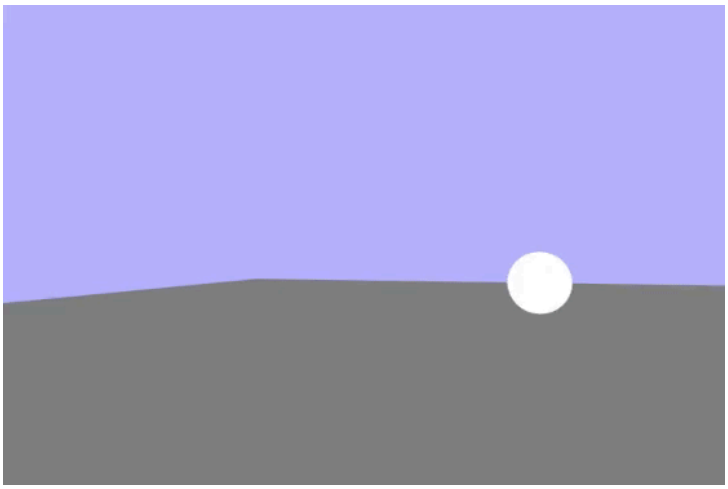
Intro to WebGL II Guide

Overview

In this project you will be manipulating some basic geometry and creating a basic first person camera.



Structure & Execution Order

As we continue to add WebGL code, it becomes more important to better organize it. All of the main application code has been moved out of the **.html** file and into a separate **app.js** file. All of the code for rendering specific objects has also been moved out into its own **.js** file.

The new structure of the code is laid out as follows:

Directory Structure

/Root *(contains main files .html and app.js)*

- ***/data*** *(contains images and models)*
- ***/math*** *(contains our math code)*
- ***/misc*** *(contains code to assist mesh creation, loading, and rendering)*
- ***/shaders*** *(contains the GPU code for rendering)*

Initialization and frame updates occur similarly to the last project. However, now we are loading files from your hard drive asynchronously via Javascript "promises". This is done for you in **app.js:*loadAssets***.

For now, this loading is used for just our shader code files and the geometry for a sphere but in future assignments, there will be more. Note that in the previous assignment our shaders were not loaded but simply added via <script> tags. We will not use that method going forward.

Our single shader program for this assignment will be composed of the code from 2 files now:

root/shaders/color.unlit.vs.glsl (vertex shader)

root/shaders/color.unlit.fs.glsl (fragment shader)

Our geometry and object rendering code will be located at:

root/misc/webgl-geometry-quad.js (contains procedural creation of vertices and interaction code with the WebGL API)

In addition, several other script files provide and encapsulate other supporting features.

root/misc/input.js (reads mouse and keyboard input)

root/misc/time.js (keeps track of time)

root/misc/camera.js (the file used to create first person camera functionality)

As before, the code is incomplete and we will be filling in the missing content as we go.

Common Problems

Be careful when applying scale transformations. You may be tempted to use ***matrix4.js:multiplyScalar*** but be aware of what that is actually doing. Which elements should be affected when applying a scale transform? Review the lecture on scale as needed.

Careful!

Unlike the previous WebGL assignment, you will not be able to run your code simply by launching the **index.html** file. This is because the code now **needs to access separate files like images and shaders from your hard drive**. This is a security violation and all browsers will block it. The recommended way to get around this is to serve your files from a local web server and then start your application by going to your localhost url.

The easiest (**recommended**) way to approach this is by using the LiveServer extension for VSCode. If you would like to be a little more hands-on, you can http-server from node.js or from Python. Alternatively, you can disable browser security by following the instructions here (not recommended).

To start, if you run the code (http://localhost:8080 with **node.js**) you will see only the clear color:

Scene Setup

The scene initially starts with nothing but a simple rectangular quad. However, as you've noticed above, we do not initially see this quad. Note how the quad is created and rendered.

```
function createScene() {
    // uncomment when directed by guide
    groundGeometry = new WebGLGeometryQuad(gl);
    groundGeometry.create();
```

...

```
function updateAndRender() {
    // ...

    // render ground
    gl.uniform4f(colorProgram.uniforms.colorUniform, 0.5, 0.5, 0.5, 1.0);
    groundGeometry.render(camera, projectionMatrix, colorProgram);
```

As you can see, it is meant to serve as our "ground" (from *app.js*) and is encapsulated as part of **WebGLGeometryQuad** (from ***webgl-geometry-quad.js***). This object type contains its own world matrix which will allow us to establish its pose (position/orientation) in the world.

```
function WebGLGeometryQuad(gl) {
      this.gl = gl;
      this.worldMatrix = new Matrix4();
```

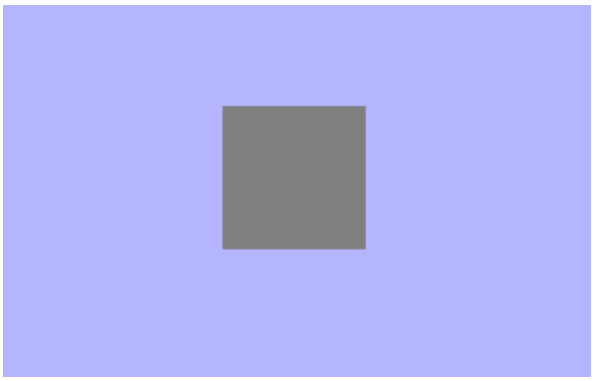
However, nothing we're doing is actually setting that so it is identity by default. So where does the identity matrix put this object in the world? Well, its local vertex coordinates are simply dropped in as if they were world coordinates.

Take a moment to note what those coordinates are (see inside *webgl-geometry-quad.js*) and how that code works.

Todo #1 Move the Quad

In order to see the quad, we have 2 choices: move the camera or move the object. For now, let's translate the quad so that it is in front of the camera. But wait, where is the camera? Take a look at ***camera.js***. Its main purpose is to create a view matrix for rendering. It does this by taking the inverse of its world matrix. However, that matrix is again identity which means the camera is in the default position.

Now you can see the problem. The quad is at the origin and so is the camera. Also, this is right handed so the camera is looking in the -z direction. This means that if we want to see the quad, it has to be somewhere in -z. In ***app.js:createScene()***, try moving the ground geometry to (0, 0, -10). You should see something similar to the following:



Todo #2 Make the Quad Ground-Like

The quad is upright by default and pretty small which is not very ground-like. Fix this by scaling it by 10 and rotating it so it lies on the XZ plane. Also, because this will line up the surface slice with the middle of the screen you won't be able to see it. To get around that, lower your ground geometry with a negative y-value translation of -1. You should see something like this:

Todo #3 Make the Sphere

All of the information about the sphere is loaded in a separately authored file - ***/data/sphere.json***. The file is loaded in ***loadAssets()*** and the raw data stored in the variable ***loadedAssets.sphereJSON***.

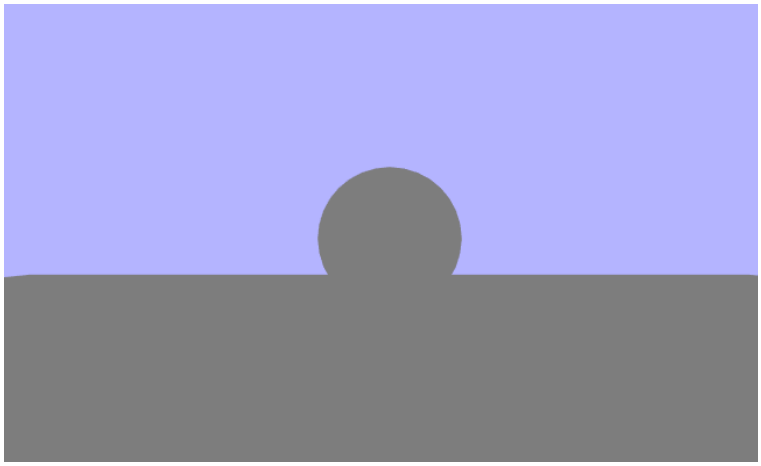
Inside of the ***createScene()*** function, create a new ***WebGLGeometryJSON*** object and pass the raw ***loadedAssets.sphereJSON*** to its ***create()*** function. This should get everything ready for us to render it.

Next, inside of ***updateAndRender()***, add a call to render your **sphereGeometry** similar to the way the ground is rendered. You should see this uninspiring image:

Note that it's the same color as the ground. In this application we are treating color a bit differently than the last project. In the last project, we had a color associated with each vertex. In this one, we're passing a single color to the fragment shader. In this case, both objects are rendered with the same shader and we haven't changed the color. It is still gray… though that still fails to explain why the whole screen is gray. Recall that we didn't author this geometry ourselves, someone else did which means the sizing might be extremely large or small compared to the sizing of other objects we have in the scene. In this case, it's too large.

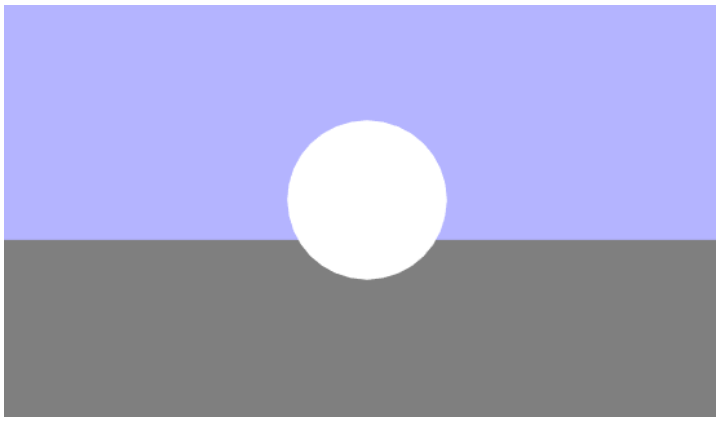
Todo #4 Transform the Sphere

Return to the ***createScene()*** function and modify the worldMatrix on the **sphereGeometry** such that it is a scale of 0.01 (one-hundredth its former size) and translate it in front of the camera at (0, 0, -5). You should see:



Todo #5 Color the Sphere

It would be nice to better distinguish our sphere from the ground plane. Before rendering the sphere, change the value of the color uniform sent to the shader to white (refer to how it is done with the ground). You should see:



Camera Setup

We've been creating mostly static scenes up until now where the viewpoint never changed. It's time to release that shackle. Here is the starting code layout of our camera:

```
function Camera(input) {
    // The following two parameters will be used to automatically create the cameraWorldMatrix in update()
    this.cameraYaw = 0;
    this.cameraPosition = new Vector3();

    this.cameraWorldMatrix = new Matrix4();

    // ------------------------------------------------------------------------
    this.getViewMatrix = function() {
        return this.cameraWorldMatrix.clone().inverse();
    }

    // ------------------------------------------------------------------------
    this.getForward = function() {
        // todo - pull out the forward direction from the world matrix and return as a vector
        //      - recall that the camera looks in the "backwards" direction
        return new Vector3();
    }
    // ------------------------------------------------------------------------
    this.update = function(dt) {
        var currentForward = this.getForward();

        if (input.up) {
            // todo - move the camera position a little bit in its forward direction
        }
```

```
        if (input.down) {
            // todo - move the camera position a little bit in its backward direction
        }

        if (input.left) {
            // todo - add a little bit to the current camera yaw
        }

        if (input.right) {
            // todo - subtract a little bit from the current camera yaw
        }

        // todo - create the cameraWorldMatrix from scratch based on this.cameraYaw and this.cameraPosition
        //      - cameraYaw will equate to a rotation
        //      - cameraPosition will equate to a translation
    }
}
```

Its main function is to return back a view matrix that our application can use to render each object in the scene. You can see this happen when we call render on an object which passes in the camera. Internally, it will use that camera object to retrieve the view matrix. Your job in the following "todos" will be to allow us to change the location and direction of the camera in response to keyboard input.

Todo #6 Determining "Forward"

To start, we will need to know which direction the camera is currently looking in. Implement ***getForward*** in ***camera.js***. The values you need can be pulled from the ***cameraWorldMatrix*** variable (recall the basis vector representation of a matrix). In the next todo, you will find out whether your values were correct.

Todo #7 Going Forward & Backward

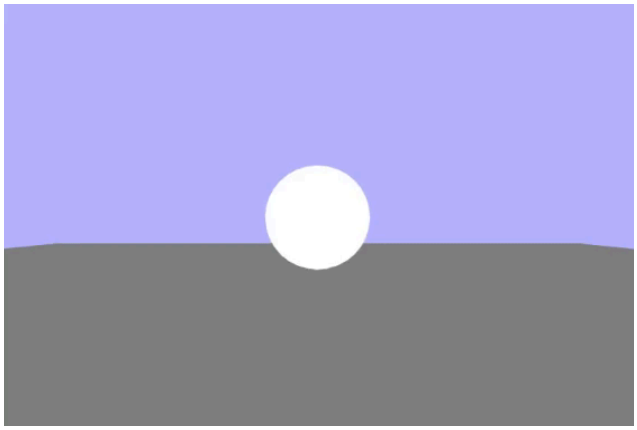
We now want to start modifying the camera position to move forward or backward based on whether the user is pushing the "up" or "down" cursor keys. If either key is held down during a frame, we will increment the camera position a bit. See the following (in ***camera.js***):

```
this.update = function(dt) {
        var currentForward = this.getForward();

        if (input.up) {
            // todo - move the camera position a little bit in its forward direction
        }
```

If the “up” cursor key is pressed, ***input.up*** will evaluate to true. Inside that code block, increment your ***cameraPosition*** a **little bit** in the ***currentForward*** direction.

At this point, nothing different will have happened because our app code only relies on ***getViewMatrix()*** which is created from ***cameraWorldMatrix***. To finish hooking things up, make sure you recreate ***cameraWorldMatrix*** such that it is now a translation to your ***cameraPosition***.



Moving backward is implemented in the same way but in the reverse direction. Make sure the down cursor causes this behavior.

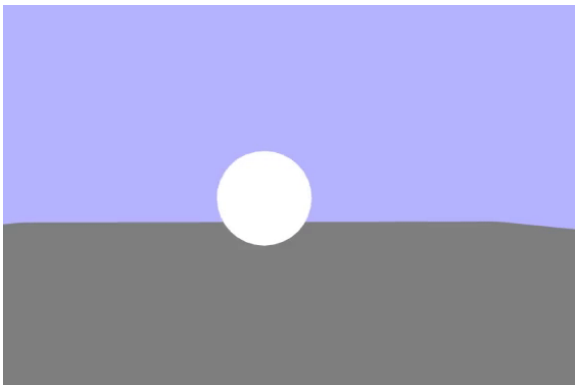
Todo #8 Changing Yaw

Much like you changed the camera position and used it to affect the camera’s world matrix, you will now be changing the value we store of the camera yaw and apply that to the camera world matrix as well. Slightly increment/decrement the yaw value in the code below:

```
if (input.left) {
   // todo - add a little bit to the current camera yaw
}

if (input.right) {
    // todo - subtract a little bit from the current camera yaw
}
```

Now when building the cameraWorldMatrix at the end of the update function, you will want it to be a composition of both a rotation and a translation. If you do this in the correct order, you should now be able to move around the scene with cursor keys.



Todo Commit

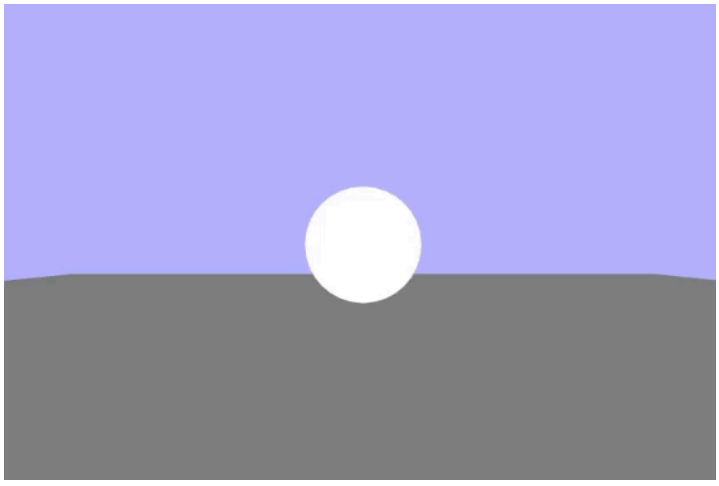
Don't forget the importance of saving your progress. Do this by committing changes to your GitHub repository and pushing those changes to Github. Commit with the message "B portion completed". 5 points will be deducted from the "A" portion for not having this.

Implementation "A"

Color

Todo #9 Oscillating Shade/Color

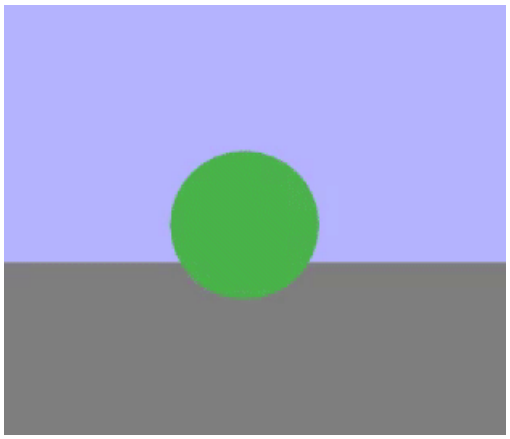
A gray ground and a white sphere does not make the most visually interesting scene. Let's play with our colors a bit such that they animate. Inside of ***updateAndRender()***, let's create a value that oscillates inside the 0 to 1 range. Start with the trig function of your choice and get something in the -1 to 1 range by passing in ***time.secondsElapsedSinceStart*** as your angle. Then remap that -1 to 1 range so that it goes from 0 to 1. Take this value and pass it to the shader through the color uniform before the sphere is rendered.



Todo #10 Fun with Color

Try coming up with your own way of animating color. In the example below, I simply inverted one of my color channels.

```
gl.uniform4f(colorProgram.uniforms.colorUniform, shade, 1.0 - shade, shade, 1.0);
```



Grading

Completing everything labeled as required for a “B” will net 85% of the available points. The remaining todos must be completed for the remaining 15%.

Bonus

Todo #1 Your own idea

If you have an idea for how to improve this app, send me a message on Teams or email proposing what you’d like to do and we can come up with a fair value for the number of extra credit points.

A couple suggestions:

- A first person camera using mouse instead of the keyboard for orientation (i.e. mouselook)
- Make a new webgl-geometry-box.js that creates a box using 8 vertices
- Make the sphere have hierarchically orbiting tiny spheres. Imagine the original sphere is the sun and you now have a planet orbiting with a moon orbiting it