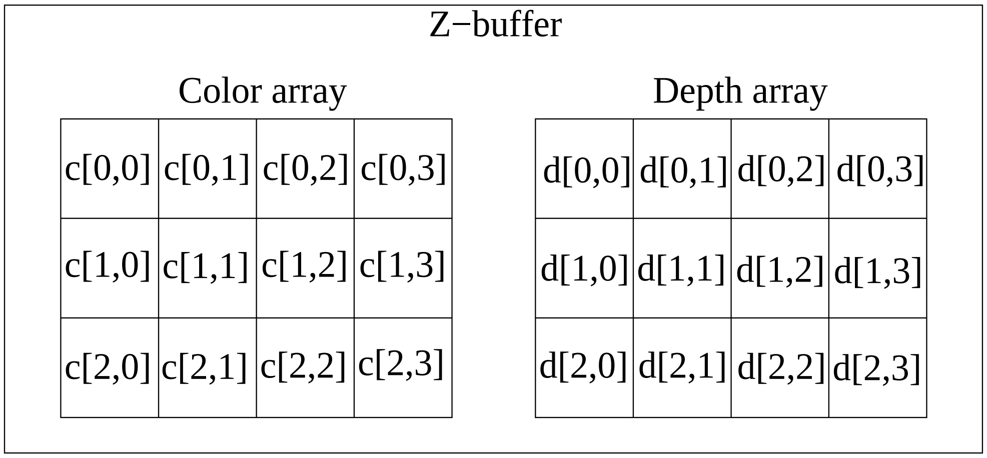
Z−buffer
Color array
Depth array
c[0,0] c[0,1] c[0,2] c[0,3]
c[1,0] c[1,1] c[1,2] c[1,3]
c[2,0] c[2,1] c[2,2] c[2,3]
d[0,0] d[0,1] d[0,2] d[0,3]
d[1,0] d[1,1] d[1,2] d[1,3]
d[2,0] d[2,1] d[2,2] d[2,3]

Figure 4: The z-buffer datastructure for a small screen with two rows and three columns.

translated, scaled and rounded to the nearest integer point in the z-buffer. Then it can be decided if the z-buffer is to be updated with the new point.

3 Projection of a line

In this section it is shown how to project a line (actually a line piece). Since different parts of a line can be both inside and outside the camera's pyramid, some extensions of the point projection are required.

3.1 When the line is fully contained in the camera's pyramid

When the line is contained in the camera's pyramid, then the line's endpoints are first of all projected to the z-buffer. Hereafter, the line piece connecting them is explicitly drawn within the z-buffer. Figure 5 shows a line with two endpoints, and the projection of these two endpoints. Figure 6 shows the two projected endpoints in the z-buffer and the explicitly drawn line piece connecting them.

3.2 When the line is not fully contained in the camera's pyramid

It may very well be the case, that some part of the line is inside the camera's pyramid whereas some other part is outside. Figure 7 shows an example of this situation. In order to handle this situation, the set of 3D points looked on when projecting a line is extended to include all 3D points.

We consider three cases: one where both of the line's endpoints have z greater than 0, one where both of the line's endpoints have z less than or equal to 0 and one where one of the endpoints has z greater than 0 and the other endpoint has z less than or equal to 0.

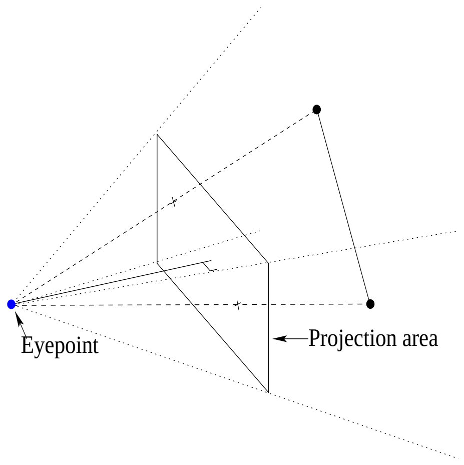

Figure 5: The line's endpoints are projected to the projection area.

3.2.1 When both endpoints have z greater than 0

In figure 7, z is greater than 0 for both endpoints. The line through the outside 3D point and the eyepoint intersect the plane $z = P$ at the cross, which is outside the projection area. If we compute the z-buffer coordinate for this projection point (by translating, scaling and rounding it to integer coordinates), then the computed coordinate will be outside the z-buffer. So one part of the line extending from the coordinate inside the z-buffer to the coordinate outside the z-buffer is inside the z-buffer, and another part is outside the z-buffer. In order to explicitly draw the part of the line which is inside the z-buffer, we can just find the point on the right edge of the z-buffer, where the line leaves the z-buffer. Then a line can be drawn between the inside point and the point on the z-buffer edge. Figure 8 shows this.

In general, there are many more possibilities than the one shown in figure 7. For example, both of the 3D line's endpoints can lie outside the camera's pyramid, while a part of the line is inside the pyramid. This requires computing two new endpoints on the edge of the z-buffer and thereafter drawing the line between them explicitly inside the z-buffer.

As a help to compute new endpoints, think of the extended z-buffer area is the two dimensional plane of integer coordinates into which the extended set of 3D points can be mapped. This area is divided into nine non-overlapping areas as illustrated in figure 9. Depending on the projected endpoints, the line between them may or may not overlap the original z-buffer area. Figure 9 is a good help in order to see how to compute new endpoints on the edge of the original z-buffer. Also, the formula for a line is a good help for this computation. The formula for a line with endpoints $(x_0,\ y_0)$ and $(x_1,\ y_1)$ is

$$f(x) = y_0 + a \cdot (x - x_0)$$

where

$$a := \frac{y_1 - y_0}{x_1 - x_0}$$

3.2.2 When both endpoints have z less than or equal to 0

In this case, the line is not projected.

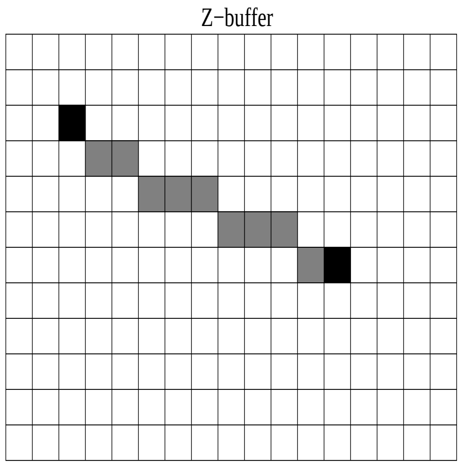

Figure 6: The projected endpoints from figure 5 are shown in black. The line piece connecting the endpoints is explicitly drawn within the z-buffer and shown in grey. The array shown represents both the color and the depth array.

3.2.3 When one of the endpoints has z greater than 0 and the other endpoint has z less than or equal to 0.

This case is illustrated in figure 10. Ideally, all of the part of the line with z greater than 0 should be projected onto the projection area. This part of the line gets infinitely close to the plane $z = 0$, and due to this infinity, we use an approximation.

For some positive value F, we use the plane $z = F$ in order to compute an approximation to the part of the line having z greater than 0. We use the intersection of the line through the two endpoints and the plane $z = F$ as a new endpoint, replacing the endpoint having z less than or equal to 0. This new endpoint has z greater than 0 and the resulting new line can then be projected as in the cases when both endpoints have z greater than 0. Figure 11 illustrates the idea. In order to compute the new endpoint, let the two endpoints be described by the coordinates (x_0, y_0, z_0) and (x_1, y_1, z_1) respectively (it does not matter which is which). Now define the number t to be

$$t := \frac{F - z_0}{z_1 - z_0}$$

Then compute the numbers x and y as

$$x := (1 - t) \cdot x_0 + t \cdot x_1$$

$$y := (1 - t) \cdot y_0 + t \cdot y_1$$

For non-negative F between z_0 and z_1 the point (x, y, F) (in the camera's coordinate system) will then be the intersection point we are looking for, and then we have the new endpoint.

Due to this last case, we make a slight change of focus on the cases relevant for projecting a line. We refer to the part of the camera pyramid having z greater than or equal to F as the bottom of the camera pyramid. The new cases are then:

- **When the line is fully contained in the bottom of the camera pyramid.**
  Project both endpoints to the z-buffer.

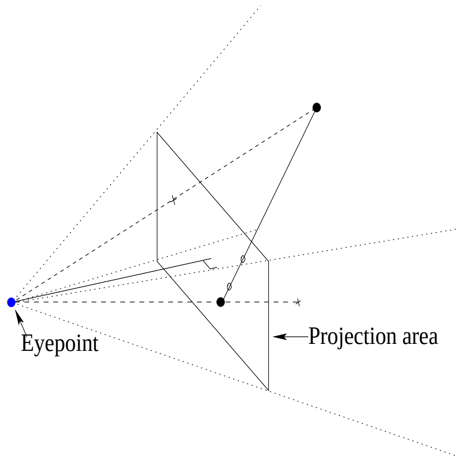

Figure 7: One part of the line is outside the camera's pyramid. The line's intersections with the projection area and with the rigth side of the pyramid are shown as circles.

- **When the line is not fully contained in the bottom of the camera pyramid.**
  Consider the following subcases:
  - **When both endpoints have z greater than or equal to F.**
    Project using the extended z-buffer illustrated in figure 9.
  - **When both endpoints have z less than F.**
    Do not project the line.
  - **When one of the endpoints has z greater than or equal to F and the other endpoint has z less than F.**
    Replace the endpoint having z less than F by the point which is the intersection of the plane $z = F$ and the line through the original endpoints. Project using the new point and the original point that has z greater than or equal to F.

4 A flyable camera

We have seen how to project a world of points and lines onto the camera (and hence onto the physical screen). In this section we will see how to make the camera flyable.

4.1 Requirements for the flyable camera

Consider figure 12. The 3D world has a world coordinate system in which every point is described by a coordinate. Likewise, the camera has a coordinate system. The camera's coordinate system describes points relative to the camera's position and orientation. For a decent flyable camera we require that it can change position so that it is able to fly forward, backward, sidewise and up and down. Also, we require, that the camera is able to change orientation so that it can rotate along any of it's three axes. Of course, camera movement should be relative to the camera's present position and orientation.

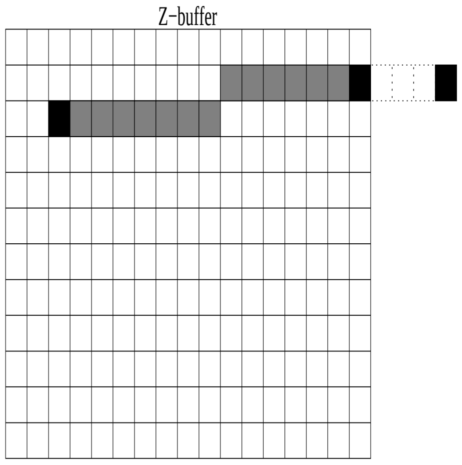

Figure 8: The point on the edge of the z-buffer where the line between the two projected points from figure 7 leaves the z-buffer. The line between the inside projected point and the point on the edge is explicitly drawn inside the z-buffer.

4.2 Changing the camera's position

We want to be able to change the camera's position so that it is able to fly forward, backward, sidewise and up and down. This can be achieved by a translation using a translation point. Allowing the translation point to be an arbitrary 3D point will cause the translation to be general, such that the camera in principle can change position in any direction.

We will make the general translation since it is very easy to do so. The specific translation points for flying forward, backward, sidewise and up and down can then be used in the general setup to achieve the requested changes in position.

In figure 13 the camera is translated using translation point T also described by translation vector $t := \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}$ in the world coordinate system. The 3D point P is described by the vector $p := \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}$. Assuming the camera has been translated having the same orientation as the world coordinate system, the point P is described in the camera's coordinate system by the vector b, which is given by

$$b = p - t = \begin{pmatrix} p_1 - t_1 \\ p_2 - t_2 \\ p_3 - t_3 \end{pmatrix}$$

So when the camera is to be translated using an arbitrary translation point T, then we subtract T's coordinates (t_1, t_2, t_3) from the coordinates of every other point in the 3D world. In this way, we obtain the 3D world as seen through the camera (described using the camera's local coordinate system). Note, that the subtraction can of course instead be done by adding the coordinates $(-t_1, -t_2, -t_3)$ to the coordinates of the points in the 3D world.

Using appropriate translation points, the requested abilities for change of camera position can then be implemented. The following lists the appropriate trans-

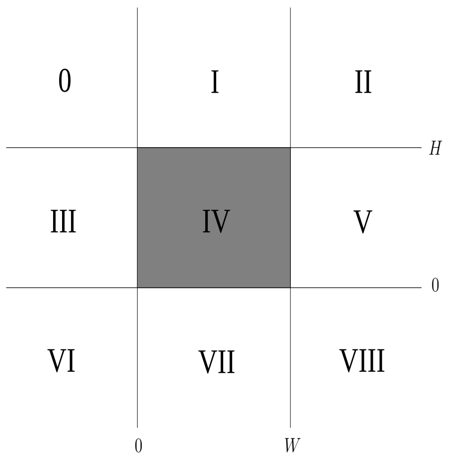

Figure 9: For a physical screen resolution of width W and height H, the division of the extended z-buffer area into nine non-overlapping areas is shown. The original z-buffer area is area number IV (the shaded area in the middle).

lation points which should be <u>added</u> to the points in the 3D world with respect to the world coordinate system (k is some positive real number):

fly forward	$(0, 0, -k)$
fly backward	$(0, 0, k)$
fly left	$(k, 0, 0)$
fly right	$(-k, 0, 0)$
fly up	$(0, -k, 0)$
fly down	$(0, k, 0)$

4.3 Changing the camera's orientation

We want to be able to rotate the camera along any of it's three axes. In this section rotation in 2D is described, and expanding on this, rotation in 3D is described.

When the camera is rotated along any axis, say the x-axis, then the points in the 3D world seem to rotate in the opposite direction. If the camera rotates down, then we can also think of the camera as sitting still while the 3D points rotate up with the same angle. This is of course as seen from the camera itself.

So using the camera's point of view we want to be able to rotate any 3D point along any of the camera's three axes. We start out by describing how to rotate a point in 2D. Consider figure 14. The point (e, f) has angle ϕ to the u-axis and is rotated into the point (e', f') by the angle θ. The point (e, f) is given by

$$(e, f) = (r\cos\phi,\ r\sin\phi)$$

and the point (e', f') is given by

$$(e', f') = (r\cos(\theta + \phi),\ r\sin(\theta + \phi))$$

Elaborating on e' and f' gives us the following:

$$\begin{aligned} e' &= r\cos(\theta + \phi) \\ &= r(\cos\theta\cos\phi - \sin\theta\sin\phi) \end{aligned}$$

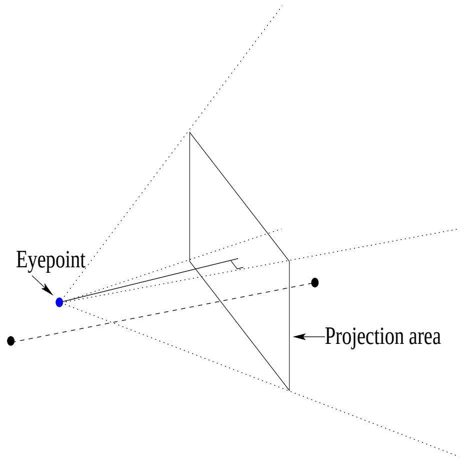

Figure 10: The part of the line having z greater than 0 should be projected (possibly be use of the extended z-buffer). The part having z less than or equal to 0 is invisible to the camera and should therefore not be projected.

$$\begin{aligned} &= r\cos\theta\cos\phi - r\sin\theta\sin\phi \\ &= e\cos\theta - f\sin\theta \end{aligned}$$

and

$$\begin{aligned} f' &= r\sin(\theta+\phi) \\ &= r(\sin\theta\cos\phi + \cos\theta\sin\phi) \\ &= r\sin\theta\cos\phi + r\cos\theta\sin\phi \\ &= e\sin\theta + f\cos\theta \end{aligned}$$

This result is summarised as follows:

$$(e,\ f) \overset{rotation\,\theta}{\longrightarrow} (e\cos\theta - f\sin\theta,\ e\sin\theta + f\cos\theta)$$

4.3.1 Rotating a 3D point along the x-axis

Looking down towards the origo from the positive part of the x-axis we say that a **positive rotation** along the x-axis is a counterclockwise rotation in the zy-plane. Similarly, a **negative rotation** along the x-axis is a clockwise rotation in the zy-plane. See figure 15.

When rotating a 3D point $(a,\ b,\ c)$ along the x-axis, the a-coordinate is unchanged, while the other coordinates b and c can change. We want to use our result above about rotation in 2D. Since we now actually want to make a rotation in the zy-plane, we let the c-coordinate correspond to the e-coordinate and we let the b-coordinate correspond to the f-coordinate. This gives us the following description of the rotation's new coordinates:

$$(a,\ b,\ c) \overset{rotation\,\theta}{\longrightarrow} (a,\ c\sin\theta + b\cos\theta,\ c\cos\theta - b\sin\theta) \tag{1}$$

4.3.2 Rotating a 3D point along the y-axis

Looking down towards the origo from the positive part of the y-axis we say that a **positive rotation** along the y-axis is a counterclockwise rotation in the xz-plane.

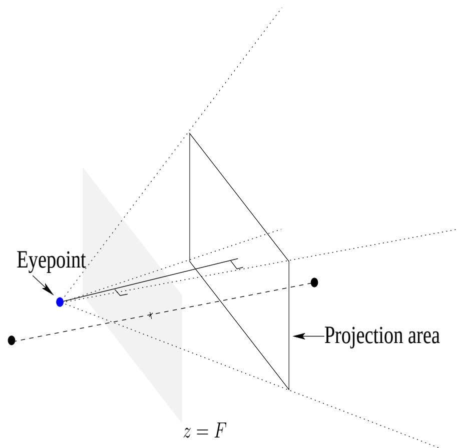

Figure 11: Using the plane $z = F$ to approximate the projection of the part of the line having z greater than 0.

Similarly, a **negative rotation** along the y-axis is a clockwise rotation in the xz-plane. See figure 16.

When rotating a 3D point $(a,\, b,\, c)$ along the y-axis, the b-coordinate is unchanged, while the other coordinates a and c can change. As before, we use our result above about rotation in 2D. Since we now actually want to make a rotation in the xz-plane, we let the a-coordinate correspond to the e-coordinate and we let the c-coordinate correspond to the f-coordinate. This gives us the following description of the rotation's new coordinates:

$$(a,\, b,\, c) \overset{rotation\,\theta}{\longrightarrow} (a\cos\theta - c\sin\theta,\, b,\, a\sin\theta + c\cos\theta) \tag{2}$$

4.3.3 Rotating a 3D point along the z-axis

Looking down towards the origo from the positive part of the z-axis we say that a **positive rotation** along the z-axis is a counterclockwise rotation in the yx-plane. Similarly, a **negative rotation** along the z-axis is a clockwise rotation in the yx-plane. See figure 17.

When rotating a 3D point $(a,\, b,\, c)$ along the z-axis, the c-coordinate is unchanged, while the other coordinates a and b can change. Once again, we use our result above about rotation in 2D. Since we now actually want to make a rotation in the yx-plane, we let the b-coordinate correspond to the e-coordinate and we let the a-coordinate correspond to the f-coordinate. This gives us the following description of the rotation's new coordinates:

$$(a,\, b,\, c) \overset{rotation\,\theta}{\longrightarrow} (b\sin\theta + a\cos\theta,\, b\cos\theta - a\sin\theta,\, c) \tag{3}$$

4.4 Storing a sequence of camera movements

At any given time, the user has made n camera movements. Hence we can represent flying around with the camera as a sequence of camera movement events:

$$E_1\, E_2\, \cdots\, E_n$$

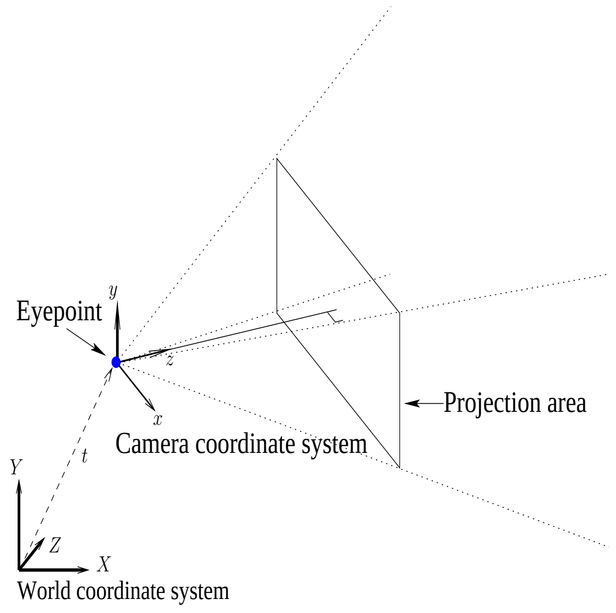

Figure 12: The world coordinate system and the camera coordinate system. The camera's position is given by the vector t relative to the world coordinate system.

Each E_i is either a camera translation (changing the camera's position) or a camera rotation (changing the camera's orientation).

For any given 3D point P think of E_i as a function applied to P giving a new 3D point P':

$$P' = E_i(P)$$

Using function composition on the whole sequence of events, we get the 3D point P' which is how P will look like from the camera's viewpoint after all it's movements have been done. P' is then the description of P with respect to the camera's coordinate system. The function composition starts with E_1 applied to P then E_2 applied to that result and so on:

$$P' = E_n\left(E_{n-1}\cdots\left(E_2\left(E_1(P)\right)\right)\cdots\right)$$

In the following, a way of representing the movement events will be described along with a way of storing a whole sequence of such events.

4.4.1 Representing movement events by matrices

As mentioned above, any movement event E_i is either a camera translation or a camera rotation. In this section it is described how to represent the possibilities by matrices.

Camera translation. We first consider E_i being a camera translation. As we saw earlier, this can also be thought of as translating any 3D point P in the "opposite" direction. Let $T := \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}$ be the translation vector used for translating

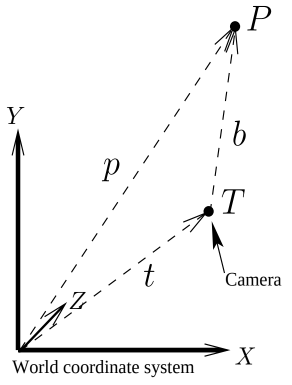

Figure 13: The camera is repositioned by translating it using translation point T with respect to the world coordinate system. The point P is described by vector b in the camera's coordinate system.

P. Then P is translated into the new point P' according to the following:

$$P' = T + P \tag{4}$$

Now, let P', P and T be represented by 3×1 matrices in the following natural way:

$$\begin{bmatrix} p'_1 \\ p'_2 \\ p'_3 \end{bmatrix} := \begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \end{pmatrix} = P'$$

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} := \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = P$$

$$\begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} := \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = T$$

Equation 4 is then still valid for the matrix versions of P', P and T.

We will then use the matrix representations of the 3D points and the translation vector together with equation 4 to represent and compute translations of 3D points.

Camera rotation. Now we consider E_i being a camera rotation. As mentioned earlier, this can also be thought of as rotating any 3D point P by the "opposite" angle.

Let P be given by $P := (a, b, c)$. Then equations 1, 2 and 3 describe which points will be the result of rotation by angle θ along the axes x, y and z respectively. For each axis, we want to find a 3×3 matrix M such that the following matrix-equations are valid:

For the x-axis:

$$MP = \begin{bmatrix} a \\ c\sin\theta + b\cos\theta \\ c\cos\theta - b\sin\theta \end{bmatrix} \tag{5}$$

For the y-axis:

$$MP = \begin{bmatrix} a\cos\theta - c\sin\theta \\ b \\ a\sin\theta + c\cos\theta \end{bmatrix} \tag{6}$$

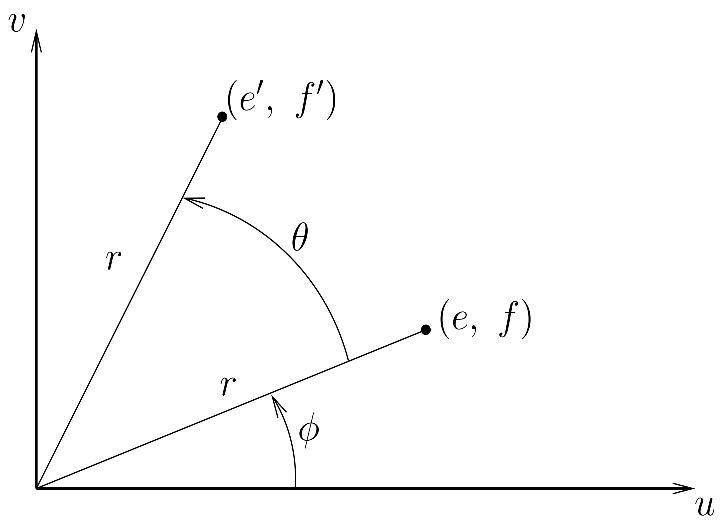

Figure 14: The point (e, f) is rotated into the point (e', f') by the angle θ.

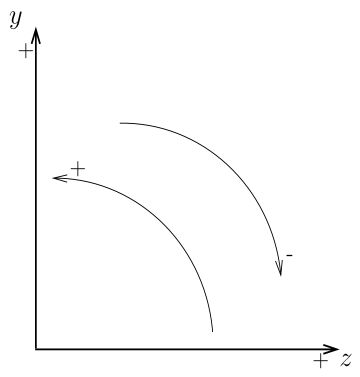

Figure 15: Rotation in the zy-plane. A positive rotation is a counterclockwise rotation, while a negative rotation is a clockwise rotation.

For the z-axis:

$$MP = \begin{bmatrix} a\cos\theta + b\sin\theta \\ b\cos\theta - a\sin\theta \\ c \end{bmatrix} \tag{7}$$

(Remember, that multiplying the 3×3 matrix M by the 3×1 matrix P gives a 3×1 matrix as result.)

A little analysis gives the following matrices:

For the x-axis:

$$M := \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}$$

For the y-axis:

$$M := \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

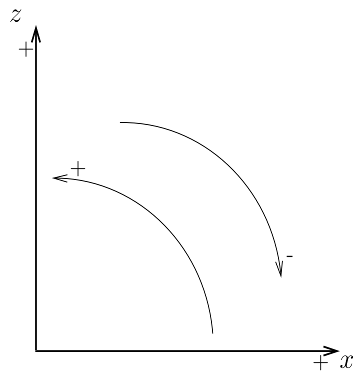

Figure 16: Rotation in the xz-plane. A positive rotation is a counterclockwise rotation, while a negative rotation is a clockwise rotation.

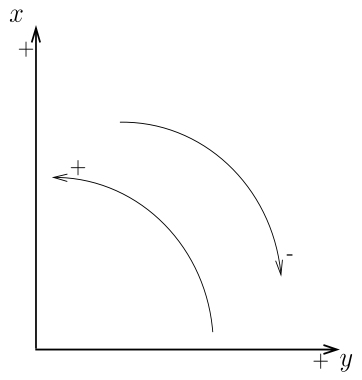

Figure 17: Rotation in the yx-plane. A positive rotation is a counterclockwise rotation, while a negative rotation is a clockwise rotation.

For the z-axis:

$$M := \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(You can check that these matrices are valid by inserting them into equations 5, 6 and 7 respectively and carrying out the corresponding matrix multiplications.)

We will then use the matrix representations of the 3D points and the rotation together with equations 5, 6 and 7 to represent and compute rotations of 3D points.

4.4.2 Storing a sequence of movement events in two matrices

Using the matrix representations of movement events E_i, we will now show how to store a sequence of movement events $E_1\, E_2\, \cdots\, E_n$ in two matrices.

Let T be any 3×1 matrix, let M be any 3×3 matrix, and let P be any 3D point. Define the 3×1 matrix to be

$$P' := T + MP \qquad (8)$$

Assume that event E_i is a translation, represented by the 3×1 matrix T_i. The result of translating P' by T_i is the 3×1 matrix $T_i + P'$. Elaborating on this gives us the following:

$$\begin{aligned} T_i + P' &= T_i + (T + MP) \\ &= (T_i + T) + MP \end{aligned}$$

The matrix $T_i + T$ is a 3×1 matrix and hence the resulting matrix $(T_i + T) + MP$ is on the same form as $T + MP$ in equation 8.

Now assume, that the event E_i is a rotation represented by the 3×3 matrix M_i. The result of rotating P' by M_i is the 3×1 matrix M_iP'. Elaborating on this gives us the following:

$$\begin{aligned} M_iP' &= M_i(T + MP) \\ &= M_iT + M_i(MP) \\ &= M_iT + (M_iM)P \end{aligned} \tag{9}$$

The matrix M_iT is a 3×1 matrix and the matrix M_iM is a 3×3 matrix. Hence the resulting matrix $M_iT + (M_iM)P$ is on the same form as $T + MP$ in equation 8.

Having seen that the form of the expression in equation 8 is preserved when updating it with new movement events, we will now present two matrices that can be used to initialize this expression. Let the 3×1 matrix T_0 be defined by

$$T_0 := \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

and let the 3×3 matrix M_0 be defined by

$$M_0 := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then the following equation is valid for any 3D point P:

$$P = T_0 + M_0P \tag{10}$$

In the beginning, when the camera has not yet been moved, it sits in the world origo with it's local coordinate system aligned with the world coordinate system. So in the beginning, every 3D point will have the same coordinates in both the camera's and in the world coordinate system. Equation 10 then shows us, that matrices T_0 and M_0 are correct initializations of T and M.

We can conclude, that we can store all movement events in two matrices: A 3×1 matrix T and a 3×3 matrix M initialized by T_0 and M_0 respectively. For each movement event E_i, we update T and M according to the following rules:

- **When E_i is a translation T_i:**

$$T := T_i + T$$

- **When E_i is a rotation M_i:**

$$T := M_iT$$

$$M := M_iM$$

4.5 Generating camera movements

Having seen how to store a whole sequence of camera movements, this subsection briefly outlines a way of generating these movements.

Generating translations. The translations can be used in order to fly the camera forward and backward, up and down and from side to side. For example, you could decide to use the keys on the keyboard to generate camera translations using the following table (k is some positive real number):

Key	**Camera movement**	**3D point translation**
w	fly forward	$(0, 0, -k)$
s	fly backward	$(0, 0, k)$
a	fly left	$(k, 0, 0)$
d	fly right	$(-k, 0, 0)$
r	fly up	$(0, -k, 0)$
f	fly down	$(0, k, 0)$

Generating rotations. The rotations can be used to rotate the camera along it's three axes. To generate some of the rotations, you could use the mouse according to the following table:

Mouse movement	**Camera movement**	**3D point rotation**
left	rotate left	rotate by $-\theta$ along the y-axis
right	rotate right	rotate by θ along the y-axis
up	rotate up	rotate by $-\theta$ along the x-axis
down	rotate down	rotate by θ along the x-axis

To generate rotation along the z-axis you could use the keys "q" and "e" as shown in the following table:

Key	**Camera movement**	**3D point rotation**
q	tilt left	rotate by θ along the z-axis
e	tilt right	rotate by $-\theta$ along the z-axis

4.6 Using the stored sequence to transform a 3D point

Consider figure 12. Assume that we have a 3D point given with respect to the world coordinate system. After a sequence of camera movements, we want to know the coordinates of the 3D point with respect to the camera's local coordinate system.

Let the sequence of camera movements be described by the following:

$$E_1\, E_2\, \cdots\, E_n$$

Each E_i is either a translation T_i or a rotation M_i. But no matter what E_i is, we have the whole sequence stored in the two matrices T and M. Let P denote the given 3D point. We want to transform P into another 3D point P' that describes P relative to the camera's local coordinate system. The transformation is simply given by the following:

$$P' := T + MP$$

Now, we can project P' onto the camera's projection area if we need to. This will show P as seen from the camera.

5 Summary

Here, at the end of the article, a short summary will be given.

Some methods have been presented that can be used in order to make a simple 3D world consisting of lines and points. Also, it has been described how to enable a user to “fly” around as he or she wishes. The methods have been grouped into two main areas:

- **Projection**
- **Camera movement**

The area on projection showed methods related to projecting 3D points and 3D lines, assuming that the 3D points were relative to the camera’s local coordinate system. The other area, on camera movement, showed methods that, no matter how the camera has been moved about, can be used in order to take any given 3D point relative to the world coordinate system, and transform that 3D point to a 3D point in the camera’s local coordinate system. Hence, the two areas combined present methods that can help to make a flyable simple 3D world.