Tutorial 6 Draw in 3D

This session practices drawing a simple 3D scene (Figure a). The scene consists of the floor, a table and a cube. The floor is a plane. The cube is of size 2x2x2 and the origin of its local coordinate system has been chosen at the centre of the cube, as shown in Figure b. The table is constructed from 5 box-like objects: four for the legs and one for the tabletop, which are obtained by ***scaling*** the cube.

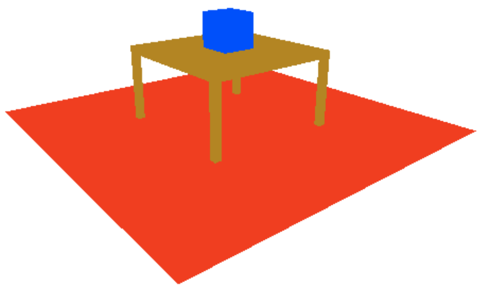
Figure a

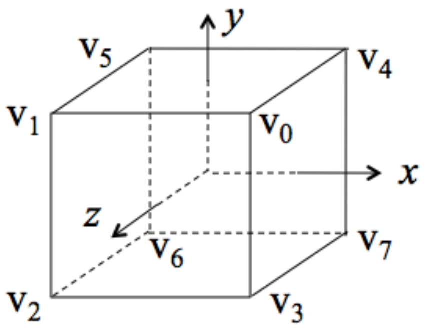

Figure b

To position and view objects in a 3D scene, we need to use the ***modelview*** and ***perspective projection*** matrices. To use them, in the vertex shader, two variables of type 4x4 matrix are declared as `uniform`, which is a type in OpenGL Shading Language for storing data that are ***constant for all vertices*** of *an object* (modelview) or the *entire scene* (perspective projection).

```
      uniform mat4 uMVMatrix;
      uniform mat4 uPMatrix;
```

where `uPMatrix` is for storing the perspective projection matrix and `uMVMatrix` is for the *modelview* matrix of an object. Notice that each object in the scene will have its own *modelview* matrix. These matrices will be updated at runtime by loading to them the current *modelview* matrix of each object and the perspective projection matrix of the scene view (if the position of camera changes, as needed by interactive navigation control). The current modelview and perspective project matrices need to be stored in the main WebGL program as two global variables. In this tutorial, they are stored in `modelViewMatrix` and `projectionMatrix`, respectively. The values for these two global matrices are uploaded to the vertex shader in the function `draw()` before drawing an object.

In this tutorial, the projection matrix is uploaded once, because we have a fixed viewing angle. The perspective projection matrix is set up by matrix operation **`mat4.perspective(fovy, aspect, near, far, dest)`**, where **`fovy`** is the vertical field of view in degrees, **`aspect`** is the aspect ratio (which equals viewport width/height), **`near`** and **`far`** are the near and far planes of the view frustum, and **`dest`** is the destination matrix which the result will be written into.

```
      mat4.perspective(60, gl.viewportWidth / gl.viewportHeight,
                       0.1, 100.0, projectionMatrix);
```

The modelview matrix must be set before drawing an object. Each object has its own modelview matrix. The initial value of the modelview matrix is set with the camera transformation using **`mat4.lookAt(eye, center, up, dest)`**. The function `lookAt()` generates a look-at matrix with the given eye position **`eye`**, (the position of the camera), the point **`centre`** (the position the camera is looking at), and the coordinate axis pointing upwards, **`up`**. The result is written into **`dest`**.

```
      mat4.lookAt([8, 5, -10],[0, 0, 0], [0, 1,0], modelViewMatrix);
```

To keep the initial value of the modelview matrix for later use by other objects, a JavaScript array `modelViewMatrixStack` is used to store the current modelview matrix before it being modified to suit the current object. After finishing drawing the current object with the appropriate modelview matrix, the stored original modelview matrix is retrieved (by popping it out of the stack) for it to be modified for the next object. For example, the floor plane is drawn first using the initial modelview matrix, because we want to place the plane at the origin of the world coordinate system. The next item to be drawn is the tabletop, but we do not want to place it at the origin. Therefore, the modelview matrix has to be modified to reflect the positional change (i.e, a translation). Before modifying the current modelview matrix for the tabletop, a copy of current modelview matrix is saved in `modelViewMatrixStack`. Then it is modified by a translation (0.0, 1.0, 0.0) to put the tabletop at the correct position and by a scale (2.0, 0.1, 2.0) to give it the required tabletop shape/dimensions. Then resulted modelview matrix is uploaded to the shader for it to be applied to the vertices of the tabletop. After drawing the tabletop, the stored intact modelview matrix is retrieved from `modelViewMatrixStack` for it to be modified for the next object, e.g., a table leg:

```
pushModelViewMatrix();
mat4.translate(modelViewMatrix, [0.0, 1.0, 0.0],
                                modelViewMatrix);
mat4.scale(modelViewMatrix, [2.0, 0.1, 2.0], modelViewMatrix);
uploadModelViewMatrixToShader();
drawCube(0.72, 0.53, 0.04, 1.0); // arguments set brown color
popModelViewMatrix();
```

Exercise: Copy `glMatrix.js` into your working directory if you have not done so. Copy the unfinished program on Moodle and modify it to have the features in the following program list. Pay attention to the statements/functions shown in the bold case.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title> Tutorial 6  Drawing In 3D </title>
<meta charset="utf-8">

<script src="glMatrix.js"></script>

<script src="webgl-debug.js"></script>


<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  varying vec4 vColor;

  void main() {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vColor = aVertexColor;
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec4 vColor;

  void main() {
    gl_FragColor = vColor;
  }
</script>

<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;

var floorVertexPositionBuffer;
var floorVertexIndexBuffer;
var cubeVertexPositionBuffer;
var cubeVertexIndexBuffer;

var modelViewMatrix;
var projectionMatrix;
var modelViewMatrixStack;

function createGLContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i < names.length; i++) {
    try {
      context = canvas.getContext(names[i]);
    } catch(e) {}
    if (context) {
      break;
    }
  }

  if (context) {
    context.viewportWidth = canvas.width;
    context.viewportHeight = canvas.height;
  } else {
    alert("Failed to create WebGL context!");
  }
  return context;
}

function loadShaderFromDOM(id) {
  var shaderScript = document.getElementById(id);
  if (!shaderScript) {
    return null;
  }
  var shaderSource = "";
  var currentChild = shaderScript.firstChild;
  while (currentChild) {
    if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
      shaderSource += currentChild.textContent;
    }
    currentChild = currentChild.nextSibling;
  }

  var shader;

  if (shaderScript.type == "x-shader/x-fragment") {
    shader = gl.createShader(gl.FRAGMENT_SHADER);
  } else if (shaderScript.type == "x-shader/x-vertex") {
    shader = gl.createShader(gl.VERTEX_SHADER);
  } else {
    return null;
  }

  gl.shaderSource(shader, shaderSource);
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(shader));
    return null;
  }
  return shader;
}

function setupShaders() {
  var vertexShader = loadShaderFromDOM("shader-vs");
  var fragmentShader = loadShaderFromDOM("shader-fs");

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Failed to setup shaders");
  }

  gl.useProgram(shaderProgram);
  shaderProgram.vertexPositionAttribute =
            gl.getAttribLocation(shaderProgram, "aVertexPosition");
  shaderProgram.vertexColorAttribute =
            gl.getAttribLocation(shaderProgram, "aVertexColor");

  shaderProgram.uniformMVMatrix =
            gl.getUniformLocation(shaderProgram, "uMVMatrix");
  shaderProgram.uniformProjMatrix =
            gl.getUniformLocation(shaderProgram, "uPMatrix");

  // Initialise the matrices
  modelViewMatrix = mat4.create();
  projectionMatrix = mat4.create();
  modelViewMatrixStack = [];

  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}

function pushModelViewMatrix() {
  var copyToPush = mat4.create(modelViewMatrix);
  modelViewMatrixStack.push(copyToPush);
}

function popModelViewMatrix() {
  if (modelViewMatrixStack.length == 0) {
    throw "Error popModelViewMatrix() - Stack was empty ";
  }
  modelViewMatrix = modelViewMatrixStack.pop();
}

function setupFloorBuffers() {
  floorVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, floorVertexPositionBuffer);

  var floorVertexPosition = [
      // Plane in y=0
       5.0,   0.0,  5.0,  //v0
       5.0,   0.0, -5.0,  //v1
      -5.0,   0.0, -5.0,  //v2
      -5.0,   0.0,  5.0]; //v3

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(floorVertexPosition),
                      gl.STATIC_DRAW);
  floorVertexPositionBuffer.itemSize = 3;
  floorVertexPositionBuffer.numberOfItems = 4;

  floorVertexIndexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, floorVertexIndexBuffer);

  var floorVertexIndices = [0, 1, 2, 3];
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
                      Uint16Array(floorVertexIndices),gl.STATIC_DRAW);
  floorVertexIndexBuffer.itemSize = 1;
  floorVertexIndexBuffer.numberOfItems = 4;
}

function setupCubeBuffers() {
  cubeVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);

  var cubeVertexPosition = [
       1.0,  1.0,  1.0, //v0
      -1.0,  1.0,  1.0, //v1
      -1.0, -1.0,  1.0, //v2
       1.0, -1.0,  1.0, //v3
       1.0,  1.0, -1.0, //v4
      -1.0,  1.0, -1.0, //v5
      -1.0, -1.0, -1.0, //v6
       1.0, -1.0, -1.0, //v7
      ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(cubeVertexPosition),
                      gl.STATIC_DRAW);

  cubeVertexPositionBuffer.itemSize = 3;
  cubeVertexPositionBuffer.numberOfItems = 8;

  cubeVertexIndexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);

  //For simplicity, each face will be drawn as gl_TRIANGLES, therefore
  //the indices for each triangle are specified.
  var cubeVertexIndices = [
            0, 1, 2,    0, 2, 3,    // Front face
            4, 6, 5,    4, 7, 6,    // Back face
            1, 5, 6,    1, 6, 2,    //left
            0, 3, 7,    0, 7, 4,    //right
            0, 5, 1,    0, 4, 5,    //top
            3, 2, 6,    3, 6, 7     //bottom
        ];

  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(cubeVertexIndices),
                      gl.STATIC_DRAW);
  cubeVertexIndexBuffer.itemSize = 1;
  cubeVertexIndexBuffer.numberOfItems = 36;
}

function setupBuffers() {
  setupFloorBuffers();
  setupCubeBuffers();
}



function uploadModelViewMatrixToShader() {
  gl.uniformMatrix4fv(shaderProgram.uniformMVMatrix, false,
                      modelViewMatrix);
}

function uploadProjectionMatrixToShader() {
  gl.uniformMatrix4fv(shaderProgram.uniformProjMatrix,
                      false, projectionMatrix);
}

function drawFloor(r,g,b,a) {
  // Disable vertex attrib array and use constant color for the floor.
  gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
  // Set colour
  gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, r, g, b, a);

  // Draw the floor
  gl.bindBuffer(gl.ARRAY_BUFFER, floorVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                        floorVertexPositionBuffer.itemSize,
                        gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, floorVertexIndexBuffer);
  gl.drawElements(gl.TRIANGLE_FAN, floorVertexIndexBuffer.numberOfItems,
                        gl.UNSIGNED_SHORT, 0);
}

function drawCube(r,g,b,a) {
  // Disable vertex attrib array and use constant color for the cube.
  gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);
  // Set color
  gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, r, g, b, a);
  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                        cubeVertexPositionBuffer.itemSize,
                        gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);
  gl.drawElements(gl.TRIANGLES, cubeVertexIndexBuffer.numberOfItems,
                        gl.UNSIGNED_SHORT, 0);
}

function drawTable(){
  // Draw table top
  pushModelViewMatrix();
  mat4.translate(modelViewMatrix, [0.0, 1.0, 0.0], modelViewMatrix);
  mat4.scale(modelViewMatrix, [2.0, 0.1, 2.0], modelViewMatrix);
  uploadModelViewMatrixToShader();
  // Draw the scaled cube
  drawCube(0.72, 0.53, 0.04, 1.0); // brown color
  popModelViewMatrix();

  // Draw table legs
  for (var i=-1; i<=1; i+=2) {
    for (var j= -1; j<=1; j+=2) {
      pushModelViewMatrix();
      mat4.translate(modelViewMatrix, [i*1.9, -0.1, j*1.9],
                        modelViewMatrix);
      mat4.scale(modelViewMatrix, [0.1, 1.0, 0.1], modelViewMatrix);
      uploadModelViewMatrixToShader();
      drawCube(0.72, 0.53, 0.04, 1.0); // argument sets brown colour
      popModelViewMatrix();
    }
  }
}

function draw() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  mat4.perspective(60, gl.viewportWidth / gl.viewportHeight,
                        0.1, 100.0, projectionMatrix);
  mat4.identity(modelViewMatrix);
  mat4.lookAt([8, 5, -10],[0, 0, 0], [0, 1,0], modelViewMatrix);
  uploadModelViewMatrixToShader();
  uploadProjectionMatrixToShader();

  // Draw floor in red color
  drawFloor(1.0, 0.0, 0.0, 1.0);

  // Draw table
  pushModelViewMatrix();
  mat4.translate(modelViewMatrix, [0.0, 1.1, 0.0], modelViewMatrix);
  uploadModelViewMatrixToShader();
  drawTable(); // Call drawTable() function
  popModelViewMatrix();

  // Draw box on top of the table
  pushModelViewMatrix();
  mat4.translate(modelViewMatrix, [0.0, 2.7 ,0.0], modelViewMatrix);
  mat4.scale(modelViewMatrix, [0.5, 0.5, 0.5], modelViewMatrix);
  uploadModelViewMatrixToShader();
  drawCube(0.0, 0.0, 1.0, 1.0);
  popModelViewMatrix()
}

function startup() {
  canvas = document.getElementById("myGLCanvas");
  gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
  setupShaders();
  setupBuffers();
  gl.clearColor(1.0, 1.0, 1.0, 1.0);
  gl.enable(gl.DEPTH_TEST);

  draw();
}
</script>
</head>

<body onload="startup();">
  <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>
</html>
```