**Lab 7 : Introduction to HTML5 Canvas and WebGL**

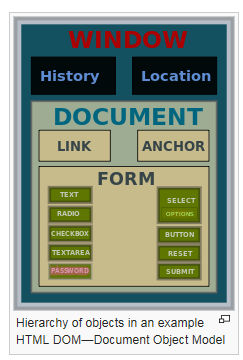
**Objectives:**

By the end of this lab session students are able to:

* Understand what HTML 5 Canvas is and how it works
* Understand what WebGL is.
* Understand the relationship between HTML5 Canvas and WebGL

1. **Introduction to HTML5**

HTML 5 is the latest specification of creating web. Previous HTML (1.0 -4.0 and XHTML) were specification for arranging text and images on a web page with some ability for automated categorization using semantic data. However, HTML5 offers more than laying out text and images, but also serves as application development platform that could support playing video and audio, interactive 2D and 3D graphics and more[1].

Part of this new specification is **Canvas** tag which has width and height attributes. It is a two-dimensional grid that able to handle graphics such as drawing which is rendered with JavaScript. The way the Canvas works is first we'll need to place the canvas tag somewhere inside the HTML document, access the canvas tag with JavaScript, create a context, and then utilize the HTML5 Canvas API to draw visualizations [2].

The Canvas can be viewed as having two parts:

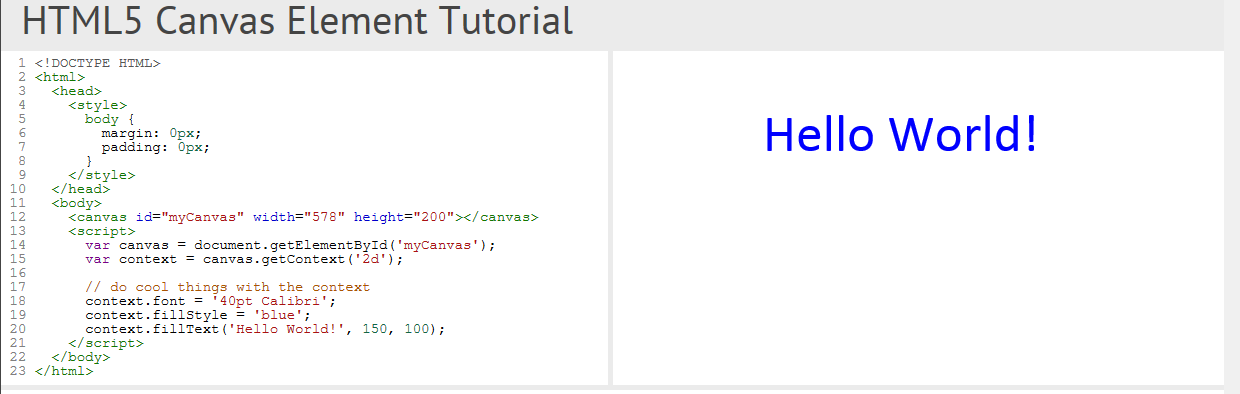
1. **Canvas element** : A DOM (Document Object Model) node that is embedded in the HTML page. Each canvas element can ONLY has one context. Image is from wikipedia.org
2. **Canvas context :** An object with properties and methods that we can use to render graphics inside the canvas element. It can be 2D or WebGL (3D). Usually renders pixel by pixel.

The summary of the complete HTML5 Canvas API for the 2D context can be found at this URL [3]:

[**http://cheatsheetworld.com/programming/html5-canvas-cheat-sheet/**](http://cheatsheetworld.com/programming/html5-canvas-cheat-sheet/)

**Creating First HTML5 Canvas element**

Using any text editor, type the following HTML5 syntax and save it as FirstCanvas.html



What is the output?\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Now, inside the <canvas> tag, add style="background-color:#33ff51". Now what happen?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Code explained:**

1. Besides the usual HTML syntax, the main part is to add an HTML <canvas> element to your page. Here the id is IMPORTANT as it will be called back by a canvas object which is rendered using JavaScript.
2. Inside a <script> tag, create a canvas object ( var canvas a.k.a. *variable named canvas*) from the HTML canvas element:

var canvas = document.getElementById(‘myCanvas’);

1. Create a 2D drawing object ( var context a.k.a variable named context) for the canvas object:

var context = canvas.getContext(‘2d’);

1. Now we have declared variables for both the canvas element and the context, we are going to start drawing on it. We are going to display “ Hello World” , a MUST for a beginner of any programming ☺. But before that we need to specify the font, size and what color to use. Choose the type of font and the size, followed by the color of the font.

context.font = ‘40pt, Calibri’;

contect.fillStyle = ‘blue’;

1. Now determine the words or sentence to be display and the X, Y position to display it on the canvas.

context.fillText (‘Hello World!’, 150, 100);

**Drawing a clock in HTML5 (based on [4])**

We are now ready to move on to the next exercise. We are going to draw a clock with animated hands.

1. Create a new HTML page for this.

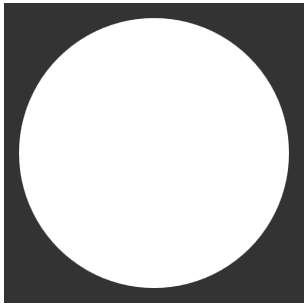
|  |
| --- |
| <!DOCTYPE html> <html> <body>  <canvas id="canvas" width="300" height="300" style="background-color:#333"></canvas> <script>  //Need to declare a canvas object and a canvas context before draw  //The drawing goes here  </script>  </body> </html> |

1. Create a canvas object and a canvas context for this inside the <script> tag

|  |
| --- |
| var canvas = document.getElementById("canvas"); var ctx = canvas.getContext("2d"); |

1. Now determine the clock radius , using the height of the canvas. This will ensure that the clock stil works if we resize the canvas.

|  |
| --- |
| var radius = canvas.height / 2; |

1. By default, anything will be drawn on (0,0) position, so we going to remap to the center of the canvas:

|  |
| --- |
| ctx.translate(radius, radius); |

1. Reduce the radius to 90% to draw the clock inside the canvas

|  |
| --- |
| radius = radius \* 0.90 |

1. We going to call a function to draw the clock:

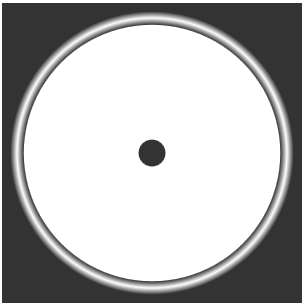
|  |
| --- |
| drawClock(); |

1. Now, create the *drawClock()* function:

|  |
| --- |
| function drawClock() {     ctx.arc(0, 0, radius, 0 , 2\*Math.PI); // draw a circle      ctx.fillStyle = "white"; //white color     ctx.fill(); } |

1. Save and display the HTML file to see what we have done so far.

**Draw a clock face**

1. Using the same file, now we are going to change the *drawClock()* function to make it more artistic ☺.

|  |
| --- |
| function drawClock() {     **drawFace(ctx, radius);**  **drawNumbers(ctx, radius);** }  function drawFace(ctx, radius) { }  function drawNumbers(ctx, radius) {  } |

1. Inside *drawFace(….)* , draw the white circle:

|  |
| --- |
| ctx.beginPath(); ctx.arc(0, 0, radius, 0, 2\*Math.PI); ctx.fillStyle = 'white'; ctx.fill(); |

1. Create a radial gradient ( 95% and 105% of the original clock radius):

|  |
| --- |
| grad = ctx.createRadialGradient(0,0,radius\*0.95, 0,0,radius\*1.05); |

1. Create 3 color stops, each for the inner, middle and outer edge of the clock face to create a 3D effect.

|  |
| --- |
| grad.addColorStop(0, '#333'); grad.addColorStop(0.5, 'white'); grad.addColorStop(1, '#333'); |

1. Define the gradient as the stroke style of the drawing object:

|  |
| --- |
| ctx.strokeStyle = grad; |

1. Define the line width of the drawing object ( 10% of the radius):

|  |
| --- |
| ctx.lineWidth = radius \* 0.1; |

1. Draw the circle:

|  |
| --- |
| ctx.stroke(); |

1. Draw the clock center:

|  |
| --- |
| ctx.beginPath(); ctx.arc(0, 0, radius\*0.1, 0, 2\*Math.PI); ctx.fillStyle = '#333'; ctx.fill(); |

1. Now, inside *drawNumbers(ctx, radius )* function, declare variable for angle and number

|  |
| --- |
| var ang;  var num; |

1. Set the font size of the drawing object to 15% of the radius

|  |
| --- |
| ctx.font = radius\*0.15 + "px arial"; |

1. Set the text alignment to the middle and the

center of the print position:

|  |
| --- |
| ctx.textBaseline="middle"; ctx.textAlign="center"; |

1. Calculate the print position (for 12 numbers) to 85% of the radius, rotated (PI/6) for each number:

|  |
| --- |
| for(num= 1; num < 13; num++) {     ang = num \* Math.PI / 6;     ctx.rotate(ang);     ctx.translate(0, -radius\*0.85);     ctx.rotate(-ang);     ctx.fillText(num.toString(), 0, 0);     ctx.rotate(ang);     ctx.translate(0, radius\*0.85);     ctx.rotate(-ang);  } |

**Draw clock hands**

1. Now, we are going to draw the hands for this clock. In the *drawClock(),* add a function call to drawTime(ctx, radius);

|  |
| --- |
| function drawClock() {     drawFace(ctx, radius);     drawNumbers(ctx, radius); **drawTime(ctx, radius);** } |

1. In the *drawTime( ctx, radius)* declare a variable for hour , minute and second.

|  |
| --- |
| function drawTime(ctx, radius){  var now = new Date(); var hour = now.getHours(); var minute = now.getMinutes(); var second = now.getSeconds(); |

1. Calculate the angle of the hour hand, and draw it a length ( 50% of radius) , and a width (7% of radius) :

|  |
| --- |
| hour=hour%12; hour=(hour\*Math.PI/6)+(minute\*Math.PI/(6\*60))+(second\*Math.PI/(360\*60)); drawHand(ctx, hour, radius\*0.5, radius\*0.07); |

1. Now calculate for minutes and seconds, and then call a function to draw the hands.

|  |
| --- |
| //minute     minute=(minute\*Math.PI/30)+(second\*Math.PI/(30\*60));     drawHand(ctx, minute, radius\*0.8, radius\*0.07);     // second     second=(second\*Math.PI/30);     drawHand(ctx, second, radius\*0.9, radius\*0.02);  } |

1. For the function *drawHand( ctx, pos, length, width)*, we going to draw straight line width a certain width from the center of the clock to the intended length according the which hand.

|  |
| --- |
| function drawHand(ctx, pos, length, width) {     ctx.beginPath();     ctx.lineWidth = width;     ctx.lineCap = "round";     ctx.moveTo(0,0);     ctx.rotate(pos);     ctx.lineTo(0, -length);     ctx.stroke();     ctx.rotate(-pos); } |

**Let the clock starts ticking**

1. To start the clock, we need to call the drawClock function at intervals:

|  |
| --- |
| var canvas = document.getElementById("canvas"); var ctx = canvas.getContext("2d"); var radius = canvas.height / 2; ctx.translate(radius, radius); radius = radius \* 0.90 //drawClock(); setInterval(drawClock, 1000); |

1. **Introduction to WebGL**

So now that you had some experience with the HTML5 Canvas, it is time for us step up. As you have notice HTML5 Canvas focuses on 2D. To visualize 3D content, we need to venture into webGL. Goto <http://akirodic.com/p/jellyfish/> to see how powerful WebGL can be.

**So what is webGL [5]?**

Disclaimer : 3D is still hard work . But do not worries, there are available toolkits for webGL

* **Web GL is the new standard for 3D graphics in web and it is part of HTML5 technology. Royalty free.**
* **Developed and maintained by the Khronos Group that also govern COLLADA, and OpenGL.**
* **WebGL API is based on OpenGL ES( embedded standard) 2.0 and cross-platform.**
* **It allows modern browser into an application platform.**
* **Suited for dynamic 3D web applications in JavaScript Programming language**
* **Uses OpenGL shader style**

So recall or refer back to Computer Graphics notes to refresh your memory on 3D coordinate system, meshes, polygon and vertices, material. Do not forget to revise matrices and transformation, camera, viewports perspective and projections as well.

**Does your browser webGL ready?**

Before we go any further, test your web browser for webGL enable using this tester:

<http://www.contextis.com/files/blog-WebGL-webgltest.html>

**The Anatomy of a WebGL Application**

WebGL is a drawing library with a lot more ability and take full advantage of the powerful GPU hardware on most machines today. WebGL uses the HTML5 <canvas> element to display 3D on the browser page.

To render WebGL into a web page, here are the minimum steps to be done:

Disclaimer : Steps 4, 5 and 6 can be in any order.

1. Create a canvas object
2. Determine a drawing context for the canvas.
3. Initialize the viewport.
4. Create one or more buffers containing the data to be rendered

( typically vertices).

1. Create one or more matrices to define the transformation

from vertex buffers to screen space.

1. Create one or more shaders with parameters
2. Draw.

**Looking into pure webGL API**

Using webGL API alone without any toolkit requires strong understanding of OpenGL as it is very low level programming. To draw a simple square could reaches up to 100+ lines of code. In order to get to know webGL, we are going to write a HTML5 with webGL to draw a square shape.

1. Create a new HTML page with the following <body and <canvas> tags. You could refer to the previous exercise to help you.

|  |
| --- |
| <body onload="onLoad();">  <canvas id="webglcanvas" style="border: none;" width="500" height="500"></canvas>  </body> |

1. In between <script> and </script> tags, we are going to start with the declaration of components needed and leave the onLoad() function to last. Start with initializing a canvas for webGL:

|  |
| --- |
| function initWebGL(canvas) {  var gl;  try {  gl = canvas.getContext("experimental-webgl");  //The type has been define by KHRONOS  } catch (e)  { var msg = "Error creating WebGL Context!: " + e.toString();  alert(msg); throw Error(msg);  }  return gl;  } |

1. Next thing to do is to initialize a viewport.

|  |
| --- |
| function initViewport(gl, canvas)  {  gl.viewport(0, 0, canvas.width, canvas.height);  } |

1. Now, let’s create two matrices, one for ModelView and one for Projection. ModelView matrix helps define where the square is positioned in the 3D coordinate system, relative to the camera. For this example we are moving the drawing or square by translating it along the negative z-axis, which is away from camera by -3.333 units.

|  |
| --- |
| var projectionMatrix, modelViewMatrix;  function initMatrices()  {  // The transform matrix for the square  //- translate back in Z for the camera    modelViewMatrix = new Float32Array(  [1, 0, 0, 0,  0, 1, 0, 0,  0, 0, 1, 0,  0, 0, -3.333, 1]);    // The projection matrix (for a 45 degree field of view)    projectionMatrix = new Float32Array(  [2.41421, 0, 0, 0,  0, 2.41421, 0, 0,  0, 0, -1.002002, -1,  0, 0, -0.2002002, 0]);    } |

1. As webGL is similar to GL, the drawing is done with *primitives* such as triangle which has vertex. So we need to create array buffer for the data or vertices to be drawn.

|  |
| --- |
| // Create the vertex data for a square to be drawn    function createSquare(gl) {  var vertexBuffer;  vertexBuffer = gl.createBuffer();  gl.bindBuffer(gl.ARRAY\_BUFFER, vertexBuffer);  var verts = [  .5, .5, 0.0,  -.5, .5, 0.0,  .5, -.5, 0.0,  -.5, -.5, 0.0  ];  gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(verts), gl.STATIC\_DRAW);    var square = {buffer:vertexBuffer, vertSize:3, nVerts:4,  primtype:gl.TRIANGLE\_STRIP};  return square;  } |

1. In order for the square to be drawn on the canvas, WebGL requires a shader for each object. However, the shader can be used for multiple objects. A shader is composed of two parts: the vertex shader and the fragment shader/ pixel shader). The vertex shader is responsible for transforming the coordinates of the object into 2D display space, while the fragment shader is responsible for generating the final color output of each pixel for the transformed vertices, based on inputs such as color, texture, lighting, and material values. So now, we are going to supply one.

function createShader(gl, str, type) {

var shader;

if (type == "fragment") {

shader = gl.createShader(gl.FRAGMENT\_SHADER);

} else if (type == "vertex") {

shader = gl.createShader(gl.VERTEX\_SHADER);

} else {

return null;

}

gl.shaderSource(shader, str);

gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE\_STATUS)) {

alert(gl.getShaderInfoLog(shader));

return null;

}

return shader;

}

1. WebGL declares the shader using GLSL ES source. The following are the declaration for the sources:

var vertexShaderSource =

" attribute vec3 vertexPos;\n" +

" uniform mat4 modelViewMatrix;\n" +

" uniform mat4 projectionMatrix;\n" +

" void main(void) {\n" +

" // Return the transformed and projected vertex value\n" +

" gl\_Position = projectionMatrix \* modelViewMatrix \* \n" +

" vec4(vertexPos, 1.0);\n" +

" }\n";

var fragmentShaderSource =

" void main(void) {\n" +

" // Return the pixel color: always output white\n" +

" gl\_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n" +

"}\n";

var shaderProgram, shaderVertexPositionAttribute,

shaderProjectionMatrixUniform, shaderModelViewMatrixUniform;

1. We now come to the final set up for the shader, which is the initialization of the shader. The previous steps are only the declaration on how the shader would be.

function initShader(gl) {

// load and compile the fragment and vertex shader

var fragmentShader = createShader(gl, fragmentShaderSource, "fragment");

var vertexShader = createShader(gl, vertexShaderSource, "vertex");

// link them together into a new program

shaderProgram = gl.createProgram();

gl.attachShader(shaderProgram, vertexShader);

gl.attachShader(shaderProgram, fragmentShader);

gl.linkProgram(shaderProgram);

// get pointers to the shader params

shaderVertexPositionAttribute = gl.getAttribLocation(shaderProgram, "vertexPos");

gl.enableVertexAttribArray(shaderVertexPositionAttribute);

shaderProjectionMatrixUniform = gl.getUniformLocation(shaderProgram, "projectionMatrix");

shaderModelViewMatrixUniform = gl.getUniformLocation(shaderProgram, "modelViewMatrix");

if (!gl.getProgramParameter(shaderProgram, gl.LINK\_STATUS)) {

alert("Could not initialise shaders");

}

}

1. Next, we are going to create a draw function to draw the square using the shader that we have declared.

|  |
| --- |
| function draw(gl, obj) {  // clear the background (with black)  gl.clearColor(0.0, 0.0, 0.0, 1.0);  gl.clear(gl.COLOR\_BUFFER\_BIT);  // set the vertex buffer to be drawn  gl.bindBuffer(gl.ARRAY\_BUFFER, obj.buffer);  // set the shader to use  gl.useProgram(shaderProgram);  // connect up the shader parameters: vertex position and projection/model matrices  gl.vertexAttribPointer(shaderVertexPositionAttribute, obj.vertSize, gl.FLOAT, false, 0, 0);  gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix);  gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);  // draw the object  gl.drawArrays(obj.primtype, 0, obj.nVerts);  } |

1. Finally, we create the onLoad() function to invoke the drawing when the webpage is loaded.

function onLoad() {

var canvas = document.getElementById("webglcanvas");

var gl = initWebGL(canvas);

initViewport(gl, canvas);

initMatrices();

var square = createSquare(gl);

initShader(gl);

draw(gl, square);

}

1. Save the file as .html file
2. View the content of the file using web browser.

References:

[1] Adam Ness, HTML vs HTML5 - What are the key differences? Which is better?, At: http://www.quora.com/HTML-vs-HTML5-What-are-the-key-differences-Which-is-better [Accessed on: 5 Feb 2015]

[2] HTML5 Canvas Tutorial, At : <http://www.html5canvastutorials.com/tutorials/html5-canvas-element/> [Accessed on 5 Feb 2015]

[3] HTML5 Canvas Cheat Sheet, At: [**http://cheatsheetworld.com/programming/html5-canvas-cheat-sheet/**](http://cheatsheetworld.com/programming/html5-canvas-cheat-sheet/)[Accessed on : 5 Feb 2015]

[4] <http://www.w3schools.com/canvas/canvas_clock.asp>

[5] Tony Parisi, 2012, WebGL: Up and Running, O’Reilly