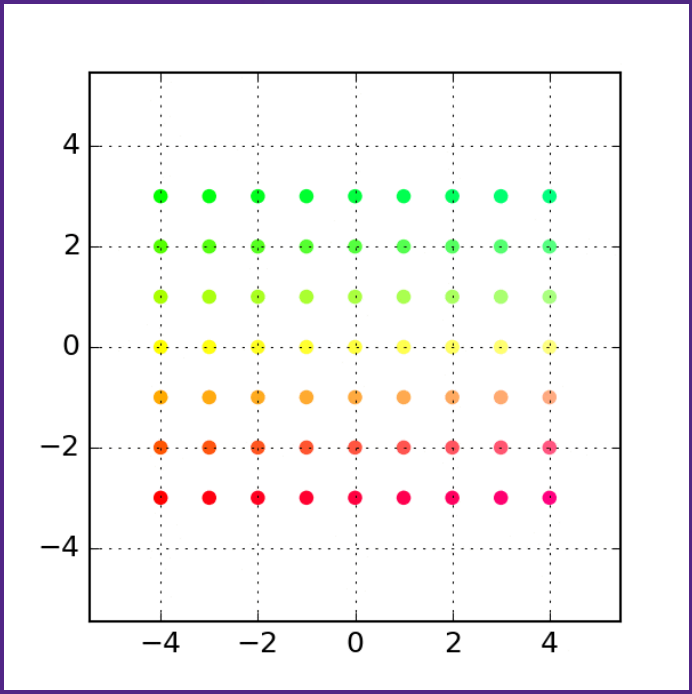
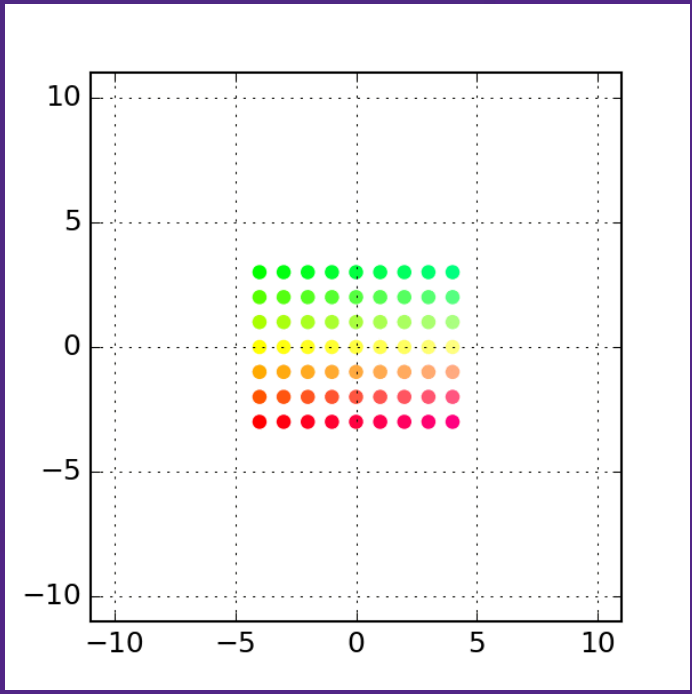
Computer Graphics I:
2D Transformations





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.

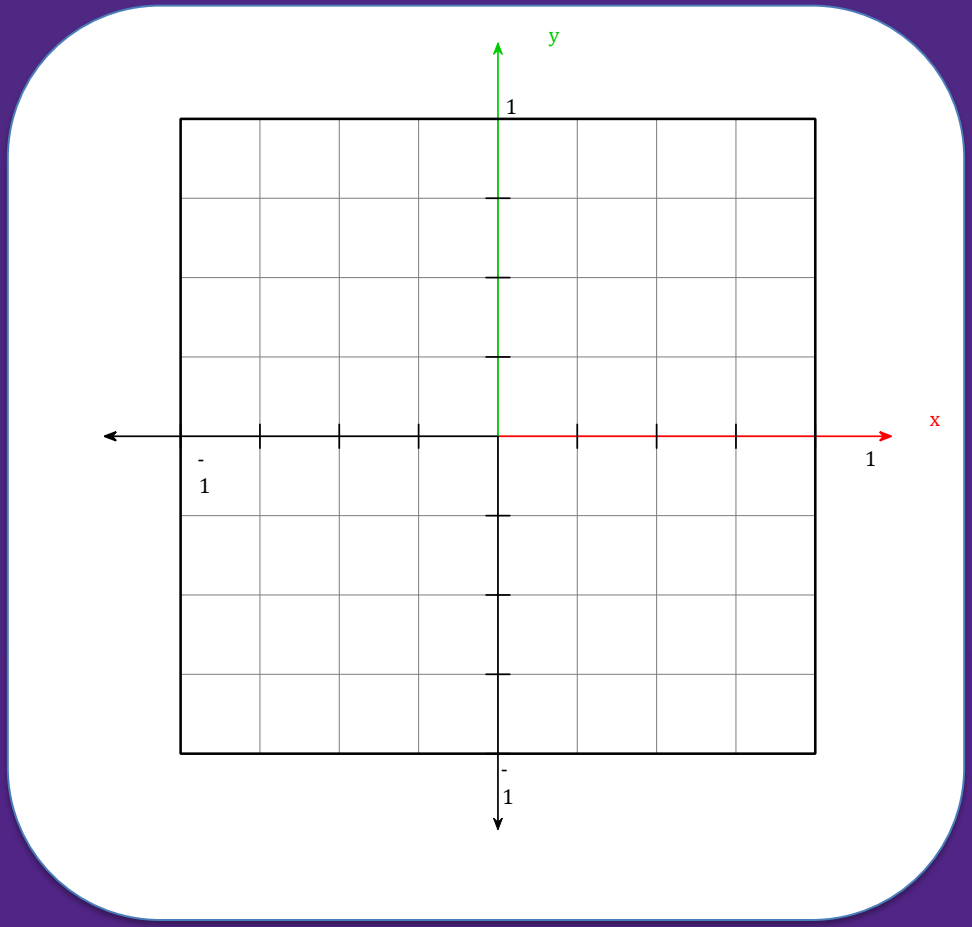
- **NDC and Screen Space**

- There are 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 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.

Screen Space and 2D Transforms



Screen Space and 2D Transforms

- 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.

Screen Space and 2D Transforms

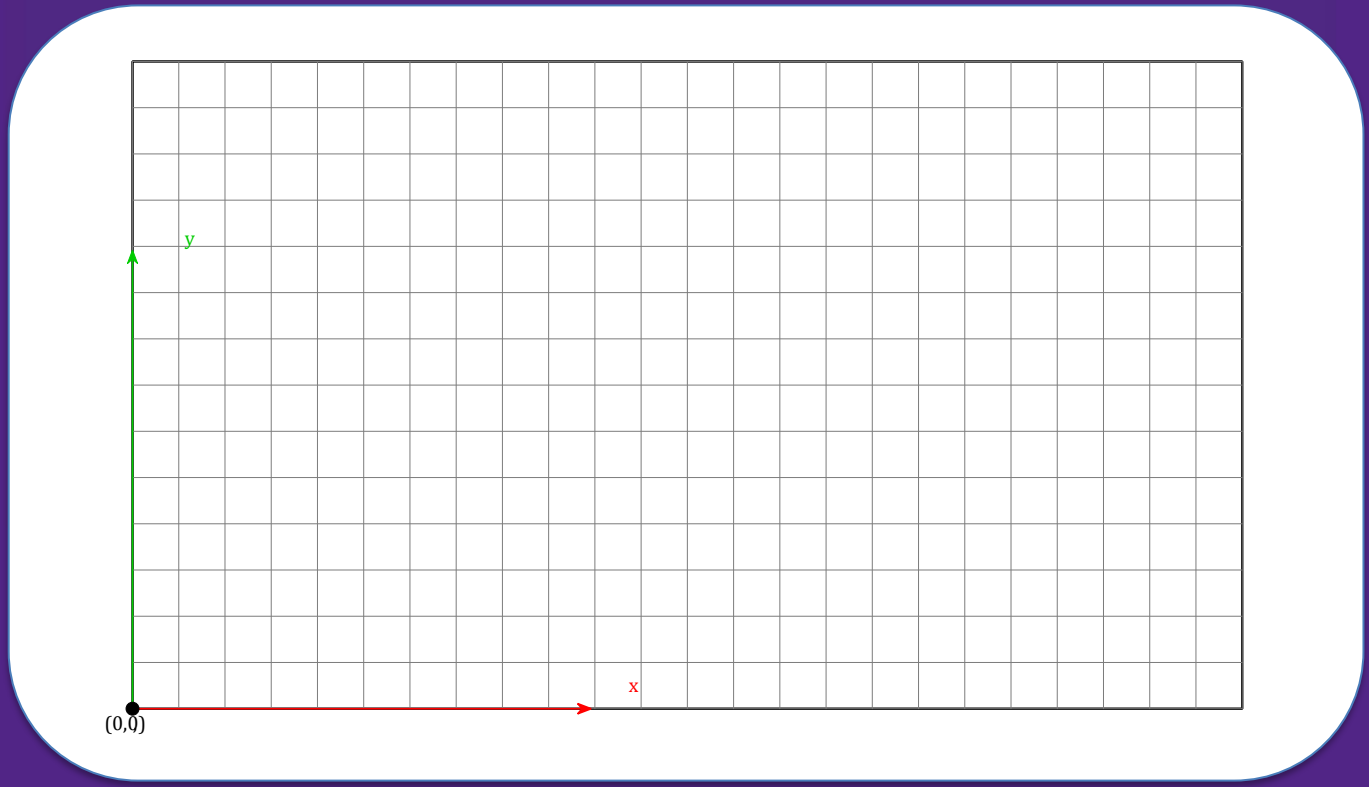
- OpenGL does that for you. Well, GLFW does that for you.
- When you create a window, it has a certain width and height. This specifies the viewport for which the drawing occurs.
- Recall that windowing systems use windows to act as independent render targets.
- Rather than a graphical program writing 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.

Screen Space and 2D Transforms

- 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.

Screen Space and 2D Transforms

Frame Buffer → window → (subset of) display



Screen Space and 2D Transforms

- 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.

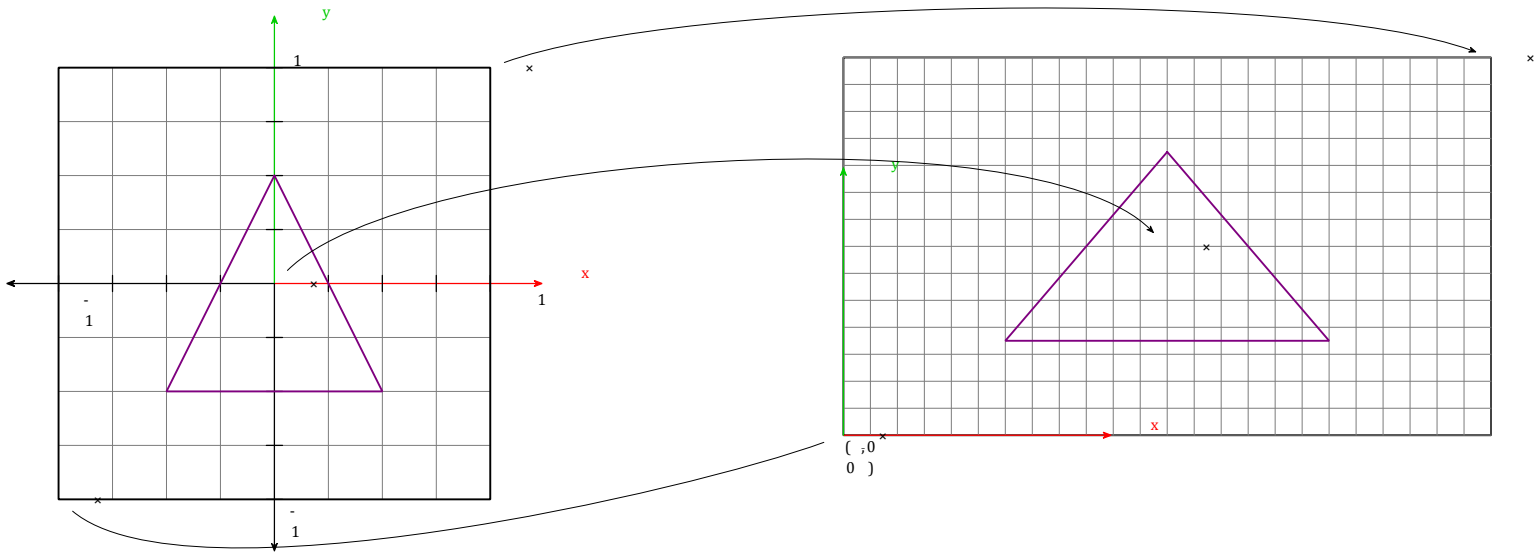
Screen Space and 2D Transforms

- 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 (x0, y0) and the viewport has width w and height h, then its screen-space position is:

$$x_s = (x+1)(w/2)+x_0$$

$$y_s = (y+1)(h/2)+y_0$$

Screen Space and 2D Transforms



Screen Space and 2D Transforms

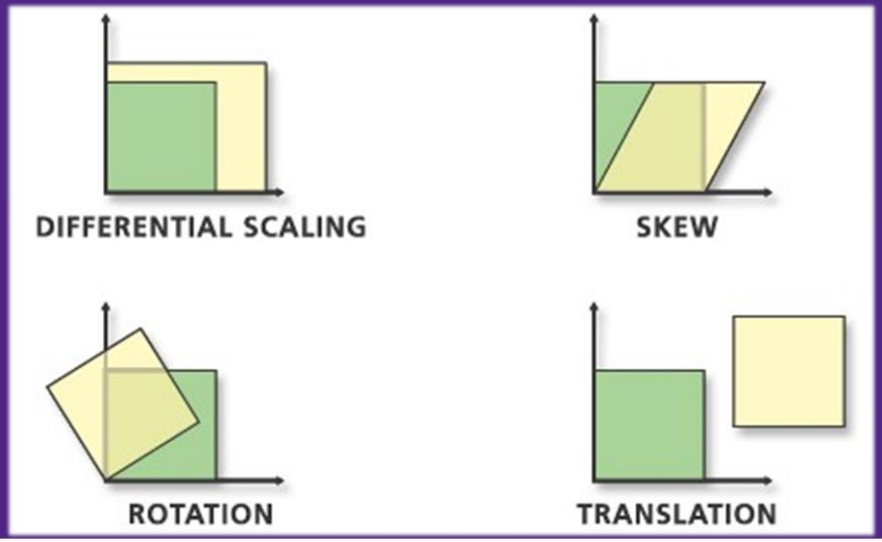
- When you create a window with glfw, the viewport is defined for you. If you want to change it, you use glViewport
  - *glViewport(x0, y0, w, h);*
- If we want to move things around the scene, we don't move entire objects or primitives around a scene, we move each vertex which defines that primitive.
- 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.

Two Dimensional Affine Transformations

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.



Two Dimensional Affine Transformations

- 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.

Two Dimensional Affine Transformations

- Affine transformations of the plane in two dimensions include pure translations, scaling in a given direction, rotation, and shear.

- An affine transformation is usually and conveniently represented in matrix notation:

$$\begin{bmatrix} y \\ \hline 1 \end{bmatrix} = \begin{bmatrix} [A] & \vec{b} \\ \hline \overrightarrow{0^T} & 1 \end{bmatrix} \begin{bmatrix} x \\ \hline 1 \end{bmatrix}$$

- using homogenous coordinates. The advantage of homogenous coordinates is that one can combine any number of affine transformations into one by multiplying the respective matrices.

- This property is used extensively in computer graphics, computer vision & robotics.

Definitions

- Affine transformation:
  - From the latin ‘affinis’, which means connected with, an affine transformation is a geometric transformation that preserves lines and parallelism but not necessarily distances or angles.

- Homogenous coordinates:
  - Typically, we are thinking in terms of Euclidean geometry, i.e., coordinates in 3d (or 2d) space: (X,Y,Z).
  - However, sometimes we want to use ‘projective geometry’ (geometric properties that are invariant with respect to projective transformations).
  - This adds in a 4^{th} dimension, W, in addition to X,Y,Z. W is the projective space, and coordinates in this projective space are called ‘homogenous coordinates’.

Linear Two-Dimensional Transformations

- Let's examine 2D transformations without the notion of homogeneous coordinates first.
- Consider the point $p = (x,y)^T$ as the one to be transformed
- Translating a point (moving it somewhere else) is accomplished with a translation and is effected in the following manner:

$$p + \vec{\Delta}p = p'$$

Linear Two-Dimensional Transformations

- This can also be represented as:

$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} x + \Delta x \\ y + \Delta y \end{bmatrix}$$

- Scaling is an affine transformation and can be described as an operation that performs a scalar multiplication on each component of a point: Sp = p’ , which can be represented as:

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

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).
- What if we want to scale a primitive's width differently than its height?

Scaling

- 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.

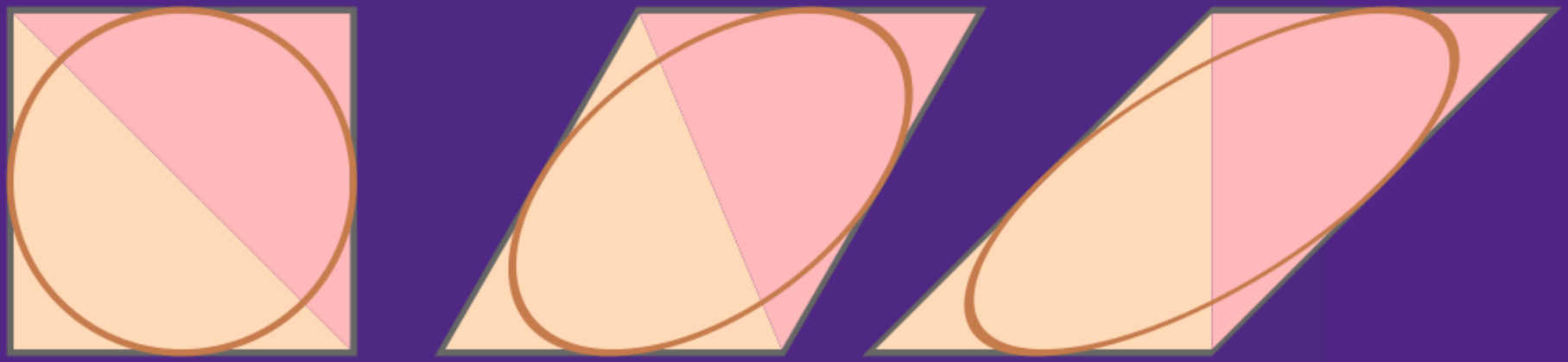
Scaling

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

- 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!

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

Shear/Skew



- 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.

Shear/Skew

- 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 1/m

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

Shear/Skew

- 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.

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

Rotation

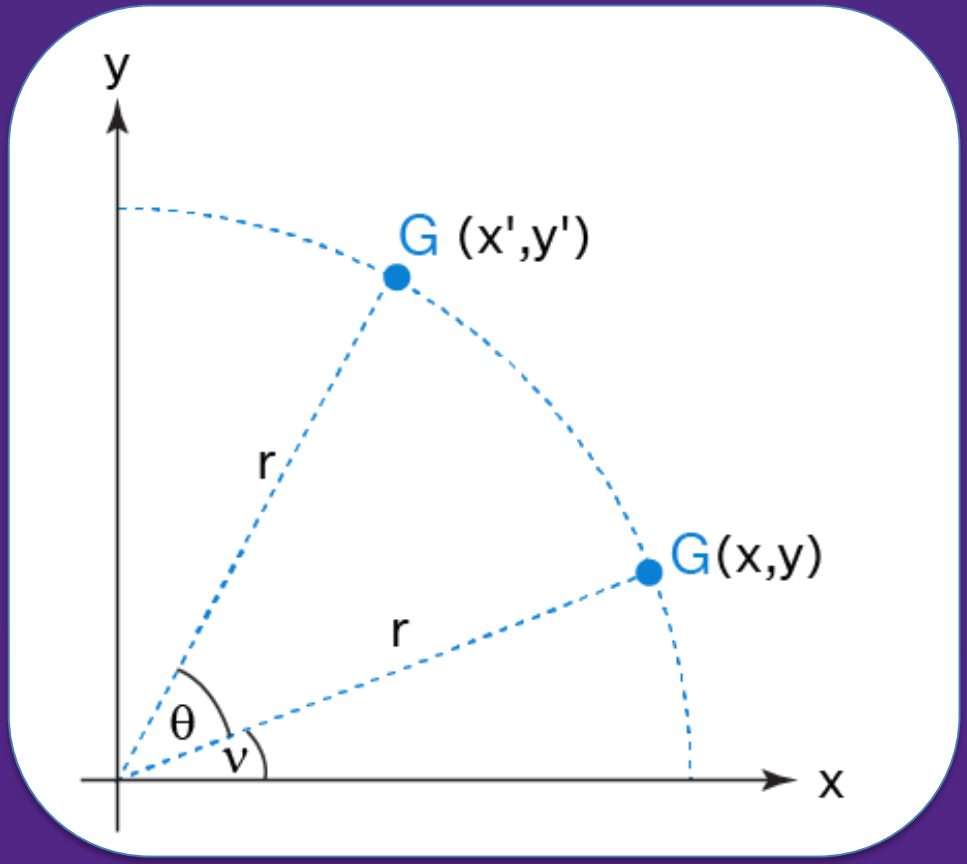
- Another common transformation is to rotate a point. To rotate a point about an origin, we use a rotation matrix, Rp = p’
- which can be represented as:

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} xcos\theta - ysin\theta \\ xsin\theta + ycos\theta \end{bmatrix}$$

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

Rotation



Rotation

- 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:

$$\cos(-\theta) = \cos(\theta)$$
$$\sin(-\theta) = -\sin(\theta)$$

Rotation

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

Reflection

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

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

- The inverse of a reflection is just the same reflection.

Reflections

- To reflect a vector about a line that goes through the origin, let

$$\vec{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

- be a vector in the direction of the line. The matrix performing this reflection is then:

$$M = \frac{1}{\|\vec{v}\|} \begin{bmatrix} v_x^2 - v_y^2 & 2v_x v_y \\ 2v_x v_y & v_y^2 - v_x^2 \end{bmatrix}$$

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!

Combined Transformations

- The correct scale and rotation matrices are then:

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

- For a vertex of the triangle (x,y), applying the transformations one after the other…

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

Combined Transformations

- Or, because of associativity, we can multiply the matrices together first and then only transform the vertex with a single matrix:

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

Translations

- 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.

Translations

- 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 $\vec{v} + \vec{d}$?

  …

  …

- No 2x2 matrix exists to do this!
- Enter homogeneous coordinates.

Homogenous 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.

Homogenous Coordinates

- When the third coordinate of a vector in homogeneous coordinates is 1, this corresponds to the a normal 2dimensional Euclidean point.

$$\text{Euclidean} \leftrightarrow \text{Homogeneous}$$

$$[x,y] \leftrightarrow [x,y,1]$$

Homogenous Coordinates

- Homogenous coordinates allow us to represent all these transformations with matrices that can be multiplied together.
- In this way, we can compose transformations in the order we choose to manipulate objects composed of 2D points.
- Remember that we have an additional ‘projection’ dimension.

Homogenous Coordinates - Translation

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

Homogenous Coordinates - Scaling

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ 1 \end{bmatrix}$$

Homogenous Coordinates - Rotation

$$\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} xcos\theta - ysin\theta \\ xsin\theta + ycos\theta \\ 1 \end{bmatrix}$$

Homogenous Coordinates

- In general, since matrix multiplication is not commutative (can't change the order of multiplication and get the same result, $AB \neq BA$), we have to pay attention to the order in which we apply transformations

- As an example, consider rotating the point p around another point, say, c, instead of the origin. To do this correctly, we first translate the point to the origin, perform the rotation, and then translate the point back:

$$T(\vec{c})R(\theta)T(-\vec{c})p$$

- Changing the order of the transformations yields bad results.

Notable 2D Transformations

- Matrices can also be used to perform reflections about an axis, and also shears (distortions in shape). To perform a reflection about the x axis, we apply the following matrix:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notable 2D Transformations

- To perform a reflection about the y axis, we apply the following matrix:

$$R_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notable 2D Transformations

- Interestingly, the matrix below is a reflection about the x=y axis

$$R_{xy} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notable 2D Transformations

- We can also apply transformations that distort points (shears).
- The matrices for an x-axis and y-axis shear are:

$$S_x = \begin{bmatrix} 1 & s_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, S_y = \begin{bmatrix} 1 & 0 & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Finding the Matrix for a Transformation

- Consider the unit vectors $\hat{e}_1(1,0)^T$ and $\hat{e}_2(0,1)^T$, which go along the axes of the coordinate system.
  (We will be using as vectors for the rest of the example, i.e., $\vec{e}_1$)

- If we know where the transformation must send these two vectors, then we can find the corresponding transformation matrix.

- Suppose we want to reflect an object about the vertical axis. We know that the transformation has the form:

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Finding the Matrix for a Transformation

- So if we want to transform $\vec{e}_1(1,0)^T$ and $\vec{e}_2(0,1)^T$:

$$M\overrightarrow{e_1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a \\ c \end{bmatrix}$$

$$M\overrightarrow{e_2} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$$

Finding the Matrix for a Transformation

- Now, to reflect the object around the vertical axis, we observe that $M\vec{e}_1 = -\vec{e}_1$, and thus we can form the following:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

- Directly yielding a = -1 and c = 0. We can perform the same with $M\vec{e}_2 = \vec{e}_2$ to find b=0 and d=1. The transformation matrix is thus:

$$M = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

Finding the Matrix for a Transformation

- Given independent unit vectors $\hat{u}_1$ and $\hat{u}_2$, we can find the transformation that sends them onto vectors $\vec{v}_1$ and $\vec{v}_2$. Let:

$$M = [\overrightarrow{u_1} \quad \overrightarrow{u_2}]$$

- Be the matrix that sends $\vec{e}_1$ and $\vec{e}_2$ onto $\vec{u}_1$ and $\vec{u}_2$.
- M^{-1} is the matrix that sends $\vec{u}_1$ and $\vec{u}_2$ onto $\vec{e}_1$ and $\vec{e}_2$. Let:

$$N = [\overrightarrow{v_1} \quad \overrightarrow{v_2}]$$

- be the transformation that sends $\vec{e}_1$ and $\vec{e}_2$ onto $\vec{v}_1$ and $\vec{v}_2$.
- We can see from here the transformation matrix is given by NM^{-1}

Transformations and Coordinate System

- Suppose two coordinate systems share an origin, but are oriented differently.

- How can we transform points from one system to the other? Let:

$$\overrightarrow{e_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad \overrightarrow{e_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and

$$\overrightarrow{u_1} = \begin{bmatrix} \frac{3}{5} \\ \frac{4}{5} \end{bmatrix}, \qquad \overrightarrow{u_2} = \begin{bmatrix} \frac{-4}{5} \\ \frac{3}{5} \end{bmatrix}$$

form the orthogonal unit vectors representing two coord. systems

Transformations and Coordinate System

- Let

$$\vec{v} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

- be a vector expressed in the coordinate system we described. Then:

$$\begin{bmatrix} \overrightarrow{u_1^T} \\ \overrightarrow{u_2^T} \end{bmatrix}$$

- Transforms the point $\vec{v}$ into the ($\vec{u}_1$, $\vec{u}_2$) coordinate system.

Transformations and Coordinate System

- Expressible as:

$$\begin{bmatrix} \frac{3}{5} & \frac{4}{5} \\ \frac{-4}{5} & \frac{3}{5} \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \end{bmatrix}$$

- Note that this transformation matrix is a rotation, and in general the inverse of a rotation matrix is its transpose, $M^{-1}=M^{T}$

Orthogonal Projections

- To project a vector orthogonally onto a line that goes through the origin, let

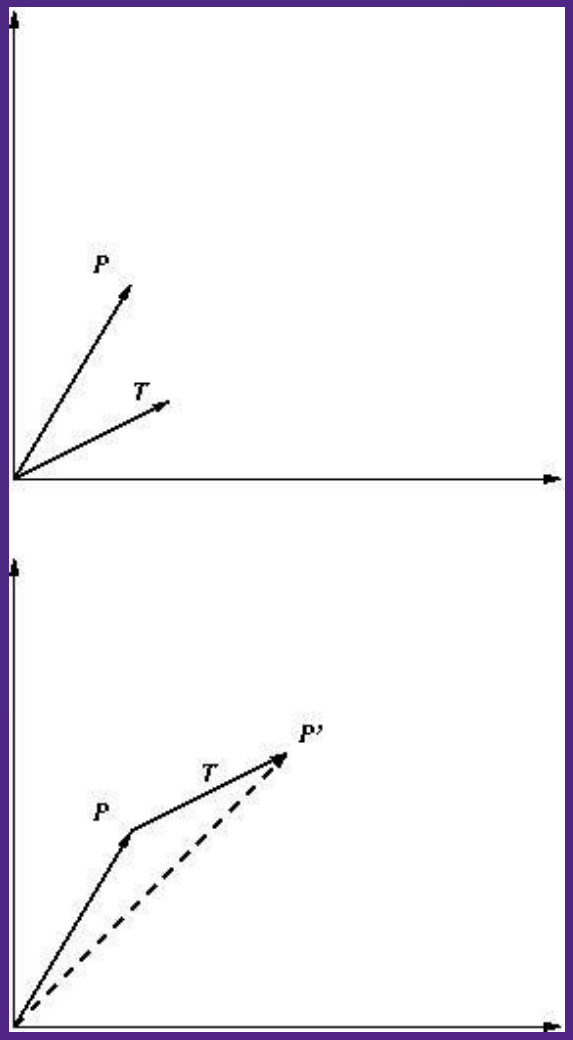
$$\vec{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

- be a vector in the direction of the line and form the following transformation matrix:

$$M = \frac{1}{\|\vec{v}\|} \begin{bmatrix} v_x^2 & v_x v_y \\ v_x v_y & v_y^2 \end{bmatrix}$$

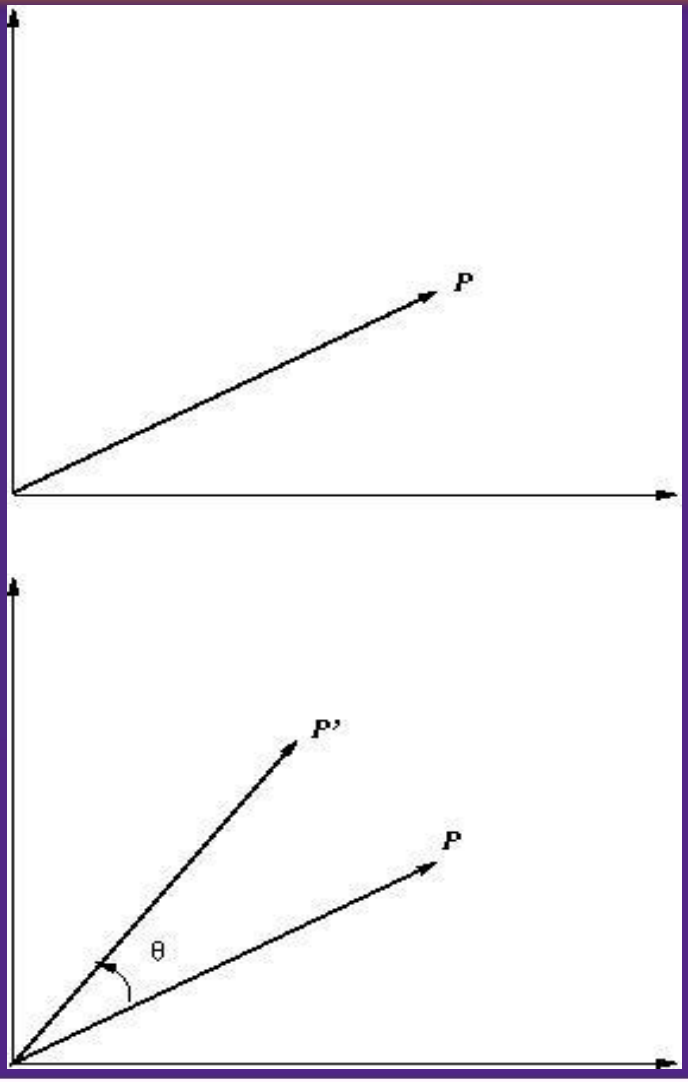
Illustrations

- Illustration 1:
- Translating a point by $\vec{T}$



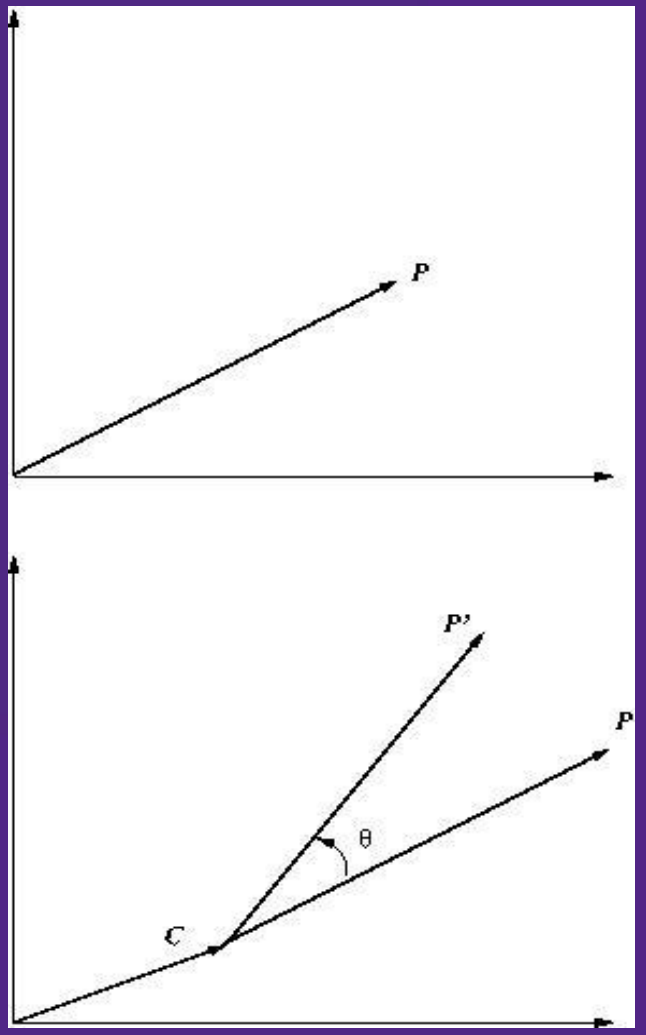
Illustrations

- Illustration 2:
- Rotating a point about the origin by θ



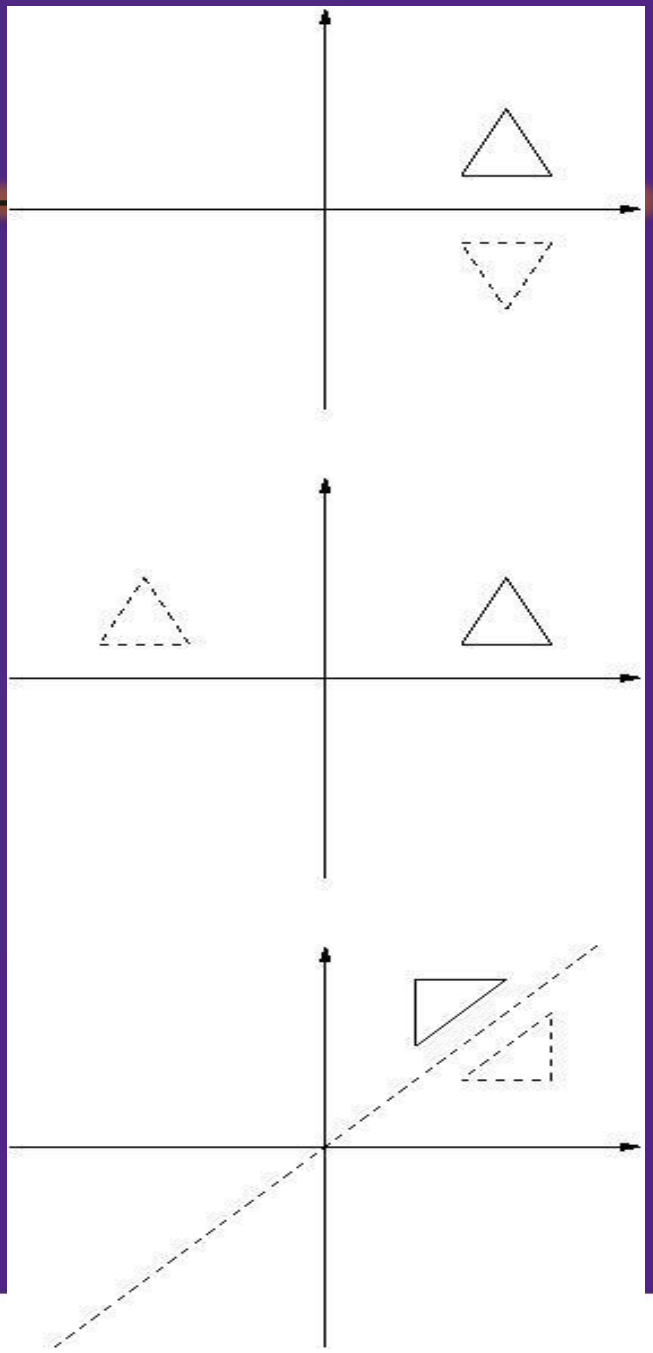
Illustrations

- Illustration 3:
- Rotating a point about $\vec{C}$ by θ



Illustrations

- Illustration 4:
- Reflection transformations



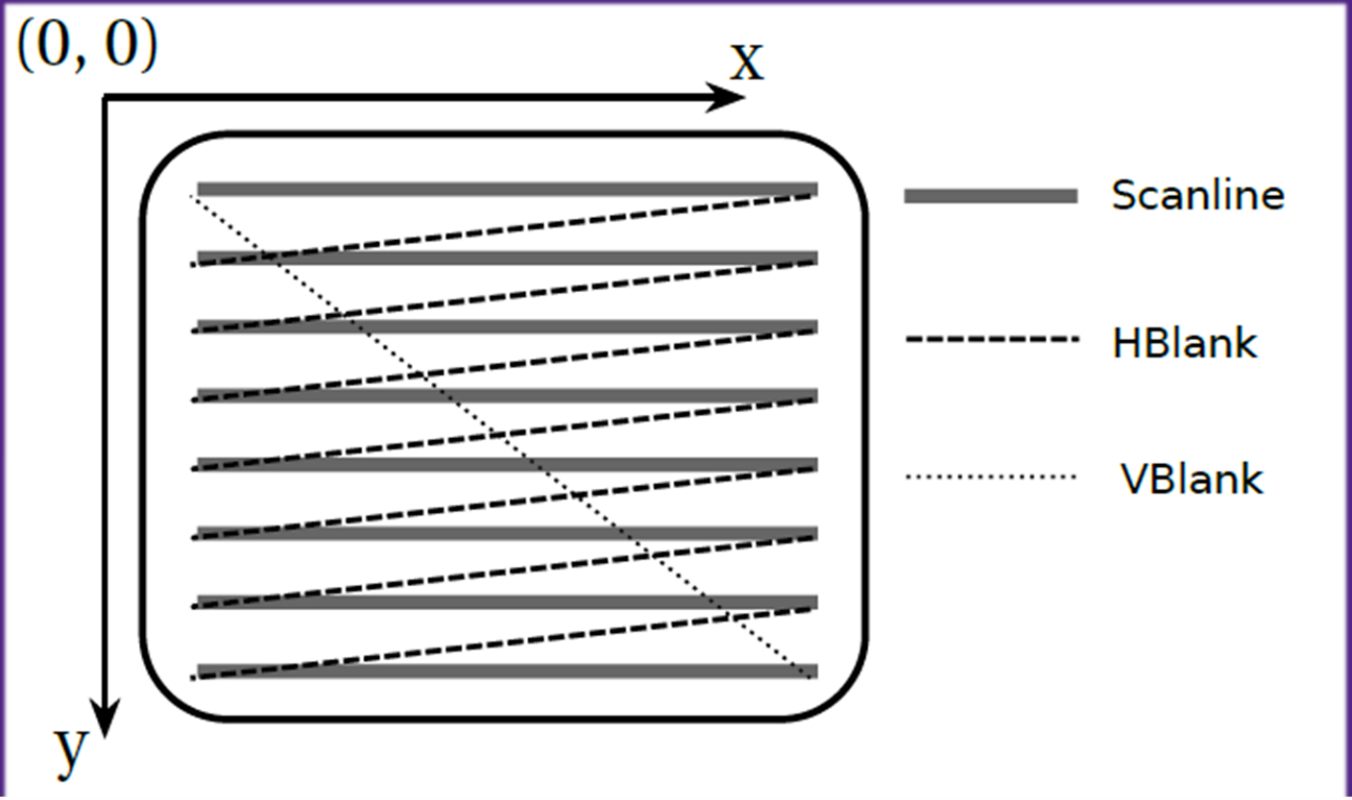
2D Transformation Hierarchy

- The following table depicts the hierarchy of 2D transformations.
- Each transform also preserves the properties listed in the rows below it.
- For instance, the similarity transformation preserves not only angles but also parallelism and straight lines

2D Transform	Degrees of Freedom	Preserves
Translation	2	Orientation
Rigid Body	3	Lengths
Similarity	4	Angles
Affine	6	Parallelism
Projective	8	Straight Lines

Affine Transformations 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.
- What if our coordinate system has a top-left origin with positive y going downwards? It's actually almost the same.



Affine Transformations in a Left-handed Coordinate System

- 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 sy > 1.
- 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?

Affine Transformations in a Left-handed Coordinate System (Rotation)

- 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 lefthanded 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 $-\theta$.

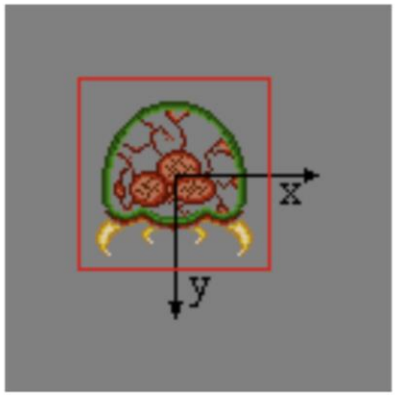
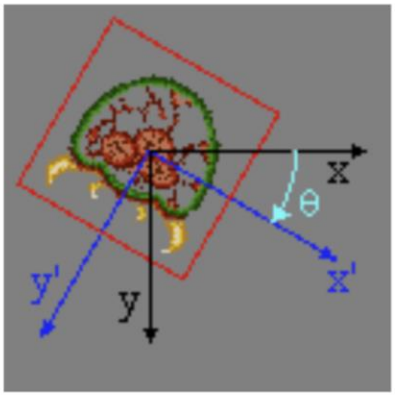
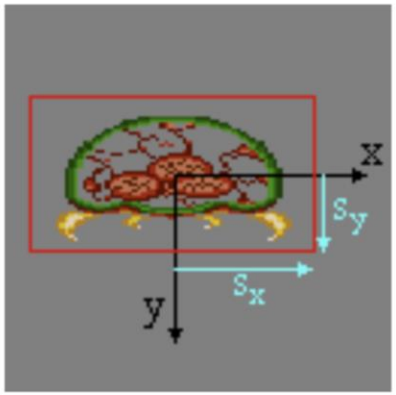
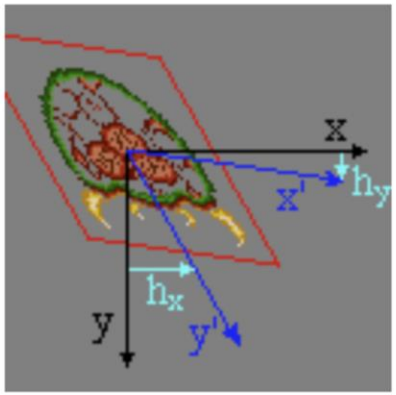
Affine Transformations in a Left-handed Coordinate System (Shear)

- 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.

Affine Transformations in a Left-handed Coordinate System (Shear)

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

Maps Between Coordinate Systems

- So what's actually happening when we apply a transformation? The previous image 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.
- c) We somehow make an image on a piece of film float and slide it from side to side.

- 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.

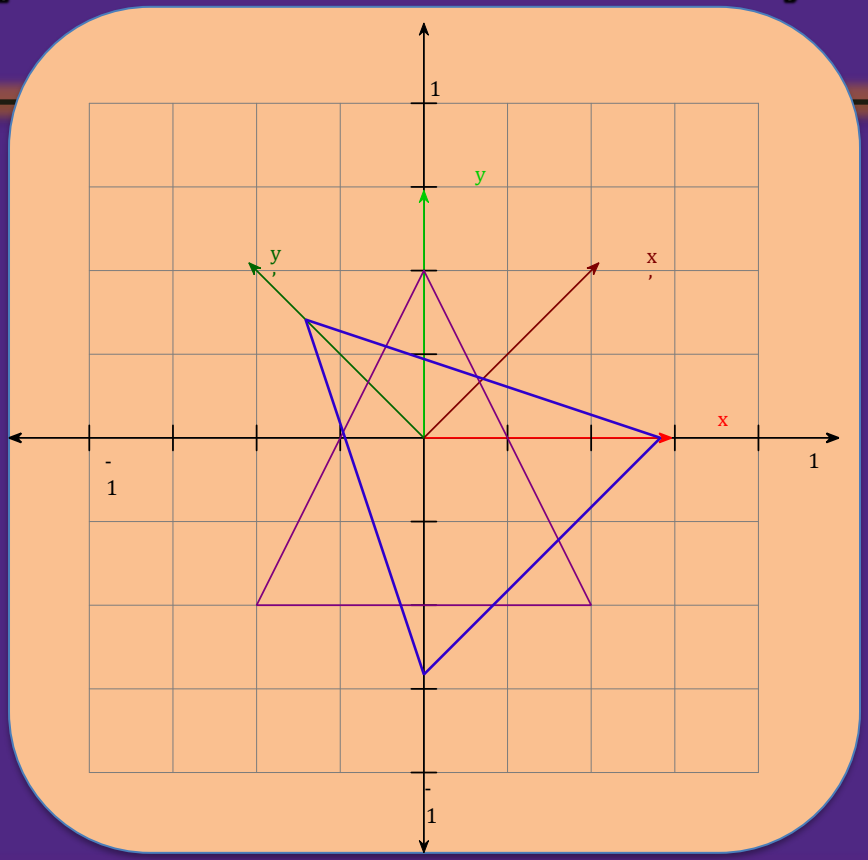
Maps Between Coordinate Systems

- 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 iˆ and y multiples of jˆ. This can be defined as a matrix-vector product, where B is the matrix whose columns are iˆand jˆ:

$$Bv = [\hat{\imath}\ \hat{\jmath}]v = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

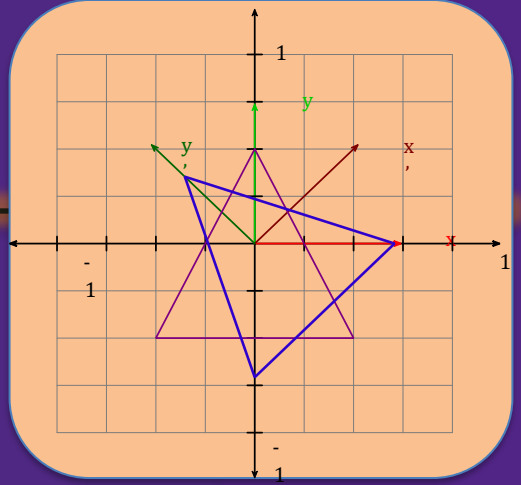
- When we apply a transformation M, we are actually defining a new coordinate system with basis vectors Miˆ and Mjˆ.
- In this new coordinate system, the vertices don't "move", they are still defined as they were.
- For example, v = (x, y) still, but the values x and y are now relative to Miˆand Mjˆ.

Maps Between Coordinate Systems



- In Figure 7, the coordinate system is rotated counter clockwise by 45°.

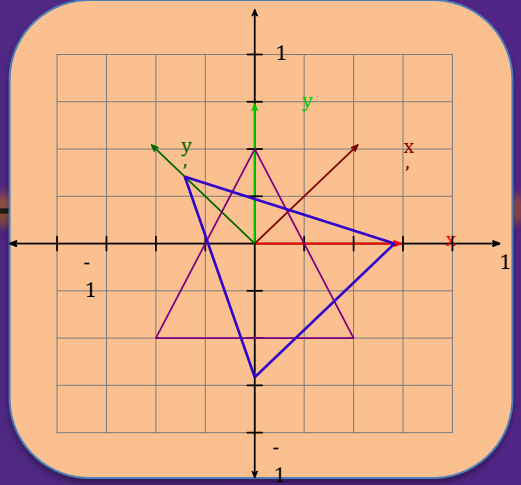
Maps Between Coordinate Systems



- In Figure 7, the coordinate system is rotated counter clockwise by 45° -

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

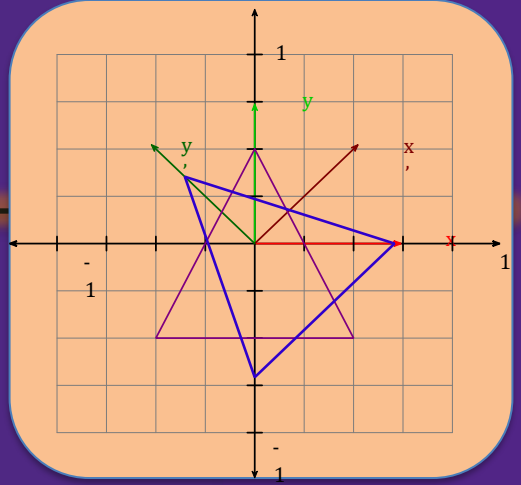
Maps Between Coordinate Systems



- This makes the new basis of this coordinate space:

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

Maps Between Coordinate Systems



- 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

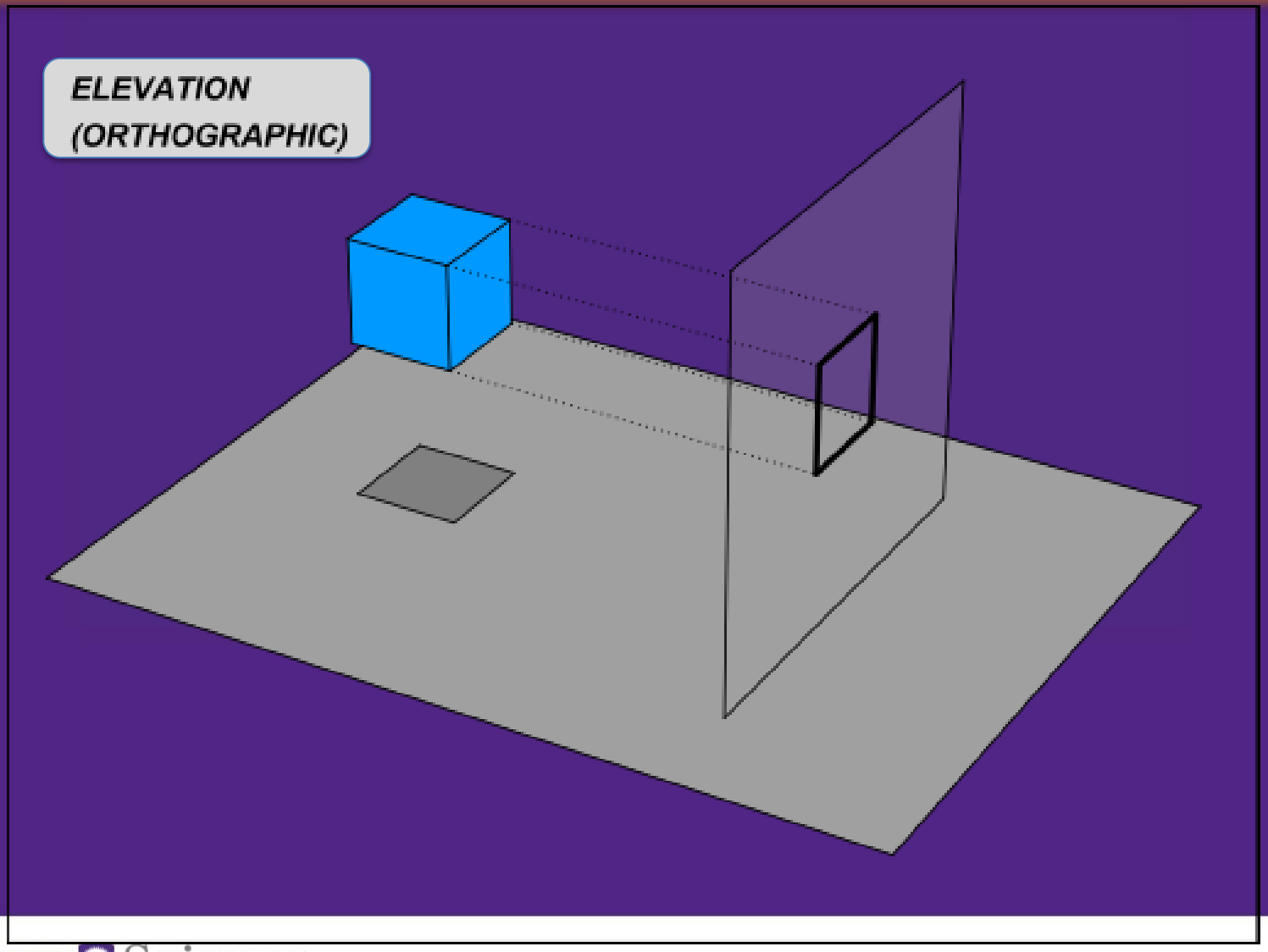
$$Bv = [R\hat{\imath}\ \ R\hat{\jmath}]v = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2}x - \frac{\sqrt{2}}{2}y \\ \frac{\sqrt{2}}{2}x + \frac{\sqrt{2}}{2}y \end{bmatrix}$$

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

Orthographic Projections

- 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 don’t represent the object as it would be recorded photographically or perceived by a viewer observing it
- An orthographic projection is one where the "projection lines" are all parallel to each other and all perpendicular to the image plane.

Orthographic Projections



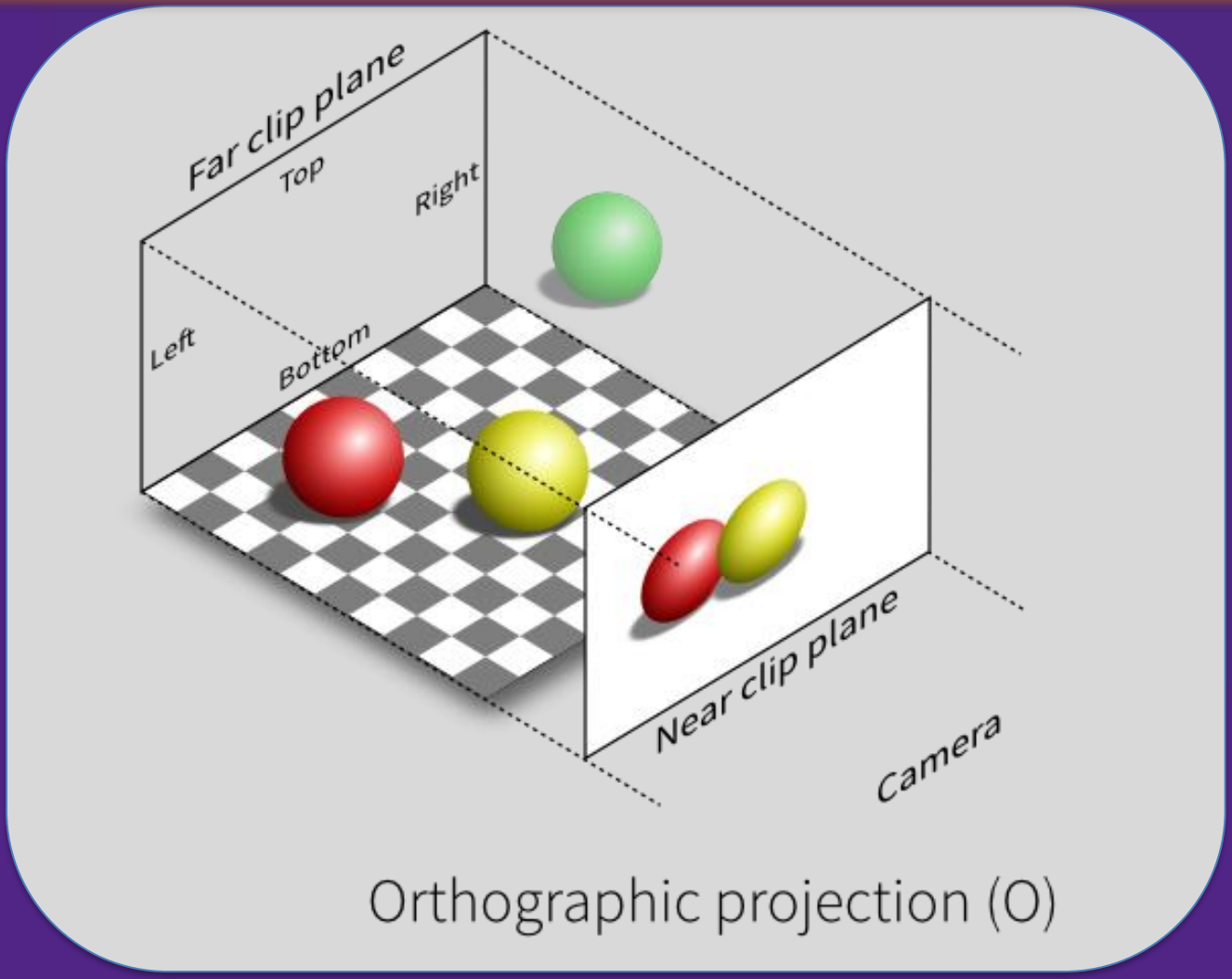
Orthographic Projections

- We 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 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.)

Orthographic Projections

- 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.

Orthographic Projections



Orthographic Projections

- 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.

Orthographic Projections

- 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 bottomleft, x extending right, and y extending up.

Orthographic Projections

- 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

Orthographic Projections

- 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.

Orthographic Projections

- 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)
```

Orthographic Projections

- 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;
}
```