

Inspecting the 3D Pipeline, Part I

By Casey Muratori (cmu@d6.com)

Rasterization has been the hot topic in game development for the past few years. Improved image quality has consistently come from the increased quality used to render individual polygons. Accordingly, most game development literature on the subject of 3D graphics skips quickly from the 3D geometry aspects of rendering to the nitty-gritty details of rasterization.

In our scramble to have the fastest, most robust rasterization in our games, have we neglected the other parts of our 3D engines? Is the part before the rasterization, the 3D pipeline, being left to decay? In game-development literature, I believe it is. Even though information about rasterization is becoming increasingly plentiful, it is still difficult to find information on the 3D pipeline from a game-developer's perspective.

A solid 3D pipeline is crucial. The care you take in developing and designing your 3D pipeline will have a profound affect on the presentation of your game. Your understanding of the 3D pipeline will dictate the control you have over your 3D objects. We all know the basics, but it's time we became familiar with the subtleties and nuances of the 3D pipeline. So, in this article, we're going to run through the 3D pipeline and give it what it's been needing: a thorough inspection. In each section, we'll try to tighten some loose bolts, polish up rusty joints, and hopefully, when we're finished, we'll have a better understanding of how everything fits together.

Dealing with Aspect Ratios

Let's start at the mouth of the 3D pipeline, and work our way up. All our polygons spill out of the pipe at a very specific place: the perspective projection. This projection is responsible for producing the perspective foreshortening we expect in a 3D game. As it is presented in most 3D texts, the projection of a point p takes the form

$$p'_x = d \frac{p_x}{p_z} + \frac{w_r}{2}$$

$$p'_y = d \frac{p_y}{p_z} + \frac{h_r}{2}$$

where w_r and h_r are the width and height of the screen (in pixels) at the current resolution, and d is the distance from the viewer to the viewplane.

Given the simplicity of the projection in this form, it is all too tempting to take the projected points that it produces and let them spill out into the rasterizer. But, if we were to do so, we'd be forgetting one underlying assumption this projection makes: *the pixels on the screen are square*. Beside the adjustment for centering, both p'_x and p'_y are computed in the same way - their equations are identical. If we were to project an actual square, it would come out of the projection as covering an equal number of pixels in both x and y ... but if pixels are wider than they are high, or higher than they are wide, the sides of the square will not be equal in length when drawn. The shape would leave the pipeline as a square, but end up a rectangle! I don't know about you, but it sure sounds like there's a screw loose to me.

The question is, then, can we safely make the assumption that pixels are always square? Sadly, we cannot. If you think about it, the size of a pixel on a given display is determined by two factors: the dimensions of the display, and the resolution used on that display. The relationship between the two is clear: the width of a single pixel is the width of the display divided by the number of pixels trying to fit in that width (the horizontal resolution). The height similarly follows suit.

Standard monitor and TV set displays are about four-thirds as wide as they are high. This proportion of horizontal to vertical size is called the *aspect ratio* of the display. It is usually expressed as a single number, which is the quotient of the width to the height - so, for the standard TV or monitor, this would be $4/3 = 1.33$. Now, the screen resolution also has a width and height, and thus an aspect ratio: at 640x480, the aspect ratio is also 1.33, and at 320x200, the aspect ratio is 1.6. When we

compare the aspect ratio of the physical display with the aspect ratio of the screen resolution, we can tell if the pixels are square. For example, 640x480 has the same aspect ratio as the monitor, 1.33, so the pixels will be square. 320x200 does not have a 4/3 aspect ratio, and so the pixels will not be square.

Fortunately, we can easily correct this problem by compensating for higher or wider pixels when we project. If we consider the physical dimensions of the screen to be w_p by h_p , and the screen resolution to be w_r pixels by h_r pixels as before, then all we need to do is solve the following proportion:

$$\frac{w_r}{w_p} = a \frac{h_r}{h_p}$$

This says that the width of a pixel is equal to the height of a pixel multiplied by some scalar, a . This scalar tells us how much the height will be scaled to equal the width. Solving for a , we get:

$$a = \frac{h_p}{w_p} \frac{w_r}{h_r}$$

or

$$a = \frac{w_r}{r_p h_r}$$

Where r_p is the proportion w_p/h_p , which, as I mentioned previously, is the common way physical aspect ratios are given.

Now we have a . It tells us how much stretching is going to occur - but we want to *correct* for this stretching. So, all we need to do is use its inverse to scale the y values we generate. This will shrink or expand all our heights in the exact opposite way, and thus compensate for non-square pixels. We can express it directly as part of the projection equation:

$$p'_y = d \frac{1}{a} \frac{p_y}{p_z} + \frac{h}{2} = d \frac{r_p h_r}{w_r} \frac{p_y}{p_z} + \frac{h}{2}$$

That's all there is to it. Now, your projection will work if you want your game to run at 640x480 on a film projector ($r_p = 1.85$), or 320x200 on a regular monitor or TV ($r_p = 1.33$).

Using Field of View

We've tightened up one loose screw in our projection, but the rest of the equation's still a bit rusty. When I look at our projection equation, the multiplier d raises a lot of questions. From the equation alone, it is not clear what value d should have, or how that value needs to change as the other parameters of the equation change. To rectify this, we need to understand what our requirements for d are, and then derive a good formula for computing d .

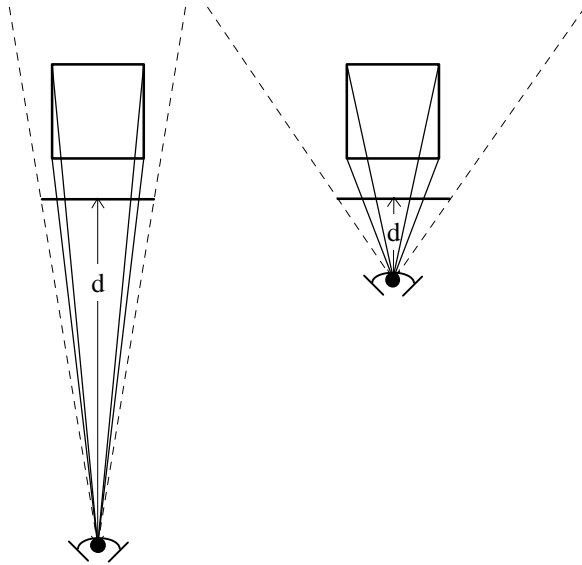


Figure 1 - Varying Values of d

First, what does d do in our equation? Figure 1 shows two perspective projections of a cube as viewed from above. The essential difference between the smaller value of d and the larger value, as you can see from the diagram, is in the lines of projection for the cube. For the smaller value, the lines diverge quickly, which means that the change in projected size between far side of the cube and near side is great. For the larger value, the lines diverge slowly, yielding little difference between far and near. Since d controls the exaggeration of the perspective in this way, it makes sense that our value of d should be specified by an intuitive measure of the amount of perspective foreshortening we wish to have.

But, from the same figure, it is also apparent that the view is much narrower when the value of d is greater... in this way,

d also affects the breadth of the view. This means that smaller values of d will make objects appear smaller, since they're not expanding but the breadth of the view is. Looking back at our projection equation, this makes sense: d scales both x and y . The smaller d is, the less scaling x and y will undergo, and thus, the smaller everything will appear.

Now look at figure 2. This shows the affect of keeping the same d , but varying the width in pixels, w_r , of the screen. Clearly, the larger the width of the screen, the larger the view becomes. So, d cannot be considered in isolation - while it affects the breadth of the view, it does not uniquely determine it. As the screen resolution changes, d must also change to compensate. We should therefore require that our value for d provide consistent results across resolutions.

Given requirements we have stated, we can derive a formula for reliably computing d . These requirements are actually not very difficult to meet if we pick a measure of perspective foreshortening and the breadth of the view that is not dependent on d or w_r . A convenient measure that many people use is the *field of view*. If you think of the lines formed by the viewer and the edges of the screen (which are shown as dotted lines in figures 1 and 2), the field of view is the measure of the angle between these lines. It is a constant measure of the breadth of the view. As the angle gets larger, the view becomes wider, and there is more perspective foreshortening. As the angles becomes smaller, the view is more narrow, and there is less foreshortening.

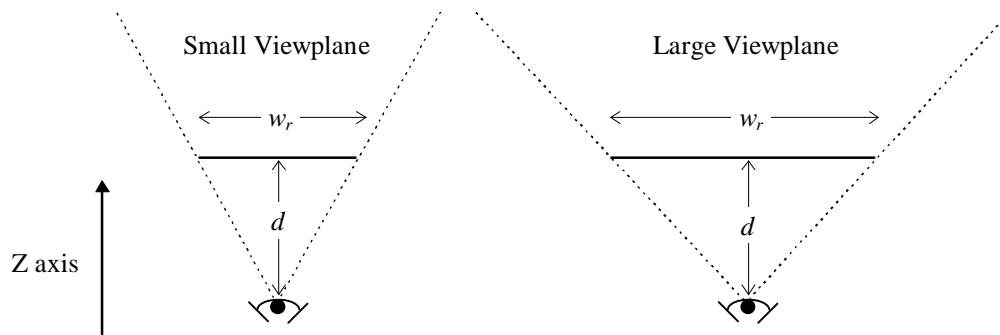


Figure 2 - Different Size Viewplanes and d

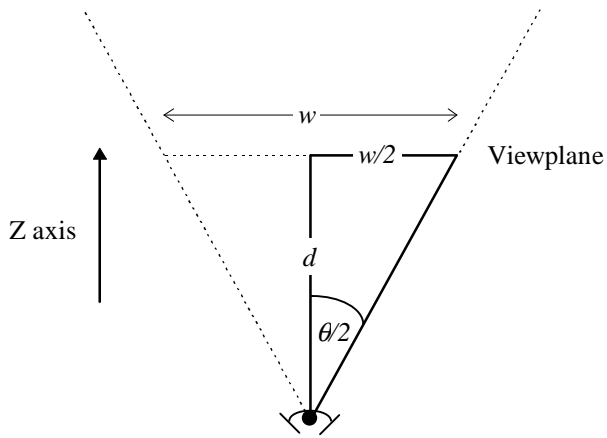


Figure 3 - Field of View Specification for d

$$d = \frac{\frac{w_r}{2}}{\tan(\frac{\theta}{2})}$$

This gives us exactly what we need: an equation for d that is based on an intuitive measure of perspective foreshortening, and which produces consistent results across screen resolutions.

With a more solid understanding of the value of d , and having already found the necessary multiplier to correct for aspect ratio, we can re-write our original perspective projection to incorporate these values. The final formulation is given by:

$$p'_x = \left(\frac{w_r}{2 \tan(\frac{\theta}{2})} \right) \frac{p_x}{p_z} + \frac{w_r}{2}$$

$$p'_y = \left(\frac{w_r}{2 \tan(\frac{\theta}{2})} \frac{r_p h_r}{w_r} \right) \frac{p_y}{p_z} + \frac{h_r}{2}$$

It is important to note that the values for w_r , h_r , r_p , and θ are all constant for a given screen mode. This means that the terms I've isolated in parentheses in the above equations can all be pre-computed, and stored as only two values: a multiplier for the x projection, and a multiplier for the y projection. Our new projection, while far more robust, is no more computationally expensive than the one we started with!

Understanding the Dot Product

A 3D pipeline, like a real pipeline, has many individual "pipes" or stages that its contents move through. If you look closely at any 3D pipeline, you'd see that, stage after stage, a single operation appears over and over again: the dot product. It's used everywhere - transforms, lighting, clipping, backface culling - you name it, the dot product's involved somewhere. Despite it's large repertoire of services, most people don't pay it much attention - after all, it is a very simple-looking operation. Simple-looking? Yes. Simple? Certainly not! As we're about to see, there's a lot more to the dot product than meets the eye.

I'm sure everyone has seen this form of the dot product, as it is the method we all use for dot product computations:

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z$$

I would also wager that quite a few of you have seen the dot product shown as

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| |\mathbf{v}| \cos(\theta),$$

Through trigonometry, we can use the field of view to derive the relationship between w_r and d that we need. Figure 3 shows the field of view as θ , and forms the triangle relating it to w_r and d . The simple trigonometric relation is:

$$\tan\left(\frac{\theta}{2}\right) = \frac{\frac{w_r}{2}}{d}$$

For any given screen resolution, we will know the width in pixels, w_r . We specify the field of view, θ , as a measure of the perspective foreshortening we want. So, the only unknown is d , and we can easily solve for it:

which is often used to compute the angle between two vectors (the equation is easily solved for θ). Did you ever wonder why there are two common equations for the dot product?

One answer to that question comes from a geometric formulation based on the law of cosines (I'm not going to prove the law of cosines here, but its derivation is straightforward and appears in most books that cover trigonometry). From the triangle formed by the vectors \mathbf{u} , \mathbf{v} , and $\mathbf{u}-\mathbf{v}$, as shown in figure 4, the law of cosines gives us:

$$|\mathbf{u} - \mathbf{v}|^2 = |\mathbf{u}|^2 + |\mathbf{v}|^2 - 2|\mathbf{u}||\mathbf{v}|\cos(\theta)$$

Does the $|\mathbf{u}||\mathbf{v}|\cos(\theta)$ part look familiar? It should - it's one of the forms for the dot product! If we solve for it, we get:

$$|\mathbf{u}||\mathbf{v}|\cos(\theta) = \frac{|\mathbf{u}|^2 + |\mathbf{v}|^2 - |\mathbf{u} - \mathbf{v}|^2}{2}$$

Substituting in the equation for Euclidean distance for the lengths, this becomes

$$|\mathbf{u}||\mathbf{v}|\cos(\theta) = \frac{\sqrt{u_x^2 + u_y^2 + u_z^2} + \sqrt{v_x^2 + v_y^2 + v_z^2} - \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2 + (u_z - v_z)^2}}{2}$$

and, after simplifying, we are left with the equality:

$$|\mathbf{u}||\mathbf{v}|\cos(\theta) = u_x v_x + u_y v_y + u_z v_z$$

This clearly shows the equivalence of the two forms of the dot product. But this method hides much of the real relationship between the two equations.

In truth, the two forms of the dot product can be thought of as representing two different things: one a definition, and the other an operation. Put another way, we can think of this equation,

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}||\mathbf{v}|\cos(\theta), \tag{1}$$

as the *definition* of the dot product. Anything we call a dot product must obey it. Now, we can seek the other form of the dot product as the *operation* we need to perform to meet the requirement set by the definition.

Instead of trying to mutate one form of the dot product into the other by means of geometry, we can simply define *what* we want our dot product to do, *where* we want it to do it, and *how* we want it to do it. The operation we desire will naturally come out of these definitions. Equation 1 is obviously the "what". It explains exactly what the dot product should do.

Now we need to define *where* we want the dot product to work. Equation 1 has no context for its demands; it does not specify how many components the vectors \mathbf{u} and \mathbf{v} must have, or what those vectors mean. Obviously, we want them to be three-dimensional, Euclidean vectors... so the space we're concerned with is the space defined by the x, y, and z axis.

In mathematics, the concept of an axis system is more formalized. Each three-dimensional coordinate must be measured relative to a vector which defines how that coordinate is interpreted. These vectors are called *basis vectors*, and for three dimensions, they are assigned the letters \mathbf{i} , \mathbf{j} , and \mathbf{k} . They are the three unit vectors that coincide with the x, y, and z axes we commonly think of.

A point \mathbf{a} is measured by its x component along the vector \mathbf{i} , its y component along \mathbf{j} , and its z component along \mathbf{k} . We can write this symbolically as:

$$\mathbf{a} = a_x \mathbf{i} + a_y \mathbf{j} + a_z \mathbf{k} \tag{2}$$

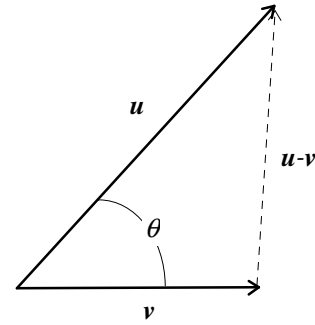


Figure 4 - The Dot Product

This says that the vector \mathbf{a} is made up of a_x units of \mathbf{i} , a_y units of \mathbf{j} , and a_z units of \mathbf{k} . This makes intuitive sense: any given vector can easily be thought of as the combination of three perpendicular vectors.

The three-dimensional basis vectors we use for 3D graphics have two important properties. First, they are mutually perpendicular. Each goes in a completely different direction than either of the other two. This property is called *orthogonality*, and we call the basis *orthogonal*. From our definition in equation 1, we know that when two vectors are perpendicular, the dot product is equal to 0 (the cosine of a right angle is 0, which makes the entire expression 0). The property of orthogonality can therefore be written as

$$\mathbf{i} \cdot \mathbf{j} = \mathbf{j} \cdot \mathbf{k} = \mathbf{k} \cdot \mathbf{i} = 0 \quad (3)$$

since \mathbf{i} , \mathbf{j} , and \mathbf{k} are mutually perpendicular and must have dot products equal to 0.

The second important property is that the basis vectors \mathbf{i} , \mathbf{j} , and \mathbf{k} are all *unit vectors*. Looking again to the definition of the dot product in equation 1, we can see that if the angle between two vectors is 0 (meaning they point in the same direction), the cosine will be equal to 1, and their dot product is equal to the product of their lengths. This means that the dot product of a vector with itself is the squared length of the vector. Using this property, we can write

$$\mathbf{i} \cdot \mathbf{i} = \mathbf{j} \cdot \mathbf{j} = \mathbf{k} \cdot \mathbf{k} = 1 \quad (4)$$

since the squares of the lengths of each of the basis vectors are all equal to 1. Because our basis vectors are all unit length, and they are orthogonal, we say they are *orthonormal*.

We have now defined what the dot product is, and where it works. We are almost ready for the alternate proof, but we still need to define *how* we want the dot product to work. First, it commutes:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a} \quad (5)$$

Second, it distributes:

$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c} \quad (6)$$

Third, multiplication by a scalar associates:

$$(d\mathbf{a}) \cdot \mathbf{b} = d(\mathbf{a} \cdot \mathbf{b}) \quad (7)$$

Finally, we are ready. Keep in mind that the real beauty in the proof is not that it shows the two forms of the dot product are equal, but rather that it embodies an elegant way of doing so. Think about the 7 definitions we have just enumerated: these are the properties of our dot product and our basis. We can think of these as requirements we have laid out that must be fulfilled. Now, we will see that we can use those requirements to generate the dot product, as some might say, “right out of thin air”. Ready? Nothing up my sleeve...

To begin, we expand the two vectors of the dot product into their component form, as we defined in equation 2.

$$\mathbf{u} \cdot \mathbf{v} = (u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k}) \cdot (v_x \mathbf{i} + v_y \mathbf{j} + v_z \mathbf{k})$$

Now, because we said the dot product distributes (equation 6), we can distribute the entire expression for \mathbf{u} over the component vectors of \mathbf{v} :

$$\mathbf{u} \cdot \mathbf{v} = (u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k}) \cdot v_x \mathbf{i} + (u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k}) \cdot v_y \mathbf{j} + (u_x \mathbf{i} + u_y \mathbf{j} + u_z \mathbf{k}) \cdot v_z \mathbf{k}$$

Distributing again, we get:

$$\begin{aligned} \mathbf{u} \cdot \mathbf{v} = & u_x \mathbf{i} \cdot v_x \mathbf{i} + u_y \mathbf{j} \cdot v_x \mathbf{i} + u_z \mathbf{k} \cdot v_x \mathbf{i} + u_x \mathbf{i} \cdot v_y \mathbf{j} + u_y \mathbf{j} \cdot v_y \mathbf{j} + u_z \mathbf{k} \cdot v_y \mathbf{j} + \\ & u_x \mathbf{i} \cdot v_z \mathbf{k} + u_y \mathbf{j} \cdot v_z \mathbf{k} + u_z \mathbf{k} \cdot v_z \mathbf{k} \end{aligned}$$

Multiplication by a scalar is associative, as we stated in equation 7, so we can group the scalars:

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x (\mathbf{i} \cdot \mathbf{i}) + u_y v_x (\mathbf{j} \cdot \mathbf{i}) + u_z v_x (\mathbf{k} \cdot \mathbf{i}) + u_x v_y (\mathbf{i} \cdot \mathbf{j}) + u_y v_y (\mathbf{j} \cdot \mathbf{j}) + u_z v_y (\mathbf{k} \cdot \mathbf{j}) + u_x v_z (\mathbf{i} \cdot \mathbf{k}) + u_y v_z (\mathbf{j} \cdot \mathbf{k}) + u_z v_z (\mathbf{k} \cdot \mathbf{k})$$

Given that the dot product of \mathbf{i} , \mathbf{j} , or \mathbf{k} with one of the other two basis vectors is equal to 0 (equations 3 and 5), the majority of the terms drop, and we get:

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x (\mathbf{i} \cdot \mathbf{i}) + u_y v_y (\mathbf{j} \cdot \mathbf{j}) + u_z v_z (\mathbf{k} \cdot \mathbf{k})$$

Finally, because the dot product of any basis vector with itself is equal to 1 (equation 4), the dot products all drop and leave the familiar

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z .$$

So there you have it. Instead of using geometric theorems, we simply stated the 7 requirements we wanted to be true of our system, and we produced the operator we call the dot product. Was it redundant to do this second prove? I would argue that we learn far more from this proof than the geometric one! First, because of our use of \mathbf{i} , \mathbf{j} , and \mathbf{k} as basis, we can see that our form of the dot product does not work with two vectors who are not in the same coordinate system... different coordinate systems means different basis vectors, and our proof clearly relies on both vectors being defined in terms of the same \mathbf{i} , \mathbf{j} , and \mathbf{k} . Second, we see that our forms of the dot product are not equivalent if our basis vectors aren't orthonormal, since we used equations 3 and 4 to do our proof. Third, and most importantly, we have seen how to *derive* a vector operation from its definition.