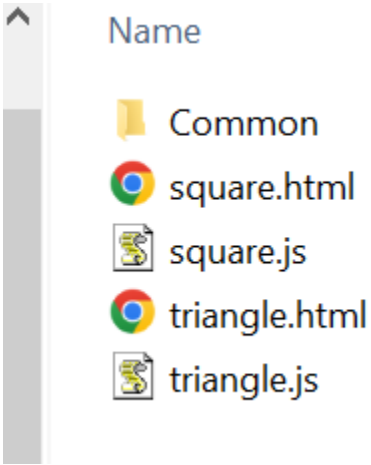
Exercises

- 1.1 The pipeline approach to image generation is nonphysical. What are the main advantages and disadvantages of such a nonphysical approach?
    - The main advantage of the pipeline is that each primitive can be processed independently. Not only does this architecture lead to fast performance, it reduces memory requirements because we need not keep all objects available. The main disadvantage is that we cannot handle most global effects such as shadows, reflections, and blending in a physically correct manner.
- 1.4 Consider the clipping of a line segment in two dimensions against a rectangular clipping window. Show that you require only the endpoints of the line segment to determine whether the line segment is not clipped, is partially visible, or is clipped out completely.
    - Suppose that the line segment is between the points (x1, y1) and (x2, y2) We can use the endpoints of the line segment to determine the slope and y-intercept of a line of which the segment is part, i.e. Note that we can deal with horizontal and vertical line segments as special cases. We can find the intersections with the sides of the window by substituting y = ymax, y = ymin, x = xmax, and x = xmin (the equations for the sides of the window) into the above equation. We can check the locations of the points of intersection to determine if they are on the line segment or only on the line of which it is part.
- 1.5 For a line segment, show that clipping against the top of the clipping rectangle can be done independently of the clipping against the other sides. Use this result to show that a clipper can be implemented as a pipeline of four simpler clippers.
    - In Exercise 1.4, we saw that we could intersect the line of which the line segment is part independently against each of the sides of the window. We could do this process iteratively, each time shortening the line segment if it intersects one side of the window.

Spinning Up The Lab Code

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



- Talk about your current code and editor and how to set that up.
    a. All you need is a text editor and a browser that supports WebGL 2.0
        - Recommend Chrome and Notepad++ or VS Code
    b. Mention other options
        - Notepad++
        - VS code

        - Vim
        - Emacs
        - Firefox (note GL support varies between browsers)
        - Edge (note GL support varies between browsers)
- In order to get a picture, you need:
    a. A place to draw (canvas/window)
    b. A thing to draw (vertices)
    c. A colour to draw (shaders)
    d. And then we need to draw it!

Draw a triangle

1. Go over the code for the triangle to show how this all works.
2. Explain each part with reference to the process above.
3. Start in triangle.js
    a. Lines 5-9 in triangle.js define an init function as an event handler for when the page load event is fired, then to find the canvas to draw on, and finally the gl context we will use to access the WebGL API. Line 10 just throws an error for us if WebGL isn't supported in the browser.
    b. Line 27 creates a viewport using the canvas
        i. Note the "canvas" is an HTML5 object used for drawing. It must be defined in the HTML code (we'll see this below in triangle.html)
    c. Line 28 defines the colour used to "cleans" the buffer that will be drawn into the viewport created on Line 27 by clearing the entire viewport with a single colour (we will see the call to do this down below). We will learn what a viewport is later in class, but it is basically what it sounds like a kind of window to view things through.
    d. Lines 18-22 create the thing/data (vertices or points in space) to draw. These have X and Y coordinates. There are three of them because we want to define the corners of a two-dimensional triangle. vec2 is a helper type defined in Common/MV.js because we will need coordinates **a lot** in this course and in graphics in general. *(we will get into MV.js more in a later lab)*.
4. Move to the triangle.html file and explain the basics of the shaders.
    a. The Shader programs are in the HTML file but are **not** HTML code, see the <script> tags and their types! These could be in separate shader-specific files, but you end up with local file loading security issues. From an engineering perspective probably a good idea to have them in separate files, from a pedagogical perspective putting the code in script tags makes it easier to focus on the good stuff instead of file loading and security issues. Note that our triangle.js file and our helper JS files are also identified in <script> tags here so they load when the page loads in your browser.
    b. Lines 4-13 define the Vertex shader written in OpenGL Shading Language (GLSL). You don't need to know what this is right now in detail, but you may guess just how many vertices are needed (we are drawing a triangle so 3! And we created 3 two dimensional points remember!). This small program will be run for each vertex!
    c. Lines 15-26 define the Fragment shader also written in GLSL. This is basically how we preprocess pixel data before the data are used to turn on actual screen pixels. The pipeline eventually turns fragments (pre-pixel data) into pixels (picture elements)! That means this is run for every single pixel we see on the screen!
    d. Line 33 If we jump down here, we will see the definition of the canvas we discussed above. Note that it is 512x512 pixels, that's 512x512 = 262,144 fragments we need to run the fragment shader on. This is where we highlight the beauty of the real-time graphics pipeline! We must run

that entire program 262,144 times to paint the screen once. Imagine a 2k video game, 2k is a generic term, but consider a typical 2560x1440 resolution. That is 3,686,400 times the fragment shader needs to be run for every single frame we render. So, say, for example, we want to render a game in 2k at 60 frames per second (fps), that means we need to run this entire fragment shader program 221,184,000 times every second! This is a trivial shader, future shaders you will work with are not!

e. The shaders don't really do anything in this case, it's what we call a *passthrough* vertex shader and a fragment shader with one colour: red.

5. Jump back to triangle.js
    a. The shaders are compiled into a shader program on line 32 in triangle.js
        i. SIDENOTE: when working with just vertex and fragment shaders (the shaders we need to make stuff appear on screen) this process is basically always the same. So, for the remainder of this course we've offloaded this onto a helper function in Common/initShaders.js. If you're interested in the process check it out there. It is similar to compiling any program (obtain sources, compile sources, link compiled objects into program) except we get back a program object instead of an executable file.
    b. The shader program is then loaded onto the GPU line 33 in triangle.js
    c. Now we need to draw everything
    d. Remember the GPU pipeline. Lines 37-39 prepare the data we defined above and create a buffer to store it.
    e. Lines 43-45 associate the buffer of data we just created with a variable in our vertex shader (note the name of the functions vertexAttribPointer, enableVertexAttribArray). The string "aPosition" is the name of a variable in the vertex shader defined in triangle.html. WebGL can find variables, or *attributes*, in both vertex and fragment shaders by their names. If you look closely, you'll see we have custom defined vec2 data type in JavaScript going into a GLSL defined vec4 variable. WebGL can handle this automagically, partly because we are telling it where, what type, and how big what we are sending the GPU via the function vertexAttribPointer(index, size, type, normalized, stride, offset). Each data we send will be two floating point values. WebGL shaders know to put those values into the first two locations of the receiving type, so we end up with a vec4 on the GPU that is something like this (x, y, 0, 0) in the vertex shader.
    f. Lines 51-54 are a Render function we've made tells the GPU to draw the thing.
        1. First on Line 52 we clear the color buffer (a data buffer where we will store colour values as we compute them in our fragment shader near the end of the pipeline). Remember in Line 28 we defined what colour to write into this buffer. What is being hidden from you, as someone writing code for WebGL, is that a colorbuffer, a buffer that is part of a framebuffer, has been written with the same colour to all the buffer values. This framebuffer is eventually drawn to the screen. A framebuffer is memory we write values to as a render target. Later we will show how we can make use of these buffers to store intermediate information too!
        2. Now on Line 53 we tell the GPU to take the data we are sending it and use it to draw triangles. Remember we bound a buffer of points, told the GPU where those points should be sent in the vertex shader, then the vertex shader is going to pass values through to the fragment shader (in between a lot of complicated stuff happens that we will learn about, the key is that GPU figures out which screen locations are inside the triangles and which are outside), the fragment shader associates a colour with this location in the colorbuffer, once the colorbuffer is completely written to the buffer data is sent off to your devices screen hardware where it is used to light up the pixels you see. In this case we

have a single triangle. SO, what you should see is a triangle of the colour we defined (in the fragment shader) on a background of the colour we defined (in the WebGL clearColor), note that you can define colours and locations and all sorts of stuff both on the CPU and the GPU. The example in this lab defines the vertex locations of a single triangle then sends that data to the GPU through the standard create data -> create a buffer -> bind the buffer -> fill the buffer -> associate the buffer with an attribute in a shader program process. If you look at the fragment shader code, we define the colour there, we could, in a similar fashion, have an attribute for the colour we set and sent that data to the GPU as well.

Draw a square

1. Very similar to drawing a triangle (you can show the similarities)
2. Now explain the differences
    a. The difference is really in the data on Lines 18-23, then Line 54
    b. Then the way we draw
        i. Previously we used gl.TRIANGLES in the gl.drawArrays function
        ii. Now we use gl.TRIANGLE_FAN
        iii. Why? Ask the class
            1. Instead of six vertices for the two triangles, we can use four triangles because two of the vertices are common/shared between them. So, when you can, these flags can make your approach data-efficient (less stuff to process, which becomes increasingly more important as scenes become more complex)

Things to Try Now

- Try changing the vertex positions in the JavaScript for both the triangle and square. What's going on here? What's the far left and far right, bottom, and top limit to the space we are drawing in?
- Try changing the colours rendered.
    - First, the Clear colour in the JavaScript passed to gl.clearColor.
        - What are these numbers? red, green, blue, alpha
    - Second the out color passed out of the fragment shader

Expected common questions

- Nothing is working! (not even provided code)
    - Have they put the files in the right place?
    - Does their browser support WebGL 2.0
        - https://get.webgl.org/
        - https://webglsamples.org/
- I can't open the code!
    - Open the files with a text editor, whatever your preferred editor is.
        - Notepad++ or VS Code are simple good options