CS3388B: Lecture 4

January 19, 2023

4 Screen Space and 2D Transforms

Up to now, we have learned how about primitives and how to draw them with the "default" settings. In particular, we may recall that this involves a black background, white color for primitives, and a window-centred origin.

4.1 NDC and Screen Space

There many many different coordinate systems in a graphics library. It is important to remember where you are and how to go from one to the other. Up to now, we have have been specifying our vertex positions in **Normalized Device Coordinates** (NDC). This is, without any other modifications, the default coordinate space for specifying vertices and primitives in OpenGL.

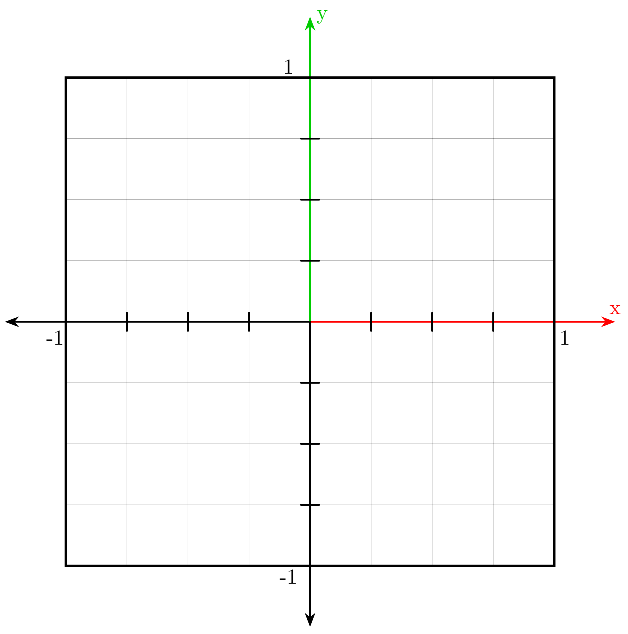


Figure 1: Normalized Device Coordinates

In NDC, the origin in at the middle of screen, positive x extends right, positive y extends up, and the space is *bounded* in the range $[-1, 1]$. Any vertex with coordinates outside this square (actually, a cube, but we'll come back to that once we talk about 3D graphics) will not be drawn.

But, now you ask: our screens and windows are not square(!), how do we translate from NDC to our window's coordinate system.

OpenGL does that for you. Well, GLFW does that for you. When you create a window, it has a certain width and height. This specifies the **viewport** for which the drawing occurs.

Recall that windowing systems use windows to act as independent **render targets**. Rather than a graphical program writing directly to the output device (e.g. the monitor), the graphical program writes to the render target specified by the windowing system. Typically this is a **frame buffer**.

The frame buffer is typically the same size as your window. It is the actual *canvas* on which the primitives are drawn. Remember raster displays and raster images? Here they are again. A frame buffer is an in-memory raster image which gets displayed within the window. Since the frame buffer is usually the same size as your window, the pixels in the frame buffer are drawn to the pixels of the window which are the subset of the output device's pixels.

Frame Buffer → window → (subset of) display

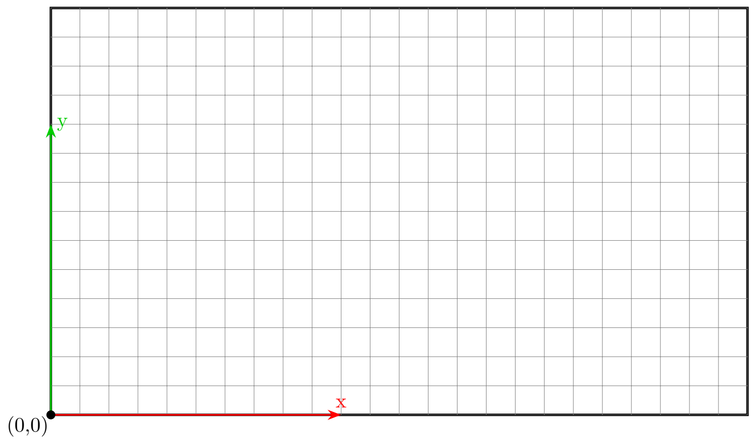


Figure 2: OpenGL Screen Space

Ultimately, a graphics library is about creating a raster image; a 2D grid of pixels. The window system handles the rest. In OpenGL, the coordinate system of the frame buffer, so-called **screen space**, has a bottom-left origin with positive x extending to the right and positive y extending up.

Okay, so back to the point. We specify vertex positions in NDC. How do they get translated to window/screen coordinates, and thus the frame buffer?

It begins by specify the *position*, *width*, and *height* of the frame buffer within its window. Typically the frame buffer and window are one-to-one. So, the position of the frame buffer is (0,0), and its width and height are window's width and height.

The **viewport matrix** then transforms NDC to screen space. It *stretches* the NDC to match the screen's aspect ratio, width and height. For now, I'm going to skip defining the matrix and just give you formulas.

Let a vertex be positioned in NDC at (x, y). If the viewport is at position (x_0, y_0) and the viewport has width w and height h, then its screen-space position is:

$$x_s = (x + 1)\left(\frac{w}{2}\right) + x_0$$
$$y_s = (y + 1)\left(\frac{h}{2}\right) + y_0$$

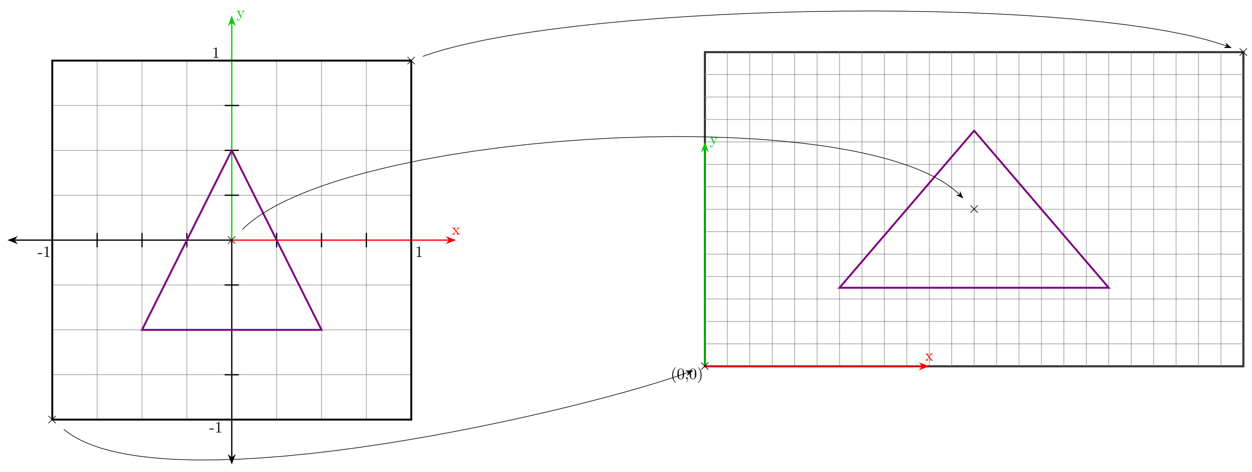


Figure 3: Converting from NDC to Screen Space

When you create a window with glfw, the viewport is defined for you. If you want yo change it, you use `glViewport`

```
glViewport(x0, y0, w, h);
```

If we want to move things around the scene, we don't move entire objects or primitives around a scene, we move *each vertex* which defines that primitive.

To move a vertex, we apply transforms so that a vertex's position in NDC is modified. We can do this manually, by specifying different values within `glVertex*`. We can also let OpenGL do this for us, automatically, using matrices.

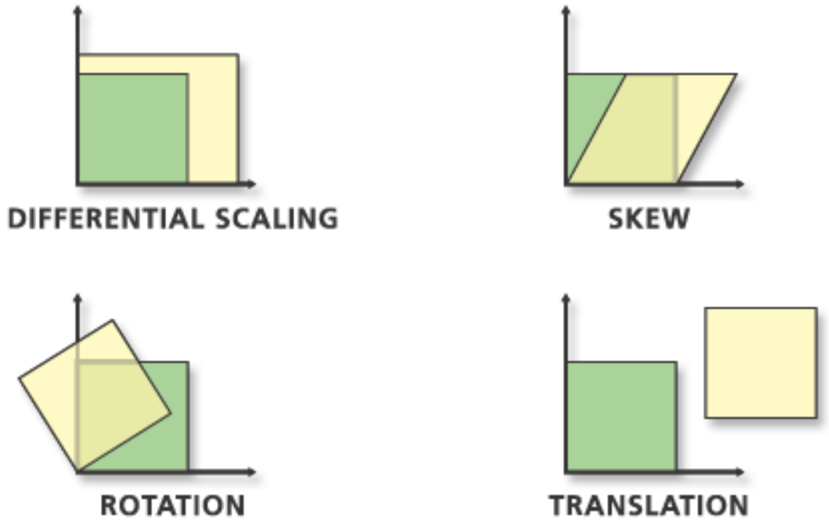
For now, let's consider the linear algebra behind these transforms and then see how to apply them in OpenGL.

4.2 Affine Transforms

An **affine transform** is any geometric manipulation or transformation that:

1. preserves lines (straight lines remain straight); and
2. preserves parallelism.

Some examples include scaling, rotation, reflection, translation, and shear/skew.



Any combination of these transforms can be applied and it still be, overall, an affine transform. We can translate, rotate, scale, translate again, rotate again, etc. and it still be an an affine transform.

We *almost* always apply affine transforms when we transform primitives. Only during *perspective projection* to be apply a non-affine transform. We'll revisit that later.

Let's see the basis of affine transforms using linear algebra. It's all about **matrix multiplication**.

In this section and the following subsections, we will assume a normal mathematical coordinate system. That is, right-handed, with positive x going right and positive y going up.

4.2.1 Scaling

Scaling is the easiest transform to understand. We simply *scale* or *multiply* a vertex's position.

If we have a triangle (0, 0), (0, 1), (1, 0) and we want to make it twice as big, we multiply the coordinates of each vertex by 2: (0, 0), (0, 2), (2, 0). Easy.

What if we want to scale a primitive's width differently than its height, we apply a different multiplier to the x coordinates (for width) and to the y coordinates (for height).

If we have a triangle (0, 0), (0, 1), (1, 0) and we want to make it twice as wide by three times as tall, we multiply the x coordinates by 2 and the y coordinates by 3: (0, 0), (0, 2), (3, 0). Easy.

Now, where is linear algebra? Consider that the position of each vertex can be modelled as a 2D vector (x, y). We can multiply any 2D vector by a 2x2 matrix to get another 2D vector. We will define a *scaling matrix* to do this. It's a simple diagonal matrix where the diagonal's first entry is the multiplier to apply to x and the diagonal's second entry is the multiplier to apply to y.

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x \cdot x \\ x_y \cdot y \end{bmatrix}$$

A nice thing about affine transforms is that they are **invertible**. How might we define the inverse of a scaling matrix? .....

Scale by the reciprocal!

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 \\ 0 & \frac{1}{s_y} \end{bmatrix}$$

4.2.2 Shear/Skew

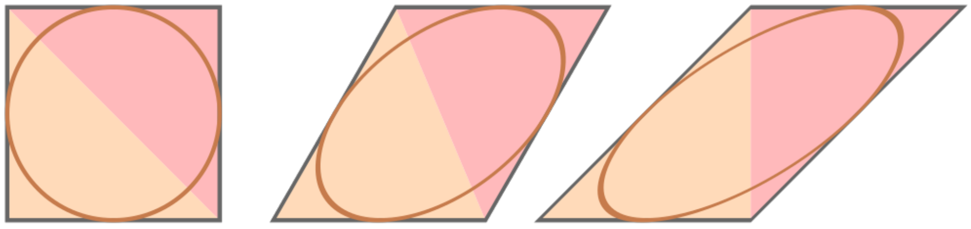


Figure 4: Horizontal Shear

A *shear* is a transformation that transforms a point an amount proportional to its distance from a particular fixed line. Typically that fixed line is the x-axis or the y-axis.

In *horizontal shear* the goal is transform the point (x, y) to the point $(x + my, y)$. This transforms vertical lines to be lines that have slope $\frac{1}{m}$.

$$\begin{bmatrix} 1 & m \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + my \\ y \end{bmatrix}$$

In *vertical shear* the roles of y and y are reversed. We transform the point (x, y) to the point $(x, y + mx)$. This transforms horizontal lines to be lines that have slope m.

$$\begin{bmatrix} 1 & 0 \\ m & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y + mx \end{bmatrix}$$

4.2.3 Rotation

To apply a rotation **around the origin**, we have a simple matrix given to us by **Givens rotations**.

A Givens rotation, and rotation matrices in general, are used to rotate a point *counter clockwise* about the origin. That is, the distance from the origin to the point is always maintained, and the point travels along the edge of a circle whose radius is the Euclidean distance of the point to the origin; see Figure 5

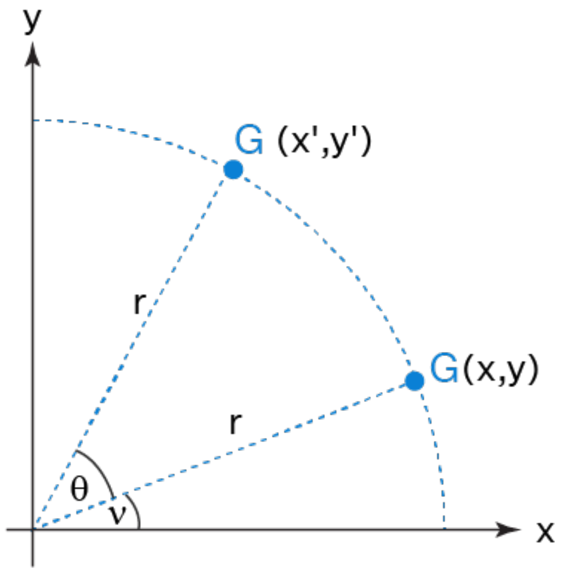


Figure 5: Rotation about the origin

So, how do we get there? Relatively simply. All we need is the angle θ that we want to rotate the point by. Then, the rotation matrix is defined as:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

For a point (x, y) this gives us:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ x\sin(\theta) + y\cos(\theta) \end{bmatrix}$$

To believe this works, draw the unit circle and triangle and work out the "SOH CAH TOA" yourself.

Naturally, the next question is ask, how do we get the inverse of a rotation? Well, it's the same as *rotating clockwise*. Or, we can think about rotating counterclockwise, as normal, but using a *negative angle*.

Now, recall that cos is an *even function* and sin is an *odd function*. Thus:

$$\begin{aligned} \cos(-\theta) &= \cos(\theta) \\ \sin(-\theta) &= -\sin(\theta) \end{aligned}$$

Therefore, this gives us:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}^{-1} = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

4.2.4 Reflection

Say we have a direction $\hat{v} = (v_x, v_y)$. How would we reflect something across a line in that direction? Without details, here is the matrix:

$$\begin{bmatrix} v_x^2 - v_y^2 & 2v_xv_y \\ 2v_xv_y & v_y^2 - v_x^2 \end{bmatrix}$$

And, hopefully obviously, the inverse of a reflection is just the same reflection!

4.2.5 Combined Transformations

There is something very very nice about modelling transformations as matrices. Consider if we wanted to rotate a triangle 45° counter-clockwise around the origin *and* scale its size up by 2.

With our affine transforms and matrices, we only need to multiply matrices to get the combined effect!

The correct scale and rotation matrices are then :

$$S = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} \frac{1}{2} & \frac{-1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Let a vertex of that triangle be (x, y). Then, just apply the transformations one after the other.

$$\begin{bmatrix} \frac{1}{2} & \frac{-1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \left(\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \right) = \begin{bmatrix} x - y \\ x + y \end{bmatrix}$$

or, because of associativity, we can multiply the matrices together first and then only transform the vertex using a single matrix.

$$\left(\begin{bmatrix} \frac{1}{2} & \frac{-1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \right) \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x - y \\ x + y \end{bmatrix}$$

4.2.6 Translation

After all this time, you may have been wondering, why have we not seen translation yet? It seems to be the most simple and obvious transformation. Take an object and move it a little bit, without actually changing any of its shape or orientation.

Consider some displacement vector $\vec{d}$ and some vertex as position $\vec{v} = (x, y)$. The displacement or *translation* of this point by $\vec{d}$ is simply $\vec{v} + \vec{d}$.

Now, I ask you. Can you give me a matrix when, when multiplied by $\vec{v}$ gives us $\vec{v} + \vec{d}$?

$\vdots$

$\vdots$

No 2x2 matrix exists to do this!

Enter *homogeneous coordinates.*

4.2.7 Homogeneous Coordinates

Homogeneous coordinates or, more generally *projective coordinates* can be used to represent every kind of affine transform as a matrix. Without getting into the mix of what is *projective geometry*, here is what we need to know.

We represent two-dimensional points as vectors with three coordinates. When the third coordinate of a vector in homogeneous coordinates is 1, this corresponds to the a normal 2-dimensional Euclidean point.

$$\begin{array}{ccc} \text{Euclidean} & \leftrightarrow & \textit{Homogeneous} \\ \begin{bmatrix} x \\ y \end{bmatrix} & \leftrightarrow & \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \end{array}$$

When the third coordinate of a vector in homogeneous coordinates is 0, this corresponds to a *directional vector*.

$$\begin{array}{cc} \text{A point} & \textit{A direction} \\ \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} & \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \end{array}$$

In homogeneous coordinates we represent two-dimensional transformations as 3x3 matrices.

4.2.8 Back to Translation

If we have a displacement vector $\vec{d} = (d_x, d_y)$. we can displace or translate a vertex at position $(x, y, 1)$ (in homogeneous coordinates) using the translation matrix:

$$T = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \\ 1 \end{bmatrix}$$

Now, recall that *directions*, rather than points, are encoded with their third coordinate equal to 0. What happens when we try to translate a direction $(x, y, 0)$?

$$\begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

It does not move! Recall that for any vector $\vec{v}$ we can "move it around the plane" and its direction does not change. Vectors only care about direction and length. Thus, translating a vector does not change its defining coordinates.

4.2.9 Affine Transforms in Homogeneous Coordinates

After all that work with 2x2 matrices, we now have to convert them all to be 3x3 matrices for homogeneous coordinates. Oh no!

Don't worry. It's simple. For all of the matrices defined in Sections 4.2.1-4.2.5, just make those 2x2 matrices the top-left submatrix of the 3x3 identity matrix.

Scale.

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \leftrightarrow \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Shear.

$$\begin{bmatrix} 1 & m \\ 0 & 1 \end{bmatrix} \leftrightarrow \begin{bmatrix} 1 & m & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ m & 1 \end{bmatrix} \leftrightarrow \begin{bmatrix} 1 & 0 & 0 \\ m & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation.

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \leftrightarrow \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

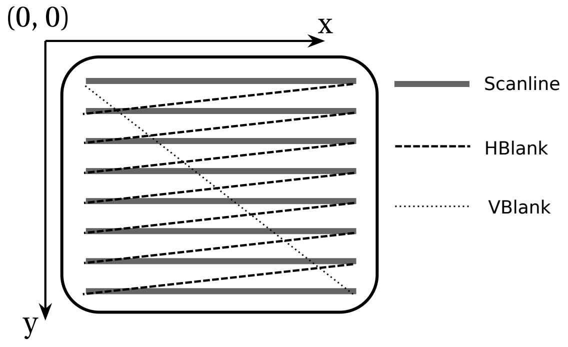
Translation.

$$T = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}$$

4.3 Affine Transforms in a Left-Handed Coordinate System

Everything we saw about transforms in the previous section applied in the normal Cartesian plane, with positive y going upwards.

But what happens if our coordinate system has a top-left origin with positive y going downwards? Shucks. It's actually *almost* the exact same.



Scale.

Scaling still has the **same matrix**. Nice! However, we must realize that scaling "up" (making things bigger) happens *in the direction of the positive y-axis*. So, objects will grow "downward" away from the origin if $s_y > 1$.

Rotation.

Rotation also has the **same matrix**! But, it's *interpretation* is different. Rather than rotating counter clockwise, the same rotation matrix will rotate an image *clockwise* about the origin. Why?

The rotation matrix we have seen actually doesn't know anything about clockwise or counter clockwise. It rotates *from* the positive x axis *toward* the positive y axis. The result in a left-handed system is to rotate clockwise.

If we want to have the same effect as a counter clockwise rotation in a right-handed system, simply use the rotation matrix for $-\theta$.

Shear.

Shear is, again, *almost* the same. Shearing now happens in the opposite directions as we would expect. (Just like rotations rotate in the opposite direction you expect).

In a left-handed system, the matrix

$$\begin{bmatrix} 1 & m \\ 0 & 1 \end{bmatrix}$$

would make the slope of vertical lines become $-m$. Similarly for vertical shear.

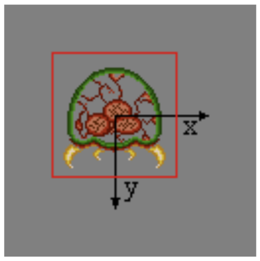
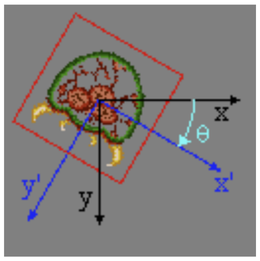
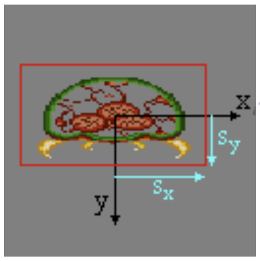
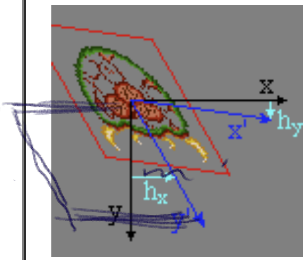
Identity	Rotation	Scaling	Shear
			
$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$	$\mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$	$\mathbf{H}(h_x, h_y) = \begin{bmatrix} 1 & h_x \\ h_y & 1 \end{bmatrix}$
$\mathbf{I}^{-1} = \mathbf{I}$	$\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$	$\mathbf{S}^{-1}(s_x, s_y) = \mathbf{S}(1/s_x, 1/s_y)$	$\mathbf{H}^{-1}(h_x, h_y) = \mathbf{H}(-h_x, -h_y) / (1-h_x h_y)$

Figure 6: Left-Hand affine transforms

4.3.1 Maps between Coordinate Systems

So what's actually happening when we apply a transformation? Figure 6 gives a small hint.

a) One way of thinking about transformations is that we transform a point into another

b) Another way of thinking about transformations is that we transform the coordinate system, and the positives of points doesn't change at all.

Let $\hat{i} = (1, 0)$, $\hat{j} = (0, 1)$ be the typical basis vectors of the 2 dimensional coordinate system and let a vertex be defined at $v = (x, y)$ in this system.

Let's think about what $v = (x, y)$ really means. It means, with respect to the basis vectors, the position of v is x multiples of $\hat{i}$ and y multiples of $\hat{j}$. This can be defined as a matrix-vector product, where B is the matrix whose columns are $\hat{i}$ and $\hat{j}$:

$$Bv = \begin{bmatrix} \hat{i} & \hat{j} \end{bmatrix} v = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

When we apply a transformation M, we are actually defining a new coordinate system with basis vectors $M\hat{i}$ and $M\hat{j}$. In this new coordinate system, the vertices don't "move", they are still defined as they were. For example, $v = (x, y)$ still, but the values x and y are now relative to $M\hat{i}$ and $M\hat{j}$.

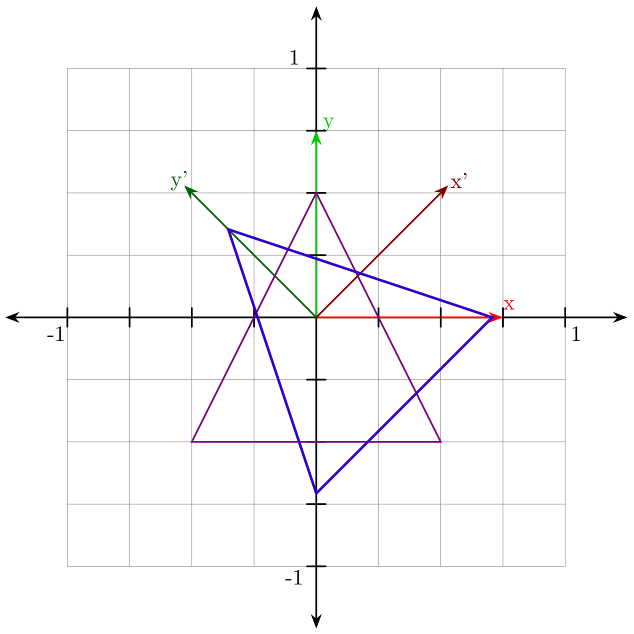


Figure 7:

In Figure 7, the coordinate system is rotated counter clockwise by 45°. The transformation matrix would be:

$$R = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

This makes the new basis of this coordinate space:

$$R\hat{\imath} = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right) \quad \text{and} \quad R\hat{\jmath} = \left(\frac{-\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$$

Then, we take the same $v = (x, y)$ and interpret its location with respect to these new basis vectors. Call B_2 the matrix of this basis.

$$B_2 v = \begin{bmatrix} R\hat{\imath} & R\hat{\jmath} \end{bmatrix} v = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2}x - \frac{\sqrt{2}}{2}y \\ \frac{\sqrt{2}}{2}x + \frac{\sqrt{2}}{2}y \end{bmatrix}$$

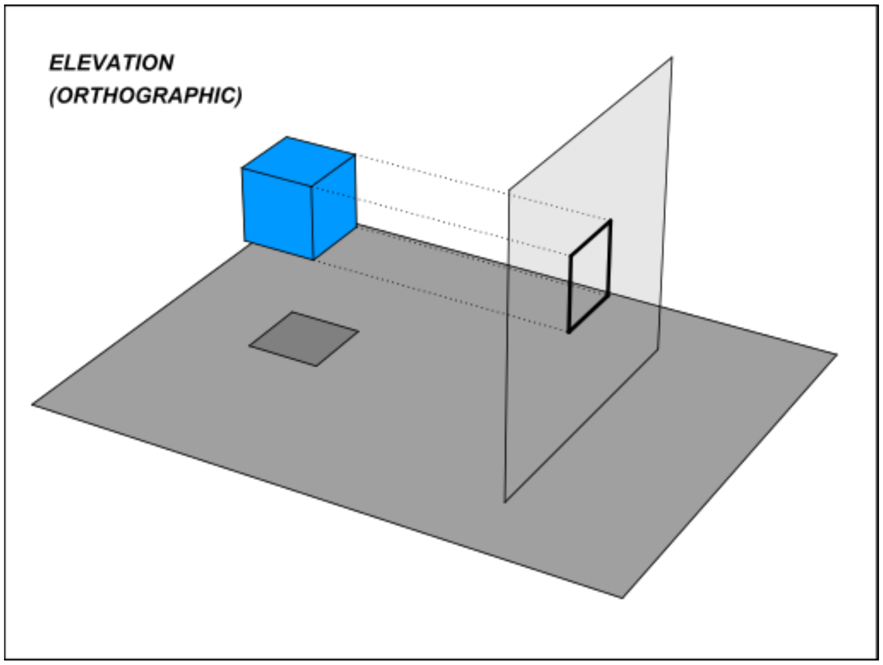
This way of thinking of transformations being applied to the coordinate system **will be very important**.

4.4 Orthographic Projection

A projection is a means of representing 3-dimensional objects as 2-dimensional objects (or higher dimensions down to lower ones). In three dimensions we will see *perspective projection.* For now, let us consider orthographic projections.

Orthographic projections do *not* represent the object as it would be recorded photographically or perceived by a viewer observing it in real life.

An orthographic projection is one where the "projection lines" are all parallel to each other and all perpendicular to the **image plane**.



I previously defined normalized device coordinates as having the origin in at the middle of screen, positive x extends right, positive y extends up, and the space is *bounded* in the range $[-1, 1]$. This was a lie. Normalized device coordinates is actually the *cube* bounded in the range $[-1, 1]$ in all three dimensions.

Up to now, we have simply been drawing on the $z = 0$ plane of this cube. As we have previously seen, OpenGL then uses the viewport to map normalized device coordinates to screen space. (This itself is also a lie, but its a useful lie for now that I won't explain until later.)

We use projections in OpenGL to define how to map the *global coordinate system* also called the **world coordinate system** to Normalized Device Coordinates. As we just saw in the previous section, we map between coordinate systems using matrices! In particular, the **projection matrix**.

Be default, the projection matrix is the identity. That's how we've been drawing "directly" in NDC. Our triangle with vertices $(0.5, 0.5)$, $(0.5, -0.25)$, $(-0.5, -0.25)$ wasn't being drawn in NDC, but rather being drawn in *world coordinates* and then mapped to NDC using the *identity matrix*.

But, NDC is not particularly useful. We're limited in the range of vertex positions $[-1, 1]$. What would be really nice is if we could draw in *almost* screen coordinates. Orthographic projection lets us do this. We will define a matrix which maps some coordinate system to NDC.

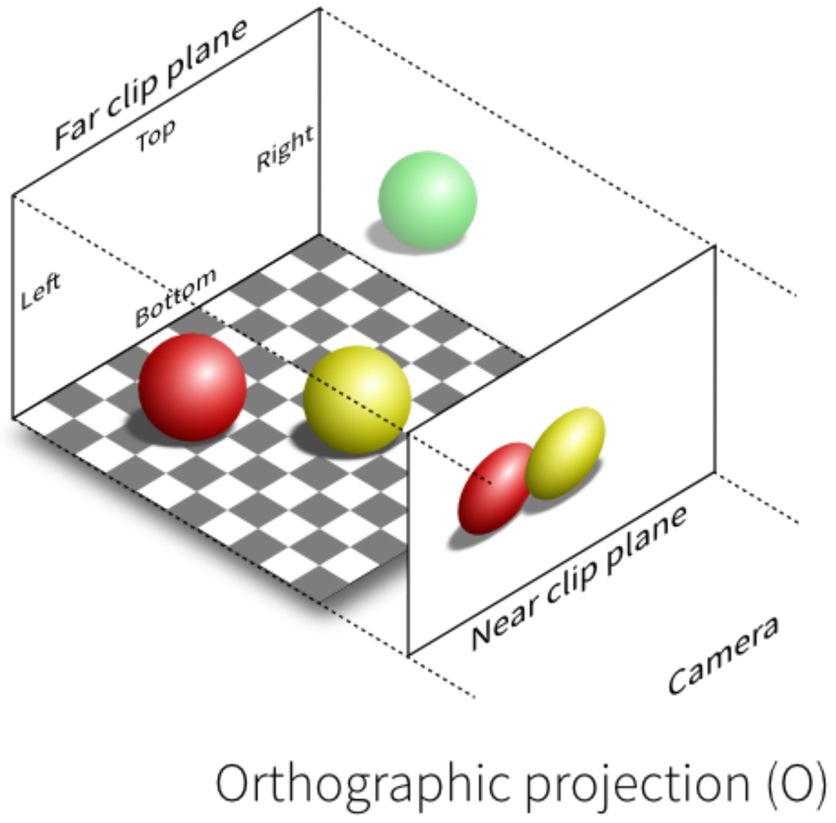


Figure 8:

We are interested in the defining a coordinate system for the **orthographic view volume**. Now, we will define a rectangular prism which is the orthographic view volume. We do this by specifying the coordinates of the left, right, top, bottom, front, and back planes of the rectangular prism. See Figure 8.

If we specify, say, left to be 0, right to be 1000, bottom to be 0, and top to be 500, then we get a coordinate system with a bottom-left origin and a width of 1000 and a height of 500.

1000 and 500 what, though? Units don't really matter. But, one common use case is set up an orthographic projection so that 1 unit in the orthographic view volume is one screen pixel. So, we typically set the view volume to be the dimensions of the window, with origin in the bottom-left, x extending right, and y extending up.

In OpenGL this is easy. We use the function `glOrtho`.

```
glOrtho(left, right, bottom, top, near, far)
```

This defines the coordinates of the six faces of the view volume. You don't even have to compute the matrix itself! Do you want to know how precisely to compute an orthographic projection? See http://learnwebgl.brown37.net/08_projections/projections_ortho.html for a nice interactive demo and some linear algebra.

Now, it's not actually *that* easy. Recall OpenGL has a global state machine. So, when we define a projection matrix it holds for every draw call until we change the projection matrix. To set the projection matrix we first have to set, in the state machine, which matrix we want to edit. (Yes, there are many different global matrices, we'll get back to that).

`glMatrixMode(GL_PROJECTION)` sets the current "editable" matrix to the projection matrix.

Now, a subtlety of the `glOrtho` function. It's documentation says:

> `glOrtho` describes a transformation (matrix) that produces a parallel projection. The current matrix (see `glMatrixMode`) is multiplied by this matrix and the result replaces the current matrix.

So, to set up our view volume we have to proceed in three steps:

```
glMatrixMode(GL_PROJECTION)
glLoadIdentity()
glOrtho(left, right, top, bottom, near, var)
```

`glLoadIdentity` replaces the current matrix with the identity matrix.

Now that we have applied an orthographic projection, we can draw in world coordinates to our heart's content!

In particular, if we only want to do 2D graphics (as we do in this current moment of the semester), we need not care about the near and far values. Be default, set them to -1 and 1. Then, we can draw 2D objects with `glVertex2f` and thus an implied $z = 0$.

The follow code draws a triangle at coordinates (200, 200), (200, 300), (300, 200) in a viewing area with bottom-left origin and 640 width and 500 height.

```c
#include <GLFW/glfw3.h>

int main(void)
{
    //... glfw init

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(640, 500, "Hello World", NULL, NULL);

    //... glfw stuff

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 640., 0, 500., -1, 1);

    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(window))
    {
        /* Poll for and process events */
        glfwPollEvents();

        /* Render here */
        glClear(GL_COLOR_BUFFER_BIT);

        glBegin(GL_TRIANGLES);
        glVertex2f(200, 200);
        glVertex2f(300, 200);
        glVertex2f(200, 300);
        glEnd();


        /* Swap front and back buffers */
        glfwSwapBuffers(window);

    }

    glfwTerminate();
    return 0;
}
```