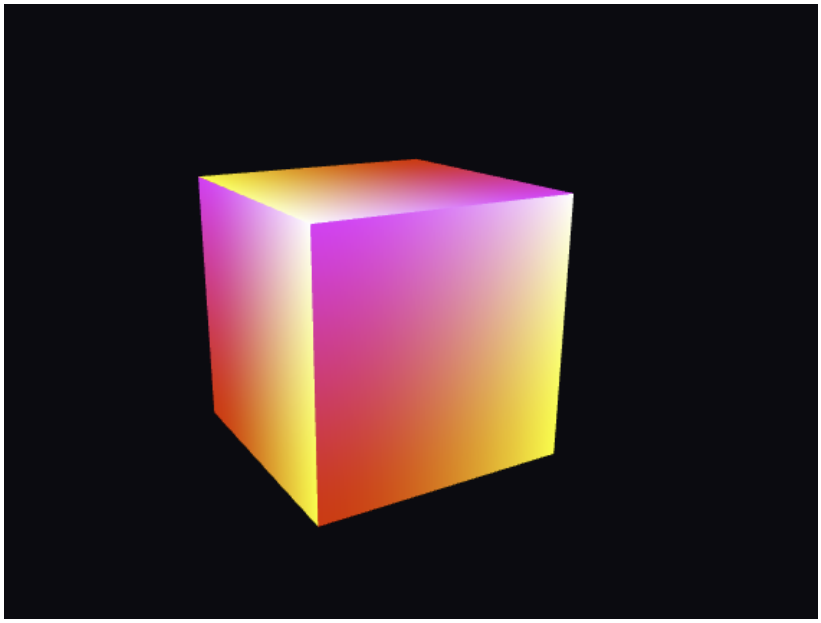
Lab Activity 3

In this lab activity we are going to take a deeper look at uv-coordinates and texturing objects in webGL. We are going to use this opportunity to get more comfortable with our shader programs as well. Head over to the sandbox template here: https://codesandbox.io/s/lab-3-template-gj8wd

The Environment

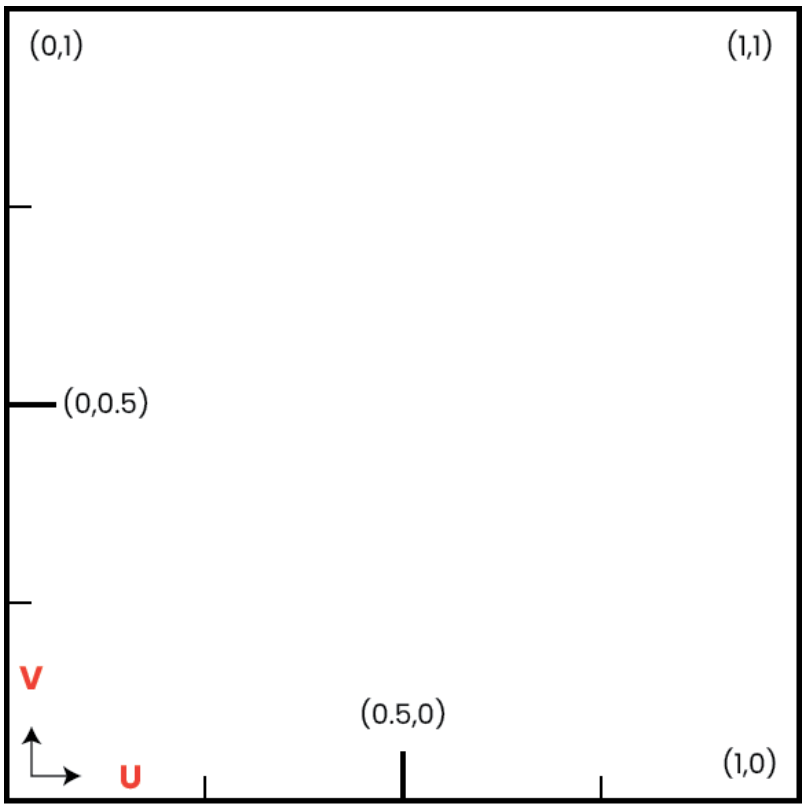
When you open the sandbox you'll find a cube. You can control the rotation of this cube by dragging on the canvas. Notice the color of this cube; the color is dependent on the UV-coordinates of this cube which have already been implemented:



Now, you can see that there is a `Cube.js` file, this is where we will be doing almost everything in this lab activity.

The UV-Coordinates

Let's dive deep into how the UV-coordinates of this cube work. We're going to map this texture onto each face of our cube:



1. Notice that the `img/` directory contains this `uvCoords.png` file. Let's import it at the top of our `index.js` file with the rest of our imports:

```
12  import Stats from "stats.js";
13  import Cube from "./Cube";
14  import Camera from "./Camera";
15  import RotateControls from "./Controls";
16  import uvImg from "./img/uvCoords.png"; // add this
```

    a. This gives us the path to the image. If you `console.log(uvImg)` it will print out a url.
2. Let's head into our Cube class in `Cube.js`:
    a. We need to map an image onto our cube, and in order to do that we need to put that image into the webGL context as a texture:

i. Create a new function **in the Cube class** called `setImage()` that takes in two parameters: *gl* and *imagePath*:

```
export default class Cube {
  constructor() {
    this.vertices = null;
    this.uvs = null;
    this.vertexBuffer = null;
    this.uvBuffer = null;
    this.texture0 = null; // add this

    this.position = new Vector3([0, 0, 0]);
    this.rotation = new Vector3([0, 0, 0]);
    this.scale = new Vector3([1, 1, 1]);
    this.modelMatrix = new Matrix4();

    this.setVertices();
    this.setUvs();
  }

  setImage(gl, imagePath) {
    // lets setup our texture here
  }
```

ii. Textures in webGL require a place in memory similar to array buffers. Let's create that in the class constructor:

```
export default class Cube {
  constructor() {
    this.vertices = null;
    this.uvs = null;
    this.vertexBuffer = null;
    this.uvBuffer = null;
    this.texture0 = null; // add this
```

iii. Now back to our function, let's setup this texture buffer and grab the uniform variable from the shader so we can send the texture to it:

```
setImage(gl, image) {
  if (this.texture0 === null) {
    this.texture0 = gl.createTexture();
  }

  // grab pointer to shader uniform variable
  const uTexture0 = gl.getUniformLocation(gl.program, "uTexture0");
}
```

iv. Images don't load immediately, so we need to tell the image to send itself to the webGL context once it *has* loaded. Let's create a new `Image()` for this purpose **inside our `setImage()` function**:

```
setImage(gl, imagePath) {
  if (this.texture0 === null) {
    this.texture0 = gl.createTexture();
  }

  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

  const uTexture0 = gl.getUniformLocation(gl.program, "uTexture0");
  if (uTexture0 < 0) {
    console.warn("could not get uniform location");
  }

  const img = new Image();

  img.onload = () => {
    gl.activeTexture(gl.TEXTURE0);

    gl.bindTexture(gl.TEXTURE_2D, this.texture0);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, img);
  };

  img.crossOrigin = "anonymous";
  img.src = imagePath;
}
```

1. If you haven't seen arrow function syntax before, it is effectively identical to regular function definitions but maintains the scope of the location it is defined in (meaning the `this` keyword will continue to point to our object, if we ended up needing it).
    a. E.g `function() {console.log("hello")}` is identical to `() => {console.log("hello")}`

v. Now we can set the image to use our `uvCoords.png` file (note that we need to set the crossorigin policy manually here due to how Codesandbox hosts files):

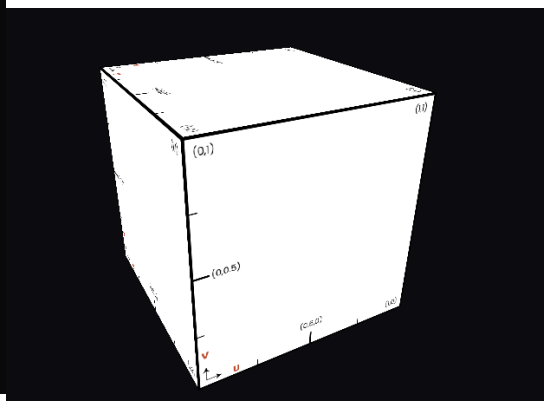
```
      gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, img);

      // set uniform to proper texture slot (should match slot set in line 32)
      gl.uniform1i(uTexture0, 0);
    };

    img.crossOrigin = 'anonymous'
    img.src = imagePath;
  }
```

1. Cross-origin policy is just a security measure that ensures that code is not downloading assets from a different domain (i.e. our sandbox is at http://codesandbox.io whereas the images in our sandbox were originally from elsewhere and that data is embedded in them). You can read more about this here: https://webglfundamentals.org/webgl/lessons/webgl-cors-permission.html

vi. Now, just alter the fragment shader to utilize this image, and call the function through our Cube instance and we should see the image on it:

```
// Fragment shader program
const FSHADER_SOURCE = `
  #ifdef GL_ES
  precision mediump float;
  #endif

  uniform sampler2D uTexture0;
  uniform sampler2D uTexture1;

  varying vec2 vUv;

  void main() {
    vec4 image0 = texture2D(uTexture0, vUv); // add this

    gl_FragColor = image0; // set this
  }
  `;
```

```
const cube = new Cube();

// ADD THIS
cube.setImage(gl, uvImg);
```



Two Textures For Price Of One

Now let's use these coordinates to help us map another texture onto our cube. Import the `dice.png` file like we did the other image:

```
import uvImg from "./img/uvCoords.png";
import diceImg from "./img/dice.png";
```

Now, let's alter our cube's `setImage()` function so it can handle a second image.

1. Head back to the `Cube.js` file and go to the `setImage()` function
    a. Add a third parameter, `index`, so we can differentiate between which texture slot we want to set
    b. Also, add an if statement to check when index is 0, and put all of our existing code from this function under that condition:

```
setImage(gl, imagePath, index) {
  // ADD THIRD PARAMETER
  if (index === 0) {
    // ADD THIS

    if (this.texture0 === null) {
      this.texture0 = gl.createTexture();
    }

    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1);

    const uTexture0 = gl.getUniformLocation(gl.program, "uTexture0");

    const img = new Image();

    img.onload = () => {
      gl.activeTexture(gl.TEXTURE0);

      gl.bindTexture(gl.TEXTURE_2D, this.texture0);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
      gl.texImage2D(
        gl.TEXTURE_2D,
        0,
        gl.RGBA,
        gl.RGBA,
        gl.UNSIGNED_BYTE,
        img
      );

      gl.uniform1i(uTexture0, 0);
    };

    img.crossOrigin = "anonymous";
    img.src = imagePath;
```

c. Add a texture buffer variable to our class:

```
export default class Cube {
  constructor() {
    this.vertices = null;
    this.uvs = null;
    this.vertexBuffer = null;
    this.uvBuffer = null;
    this.texture0 = null;
    this.texture1 = null; // ADD THIS
```

d. Now check for when index is 1, and we'll do the same thing but for texture slot 1 (**this is where many people would copy/paste, if doing so be very careful about what needs to change**):

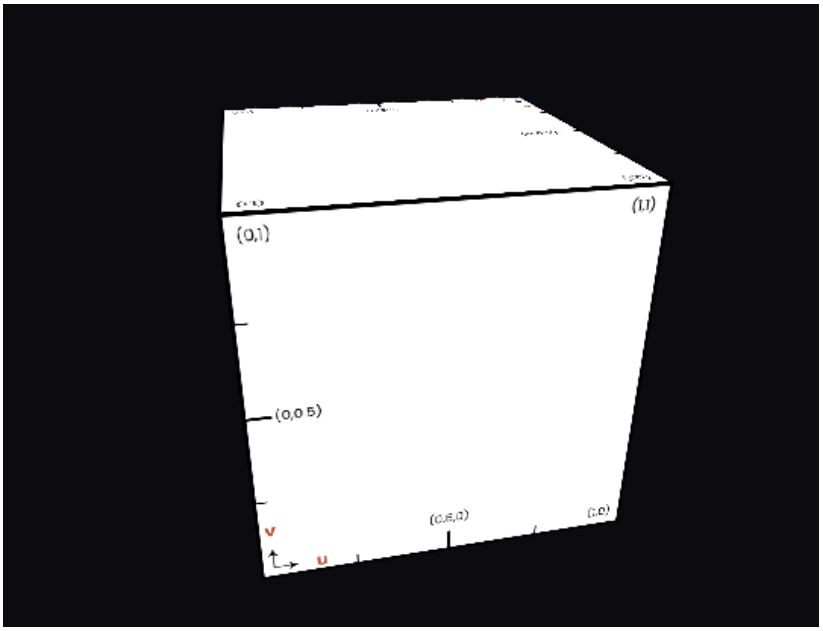
```
    img.src = imagePath;
  } else if (index === 1) {
    if (this.texture1 === null) {
      this.texture1 = gl.createTexture();
    }

    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1);

    const uTexture1 = gl.getUniformLocation(gl.program, "uTexture1");
    if (!uTexture1) console.error("Could not get uniform");

    const img = new Image();

    img.onload = () => {
      gl.activeTexture(gl.TEXTURE1);

      gl.bindTexture(gl.TEXTURE_2D, this.texture1);
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
      gl.texImage2D(
        gl.TEXTURE_2D,
        0,
        gl.RGBA,
        gl.RGBA,
        gl.UNSIGNED_BYTE,
        img
      );

      gl.uniform1i(uTexture1, 1); // NOTE THIS MATCHES THE TEXTURE SLOT #
    };

    img.crossOrigin = "anonymous";
    img.src = imagePath;
  }
```

i. Notably, we are changing all references of texture0 to texture1

e. Now change our first call to the function in `index.js` and add the third parameter `index = 0`, and make a second call with the new image and `index = 1`:

```
71    const cube = new Cube();
72    cube.setImage(gl, uvImg, 0);
73    cube.setImage(gl, diceImg, 1);
```

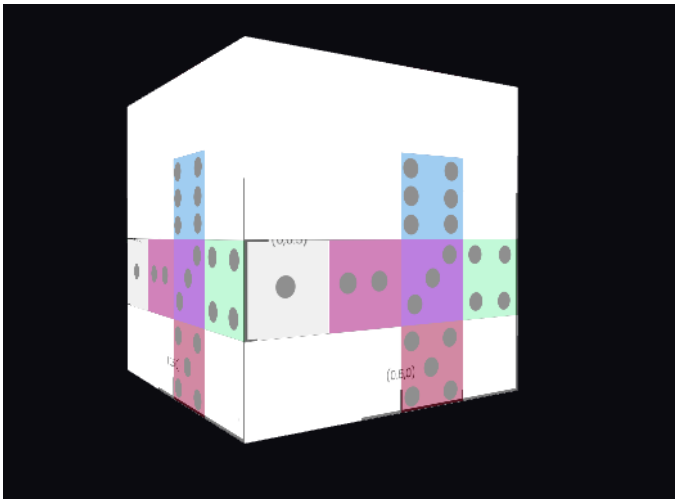
f. If everything went well, you should still have this:



g. Let's get our second image on there. Change the fragment shader to sample the second image and `mix` the two together (*note that the* `mix()` *function performs a linear interpolation on the first two parameters with the third parameter being the progress from the first to the second. Play with the third value to understand this better*):

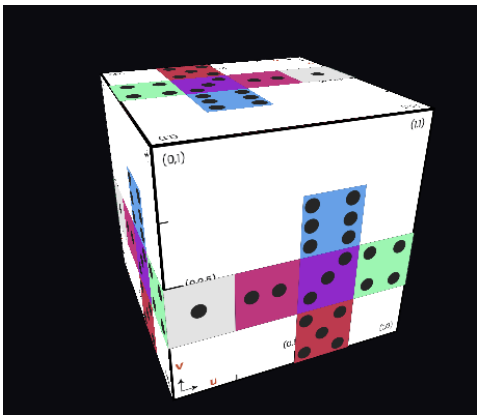
```
// Fragment shader program
const FSHADER_SOURCE = `
  #ifdef GL_ES
  precision mediump float;
  #endif

  uniform sampler2D uTexture0;
  uniform sampler2D uTexture1;

  varying vec2 vUv;

  void main() {
    vec4 image0 = texture2D(uTexture0, vUv);
    vec4 image1 = texture2D(uTexture1, vUv);

    vec4 color = mix(image0, image1, 0.5);

    gl_FragColor = color;
  }
  `;
```

h. You should see something like this:



i. In order to see both images more clearly, I would suggest mixing them as such:

```
53  void main() {
54    vec4 image0 = texture2D(uTexture0, vUv);
55    vec4 image1 = texture2D(uTexture1, vUv);
56
57    vec3 color = mix(image0.rgb, image1.rgb, image1.a);
58
59    gl_FragColor = vec4(color, 1.0);
60  }
```



j. This is a good example of how we can use the vectors available to us in the shader programs.
   i. Note we use `.rgb` rather than `.xyz` on line 57. These are interchangeable and are semantic only, meaning, using `.rgb` just indicates that it is a color. The `.a` portion of a vector is the fourth element (while r,g and b are the first, second and third) and it represents the alpha of the image here.
   ii. Since the dice image is fully transparent in the background, we can use it's alpha value in our mix() function such that:
      1. When image1.a is 1, we show image1 on that 'fragment'
      2. When image1.a is 0, we show image0 instead
         a. Remember, this is just a linear interpolation. Refer to the formula in the book(s) if this doesn't quite make sense.
   iii. This will effectively overlay the dice texture on top of our coordinates texture.

Altering Our UV-Coordinates

Now, let's alter our UV-coordinates such that we can map the dice faces onto each face of our cube.

1. Head into the Cube.js file and go to the setUvs() function.
2. It might be easier to do this without the cube auto-rotating, so you can turn that off by commenting out the line in the `tick()` function in `index.js`.
3. Now, let's get the die face representing 1 on the top of our cube
   a. Notice where this face is on our coordinates texture. The top left corner is at (0, 0.5), the bottom left corner is at (0, 0.25) and the top and bottom right corners are at (0.25, 0.5) and (0.25, 0.25).

b. Our current UV-coordinates map these corners just like the coordinates texture shows: top left is (0, 1) top right is (1, 1) etc etc. So, we just need to change our top faces UV coordinates from these to the ones we want for the face of the die. Let's look at the given UV's:

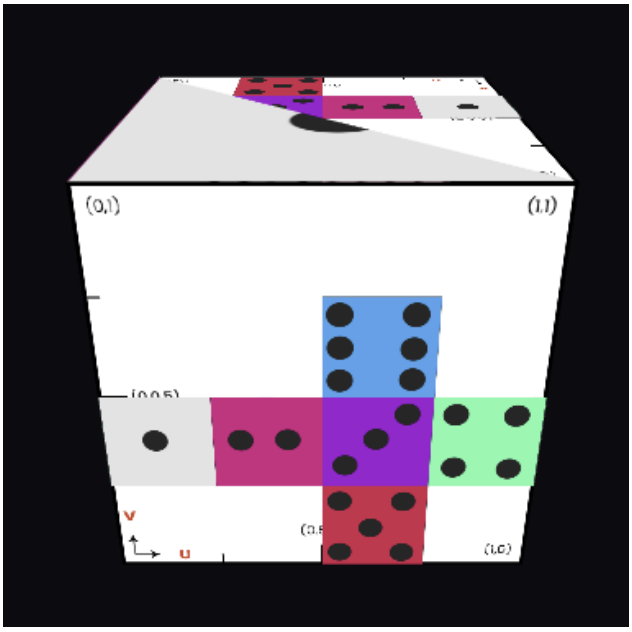
```
setUvs() {
  // prettier-ignore
  this.uvs = new Float32Array([
    // FRONT
    0,1, 0,0, 1,0, 0,1, 1,0, 1,1,
    // LEFT
    0,1, 0,0, 1,0, 0,1, 1,0, 1,1,
    // RIGHT
    0,1, 0,0, 1,0, 0,1, 1,0, 1,1,
    // TOP
    1,0, 1,1, 0,1, 1,0, 0,1, 0,0,
    // BACK
    0,1, 0,0, 1,1, 1,1, 0,0, 1,0,
    // BOTTOM
    0,1, 0,0, 1,0, 0,1, 1,0, 1,1,
  ]);
}
```

c. Change the first 3 sets of UV's (the first of the two triangles) to our new, desired UV's:

```
// TOP
0.25,0.25, 0.25,0.5, 0,0.5, 1,0, 0,1, 0,0,
```

i. See how (1,0) -> (0.25,0.25) and (1,1) -> (0.25,0.5) and (0,1) -> (0, 0.5)

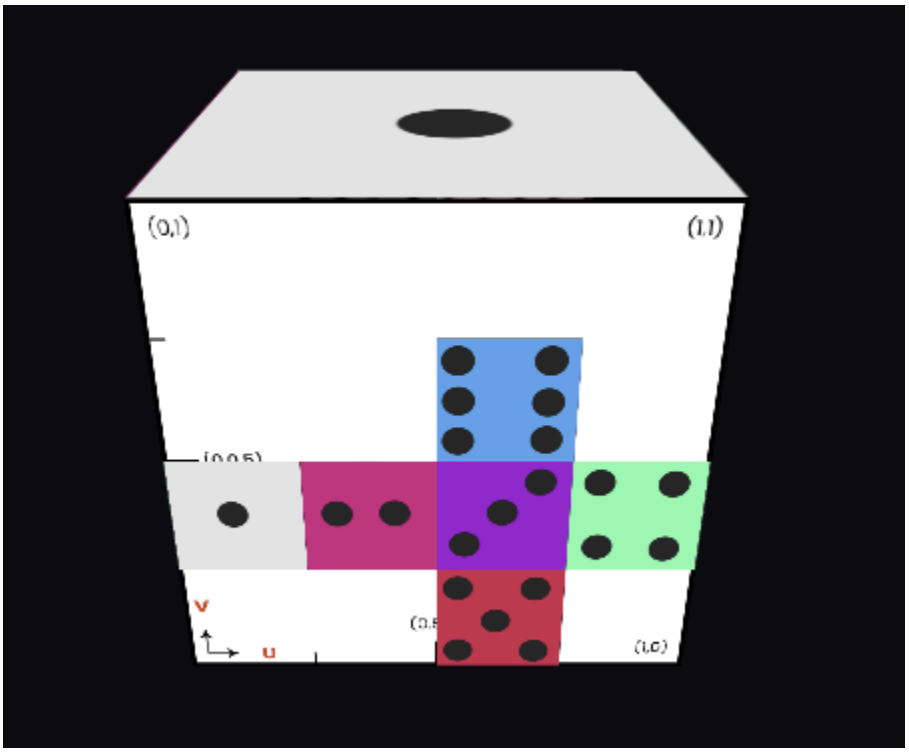
d. You should see the first triangle has our desired face on it now:



e. Do this for the last three UV's as well:

```
// TOP
0.25,0.25, 0.25,0.5, 0,0.5, 0.25,0.25, 0,0.5, 0,0.25,
```

And you should get this:

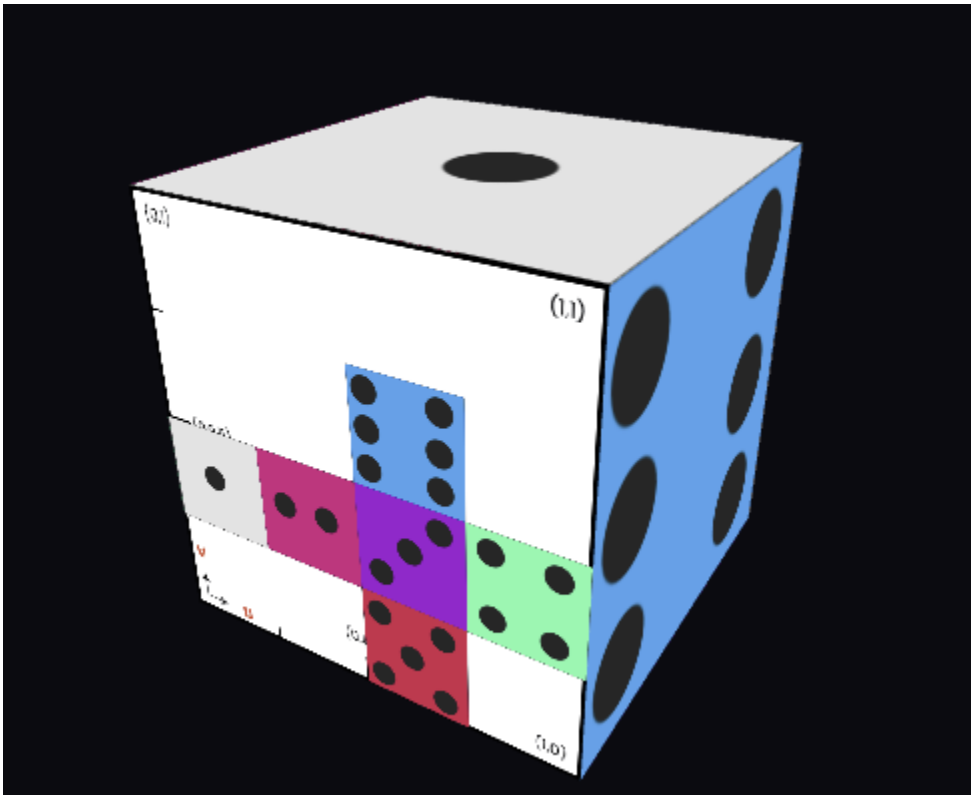


4. Now let's get the blue face onto the right side of our cube:

a. Map the UV's to the new ones:

```
// RIGHT
0.5,0.75, 0.5,0.5, 0.75,0.5, 0.5,0.75, 0.75,0.5, 0.75,0.75,
```

b. Rotate your cube and you should see this:



5. Get the rest of the sides mapped and you should get a proper 6-sided dice! Play around with adding some more, if you want or maybe play with the colors in the shader: