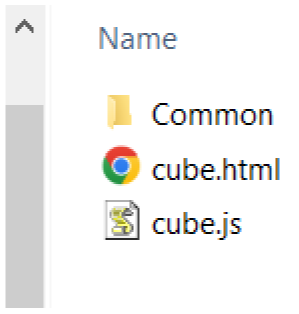
## Exercises

- No exercises! Focus on getting through the code and letting students mess around with it.

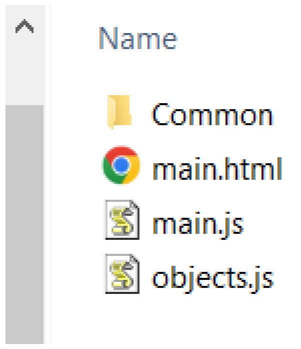
## Spinning Up the Lab Code

- There are two sets of code in this lab.
    a. The first is a simple spinning cube that is animated by passing a uniform time variable to the shader and then applying transformations *in the shader.*
    b. The second is the assignment base code. This is more significant in a few ways, but mainly it implements the entire pipeline and has examples of using the transformation stack, creating objects, and animating objects the way we will for A1 and A2.
- Setup for the Spinning Cube example (Lab3_Code.zip)



Name
Common
cube.html
cube.js

- Setup for the Assignment base code (BaseCode.zip)



Name
Common
main.html
main.js
objects.js

## Spin a Cube!

1. Let’s start with the cube.js file first

    a. Lines 20-55 Much of the previous code is actually the same! We need to setup a canvas and GL context, bind our buffers, and pass data through. We are still using vertex data and colours.
    b. Line 57 we introduce a uniform variable for the first time. Remember this is a global in the shaders and can not be changed there. It is set in our application then sent off to the shader. Note that this line is effectively going to point to the location of the uniform uTheta in the shader. In our application, we have a theta variable that is an array of three values (initialized on line 16). Later, on line 130, we will set uTheta in the shader to the values of theta.
    c. Line 61-69 setup callbacks for buttons that are in our HTML. These button callbacks set which axis is being rotated on. Note that xAxis, yAxis, and zAxis are just 0,1,2 the indices of the elements in theta!
    d. Lines 74-123 setup the coloured cube data by defining the cube as 6 quads. One quad for each side of the cube. A quad is a four-sided figure, typically a square made by drawing two triangles.
    e. Line 127 clears the canvas as before.
    f. Line 129 will increment the currently set axis (one of 0,1, or 2) in theta by 2.0
    g. Line 130 will send the current theta array data to the uniform variable. Note the format here: uniform3fv
        i. Uniform = type
        ii. 3 = size
        iii. f = floating point values
        iv. v = it’s an array
        v. So, we know this uniform is an array of three floating point values, which is exactly what theta is.
    h. Line 132 we draw our triangles as before.
    i. Line 133 we requestAnimationFrame. This is actually part of the web API and has some W3C conventions. You can read about it here:
        i. [https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame](https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame)
2. Now let's look at the cube.html file
    a. This is an example showing that you can apply transformations in a shader too!
    b. Line 11 is new; this is the uniform we were setting in the application code
    c. Line 17 converts the uTheta uniform array degree values to radians. This is key to understanding the speed of the rotation as we were incrementing whatever the active axis was by 2.0, so that was 2 degrees. The speed of this is tricky though because we are relying entirely on the refresh rate of the browser. This could be 60fps, so 60*2 = 120 degrees per second. Or it could be browser specific. See the MDN link above for details if needed. But you can just change the 2.0 on Line 129 of cube.js to a bigger or smaller value.
    d. Line 18 and 19 precompute the sines and cosines of all the values in theta, (so c and s are vectors too, just to store the data)
    e. Lines 22-36 define our three roll rotation matrices for x-roll, y-roll, and z-roll, with the values just computed.

f. Line 39 applies our rotations to the position in the vertex shader, note how the matrix multiplication is natively handled.
g. Lines 44-57 are the fragment shader and as we can see nothing really happens here. Just another pass through shader. However, note how all the triangles that made up the quads that make up the cube are all filled in, rasterization! The position and colour attributes are vertex attributes. These types of attributes are always auto interpolated as part of rasterization. That means the graphics pipeline took information assigned at the vertex and used that to fill in the shape on the screen! We will talk about this in class when we get nearer the end of the graphics pipeline.

## Introduction to the Assignment Basecode

1. First, there is a lot going on here. It is complicated because it fully implements everything we need to build animated and rendered scenes with a camera, lights, and objects all in 3D. The problem is we haven't learned how to do all of this in the lecture yet! So, we will be ignoring big chunks and focusing on what has been learned and what is needed for assignment 1.
2. Let's start with objects.js we don't need to necessary look at at it just know what it provides us. This is a helper file that is used to create objects for us. Specifically, we will focus on Cube, Cylinder, Cone, and Sphere. Object.js gives us the following functions:
    a. Spheres
        i. Sphere.Init(n, program)
            1. Procedurally generates a sphere. N controls how smooth it is! Bigger N = better-looking sphere but more vertices and triangles to deal with internally.
        ii. Sphere.Draw()
            1. Draws the sphere by setting the attributes and calling draw arrays (this is what we were doing in the render function before!)
        iii. Cone.Init(n, program)
            1. Procedurally generates a sphere. N controls how smooth it is! Bigger N = better-looking cone but more vertices and triangles to deal with internally.
        iv. Cone.Draw()
            1. Draws the cone by setting the attributes and calling draw arrays (this is what we were doing in the render function before!)
        v. Cylinder.Init(n, program)
            1. Procedurally generates a sphere. N controls how smooth it is! Bigger N = better-looking cylinder but more vertices and triangles to deal with internally.
        vi. Cylinder.Draw()
            1. Draws the cone by setting the attributes and calling draw arrays (this is what we were doing in the render function before!)

3. Now let’s look at our actual application code in main.js. This file also provides us with several helper functions that ensure we prepare matrices correctly for the “pipeline” when we are creating these objects. The functions we will focus on are related to the modeling stack (it is technically the model view stack, but the view part may not be clear yet, we’ll get there). Each function has comments that provide insight into the shape or use.
    a. Lines 164-170 drawCube will draw a 2x2x2 cube centred at the origin.
    b. Lines 172-178 drawSphere will draw a sphere centred at the origin of radius 1.0
    c. Lines 180-187 drawCylinder will draw a cylinder along z of height 1 centred at the origin and radius 0.5.
    d. Lines 189-196 drawCone will draw a cone along z of height 1 centred at the origin and base radius 1.0.
        i. Note that each of the drawing functions above run the shapes draw function after setting up the matrix data using setMV (Line 151) this function basically (for now this is all they need to know) sending the transform matrices to the GPU. All data after that will be transformed using this transform.
    e. Lines 261-316 show an example how to use each of these in practice. That includes putting them in the right place and then drawing them. These chunks of code use the stack!
    f. Line 41 The stack is initialized.
    g. Lines 216-224 implement helper functions for pushing and popping on and off the stack respectively. These are gPush and gPop
    h. Lines 49-62 gives initialize variables to keep track of object states. Here we have four object we want to draw and animate, but you can have as many as you want. You can also store this info in many different ways (for example, an array of joint angles in a character). But you need to store and use it consistently. Here we simply have a 3-element array for position and a 3 element array for rotation. We are going to manipulate these for animation (see the provided slide deck on animation via evolving degrees of freedom)
    i. Lines 247-259 give us dt or delta time–the difference in time from one render frame to the next. We are going to use this to animate. Think of it this way if speed is in m/s and we multiply that by a duration in s then we get a position in m. Basic integration is explained on **lines 249-256**.
    j. Lines 280-284 give an example of animating a rotation with comments explaining how it works.
    k. Similar examples are seen in the following shapes from Lines 291-315. (Note the difference is just the speed).

## Things to Try Now

- Try changing the colour of the shapes.

- Try changing the shape of the shapes (make a sphere an ellipsoid or the cube a rectangular prism) (hint it’s all in the transformations!)
- Try changing the speed of the rotation.
- Try changing the animation to a translation (make the objects move!)
- Try to make a shape oscillate (rotate back and forth between an angle). Hint, sine(w) and cosine(w) oscillate between positive and negative as w increases.