Homogeneous Coordinates and Computer Graphics

Tom Davis
tomrdavis@earthlink.net
http://www.geometer.org/mathcircles
November 20, 2001

The relationship between Cartesian coordinates and Euclidean geometry is well known. The theorems from Euclidean geometry don't mention anything about coordinates, but when you need to apply those theorems to a physical problem, you need to calculate lengths, angles, et cetera, or to do geometric proofs using analytic geometry.

Homogeneous coordinates and projective geometry bear exactly the same relationship. Homogeneous coordinates provide a method for doing calculations and proving theorems in projective geometry, especially when it is used in practical applications.

Although projective geometry is a perfectly good area of "pure mathematics", it is also quite useful in certain real-world applications. The one with which the author is most familiar is in the area of computer graphics. Since it is almost always easier to understand mathematics when there are concrete examples available, we'll use computer graphics in this document as a source for almost all the examples.

The prerequisites for the material contained herein include matrix algebra (how to multiply, add, and invert matrices, and how to multiply vectors by matrices to obtain other vectors), a bit of vector algebra, some trigonometry, and an understanding of Euclidean geometry.

1 Computer Graphics Problems

We'll begin the study of homogeneous coordinates by describing a set of problems from three-dimensional computer graphics that at first seem to have unrelated solutions. We will then show that with certain "tricks", all of them can be solved in the same way. Finally, we will show that this "same way" is in fact just a recasting of the original problems in terms of projective geometry.

1.1 Overview

Much of computer graphics concerns itself with the problem of displaying three-dimensional objects realistically on a two-dimensional screen. We would like to be able to rotate, translate, and scale our objects, to view them from arbitrary points of view, and finally, to be able to view them in perspective. We would like to be able to display our objects in coordinate systems that are convenient for us, and to be able to reuse object descriptions when necessary.

As a canonical problem, let's imagine that we want to draw a scene of a highway with a bunch of cars on it. To simplify the situation, we'll have all the cars look the same, but they are in different locations, moving at different speeds, et cetera.

We have the coordinates to describe a tire, for example, in a convenient form where the axis of the tire is aligned with the x-axis of our coordinate system, and the center of the tire is at $(0, 0, 0)$. We would like to use the same description to draw all the tires on a car simply by translating them to the four locations on the body. Our car body, of course, is also defined in a nice coordinate system centered at $(0, 0, 0)$ and aligned with the x, y, and z-axes.

Once we get the tires "attached" to the body, we'd like to make multiple copies of the car in different orientations on the road. The cars may be pointing in different directions, may be moving uphill and downhill, and the one that was involved in a crash may be lying upside-down.

Perhaps we want to view the entire scene from the point of view of a traffic helicopter that can be anywhere above the highway in three-dimensional space and tilted at any angle.

We'll deal with the viewing in perspective later, but the first three problems to solve are how to translate, rotate, and scale the coordinates used to describe the objects in the scene. Of course we want to be able to perform combinations of those operations[1].

We assume that every object is described in terms of three-dimensional cartesian coordinates like (x, y, z), and we will not worry how the actual drawing takes place. (In other words, whether the coordinates are vertices of triangles, ends of lines, or control points for spline surfaces—it's all the same to us—we just transform the coordinates and assume that the drawing will be dragged around with them.)

Finally, except in the cases of rotation and perspective transformation, it is easier to visualize and experiment with two-dimensional drawings and the extension to three dimensions is obvious and straightforward.

1.2 Translation

Translation is the simplest of the operations. If you have a set of points described in cartesian coordinates, and if you add the same amount to the x-coordinate of every one, all will move by the same amount in the x-direction, effectively moving the drawing by that amount. Adding a positive amount moves to the right; a negative amount to the left.

Similarly, additions of a constant value to the y or z-coordinate cause uniform translations in those directions as well. The translations are independent and can be performed in any order, including all at once. If an object is moved one unit to the right and one unit up, that's the same as moving it one unit up and then one to the right. The net result is a motion of length $\sqrt{2}$ units to the upper-right.

We can define a general translation operator $\mathcal{T}(x, y, z)$ as follows:

$$\mathcal{T}_{t_x,t_y,t_z}(x, y, z) = (x + t_x, y + t_y, z + t_z),$$

where t_x, t_y, and t_z are the translation distances in the directions of the three coordinate axes. They may be positive, negative, or zero. $\mathcal{T}_{t_x,t_y,t_z}$ is a function mapping points of three-dimensional space into itself. This $\mathcal{T}_{t_x,t_y,t_z}$ has an inverse, $\mathcal{T}^{-1}_{t_x,t_y,t_z} = \mathcal{T}_{-t_x,-t_y,-t_z}$ which simply translates in the opposite directions along each coordinate axis. Clearly:

$$\mathcal{T}_{t_x,t_y,t_z}(\mathcal{T}_{-t_x,-t_y,-t_z}(x, y, z)) = \\ (x - t_x + t_x, y - t_y + t_y, z - t_z + t_z) = (x, y, z).$$

1.3 Rotation about an Axis

We'll begin by considering a rotation in the x-y plane about the origin by an angle θ in the counter-clockwise direction. Clearly, all of the z-coordinates will remain the same after the rotation. We will denote this rotation by $\mathcal{R}_{z,\theta}$. The z subscript is because the rotation is, in fact, a rotation about the z-axis.

Figure 1 shows how to obtain the equation for the rotation. We begin with a point $P = (x, y)$ and we wish to find the coordinates of $P' = (x', y')$ which result from rotating P by an angle θ counter-clockwise about the z-axis. The equations are most easily obtained by using polar coordinates, where $r = \sqrt{x^2 + y^2}$ is the distance from the origin to P (and to P') , and ϕ is the angle the line connecting the origin to P makes with the x-axis.

As we can see in the figure, $x = r\cos\phi$ and $y = r\sin\phi$, while $x' = r\cos(\phi + \theta)$ and $y' = r\sin(\phi + \theta)$.

[1] There is one other operation that can easily be performed called "shearing", but it is not particularly useful. The general homogeneous transformations that we'll discover also handle all the shearing operations seamlessly.

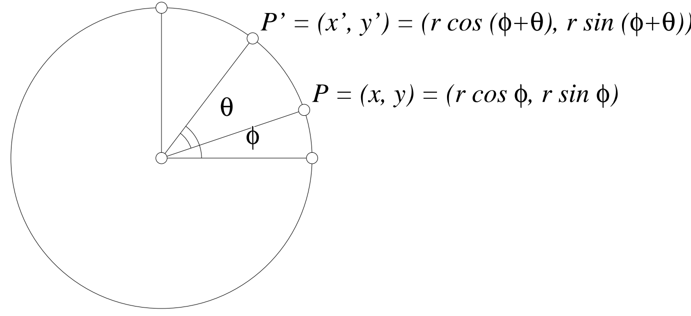

Figure 1: Rotation about the z-axis

Using the addition formulas for sine and cosine, we obtain:

$$\begin{aligned}(x', y') &= (r\cos(\phi+\theta), r\sin(\phi+\theta)) \\ &= (r\cos\phi\cos\theta - r\sin\phi\sin\theta, r\cos\phi\sin\theta + r\sin\phi\cos\theta) \\ &= (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta).\end{aligned}$$

Thus we obtain:

$$\mathcal{R}_{z,\theta}(x, y, z) = (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta, z). \quad (1)$$

As was the case with translation, rotation in the clockwise direction is the inverse of rotation in the counter-clockwise direction and vice versa: $\mathcal{R}_{z,\theta}^{-1} = \mathcal{R}_{z,-\theta}$. It's a good exercise to check this by applying equation 1 and its inverse to a point (x, y).

The equations for rotation about the x and y axes can be obtained similarly, and for reference, here are all three equations together:

$$\begin{aligned}\mathcal{R}_{x,\theta}(x, y, z) &= (x, y\cos\theta - z\sin\theta, y\sin\theta + z\cos\theta) && (2) \\ \mathcal{R}_{y,\theta}(x, y, z) &= (x\cos\theta + z\sin\theta,\ y,\ -x\sin\theta + z\cos\theta) && (3) \\ \mathcal{R}_{z,\theta}(x, y, z) &= (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta, z). && (4)\end{aligned}$$

At first glance, it appears that we have made an error in the signs in equation 3 for the rotation about the y-axis, since they are reversed from those for rotations about the x and z-axes. But all are correct. The apparent problem has to do with the fact that the standard three-dimensional coordinate system is right-handed—if the x and y-axes are drawn as usual on a piece of paper, we must decide whether the positive z-axis is above or below the paper. We have chosen to place positive z values above the paper.

A left-handed system, where the positive z-axis goes down, is perfectly reasonable, but the usual convention is the other way, and the difference is that the signs in some operations are switched around. The normal orientation is called a "right-handed" coordinate system and the other, "left-handed". Even if we had used a left-handed system, the signs would not be the same throughout the equations 2-4; a different set of signs would be flipped.

Think of the orientation of a rotation as follows: to visualize rotation about an axis, put your eye on that axis in the positive direction and look toward the origin. Then a positive rotation corresponds to a counter-clockwise rotation. We've done this with rotation about the z-axis—your eye is above the paper looking down on a standard x-y coordinate system. But visualize the situation looking from the positive y and positive z directions in a right-handed coordinate system. Looking from the x direction, the y goes to the right and the z goes up, but looking from the positive y-axis, the x-axis goes to the *left*, while the z-axis goes up.

1.4 General Rotation

What if you want to rotate about an axis that does not happen to be one of the three principal axes (the x, y, and z axes are called the "principal axes")? What if you want to rotate about a point other than the origin? It turns out that both of these problems can be solved in terms of operations that we already know how to do. Let's begin by looking at rotation about non-principal axes that do pass through the origin.

The strategy is this: we will do one or two rotations about the principal axes to get the axis we want aligned with the x-axis. Then we'll rotate about the x-axis, and finally, we'll undo the rotations we did to align your axis with the x-axis. To do this, assume that the axis of rotation you want points along the vector (x, y, z). We'd like to have it along another vector with its y and z-coordinates zero. If the y-coordinate is non-zero, do a rotation about the x-axis to make the y-coordinate zero. If the z-coordinate is still non-zero, do a rotation about the y-axis to make the z-coordinate zero. Since the rotation is about the y-axis, the y-coordinate (which you previously rotated to be zero) will not be affected. Thus at most two rotations will align an arbitrary axis with the x-axis.

So if the problem is to rotate about the origin by an angle θ, but with an arbitrary axis, what we need to do is perform two rotations to do the alignment. For concreteness, assume those rotations are by ϕ about the x-axis and then by ψ about the y-axis. If $P = (x, y, z)$ is any point in space, the new point P' that results from a rotation of P about this oddball axis is:

$$P' = \mathcal{R}_{x,-\phi}(\mathcal{R}_{y,-\psi}(\mathcal{R}_{x,\theta}(\mathcal{R}_{y,\psi}(\mathcal{R}_{x,\phi}(P))))). \tag{5}$$

To interpret equation 5 remember that the operations are performed from the innermost parentheses outward. First, rotate P about the x-axis by an angle ϕ. Rotate the resulting point about the y-axis by an angle ψ. At this point, the oddball axis is aligned with the x-axis, so the rotation you wanted to do originally can now be done with the $\mathcal{R}_{x,\theta}$ operator. Finally, the two outermost operations return the axis to its original orientation.

Obviously, combining five levels of calculations from equations 2-4 will result in a nightmarish system of equations, but something that is not difficult for a computer to deal with.

Finally, what if the rotation is not about the origin? This time the translation operations from Section 1.2 come to the rescue together with a similar strategy to what we used above for a non-standard axis. We simply need to translate the center of rotation to the origin, perform the rotation, and translate back. If we denote by $\mathcal{R}$ any sort of rotation about any axis through the origin (possibly constructed as a composition of five standard rotations as illustrated in equation 5 above), and we wish to perform that rotation about the point (x, y, z), here is the equation that relates an arbitrary point P to its position P' after rotation:

$$P' = \mathcal{T}_{x,y,z}(\mathcal{R}(\mathcal{T}_{-x,-y,-z}(P))).$$

1.5 Scaling (Dilatation) and Reflection

What if we want to make things larger or smaller? For example, if we have the coordinates that describe an automobile, what are the coordinates that would describe a scale model of an automobile that is 10 times smaller than the original?

It's fairly clear that if we multiply all of our coordinates by $1/10$ we will get a model that's $1/10$ the size, but notice that if the original coordinates had described a car a mile from the origin, the resulting miniature car would be only about $1/10$ mile from the origin. Thus our strategy does scale down the size of the car, but the scaling occurs about the origin. If we wanted to scale about the original car a mile from the origin, we could use the same trick we did with rotations—translate the car to the origin, multiply all the coordinates by $1/10$, and finally translate the resulting coordinates back to the original position.

Non-uniform scaling is also easy to do—if we wish to make an object twice as large in the x-direction, three times as large in the y-direction, and to leave the z size unchanged, we simply multiply all the x-coordinates by 2, all the y-coordinates by 3, and leave the z-coordinates unchanged (or equivalently, multiply them all by 1).

Thus, the most general scaling operation about the origin is given in terms of the scale factors s_x, s_y, and s_z, and the formula for such a function is:

$$\mathcal{S}_{s_x,s_y,s_z}(x, y, z) = (s_x x, s_y y, s_z z). \quad (6)$$

In equation 6, if the scale values are larger than 1, the object's size increases; if they are less than 1, it decreases, and if they are equal to one, the size is unchanged.

Negative scale values correspond to a combination of a size change and a reflection across the plane perpendicular to the axis in question passing through the origin. If $s_x = -1$, there is no size change, but the object is reflected through the plane $x = 0$.

The same ideas used previously can be used to produce scaling functions in directions not aligned with the principal axes. If scaling is to occur about a non-principal axis, rotate that axis to be the x-axis, scale in the x-direction, and then rotate back.

We've considered positive and negative scalings, but what happens if, say, $s_x = 0$? All x-values are collapsed to zero, and it's as if the entire three-dimensional space is projected to the plane $x = 0$. We are going to need this operation (or something similar) when we draw our three-dimensional space on a two-dimensional computer screen.

As long as all three values s_x, s_y, and s_z are non-zero, the scale function has an inverse. It should be obvious that $\mathcal{S}^{-1}_{s_x,s_y,s_z} = \mathcal{S}_{1/s_x,1/s_y,1/s_z}$, and the inverse only makes sense if all three scale values are non-zero. From a physical point of view it's easy to see why. If the scaling operation is really a projection to a plane, there is no way to undo it. Any point on the plane could have come from any of the points on the line through space that projected to that point.

2 Combining Rotation, Translation, and Scaling

As we have seen above, it is often advantageous to combine the various transformations to form a more complex transformation that does exactly what we want. If we simply do the algebra, things can get complicated in a hurry. To illustrate the problem, consider a relatively simple problem—we'd like to rotate clockwise by an angle θ about an axis parallel to the z-axis but passing through the point $(x_1, y_1, 0)$.

The combined transformation of the point (x, y, z) to (x', y', z') is this:

$$\begin{aligned}(x', y', z') &= \mathcal{T}_{x_1,y_1,0}(\mathcal{R}_{x,\theta}(\mathcal{T}_{-x_1,-y_1,0}(x, y, z))) \\ &= ((x - x_1)\cos\theta - (y - y_1)\sin\theta + x_1, \\ &\qquad\qquad (x - x_1)\sin\theta + (y - y_1)\cos\theta + y_1, z),\end{aligned}$$

and this one is pretty straight-forward. Imagine what the combination of 5 rotations would look like.

But there is an easy method, and we'll begin by looking at how to combine rotations. It turns out that all of the rotations about the origin can be easily expressed in terms of matrix multiplication. If we consider our points $P = (x, y, z)$ to be three-dimensional column vectors[2], every rotation corresponds exactly to a multiplication by a certain matrix. Here is the matrix that corresponds to a counter-clockwise rotation by an angle θ about the z axis:

$$\mathcal{R}_{z,\theta}\begin{pmatrix}x\\y\\z\end{pmatrix} = \begin{pmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix} = \begin{pmatrix}x\cos\theta - y\sin\theta\\ x\sin\theta + y\cos\theta\\ z\end{pmatrix}.$$

The other two matrices are equally simple:

[2]For technical reasons, it is better to represent the vectors as column vectors. We could use row vectors, but there are disadvantages that are difficult to explain at this point. However, to save space in the text, we will write the components as usual: (x, y, z) within paragraphs with the understanding that when they are expressed as vectors in equations, they will be turned vertical to make column vectors. When we talk about projective lines later on, we'll need to use actual row vectors, and to indicate this within a paragraph, we'll use the somewhat surprising $(x, y, z)^T$.

$$\mathcal{R}_{y,\theta}\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}\cos\theta & 0 & \sin\theta\\0 & 1 & 0\\-\sin\theta & 0 & \cos\theta\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}x\cos\theta+z\sin\theta\\y\\-x\sin\theta+z\cos\theta\end{pmatrix},$$

and

$$\mathcal{R}_{x,\theta}\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\0 & \cos\theta & -\sin\theta\\0 & \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}x\\y\cos\theta-z\sin\theta\\y\sin\theta+z\cos\theta\end{pmatrix}.$$

The beautiful thing about the matrix representation is that repeated rotations about different axes corresponds to matrix multiplication. Thus if you need to rotate a million vertices that describe the skin of a dinosaur in Jurassic Park VI about some weird axis, you don't need to multiply each point by five different matrices; you simply multiply the five matrices together once and multiply each dinosaur point by that one matrix. It's a savings of almost four million matrix multiplications which can take time, even on a fast computer.

The scaling matrix is even simpler:

$$\mathcal{S}_{s_x,s_y,s_z}\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}s_x & 0 & 0\\0 & s_y & 0\\0 & 0 & s_z\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}s_x x\\s_y y\\s_z z\end{pmatrix}.$$

It can be combined in any combination with the rotation matrices above to make still more complex transformations. As with the rotation matrices alone, the combination of operations simply corresponds to matrix multiplication.

Unfortunately, when we try to do the same thing with the seemingly simpler translation operation, we are dead. It just will not and cannot work this way. It's easy to see why. If you multiply the column vector $(0,0,0)$ by *any* 3×3 matrix, the result will be $(0,0,0)$. The origin is fixed by every matrix multiplication, yet for a translation, we *require* that the origin move.

Fortunately, there is a trick[3] to get the job done. We will simply add an artificial fourth component to each vector and we will always set it to be 1. In other words, the point we used to refer to as (x,y,z), we will now refer to as $(x,y,z,1)$. If you need to find the actual three-dimensional coordinates, simply look at the first three components and ignore the 1 in the fourth position[4].

Of course none of the matrices above will work either, until we add a fourth row and fourth column with all the elements equal to zero except for the bottom corner. For example:

$$\mathcal{R}_{z,\theta}\begin{pmatrix}x\\y\\z\\1\end{pmatrix}=\begin{pmatrix}\cos\theta & -\sin\theta & 0 & 0\\\sin\theta & \cos\theta & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\y\\z\\1\end{pmatrix}=\begin{pmatrix}x\cos\theta-y\sin\theta\\x\sin\theta+y\cos\theta\\z\\1\end{pmatrix}.$$

But now, with this artificial fourth coordinate, it is possible to represent an arbitrary translation as a matrix multiplication:

$$\mathcal{T}_{t_x,t_y,t_z}\begin{pmatrix}x\\y\\z\\1\end{pmatrix}=\begin{pmatrix}1 & 0 & 0 & t_x\\0 & 1 & 0 & t_y\\0 & 0 & 1 & t_z\\0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\y\\z\\1\end{pmatrix}=\begin{pmatrix}x+t_x\\y+t_y\\z+t_z\\1\end{pmatrix}.$$

Using this scheme, every rotation, translation, and scaling operation can be represented by a matrix multiplication, and any combination of the operations above corresponds to the products of the corresponding matrices.

[3] Computer scientists, of course, refer to a "trick" as a "hack".

[4] But if it is *not* 1, beware. We'll handle this case later, when we run into the problem head-on. For now, things are nice.

3 A Simple Perspective Transformation

We know that by setting one of the scale factors to zero, we can collapse all of the z-coordinates, say, to $z = 0$. If we think of our computer screen as having x and y-coordinates in the usual way, we want to do something like this to find the screen coordinates for our points.

But setting the z scale factor to zero simply projects each point in space to the x-y plane in a perpendicular direction. To model the real world, we'd like to imagine looking at real three-dimensional objects, and projecting them, wherever they are, to a rectangular piece of glass (the computer screen) that is a few inches in front of our eye. These rays are not projected perpendicular to the screen; they are all projected at the eye, and they should be drawn on the screen wherever the ray from the object to the eye hits the screen.

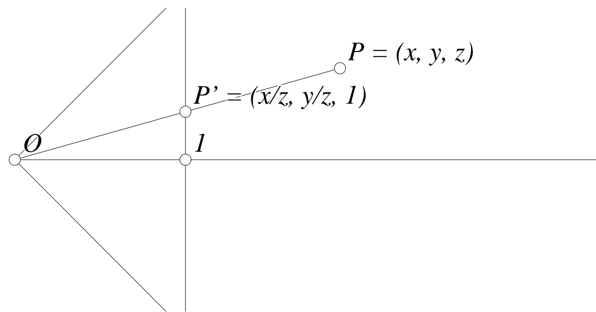

Figure 2: A Simple Perspective Projection

For definiteness, imagine that you are looking from the origin in the direction of the negative z-axis. Remember that if you look from the positive z-axis toward the negative, the x-y plane looks normal to you, with the positive x-axis to the right and the positive y-axis pointing up. Imagine that we would like to project all the points in front of you (they will be the points with negative z-coordinates) onto the plane one unit in front of you (it will have z-coordinate equal to -1). Most computer graphics folks don't like working with these negative coordinates, so they now switch to a left-handed coordinate system so that the point $z = 1$ is in front of them, looking into the plane. We'll do that here, just so we don't need to mess with -1 as the z-coordinate of projection.

Figure 2 shows how we would like to project an arbitrary point $P = (x, y, z)$ to a point P' on the plane. Similar triangles will show you that the coordinates of P' are $(x/z, y/z, 1)$.

This is bad news—we've got to do a division by the z-coordinate, and the only operations we can perform with matrix multiplications are multiplications and additions. How can we combine this perspective transform with the rotations, translations, and scales that work so well with a uniform type of matrix representation?

But with one more "trick" (one more "hack", if you're a computer scientist), we can perform the division as well. It seems a shame that we have to drag around that final fourth coordinate when it's always going to be equal to 1, but without it the matrix multiplications don't make sense. Here is the *big* trick: We will consider two sets of coordinates to represent the same point if one is a non-zero multiple of the other. In other words, the point (x, y, z, w) represents the same point as does $(\alpha x, \alpha y, \alpha z, \alpha w)$.

Normally, of course, $w = 1$, but this shows how we can convert to our standard form if we get a point whose fourth coordinate does not happen to be 1. As long as it is non-zero, we just let $\alpha = 1/w$, and we find that the points (x, y, z, w) and $(x/w, y/w, z/w, 1)$ are equivalent.

This may seem weird at first, but you should not feel at all uncomfortable with it. After all, you do it all the time with fractions. Everybody knows that the fraction $1/3$ is normally written that way, but certain calculations give results like $2/6$, $3/9$, or even $17/51$. All are equivalent to $1/3$.

When we represent our three-dimensional points with four coordinates, it's sort of like having a funny sort of fraction where the three numerators share a common denominator. And exactly as is the case with

fractions, as long as the "denominator" (the fourth term) is not equal to zero, we'll have no trouble.

Here is the matrix that performs the perspective calculation shown in Figure 2 that takes the three-dimensional point (x, y, z) to $(x/z, y/z, 1)$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z \end{pmatrix}.$$

The resulting vector after the multiplication is (x, y, z, z), but remember that we can multiply all the components by the same non-zero number and get an equivalent point, and since we're looking at values of z in the half-space with negative z-coordinates, we can multiply all of the coordinates by $1/z$ to obtain the equivalent representation: $(x/z, y/z, 1, 1)$.

There is one problem with this matrix: it is singular, meaning that it does not have an inverse. The reason is that our calculation effectively projected every point on the line connecting P and P' to the same point on the plane $z = 1$. Any function or transformation that maps two points to the same point cannot be undone, or inverted.

At first, this seems like a purely aesthetic problem. After all, aren't we planning to map all the points along that line to the same point on the computer screen? The problem is that in computer graphics, you often want to find where to draw them on the screen but to avoid drawing them until you've found all the points to be drawn there, and then to draw only the point that's nearest the eye of the viewer. That's because in the real world, objects nearer your eye block your vision of objects behind them.

We can solve the problem easily, and at the same time, create a non-singular (invertable) matrix. We don't really care that the z coordinate is 1 after transformation—all we care about are the x and y coordinates that tell us where to paint the point on the screen. Here's a better perspective matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z+1 \\ z \end{pmatrix}. \tag{7}$$

This transforms our point $(x, y, z, 1)$ to $(x/z, y/z, 1+1/z, 1)$ (at least after we multiply all the coordinates by $1/z$). The final x and y coordinates are the same as before—everything on that line from P to P' goes to points with the same x and y coordinates—but the z values are all different. The order of the points is inverted, but at least it's easy to see from the transformed coordinates which ones were closer to the eye[5].

The matrix in 7 is non-singular; its inverse is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix}$$

Note that since we are dealing with transformations of three-dimensional space, all the transformation matrices are 4×4. If we were working only with points on the plane (two-dimensional space), the transformation matrices would have been 3×3. For a line, they would have been 2×2, et cetera.

[5] In standard computer graphics packages, a more sophisticated version of the perspective matrix is generally used to control the various aspect ratios and to control the range of z values that emerge after the calculation. If you're interested, look at a book on computer graphics for the exact forms, but the basic idea is identical to that illustrated in transformation 7.

4 Homogeneous Coordinates

Homogeneous coordinates provide a method to perform certain standard operations on points in Euclidean space by means of matrix multiplications. As we shall see, they provide a great deal more, but let's first review what we know up to this point.

The usual cartesian coordinates for a point consist of a list of n points, where n is the dimension of the space. The homogeneous coordinates corresponding to the same point require $n + 1$ coordinates.

Normally, we add a coordinate to the end of the list, and make it equal to 1. Thus the two-dimensional point (x, y) becomes $(x, y, 1)$ in homogeneous coordinates, and the three-dimensional point (x, y, z) becomes $(x, y, z, 1)$. To learn more, it is often useful to look at the one-dimensional space (points on a line), and it is also useful to remember that the same method can be applied to higher-dimensional Euclidean spaces. Homogeneous coordinates in a seven-dimensional Euclidean space have eight coordinates.

The final coordinate need not be 1. Since the most common use of homogeneous coordinates is for one, two, and three-dimensional Euclidean spaces, the final coordinate is often called "w" since that will not interfere with the usual x, y, and z-coordinates. In fact, two points are equivalent if one is a non-zero constant multiple of the other. Points corresponding to standard Euclidean points all have non-zero values in the final (w) coordinate.

5 Projective Geometry

What's really going on is, in a sense, far simpler. Homogeneous coordinates are *not* Euclidean coordinates; they are the natural coordinates of a different type of geometry called projective geometry.

Here is the real definition of homogeneous coordinates in projective geometry, where we will consider the two-dimensional version (with three coordinates) for concreteness.

Every vector of three real numbers, (x, y, w), where at least one of the numbers is non-zero, corresponds to a point in two-dimensional projective geometry. The coordinates for a point are not unique; if α is any non-zero real number, then the coordinates (x, y, w) and $(\alpha x, \alpha y, \alpha w)$ correspond to exactly the same point.

If the w-coordinate is non-zero, it will correspond to a Euclidean point, but if $w = 0$ (and at least one of x or y is non-zero), it will correspond to a "point at infinity" (see Section 7).

Furthermore, the allowable transformations in (two-dimensional) projective geometry correspond to multiplication by arbitrary non-singular 3×3 matrices. Obviously, if two matrices are related by the fact that one is a constant non-zero multiple of the other, they represent the same transformation.

The coordinates are called "homogeneous" since they look the same all over the space, and with the complete flexibility of multiplication by an arbitrary non-singular matrix we can convert any line to be the line at infinity, or convert points at infinity to points in normal space, et cetera. In fact, if you ever took a perspective drawing class, the "vanishing points" on the horizon are really places, where, under a perspective transformation, points at infinity wind up in normal space.

There is more to projective geometry, of course. There are equations for lines, for conic sections, methods to find intersections of lines or for finding the lines that pass thorough a pair of points, et cetera, but we will get to those later. Let us begin by describing a nice mental model for two-dimensional projective geometry.

6 Euclidean and Projective Geometry

Projective geometry is not the same as Euclidean geometry, but it is closely related. The two have many things in common. Just as we can discuss Euclidean geometry in any finite number of dimensions, we can do the same for projective geometry. Of course real-world applications are typically two and three-

dimensional in both geometries, but we'll sometimes find it useful to think about the one-dimensional version of both.

A nice way to think about various types of geometry is in terms of the allowable operations and the properties that are preserved under those operations. For example, in Euclidean geometry, we can move figures around on the plane, rotate them, or flip them over, and if we do these things, the resulting transformed figures remain congruent to the originals. Two figures are congruent if all the measurements are the same—lengths of sides, angles, et cetera.

In projective geometry, we are going to allow projection as the fundamental operation. It's easy to see what projection means in one and two dimensions, so we'll begin with those.

Suppose you have a figure drawn on a plane. You can project it to another plane as follows: pick some point of projection that is on neither of the planes. Draw straight lines through every point of the original figure that pass through the point of projection. The image of each point is the intersection of that line with the other plane.

Note also that we can obtain projections perpendicular to the plane of projection simply by projecting from a "point at infinity"—see Section 7.

Notice that we have said nothing about the orientations of the two planes—they need not be parallel, for example. In your mind's eye, try to imagine some of these two-dimensional projections.

7 Visualizing Projective Geometry

Here are two postulates from two-dimensional Euclidean geometry:

- Every two points lie on a line.
- Every two lines lie on a point, unless the lines are parallel, in which case, they don't.

In two-dimensional projective geometry, these postulates are replaced by:

- Every two points lie on a line.
- Every two lines lie on a point.

That's basically the whole difference. How can we visualize a model for such a thing? The model must describe all the points, all the lines, what points are on what lines, and so on.

The easiest way is to take the points and lines from a standard two-dimensional Euclidean plane and add stuff until the projective postulates are satisfied. The first problem is that the parallel lines don't meet. Lines that are almost parallel meet way out in the direction of the lines, so for parallel lines, add a single point for each possible direction and add it to all the parallel lines going that way. You can think of these points as being points at infinity—at the "ends" of the lines. Note that each line includes a single point at infinity—the north-south line doesn't have both a north and south point at infinity. If you "go to infinity" to the north and keep going, you will find yourself looping around from the south. Lines in projective geometry form loops.

Now take all the new points at infinity and add a single line at infinity going through all of them. It, too, forms a loop that can be imagined to wrap around the whole original Euclidean plane. These points and lines make up the projective plane.

You might make a mental picture as follows. For some small configuration of points and lines that you are considering, imagine a *really* large circle centered around them, so large that the part of the figure of interest is like a dot in its center. Now any parallel lines that go through that "dot" will hit the large circle very close together, at a point that depends only on their direction. Just imagine that all parallel lines hit the circle at that point. This large circular line surrounding everything is the "line at infinity".

Check the postulates. Two points in the Euclidean plane still determine a single projective line. One point in the plane and a point at infinity determine the projective line through the point and going in the given direction. Finally, the line at infinity passes through any two points at infinity.

How about lines? Two non-parallel lines in the Euclidean plane still meet in a point (the standard Euclidean point), and don't meet anywhere else. Parallel lines have the same direction, so meet at the point at infinity in that direction. Every line on the original plane meets the line at infinity at the point at infinity corresponding to the line's direction.

Note: The projective postulates do not distinguish between points and lines in the sense that if you saw them written in a foreign language:

- Every two glorphs lie on a smynx,
- Every two smynxes lie on a glorph,

there is no way to figure out whether a smynx is a line and a glorph is a point or vice-versa. If you take any theorem in two-dimensional projective geometry and replace "point" with "line" and "line" with "point", it makes a new theorem that is also true. This is called "duality"—see any text on projective geometry.

8 Back to the Homogeneous Coordinates

So we've got a nice mental picture—how do we assign coordinates and calculate with them? The answer is that every triple of real numbers (x, y, w) except $(0, 0, 0)$ corresponds to a projective point. If w is non-zero, (x, y, w) corresponds to the Euclidean point $(x/w, y/w)$ in the original Euclidean plane; $(x, y, 0)$ corresponds to the point at infinity corresponding to the direction of the line passing through $(0, 0)$ and (x, y). Generally, if α is any non-zero number, the homogeneous coordinates (x, y, w) and $(\alpha x, \alpha y, \alpha w)$ represent the same point.

Since projective points and lines are in some sense indistinguishable, it had better be possible to give line coordinates as sets of three numbers (with at least one non-zero). If the points are column vectors, the lines will be row vectors[6] (written with a "T" exponent that represents "transpose"), so $(a, b, c)^T$ represents a line. The point $P = (x, y, w)$ lies on the line $L = (a, b, c)^T$ if $ax + by + cw = 0$. In the Euclidean plane, the point (x, y) can be written in projective coordinates as $(x, y, 1)$, so the condition becomes $ax + by + c = 0$—high-school algebra's equation for a line. The line passing through all the points at infinity has coordinates $(0, 0, 1)^T$. As with points, for any non-zero α, the line coordinates $(a, b, c)^T$ and $(\alpha a, \alpha b, \alpha c)^T$ represent the same line. Also, as with points, at least one of a, b, or c must be non-zero to have a set of valid line coordinates.

In matrix notation, the point P lies on the line L if and only if $LP = 0$. This is like the dot product of the vectors. Since L is a row vector and P is a column vector of the same length, the product is essentially a 1×1 matrix, or basically, a scalar. If $P = (x, y, w)$ and $L = (a, b, c)^T$, then $LP = ax + by + cw$. If we had chosen to represent lines as column vectors and points as row vectors, that would work fine, too. It has to work because points and lines are dual concepts.

9 Projective Transformations

Projective transformations transform (projective) points to points and (projective) lines to lines such that incidence is preserved. In other words, if $\mathcal{T}$ is a projective transformation and points P and Q lie on line L then $\mathcal{T}(P)$ and $\mathcal{T}(Q)$ lie on $\mathcal{T}(L)$. (Warning: $\mathcal{T}(L)$—the transformation of a line—does not simply use the same matrix as for transforming points. See later in this section.) Similarly, if lines L and M meet at point P, then the lines $\mathcal{T}(L)$ and $\mathcal{T}(M)$ meet at the point $\mathcal{T}(P)$.

[6] Remember that we are writing column vectors in the text as rows, so we're going to have to have a special notation to indicate that a vector in the text is *really* a row vector. That's what the transpose will be used for.

The reason projective transformations are so interesting is that if we use the model of the projective plane described in Section 7 where we've simply added some stuff to the Euclidean plane, the projective transformations restricted to the Euclidean plane include all rotations, translations, non-zero scales, and shearing operations. This would be powerful enough, but if we don't restrict the transformations to the Euclidean plane, the projective transformations also include the standard projections, including the very important perspective projection.

Rotation, translation, scaling, shearing (and all combinations of them) map the line at infinity to itself, although the points on that line may be mapped to other points at infinity. For example, a rotation of 5 degrees maps each point at infinity corresponding to a direction to the point corresponding to the direction rotated 5 degrees. Pure translations preserve the directions, so a translation maps each point at infinity to itself.

The standard perspective transformation (with a $90°$ field of view, the eye at the origin, and looking down the y-axis) maps the origin to the point at infinity in the y-direction. The viewing trapezoid maps to a square.

Every non-singular 3×3 matrix (non-singular means that the matrix has an inverse) represents a projective transformation, and every projective transformation is represented by a non-singular 3×3 matrix. If $\mathcal{M}$ is such a transformation matrix and P is a projective point, then $\mathcal{M}P$ is the transformed point. If L is a line, $L\mathcal{M}^{-1}$ represents the transformed line. It's easy to see why this works: if P lies on L, $LP = 0$, so $L\mathcal{M}^{-1}\mathcal{M}P = 0$, so $(L\mathcal{M}^{-1})(\mathcal{M}P) = 0$. The matrix representation is not unique—as with points and lines, any constant multiple of a matrix represents the same projective transformation.

Combinations of transformations are represented by products of matrices; a rotation represented by matrix $\mathcal{R}$ followed by a translation (matrix $\mathcal{T}$) is represented by the matrix $\mathcal{TR}$.

A (two-dimensional) projective transformation is completely determined if you know the images of 4 independent points (or of 4 independent lines). This is easy to see. A 3×3 matrix has nine numbers in it, but since any constant multiple represents the same transformation, there are basically 8 degrees of freedom. Each point transformation that you lock down eliminates 2 degrees of freedom, so the images of 4 points completely determine the transformation.

Let's look at a simple example of how this can be used by deriving from scratch the rotation matrix for a $45°$ counter-clockwise rotation about the origin. The origin maps to itself, the points at infinity along the x and y axes map to points at infinity rotated $45°$, and the point $(1, 1)$ maps to $(0, \sqrt{2})$.

If $\mathcal{R}$ is the unknown matrix:

$$\begin{aligned}
\mathcal{R}\begin{pmatrix}0\\0\\1\end{pmatrix} &= k_1\begin{pmatrix}0\\0\\1\end{pmatrix} & \mathcal{R}\begin{pmatrix}1\\0\\0\end{pmatrix} &= k_2\begin{pmatrix}\sqrt{2}\\\sqrt{2}\\0\end{pmatrix}\\
\mathcal{R}\begin{pmatrix}0\\1\\0\end{pmatrix} &= k_3\begin{pmatrix}-\sqrt{2}\\\sqrt{2}\\0\end{pmatrix} & \mathcal{R}\begin{pmatrix}1\\1\\1\end{pmatrix} &= k_4\begin{pmatrix}0\\\sqrt{2}\\1\end{pmatrix}
\end{aligned}$$

The $k_1, \ldots, k_4$ can be any constants since any multiple of a projective point's coordinates represents the same projective point. The matrix $\mathcal{R}$ has basically 8 unknowns, so those 8 plus the 4 k_i's make 12. Each matrix equation represents 3 equations, so we have a system of 12 equations and 12 unknowns that can be solved. The computations may be ugly, but it's a straight-forward brute-force solution that gives the rotation matrix $\mathcal{R}$ as any multiple of:

$$\mathcal{R} = \begin{pmatrix}\sqrt{2}/2 & -\sqrt{2}/2 & 0\\ \sqrt{2}/2 & \sqrt{2}/2 & 0\\ 0 & 0 & 1\end{pmatrix}.$$

There's nothing special about rotation. Every projective transformation matrix can be determined in the same brute-force manner starting from the images of 4 independent points.

10 Three-Dimensional Projective Space

Three-dimensional projective space has a similar model. Take three-dimensional Euclidean space, add points at infinity in every three-dimensional direction, and add a plane at infinity going through the points. In this case there will also be an infinite number of lines at infinity as well. In three dimensions, points and planes are dual objects.

Projective transformations in three dimensions are exactly analogous. Points are represented by 4-tuple column vectors: (x, y, z, w), and planes by row vectors: $(a, b, c, d)^T$. Any multiple of a point's coordinates represents the same projective point. A point P lies on a plane M if $MP = 0$. All three dimensional projective transformations are represented by 4×4 non-singular matrices.

In three dimensions, the images of 5 independent points (or planes) completely determine a projective transformation. (A 4×4 matrix has 16 numbers, but 15 degrees of freedom because any multiple represents the same transformation. Each point transformation that you nail down eliminates 3 degrees of freedom, so the images of 5 independent points completely determine the transformation.)[7]

The brute-force solution has 20 equations and 20 unknowns (there will be 5 k_i's in addition to the 15 unknowns), and although the solution is time-consuming, it is straight-forward.

The calculation can be simplified. Suppose you want a transformation that takes P_1 to Q_1, ..., and P_5 to Q_5. Let

$$\begin{aligned} I_1 &= (1, 0, 0, 0) \\ I_2 &= (0, 1, 0, 0) \\ I_3 &= (0, 0, 1, 0) \\ I_4 &= (0, 0, 0, 1) \\ I_5 &= (1, 1, 1, 1) \end{aligned}$$

Find the transformation $\mathcal{P}$ that takes P_i to I_i and the transformation $\mathcal{Q}$ that takes Q_i to I_i. Because of all the zeroes, these are much easier to work out. The transformation you want is $\mathcal{Q}^{-1}\mathcal{P}$.

10.1 Construction of an Arbitrary Transformation

Based upon the idea above, here is a purely mechanical method to construct a transformation from any four independent points to any other four points in two-dimensional projective geometry. The method can obviously be extended to any number of dimensions, where the images of $n + 2$ points are required to determine the transformation.

Suppose we seek a matrix $\mathcal{M}$ that performs the following map:

$$\begin{aligned} \mathcal{M} : (x_1, y_1, w_1) &\rightarrow (X_1, Y_1, W_1) \\ \mathcal{M} : (x_2, y_2, w_2) &\rightarrow (X_2, Y_2, W_2) \\ \mathcal{M} : (x_3, y_3, w_3) &\rightarrow (X_3, Y_3, W_3) \\ \mathcal{M} : (x_4, y_4, w_4) &\rightarrow (X_4, Y_4, W_4) \end{aligned}$$

We will construct the matrix $\mathcal{M}$ as the product $\mathcal{Q}\mathcal{P}^{-1}$ where:

[7]The concept generalizes to n-dimensional space. Transformations are denoted by $(n+1) \times (n+1)$ matrices having $(n+1)^2$ entries, but an arbitrary constant multiple reduces this to $(n+1)^2 - 1 = n^2 + 2n = n(n+2)$ degrees of freedom. Each time you nail down the image of an n-dimensional point, you remove n degrees of freedom, so the images of $n + 2$ independent points are required to completely determine a projective transformation in projective n-space.

$$\begin{array}{lll} \mathcal{P}:(1,0,0) & \rightarrow & (x_1,y_1,w_1) \\ \mathcal{P}:(0,1,0) & \rightarrow & (x_2,y_2,w_2) \\ \mathcal{P}:(0,0,1) & \rightarrow & (x_3,y_3,w_3) \\ \mathcal{P}:(1,1,1) & \rightarrow & (x_4,y_4,w_4) \end{array} \quad \text{and} \quad \begin{array}{lll} \mathcal{Q}:(1,0,0) & \rightarrow & (X_1,Y_1,W_1) \\ \mathcal{Q}:(0,1,0) & \rightarrow & (X_2,Y_2,W_2) \\ \mathcal{Q}:(0,0,1) & \rightarrow & (X_3,Y_3,W_3) \\ \mathcal{Q}:(1,1,1) & \rightarrow & (X_4,Y_4,W_4) \end{array}$$

The construction of $\mathcal{P}$ and $\mathcal{Q}$ is obviously identical, so we will show the construction of $\mathcal{P}$ only.

We will denote the unknown entries in the matrix $\mathcal{P}$ by p_{ij}. As usual, we will also use k_i as the arbitrary constants in that multiply the homogeneous coordinates of our result. The matrix $\mathcal{P}$ must satisfy:

$$\begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} k_1x_1 & k_2x_2 & k_3x_3 & k_4x_4 \\ k_1y_1 & k_2y_2 & k_3y_3 & k_4y_4 \\ k_1w_1 & k_2w_2 & k_3w_3 & k_4w_4 \end{pmatrix}. \tag{8}$$

There are thirteen variables including the nine p_{ij} and the four k_i, of which we can fix any one. We choose to let $k_4 = 1$. We can actually perform the matrix multiplication on the left of equation 8 and set $k_4 = 1$ to obtain:

$$\begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{11}+p_{12}+p_{13} \\ p_{21} & p_{22} & p_{23} & p_{21}+p_{22}+p_{23} \\ p_{31} & p_{32} & p_{33} & p_{31}+p_{32}+p_{33} \end{pmatrix} = \begin{pmatrix} k_1x_1 & k_2x_2 & k_3x_3 & x_4 \\ k_1y_1 & k_2y_2 & k_3y_3 & y_4 \\ k_1w_1 & k_2w_2 & k_3w_3 & w_4 \end{pmatrix}. \tag{9}$$

From equation 9, we can immediately conclude that $p_{1j} = k_jx_j$, $p_{2j} = k_jy_j$, and that $p_{3j} = k_jw_j$. We don't yet know the values of k_j except that $k_4 = 1$, but we can now rewrite equation 9 as:

$$\begin{pmatrix} k_1x_1 & k_2x_2 & k_3x_3 & k_1x_1+k_2x_2+k_3x_3 \\ k_1y_1 & k_2y_2 & k_3y_3 & k_1y_1+k_2y_2+k_3y_3 \\ k_1w_1 & k_2w_2 & k_3w_3 & k_1w_1+k_2w_2+k_3w_3 \end{pmatrix} = \begin{pmatrix} k_1x_1 & k_2x_2 & k_3x_3 & x_4 \\ k_1y_1 & k_2y_2 & k_3y_3 & y_4 \\ k_1w_1 & k_2w_2 & k_3w_3 & w_4 \end{pmatrix}. \tag{10}$$

The first three columns of equation 10 don't help at all, but we can re-write the fourth column as follows:

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ w_1 & w_2 & w_3 \end{pmatrix} \begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix} = \begin{pmatrix} x_4 \\ y_4 \\ z_4 \end{pmatrix}. \tag{11}$$

We can solve equation 11 for the k_j:

$$\begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ w_1 & w_2 & w_3 \end{pmatrix}^{-1} \begin{pmatrix} x_4 \\ y_4 \\ z_4 \end{pmatrix}. \tag{12}$$

Using the values of the k_j obtained from equation 12 and substituting those into the first three columns of equation 9 we can find the unknown matrix $\mathcal{P}$:

$$\mathcal{P} = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix} = \begin{pmatrix} k_1x_1 & k_2x_2 & k_3x_3 \\ k_1y_1 & k_2y_2 & k_3y_3 \\ k_1w_1 & k_2w_2 & k_3w_3 \end{pmatrix}.$$