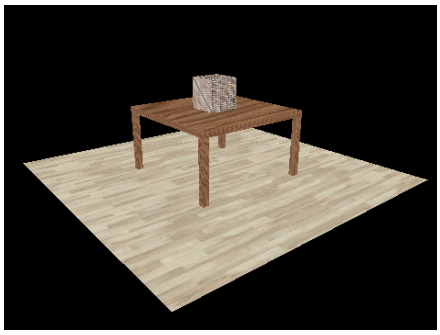
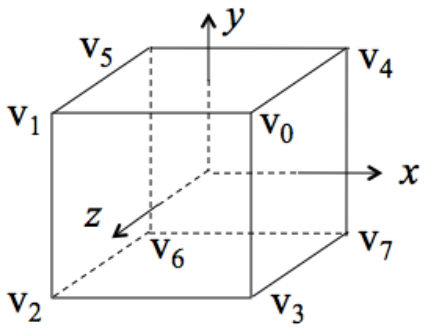
Tutorial 07 Texture Mapping

This tutorial introduces the mechanism of texture mapping in WebGL pipeline. We are going to map textures to the table scene (see last Tutorial). In addition to texture mapping, this tutorial also introduces the techniques for developing more robust application: handling of lost context. Use Firefox browser if Google Chrome does not load the texture images.

Please note how we define the cube this time. We treat the cube as 6 independent square surfaces, i.e., we use 24 pairs vertex coordinates. This is to ensure the number of vertex coordinates matches the number of texture coordinates (and vertex normals, in future tutorial sessions), which makes texture mapping easier without drastically change other part of the program.





1. Texture mapping

To use texture, the texture coordinates must be assigned to the vertices of each object using appropriate buffers. This is done by modify the corresponding setup buffer functions. Notice that, because textures are used, colours are no longer needed, and they are removed from the buffers and the shaders. Textures are setup and loaded through three functions: **`setupTextures()`**, **`loadImageForTexture()`**, and **`textureFinishedLoading()`**.

2. Store properties in a global variable

In previous tutorials, we have added new properties to the different WebGL resource objects that are created from the WebGL API. For example, the `itemSize` and `numberOfItems` are added to the created WebGLBuffer object, as shown in the following code segment:

```
floorVertexPositionBuffer = gl.createBuffer();
  . . .
floorVertexPositionBuffer.itemSize = 3;
floorVertexPositionBuffer.numberOfItems = 4;
```

While this might be a convenient way to organize the code, it is not a way to have robust code if the context of the WebGL program is lost (see below). When the context is lost, the method `gl.createBuffer()` returns `null` and an exception will be thrown when you try to add the properties `itemSize` and `numberOfItems` to the `floorVertexPositionBuffer` that is `null`. For the same reason, we should avoid adding properties to any other WebGL objects, such as texture objects, shader objects, program objects, and so on.

A better way to handle such WebGL object-related properties is to store them in a **global** object that is not created by WebGL. In following code snippet, a JavaScript **object** called `pwgl` (*standing for properties of WebGL*, or whatever name you like) is used to store the items.

```
// globals
var pwgl = {}; // declare a global JavaScript object.
  . . .
```

```
pwgl.floorVertexPositionBuffer=gl.createBuffer();
  . . .
pwgl.FLOOR_VERTEX_POS_BUF_ITEM_SIZE = 3;
pwgl.FLOOR_VERTEX_POS_BUF_NUM_ITEMS = 4;
```

Pay attention to the use of object `pwgl` through out the program to see what have been stored in it.

3. Handling lost context

We have learnt that to run a WebGL application, a WebGL context must be created for the application. The context provides an environment for, and access to, the WebGL API. However, this context could be lost. There are several reasons why a WebGL application could lose its context. For example, when a call to drawing functions takes too long to execute and the system becomes unresponsive or hangs. Once the context is lost, any access to WebGL resources becomes unavailable and the application will stop running.

When a WebGL application has lost its context, the default behaviour is that the device will not try to restore the context and the user must reload the application manually in the browser. When lost context happens, the operating systems and device drivers will find that the GPU becomes unresponsive. To recover, the GPU will be reset and an event **`webglcontextlost`** is sent to the WebGL application where it will trigger the event handling (lost context) procedure provided by the programmer.

The lost context handling mechanism in this tutorial will rely on the detection of this event. On detecting the **`webglcontextlost`** event, the program will prevent the default behaviour (not restoring the lost context) from working and try to restore the lost context. Once the context is restored, another event **`webglcontextrestored`** will be sent to the application. On receiving this event, a procedure that re-initialises the shaders and buffers will be launched.
See http://www.khronos.org/webgl/wiki/HandlingContextLost for more details.

Detection of `webglcontextlost` and `webglcontextrestored` events is done by registering event listeners to canvas object:

```
canvas.addEventListener('webglcontextlost',
                        handleContextLost, false);
canvas.addEventListener('webglcontextrestored',
                         handleContextRestored, false);
```

where the first argument are the events that the event listeners are registered for. The second argument are the listeners**,** `handleContextLost` and `handleContextRestored`, which are two functions. The third argument is a Boolean that specifies whether the event handler should capture events during what is called the capturing phase of the event propagation. In this case, we don't need to capture any events during the capturing phase, so `false` is set.

When the listener for the `webglcontextlost` event is called, the program first stops the default action (i.e., the lost context will not be restored). This is done by calling `preventDefault()` on the detection of the event. Meanwhile, the rendering loop is stopped. So far, we have worked with examples where the drawing method is called once. When working with animations, the drawing method must be called every frame in a loop. This loop is created by calling

```
pwgl.requestId = requestAnimFrame(draw,canvas);
```

which returns an ID (a value other than zero) for the callback. The method `cancelRequestAnimFrame()` stops the callback by its ID.

The second listener `handleContextRestored`, when called, will restore the lost context, but all resources that you have allocated/created through WebGL, e.g., textures, buffers, shaders and shader

program, will not be recovered. These resources must be re-initialised. Also, we need to re-start the rendering loop (from where it was lost) and re-initialise the needed resources when the context is restored.

```
function handleContextRestored(event) {
      setupShaders();
      setupBuffers();
      setupTextures();
      gl.clearColor(0.0, 0.0, 0.0, 1.0);
      pwgl.requestId = requestAnimFrame(draw,canvas);;
}
```

To test if a WebGL application can handle the events correctly, we can simulate the context lost event by creating a test environment. The simulation environment is created by calling the function in library `webgl-debug.js`:

```
canvas =
  WebGLDebugUtils.makeLostContextSimulatingCanvas(
         canvas);
```

The method `makeLostContextSimulatingCanvas()` creates a wrapper around the original canvas. The wrapper simulates the `webglcontextlost` and `webglcontextrestored` events. This tutorial makes use this simulated environment. On successful running of the program, you will see a change in background colour when you press a mouse button, which indicates the re-initialisation of the application.

Context lost is checked when compile and link the shaders.

```
gl.compileShader(shader);
if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS) &&
                !gl.isContextLost()) {
  . . .
}
 . . .
gl.linkProgram(shaderProgram);
if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS) &&
                              !gl.isContextLost()) {
   . . .
}
```

Exercise: Copy the library and texture files from Moodle. Start from the program provided. The program is incomplete. As it is, the program draws a black background colour on the canvas. When you click, you should see changes in the background colour (simulating the event of context lost, the detection and handling of the event). Once the program works correctly, complete it by adding the missing statements in the shaders, floor texture coordinates, and texture set up functions.

Note: If your program works but fails to load the textures (Google chrome or Firefox), open the page in with **Firefox.** In Firefox URL type **`about:config`** and navigate to

`security.fileuri.strict_origin_policy` and turn its value to **`false`**.
Reload the application.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Using Textures on Floor, Table and Box.</title>
<script src="webgl-debug.js"></script>
<script type="text/javascript" src="glMatrix.js"></script>
<script src="webgl-utils.js"></script>

<meta charset="utf-8">
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec2 aTextureCoordinates;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  varying vec2 vTextureCoordinates;

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

<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec2 vTextureCoordinates;
  uniform sampler2D uSampler;
  void main() {
    gl_FragColor = texture2D(uSampler, vTextureCoordinates);
  }
</script>

<script type="text/javascript">
// globals
var gl;
var pwgl = {};

var inc=0; //variable for lost context test

pwgl.ongoingImageLoads = [];
var canvas;
```

```
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 we don't find an element with the specified id
  // we do an early exit
  if (!shaderScript) {
    return null;
  }

  // Loop through the children for the found DOM element and
  // build up the shader source code as a string
  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) &&
      !gl.isContextLost()) {
    alert(gl.getShaderInfoLog(shader));
    return null;
  }
  return shader;
}
```

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

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

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

  gl.useProgram(shaderProgram);

  pwgl.vertexPositionAttributeLoc = gl.getAttribLocation(shaderProgram,
                                "aVertexPosition");
  pwgl.vertexTextureAttributeLoc = gl.getAttribLocation(shaderProgram,
                                "aTextureCoordinates");
  pwgl.uniformMVMatrixLoc = gl.getUniformLocation(shaderProgram,
                                "uMVMatrix");
  pwgl.uniformProjMatrixLoc = gl.getUniformLocation(shaderProgram,
                                "uPMatrix");
  pwgl.uniformSamplerLoc = gl.getUniformLocation(shaderProgram,
                                "uSampler");

  gl.enableVertexAttribArray(pwgl.vertexPositionAttributeLoc);
  gl.enableVertexAttribArray(pwgl.vertexTextureAttributeLoc);

  pwgl.modelViewMatrix = mat4.create();
  pwgl.projectionMatrix = mat4.create();
  pwgl.modelViewMatrixStack = [];
}

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

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

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

  //vertex coordinates of the floor
  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);

  pwgl.FLOOR_VERTEX_POS_BUF_ITEM_SIZE = 3;
  pwgl.FLOOR_VERTEX_POS_BUF_NUM_ITEMS = 4;

  // floor index
  pwgl.floorVertexIndexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.floorVertexIndexBuffer);
  var floorVertexIndices = [0, 1, 2, 3];

  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
                  Uint16Array(floorVertexIndices), gl.STATIC_DRAW);

  pwgl.FLOOR_VERTEX_INDEX_BUF_ITEM_SIZE = 1;
  pwgl.FLOOR_VERTEX_INDEX_BUF_NUM_ITEMS = 4;



  //Floor texture coordinates
  pwgl.floorVertexTextureCoordinateBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexTextureCoordinateBuffer);

  //floor texture coordinates. Note that wrapping is used
  var floorVertexTextureCoordinates = [
      2.0, 0.0,
      2.0, 2.0,
      0.0, 2.0,
      0.0, 0.0
  ];

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

  pwgl.FLOOR_VERTEX_TEX_COORD_BUF_ITEM_SIZE = 2;
  pwgl.FLOOR_VERTEX_TEX_COORD_BUF_NUM_ITEMS = 4;
}

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

  //draw an illustration to understand the coordinates, if necessary
   var cubeVertexPosition = [

       // Front face
       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

       // Back face
       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
```

```
     // Left face
    -1.0,  1.0,  1.0, //v8
    -1.0,  1.0, -1.0, //v9
    -1.0, -1.0, -1.0, //v10
    -1.0, -1.0,  1.0, //v11

     // Right face
     1.0,  1.0,  1.0, //12
     1.0, -1.0,  1.0, //13
     1.0, -1.0, -1.0, //14
     1.0,  1.0, -1.0, //15

     // Top face
      1.0,  1.0,  1.0, //v16
      1.0,  1.0, -1.0, //v17
     -1.0,  1.0, -1.0, //v18
     -1.0,  1.0,  1.0, //v19

      // Bottom face
      1.0, -1.0,  1.0, //v20
      1.0, -1.0, -1.0, //v21
     -1.0, -1.0, -1.0, //v22
     -1.0, -1.0,  1.0, //v23
];

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

pwgl.CUBE_VERTEX_POS_BUF_ITEM_SIZE = 3;
pwgl.CUBE_VERTEX_POS_BUF_NUM_ITEMS = 24;

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

 var cubeVertexIndices = [
          0, 1, 2,      0, 2, 3,    // Front face
          4, 6, 5,      4, 7, 6,    // Back face
          8, 9, 10,     8, 10, 11,  // Left face
          12, 13, 14,   12, 14, 15, // Right face
          16, 17, 18,   16, 18, 19, // Top face
          20, 22, 21,   20, 23, 22  // Bottom face
      ];

gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
          Uint16Array(cubeVertexIndices), gl.STATIC_DRAW);
pwgl.CUBE_VERTEX_INDEX_BUF_ITEM_SIZE = 1;
pwgl.CUBE_VERTEX_INDEX_BUF_NUM_ITEMS = 36;


// Setup buffer with texture coordinates
pwgl.cubeVertexTextureCoordinateBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexTextureCoordinateBuffer);

//Think about how the coordinates are assigned. Ref. vertex coords.
var textureCoordinates = [
  //Front face
  0.0, 0.0, //v0
  1.0, 0.0, //v1
  1.0, 1.0, //v2
```

```
    0.0, 1.0, //v3

    // Back face
    0.0, 1.0, //v4
    1.0, 1.0, //v5
    1.0, 0.0, //v6
    0.0, 0.0, //v7

    // Left face
    0.0, 1.0, //v1
    1.0, 1.0, //v5
    1.0, 0.0, //v6
    0.0, 0.0, //v2

    // Right face
    0.0, 1.0, //v0
    1.0, 1.0, //v3
    1.0, 0.0, //v7
    0.0, 0.0, //v4

    // Top face
    0.0, 1.0, //v0
    1.0, 1.0, //v4
    1.0, 0.0, //v5
    0.0, 0.0, //v1

    // Bottom face
    0.0, 1.0, //v3
    1.0, 1.0, //v7
    1.0, 0.0, //v6
    0.0, 0.0, //v2
  ];

  gl.bufferData(gl.ARRAY_BUFFER, new
            Float32Array(textureCoordinates),gl.STATIC_DRAW);
  pwgl.CUBE_VERTEX_TEX_COORD_BUF_ITEM_SIZE = 2;
  pwgl.CUBE_VERTEX_TEX_COORD_BUF_NUM_ITEMS = 24;

}

function setupTextures() {
  // Texture for the table
  pwgl.woodTexture = gl.createTexture();
  loadImageForTexture("wood_128x128.jpg", pwgl.woodTexture);

  // Texture for the floor
  pwgl.groundTexture = gl.createTexture();
  loadImageForTexture("wood_floor_256.jpg", pwgl.groundTexture);

  // Texture for the box on the table
  pwgl.boxTexture = gl.createTexture();
  loadImageForTexture("wicker_256.jpg", pwgl.boxTexture);
}

function loadImageForTexture(url, texture) {
  var image = new Image();
  image.onload = function() {
    pwgl.ongoingImageLoads.splice(pwgl.ongoingImageLoads.indexOf(image),
1);
```

```
    //The splice() method adds/removes items to/from an array, and returns
    //the removed item(s).
    //Syntax: array.splice(index,howmany,item1,.....,itemX)
    textureFinishedLoading(image, texture);
  }
  pwgl.ongoingImageLoads.push(image);
  image.src = url;
}

function textureFinishedLoading(image, 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,
                image);

  gl.generateMipmap(gl.TEXTURE_2D);

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

  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.MIRRORED_REPEAT);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.MIRRORED_REPEAT);
  gl.bindTexture(gl.TEXTURE_2D, null);
}

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

function uploadModelViewMatrixToShader() {
  gl.uniformMatrix4fv(pwgl.uniformMVMatrixLoc, false,
                              pwgl.modelViewMatrix);
}

function uploadProjectionMatrixToShader() {
  gl.uniformMatrix4fv(pwgl.uniformProjMatrixLoc,
                      false, pwgl.projectionMatrix);
}

function drawFloor() {

  // Draw the floor
  gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexPositionBuffer);
  gl.vertexAttribPointer(pwgl.vertexPositionAttributeLoc,
                         pwgl.FLOOR_VERTEX_POS_BUF_ITEM_SIZE,
                         gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexTextureCoordinateBuffer);
  gl.vertexAttribPointer(pwgl.vertexTextureAttributeLoc,
                         pwgl.FLOOR_VERTEX_TEX_COORD_BUF_ITEM_SIZE,
                         gl.FLOAT, false, 0, 0);

  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, pwgl.groundTexture);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.floorVertexIndexBuffer);
  gl.drawElements(gl.TRIANGLE_FAN, pwgl.FLOOR_VERTEX_INDEX_BUF_NUM_ITEMS,
```

```
                gl.UNSIGNED_SHORT, 0);
}

function drawCube(texture) {
  gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexPositionBuffer);
  gl.vertexAttribPointer(pwgl.vertexPositionAttributeLoc,
                         pwgl.CUBE_VERTEX_POS_BUF_ITEM_SIZE,
                         gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexTextureCoordinateBuffer);
  gl.vertexAttribPointer(pwgl.vertexTextureAttributeLoc,
                         pwgl.CUBE_VERTEX_TEX_COORD_BUF_ITEM_SIZE,
                         gl.FLOAT, false, 0, 0);

  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, texture);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.cubeVertexIndexBuffer);

  gl.drawElements(gl.TRIANGLES, pwgl.CUBE_VERTEX_INDEX_BUF_NUM_ITEMS,
                              gl.UNSIGNED_SHORT, 0);
}

function drawTable(){
  // Draw table top
  pushModelViewMatrix();
  mat4.translate(pwgl.modelViewMatrix, [0.0, 1.0, 0.0],
                                       pwgl.modelViewMatrix);
  mat4.scale(pwgl.modelViewMatrix, [2.0, 0.1, 2.0], pwgl.modelViewMatrix);

  uploadModelViewMatrixToShader();

  // Draw the actual cube (now scaled to a cuboid) with woodTexture
  drawCube(pwgl.woodTexture);
  popModelViewMatrix();

  // Draw table legs
  for (var i=-1; i<=1; i+=2) {
    for (var j= -1; j<=1; j+=2) {
      pushModelViewMatrix();
      mat4.translate(pwgl.modelViewMatrix, [i*1.9, -0.1, j*1.9],
                              pwgl.modelViewMatrix);
      mat4.scale(pwgl.modelViewMatrix, [0.1, 1.0, 0.1],
                              pwgl.modelViewMatrix);
      uploadModelViewMatrixToShader();
      drawCube(pwgl.woodTexture);
      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,
                              1, 100.0, pwgl.projectionMatrix);
  mat4.identity(pwgl.modelViewMatrix);
  mat4.lookAt([8, 5, -10],[0, 0, 0], [0, 1,0], pwgl.modelViewMatrix);
```

```
  uploadModelViewMatrixToShader();
  uploadProjectionMatrixToShader();
  gl.uniform1i(pwgl.uniformSamplerLoc, 0);

  drawFloor();

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

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

  pwgl.requestId = requestAnimFrame(draw,canvas);
}


function startup() {
  canvas = document.getElementById("myGLCanvas");
  canvas = WebGLDebugUtils.makeLostContextSimulatingCanvas(canvas);
  canvas.addEventListener('webglcontextlost', handleContextLost, false);
  canvas.addEventListener('webglcontextrestored', handleContextRestored,
                              false);

  window.addEventListener('mousedown', function() {
     canvas.loseContext();
  });

  gl = createGLContext(canvas);
  setupShaders();
  setupBuffers();
  setupTextures();
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.enable(gl.DEPTH_TEST);

  draw();
}

function handleContextLost(event) {
  event.preventDefault();
  cancelRequestAnimFrame(pwgl.requestId);

   // Ignore all ongoing image loads by removing
   // their onload handler
   for (var i = 0; i < pwgl.ongoingImageLoads.length; i++) {
     pwgl.ongoingImageLoads[i].onload = undefined;
   }
   pwgl.ongoingImageLoads = [];
}
```

```
function handleContextRestored(event) {
  setupShaders();
  setupBuffers();
  setupTextures();
  inc=inc+0.1;
  gl.clearColor(0.0+inc, 0.0+inc, 0.0+inc, 1.0);
  gl.enable(gl.DEPTH_TEST);
  pwgl.requestId = requestAnimFrame(draw,canvas);
}
</script>

</head>

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

</html>
```