```
// Import the shader code
import vsSource from './shaders/vertexShader.js';
import fsSource from './shaders/fragmentShader.js';

let squareRotation = 0.0;
let deltaTime = 0;

main();

// == HELPER FUNCTION FOR TASK 3a ==
function calcVertexColor(pCoordinate) {
  if (pCoordinate[1] < -8)
    return vec4.fromValues(1.0, 0.25, 0.25, 1.0);
  if (pCoordinate[1] < 9)
    return vec4.fromValues(0.25, 1.0, 0.25, 1.0);
  return vec4.fromValues(0.25, 0.25, 1.0, 1.0);
}
// ==== END HELPER FUNCTION ====

function main() {

  // Get the canvas declared in the html
  const canvas = document.getElementById("glcanvas");

  if (canvas == null) {
    alert ("Cannot instantiate canvas. Consider using vscode-preview-server.");
  }

  // Initialize the GL context
  const gl = canvas.getContext("webgl"); // use "experimental-webgl" for Edge
browser

  // Only continue if WebGL is available and working
  if (gl === null) {
    alert(
      "Unable to initialize WebGL. Your browser or machine may not support it."
    );
    return;
  }

  // We use unsigned short (16bit) to represent vertex indices. Uncomment the
line below if you want
  // to use unsigned ints (i.e., if there are more than 2^16 vertices, up to
2^32 vertices). And don't
  // forget to adjust the array type from Uint16 to Uint32 when initializing the
index buffer.
  // ...
  // gl.getExtension('OES_element_index_uint');

  // load in the mesh (this reads all vertex and face information from the .obj
file)
  var objStr = document.getElementById('decorations.obj').innerHTML;
  var mesh = new OBJ.Mesh(objStr);
  OBJ.initMeshBuffers(gl, mesh);
```

```
  // initialize the normals to zero (will overwrite later)
  for (var i = 0; i < mesh.vertices.length; i++) {
    mesh.vertexNormals[i] = 0;
  }

  // helper function to extract vertex position from array of vertices
  function vec3FromArray(vertices, index) {
    return vec3.fromValues(vertices[3*index+0], vertices[3*index+1],
vertices[3*index+2]);
  }

  // ======== TASK 2 ========

  // go through all triangles
  for (var i = 0; i < mesh.indices.length/3; i++) {
    // get vertices of this triangle
    var v0 = vec3FromArray(mesh.vertices, mesh.indices[3*i+0]);
    var v1 = vec3FromArray(mesh.vertices, mesh.indices[3*i+1]);
    var v2 = vec3FromArray(mesh.vertices, mesh.indices[3*i+2]);
    // compute edges `a` and `b`
    var a = vec3.create();
    vec3.subtract(a, v1, v0);
    var b = vec3.create();
    vec3.subtract(b, v2, v0);
    // normal is normalized cross product of edges
    var n = vec3.create();
    vec3.cross(n, a, b);
    vec3.normalize(n, n);
    // add normal to all vertex normals of this triangle
    for (var j = 0; j < 3; j++) {
      var vIndex = mesh.indices[3*i+j];
      for (var k = 0; k < 3; k++) {
        mesh.vertexNormals[3*vIndex+k] += n[k] * Math.acos(vec3.dot(a, b)); //
Note: weighting by the angle is more accurate, but not necessary in this example
(negligible effect)
      }
    }
  }

  // since we've added normals of all triangles a vertex is connected to,
  // we need to normalize the vertex normals
  for (var i = 0; i < mesh.vertexNormals.length/3; i++) {
    var n = vec3FromArray(mesh.vertexNormals, i);
    vec3.normalize(n, n);
    // and copy back to mesh normal array
    for (var k = 0; k < 3; k++) {
      mesh.vertexNormals[3*i+k] = n[k];
    }
  }
  // ====== END TASK 2 ======

  // ======== TASK 3a ========
```

```
  // initialize the color array (Hint: remember that mesh.vertices is of length
3 * number of vertices (x, y, z each))
  mesh.vertexColors = new Array(mesh.vertices.length/3*4).fill(1.0);

  // for each vertex set a custom color
  for (var i = 0; i < mesh.vertices.length/3; i++) {
    var new_color = calcVertexColor(vec3FromArray(mesh.vertices, i));
    for (var k = 0; k < 4; k++)
      mesh.vertexColors[4*i+k] = new_color[k];
  }
  // ====== END TASK 3a ======

  // Initialize a shader program; this is where all the lighting
  // for the vertices and so forth is established.
  const shaderProgram = initShaderProgram(gl, vsSource, fsSource);

  // ======== TASK 1a ========

  // Vertices
  var tmpBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, tmpBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(mesh.vertices),
gl.STATIC_DRAW);
  var location = gl.getAttribLocation(shaderProgram, "aVertexPosition");
  gl.vertexAttribPointer(location, 3, gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(location);

  // Vertex Normals
  var tmpBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, tmpBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(mesh.vertexNormals),
gl.STATIC_DRAW);
  var location = gl.getAttribLocation(shaderProgram, "aVertexNormal");
  gl.vertexAttribPointer(location, 3, gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(location);

  // Vertex Colors
  var tmpBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, tmpBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(mesh.vertexColors),
gl.STATIC_DRAW);
  var location = gl.getAttribLocation(shaderProgram, "aVertexColor");
  gl.vertexAttribPointer(location, 4, gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(location);

  // Faces (i.e., vertex indices for forming the triangles)
  var tmpBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, tmpBuffer);
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(mesh.indices),
gl.STATIC_DRAW);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, tmpBuffer);

  // ====== END TASK 1a ======
```

```
  // Draw the scene
  let then = 0;

  // Draw the scene repeatedly
  function render(now) {
    now *= 0.001; // convert to seconds
    deltaTime = now - then;
    then = now;

    gl.clearColor(0.0, 0.0, 0.0, 1.0); // Clear to black, fully opaque
    gl.clearDepth(1.0); // Clear everything
    gl.enable(gl.DEPTH_TEST); // Enable depth testing
    gl.depthFunc(gl.LEQUAL); // Near things obscure far things

    // Clear the canvas before we start drawing on it.
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    // define a light position in world coordinates
    let r = 55.0;
    var lightPos = vec4.fromValues(Math.sin(Math.PI/2) * r, Math.cos(Math.PI/2)
* r, -60.0, 1.0);

    // build the projection and model-view matrices
    const fieldOfView = (45 * Math.PI) / 180; // in radians
    const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
    const zNear = 0.1;
    const zFar = 100.0;
    const projectionMatrix = mat4.create();
    mat4.perspective(projectionMatrix, fieldOfView, aspect, zNear, zFar);

    // maps object space to world space
    // translation by (0, -25, -60), rotation by -pi/2 around x, rotation by
pi/3 around z
    const modelViewMatrix = mat4.fromValues(
           0.5,       0,  -0.86602,       0,
      -0.86602,       0,      -0.5,       0,
             0,       1,         0,       0,
             0,     -25,       -60,       1
      );

    // ======== TASK 3b ========
    mat4.rotate(
      modelViewMatrix, // destination matrix
      modelViewMatrix, // matrix to rotate
      squareRotation/2, // amount to rotate in radians
      [0, 0, 1], // axis to rotate around
    );
    // ====== END TASK 3b ======

    // ======== TASK 1b ========

    // set the shader program
    gl.useProgram(shaderProgram);
```

```
    // bind the location of the uniform variables
    gl.uniformMatrix4fv(gl.getUniformLocation(shaderProgram,
"uProjectionMatrix"), false, projectionMatrix);
    gl.uniformMatrix4fv(gl.getUniformLocation(shaderProgram,
"uModelViewMatrix"), false, modelViewMatrix);
    gl.uniform4fv(gl.getUniformLocation(shaderProgram, "uLightPos"), lightPos);

    // finally, draw the mesh
    gl.drawElements(gl.TRIANGLES, mesh.indices.length, gl.UNSIGNED_SHORT, 0);

    // ====== END TASK 1b ======

    // rotation update based on the elapsed time
    squareRotation += deltaTime;

    // render the output on the screen
    requestAnimationFrame(render);
  }

  // render the output on the screen
  requestAnimationFrame(render);
}

// Initialize a shader program, so WebGL knows how to draw our data
function initShaderProgram(gl, vsSource, fsSource) {
  const vertexShader = loadShader(gl, gl.VERTEX_SHADER, vsSource);
  const fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER, fsSource);

  // Create the shader program
  const shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  // If creating the shader program failed, alert
  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert(
      `Unable to initialize the shader program: ${gl.getProgramInfoLog(
        shaderProgram,
      )}`,
    );
    return null;
  }

  return shaderProgram;
}

//
// creates a shader of the given type, uploads the source and
// compiles it.
//
function loadShader(gl, type, source) {
  const shader = gl.createShader(type);
```

```
  // Send the source to the shader object
  gl.shaderSource(shader, source);

  // Compile the shader program
  gl.compileShader(shader);

  // See if it compiled successfully
  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert(
      `An error occurred compiling the shaders: ${gl.getShaderInfoLog(shader)}`,
    );
    gl.deleteShader(shader);
    return null;
  }

  return shader;
}
```