Intro To HTML5 Canvas Rendering

HTML5 Canvas

W3 Schools References:

- Overview: http://www.w3schools.com/html/html5_canvas.asp
- API: http://www.w3schools.com/tags/ref_canvas.asp

Other References:

- http://diveintohtml5.info/canvas.html
- http://www.canvasdemos.com/

HTML5 introduced a new <canvas> element to HTML. The element is only a container for 2D graphics, it requires scripting (e.g., JavaScript) to perform the rendering. While the API isn't huge, it is sufficient for a wide range of rendering tasks, including:

- Lines
- Rectanges
- Paths (think curves)
- Text/Fonts
- Images, and Pixel Manipulation
- Transformations (rotation, scaling)

Defining a Canvas Element

```
<canvas id = "id-canvas" width = "500" height = "500"></canvas>
```

```
<canvas id = "id-canvas" width = "500" height = "500" style = "width: 50%; height: 50%;"></canvas>
```

The non-CSS width and height specify the drawing coordinate size of the canvas itself.

The CSS width and height specify the space taken up by the element on the HTML page.

The CSS width/height properties are not necessary. If they are not used, the element is sized to the pixel size specified with the non-CSS width/height properties.

Canvas Coordinate System

Upper Left: 0,0

Lower Right: (width – 1), (height – 1)

(Draw this on the board to better demonstrate)

This is not what most drawing systems use, Y is reversed. But this makes sense in the context of the way a browser works.

Obtaining the Canvas Object

```
var canvas = document.getElementById('id-canvas');
var context = canvas.getContext('2d');
```

The `context` object is where the API exists, it is what you'll use to do all of your rendering tasks.

Clearing the Canvas

If you are used to APIs like XNA, OpenGL, or DirectX, you know that you have to re-draw everything every frame. That isn't actually true for an HTML5 Canvas. Once you draw something, it persists until you draw over it. This means you can draw something on one frame, draw on the next frame, and both results stay on the canvas. The canvas is just a buffer where pixel results are stored.

With the above said, most of the time, you actually do want to re-draw the entire canvas between each frame. The way this is done is to make a call to `clearRect` to wipe out what is contained within that rectangle.

```
canvas.clearRect(0, 0, canvas.width, canvas.height);
```

Using what we know about JavaScript and prototypes, we can add a clear function to the context object. It looks like this...

```
CanvasRenderingContext2D.prototype.clear = function() {
            this.clearRect(0, 0, canvas.width, canvas.height);
      };
```

When you see the code I provide, there is more code than this. The extra code ensures the clear works regardless of the state of the canvas, and then returns it back to its original state after clearing.

Drawing Shapes

Rectangle

```
context.fillStyle = 'rgb(red, green, blue, alpha)';
context.fillRect(startX, startY, width, height);

context.strokeStyle = 'rgb(red, green, blue, alpha)';
context.strokeRect(startX, startY, widht, height);
```

`red`, `green`, and `blue` are all values in the range of [0, 255] or as a percentage [0,100]%.

`alpha` is a transparency value in the range of [0,1]; with 0 fully transparent and 1 fully opaque.

`fillStyle` and `strokeStyle` can also be gradients or patterns. Refer to the W3 Schools web site for details on defining these.

Polygon

```
context.beginPath();
context.moveTo(pt1.x, pt2.y);
context.LineTo(pt2.x, pt2.y);
context.LineTo(pt3.x, pt3.y);
context.closePath();

context.fillStyle = 'rgb(red, green, blue, alpha)';
context.fill();
```

```
context.strokeStyle = 'rgb(red, green, blue, alpha)';
context.stroke();
```

Rotating Shapes (or anything for that matter)

Rotation, in the general sense, requires three operations:

1. Translation based upon the object to be rotated center
2. Rotation based upon the object center
3. Negative translation based upon the object center

In Canvas, the thing being translated and rotated is the canvas, not an object, don't forget this!! It isn't the same as XNA, OpenGL, and DirectX...I've made this mistake many times already.

If we know the `center` of an object as center, with `x` and `y` coordinates, in-place rotation looks like:

```
context.translate(center.x, center.y);
context.rotate(angle);
context.translate(-center.x, -center.y);

...draw your shape here...
```

But, there is a problem...you've left the canvas in a rotated state. You need to restore the canvas rotation after done with the object rotation. One approach is to just perform the negative rotation after you are done drawing. Thankfully, however, canvas provides a facility to quickly save and restore context states, through the use of a context stack. Using the context stack, in-place rotation now looks like:

```
context.save();

context.translate(center.x, center.y);
context.rotate(angle);
context.translate(-center.x, -center.y);


...draw your shape here...


context.restore();
```

Maybe make a mention of the `clear` function written previously and how it uses this to restore the canvas state after clearing.

At this point, go ahead and show the shape rendering demo...