# Exercises

- 4.1 Show that the following sequences commute: a. a rotation and a uniform scaling b. two rotations about the same axis c. two translations
    - Solution (*this is a, b, and c in order*)

4.1 If the scaling matrix is uniform then

$$\mathbf{RS} = \mathbf{RS}(\alpha, \alpha, \alpha) = \alpha\mathbf{R} = \mathbf{SR}$$

Consider $\mathbf{R}_x(\theta)$, if we multiply and use the standard trigonometric identities for the sine and cosine of the sum of two angles, we find

$$\mathbf{R}_x(\theta)\mathbf{R}_x(\phi) = \mathbf{R}_x(\theta + \phi)$$

By simply multiplying the matrices we find

$$\mathbf{T}(x_1, y_1, z_1)\mathbf{T}(x_2, y_2, z_2) = \mathbf{T}(x_1 + x_2, y_1 + y_2, z_1 + z_2)$$

- If we are interested in only two-dimensional graphics, we can use three dimensional homogeneous coordinates by representing a point as p = [x y 1]T and a vector as v = [a b 0]T. Find the 3 × 3 rotation, translation, scaling, and shear matrices. How many degrees of freedom are there in an affine transformation for transforming two-dimensional points?
    - Solution (first is the elementary affine transformations in 2D using 3D, 3x3 matrices. Then we talk about the general form which highlights how many Degrees of Freedom these four transformations give you.)

4.4 Translation

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Shear

$$\mathbf{H} = \begin{bmatrix} 1 & \mathrm{a} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

**The last here is a shear in x along the y axis (makes the x values a function of x and y).**

The general form of a homogeneous coordinate transformation matrix for working with two dimensional graphics is

$$\mathbf{M} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

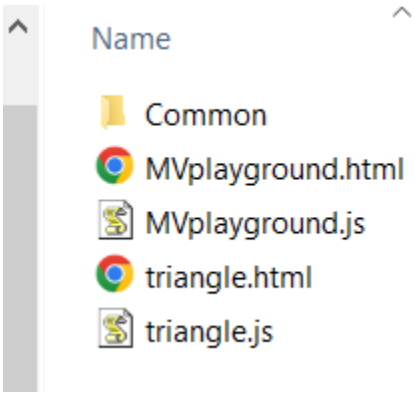
**In the end, we get six degrees of freedom (a,b,c,d,e,f here).**

- How must we change the rotation matrices if we are working in a left-handed system and we retain our definition of a positive rotation?
    - The signs on the sine terms in the rotation matrices must all be changed. You can check this result by noting that a positive 90-degree rotation about z in a

right–handed system brings the positive x axis to the positive z axis, while in a left–handed system a positive 90 degree rotation about the z axis brings the positive y axis to the positive x axis. Similar results hold for rotations about the x and y axes.

# Spinning Up The Lab Code

- Setup the code (extract and ensure the structure is correct)




- First, we are going to go through adding more data to the vertices of our triangle example from the previous lab. This data will be colour data. It needs to be created and passed in from the Javascript just like the vertex position data was. But now the vertex shader needs to hand it off to the fragment shader and the fragment shader needs to pass out the colour. The cool thing is, inside the triangle, the rasterizer handles the interpolation of colours for us. In fact, it's not just the colour, the positions are also interpolated, that's how the pipeline knew where to put the red pixels in the last lab when we just used a simple pass-through shader. In fact, we can interpolate any vertex data and use it in the interior of triangles (that is, you can store other stuff at vertices).

# Draw a triangle….with colours!

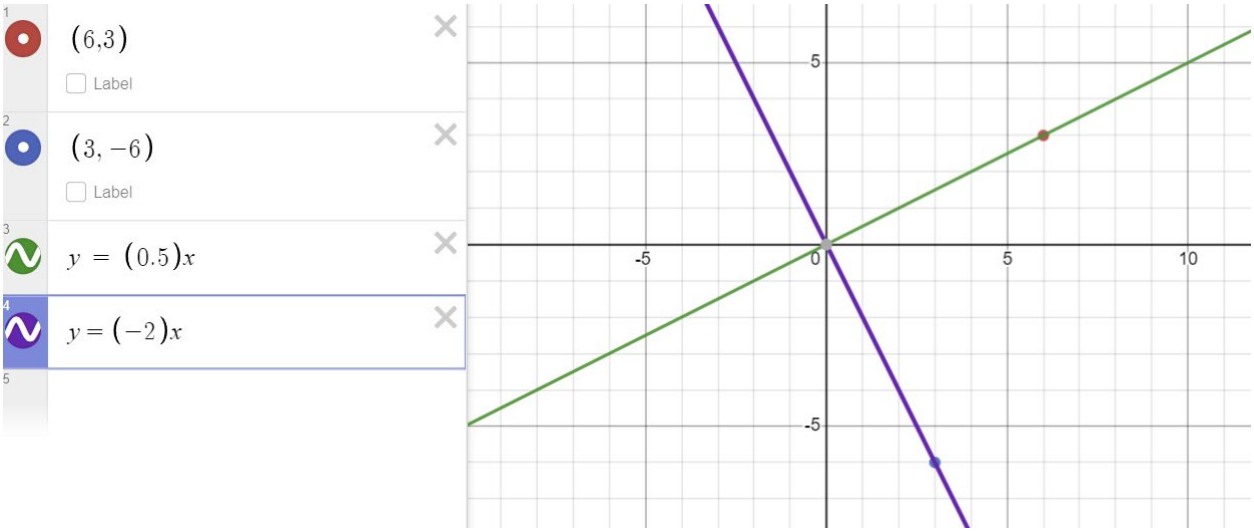
1. Go over the code for the triangle to show how this all works.
2. Explain each part with reference to the process above.
    a. Lines 10-13 in triangle.js find the window and canvas to draw on
    b. Line 37 creates the viewport in the html5 canvas (same as last lab)
    c. Line 38 sets the clear colour of the viewport (same as Lab 1)
    d. Lines 21-25 in triangle.js create the thing/geometry (vertices) to draw.
    e. Lines 27-32 in triangle.js create the data (colours) to use when drawing.

f. Lines 42-43 initialize (load and compile) our shaders, note “vertex-shader” and “fragment-shader” are the ids of the script tags containing the text for our shaders in the triangle.html file.
g. Lines 47-49 create a buffer, bind that buffer, and load the point/vertex position data into it (same as the last lab)
h. Lines 53-55 associate that position data with a particular variable “aPosition” in our shaders while telling us the size and type of that value
i. Lines 59-61 do the same as (g) above but now for the colour data buffer
j. Lines 63-65 do the same as (h) above but now for the colour data for the shader variable “aColour”
k. Move to the triangle.html file and explain the basics of the shaders.
    i. The Shader programs are in HTML, see the <script> tags and their types!
    ii. See lines 10 and 13, Note the vertex shader now has an **in** variable with the corresponding name from the JavaScript “aColour” that we setup in (j) above and an **out** variable in line 13. In coming from the data steam off the CPU side and out going from the vertex shader program to the fragment shader program (technically it goes through a number of processes in between during rasterization).
    iii. This colour data is passed to the fragment shader (technically rasterization, then fragment shader) in line 20
    iv. Now something invisible and amazing happened in between, the data was interpolated (using a special interpolation function that preserves perspective, this is later in the course). This is the “rasterization” step noted above.
l. Now we need to draw everything (same as last lab)

# Mess around with MV.js functions

1. There are a lot of functions in this file. The point of this exercise is to mess around with those that relate to vectors, matrices, and transformations. We talked about the math behind these in class to a certain extent. The files for this Lab are MVplayground.js and html. The JS file is where we play with transformations, printing the results out to the console.
2. My suggestion is not to dump this file on the screen and just say “here it is”, but rather to have a separate empty version open where you add one chunk at a time and look at the additional difference in console output.
3. I’ve already commented MVplayground.js to explain each chunk. You should make sure you also understand.
4. Note you need to open the browser console, so in chrome go to more tools -> developer tools then click console. *Alternatively, I think F12 opens the console directly.*
5. Note that very likely the output will include extremely small values like something to the power of -50+.. This is just a zero with some floating point error. Or sometimes a value like 0.99999999999999999, this is just a 1 with floating point error. In fact, floating point error ends up being a real problem in graphics and animation.

6. Here's a visual for line 99. Note the reason I drew the lines here through the transformed points was to visualize that the one has been rotated 90 degrees to the other. Basically, it shows the note I detail on line 104 in the code.




# Things to Try Now

- Try changing the colours rendered by changing the colour data in the JavaScript (note it is RGB values)
- Try composing more transformations and checking your understanding with the console log.
- Try other transformations in MV.js
- Try to recreate equivalent rotation matrices using the axis-specific rotation functions and the angle axis rotation function.