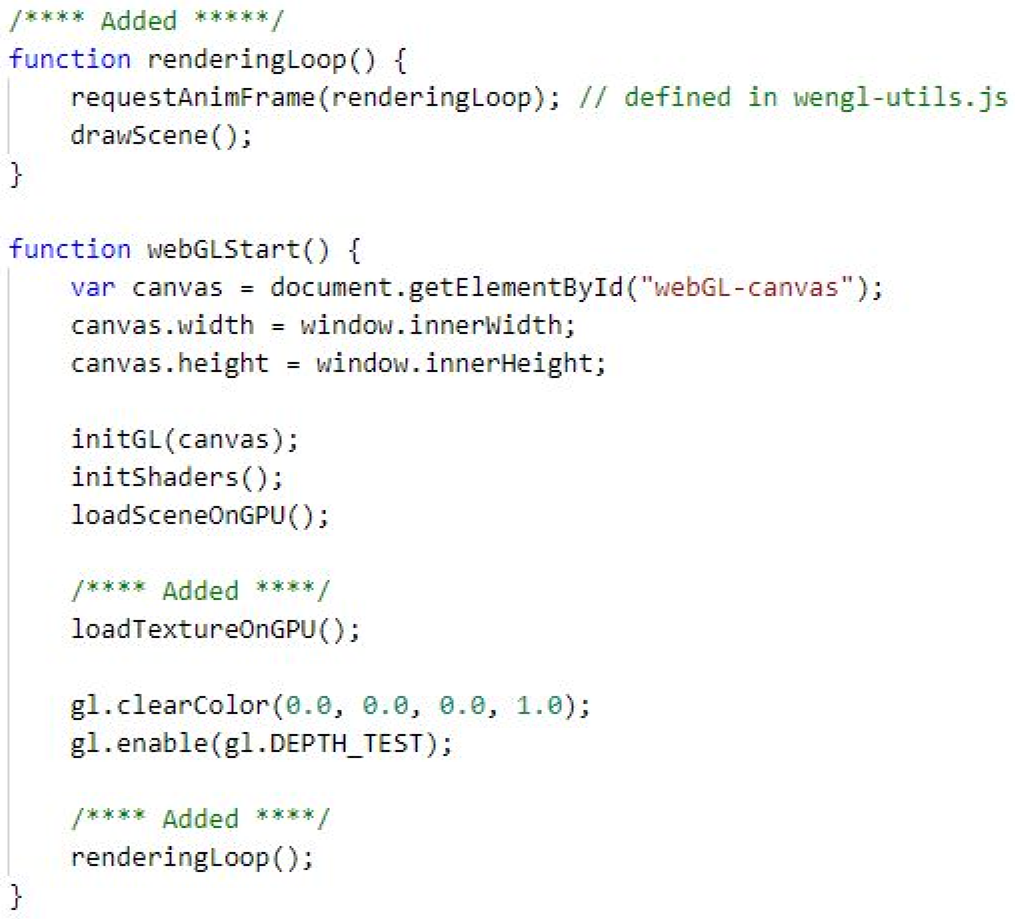
Slides recap...

Additions in the webGLStart() function

Add the new proposed code to have you OpenGL canvas rendered several times per frame. With the following additions, your function *drawScene()* will be called continuously:



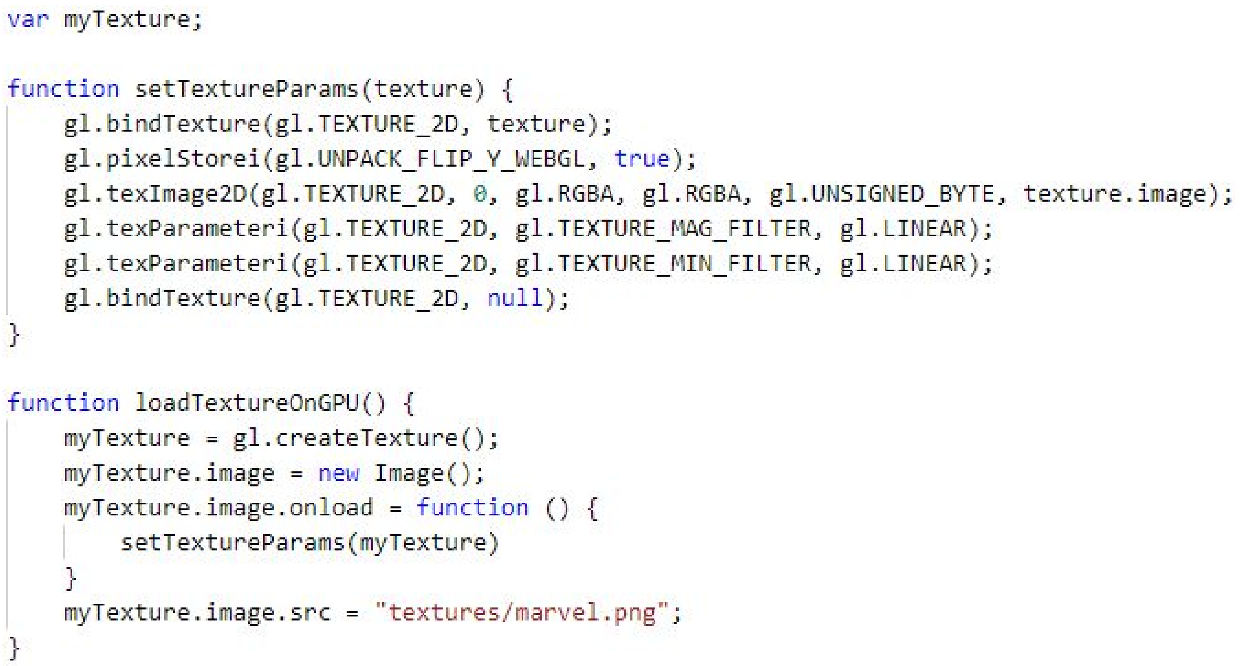
```
/**** Added *****/
function renderingLoop() {
    requestAnimFrame(renderingLoop); // defined in wengl-utils.js
    drawScene();
}

function webGLStart() {
    var canvas = document.getElementById("webGL-canvas");
    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;

    initGL(canvas);
    initShaders();
    loadSceneOnGPU();

    /**** Added ****/
    loadTextureOnGPU();

    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);

    /**** Added ****/
    renderingLoop();
}
```

Texture loading

In the previous code, you will see that a new function *loadTextureOnGPU()* is called from *webGLStart()*. This function will load the texture we will work with on the GPU. Now we will need to add several things:

a. The implementation of that function.
b. A global variable *myTexture* in which the OpenGL texture identifier will be stored. We make it global so that we can access to it later from our *drawScene()* function.
c. A callback *setTextureParams()* that will upload the texture to the GPU when the image has finished loading from disk (this is needed because image loading is done asynchronously in a web-based html+javascript application).

To do the previously mentioned stuff, copy the following lines of code into your application, before the functions *drawScene()* and *webGLStart()* are defined.

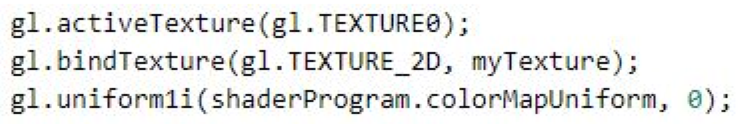


```
var myTexture;

function setTextureParams(texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, texture.image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    gl.bindTexture(gl.TEXTURE_2D, null);
}

function loadTextureOnGPU() {
    myTexture = gl.createTexture();
    myTexture.image = new Image();
    myTexture.image.onload = function () {
        setTextureParams(myTexture)
    }
    myTexture.image.src = "textures/marvel.png";
}
```

Exercise 3-a: screen-filling textured quad

Duplicate the previous Exercise 2-b *(see Figure 1)* and start from that point. Draw a textured screen-filling quad as in *Figure 2*. To do that, you will have to:

- Modify the shaders to receive texture coordinates and a 2d texture from WebGL, and use them to finally obtain per-pixel colors from the texture.
- Modify the *initShaders()* function:
  - Obtain the location of the texture coordinate attribute in the vertex shader (remove the code that obtained the previous color attribute).
  - Obtain the location of the new sampler2D uniform variable in the fragment shader.
- In *loadSceneOnGPU()*, remove the unnecessary code in the previous exercise (i.e. color vertex attributes) and add per-vertex texture coordinates attributes to the quad (create new buffer with texture-coords).
  - Actually, you could modify the previous color attribute buffer to contain texture coordinates instead. Be careful: *itemSize* will change.
- Modify *drawScene()*:
  - Send the texture to the shader with the following code (replace *colorMapUniform* by the name of your variable, given in *initShaders*):



```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, myTexture);
gl.uniform1i(shaderProgram.colorMapUniform, 0);
```

  - Each vertex of the quad we want to render will have two attributes (position and texture coordinate). So, specify both the positions and the texture coordinate buffers using calls to *bindBuffer* and *vertexAttribPointer*.

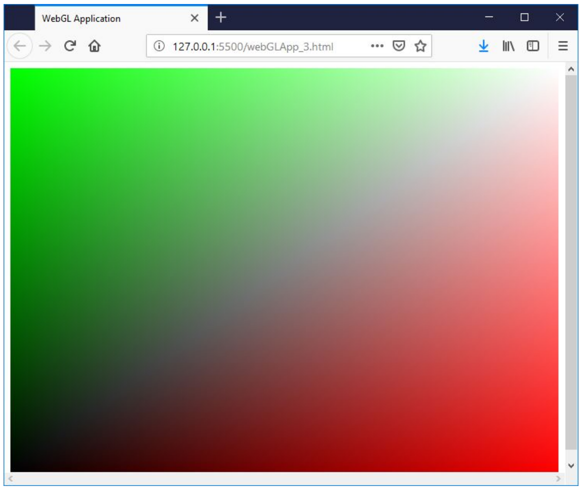


Figure 1. Exercise 2-b result

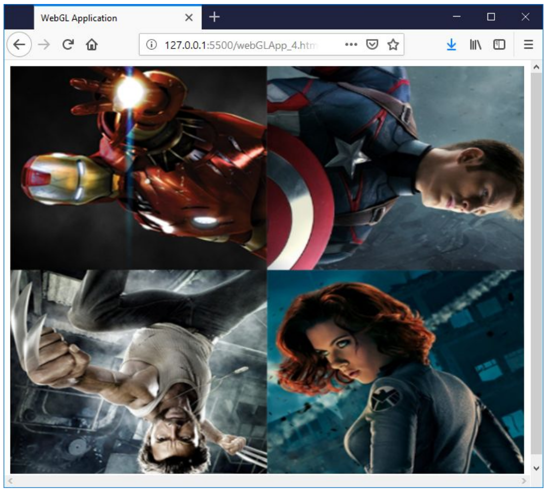


Figure 2. Exercise 3-a result

Exercise 3-b: more textured objects

Duplicate the solution of the previous exercise in your *Visual Studio Code* workspace. Extend the code accordingly to create two textured mapped shapes looking similar to those in *Figure 3*.

As you may guess, you will have to:

- Use the same shader as in the previous exercise.
- As the shader did not change, leave *initShaders()* untouched.
- We have different geometry, so modify *loadSceneOnGPU()* to create the appropriate vertex buffers.
- Modify *drawScene()* to transform and draw both shapes.

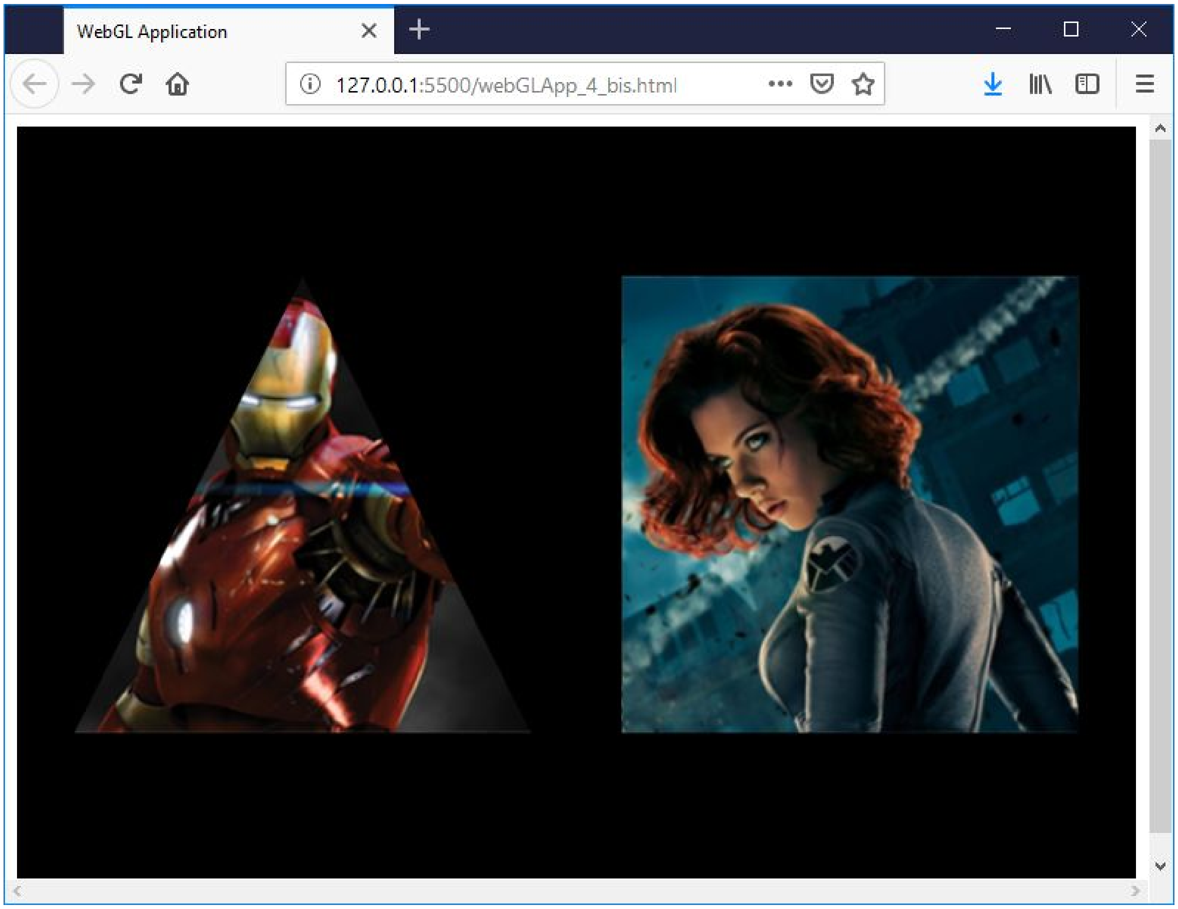


Figure 3. Two textured shapes: a triangle and a quad. Each shape has its own vertex buffers for storing vertex positions and texture coordinates.