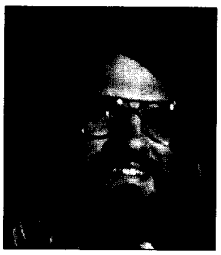
Jim Blinn's Corner



There has always been something creepy about the window-to-viewport transformation. I extracted the desirable aspects of W-to-V while discarding the disquieting ones.

A Trip Down the Graphics Pipeline: Grandpa, What Does "Viewport" Mean?

James F. Blinn, Caltech

Computer graphics has never been much for uniformity in terminology, what with left- or right-handed coordinate systems, row- or column-vectors notation for points, and clockwise or counterclockwise rotation rules and polygon ordering. This is, I guess, the result of different research groups from all over the world approaching similar problems with their own notational conventions.

One casualty in the war of words is the term "window" and its relation to the term "viewport." Back in the Old Days (a phrase I find myself using increasingly frequently) there was something called a window-to-viewport transformation that programmers used to specify which parts of a geometrical database should be mapped to what part of the screen. In that scenario, the term "viewport" meant the rectangle on the screen where the picture went, and "window" meant the region of the database coordinate system that should be mapped there. But now there's considerable commercial and political pressure to use the term "window" to mean all those rectangular pieces of screen with their enclosing frames, title bars, and assorted bric-a-brac. Confusion reigns if you are not sure which type of window somebody might be talking about.

This is not all bad. There has always been something creepy about the window-to-viewport transformation that I have just recently resolved in my own mind. So this month I'm going to describe how I extracted the desirable aspects of W-to-V transformation while discarding the disquieting ones.

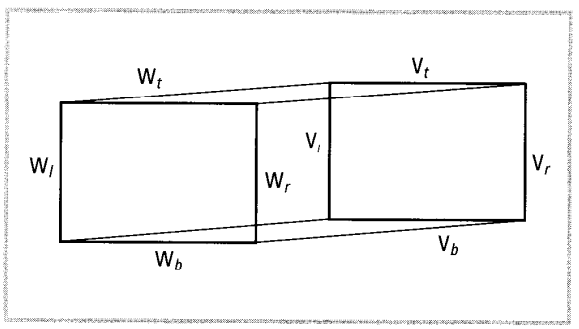


Figure 1. The window-to-viewport map.

The classic window-to-viewport transform

The original idea here is pretty straightforward. In their desired coordinate space, users specify a rectangular region that they want to display, the *window*, as W_l, W_r, W_b, W_t. Then they pick a rectangular region on the screen, the *viewport*, where they want to put it, V_l, V_r, V_b, V_t. This looks like Figure 1. The computer then miraculously calculates and performs the necessary scale and offset to get from one region to the other:

$$X_{view} = s_x X_{wind} + d_x$$
$$Y_{view} = s_y Y_{wind} + d_y \qquad (1)$$

where the scale and displacement come from the requirements

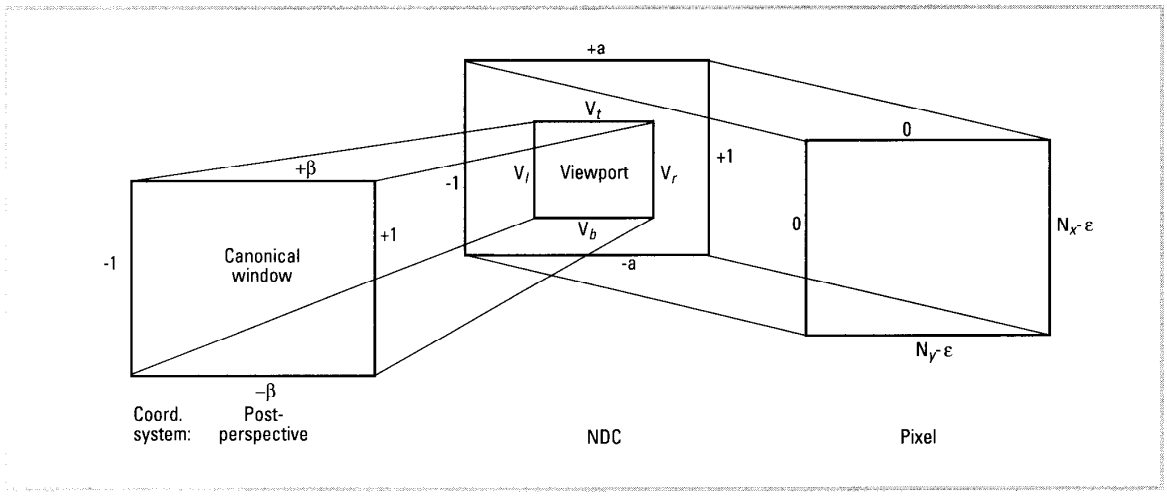


Figure 2. The canonical window-to-pixel map.

$$W_l \mapsto V_l$$
$$W_r \mapsto V_r$$
$$W_b \mapsto V_b$$
$$W_t \mapsto V_t$$

This, using the exemplary transformation technique mentioned in my earlier article ("A Trip Down the Graphics Pipeline: Pixel Coordinates," *CG&A*, July 1991), gives us

$$s_x = \frac{V_r - V_l}{W_r - W_l}$$

$$d_x = V_l - s_x W_l$$

$$s_y = \frac{V_t - V_b}{W_t - W_b}$$

$$d_y = V_b - s_y W_b \qquad (2)$$

We expect to precalculate these scales and offsets during some initialization routine and save them in some static variables. Then we use them in drawing routines that transform points via Equation 1.

How does this fit in?

How does this relate to the graphics pipeline I have been discussing in the last few columns? Let's review the coordinate systems in the pipeline (leaving out clipping for now).

Definition: You define objects in this system.

Universe: Objects are placed via possibly nested modeling transformations in a consistent universe model.

Eye: An eye location and viewing direction are selected and the universe is transformed so that the eye is at the origin and the viewing direction is down the z axis.

Postperspective: A perspective distortion transforms the viewing pyramid to a parallel-sided rectangular parallelopiped (a "brick" in the vernacular).

Normalized device coordinates (NDC): The viewable brick is squashed into a region of a device-independent screen coordinate system.

Pixel: Device-dependent code reinterprets the NDC into the correct hardware pixel coordinates.

According to the original usage of the term "window-to-viewport transformation," the window would be in definition space and the viewport in pixel space. But this doesn't work very well. First, it's better to specify the viewport in NDC space for device independence. No problem; that doesn't change things conceptually. But the window specification in definition space is usually inconvenient if we are doing complex nested 3D transforms. In fact, we already have a mechanism (full 4×4 homogeneous matrices) to do much more general transformations than the simple scale-and-offset of W-to-V. Furthermore, there is something clumsy about the W-to-V we have so far defined—nonuniform scale factors. If we pick a window and a viewport that don't happen to have the same aspect ratio, that is, if

$$\frac{W_t - W_b}{W_r - W_l} \neq \frac{V_t - V_b}{V_r - V_l}$$

the scale factor will be different in x and y. I have rarely found this to be useful, and in fact it's usually something to avoid. Though we can force the above two ratios to be equal by fiddling with one or the other of the rectangles, it's a nuisance.

For these reasons, I adopted a modified form of the W-to-V transform as part of the initialization of my image rendering system. The initialization takes a viewport (though the modern term would now be "window") specified in NDC. In fact, for animation I find it more convenient to describe the viewport in terms of its center (V_x, V_y), size (V_s), and aspect ratio (V_a). I then generate the boundaries via

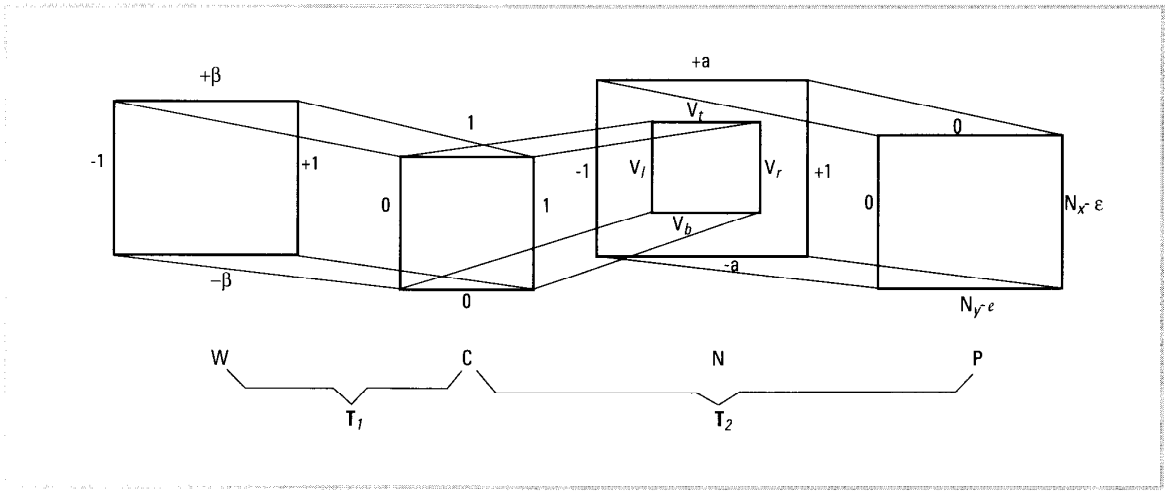


Figure 3. The window-to-pixel map including clipping.

$$V_l = V_x - V_s$$
$$V_r = V_x + V_s$$
$$V_b = V_y - V_s V_a$$
$$V_t = V_y + V_s V_a$$

I then force the viewport to be mapped from a "canonical window" in postperspective space with the same aspect ratio (see Figure 2). This canonical window always has the boundaries

$$W_l = -1$$
$$W_r = +1$$
$$W_b = -\beta$$
$$W_t = +\beta$$

where β is the aspect ratio of the viewport, that is,

$$\beta = \frac{V_t - V_b}{V_r - V_l}$$

Plugging these constants into Equations 2 gives

$$s_x = \frac{V_r - V_l}{2}$$
$$s_y = \frac{V_t - V_b}{2\beta}$$

but using the definition of β, we figure that this degenerates to

$$s_x = s_y = \frac{V_r - V_l}{2}$$

Voila! Equal scale factors in x and y. Whereupon

$$d_x = \frac{V_r + V_l}{2}$$

$$d_y = \frac{V_b + V_t}{2}$$

I then merge this transformation with the NDC-to-pixel transformation built up in the pixel coordinates article (*CG&A*, July, 1991) to form one big fat scale and displacement to go from postperspective coordinates directly to pixel coordinates. This is what I use to initialize the 4 × 4 "current transformation matrix" (CTM) that multiplies all points entered into the graphic system. After this initialization, the user can multiply in other transformations as described in the article "Nested Transformations and Blobby Man" (*CG&A*, October 1987). Multiplying in a perspective matrix makes the CTM go from eye space to pixel space. Multiplying in a pure rotation and a translation makes it go from universe space to pixel space. Multiplying in some other modeling transformations makes it go from definition space to pixel space. The whole trick is that you start not with an identity matrix, but with the above scale and offset from the canonical window to the user-selected viewport.

Clipping

Not quite. We still have to fit clipping into our world view. In my January 1991 column (*CG&A*, "A Trip down the Graphics Pipeline: Line Clipping") I described how clipping can be real fast if it's done to its own standardized region: (0, 1) in x, y, and z. To accommodate this, we split our newly minted W-to-V transformation into two parts (see Figure 3).

First, let's pause for notation control. We are shortly going to experience a whole lot of scales and displacements between a whole lot of intermediate coordinate systems. I will abbreviate the coordinate systems with the single letters:

P: Pixel space
N: NDC space. The viewport is defined here.

C: Clip space

W: Postperspective space. We already used P, and the canonical window is defined in this space, hence W.

Then the scale and displacement to take x from clip space to NDC space would be denoted

$$S_{x\mathrm{CN}},\ D_{x\mathrm{CN}}$$

and what we've been calling s_x is really $S_{x\mathrm{WN}}$, and similarly for s_x, d_x, and d_y.

Now back to our program. Part one of the fragmented W-to-V maps the canonical window to the canonical clipping region. In x

$$-1 \mapsto 0 \text{ and } +1 \mapsto 1$$

In y

$$-\beta \mapsto 0 \text{ and } +\beta \mapsto 1$$

Result:

$$S_{x\mathrm{WC}} = 1/2,\ D_{x\mathrm{WC}} = 1/2$$
$$S_{y\mathrm{WC}} = 1/(2\beta),\ D_{y\mathrm{WC}} = 1/2$$

Part two of the transform maps the clipping region to the viewport. In x

$$0 \mapsto V_l \text{ and } 1 \mapsto V_r$$

In y

$$0 \mapsto V_b \text{ and } 1 \mapsto V_t$$

Result:

$$S_{x\mathrm{CN}} = V_r - V_l,\ D_{x\mathrm{CN}} = V_l$$
$$S_{y\mathrm{CN}} = V_t - V_b,\ D_{y\mathrm{CN}} = V_b$$

Now, if all is right with the world, the composition of these two should give us back the new-and-improved W-to-V transform we derived in the previous section. Let's see:

$$s_x = S_{x\mathrm{WC}}S_{x\mathrm{CN}} = \frac{V_r - V_l}{2}$$

$$d_x = D_{x\mathrm{WC}}S_{x\mathrm{CN}} + D_{x\mathrm{CN}} = \frac{V_r + V_l}{2}$$

$$s_y = S_{y\mathrm{WC}}S_{y\mathrm{CN}} = \frac{V_t - V_b}{2\beta} = \frac{V_r - V_l}{2}$$

$$d_y = D_{y\mathrm{WC}}S_{y\mathrm{CN}} + D_{y\mathrm{CN}} = \frac{V_r + V_l}{2}$$

It works. This means that the insertion of clip space between perspective and NDC spaces has no net effect on the geometry of viewing or where the picture gets mapped.

The WC transform is what we actually use to initialize the current transformation matrix. I called it $\mathbf{T}_1$ in the clipping article. Next we compose the CN part of the transform with the NP transform (NDC to pixel) to take us straight from clip space to pixel space. I called this scale and displacement $\mathbf{T}_2$ in the clipping article, and we apply it after clipping and after the homogeneous division by w.

Off-screen viewports

Now what if some Bozo—I'm sorry—what if some user tries to initialize a viewport with coordinates that extend outside the allowable NDC range of $(-1, +1)$ in x or $(-a, +a)$ in y? What do we do? Well, what would be reasonable to do? If users move the viewport past the edge of the screen, they would expect to see the part of the viewport that is still visible with the dangling part clipped off. It so happens that we can use the same clipping machinery we already have lying around to accomplish this. We just do some more elaborate initialization of $\mathbf{T}_1$ and $\mathbf{T}_2$. We'll do this by a four-step process:

1. Find the intersection of the given viewport rectangle with the available NDC range. This forms the *visible viewport range*, a subset of the given viewport.
2. Inverse transform the visible viewport range to postperspective space to determine that chunk of the canonical window that the clipper should keep.
3. Find the transform from that region of the canonical window to the $(0, 1)$ clip region. This is $\mathbf{T}_1$.
4. Find the transform from the clip region to the visible viewport range. Merge this with the NP transform. This is $\mathbf{T}_2$.

First, let's take care of the extreme situation: What if someone shoves the viewport completely off the screen? It might happen, you never can tell. This occurs if

$$V_l \geq 1 \text{ or } V_r \leq -1 \text{ or } V_b \geq +a \text{ or } V_t \leq -a$$

In this case, we want to set things up so that *everything* gets clipped off. One way to do this is just to force

$$S_{x\mathrm{WC}} = 0,\ D_{x\mathrm{WC}} = -1$$

$$S_{y\mathrm{WC}} = 0,\ D_{y\mathrm{WC}} = -1$$

That is, the universe is shrunk to a point outside the clipping boundaries. Alternatively, you can set a switch inside the clipper, causing it to return immediately upon each call.

But what if the viewport is at least somewhat visible? (Look at Figure 4.) In general, the visible viewport range has the boundaries

$$U_l = \max(V_l, -1)$$
$$U_r = \min(V_r, +1)$$
$$U_b = \max(V_b, -a)$$
$$U_t = \min(V_t, +a)$$

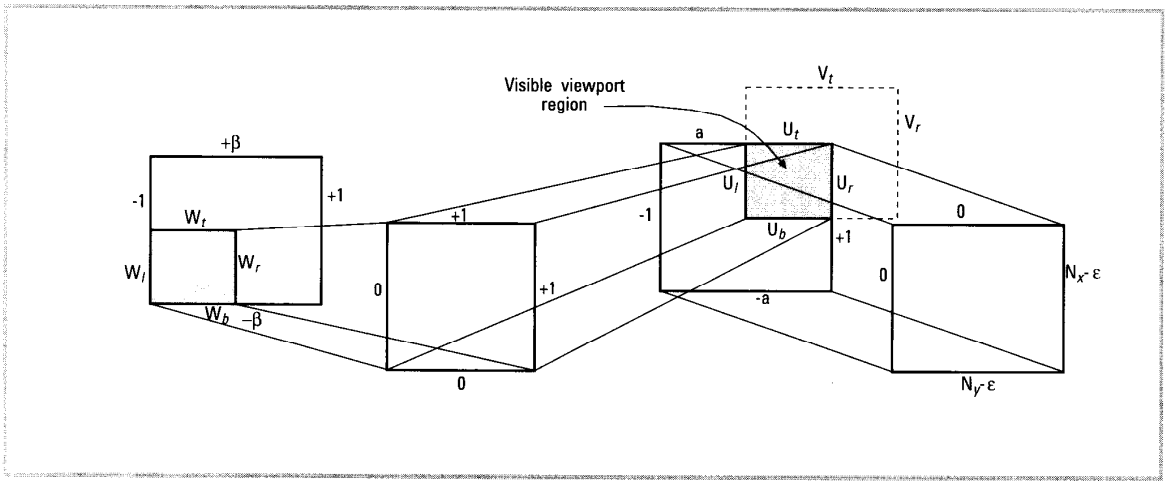


Figure 4. The off-screen viewports.

Now inverse transform these boundaries back to postperspective space to get the new window boundaries

$$W_l = \frac{(U_l - d_x)}{s_x} = \frac{2U_l - V_r - V_l}{V_r - V_l}$$

$$W_r = \frac{(U_r - d_x)}{s_x} = \frac{2U_r - V_r - V_l}{V_r - V_l}$$

$$W_b = \frac{(U_b - d_y)}{s_x} = \frac{2U_b - V_t - V_b}{V_r - V_l}$$

$$W_t = \frac{(U_t - d_y)}{s_x} = \frac{2U_t - V_t - V_b}{V_r - V_l}$$

The denominators above for y are not typos. Remember that $s_x = s_y$. Note that, as a reality check, if $U_l = V_l$, the above expression for W_l reduces to –1, and similarly for the other boundaries.

To get the new $\mathbf{T}_1$, we map in x

$$W_l \mapsto 0 \text{ and } W_r \mapsto 1$$

and in y

$$W_b \mapsto 0 \text{ and } W_t \mapsto 1$$

with the result

$$S_{xWC} = 1/(W_r - W_l),\ D_{xWC} = -W_l/(W_r - W_l)$$
$$S_{yWC} = 1/(W_t - W_b),\ D_{yWC} = -W_b/(W_t - W_b)$$

If you want, you can plug in the W definitions and boil this down to

$$S_{xWC} = \frac{V_r - V_l}{2(U_r - U_l)},\ D_{xWC} = \frac{-2U_l + V_r + V_l}{2(U_r - U_l)}$$

$$S_{yWC} = \frac{V_r - V_l}{2(U_t - U_b)},\ D_{yWC} = \frac{-2U_b + V_t + V_b}{2(U_t - U_b)}$$

and you never have to explicitly calculate the Ws. Warning! Danger: There is a potential for zero division here. You'd better check for $U_r = U_l$ and $U_t = U_b$ and avoid having the rug pulled out from under you if they are equal. This can only happen if the requested viewport has an x or y size of zero. In that case, just use the transformation from the previous section.

Finally, part two of the transform maps the clipping region to the visible viewport. In x

$$0 \mapsto U_l \text{ and } 1 \mapsto U_r$$

and in y

$$0 \mapsto U_b \text{ and } 1 \mapsto U_t$$

with the result

$$S_{xCN} = U_r - U_l,\ D_{xCN} = U_l$$
$$S_{yCN} = U_t - U_b,\ D_{yCN} = U_b$$

Merge this with the NP transform and you're in business.

So what?

I've rarely seen any computer graphics books address the problem of nonsquare screens during initialization of the transforms of the graphics pipeline. The off-screen viewport calculation does this nicely. You just initialize a default viewport to go (–1, +1) in both x and y. If this sticks out beyond the NDC range, for example in y for a landscape-oriented display, the proper visible subregion will be automatically extracted with CTM set up to map that region to the standard clip region. The clipper is none the wiser, but you get a nondistorted picture on the screen. It's taken me a good long time to figure this out, but now I'm finally happy with it. ❑