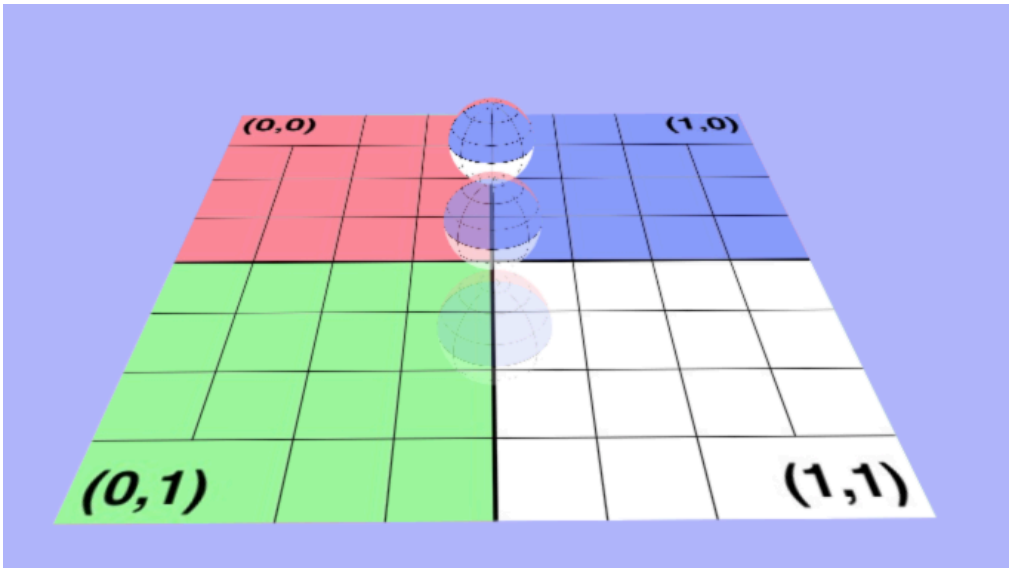
Texturing & Transparency Guide

Overview

In this assignment, you will create a WebGL application showcasing both texturing and transparency. It will consist of 2 triangles making up the ground (a quad) and 3 spheres with varying levels of alpha transparency.



It is assumed that you have covered the lectures on texturing and transparency.

Use the following resources for implementation details:

- WebGL Programming Guide (texturing in ch 5 and alpha blending in ch 10)
  - Pages 171-172 and 174-176 may be especially useful
- https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Using_textures_in_WebGL
- https://webglfundamentals.org/webgl/lessons/webgl-3d-textures.html

Structure & Execution Order

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 creating, loading, and rendering)*
- ***/shaders*** *(contains the GPU code for rendering)*

Our assets consist of an image to be used as a texture, a 3D model of a sphere, and our shader code files.

The single shader program for this assignment will be composed of the following:

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

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

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

root/misc/webgl-geometry-json.js (loads data from a .json file and renders)

root/misc/webgl-geometry-quad.js (procedurally creates itself and renders)

root/misc/input.js (takes care of reading mouse and keyboard input)

root/misc/time.js (takes care of keeping track of time)

root/misc/camera.orbit.js (enables using the mouse to rotate the camera around the scene)

Common Mistakes

1) Trying to assign a value to a “uniform” or an “attribute”. Uniforms are constant throughout the render of an object. Attributes are the input data (a single vertex) to the vertex shader. They no longer exist after rasterization (before the fragment shader).
2) Not setting the wrapping mode for both directions. This may cause WebGL errors.
3) Not setting both filtering modes (magnification and minifications). This may cause WebGL errors.
4) Not using a decimal point for floating point numbers in the shader. Shader code is notoriously strict and will fail with ANY syntax issue.

Careful!

Unlike previous assignments, 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 (rather than a server). 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 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 use node.js or Python. Alternatively, you can disable browser security altogether by following the instructions here (not recommended).

To start, if you run the code you will see only the clear color:

You will have several errors in your console which you can ignore for now:

WebGL: INVALID_VALUE: vertexAttribPointer: index out of range

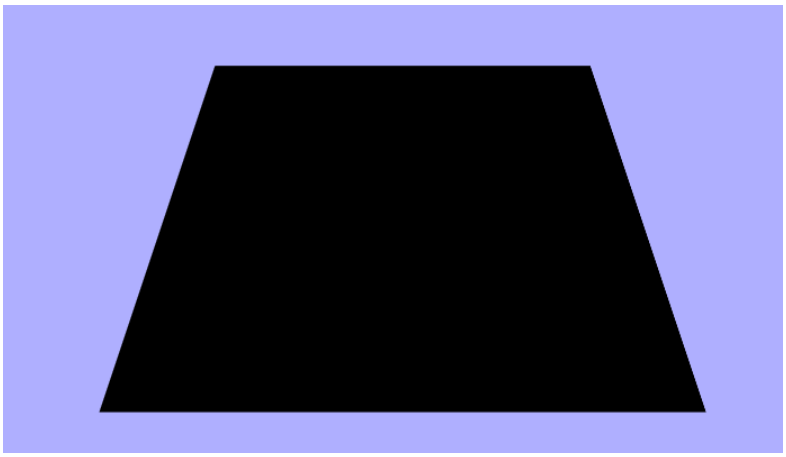
WebGL: INVALID_VALUE: enableVertexAttribArray: index out of range

WebGL: INVALID_VALUE: disableVertexAttribArray: index out of range

Todos

Todo #1 Create and return a correct view matrix from the orbit camera

It will be useful to orbit our eye/camera/view around the scene to see our results. To this end, we will be using an orbiting camera rather than first-person. This code is provided for you, see ***root/misc/camera.orbit.js***. However, ***getViewMatrix()*** is not implemented. Implement this function by creating and returning the appropriate matrix (see the lecture on geometry processing). If you do this correctly, you should see the following and be able to orbit the camera by pressing and holding the left mouse button and dragging.



Todo #2 Pass texture coordinates for interpolation to fragment shader

In ***unlit.textured.vs.glsl***, our texture coordinates enter as part of a vertex (an attribute). Make sure that you put that texture coordinates in the right place to enable its value to be interpolated and caught in the fragment shader (refer to the lecture on shaders). You will be able to verify what you do here after completing the next todo.

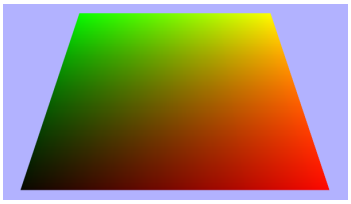
```
precision mediump float;

attribute vec3 aVertexPosition;
attribute vec2 aTexcoords;

uniform mat4 uWorldMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjectionMatrix;

// todo - make sure to pass texture coordinates for interpolation to fragment shader (varying)
//        1. Declare the variable, 2. Set it correctly

void main(void) {
    gl_Position = uProjectionMatrix * uViewMatrix * uWorldMatrix * vec4(aVertexPosition, 1.0);
}
```

Todo #3 Receive the texture coordinates in the fragment shader

Set this up correctly and then visualize the result by setting the output color red and green channels equal to the u and v of the texture coordinates. You may want to review the lecture on “Shaders”.

```
precision mediump float;

uniform sampler2D uTexture;
uniform float uAlpha;

// todo receive texture coordinates and verify correctness by
// using them to set the pixel color as indicated below

void main(void) {
    gl_FragColor = vec4(/*your "u"*/, /* your "v" */, 0.0, 1.0);
}
```

You should see something similar to this:



Note how the colors indicate the UV coordinates at each point.

Todo #4 Set up texturing for the ground

In ***weblg-geometry-quad.js:create()***, add code to support texturing (see chapter 5 in the WebGL Programming Guide).

```
if (rawImage) {
    // 1. create the texture (uncomment when ready)
    // this.texture = ?;

    // 2. todo bind the texture

    // needed for the way browsers load images, ignore this
    this.gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

    // 3. todo set wrap modes (for s and t) for the texture
    // 4. todo set filtering modes (magnification and minification)
    // 5. send the image WebGL to use as this texture

    // We're done for now, unbind
    this.gl.bindTexture(gl.TEXTURE_2D, null);
}
```

See documentation for texParameteri and texImage2D for more help.

Then, inside of ***webgl-geometry-quad.js:render()***, make sure that the texture is correctly bound in the correct slot so that it will be available during rendering. The code is already set up here for you, you just need to uncomment it and plug in the correct arguments to the functions.

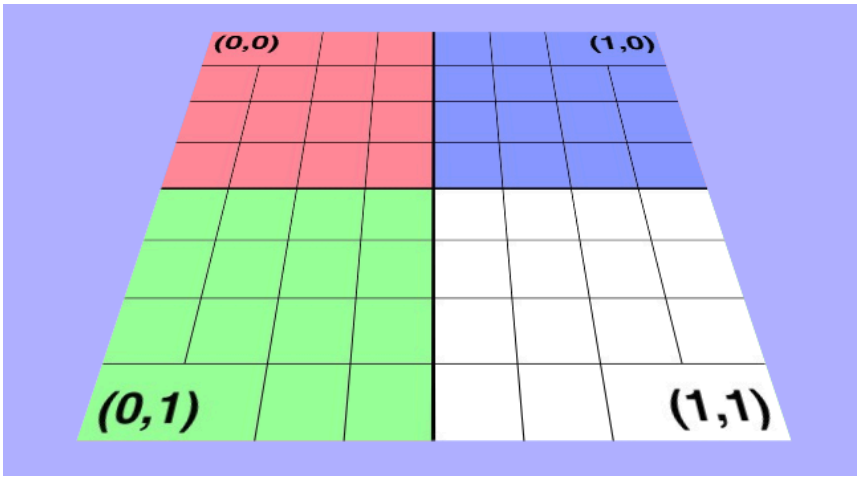
```
if (this.texture) {
    // uncomment when ready
    // gl.activeTexture(?);
    // gl.bindTexture(?, ?);
}
```

You will be able to test the result of this after the next todo.

Todo #5 Sample from the texture in the shader

Instead of setting the final color equal to the texture coordinates, now set the final color to be the result of sampling the texture ***uTexture*** using your texture coordinates (use the GLSL shader function *texture2D*). Documentation for GLSL may be tricky to navigate but in any case you can always refer to the example mentioned in the top of this guide.

Afterward, you should see something like this:



Todo #6 Trilinear filtering (i.e. mip mapping)

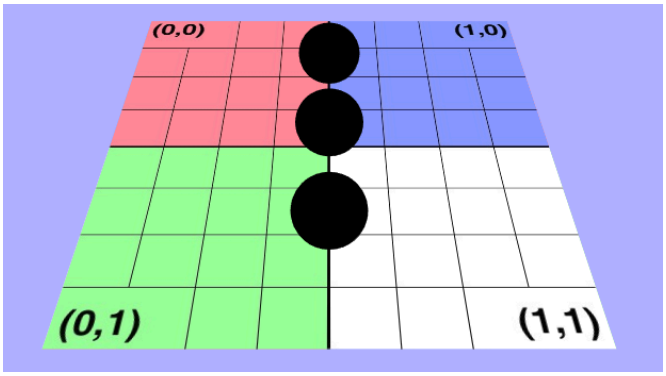
In ***weblg-geometry-quad.js:create***, add code to support mip-mapping and trilinear filtering. This will require a call to generateMipMaps and a modification to part of what you did in todo #4 with texParameteri. You should notice a minor improvement to image quality afterward. As a temporary test, try increasing the scale of the plane by 100 and then move the camera to look down the horizon. Undo your test modifications before proceeding.

Todo #7 Implement the spheres

First, uncomment the code that renders the sphere in app.js:***updateAndRender***.

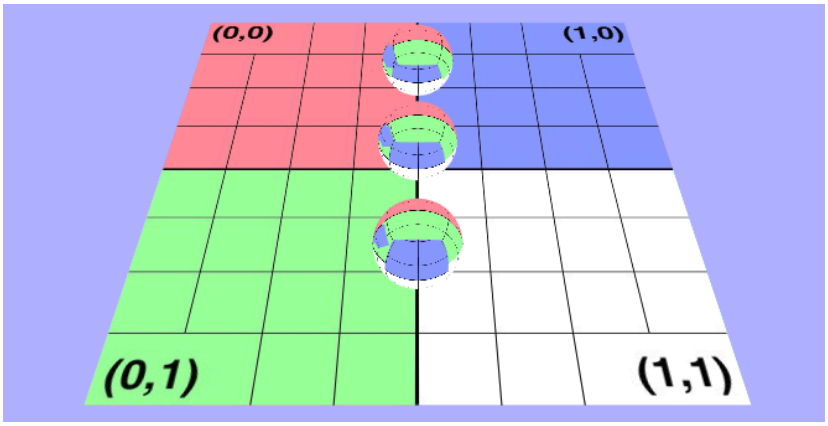
```
// for (var i = 0; i < sphereGeometryList.length; ++i) {
//     sphereGeometryList[i].render(camera, projectionMatrix, textureShaderProgram);
// }
```

You should be able to see them but they will be black.



Next, in *webgl-geometry-json.js:**create*** and *webgl-geometry-json:**render***, set up texturing the same way you did for the quad.

This will allow you to see the texture on the spheres similar to this:



At first glance this may appear as if it is working correctly but try rotating the camera around and looking from different angles. Can you guess what's wrong?

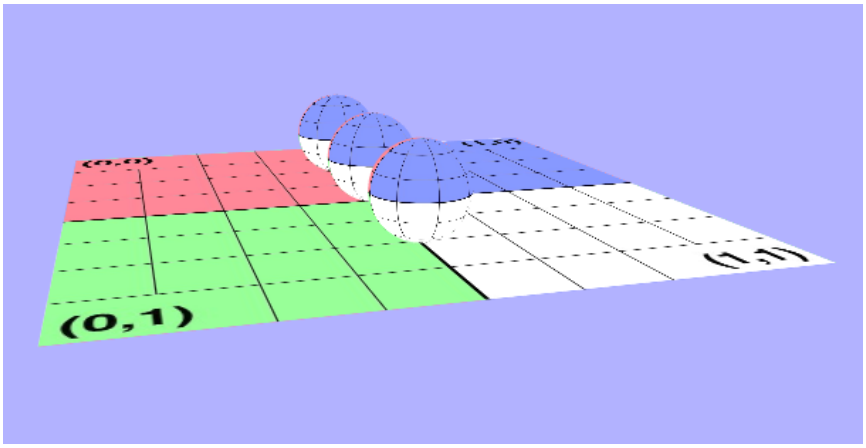
Also, you should no longer see errors in your console.

Todo #8 Enable depth testing and backface culling

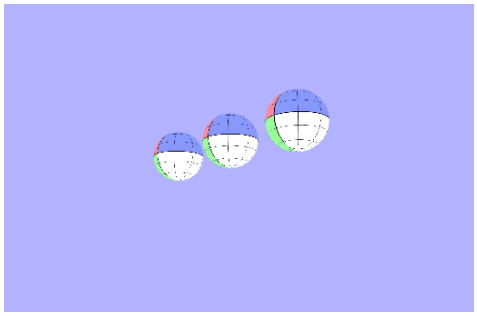
In *app.js*:***initializeAndStartRendering***, execute the gl commands that turn on depth testing (using the z-buffer) as well as enabling backface culling (not enabled by default).

```
// --------------------------------------------------------------------------
async function initializeAndStartRendering() {
    gl = getWebGLContext("webgl-canvas");

    await assetLoader.loadAssets(assetList);

    createShaders();
    createScene();

    // todo #8
    // - enable depth test (z-buffering)
    // - enable backface culling

    updateAndRender();
}
```

If you've done this correctly, then the spheres should now only show they're "outsides".



Also, if you look at the scene from below, the ground plane will be culled.



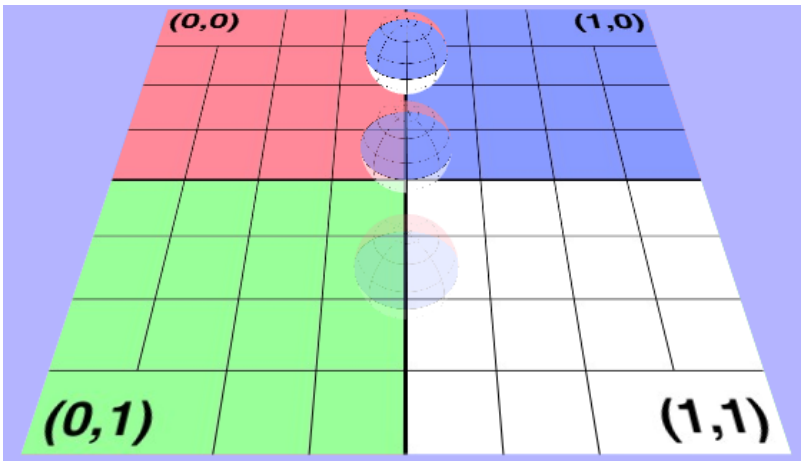
Todo #9 Enable alpha blending for the spheres

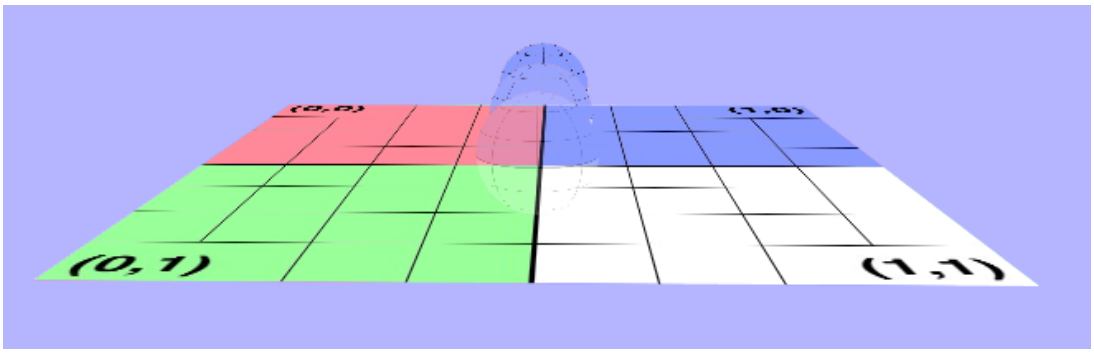
First, make sure to set the alpha channel of *gl_FragColor* to be equal to *uAlpha* instead of using whatever happens to be in the alpha channel of the texture. This is necessary to see the result of blending.

In *app.js:**updateAndRender***, change the WebGL state before rendering the spheres to make them semi-transparent.

```
// todo
//   1. enable blending
//   2. set blend mode source to gl.SRC_ALPHA and destination to gl.ONE_MINUS_SRC_ALPHA

for (var i = sphereGeometryList.length - 1; i >= 0; --i) {
    sphereGeometryList[i].render(camera, projectionMatrix, textureShaderProgram);
}

// todo - done with blending, disable it
```

You should get something similar to this:





Notice in the bottom picture how the closer spheres don't blend with the ones in back.

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"

Todo #10 Add getPosition() functions to the camera and geometry objects

In the next todo, we will be applying the painter's algorithm. In order to do this, we need to be able to compare the distance from each object to the camera so let's make things a little easier for ourselves by creating functions that just return the positions of those things.

In webgl-geometry-json.js:

```
this.getPosition = function() {
    // todo - return a vector3 of this object's world position contained in its matrix
```

```
}
```

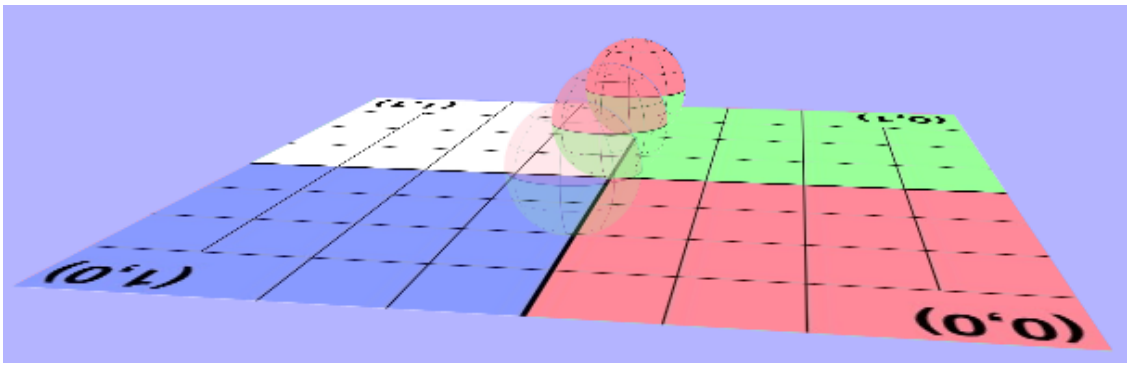
In camera.orbit.js:

```
this.getPosition = function() {
   // todo - return a vector3 of the camera's world position contained in its matrix
}
```

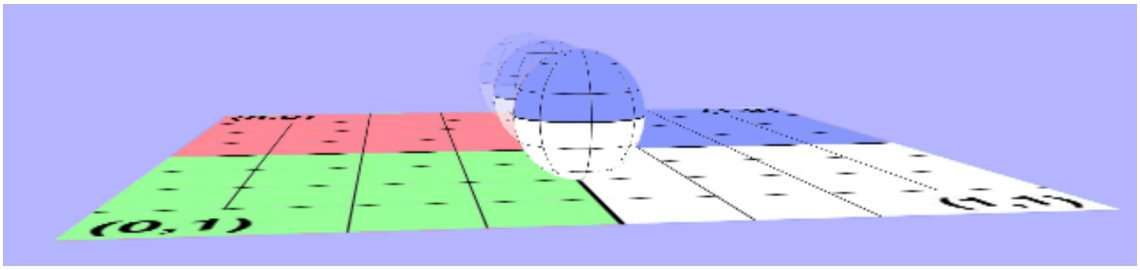
Todo #11 Use the painter's algorithm to ensure the spheres draw in the correct order

Instead of rendering the spheres in the same order every time, now render them such that the furthest sphere from the camera draws first, the 2nd farthest next, etc… You will need to find a way to get each object's distance from the camera in order to do this. You may find the javascript array sort function useful combined with the *getPosition()* functions you made in the previous todo.

If you did this correctly, transparency should look correct no matter where you are viewing them from. Try orbiting the camera around the scene to test.



All 3 spheres are blended such that colors from all of them contribute to the final image.



From this angle, the opaque sphere occludes the other 2.

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 Animate the texture across the surfaces (3 pts)

Create a new float uniform for your fragment shader and send it the value of **time.secondsElapsedSinceStart**. Then use that value to offset your texture coordinates. If the texture looks messed up, make sure that your texture wrap mode is set to “repeat”.

Todo #2 Use multiple textures (5 pts)

Load a new texture of your choosing (in addition to the one that is already loaded) and send it to the shader as a new uniform. Then, create the final color in the fragment shader by taking the average of the two texture colors.

Todo #3 Use anisotropic filtering (2 pts)