

Chapter 1

projection

Two-thirds of the rendering procedure have fallen into place: on one hand, we have a general method (affine transformation) for moving around in a three-dimensional scene; and on the other hand, we know how to apply scan conversion to convert a two-dimensional geometrical description of our scene into an array of pixels suitable for display on the output device. What is still lacking is a way to get from three to two dimensions, from the spatial description of the scene to the planar snapshot of the scene that would be visible to an observer at our chosen point of view. In other words, we need a mathematical description of how to take a photograph.

1.1 The basic pinhole model

The essential behavior of a camera can be analyzed readily with the model of a pinhole camera, probably the simplest optical device imaginable, consisting of two parallel planes, one of which has a small hole pierced through it as shown in Figure ???. The pierced plane in front screens out all incoming light rays except those which pass through the hole. A piece of film is placed on the back plane; the color and intensity of a specific point on the film is proportional to the amount of light traveling along the ray connecting the pinhole to that point.

When the film is developed, the image is reflected. As a result, one can think of the geometry of measurement process as shown in Figure ??, with the film plane *in front* of the pin hole. Each image point measures the light along the ray connecting the film point to the pinhole. This interpretation shows clearly why a pinhole image looks like a 3d snapshot to a human. If in Figure ??, we replace the film with the final image, and we replace the pinhole with the eye of a human observer, we see that the observer will see along each ray exactly what would have been seen in the captured 3D scene. If the eye is not placed exactly where the pinhole was, then this will not be an exact reconstruction, but it usually still looks quite realistic.

Our goal then is to mathematically model the projection process of a pinhole camera. In particular, we need to determine mathematically where an object at a specified point in space shows up on the film plane of a pinhole camera with the pinhole at our viewpoint.

We have our choice of coordinate systems, so we might as well select one in which the camera is oriented in a way that will make our calculations easier. Specifically, we will assume that we are using the eye coordinate system \mathbf{e}^t . In this coordinate system,

1. The camera shall sit at the origin of our coordinate system.
2. The camera shall look straight down the negative z -axis.
3. The film plane shall be the plane $z = -1$.

The first of these is quite natural: if we allowed the camera to be wherever it wanted, we'd have to drag around three extra variables (x -, y -, and z -coordinates) to describe the location of the camera, and these extra variables would clutter up our formulae a lot. Similar reasons justify fixing the direction in which the camera is pointing—we avoid another two or three parameters this way. If we want to continue to use the x - and y -axes as the coordinate axes in the two-dimensional image plane, as we did earlier, then the camera direction, which is perpendicular to the image plane, needs to be the direction perpendicular to both the x -axis and the y -axis—namely, the z -axis. Our choice of sign makes the coordinate system right-handed.¹ The last constraint places the film plane in front of the pinhole, which describes the “post-development” geometry.

The configuration resulting from our three assumptions is depicted in Figure [nothereyet]. Here \tilde{p} is intended to be a general point in the 3D scene. Here and in the sequel we shall customarily use coordinates $[x_e, y_e, z_e]^t$ to express the position of such a point with respect to the eye coordinate system.

During projection we transform $\tilde{p} \Rightarrow \tilde{q}$ where \tilde{q} is the corresponding point on the image plane. [Description of figure follows.] We will use a subscript i on the coordinate vector of a projected point in the image plane. If we want to be pedantic, we should specify points on the image plane not as $[x_i, y_i]^t$ but as $[x_i, y_i, -1]^t$, because in our geometrical configuration they are points in three-dimensional space that lie on the plane $z = -1$. In what follows, however, we shall usually suppress the -1 , which tells us nothing that we didn't already know, and write simply $[x_i, y_i]^t$.

1.1.1 pin hole transformation

What we want is a formula in terms of x_e, y_e , and z_e , which we know, for x_i and y_i , which we don't. This problem can be dispatched quickly with a dose of high-school geometry once we observe that the triangles OQ_iR_i and OQ_eR_e are similar. The

¹Some APIs have the camera looking down the positive Z axis, with the x axis pointing right and the y axis pointing down.

corresponding sides $\overline{Q_i R_i}$ and $\overline{Q_e R_e}$ have a (signed) lengths x_i and x_e , while the sides $\overline{O R_i}$ and $\overline{O R_e}$ have lengths -1 and z_e . Consequently we have $x_i/(-1) = x_e/z_e$ by similarity, which gives us the value of x_i . By analogous logic applied to a slightly different diagram, we can obtain the companion equation $y_i/(-1) = y_e/z_e$. Thus we have finally that

$$\begin{aligned} x_i &= -x_e/z_e \\ y_i &= -y_e/z_e \end{aligned}$$

The transformations that we have investigated previously had the useful property of being expressible as matrix operations. For linear transformations this was automatic. General affine transformations were slightly more troublesome because a transformation performing a nontrivial translation is not linear in $[x, y, z]^t$; but because a translation is linear in $[x, y, z, 1]^t$, we could augment our coordinate vectors by a fourth coordinate fixed at 1 and express affine transformations as multiplication by suitable 4×4 matrices. Unfortunately, the projective transformation doesn't appear to have anything to do with matrix arithmetic because of that nasty division by z_e .

Nevertheless, we can sort of think of this projection transformation as being performed by the following matrix operation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ & & -1 & \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} x_i w_i \\ y_i w_i \\ - \\ w_i \end{bmatrix}$$

We have placed asterisks in “don't care locations”. After the multiplication, we have a coordinate vector made up of the values $[x_i w_i, y_i w_i, -, w_i]^t$. The image coordinates of the projected point can now be obtained by dividing out the w_i value.

1.1.2 All kinds of pin hole cameras

Suppose that I wish to model a pinhole camera where the pin hole is at the origin, but the film plane is at the $z = d$ plane. We will typically be thinking of d as a negative number. We can determine that the transformation for this camera is

$$\begin{aligned} x_c &= x_e d / z_e \\ y_c &= y_e d / z_e \end{aligned}$$

which can be expressed as the matrix equation

$$\begin{bmatrix} x_i w_i \\ y_i w_i \\ - \\ w_i \end{bmatrix} = \begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

Suppose that we wish to model a pin hole camera with the film plane at $z = -1$ as we did before, but then i wish to enlarge the image by a factor of s_x horizontally and s_y vertically. We can simply express this as the composition of two appropriate matrices. We can delay the division by w_i until the end.

$$\begin{bmatrix} x_i w_i \\ y_i w_i \\ - \\ w_i \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

If we choose $s_x = s_y = -d$ (recall that d is a negative number) we obtain

$$\begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

So we see that changing the focal distance is the same as applying a scale factor to the image.

Suppose we start with a pinhole camera with the film plane at $z = -1$. In this camera, the point on the film plane closest to the pinhole, called the center of projection, has eye coordinates $[0, 0, -1, 1]^t$. This point after projection will have image coordinates $[0, 0]^t$. Suppose we then wish to shift all of the image coordinates by $[c_x, c_y]^t$, now the center of projection will be at $[c_x, c_y]^t$. We call such a camera a “skewed” camera.

This can be obtained using the following matrix equation

$$\begin{bmatrix} x_i w_i \\ y_i w_i \\ - \\ w_i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -c_x & 0 \\ 0 & 1 & -c_y & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

There are a few reasons for modeling skewed cameras. First of all, in many real cameras, the center of projection is not in the actual center of the image rectangle. If we wish to model such cameras, we will typically use a skewed camera model. Secondly, in computer graphics there are situations where a skewed camera is needed to get the correct effect. For example in a flight simulator, there may be a computer screen off to one side of the cockpit which represents a window of the plane. The closest point of the screen to the users eye may not be in the center of the screen. If we were to simulate a camera where the pin hole is the user’s eye and the image plane is the computer screen, we would have a skewed camera model.

Another example where skewed cameras are useful is for stereo viewing. One typical stereo viewing situation has the user sitting with their nose centered with the

screen. The user wears special glasses which toggle between letting light in to the right and left eyes. Synchronized with these glasses, appropriate left-eye-images and right-eye-images are toggled on the screen. Since the users nose is centered on the screen, their left eye is somewhat to the left while their right eye is somewhat off to the right. If we simulate a camera for the left eye, with the pin hole in the user's left eye and the image plane on the screen, we will be using a skewed camera model.

1.2 Graphics pipeline

Now that we know how to model a pin hole projection, we are in a position to use it to create images of a 3d scene. To do this we must setup our pinhole transformation to work along with our world to eye matrix, and the viewport transform.

One begins with the world coordinates of some point \tilde{p}

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

This point can be expressed in the eye coordinate system using some modeling matrix M .

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = M \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

Projection is performed using a projection matrix P

$$\begin{bmatrix} x_i w_i \\ y_i w_i \\ - \\ w_i \end{bmatrix} = P \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

As we saw in chapter ??, we can find the pixel coordinates of a point given its image coordinates using the transformation

$$\begin{aligned} x_p &= \frac{w}{2} x_i + \frac{w-1}{2} \\ y_p &= \frac{h}{2} y_i + \frac{h-1}{2} \end{aligned}$$

This can be expressed as a matrix operation using the viewport matrix V .

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = V \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{w}{2} & 0 & 0 & \frac{w-1}{2} \\ 0 & \frac{h}{2} & 0 & \frac{h-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}$$

Thus the entire set of transformations from world to pixel coordinates can be expressed as

$$\begin{bmatrix} x_p w_i \\ y_p w_i \\ - \\ w_i \end{bmatrix} = \mathbf{VPM} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

After division by w_i , we have the coordinates in pixel coordinates which can be used for scan conversion.

We should note that in practice, these three matrices are not actually multiplied together to obtain a single matrix. This is done so that we can perform surface shading in the eye coordinate system. We cannot do shading using image or pixel coordinates since the projective transform does not preserve angles.

1.2.1 Frustum transform I

The viewport transformation assumes that the bounds of the picture in image coordinates are between -1 and 1; everything outside these bounds will not appear on the image. If we are going to use a projection matrix \mathbf{P} in concert with the viewport matrix, we should make sure that the projection matrix maps the desired region of the world to the canonical square.

This is often done using the notion of a frustum transformation. To setup this transformation, the user places the image plane at $z = n$, where n stands for the near plane. Again, typically n will be a negative number. The user then describes the boundaries of the valid image domain on the near plane. The “x” extent on the near plane is described with a left and right parameter, while the “y” extent is described with a bottom and top parameter.

$$\begin{aligned} l &< x_e < r \\ b &< y_e < t \end{aligned}$$

Given these extents, the pin hole transformation that maps this part of the image to the canonical square is

$$\begin{bmatrix} -\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

If the user wishes there to be no non-uniform scaling in the final image, one should choose the aspect ratio of the bounds $\frac{r-l}{t-b}$ to be equal to the aspect ratio of the final w by h pixel image.

If the camera is not a skewed camera then the frustum can be specified by simply giving the the vertical field of view of the camera θ , and the aspect ratio a of the desired output image. Using these paramters one can directly solve for frustum matrix as

$$\begin{bmatrix} \frac{1}{a \tan(\frac{\theta}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\theta}{2})} & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

1.3 The z -coordinate and its uses.

We have not bothered even to define the z -coordinate yielded by our projective transformations because geometrical projections do not produce one: they map $[x_e, y_e, z_e]^t$ to a new pair $[x_i, y_i]^t$, the z -coordinate having been normalized out of consequential existence in the mathematics of the projection. But in order to implement visibility computation, such as the z -buffer, we need to have some sort of depth value. A reasonable first effort in this direction involves filling in the previously neglected third column of the transformation matrix to produce the new matrix

$$\begin{bmatrix} x_i w_i \\ y_i w_i \\ z_i w_i \\ w_i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

This matrix now describes a full 3D to 3D transformation, taking us from $[x_e, y_e, z_e]^t$ to $[x_i, y_i, y_z]^t$ coordinates.

This transformation has the important property, that for all points in front of the eye, this 3D “projective” transformation preserves depth ordering. In particular for any two points lying in front of the eye and on a single ray from the eye, the point with the greatest (i.e., least negative) z -coordinate remains in front of the other; because we have $z_0 < z_1 \Leftrightarrow -1/z_0 < -1/z_1$. So the transformed z_i -coordinates can be used to perform correct z -buffer occlusion calculations for points lying along one ray from the eye.

This knowledge isn’t strong enough to prove that we can do z -buffering correctly with the new z -coordinates, for in scan conversion, we don’t calculate the transformed z for most points; instead, we calculate the transformed coordinates only for vertices of triangles in our model and interpolate linearly to get the coordinates for points corresponding to intermediate pixels. If the interpolated values do not agree with the true values, then the z -buffering calculations are likely to turn out wrong.

An example of what could conceivably happen is shown in Figure [nothereyet]. Here two intersecting line segments have been mapped to z -buffer values by a function that preserves the depth ordering; nevertheless, linear interpolation of the depth values of the vertices produces a spurious intersection well to the left of the correct intersection point, which will cause the rendered image to show the lines crossing over at the wrong place. The reason for this anomaly is evident in the diagram: the nonlinear curves produced by the depth mapping do not coincide with the lines produced by linear interpolation, so we can hardly expect the intersection points to coincide.

Fortunately our 3D projective transform is better behaved than this because it maps straight lines in 3D into straight lines in $3D^2$. Therefore the segments generated by linear interpolation are in fact the exact images of the original segments under the projective transformation.

1.3.1 clipping

Our z -transformation using the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

works just fine when all of the geometry is in front of the viewer, but it runs into trouble when some of the geometry spans across the $z_e = 0$ plane.

Since we are modeling a camera that only sees in one direction, we want to not draw any geometry that is behind the camera. Geometry behind the camera will have $z_i < 0$. So if all three vertices have $z_i < 0$, we can cull this primitive. What should we do if we are rendering a triangle that has at least one vertex in front of the viewer and at least one vertex behind the viewer. The correct thing to do would be to look at all of the points on the triangle, project them, and only keep those with $z_i > 0$. Will we get the same result if we simply project the vertices, perform linear interpolation, and cull points with interpolated $z_i < 0$?

To answer this question, lets see how this projective tranformation performs on some example z_e values. We will start from points very far in front of the camera and move to points very far behind the camera

$$\begin{aligned} z_e \approx -\infty &\Rightarrow z_i = \epsilon \\ z_e = -1 &\Rightarrow z_i = 1 \\ z_e = \epsilon &\Rightarrow z_i \approx +\infty \\ z_e = -\epsilon &\Rightarrow z_i \approx -\infty \\ z_e = 1 &\Rightarrow z_i = -1 \\ z_e \approx \infty &\Rightarrow z_i = \epsilon \end{aligned}$$

²This fact will be shown in section ??

We see that as points pass through the $z_e = 0$ plane, their z_i flip from being large positive to large negative numbers. So the proper mapping of a triangle with one vertex in front of and one behind the camera should actually pass through $z_i = \infty$. But this is not what we get when we only project the vertices and then use linearly interpolation for the intermediate values. When we project the two vertices, the one in front of the camera will have $z_i > 0$, and the one behind will have $z_i < 0$. When we linearly interpolate, we will be interpolating through $z_i = 0$, not through $z_i = \infty$; we are going the wrong way.

The robust solution to this problem takes somewhat more effort: to draw a triangle that straddles the included and excluded region, we must compute its intersection with the included region, dissect this intersection into triangles (for it may be a quadrilateral), and render the triangles by the usual method. This is called “clipping”, and must be performed for all such triangles.

1.3.2 z resolution

One problem with our transformation is that it makes poor use of the bits representing the z_i values. As z increases toward zero, the calculated z -buffer value increases without bound. This sort of behavior is tolerable on the average FPU (a `float` can exceed 10^{38} before anything breaks), but special-purpose rendering hardware typically uses fixed-point arithmetic with numbers in a small range such as $[-1, 1]$. Very large (and even moderately large) z -values are quite unacceptable in such a setting.

An even worse problem is that for objects far away from eye, with $z_e \ll 0$, depth distances get greatly compressed. When using a fixed point representation, a wide range of z_e depth values may get mapped to the exact same z_i fixed point value. In this case, we will be unable to properly resolve occlusion.

We can do better by first noting that because the function $z_i = -1/z_e$ preserves the ordering of z -values, so also does the function $z_i = -a + b/z_e$, where a and b are constants of our choice subject only to the constraint that b be positive. For suitable choices of a and b , the behavior of the computed z -buffering value may be more graceful in a large region of input values z . This can be achieved using the projection matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

In particular suppose we define a near plane at $z_e = n$ and a far plane at $z_e = f$ (typically n and f will be negative values) and we set

$$a = \frac{f + n}{f - n}$$

$$b = -\frac{2fn}{f-n}$$

Using these settings we will obtain a transformation that maps the range $z_e \in [f..n]$ reasonably well to $z_i \in [-1..1]$.

Then we only allow the system to render geometry within these depth bounds. All geometry outside of these bounds is culled, and geometry that straddles these bounds are clipped.

1.3.3 Frustum II

If we add n and f into the description of the frustum transformation above, we obtain the transformation matrix

$$\begin{bmatrix} -\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

In OpenGL, this matrix is obtained using the call³
`glFrustum(left, right, bottom, top, -near, -far)`

1.4 Texture mapping

In principle, the technique of modeling scenes as collections of polygons that we have been developing for some time is applicable to the rendering of any scene, for any scene can be approximated by colored polygons to the precision of the display device. In practice, however, there are common situations in which piecing together polygons is a grossly inefficient way to obtain the desired result.

Consider, for example, a polygonal model of the interior of an art museum. The macroscopic structure of the museum is likely to be quite simple, at least if its architect was sober while producing the blueprints. A typical room will have a floor, a ceiling, and several walls (some with rectangular doorways); these can be subdivided into a few triangles each, resulting in a simple model that can be rendered with reasonable efficiency. But the paintings themselves, presumably the focus of interest, are another matter entirely. You aren't likely to manage to concoct a tolerably faithful rendition of the surface of the Mona Lisa from less than thousands of triangles, most of which will be very small. Running through the loop control for our scan-conversion algorithm on thousands of very small triangles is not going to be particularly fast; applying a

³OpenGL uses -near and -far due to historical reasons.

perspective projection to the vertices of thousands of very small triangles will be much worse.

The fundamental problem in this scenario is that we are attempting to apply the polygonal models in a setting in which their primary advantages are nullified while their primary disadvantages are exposed. The primary benefit of cutting up a scene into polygons is that it allows perspective projections and occlusion calculations (z -buffering) to be done with a minimum of computational waste. In the case of the art museum, however, the polygonal representation of a painting is actually very wasteful. Geometrically, the painting is just the same rectangle that the canvas was before the artist took a paintbrush to it. The scan-conversion algorithm does not know that the many triangles we are handing it actually fit together to form a square, so it must repeatedly perform perspective projections on the vertices of intermediate triangles to locate points that it could have correctly linearly interpolated. The thousands of superfluous projections expend a lot of time, probably enough to degrade performance visibly in a real-time application.

In such a situation we should like to decouple the analysis of the macroscopic geometry of an object from the rendering of its surface details. As far as the basic scan-conversion algorithm is concerned, two pictures of the same dimensions, however different the images that may be depicted on them, are effectively the same rectangle. Conceptually, at least, we want to scan-convert the rectangle and thereafter somehow glue the surface detail, which should have an independent existence in some form, to the scan-converted rectangle.

1.4.1 Texture coordinate interpolation

The standard recipe for performing this unspecified “gluing” operation, known as *texture mapping*, allows the surface detail of an object to be represented as an ordinary rectangular bitmap of whatever size we please. The user describes how the detail bitmap should be attached to a particular triangle by supplying the coordinates of the points of the bitmap that correspond to the vertices of the triangle. We will denote such “texture” coordinates as $[x_t, y_t]^t$. If we know the texture coordinates corresponding to a point of a triangle, then we can determine the color of that point by looking up the color of the associated pixel in the texture image.

Having specified the bitmap coordinates of the vertices of a triangle, we need to determine the bitmap coordinates, whence also the colors, of any other points on the triangle by some proper interpolation. Unfortunately, straightforward linear interpolation such as we did for z -buffering will produce visibly wrong results. The “visibly” is no exaggeration, as you can see in Figure [nothereyet], which depicts a square (two abutting triangles) onto which a checkerboard pattern has been texture-mapped with simple linear interpolation of texture coordinates. Particularly at the junction between the two triangles, it is painfully obvious that something has been done wrong.

What has been done wrong is that we have assumed not only that a projection maps lines to lines, which is true, but also that it respects ratios of lengths on lines, which is

false. If texture coordinate values linearly interpolated in the image plane are to match the values linearly interpolated on the triangle in space (the correct values), then the projection must map motion at constant velocity along a line in the world to motion at some constant velocity along the image of this line under the projection. In particular, if we mark off a series of evenly spaced points on a line in world space, the images of these points must be evenly spaced in the image plane. But this is not the case. For instance, let us draw in world space a line parallel to the z -axis and passing through a point to the right of the eye, and suppose that marks are made one inch apart all the way to the horizon. The fact that all these marks after a certain distance lie in our field of view implies that they cannot be evenly spaced: if they were, only finitely many of the marks could fit into our field of view.

Correct interpolation can in fact be managed but requires use of an element of the projection operation that we have so far disposed of rather unceremoniously, namely, the fourth coordinate w_i for a point in the projected image. We shall find in a moment that if we retain the value of w_i along with the texture coordinates at the vertices, we have enough information to generate correct texture coordinates at intermediate points of a triangle by an appropriate sort of interpolation process.

Given our projection matrix \mathbf{P} , recall that

$$\begin{bmatrix} x_i & w_i \\ y_i & w_i \\ z_i & w_i \\ w_i & \end{bmatrix} = \mathbf{P} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

As a result, we can also say that

$$\begin{bmatrix} x_e/w_i \\ y_e/w_i \\ z_e/w_i \\ 1/w_i \end{bmatrix} = \mathbf{P}^{-1} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}$$

From this equation, we see that that the value $\frac{1}{w_i}$ (the fourth row of the expression) is an affine function of $[x_i, y_i, z_i]^t$, and so can be computed correctly for each point on a triangle using simple linear interpolation of the values at the vertices during scan conversion.

Now if we want the texture coordinates to have no nonlinear stretching when they are put on the geometry, we want them to be affine functions of the eye coordinates. In other words, there must exist some e, f, g, h such that

$$x_t = \begin{bmatrix} e & f & g & h \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

This can be rewritten as

$$\begin{aligned}
 x_t &= [e \ f \ g \ h] w_i \begin{bmatrix} x_e/w_i \\ y_e/w_i \\ z_e/w_i \\ 1/w_i \end{bmatrix} \\
 &= [e \ f \ g \ h] w_i \mathbf{P}^{-1} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \\
 &= \frac{[e \ f \ g \ h] \mathbf{P}^{-1} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}}{\frac{1}{w_i}}
 \end{aligned}$$

In the last line, we have expressed the computation of the texture coordinate as a fraction. The important thing about this form, is that as we said above, the value of the denominator is an affine function of $[x_i, y_i, z_i]^t$, and we also have a numerator which is an affine function of $[x_i, y_i, z_i]^t$. So the texture coordinates of intermediate points of a triangle can be properly computed by performing two linear interpolations, one for the numerator and one for the denominator. At each pixel point in the image, one needs to perform the division to obtain the actual texture coordinate. This type of interpolation is called “linear-rational” interpolation.

In practice, we are given the texture coordinates at the vertices, this is the value of $\frac{\text{numerator}}{\text{denominator}}$. We can also easily compute denominator $= \frac{1}{w_i}$. As a result, we can easily compute the numerator at each vertex. Given the numerator and denominator at each vertex, we are then in a position to perform linear interpolation on these two values during scan conversion. All we need to do then is perform this interpolation, and also do one division per pixel as we scan convert.

1.4.2 Projective texture mapping

A minor variant on the “gluing” texture mapping is called projective texture mapping. In projective texture mapping, we pretend that the texture is being projected onto the scene using a slide projector. During the rendering process, for each image point, we determine what geometric point we are looking at, and from there determine which point in the texture would get projected through the slide projector to that point. This kind of texture mapping can also be efficiently computed using a slightly different linear rational interpolation.

More specifically, let us model the slide projector as a pin hole camera using a

matrix \mathbf{Q} .

$$\begin{bmatrix} x_t & w_t \\ y_t & w_t \\ - \\ w_t \end{bmatrix} = \mathbf{Q} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

removed for class

Both the numerator and denominator of this expression are affine functions of $[x_i, y_i, z_i]^t$. As such, the projective texture coordinates can be computed efficiently over a triangle in image coordinates using linear rational interpolation. This linear rational interpolation uses a different numerator and denominator than the “glueing” texture mapping.

Projective Texturing in OpenGL In OpenGL there are two ways to perform projective texturing. One way is to pass as with each vertex the texture coordinates `glTexCoords4f($x_t w_t, y_t w_t, 0, w_t$)`.

The other way is to load \mathbf{Q} as the `textureMatrix`, and then pass the texture coordinates as `glTexCoords3f(x_e, y_e, z_e)`.

1.4.3 Variants on texture mapping.

The texture-mapping machinery is quite general and admits a variety of applications in the rendering process. First of all, it is not an essential requirement of the method that the texture should be represented as a two-dimensional bitmap, and there are some reasonable alternatives to the 2D bitmap that are sometimes useful. The same formalism just as well accommodates *solid textures* depending on three coordinates rather than two; such three-dimensional textures are useful for simulating materials such as marble, although the memory requirements of a sampled 3D image of sufficient resolution may be troublesomely demanding. There is also no reason why a 2D or 3D texture must be represented as a sampled image. Interpolated texture coordinates, instead of being used to index into a bitmap, can be passed to an arbitrary color-computing function. This is considerably more powerful—the function may do whatever it likes to calculate the color, which may even be randomized in some way—and, particularly in the 3D case, can eliminate the need for an excessively large auxiliary image. The downside to procedural textures, of course, is the substantial extra expense of calling a function at every pixel to be textured, particularly if the function does anything complicated to find the color. Real-time applications will have to be content with sampled textures.

In another direction, that the “texture” being applied should consist of an object’s surface coloration is also unnecessary. A useful alternative is to store normal vectors in the sampled texture instead of colors; the interpolation then provides a normal to the textured object at each sampled point, which will influence the shading calculations. This *normal mapping* technique can be used to render small-scale geometrical

features of an object without complicating the triangulated model of the object unduly. A variation on this technique known as *bump mapping* assumes that we already have surface normals and textures the surface with small perturbations to the existing normals, giving the illusion of small-scale structure after the shading computations have been performed.

Environment mapping, a device for roughly simulating reflection, stores in the *environment map* a color for each direction (rather, for a finely spaced set of directions) in three-dimensional space, the color of whatever an observer at a certain distinguished point looking in that direction would see. Whenever a reflective object in the scene is rendered, its surface normals can be used to look up in the environment map the color that probably should be reflected in the object at each pixel. The environment map gives the correct reflected color if the reflecting point is the distinguished point; depending on the scene, it may be a fairly accurate approximation over a wider region.

1.5 Projective Spaces

The idea of using four by four matrices to perform 3d transformations is borrowed from the topic of projective geometry. Here we briefly take a look at projective spaces, and see how this sheds light on the camera transformations we have been discussing.

To define the three dimensional projective space P^3 , we start with the four dimensional vector space R^4 . The first thing we will do is throw out the zero vector $\vec{0}$. The next thing we will do is define the equivalence class \check{v} to be all non zero scalar multiples of the 4D vector \vec{v} .

$$\alpha\vec{v} \in \check{v}$$

We call each of these equivalence classes a projective point. Since this space has 3 degrees of freedom (we started with 4 and the equivalence classes removed one), we call this space P^3 .

If we fix a basis, then we can look at coordinates of these projective points. Since all scalar multiples of a coordinate vector express the same projective point, we can define an equivalence class of these coordinates as well

$$\alpha\mathbf{c} \in \check{\mathbf{c}}$$

When we write out projective coordinates, we will pick one of the particular scales of \mathbf{c} and use double brackets to emphasise the invariance to scale.

$$\left[\left[\begin{array}{c} x \\ y \\ z \\ w \end{array} \right] \right]$$

For example, we have

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 9 \\ 12 \end{bmatrix} = \begin{bmatrix} -1/2 \\ -1 \\ -3/2 \\ -2 \end{bmatrix},$$

for the second and third projective points are the results of scaling the first by factors of 3 and $-1/2$ respectively.

Once we choose a basis for R^4 , there is a natural way to assign a point in real, live three-dimensional space to each (well, almost each) of our projective points. Provided that $w \neq 0$ we have that

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix},$$

and we can interpret $[x/w, y/w, z/w, 1]^t$ as the coordinates of point in an affine space A^3 . Conversely, every three-dimensional point $[x, y, z, 1]^t$ corresponds to a unique projective point, namely $[[x, y, z, 1]^t$.

We have shown that every ordinary point may be identified with a unique projective point and that every projective point whose fourth coordinate is nonzero can be identified with a unique ordinary point. Curiously, there are still projective points that do not correspond to any point the affine three-dimensional space—the projective points with fourth coordinate 0 do not correspond to any affine point because rescaling a representation of such a projective point results in another representation with fourth coordinate 0, so the procedure above does not lead to a corresponding affine point. This means that the space P^3 of projective points is a strict superset of A^3 : we have added extra points not present in the affine space. We call these special projective points “points at infinity”.

The reason that we refer to these points as at infinity is that as the w value in $[[x, y, z, w]^t$ gets smaller and smaller, the associate affine point is farther and farther away from the origin. As w approaches 0, we can say that we are approaching infinity in some direction. So each point at infinity is associated with some direction (or the negative of that direction). This set of infinite points infact has the topology of P^2 , so we can also call this the plane at infinity.

If we look at a line of points in P^3 there is a single infinite point on that line. This is the point at infinity associated with the direction of the line. Two parallel lines share the same direction, so they in fact do intersect at a single point, which is at infinity.

Because projective points are represented with coordinates, it is reasonable to try multiplying them by matrices. We have to check that this operation makes sense, though: if P is a matrix and $\mathbf{w} \in \tilde{\mathbf{w}}$ is any coordinate vector for a projective point, then $P\mathbf{w}$ should represent the same projective point independent of the particular \mathbf{w} chosen for the class $\tilde{\mathbf{w}}$. In fact

$$P(\alpha\mathbf{w}) = \alpha(P\mathbf{w}) \in \tilde{P\mathbf{w}}$$

because scalar multiplication commutes with matrix multiplication. So multiplication by P performs the same transformation no matter how \mathbf{w} may be rescaled. We still might not end up with a projective point because it could happen that $P\mathbf{w} = [0, 0, 0, 0]^t$; but if we restrict P to be *nonsingular*, then this can't happen. We call 3d transformation from P^3 to P^3 that is implemented using a four by four matrix a “projective transformation”.

Note that such *projective transformations* remain unchanged if the matrix P is rescaled: if $\alpha \neq 0$ then

$$(\alpha P)\mathbf{w} = \alpha(P\mathbf{w}) = \tilde{P}\mathbf{w}$$

for the same reason as before, that matrix and scalar multiplication commute.

As for projective points, we will double the brackets around the four by four matrix used for a projective transformation.

$$\left[\begin{array}{cccc} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{array} \right]$$

This will remind ourselves that projective transformations are scale-independent. A general projective transformation has fifteen ($4 \times 4 - 1$) degrees of freedom.

The general projective transform certainly has all the power of the affine transforms with which we are already familiar. If we choose a matrix whose last row is some scalar multiple of $[0, 0, 0, 1]$, then the projective transformation will be equivalent to an affine transformation. As an example, we can easily scale the scene about the origin by a factor of s by multiplying by the diagonal matrix:

$$\left[\begin{array}{cccc} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1/s \end{array} \right] = \left[\begin{array}{cccc} s & & & \\ & s & & \\ & & s & \\ & & & 1 \end{array} \right]$$

indeed

$$\left[\begin{array}{cccc} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1/s \end{array} \right] \left[\begin{array}{c} x \\ y \\ z \\ 1 \end{array} \right] = \left[\begin{array}{c} sx \\ sy \\ sz \\ 1/s \end{array} \right] = \left[\begin{array}{c} sx \\ sy \\ sz \\ 1 \end{array} \right].$$

In a general projective transform, we can choose the fourth row of our matrix more or less arbitrarily. In particular, we can use the matrix

$$\left[\begin{array}{cccc} 1 & & & \\ & 1 & & \\ 0 & 0 & a & b \\ & & -1 & \end{array} \right]$$

which implements a transformation that should look familiar. In fact this matrix maps

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}$$

precisely the operation we needed to perform to obtain image coordinates from world coordinates. Thus we have managed to express this nonlinear transform as a matrix multiplication in projective space; the formerly troublesome division is completely tucked away in the definition of projective space.

1.5.1 Properties of Projective Transforms

We can think of a projective transform as a 3D transform from P^3 to P^3 . Assuming the matrix is non-singular, this will be a one to one transform. If and only if the bottom row of the matrix is a scalar multiply of $[0, 0, 0, 1]$, the transform will map all points at infinity to other points at infinity. But if the matrix has a general fourth row, the transformation will map infinite points to finite points and vice versa.

If one has a set of points on some single line in P^3 , then under a projective transformation, these will all map to points on some other single line. This is easy to see by looking at the representative coordinate vectors in R^4 . All projective points along a (1D) line in P^3 have coordinates in some (2D) plane through the origin in R^4 . Under some four by four matrix P , all points on this plane in R^4 will map to some other plane in R^4 through the origin. This new plane of coordinates in R^4 must represent a set of points in P^3 that lie along some other single line.

We used this property above when we said that we could use linear interpolation to compute the z_i coordinates of projected triangle.

1.5.2 Frustum III

Looking at four by four matrices as mappings in P^3 , we can now get a better understanding of the frustum transformation.

If we write our points as projective coordinates, then when we are dealing with finite points ($w \neq 0$) we have

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} -\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & -\frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

We can visualize this transform by seeing how it maps specific points. In particular

the points at the corners of the frustum

$$\begin{bmatrix} l \\ b \\ n \\ 1 \end{bmatrix}, \begin{bmatrix} l \\ b \\ f \\ 1 \end{bmatrix}, \begin{bmatrix} l \\ t \\ n \\ 1 \end{bmatrix}, \begin{bmatrix} l \\ t \\ f \\ 1 \end{bmatrix}, \begin{bmatrix} r \\ b \\ n \\ 1 \end{bmatrix}, \begin{bmatrix} r \\ b \\ f \\ 1 \end{bmatrix}, \begin{bmatrix} r \\ t \\ n \\ 1 \end{bmatrix}, \begin{bmatrix} r \\ t \\ f \\ 1 \end{bmatrix}$$

are mapped to the points at the corners of the canonical cube

$$\begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

More interestingly, the center of the pinhole

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

is mapped to a point at infinity in the z direction

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

This means that the set of lines that intersect at the center of pinhole (the lines of site for the pixels) map to a set of lines that intersect at the point at infinity. This means that all of these lines are parallel with the z direction.

1.5.3 How many points specify a projective transform

A projective transform in P^3 is described with a four by four matrix, which takes 16 numbers. Any scalar multiple of a four by four matrix gives the same projective transform, so there are really only 15 real degrees of freedom.

A point in P^3 has three degrees of freedom. This suggest that if I tell you how 5 points transform, I can figure out what the matrix is.

1.5.4 Texture mapping is a projective transform

We saw above that during texture mapping, the map from image to texture coordinates can be computed using linear rational interpolation. This means that

$$\begin{aligned} x_t &= \frac{ax_i + by_i + c}{dx_i + ey_i + f} \\ x_t &= \frac{gx_i + hy_i + i}{dx_i + ey_i + f} \end{aligned}$$

In particular note that both linear rational functions share the same denominator. This implies that this transform can be expressed (whenever $w \neq 0$) as

$$\begin{bmatrix} x_t \\ y_t \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ g & h & i \\ d & e & f \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

It is in fact a projective transform in P^2

Projective transforms in P^2 have many uses. One interesting use is called reprojection. Suppose I take a picture using with one camera

$$\begin{bmatrix} x_{i1} \\ y_{i1} \\ z_{i1} \\ 1 \end{bmatrix} = V_1 P_1 R_1 T_1 \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

where V is a viewport matrix, P is a projection matrix, R is a rotation matrix and T is a translation matrix. Then suppose I want to take this data and simulate a second camera which shares the same pin hole position. This second camera can be represented by the matrix $V_2 P_2 R_2 T_1$.

The transformation from the first to second image will be

$$\begin{bmatrix} x_{i2} \\ y_{i2} \\ z_{i2} \\ 1 \end{bmatrix} = V_2 P_2 R_2 T_2 T_1^{-1} R_1^{-1} P_1^{-1} V_1^{-1} \begin{bmatrix} x_{i1} \\ y_{i1} \\ z_{i1} \\ 1 \end{bmatrix} = V_2 P_2 R_3 P_1^{-1} V_1^{-1} \begin{bmatrix} x_{i1} \\ y_{i1} \\ z_{i1} \\ 1 \end{bmatrix} = M \begin{bmatrix} x_{i1} \\ y_{i1} \\ z_{i1} \\ 1 \end{bmatrix}$$

It can be shown that the values of x_{i2} and y_{i2} do not depend on z_{i1} . This means that if we are not interested in z_{i2} , we can toss out the third row and column on M to obtain a three by three matrix N . This matrix can be used to map

$$\begin{bmatrix} x_{i2} \\ y_{i2} \\ 1 \end{bmatrix} = N \begin{bmatrix} x_{i1} \\ y_{i1} \\ 1 \end{bmatrix}$$

And we see that this kind of mapping is in fact a 2D projective transform. This in fact should not be such a surprise. If the camera is not translated, then any geometry along a ray must map the same exact way.

1.6 Scan-conversion in review.

We have now completely developed the following three-step procedure for converting a geometrical model of the world into a raster image.

1. We are given a camera position and orientation and a collection of triangles in three-dimensional space. Each vertex of a triangle is equipped with an RGB vector, a pair or triple of texture coordinates, or some other descriptor of its color suitable for linear interpolation during scan-conversion. We apply affine transformations to arrive in a coordinate system in which the camera is situated at the origin looking down the negative z -axis.
2. We apply the camera projection to the vertices of each triangle, obtaining a quadruple of unnormalized coordinates $[x_i w_i, y_i w_i, z_i w_i, w_i]^t$ for each. We normalize to obtain the transformed three-dimensional coordinates $[x_i, y_i, z_i, 1]^t$ of the vertices, retaining the value of $1/w_i$ for later use in texture coordinate interpolation.
3. We apply the viewport transformation. Now we can scan-convert each triangle, handling occlusions with the usual z -buffering technique. The color at each pixel is obtained from linear-rational interpolation using the colors at the vertices of the triangle being drawn as well as the values of $1/w_i$ at the vertices, as detailed previously.