Exercises

5.1

Not all projections are planar geometric projections. Give an example of a projection in which the projection surface is not a plane and another in which the projectors are not lines.

Eclipses (both solar and lunar) are good examples of the projection of an object (the moon or the earth) onto a nonplanar surface. Any time a shadow is created on a curved surface, there is a nonplanar projection.

All the maps in an atlas are examples of the use of curved projectors. If the projectors were not curved, we could not project the entire surface of a spherical object (the Earth) onto a rectangle.

Mercator projection, for example, stretches the earth out at the poles.

5.2

Consider an airplane whose position is specified by the roll, pitch, and yaw and by the distance from an object. Find a model matrix in terms of these parameters.

Suppose that the axis of the plane is its z direction and up is the y direction. In the airplane's coordinate system, the roll, pitch, and yaw correspond to rotations about the z, x and y axes respectively.

Thus, we can control the orientation relative to the origin by a rotation of the form

Rz(roll) * **R**x(pitch) * **R**y(yaw)

We must also do a translation to move the airplane to its desired location.

T(x,y,z)

Multiply the matrices together and we get the model matrix.

5.5

Can we obtain an isometric of the cube by a single rotation about a suitably chosen axis? Explain your answer.

Yes. an isometric of a cube is just a cube with a certain rotation. To reach this specific rotation we can use the fact that any sequence of rotations is equivalent to a single rotation about a suitably chosen axis (Euler's rotation theorem). One way to compute this rotation matrix is to form the matrix by a sequence of simple rotations, such as R = RxRyRz. The desired axis is an eigenvector of this matrix.

An eigenvector of this matrix is a vector that when multiplied by this matrix or rotation, remains the same. For any rotation about any axis, this would be that axis of rotation.

The eigenvalues of a matrix A are the roots of

$$|A - \lambda I|$$

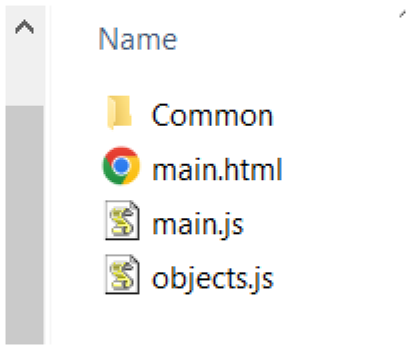
Where || is the matrix determinant. This will give you a polynomial, the roots of that polynomial are the eigenvalues of matrix A. You can then plug those eigne values into the equation:

$$(A \; - \; \lambda I)v = 0$$

Solve for the components of v given an eigenvalue and you will get one of the eigenvectors.

Spinning Up The Lab Code

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



Common
main.html
main.js
objects.js

Introduction to the Assignment Basecode

1. Some reminders/wrap up of model matrix stuff.
    a. Quick reminder that uniform is a global shader variable.
    b. The location of these are saved for later use (we need to send data from CPU to the GPU),
        i. So, we find ModelView matrix location in shader on line 112 of main.js, called modelViewMatrix in the shader.

        ii. The projection matrix on line 114 of main.js, called projectionMatrix in the shader.
    c. Then when we need to set these for the shader we are calling setAllMatrices (Lines 152-156 sets the projectionMatrix on the GPU then calls setMV) and/or setMV (Lines 144-150 multiplies the Model and the View matrices and sets this on the GPU) which just uses gl.uniformMatrix4fv to actually set the values in the shader
        i. Line 150-156 is the setMV function and note it premultiplies the view matrix to the current state of the model matrix in line 152, then sets the value of the uniform on the GPU to this matrix (it is flattened since that is how these matrices are passed to the GPU). If anyone is interested the flatten operation is Line 622-653 of MV.js in the common directory.
    d. Note how we do this for each object as we position it in the world and then draw it by sending the objects vertex data into the pipeline.
        i. You can see the order of operations in the render function.
            1. For example, on Lines 255 to 266 we position and draw a sphere
                a. Line 263 calls drawSphere which is a helper function.
                b. Lines 166-172 implements drawSphere. Here we call setMV(); first and then the draw function on the Sphere object.
                c. If we trace this through the Sphere object draw function in objects.js, there it setAttribPointers then gl.drawArrays on lines 505-506
2. Let's look at how to move the camera and set up the camera movement
    a. Line 225 of main.js defines a variable to store the origin of the view coordinate system, i.e., the eye position in world coordinates.
    b. Line 232 of main.js calls a function to set the actual view matrix up (note *at* and *up* are defined on lines 34 and 35).
    c. Lines 386 - 423 of MV.js build the matrix when we call the lookAt function.
3. Let's look at how to define projections
    a. Line 235 of main.js we call a function to build the projection matrix. In the base code we have this set to the ortho function defined in MV.js. This gives us back an orthographic, or parallel projection matrix. There is also a perspective projection function that build the perspective project matrix for us. We are currently, or will be soon, learning how to build these or similar matrices ourselves in lecture. For now, note that you need to somehow project a 3D world to a flat screen and the two standard options are orthographic or perspective.
    b. Lines 430 to 449 of MV.js actually builds the matrix for an orthographic projection. HINT: The function to build a perspective projection matrix is nearby.
4. The vertex portion of the pipeline, going from object to model to view to projection in normalized device coordinates.
    a. Line 48 of main.html. The multiplication of the vertex vPosition with the modelViewMatrix and the projection matrix in the vertex shader. It's interesting because that one line essentially implements half the graphics pipeline we've

been looking at in the slides and because it's a programmable shader-based pipeline now we have direct control over this...ie you don't have to do this at all, or you could do it in a different way (if you can figure one out)—also certain effects we may want to render require careful and creative manipulations of data through this pipeline. We'll see this later in the course!

Things to Try Now

- Try changing the parameters of the orthographic projection.
    - Note what effects these have on what is rendered to the screen.
- Try changing the projection itself to perspective.
    - Function call: perspective(fovy, aspect, near, far)
    - Change main.js line 235 to projectionMatrix = perspective(60.0, 1, 1, 20)
    - Note the parallel lines of the cube should now no longer remain parallel when it is rotating unless a face is exactly parallel to the view plane.
    - Try changing the parameters of the perspective projection.
        - Note what effects these have on what is rendered to the screen
        - Can you slice objects with near and far?
- Try animating the camera.
    - What happens when you give it a look at point then rotate the camera?
    - What happens when you update the lookat as you rotate?