

# Jim Blinn's Corner



*It turns out that the obvious way to transform NDC to pixel space is wrong. To show you why, I'll have to go through several intermediate stages before I get to the correct way.*

## A Trip Down the Graphics Pipeline: Pixel Coordinates

*James F. Blinn, Caltech*

There are some problems in graphics that I've spent a lot of time trying to figure out even though they seem real simple. I don't know if this is because I'm dumb or because these problems are really profound. I've never seen any of them described anywhere in the literature; maybe the people who write books just haven't come across them.

One set of such problems comes from the use of transformations in the graphics pipeline. No, I'm not going to beleaguer you with yet more ravings about how wonderful homogeneous coordinates are. You already know that. (And if you don't, you might want to refer to the Big White Book.<sup>1</sup>) What I am going to do is discuss a few of the odd little quirks of the transformation process that I've discovered over the years and how I learned to deal with them. Some of this may look like I'm making a big deal out of nothing, but it's important to get it right.

More specifically, over the next few issues I am going to talk about

1. a detailed examination of what it means to be a pixel,
2. a nifty way to merge the old concept of the window-to-viewport transformation with clipping space that makes it easy to clip viewports, and
3. what's really going on with the homogeneous perspective transform.

These are part of a series begun with my column in the January 1991 issue ("A Trip down the Graphics Pipeline: Line

July 1991

Clipping"). For variety, I probably will intersperse these discussions with columns on other topics.

### Overview of the pipeline

Let's start with a general overview of the classic rendering pipeline. Basically the pipeline transforms points (line endpoints, polygon vertices, patch control points, and so forth) from some object's definition space to pixel raster space and, along the way, clips the shapes to the boundaries of some given region of space. The pipeline is first *initialized* by setting up various global parameters (primarily transformations), and then it is *used* by passing points through it. There are two ways to look at the setup process: from the user's *external* point of view and from the pipeline implementor's *internal* point of view. The user specifies the parameters that describe what he/she/it wants. The pipeline implementor figures out how they are stored and used by the pass-through process.

### Coordinate Spaces

First let's look at the user's view. There are various coordinate systems that a point experiences as it journeys to the screen. In the interests of solidarity I'll try to relate my notation to that used by Pixar's Renderman interface, which I will refer to as the politically correct Renderperson. In the following list the name used by Renderperson appears in parentheses. I'll list the coordinate spaces in the order in which an object sees them.

1. Definition (object)—The space in which the original object is defined.
2. Universe (world)—A consistent universe into which all modeled objects are placed.
3. Eye (camera)—Eye at origin looking down the  $z$  axis.
4. Perspective—Distorted space after the perspective transform has been performed. This one is optional. If there is an orthographic projection, this will be the same as eye space.
5. Clip—A special space chosen to make the clipper's arithmetic simple. It maps the desired visible chunk of perspective space to a fixed region.
6. Normalized device coordinates (which I will refer to as NDC) (screen)—A standardized screen coordinate system that doesn't include explicit pixel dimensions.
7. Pixel (raster)— $X, Y$  coordinates (usually integers) in the hardware pixel space. In modern usage, with multiple screen windows, this is the screen real estate we have been allocated by the windowing system.

Here are some notes about the transformations between these coordinate systems (and advertisements for back issues of this magazine):

*Definition to Universe* is generally a combination of rotation, translation, and possibly nonuniform scales used to squash primitives to the desired shape and place them in the universe. It is sometimes something more bizarre, like the bend-me, twist-me, hurt-me transforms used in general deformation systems. The ability of a  $4 \times 4$  matrix to do perspective is rarely used in modeling except in rational polynomial curves and surfaces or local light source shadows. I discussed the latter in January 1988 (*CG&A*, "Me and My (Fake) Shadow").

*Universe to Eye* is just a translation and pure rotation to place and orient the simulated camera. I wrote about this in the July 1988 column ("Where Am I, What Am I Looking at").

*Eye to Perspective* is a homogeneous perspective transform, the subject of an upcoming column.

*Perspective to Clip to NDC* is the subject of another upcoming column that fills in the blanks from my January 1991 effort ("Line Clipping").

*NDC to Pixel* is the subject of most of this column and its sequel.

### Internal operation

Now let's look at internals. Many of the above coordinate systems exist only in our minds. In actual implementation all the transforms from definition to clip space are merged into one transform which we'll call  $T_1$ . This is usually a full  $4 \times 4$  homogeneous matrix. The transforms from clip to pixel space are merged into  $T_2$ . This usually requires only a scale and offset in  $X$  and  $Y$ .

Once the transforms have been set up, processing an object through the pipe requires four steps:

1. Transform to clipping space via  $T_1$
2. Clip

3. Divide by  $w$
4. Transform to pixel space via  $T_2$

### Exemplary transformations

The only transformation I will need for the next few columns is a simple scale and offset in  $X$  and  $Y$  from some input coordinate system to an output coordinate system.

$$s_x X_{in} + d_x = X_{out}$$

$$s_y Y_{in} + d_y = Y_{out}$$

It is easiest to specify these as exemplary transformations. No, this doesn't mean that they are more admirable than others. It just means that they are specified in terms of two *example* input coordinate values and their associated desired output values. I'll denote this for, say,  $X$  as

$$X_{in1} \mapsto X_{out1}$$

$$X_{in2} \mapsto X_{out2}$$

Using the magic of linear transformations, we can derive the scale and offset as

$$s_x = \frac{X_{out2} - X_{out1}}{X_{in2} - X_{in1}}; \quad d_x = X_{out1} - s_x X_{in1}$$

These values are what is actually stored for use during the pass-through stage. For the transforms derived below, I'll just use these equations and show you the results.

### NDC to pixel transformation

Let's start with what might at first seem the simplest transformation: NDC to pixel space. The transform is

$$s_x X_{NDC} + d_x = X_{pixel}$$

$$s_y Y_{NDC} + d_y = Y_{pixel}$$

A user/programmer does all screen design in NDC. There are three nasty realities of the hardware that NDC hides from us:

1. The actual number of pixels in  $x$  and  $y$ .
2. Nonsquare pixels. Distances in NDC space are geometrically uniform.
3. Up versus down for the  $Y$  coordinate. It will invert  $Y$  if necessary so that  $Y$  in NDC points up.

The transformation to pixel space is implicit in the system and is set up during device initialization. It turns out that the obvious way to transform NDC to pixel space is wrong. To show you why, I'll have to go through several intermediate stages before I get to the correct way. But without all the false starts the correct way will seem kind of unmotivated. So don't take any of these stepping stone transformations too much to heart until I tell you we've arrived at the ultimate one.

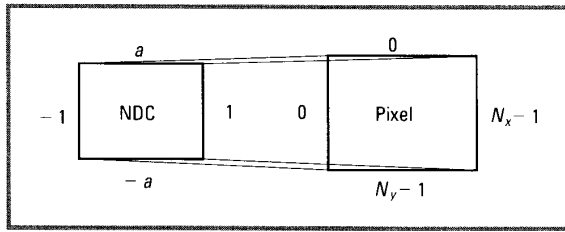


Figure 1. The naive NDC to pixel model.

First let's specify precisely what I mean by the different coordinate systems.

### Pixel space

Define the pixel coordinate system according to the following typical scheme. The pixel array is  $N_x$  pixels across, numbered from 0 on the left to  $N_x - 1$  on the right. The array is  $N_y$  pixels high, numbered from 0 on the top to  $N_y - 1$  on the bottom. Now, in addition to deriving general formulas, I'm going to use explicit values in examples below. I'll use 512 pixels in  $X$  and 480 pixels in  $Y$ , not just because they are typical but also because they are familiar and you'll be able to see what's going on with them.

Sometimes the pixel array doesn't start at  $(0, 0)$  but at some offset. I won't specifically deal with this here; it only requires adding the constant  $(X, Y)$  offset to the pixel coordinates we calculate. Also, sometimes  $Y = 0$  at the bottom of the screen, instead of at the top. I'll let you figure out the slightly different transform necessary in that case.

Now in addition to pixel counts the display hardware will imply a physical aspect ratio for this rectangle of pixels that might not necessarily equal  $N_y/N_x$ . Thus, pixels might not be square. For our numerical example, I'll use a standard TV screen which has an aspect ratio of  $3:4 = 0.75$ . This number represents the height divided by the width of the region that the display electronics will scan out. For some reason Renderperson describes the aspect ratio as width over height.

At some level it might not make any difference. I think the aspect-ratio-sensing neurons of the general public are getting burned out because so many computer displays are not really adjusted to the proper hardware aspect ratio and are distorted horizontally. In addition, people see so many images on TV that are distorted via digital video effects that pretty soon they don't even notice it anymore. This is similar to how my flicker sensors have been burned out from my early years of looking at displays that refreshed only about 10 times per second.

### NDC space

Normalized device coordinates run from  $-1$  to  $+1$  in  $X$  and  $-a$  to  $+a$  in  $Y$ , where  $a$  is the above mentioned physical aspect ratio. NDC thus has a little bit of device dependency (see the left side of Figure 1).

The Renderperson approach is perhaps a bit better. It guarantees that the region  $-1$  to  $+1$  is visible in both  $X$  and  $Y$ ; the longer dimension then extends to values beyond 1. They would therefore say that TV screen space extends from  $-1.333$  to  $+1.333$  in  $X$  and from  $-1$  to  $+1$  in  $Y$ . I'll stick to my way for now so I don't get confused.

### The naive transform

In Figure 1 we see our first attempt at the required transformation. Let's derive the exemplary transformation from this figure. In  $X$

$$-1 \mapsto 0$$

$$1 \mapsto N_x - 1$$

This leads to

$$s_x = \frac{N_x - 1}{2}; d_x = \frac{N_x - 1}{2}$$

For our sample pixel range this leads to

$$s_x = 255.5; d_x = 255.5$$

Now we do the same thing for  $Y$ . Here we incorporate the physical aspect ratio of the screen,  $a$ , as the vertical bounds for NDC. This implicitly corrects for nonsquare pixels.

$$-a \mapsto N_y - 1$$

$$a \mapsto 0$$

so

$$s_y = \frac{N_y - 1}{-2a}; d_y = \frac{N_y - 1}{2}$$

For our sample pixel range this leads to

$$s_y = -319.333; d_y = 239.5$$

Every computer graphics book I've seen (at least every one that describes this explicitly enough to tell) uses this transformation.

### A crinkle

There's a problem lurking here, though. The scale factor is icky. In the  $X$  direction we are taking two units of NDC space and dividing it up into  $N_x$  pixels. We sort of expect the scale factor to be  $N_x/2$ . What did we do wrong? We forgot that pixels have size. Imagine a row of pixels with a Gaussian spot centered at each one and you realize that the resultant intensity swath sticks out about  $1/2$  pixel width to the left and right.

This little offset doesn't seem like much, but it can really screw things up for more advanced rendering situations. I did this wrong in my original teapot pictures.<sup>2</sup> There, due to memory limitations, I had to make each  $512 \times 512$ -pixel image from four  $256 \times 256$ -pixel quadrants using a variant of the naive pixel mapping. For example, for the left two quadrants, the  $X$  mapping was  $-1 \mapsto 0$  and  $0 \mapsto 255$ , and for the right

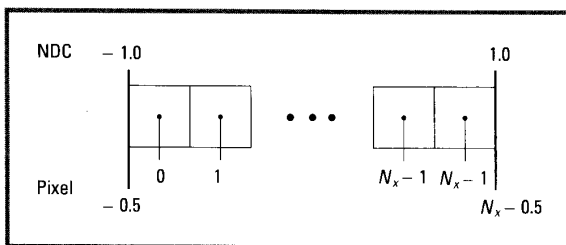


Figure 2. Microscopic view of pixels at the edge of NDC.

two quadrants, it was  $0 \mapsto 0$  and  $1 \mapsto 255$ . When the four pictures were composited, you could see a seam. (This was only really noticeable when the digital image was zoomed up—another example of the principle that if you shrink a crummy picture down, it looks good.)

So what's the right way? Looking at the pixels through a microscope, what we *really* want is shown in Figure 2. For the  $X$  coordinates, we use

$$\begin{aligned} -1 &\mapsto -0.5 \\ 1 &\mapsto N_x - 0.5 \end{aligned}$$

so

$$s_x = \frac{N_x}{2}; \quad d_x = \frac{N_x - 1}{2}$$

With explicit coordinates we get

$$s_x = 256; \quad d_x = 255.5$$

For the  $Y$  coordinate, we use

$$\begin{aligned} -a &\mapsto N_y - 0.5 \\ a &\mapsto -0.5 \end{aligned}$$

so

$$s_y = \frac{-N_y}{2a}; \quad d_y = \frac{N_y - 1}{2}$$

and

$$s_y = -320; \quad d_y = 239.5$$

### Integerizing

When we go from floating-point pixel coordinates to integer pixel coordinates, we need to round to the nearest pixel. This is the same as adding 0.5 and truncating. We might as well build this into the displacement part of our transformation. This implies an  $X$  map of

$$\begin{aligned} -1 &\mapsto 0 \\ 1 &\mapsto N_x \end{aligned}$$

and a  $Y$  map of

$$\begin{aligned} -a &\mapsto N_y \\ a &\mapsto 0 \end{aligned}$$

This changes just the displacements to

$$d_x = \frac{N_x}{2} = 256$$

$$d_y = \frac{N_y}{2} = 240$$

This works as long as all the numbers are positive, which they are here.

### The dangling edge

Now there's another problem. An NDC  $X$  coordinate value of 1.0 will map, after truncation, into  $N_x$ , which is one pixel beyond the right edge of the screen. There are two ways we can react to this situation. First, we can disallow the value of +1.0 as a visible coordinate, that is, fix it so the clipper keeps things only in the region  $-1 \leq X < +1$ . But this leads to some unexpected and undesirable results. For example, if you try to frame the screen by lines tracing what you think are the screen boundaries ( $-1$  and  $+1$ ), you won't get to see the line on the right.

Second, we can arrange to keep this one extra NDC value in a way that won't mess us up. Set up the clipper to keep both ends:  $-1 \leq X \leq +1$ . We can then test for the pixel coordinate of  $N_x$  and explicitly set it back to  $N_x - 1$ , but this is not what I do. I do what all good computer graphicists do, I cheat. I contract the NDC space by a microscopic amount to fit the extra one value into the allowable pixel range.

In  $X$ ,

$$\begin{aligned} -1 &\mapsto 0 \\ +1 &\mapsto N_x - \epsilon \end{aligned}$$

This will make  $+1$  truncate to  $N_x - 1$ . The scale and displacement are

$$s_x = \frac{N_x - \epsilon}{2}; \quad d_x = \frac{N_x - \epsilon}{2}$$

You can use numerical analysis to figure out the optimal value for  $\epsilon$  but I just use 0.001, which is pretty close to the minimum value whose effect still fits in a single precision number.

$$s_x = 255.9995; \quad d_x = 255.9995$$

In  $Y$ ,

$$\begin{aligned} -a &\mapsto N_y - \epsilon \\ +a &\mapsto 0 \end{aligned}$$

so

$$s_y = \frac{N_y - \epsilon}{-2a}, \quad d_y = \frac{N_y - \epsilon}{2}$$

and

$$s_y = -319.9995; \quad d_y = 239.9995$$

This, then, is the final approved NDC to pixel transformation. Write this down.

### What is pixel space really?

So what have we done here? Considering just the  $X$  direction, we've divided the conceptually continuous  $-1$  to  $+1$  range of NDC up into  $N_x$  discrete bins, each of which maps to a pixel. There are actually two ways to think about this. We can place the integer pixel coordinate at the center of the pixel and then round each coordinate to the nearest integer. (This was one of our intermediate techniques.) Alternatively, we can place the integer pixel coordinate in the cracks between pixels and truncate to the next lowest integer. This is what we finally wound up with. Note that the pixel areas are geometrically the same for both techniques. The left-most pixel covers  $2/N_x$  amount of distance in NDC, abutting the left edge of the screen.

Simple line drawing algorithms and many early polygon scan conversion algorithms accept line endpoints and polygon vertices as integer pixel coordinates. That's great for quick-look line drawings, plotters, or other output devices that take integer pixel coordinates. But there's more to life than line drawings.

Mulling over some more advanced rendering applications leads to two other requirements of the NDC to pixel mapping: subpixel positioning of endpoints/vertices and subsampling of the image for antialiasing purposes. These two enhancements to the transformation interact in strange and mysterious ways and will be the topic of next issue's column. □

### References

1. J.D. Foley et al., *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, Mass., 1990.
2. J. Blinn, "Texture and Reflection in Computer-Generated Images," *CACM*, Vol. 19, No. 10, Oct. 1976, pp. 542-47.

### Post mortem on the last issue

I was pretty happy about the appearance of the diagrams from last issue's column about printing quality ("WYSBOAVRTWYG"). I was especially happy that *all* my very thin lines survived the printing process. It did look like the 0.08 (2.8 pixel) and 0.04 (1.4 pixel) point lines were the same width, though. The halftone lines didn't look bad either, despite my worries about line position interacting with the halftone pattern. You *can* see the effect I was talking about in Figure 2 by looking at Figure 3 through a loupe (that's a small magnifying glass for those of you who don't speak French).

July 1991

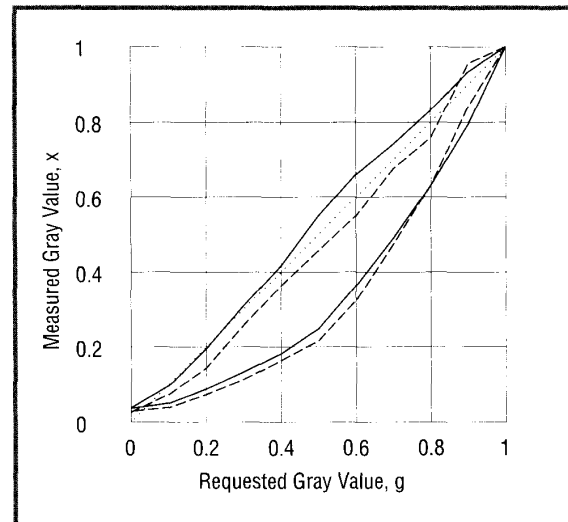


Figure 3. Measurement of compensation of CG&A Postscript-to-print distortion.

The compensation table example in Figure 14 looked a lot less murky than the uncompensated version in Figure 8. Because of the magazine's layout, you can easily compare the figures side by side by folding the page over. Finally, as promised, Figure 3 is a graph of the measured gray values of the compensated wedge (which should ideally be an identity function) compared to the uncompensated wedge. The dotted line represents the ideal identity function. On top of this I have plotted measurements of the original and compensated wedges as measured by two different densitometers (one meter is the solid lines and the other is the dashed lines). Using two meters gives you an idea of how accurate a densitometer reading is likely to be. The two sagging lines are from the uncorrected wedge. The two straightish lines are from the corrected wedge, not too bad a result, considering that the original data was printed almost six months before I used it to make the compensation function.

**Editor's Note:** Jim Blinn recently received a MacArthur Fellowship for his work in education and computer graphics. Awarding Blinn a grant totaling \$265,000, the John D. and Catherine T. MacArthur Foundation stated that, "in designing computer graphics, he works simultaneously as an artist, educator, mathematician, and scientist." Blinn is among the 31 scientists, scholars, and artists who have received this prestigious award this year. This news reached our offices right before press time. More complete coverage will appear in our next issue.