# Department of Computing

# CS361: Computer Graphics

# Class: BESE-4ABC

# Lab09: Shaders

# Date: 28th November, 2017

# Time: 9:00am- 12:00pm

# Lab 9: Shaders

# Introduction

Computer graphics methods are now commonly used to produce animations for a variety of application including entertainment (motion pictures and cartoons), advertising scientific and engineering, studies and training and education.

**Objectives**

After performing this lab students should be able to create basic:

Shaders

**Tools/Software Requirement**

For testing HTML 5, CSS, JS

**Description**

Helpful link: https://webglfundamentals.org/webgl/lessons/webgl-3d-textures.html

**Lab Task**

1. Rewrite and Understand the given code (file.html) and write the description of each line of the code in the document.
2. Write a vertex and fragment shaders for texture mapping.

Helpful link: https://webglfundamentals.org/webgl/lessons/webgl-3d-textures.html

**Deliverable**

Upload your code with snap shots of the output.

**Task 1**

|  |
| --- |
| <html>  <body>  <script>  /\*============= Creating a canvas ==================\*/  var canvas = document.getElementById('my\_Canvas'); //get canvas element from html  gl = canvas.getContext('experimental-webgl'); //initialize canvas as a webgl drawing context  /\*========== Defining and storing the geometry ==========\*/  var vertices = [ //Define vertices to be drawn as a JS array  -0.5,0.5,0.0,  -0.5,-0.5,0.0,  0.5,-0.5,0.0,  0.5,0.5,0.0  ];  var colors = [0,0,1, 1,0,0, 0,1,0, 1,0,1,]; //define colors for each of the vertices (3 values for rgb of each vertex)  indices = [3,2,1,3,1,0]; //define which vertices will be drawn as TRIANGLES  /\*  0\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 1  | |  | |  | |  | |  3 \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2 3, 2, 1 is a triangle and 3,1,0 is a triangle  \*/  // Create an empty buffer object and store vertex data  var vertex\_buffer = gl.createBuffer(); //creates a webgl buffer for vertices  gl.bindBuffer(gl.ARRAY\_BUFFER, vertex\_buffer); //binds that buffer to target  gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(vertices), gl.STATIC\_DRAW); //initializes and creates buffer store  gl.bindBuffer(gl.ARRAY\_BUFFER, null); //binds null to target  // Create an empty buffer object and store Index data  var Index\_Buffer = gl.createBuffer(); //creates a webgl buffer for indices  gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, Index\_Buffer); //binds that buffer to target  gl.bufferData(gl.ELEMENT\_ARRAY\_BUFFER, new Uint16Array(indices), gl.STATIC\_DRAW);//initializes and creates buffer store  gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, null); //binds null to target  // Create an empty buffer object and store color data  var color\_buffer = gl.createBuffer (); //creates buffer for color  gl.bindBuffer(gl.ARRAY\_BUFFER, color\_buffer); //binds buffer to target  gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(colors), gl.STATIC\_DRAW); //initializes and creates buffer store  /\*======================= Shaders =======================\*/  // vertex shader source code  var vertCode = 'attribute vec3 coordinates;'+ //gets data from buffer  'attribute vec3 color;'+  'varying vec3 vColor;'+ //interpolates and gives values for fragment shader  'void main(void) {' +  ' gl\_Position = vec4(coordinates, 1.0);' + //computes position  'vColor = color;'+ //assigns color to vertex  '}';  // Create a vertex shader object  var vertShader = gl.createShader(gl.VERTEX\_SHADER);  // Attach vertex shader source code  gl.shaderSource(vertShader, vertCode);  // Compile the vertex shader  gl.compileShader(vertShader);  // fragment shader source code  var fragCode = 'precision mediump float;'+ //assigns default precision as medium precision  'varying vec3 vColor;'+ //varying variable color from vertex shader  'void main(void) {'+  'gl\_FragColor = vec4(vColor, 1.);'+ //interpolates and computes fragment color  '}';  // Create fragment shader object  var fragShader = gl.createShader(gl.FRAGMENT\_SHADER);  // Attach fragment shader source code  gl.shaderSource(fragShader, fragCode);  // Compile the fragmentt shader  gl.compileShader(fragShader);  // Create a shader program object to  // store the combined shader program  var shaderProgram = gl.createProgram();  // Attach a vertex shader  gl.attachShader(shaderProgram, vertShader);  // Attach a fragment shader  gl.attachShader(shaderProgram, fragShader);  // Link both the programs  gl.linkProgram(shaderProgram);  // Use the combined shader program object  gl.useProgram(shaderProgram);  /\* ======== Associating shaders to buffer objects =======\*/  // Bind vertex buffer object  gl.bindBuffer(gl.ARRAY\_BUFFER, vertex\_buffer);  // Bind index buffer object  gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, Index\_Buffer);  // Get the attribute location  var coord = gl.getAttribLocation(shaderProgram, "coordinates");  // point an attribute to the currently bound VBO  gl.vertexAttribPointer(coord, 3, gl.FLOAT, false, 0, 0);  // Enable the attribute  gl.enableVertexAttribArray(coord);  // bind the color buffer  gl.bindBuffer(gl.ARRAY\_BUFFER, color\_buffer);  // get the attribute location  var color = gl.getAttribLocation(shaderProgram, "color");  // point attribute to the volor buffer object  gl.vertexAttribPointer(color, 3, gl.FLOAT, false,0,0) ;  // enable the color attribute  gl.enableVertexAttribArray(color);  /\*============Drawing the Quad====================\*/  // Clear the canvas  gl.clearColor(0.5, 0.5, 0.5, 0.9);  // Enable the depth test  gl.enable(gl.DEPTH\_TEST);  // Clear the color buffer bit  gl.clear(gl.COLOR\_BUFFER\_BIT);  // Set the view port  gl.viewport(0,0,canvas.width,canvas.height);  //Draw the triangle  gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED\_SHORT,0);  </script>  </body>  </html> |

**Task 2**

|  |
| --- |
| var cubeRotation = 3.14;  main();  //  // Start here  //  function main() {  const canvas = document.querySelector('#glcanvas');  const gl = canvas.getContext('webgl');  // If we don't have a GL context, give up now  if (!gl) {  alert('Unable to initialize WebGL. Your browser or machine may not support it.');  return;  }  // Vertex shader program  const vsSource = `  attribute vec4 aVertexPosition;  attribute vec2 aTextureCoord;  uniform mat4 uModelViewMatrix;  uniform mat4 uProjectionMatrix;  varying highp vec2 vTextureCoord;  void main(void) {  gl\_Position = uProjectionMatrix \* uModelViewMatrix \* aVertexPosition;  vTextureCoord = aTextureCoord;  }  `;  // Fragment shader program  const fsSource = `  varying highp vec2 vTextureCoord;  uniform sampler2D uSampler;  void main(void) {  gl\_FragColor = texture2D(uSampler, vTextureCoord);  }  `;  // Initialize a shader program; this is where all the lighting  // for the vertices and so forth is established.  const shaderProgram = initShaderProgram(gl, vsSource, fsSource);  // Collect all the info needed to use the shader program.  // Look up which attributes our shader program is using  // for aVertexPosition, aTextureCoord and also  // look up uniform locations.  const programInfo = {  program: shaderProgram,  attribLocations: {  vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition'),  textureCoord: gl.getAttribLocation(shaderProgram, 'aTextureCoord'),  },  uniformLocations: {  projectionMatrix: gl.getUniformLocation(shaderProgram, 'uProjectionMatrix'),  modelViewMatrix: gl.getUniformLocation(shaderProgram, 'uModelViewMatrix'),  uSampler: gl.getUniformLocation(shaderProgram, 'uSampler'),  },  };  // Here's where we call the routine that builds all the  // objects we'll be drawing.  const buffers = initBuffers(gl);  const texture = loadTexture(gl, 'cubetexture.png');  var then = 0;  // Draw the scene repeatedly  function render(now) {  now \*= 0.001; // convert to seconds  const deltaTime = now - then;  then = now;  drawScene(gl, programInfo, buffers, texture, deltaTime);  requestAnimationFrame(render);  }  requestAnimationFrame(render);  }  //  // initBuffers  //  // Initialize the buffers we'll need. For this demo, we just  // have one object -- a simple three-dimensional cube.  //  function initBuffers(gl) {  // Create a buffer for the cube's vertex positions.  const positionBuffer = gl.createBuffer();  // Select the positionBuffer as the one to apply buffer  // operations to from here out.  gl.bindBuffer(gl.ARRAY\_BUFFER, positionBuffer);  // Now create an array of positions for the cube.  const positions = [  // Front face  -1.0, -1.0, 1.0,  1.0, -1.0, 1.0,  1.0, 1.0, 1.0,  -1.0, 1.0, 1.0,  // Back face  -1.0, -1.0, -1.0,  -1.0, 1.0, -1.0,  1.0, 1.0, -1.0,  1.0, -1.0, -1.0,  // Top face  -1.0, 1.0, -1.0,  -1.0, 1.0, 1.0,  1.0, 1.0, 1.0,  1.0, 1.0, -1.0,  // Bottom face  -1.0, -1.0, -1.0,  1.0, -1.0, -1.0,  1.0, -1.0, 1.0,  -1.0, -1.0, 1.0,  // Right face  1.0, -1.0, -1.0,  1.0, 1.0, -1.0,  1.0, 1.0, 1.0,  1.0, -1.0, 1.0,  // Left face  -1.0, -1.0, -1.0,  -1.0, -1.0, 1.0,  -1.0, 1.0, 1.0,  -1.0, 1.0, -1.0,  ];  // Now pass the list of positions into WebGL to build the  // shape. We do this by creating a Float32Array from the  // JavaScript array, then use it to fill the current buffer.  gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(positions), gl.STATIC\_DRAW);  // Now set up the texture coordinates for the faces.  const textureCoordBuffer = gl.createBuffer();  gl.bindBuffer(gl.ARRAY\_BUFFER, textureCoordBuffer);  const textureCoordinates = [  // Front  0.0, 0.0,  1.0, 0.0,  1.0, 1.0,  0.0, 1.0,  // Back  0.0, 0.0,  1.0, 0.0,  1.0, 1.0,  0.0, 1.0,  // Top  0.0, 0.0,  1.0, 0.0,  1.0, 1.0,  0.0, 1.0,  // Bottom  0.0, 0.0,  1.0, 0.0,  1.0, 1.0,  0.0, 1.0,  // Right  0.0, 0.0,  1.0, 0.0,  1.0, 1.0,  0.0, 1.0,  // Left  0.0, 0.0,  1.0, 0.0,  1.0, 1.0,  0.0, 1.0,  ];  gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(textureCoordinates),  gl.STATIC\_DRAW);  // Build the element array buffer; this specifies the indices  // into the vertex arrays for each face's vertices.  const indexBuffer = gl.createBuffer();  gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, indexBuffer);  // This array defines each face as two triangles, using the  // indices into the vertex array to specify each triangle's  // position.  const indices = [  0, 1, 2, 0, 2, 3, // front  4, 5, 6, 4, 6, 7, // back  8, 9, 10, 8, 10, 11, // top  12, 13, 14, 12, 14, 15, // bottom  16, 17, 18, 16, 18, 19, // right  20, 21, 22, 20, 22, 23, // left  ];  // Now send the element array to GL  gl.bufferData(gl.ELEMENT\_ARRAY\_BUFFER,  new Uint16Array(indices), gl.STATIC\_DRAW);  return {  position: positionBuffer,  textureCoord: textureCoordBuffer,  indices: indexBuffer,  };  }  //  // Initialize a texture and load an image.  // When the image finished loading copy it into the texture.  //  function loadTexture(gl, url) {  const texture = gl.createTexture();  gl.bindTexture(gl.TEXTURE\_2D, texture);  // Because images have to be download over the internet  // they might take a moment until they are ready.  // Until then put a single pixel in the texture so we can  // use it immediately. When the image has finished downloading  // we'll update the texture with the contents of the image.  const level = 0;  const internalFormat = gl.RGBA;  const width = 1;  const height = 1;  const border = 0;  const srcFormat = gl.RGBA;  const srcType = gl.UNSIGNED\_BYTE;  const pixel = new Uint8Array([0, 0, 255, 255]); // opaque blue  gl.texImage2D(gl.TEXTURE\_2D, level, internalFormat,  width, height, border, srcFormat, srcType,  pixel);  const image = new Image();  image.onload = function() {  gl.bindTexture(gl.TEXTURE\_2D, texture);  gl.texImage2D(gl.TEXTURE\_2D, level, internalFormat,  srcFormat, srcType, image);  // WebGL1 has different requirements for power of 2 images  // vs non power of 2 images so check if the image is a  // power of 2 in both dimensions.  if (isPowerOf2(image.width) && isPowerOf2(image.height)) {  // Yes, it's a power of 2. Generate mips.  gl.generateMipmap(gl.TEXTURE\_2D);  } else {  // No, it's not a power of 2. Turn of mips and set  // wrapping to clamp to edge  gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_S, gl.CLAMP\_TO\_EDGE);  gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_T, gl.CLAMP\_TO\_EDGE);  gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.LINEAR);  }  };  image.src = url;  return texture;  }  function isPowerOf2(value) {  return (value & (value - 1)) == 0;  }  //  // Draw the scene.  //  function drawScene(gl, programInfo, buffers, texture, deltaTime) {  gl.clearColor(0.5, 0.5, 0.5, 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);  // Create a perspective matrix, a special matrix that is  // used to simulate the distortion of perspective in a camera.  // Our field of view is 45 degrees, with a width/height  // ratio that matches the display size of the canvas  // and we only want to see objects between 0.1 units  // and 100 units away from the camera.  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();  // note: glmatrix.js always has the first argument  // as the destination to receive the result.  mat4.perspective(projectionMatrix,  fieldOfView,  aspect,  zNear,  zFar);  // Set the drawing position to the "identity" point, which is  // the center of the scene.  const modelViewMatrix = mat4.create();  // Now move the drawing position a bit to where we want to  // start drawing the square.  mat4.translate(modelViewMatrix, // destination matrix  modelViewMatrix, // matrix to translate  [-0.0, 0.0, -6.0]); // amount to translate  mat4.rotate(modelViewMatrix, // destination matrix  modelViewMatrix, // matrix to rotate  cubeRotation, // amount to rotate in radians  [0, 0, 1]); // axis to rotate around (Z)  mat4.rotate(modelViewMatrix, // destination matrix  modelViewMatrix, // matrix to rotate  cubeRotation \* .7,// amount to rotate in radians  [0, 1, 0]); // axis to rotate around (X)  // Tell WebGL how to pull out the positions from the position  // buffer into the vertexPosition attribute  {  const numComponents = 3;  const type = gl.FLOAT;  const normalize = false;  const stride = 0;  const offset = 0;  gl.bindBuffer(gl.ARRAY\_BUFFER, buffers.position);  gl.vertexAttribPointer(  programInfo.attribLocations.vertexPosition,  numComponents,  type,  normalize,  stride,  offset);  gl.enableVertexAttribArray(  programInfo.attribLocations.vertexPosition);  }  // Tell WebGL how to pull out the texture coordinates from  // the texture coordinate buffer into the textureCoord attribute.  {  const numComponents = 2;  const type = gl.FLOAT;  const normalize = false;  const stride = 0;  const offset = 0;  gl.bindBuffer(gl.ARRAY\_BUFFER, buffers.textureCoord);  gl.vertexAttribPointer(  programInfo.attribLocations.textureCoord,  numComponents,  type,  normalize,  stride,  offset);  gl.enableVertexAttribArray(  programInfo.attribLocations.textureCoord);  }  // Tell WebGL which indices to use to index the vertices  gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, buffers.indices);  // Tell WebGL to use our program when drawing  gl.useProgram(programInfo.program);  // Set the shader uniforms  gl.uniformMatrix4fv(  programInfo.uniformLocations.projectionMatrix,  false,  projectionMatrix);  gl.uniformMatrix4fv(  programInfo.uniformLocations.modelViewMatrix,  false,  modelViewMatrix);  // Specify the texture to map onto the faces.  // Tell WebGL we want to affect texture unit 0  gl.activeTexture(gl.TEXTURE0);  // Bind the texture to texture unit 0  gl.bindTexture(gl.TEXTURE\_2D, texture);  // Tell the shader we bound the texture to texture unit 0  gl.uniform1i(programInfo.uniformLocations.uSampler, 0);  {  const vertexCount = 36;  const type = gl.UNSIGNED\_SHORT;  const offset = 0;  gl.drawElements(gl.TRIANGLES, vertexCount, type, offset);  }  // Update the rotation for the next draw  // cubeRotation += deltaTime;  }  //  // 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;  } |

