OpenGL Viewport Transform: Derivation

3D graphics pipes use two steps called ***shoot*** and ***print*** to display images on computer and television display devices. The *shoot* step is similar to capturing an image of the scene on photo film using an analog camera. Instead of capturing the image on a 2D photo film, the graphics pipe captures the image of the scene in a 3D normalized device context (NDC) box which is the graphics pipe equivalent of 2D photo film. The *print* step involves the proportional scaling of the NDC image to fit the rectangular viewport. This step is analogous to printing on photosensitive paper the image captured on the 2D photo film. This document describes the mathematical derivations of the *print* step implemented by OpenGL graphics drivers.

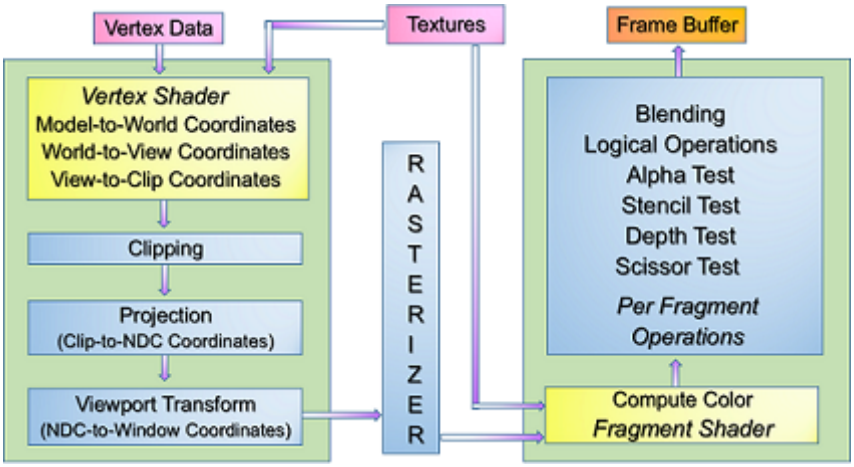

Figure 1: Simplified 3D graphics pipe consisting of vertex and fragment shaders fragments

Brief Overview of the *shoot* step

To provide some context to the *print* step and for the sake of completeness, let's briefly review the *shoot* step. Figure 1 illustrates the conceptual model of a 3D graphics rendering pipeline while Figure 2 illustrates the various coordinate systems encountered by vertices during their trip through the graphics pipe.

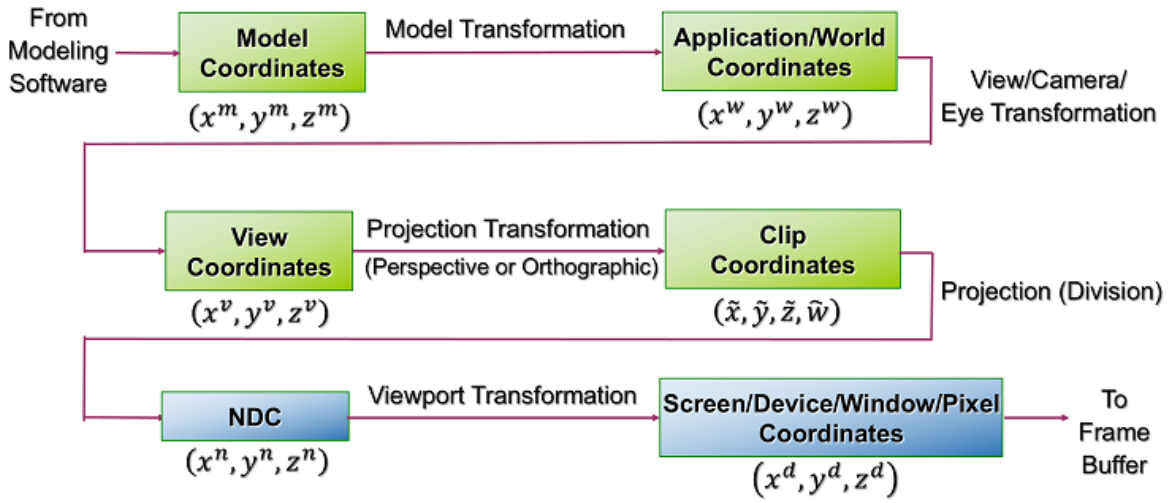

Figure 2: Coordinate systems encountered during trip through graphics pipeline

Databases of graphics applications contain geometric data describing 3D ***models***. with each model described by the modeling software in a convenient coordinate coordinate called ***model coordinate system***. In modern graphics APIs, these models are buffered in GPU memory. Applications use the idea of ***instancing*** to build and populate huge virtual worlds using a limited

number of models. For example, a game requiring millions of butterflies will buffer a single butterfly model in GPU memory and reuse the buffered model to render many instances with different dimensions, orientations, and at different locations in the game world. Each instance of a model is called an ***object***. To specify each object's dimension, orientation, and location in the world, the application will compute a ***model-to-world*** (or *model* or *object*) ***transformation matrix***. Next, a camera is inserted into the scene. Using the position and orientation of the camera, the application computes a ***view transformation matrix*** that transforms vertices described in world coordinates to ***viewing*** (or ***camera*** or ***eye***) ***coordinate system***. The origin and coordinate axes of the viewing coordinate system are specified by the camera's position and orientation in the world. A film camera creates an image by letting reflected light from objects in its field of view to be incident on photo film and cause a chemical reaction. To simulate this behavior, graphics pipes use two steps. In the first step, a projection (either orthographic or perspective) transformation matrix maps points from view coordinate system to ***clip coordinate system*** where ***clipping*** is implemented. The purpose of the clipping stage is to remove portions of the scene outside the virtual camera's field of view.

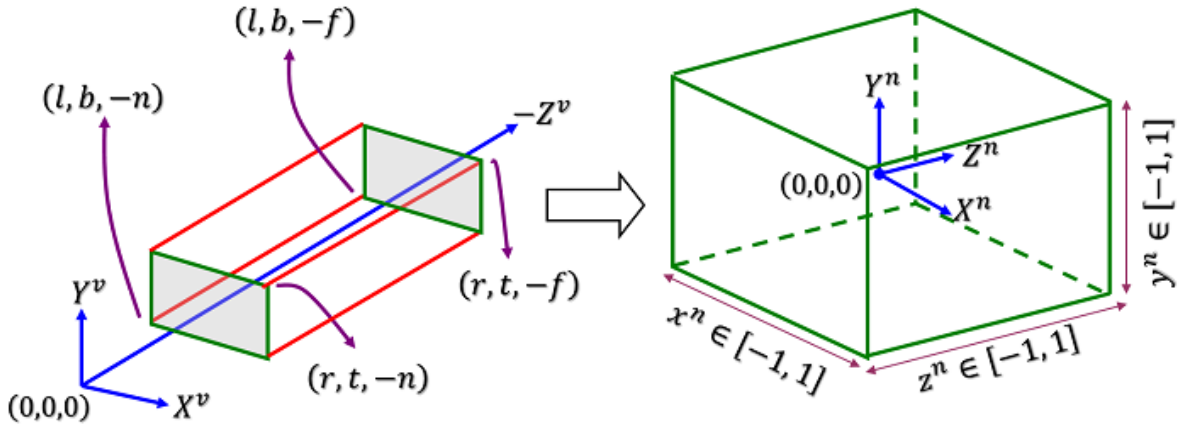

Figure 3: Projection of orthographic viewing box to NDC box

Figure 3 illustrates the viewing box generated by orthographic projection. Graphics primitives inside the viewing box are visible to the camera but those straddling the box or completely outside the box must be clipped out. Film cameras don't require clipping because laws of physics will prevent reflected light rays from outside the camera's field of view to enter the camera. In the second step labeled ***projection***, points in the clip coordinate system are projected from clip coordinates to a box in ***normalized device context (NDC) coordinates***. The OpenGL specification defines NDC box extents as $x^n \in [-1, 1]$, $y^n \in [-1, 1]$, $z^n \in [-1, 1]$ while DirectX defines these extents as $x^n \in [-1, 1]$, $y^n \in [-1, 1]$, $z^n \in [0, 1]$. The superscripts v and n in Figure 3 indicate the ***v***iewing and ***N***DC coordinate systems. The OpenGL NDC box is shown on the right side of Figure 3. The NDC box in a graphics pipe is analogous to photo film in a camera. After a photo is taken, the film is developed in a darkroom and printed to special photosensitive paper of a specific size to obtain a photograph. Analogously in a graphics pipe, the projected contents of the NDC box must be mapped to the coordinates described by the physical device on which the image is being viewed.

What is a *viewport*?

The basic unit of display measure in a display device is typically called the ***pixel***. It is common for operating systems to describe pixels on the display screen with origin at top-left corner of the display device, x axis oriented right, and y axis oriented down. We'll refer to this coordinate system as the ***device coordinate system***. The left side of Figure 4 shows the display region of a device such as a computer monitor or a television screen.

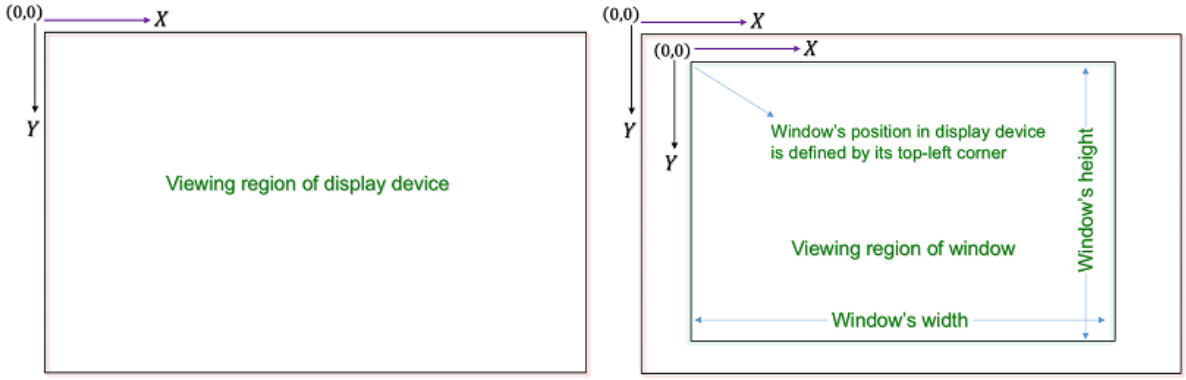

Figure 4: Left: Device coordinate system. Right: Windows positioned and sized in display devices using screen coordinate system.

Modern operating systems use the concept of windows to display formatted text and images. A window is positioned and sized in the device coordinate system. The window itself represents a base coordinate system for any applications that draw graphics images or text into the window with most operating systems specifying the window's top-left corner as its origin. Points in the window are addressed using ***client coordinates***. As shown by the right side of Figure 4, client coordinates allow applications to deal with displaying things in a window without concerning themselves with where the window itself is located on the desktop. The system will perform the necessary offset calculations to map client coordinates of things displayed in a window to device coordinates.

A graphics rendering pipelines' task (see Figure 1) is to render the scene from a camera's point of view to a region of GPU memory called the ***framebuffer***. In non-windowed systems such as consoles, the framebuffer corresponds to the entire viewing region of the display device. In a windowing system, a graphics application's notion of a framebuffer corresponds to the drawing context of a particular window assigned to the application. OpenGL requires windowing systems to provide a window to which OpenGL's framebuffer is attached. The storage, dimensions, allocation, and format of the images attached to this framebuffer are managed entirely by the windowing system. However, OpenGL uses a different client coordinate system than windowing systems to identify points or pixels in the window to which OpenGL's framebuffer is attached. *Confusingly*, the OpenGL specification uses the term ***window coordinate system*** to identify this coordinate system. Unlike windowing systems, OpenGL identifies points in the framebuffer using the bottom-left corner (of the framebuffer) as origin, x axis oriented right, and y axis oriented up. The left side of Figure 5 shows OpenGL's window coordinate system in relation to the operating system's windowing coordinate system. Notice that the document will refer to window coordinates using superscript d because superscript w is reserved for specifying world coordinates. When using OpenGL, programmers are not concerned about the mapping of pixels defined in OpenGL's window coordinate system to the windowing system's coordinate system. This mapping is implemented by graphics drivers and is transparent to OpenGL programmers.

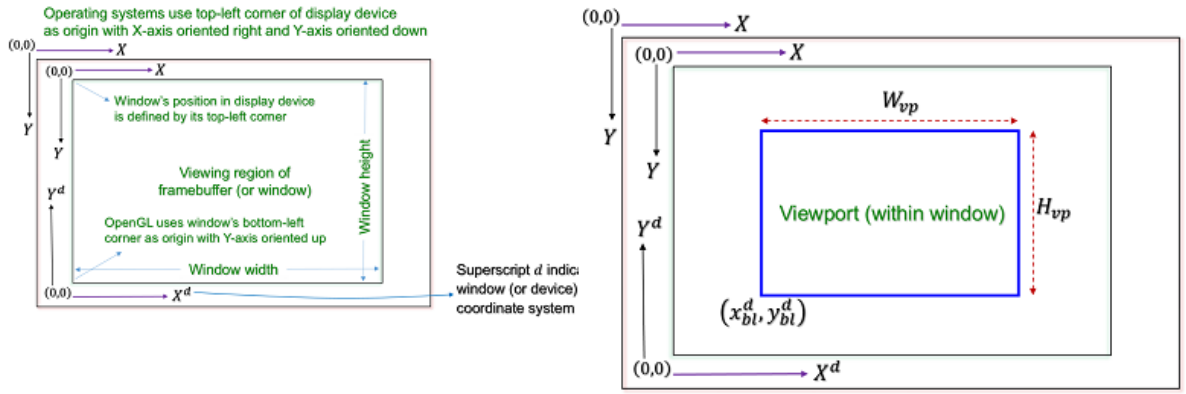

Figure 5: Left: OpenGL framebuffer and window coordinates. Right: OpenGL viewport in framebuffer.

Graphics applications need not use the entire framebuffer to display images. Instead, they use the concept of viewports to present multiple images to the framebuffer. A ***viewport*** defines the region of a framebuffer where contents of the NDC box are displayed. To display multi-player console games on television screens, each player is assigned a camera whose associated NDC image is mapped to a corresponding viewport in the framebuffer. Single player games use multiple viewports too. For example, a racing game will use most of the framebuffer to show the image representing the driver's view of the world, a second viewport presents the image seen by a rear-view camera, two additional viewports might present images seen by side-mirrors, dashboard gauges are presented using additional viewports, another viewport might present real-time positions of cars in the race circuit, and so on.

The *Print* Step

The *print* step maps x^n and y^n values of projected points in NDC box to a viewport. The z^n values are not used by the *print* step but are instead used to implement hidden surface removal and to generate shadows. Thus, the *print* step involves the proportional scaling of the 2D NDC image (ignoring z^n values) to fit the rectangular viewport and is called the ***viewport transformation***. Recall x^n and y^n values in NDC are in range $x^n \in [-1, 1]$ and $y^n \in [-1, 1]$. Consider an arbitrary viewport in the framebuffer with width W_{vp} pixels and height H_{vp} pixels and bottom-left corner with window coordinates (x^d_{bl}, y^d_{bl}). Given NDC coordinates of point $P^n(x^n, y^n)$, the viewport transformation determines corresponding viewport coordinates $P^d(x^d, y^d)$. Figure 6 illustrates the mapping problem.

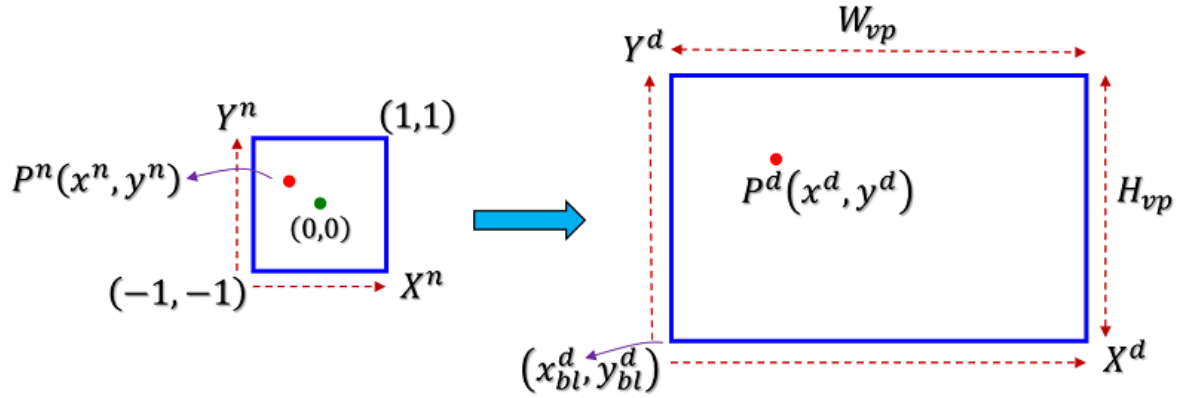

Figure 6: The viewport transform mapping NDC points to viewport.

To determine the viewport transformation matrix, the divide-and-conquer strategy is employed by dividing the 2D problem into two simpler 1D problems, one dealing with mapping x values and the other dealing with mapping y values. Begin by solving the problem of mapping input values $x^n \in [-1, 1]$ to output values $x^d \in [x^d_{bl}, x^d_{bl} + W_{vp}]$. From Figure 6, it is clear that output x^d values and input x^n values have a linear relationship which can be mathematically expressed as

$$x^d = ax^n + b \tag{1}$$

It is easiest to compute constants a and b in Equation (1) in terms of two example input values -1 and 1 and their corresponding output values x^d_{bl} and $x^d_{bl} + W_{vp}$, respectively. That is,

$$\begin{aligned} -1 &\longmapsto x^d_{bl} \\ 1 &\longmapsto x^d_{bl} + W_{vp} \end{aligned} \tag{2}$$

Substituting the two examples from Equation (2) into Equation (1) leads to the following equations

$$x^d_{bl} = -a + b \tag{3}$$
$$x^d_{bl} + W_{vp} = a + b \tag{4}$$

Adding Equations (3) and (4) leads to the computation of constant b:

$$\begin{aligned} 2x_{bl}^d + W_{vp} &= 2b \\ \implies b &= x_{bl}^d + \frac{W_{vp}}{2} \end{aligned} \tag{5}$$

Constant a is computed by substituting Equation (5) into Equation (3):

$$\begin{aligned} x_{bl}^d &= -a + x_{bl}^d + \frac{W_{vp}}{2} \\ \implies a &= \frac{W_{vp}}{2} \end{aligned} \tag{6}$$

Using Equations (5) and (6), the mapping of input values $x^n \in [-1, 1]$ to output values $x^d \in [x_{bl}^d, x_{bl}^d + W_{vp}]$ can be written as

$$x^d = \frac{W_{vp}}{2} x^n + (x_{bl}^d + \frac{W_{vp}}{2}) \tag{7}$$

This exemplary method is repeated to map input values $y^n \in [-1, 1]$ to output values $y^d \in [y_{bl}^d, y_{bl}^d + H_{vp}]$:

$$\begin{aligned} y^d &= cy^n &+ d \\ -1 &\mapsto y_{bl}^d \\ 1 &\mapsto y_{bl}^d &+ H_{vp} \\ y_{bl}^d &= -c &+ d \\ y_{bl}^d + H_{vp} &= c &+ d \\ \implies d &= y_{bl}^d &+ \frac{H_{vp}}{2} \\ \implies c &= \frac{H_{vp}}{2} \end{aligned} \tag{8}$$

Substituting c and d from Equation (8) into $y^d = cy^n + d$, the mapping of input values $y^n \in [-1, 1]$ to output values $y^d \in [y_{bl}^d, y_{bl}^d + H_{vp}]$:

$$y^d = \frac{H_{vp}}{2} y^n + (y_{bl}^d + \frac{H_{vp}}{2}) \tag{9}$$

Writing Equations (7) and (9) as linear functions of x^n and y^n:

$$\begin{aligned} x^d &= \frac{W_{vp}}{2} x^n &+ (0)y^n &+ (x_{bl}^d + \frac{W_{vp}}{2}) \\ y^d &= (0)x^n &+ (\frac{H_{vp}}{2})y^n &+ (y_{bl}^d + \frac{H_{vp}}{2}) \end{aligned} \tag{10}$$

Adding one final equation to the system of two linear equations in Equation (10):

$$\begin{aligned} x^d &= \frac{W_{vp}}{2} x^n &+ (0)y^n &+ (x_{bl}^d + \frac{W_{vp}}{2}) \\ y^d &= (0)x^n &+ (\frac{H_{vp}}{2})y^n &+ (y_{bl}^d + \frac{H_{vp}}{2}) \\ 1 &= (0)x^n &+ (0)y^n &+ (1) \end{aligned} \tag{11}$$

Equation (11) represents a system of three linear equations which can be expressed as a matrix equation of the form $\mathbf{M}_{viewport} P^n = P^d$ where $\mathbf{M}_{viewport}$ represents the 3×3 viewport transformation matrix, $P^n(x^n, y^n)$ represents a point in NDC box, while $P^d(x^d, y^d)$ represents a viewport point in window coordinates. From Equation (11), the viewport transformation matrix can be written as:

$$\mathrm{M}_{viewport} = \begin{bmatrix} \frac{W_{vp}}{2} & 0 & x_{bl}^{d} + \frac{W_{vp}}{2} \\ 0 & \frac{H_{vp}}{2} & y_{bl}^{d} + \frac{H_{vp}}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{12}$$

Viewport Transformation in OpenGL

As shown in Figure 1, the viewport transform is implemented by a fixed stage of the GPU. This means users must provide the width, height, and bottom-left corner of the viewport within the framebuffer to OpenGL using command `glViewport`. Consider an application initialized with a framebuffer having width and height of 2400 and 1350 pixels, respectively. To render to a viewport that covers the entire framebuffer, the call to `glViewport` is

```
glViewport(0, 0, 2400, 1350);
```

To render to a viewport that defines the window's top-right quarter, the call to `glViewport` is

```
glViewport(2400/2, 1350/2, 2400/2, 1350/2);
```