CS3240 Interaction Design: Lab 03

HTML5: Canvas

Prepared By: Wong Zhouchun, Steve Ng, Dr Bimlesh Wadhwa, Satyam Agarwala & Low Guan Hong

Contents

Introducing HTML5

HTML5 represents the latest in the evolution of the Internet. The last major iteration HTML4 was introduced in 1997 and has been tweaked and modified since to keep up with the rapid growth and demands of the modern web. To work with multimedia forms like audio and video companies like Microsoft and Adobe introduced plugins like Silverlight and Flash which brought the rich multimedia experience to the Internet. However this was not without cost. You have all probably faced the pitfalls of using flash.

HTML5 brought with it the rich multimedia experience built in to the language itself. It changed the entire industry and became especially significant in the past few years with the explosion of smartphones and tablets that do not fare well with plugins that are very processor intensive.

With web applications being favored over their desktop counterparts due to the ubiquitous accessibility of the former, HTML5 represents a giant leap in the shift in computing paradigm. It gives a lot of firepower to the web browser that till before was but a tool. Now it is poised to be what the graphical user interface was in the 1970's and HTML5 is powering that.

The Ubiquity & power of HTML5

The beauty of HTML5 is it in wide spread usability. It is lightweight, hence usable on any device. You might see it being used in microwaves if they ever need a browser. HTML5 today is supported in every modern web browser (and hence almost every computing device) and continues to evolve to add more features everyday. It frees users of the use of individual plugins and creates a true web standard.

Even though it is available for use on almost any device it does not compromise on ability. Most of what could be done before using various third party add ons can today be achieved using HTML5.

The <canvas> element

One of the caveats of learning HTML5 is that it is still a very dynamic standard. Due to its modular nature many different components though adopted widely are still in a state of constant evolution. This makes learning HTML5 difficult. That is why we will focus this lab on one of the core elements of the HTML5 specification, the canvas element. The canvas element allows for dynamic scriptable rendering of 2D and 3D shapes. It is this element that can provide a possible replacement of plugins like flash when used with other accompanying technologies like CSS3 and JavaScript.
We will introduce you to drawing basic shapes on the canvas and how to do animations. There is much to be learned about the canvas beyond the material presented in this lab. At every stage you are encouraged to play around with the example code to better understand the concepts presented. You should try creating shapes and graphics that you think of yourself using the techniques presented here. This will increase your confidence in being able to use the canvas element effectively.

For this lab, you are advised to use a modern browser like Firefox or Google Chrome. For editing you can use Notepad++ on Windows and on Mac any editor like TextMate or SublimeText 2 will suffice.

<canvas> basics

Before we can start having fun with the canvas we need to understand some basic ideas about the canvas element.

The canvas element is like any other HTML tag. Using the canvas element is really simple.

To include it in a page all the code you need is

Usage

```
<canvas id="canvas" width="150" height="150">
        <!-- Insert fallback content here -->
</canvas>
```

A few points to note:

- The "width" and "height" attributes are optional. If not defined the default canvas size will be 300 pixels wide and 150 pixels high. It is, however, good practice to always specify the "width" and "height" attributes.

- The "id" attribute is common to most html elements. It is essential in order to identify the canvas element in our scripts, which will be used to manipulate the canvas.

- Because the `<canvas>` element is still relatively new and isn't implemented in some browsers (such as Internet Explorer versions below IE9), we need a means of providing fallback content when a browser doesn't support the element.

  This is very straightforward: we just provide alternative content inside the canvas element. Browsers, which don't support <canvas>, will ignore the container and render the fallback content inside it. Browsers, which do support <canvas>, will ignore the content inside the container, and just render the canvas normally.

  For example, we could provide a text description of the canvas content or provide a static image of the dynamically rendered content.

Rendering Context

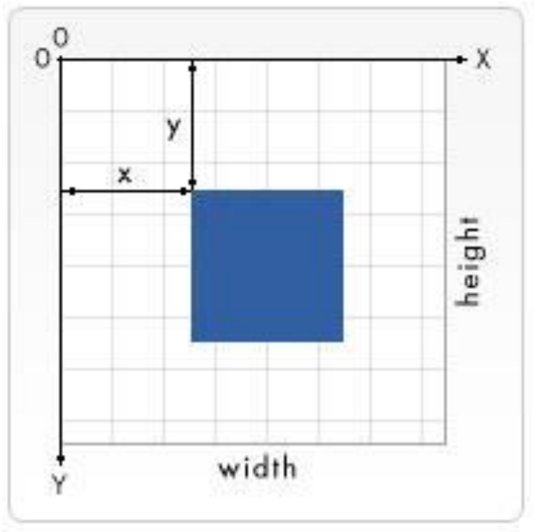
Simply adding a <canvas> element is not of much use. The real magic happens in the rendering context of a canvas element. Let's reiterate on that point. You do not manipulate the <canvas> element but the rendering context of the <canvas> element. In this lab we will introduce the 2D rendering context. There exists a 3D rendering context as well that uses WebGL and is used for HTML5 games and a variety of other functions.

To access the 2D rendering context we use the DOM method getContext(). The function takes one parameter that specifies what type of context to retrieve. In this lab we will be working with the 2D context. The following code shows how to get the rendering context

Accessing the rendering context will be the first step in any canvas manipulation.

The canvas coordinate system

The 2d rendering context is a standard screen-based drawing platform. Like other 2d platforms, it uses a flat Cartesian coordinate image of the dynamically rendered content system with the origin (0, 0) at the top left. Moving to the right will increase the x value, and moving downwards will increase the y value. Understanding how the coordinate system works is integral if you want to have things draw in the right place.



A single unit in the coordinate system is usually equivalent to 1 pixel on the screen, so the position (24, 30) would be 24 pixels right and 30 pixels down. There are some occasions where a unit in the coordinate system might equal 2 pixels; like with high definition displays, but the general rule of thumb is that 1 coordinate unit equals 1 screen pix

Drawing on the canvas

The 2D context allows us to draw literally anything we want. It provides some fundamental methods for drawing which when combined can be very versatile in its usage.

Shapes

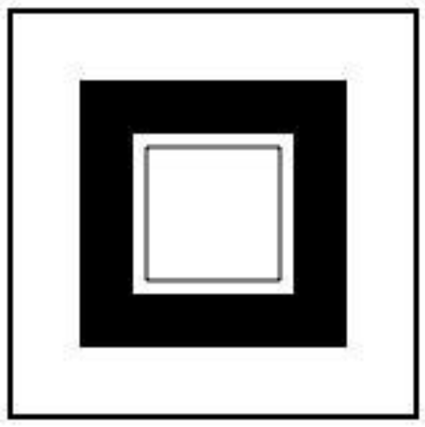
The only *shape* supported out of the box is the rectangle. All other shapes such as circles, polygons need to be constructed using paths, introduced next. Three functions are provided to draw rectangles. They are

fillRect(x,y,width,height)	Draws a filled rectangle
strokeRect(x,y,width,height)	Draws a rectangular outline
clearRect(x,y,width,height)	Clears the specified area and makes it fully transparent

(x,y) specify the top left corner of the rectangle relative to the origin. The width and height parameters are self-explanatory. Lets see these methods in action

```
function draw(){
    var canvas = document.getElementById('canvas'); if
    (canvas.getContext){
        var ctx = canvas.getContext('2d');

        ctx.fillRect(25,25,100,100);
        ctx.clearRect(45,45,60,60);
        ctx.strokeRect(50,50,50,50);
    }
}
```

The result is shown alongside.



Paths

Rectangles are easy to draw. However they are nowhere near sufficient. In order to draw more complex shapes we need to be able to draw curves and custom shapes. This is where paths come into the picture.

Drawing paths is more complex than drawing rectangles. We need to specify when the path begins, ends and how to color the path. The procedure is as follows

- You indicate the start of a path by calling the method beginPath() method of the 2D context. Paths are stored internally as a **list of sub-paths** Every time you call this method that list is reset and we can start making new shapes.

- Once you have begun the path you need to specify the actual paths to be drawn. These will be introduced shortly.

- The third step is to close the path once you have specified all the sub paths by calling closePath(). This is optional. If your list of sub paths in the above step specifies a closed path you needn't call this method. If the final shape is open and you call this method it will simply draw a straight line from the last point to the first point.

- The final step is to specify the fill or stroke for the shape. This is the step that makes the shape visible on the canvas. You can call the following methods.
  - The fill() method fills the shape with a solid color (we will see how to change color in a later section). Any open paths will be closed.
  - The stroke() method draws an outlined shape. The stroke weight can be changed, as we will see in a later section.

beginPath()	Starts the path. All methods after are added in order to a list. This does not actually draw anything on the canvas
closePath()	Ends the path. We can stroke or fill the path after this call. Optional as explained above
fill()	Fills the shape with the default color or specified color
stroke()	Creates an outline shape using the default shape and stroke weight which can be changed.

We will now move on to the different paths available and how to draw them. We will explore how to draw shapes using straight lines and then move on to arcs.

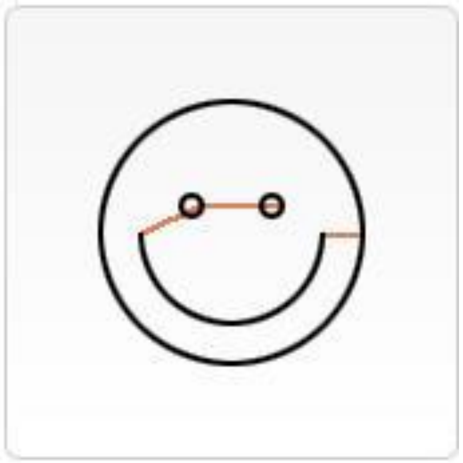
However before we move on to the paths there is one important function to familiarize ourselves with. The function call is moveTo(x,y). The function does not actually draw anything but moves the current position to (x,y) as specified by you. You can probably best think of this as lifting a pen or pencil from one spot on a piece of paper and placing it on the next.

moveTo(x,y)	Moves the starting point to a new point specified by (x,y)

Although moveTo(x,y) does not actually draw anything it is part of the path list since it is essential to draw any shape. Typically when you begin a new path you will want to use the function to specify where the new path starts. Try the example below

```
function draw(){
        var canvas = document.getElementById('canvas'); if
        (canvas.getContext){
             var ctx = canvas.getContext('2d');

             ctx.beginPath(); ctx.arc(75,75,50,0,Math.PI*2,true); //
             Outer circle ctx.moveTo(110,75);
             ctx.arc(75,75,35,0,Math.PI,false); // Mouth (clockwise)
             ctx.moveTo(65,65);
             ctx.arc(60,65,5,0,Math.PI*2,true); // Left eye
             ctx.moveTo(95,65);
             ctx.arc(90,65,5,0,Math.PI*2,true); // Right eye
             ctx.stroke();
        }
   }
```

This produces a smiley face shown below. The red lines show the points where the moveTo function has been used.



Don’t worry about the functions that have not been introduced yet. They are up next.

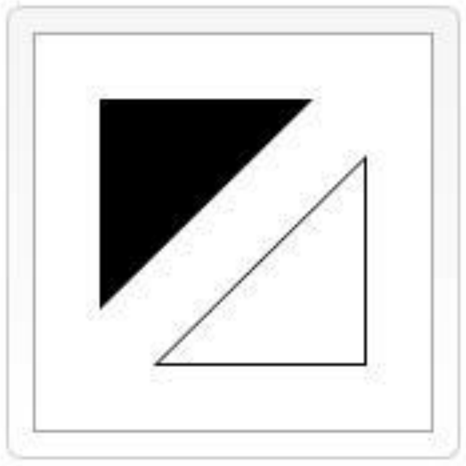
Lines

You can draw lines on the canvas using the lineTo(x,y) function. The specification of the function is given below

lineTo(x,y)	Draws a straight line from current position to point (x,y) on the canvas

The starting point of the line depends on the previous paths drawn. If this is the first path you will want to use moveTo to specify the starting point. Lets look at the example below.

```
function draw(){
     var canvas = document.getElementById('canvas');
     if (canvas.getContext){
          var ctx = canvas.getContext('2d');

          // Filled triangle
          ctx.beginPath();
          ctx.moveTo(25,25);
          ctx.lineTo(105,25)
          ;
          ctx.lineTo(25,105)
          ; ctx.fill();

          // Stroked triangle
          ctx.beginPath();
          ctx.moveTo(125,125);
          ctx.lineTo(125,45);
          ctx.lineTo(45,125);
          ctx.closePath();
          ctx.stroke();
     }
}
```

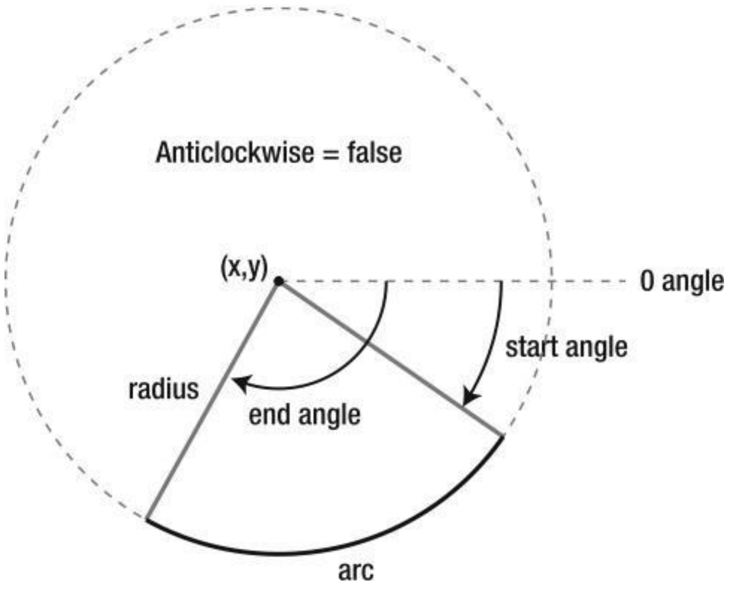


Simple Arcs

Arcs can be drawn on the canvas using the arc() method which takes five parameters

arc(x, y, radius, startAngle, endAngle, anticlockwise)	x and y are the coordinates of the circle's center. Radius is self-explanatory.

	The startAngle and endAngle parameters define the start and end points of the arc in radians. The starting and closing angle are measured from the x axis. The anticlockwise parameter is a Boolean value which when true draws the arc anticlockwise, otherwise in a clockwise direction.



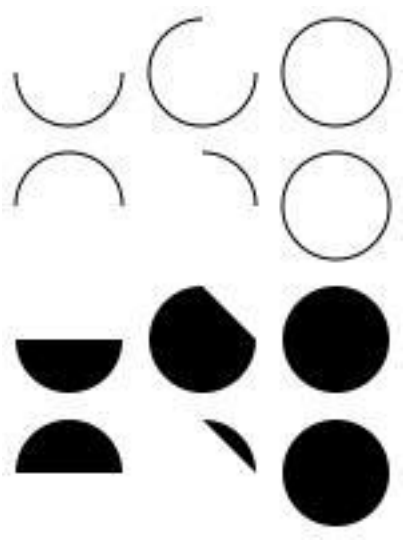
The image above gives a visual representation of how the arc function works. Lets try out an example using the arc function. This example is slightly tedious but try to walk through it step by step. The usage of the arc function will be clear. You are encouraged to play around with the code to understand the function better.

You will need to increase your canvas height to 200 pixels for this example.

```
function draw(){
    var canvas = document.getElementById('canvas'); if
    (canvas.getContext){
        var ctx = canvas.getContext('2d');

        for(var i=0;i<4;i++){
            for(var j=0;j<3;j++){
                ctx.beginPath();

                // x coordinate
                var x = 25+j*50;
                // y coordinate
```

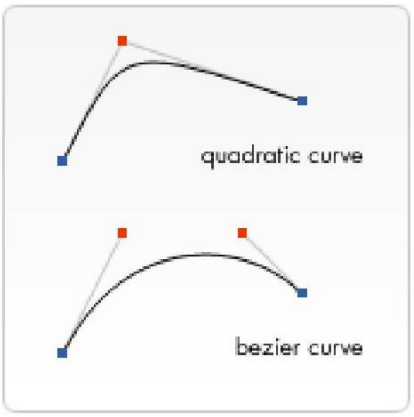
```
                    var y = 25+i*50;
                    // Arc radius var
                    radius = 20;
                    // Starting point on circle
                    var startAngle = 0;
                    // End point on circle
                    var endAngle = Math.PI+(Math.PI*j)/2; //
                    clockwise or anticlockwise
                    var anticlockwise = i%2==0 ? false : true;

                         ctx.arc(x,y,radius,startAngle,endAngle,  anticlockwise);

                    if (i>1){
                        ctx.fill();
                    } else {
                        ctx.stroke();
                    }
                }
            }
        }
}
```

This produces a grid of different arcs with alternate fill style, shown below



Bezier Curves

There are two more complex functions available to draw curves, namely quadratic and Bezier curves. To understand the difference between the two consider the visual alongside. A quadratic curve has one control point (in red) whereas a Bezier curve has two.



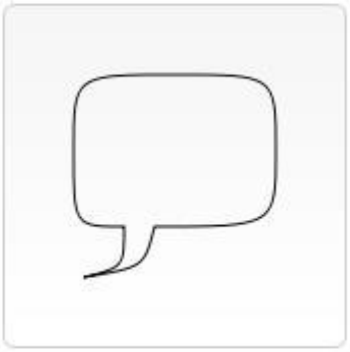
If you have worked with vector graphic software like Adobe Illustrator this concept should not be alien to you. However it can be quite challenging to use these functions since you do not have direct visual feedback on the actions you are taking.

However if you have the patience very complex shapes can be created using these functions. We will work with two simple examples to show the usage of both functions listed below.

quadraticCurveTo(cp1x, cp1y, x,y)	(x, y) specify the end point of the curve starting from the current position. (cp1x, cp1y) specify the control point.
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)	(x, y) specify the end point of the curve starting from the current position. (cp1x, cp1y) and (cp2x, cp2y) specify the control points.

Lets see how to draw shapes using quadratic curve

```
function draw(){
   var canvas = document.getElementById('canvas');
   if (canvas.getContext){
       var ctx = canvas.getContext('2d');

       ctx.beginPath();
       ctx.moveTo(75,25);
       ctx.quadraticCurveTo(25,25,25,62.5);
       ctx.quadraticCurveTo(25,100,50,100);
       ctx.quadraticCurveTo(50,120,30,125);
       ctx.quadraticCurveTo(60,120,65,100);
       ctx.quadraticCurveTo(125,100,125,62.5);
       ctx.quadraticCurveTo(125,25,75,25);
       ctx.stroke();
   }
```



Here is the Bezier curve

```
function draw(){
    var canvas = document.getElementById('canvas'); if
    (canvas.getContext){
        var ctx = canvas.getContext('2d');

        ctx.beginPath();
        ctx.moveTo(75,40);
        ctx.bezierCurveTo(75,37,70,25,50,25);
        ctx.bezierCurveTo(20,25,20,62.5,20,62.5);
        ctx.bezierCurveTo(20,80,40,102,75,120);
        ctx.bezierCurveTo(110,102,130,80,130,62.5);
        ctx.bezierCurveTo(130,62.5,130,25,100,25);
        ctx.bezierCurveTo(85,25,75,37,75,40);
        ctx.fill();
    }
}
```

We have introduced many different methods to draw shapes. There is a lot of scope to combine the methods and create elaborate shapes and curves.

Image Manipulation

The canvas element allows us to import images and manipulate them. This makes the canvas element very versatile in its applicability.

Adding Images to the Canvas

To add images to the canvas we need to define a javascript image object. Directly providing the image url or path in the canvas code will not work. Here is an example to help you better understand the process of importing an image.

We start by defining a JS image object

```
    var img = new Image(); // Create new img element
    img.src = 'myImage.png'; // Set source path
```

The second line specifies the source path of the image relative to the parent file. Once the second line is executed the image starts loading. We need to be told when the image has finished loading so that we can manipulate/display it on the canvas. The image object has a event handler called onload which does exactly that.

```
    var img = new Image();   // Create new img element
    img.onload = function(){
        // execute drawImage statements here
    };
    img.src = 'myImage.png'; // Set source path
```

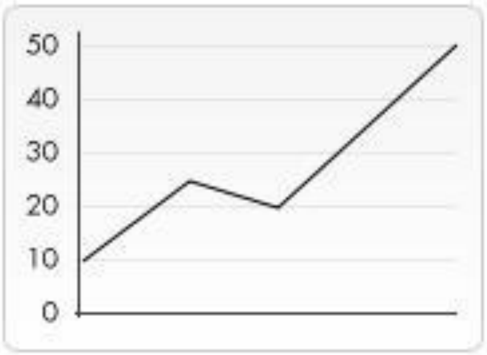
Once myImage.png finishes loading the onload function will be called. That is where we will execute our code for manipulating and displaying the image. We will look at a few methods to manipulate the images shortly. Before that we need to learn how to display the image on the canvas.

Once the image is loaded we can display it on the canvas by calling the drawImage function.

drawImage(image, x, y)	Image refers to the JS image object (x, y) is the point on the canvas where the upper left edge of the image will be placed, thereby specifying the positioning of the image.

Lets look at an example

```
    function draw() {
        var ctx = document.getElementById('canvas').getContext('2d'); var
        img = new Image();
        img.onload = function(){
            ctx.drawImage(img,0,0);
            ctx.beginPath();
            ctx.moveTo(30,96);
            ctx.lineTo(70,66);
            ctx.lineTo(103,76);
            ctx.lineTo(170,15);
            ctx.stroke();
        };
        img.src = 'images/backdrop.png';
    }
```



The example draws a path to visualize the growth rate. The backdrop of the chart is an image 'backdrop.png'. Walk through the example and convince yourself that it produces what is shown above.

In the next two sections we will see that the drawImage function is overloaded with additional parameters to provide additional image manipulation capabilities.

Scaling

The second variant of drawImage adds two new parameters that allow us to place a scaled image on the canvas.

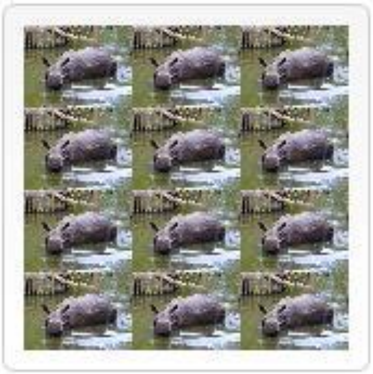
drawImage(image, x, y, width, height)	The first three parameters are the same as before. Width and height are the desired dimension of the scaled image to be placed on the canvas.

In the example that follows we will fill the canvas with scaled down versions of the image below.

Keep in mind that images can become blurry when scaled up too much or text in the image can become illegible when scaled down too much.

```
    function draw() {
            var ctx = document.getElementById('canvas').getContext('2d');
            var img = new Image();
           img.onload = function(){
                for (var i=0;i<4;i++){
                    for (var j=0;j<3;j++){
                        ctx.drawImage(img,j*50,i*38,50,38);
                    }
                }
           };
            img.src = 'images/rhino.jpg';
       }
```

You should see this in the browser

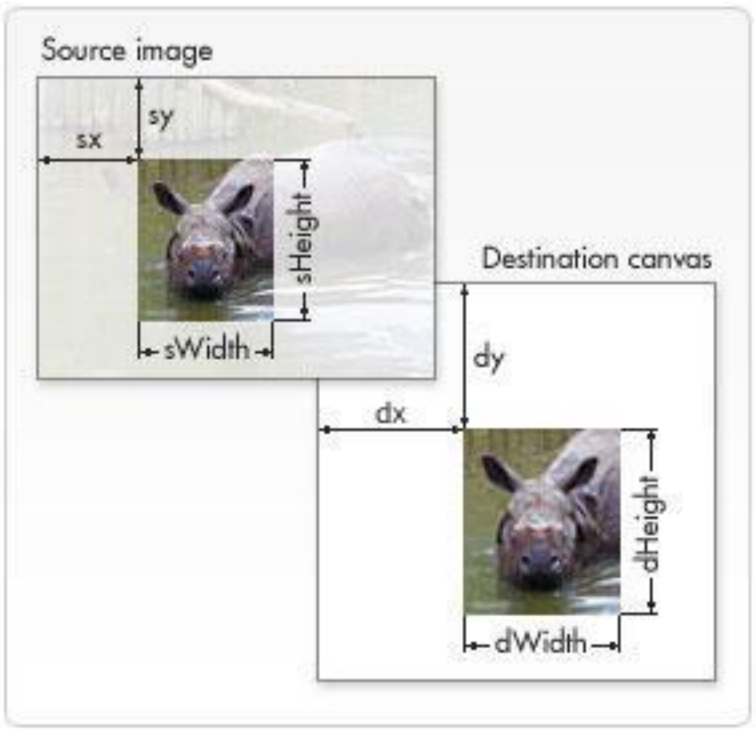


Slicing

The final variant of the drawImage function introduces eight new parameters and allows us to slice parts of an image and draw them on the canvas.

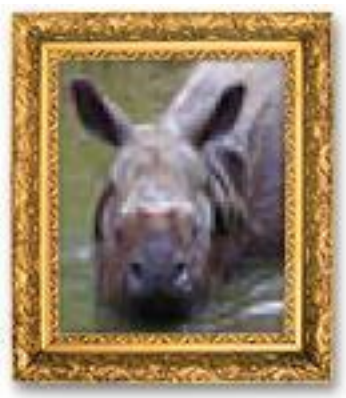
drawImage(image, x, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)	Image refers to the JS image object. (.sy, sy) = refer to the upper-left corner of the slice in the source image. sWidth and sHeight specify the dimension of the slice from the source image. (dx, dy) specify the co-ordinates where the slice should be placed in canvas. dWidth and dHeight allow you to scale the slice before placing it on the canvas.

To better understand the parameters look at the image below. It gives a visual snapshot of what the parameters are and their function.



```
function draw() {
    var canvas = document.getElementById('canvas'); var
    ctx = canvas.getContext('2d');
    var img1 = new Image();
    var img2 = new Image();

    img1.onload = function() {
    // Draw slice
    ctx.drawImage(img1,20,40,50,50,20,20,87,104);

    // Draw frame
    ctx.drawImage(img2,0,0);
    };

    img1.src = 'images/rhino.jpg';
    img2.src = 'images/frame.jpg';
}
```

This result should be like



Adding Color and Style

So till this point our canvas has been pretty black and white. The shapes that we drew used the default settings for color and style. We are now going to see what options are available to brighten up the canvas.

Color

To add color to the canvas there are two properties that can be used. They are the strokeStyle and the fillStyle properties of the rendering context. Remember, these aren't function calls, but properties.

strokeStyle	Specify color using the following methods: • Color name e.g. "red" • Hex representation #FF0000 • Rgb function e.g. rgb(255,0,0) • Rgba function e.g. rgb(255,0,0,0)
fillStyle	

strokeStyle specifies the color for any outlines drawn from the point where you set it. Similarly fillStyle specifies the fill color for all shapes. Once you set the property it becomes default for **all** shapes drawn from that point on. If you want a different color you will have to reassign the property.

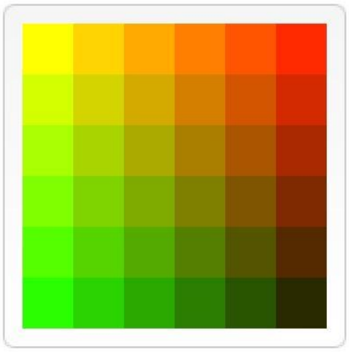
Lets look at two examples. The first one is for fillStyle.

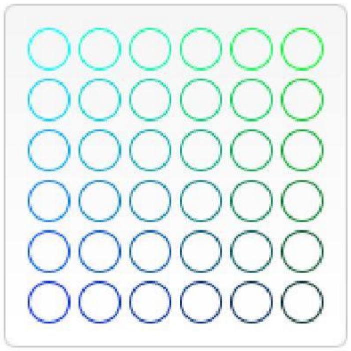
```
function draw() {
    var ctx = document.getElementById('canvas').getContext('2d'); for
    (var i=0;i<6;i++){
        for (var j=0;j<6;j++){
            ctx.fillStyle = 'rgb(' + Math.floor(255-42.5*i) + ',' +
            Math.floor(255-42.5*j) + ',0)';
            ctx.fillRect(j*25,i*25,25,25);
        }
    }
}
```

And for strokeStyle

```
function draw() {
    var ctx = document.getElementById('canvas').getContext('2d'); for
    (var i=0;i<6;i++){
        for (var j=0;j<6;j++){
            ctx.strokeStyle = 'rgb(0,' + Math.floor(255-42.5*i) + ',' +
            Math.floor(255-42.5*j) + ')';
            ctx.beginPath();
            ctx.arc(12.5+j*25,12.5+i*25,10,0,Math.PI*2,true);
            ctx.stroke();
        }
    }
}
```

The result is shown on then next page. Convince yourself that the code above should produce what is shown. You are once again encouraged to play with the code to get a deeper understanding of color manipulation on the canvas.


fillStyle result


strokeStyle result

Transparency

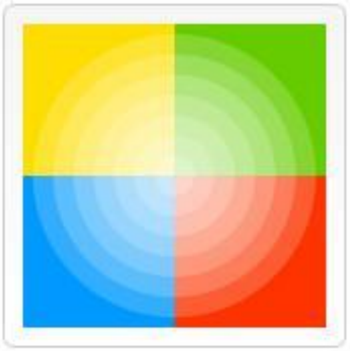
Canvas allows us to draw semi opaque shapes as well. This is possible by changing the value for the globalAlpha property of the 2D context.

globalAlpha	Numerical value ranging from 0.0 (fully transparent) to 1.0 (fully opaque)

This transparency value would be applicable to **all** shapes drawn. In that light a more pragmatic approach would be to change the fillStyle and strokeStyle properties introduced earlier. These properties accept rgba values. Setting the alpha property for these styles will allow you to control the transparency of individual strokes.

Here is an example of the globalAlpha property in use

```
    function draw() {
        var                    ctx                    =
document.getElementById('canvas').getContext( '2d');
        // draw background
        ctx.fillStyle = '#FD0';
        ctx.fillRect(0,0,75,75);
        ctx.fillStyle = '#6C0';
        ctx.fillRect(75,0,75,75);
        ctx.fillStyle = '#09F';
        ctx.fillRect(0,75,75,75);
        ctx.fillStyle = '#F30';
        ctx.fillRect(75,75,75,75)
```

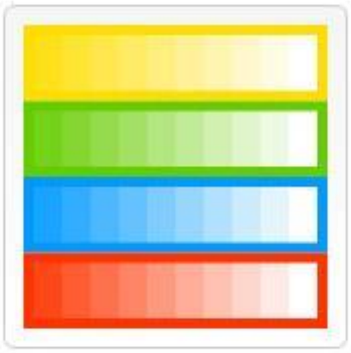


```
        ctx.fillStyle = '#FFF';
        //  set transparency value
        ctx.globalAlpha = 0.2;
        // Draw semi transparent circles
        for (var i=0;i<7;i++){
            ctx.beginPath();
            ctx.arc(75,75,10+10*i,0,Math.PI*2,true);
            ctx.fill();
        }
    }
```

The result is shown alongside the code. The first part draws the background, the four solid squares. The for loop draws concentric circles with a transparency of 0.2

The limitation of this method becomes apparent if we asked you to draw concentric circles of different transparencies. You would have to change the globalAlpha property before drawing each circle. Using rgba we can achieve this easily.

We will slightly change the example now. We will draw rectangles of different transparencies on a background.



```
    function draw() {
        var                   ctx                   =
document.getElementById('canvas').getContext('2d ');

        // Draw background
        ctx.fillStyle = 'rgb(255,221,0)';
        ctx.fillRect(0,0,150,37.5);
        ctx.fillStyle = 'rgb(102,204,0)';
        ctx.fillRect(0,37.5,150,37.5);
        ctx.fillStyle = 'rgb(0,153,255)';
        ctx.fillRect(0,75,150,37.5);
        ctx.fillStyle = 'rgb(255,51,0)';
        ctx.fillRect(0,112.5,150,37.5);

        // Draw semi transparent rectangles
        for (var i=0;i<10;i++){
            ctx.fillStyle = 'rgba(255,255,255,'+(i+1)/10+')';
            for (var j=0;j<4;j++){
                ctx.fillRect(5+i*14,5+j*37.5,14,27.5)
            }

        }

    }
```

Using the for loop we can change the alpha property and draw the rectangles by assigning the fillyStyle property of the context.

Gradients

Another nifty feature of the canvas element is the ability to create gradients. There are a few steps involved in creating a gradient.

- You need to create a canvasGradient object using one of two methods. One produces a linear gradient while the other produces a radial gradient.

createLinearGradient(x1, y1, x2, y2)	(x1, y1) specifies the starting point of the gradient and (x2, y2) specifies the end.
createRadialGradient(x1, y1, r1, x2, y2, r2)	(x1, y1) specifies the center of the starting circle with radius r1 and (x2, y2) specifies the ending circle with radius r2.

- Once you have created the gradient object we need to specify the colors in the gradient. We can add as many colors as we want using the addColorStop method.

addColorStop(position, color)	position is a numerical value between 0.0 and 1.0 that specifies the relative position of the color in the gradient. color specifies what the color to be used for that stop is

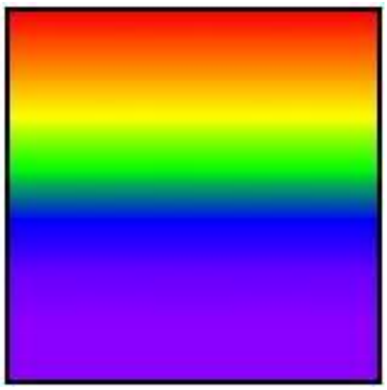
- You can then assign a canvasGradient object to a fillStyle or strokeStyle to draw the shapes.

Lets look at an example to see how this works

We will create a rainbow in the canvas. For this we need to specify the seven different colors with each color merging into the next. Here is the code

```
function draw() {
     var ctx = document.getElementById('canvas').getContext('2d');

     // Create gradients
     var lingrad = ctx.createLinearGradient(0,0,0,150);
     lingrad.addColorStop(0, '#FF0000');
```

```
        lingrad.addColorStop(0.142,     '#FF7F00');
        lingrad.addColorStop(0.284,     '#FFFF00');
        lingrad.addColorStop(0.426,     '#00FF00');
        lingrad.addColorStop(0.568,     '#0000FF');
        lingrad.addColorStop(0.710,     '#6600FF');
        lingrad.addColorStop(0.852, '#8B00FF');

        // assign  gradients  to  fill  styles
        ctx.fillStyle = lingrad;

        // draw shapes
        ctx.fillRect(0,0,150,150);
 }
```



Note the different position values. Each color should occupy 14.2% of the canvas, which is why the stop increases with a value of 0.142 every step. The colors are specified using the hex values (obtained from Wikipedia). The result is shown above.

We will leave it to you to explore radial gradients.

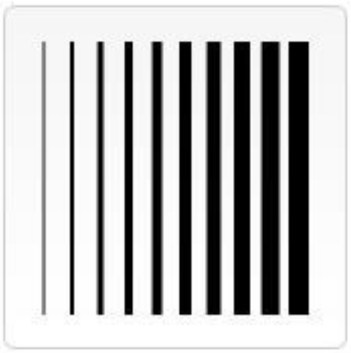
Stroke

We can make changes to a bunch of context properties to manipulate the stroke style of lines and paths. Some of these properties are

lineWidth	Specifies the width of the stroke in pixels. Default is 1.0
lineCap	Determines how the end point of a line are drawn
lineJoin	Determines how two line segments are joined

We will explore the first two in this lab. Lets jump in with examples. We will first introduce the lineWidth property

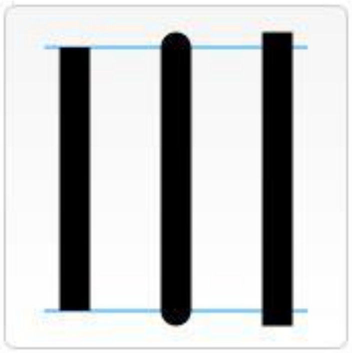
```
 function draw() {
            var                   ctx                    =
 document.getElementById('canvas').getContext('
 2d');
            for (var i = 0; i < 10; i++){
                 ctx.lineWidth = 1+i;
                 ctx.beginPath();
                 ctx.moveTo(5+i*14,5);
                 ctx.lineTo(5+i*14,140);
                 ctx.stroke();
            }
        }
```



The image on the right shows what you should see in the canvas. The code should be self-explanatory by now.

For the lineCap property there are three valid values. They are 'butt', 'round' and 'square'. The example shows the difference between the three

```
function draw() {
        var                    ctx                    =
document.getElementById('canvas').getContext( '2d');
        var lineCap = ['butt','round','square'];

        // Draw guides
        ctx.strokeStyle = '#09f';
        ctx.beginPath();
        ctx.moveTo(10,10);
        ctx.lineTo(140,10);
        ctx.moveTo(10,140);
        ctx.lineTo(140,140);
        ctx.stroke();

        // Draw lines
        ctx.strokeStyle = 'black';
        for   (var   i=0;i<lineCap.length;i++){
            ctx.lineWidth = 15;
            ctx.lineCap = lineCap[i];
            ctx.beginPath();
            ctx.moveTo(25+i*50,10);
            ctx.lineTo(25+i*50,140);
            ctx.stroke();
        }
    }
```



The first portion draws the blue lines you see in the image so that the difference in the line caps is clear.

That brings us to the end of the section on color and style. If you are interested there are more properties that can be explored, such as shadows and patterns. Google it.

Transformation

Co-ordinate system transformations are a very powerful idea. They allow us to make complex shapes using relatively simple code over what we would need without transformations. In this section we will explore the two basic transformations, translation and rotation.

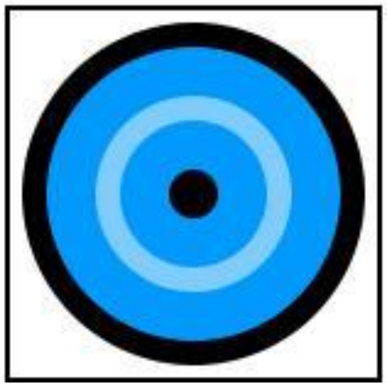
Canvas State

Before we jump into the transformations, we need to familiarize ourselves with the notion of a canvas state. The canvas state includes values for all the properties we have discusses so far like fillStyle, globalAlpha, lineWidth etc. Canvas states can be saved and restored. This makes life easier while performing transformations because we need not perform the reverse transformation to go back to the original state. You will see this in an example shortly. Let's look at the methods to save and restore the canvas state first

save()	Saves the canvas state by pushing the state on a stack
restore()	Restores the last saved state by popping from the stack

A stack is used because you can save the state multiple times before restoring. The save/restore follows a Last In First Out (LIFO) ordering. Lets look at a simple example using the state to draw concentric rectangles and how things become simpler by saving and restoring the state.

```
 function draw() {
     var ctx = document.getElementById('canvas').getContext('2d');

     //Draw a circle with the default settings
     ctx.arc(75,75,70,0,2*Math.PI,false);
     ctx.fill();
     ctx.save();

     ctx.fillStyle = '#09F'          // Make changes to the settings
     ctx.beginPath();                 // Save the default state
     //Draw a circle with 2nd state
     ctx.arc(75,75,60,0,2*Math.PI,false);
     ctx.fill();

     ctx.save();                     // Save the 2nd   state
     ctx.fillStyle =  '#FFF'         // Make changes   to the settings
     ctx.globalAlpha  = 0.5;
     ctx.beginPath();
     //Draw a circle with the 3rd state
     ctx.arc(75,75,40,0,2*Math.PI,false);
     ctx.fill();

     ctx.restore();                  // Restore 2nd state
     ctx.beginPath();
     //Draw a circle. Will be drawn with settings
 of the second state
    ctx.arc(75,75,30,0,2*Math.PI,false);
    ctx.fill();

    ctx.restore();                      // Restore
    default state
    ctx.beginPath();
    ctx.arc(75,75,10,0,2*Math.PI,false);
    ctx.fill();
 }
```



This should produce the shape on the right.

Translating

The first transformation we will look at is translating the canvas origins. This transformation allows us to construct complex shapes without having to explicitly calculating the coordinates of different points that would be cumbersome to do as you will see shortly.

It's a good idea to save the canvas state before you translate the coordinates so that you can go back to the original state easily. This is especially critical if you intend on making multiple translations. It is easier to make a translation by reverting back to the original state and then making a new translation. You will see this in action in the example below.

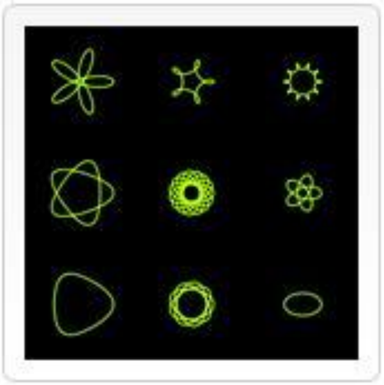
We can translate using the function below

translate(x, y)	Translates the canvas coordinate system by shifting the origin to (x, y) coordinate system.

In this example we will draw nine Spiro graphs (http://en.wikipedia.org/wiki/Spirograph) on the canvas by shifting the canvas coordinates to place the origin at the point where we want each Spiro graph to be drawn.

To draw the Spiro graph we have defined a separate function. You need to concentrate only on the draw() function. While you do not need to know the details of how to draw a Spiro graph you are welcome to explore the method if you are interested. You can make elegant visuals using Spiro graphs. The Wikipedia link above will help you understand more about Spiro graphs.

Remember to change the canvas dimensions to 300X300 before trying this example out.



```
function draw() {
     var ctx = document.getElementById('canvas').getContext('2d');
     ctx.fillRect(0,0,300,300);
     for (var i=0;i<3;i++) {
          for (var j=0;j<3;j++) { //save
               the default state
               ctx.save();
               ctx.strokeStyle = "#9CFF00";
               //translate the origin to the
center of the next Spiro
               ctx.translate(50+j*100,50+i*100);

drawSpirograph(ctx,20*(j+2)/(j+1),-
8*(i+3)/(i+1),10);
               //restore the state to prepare for
the next translation
               ctx.restore();
          }
     }
}
function drawSpirograph(ctx,R,r,O){
     var x1 = R-O;
     var y1 = 0;
     var i = 1;
     ctx.beginPath();
     ctx.moveTo(x1,y1);
     do {
          if (i>20000) break;
          var x2 = (R+r)*Math.cos(i*Math.PI/72) -
(r+O)*Math.cos(((R+r)/r)*(i*Math.PI/72))
          var y2 = (R+r)*Math.sin(i*Math.PI/72) -
(r+O)*Math.sin(((R+r)/r)*(i*Math.PI/72))
          ctx.lineTo(x2,y2);
          x1 = x2;
          y1 = y2;
          i++;
     } while (x2 != R-O && y2 != 0 );
     ctx.stroke();
}
```

The translating frees us of the requirement to calculate the center of every Spiro graph explicitly in the drawSpirograph function. It just has to draw around the origin. Also convince yourself that if we did not restore the state at the end of every loop

Rotating

The rotation transformation has a similar function to the translation transformation. It makes the code for drawing, what would have been difficult, much simpler and intuitive once you get the grasp of it.

In this example we are going to draw the image on the right. What you should determine is that it is possible to draw all those dots by explicitly calculating the center coordinates of each dot using the sine and cosine functions and the angle with respect to the x-axis. You should try and be convinced that it is a cumbersome method.

rotate(angle)	Rotates the canvas by ‘angle’ **radians**

You will now see how simple it becomes with the rotation transformation.

```
function draw() {
    var ctx = document.getElementById('canvas').getContext('2d');
    ctx.translate(75,75);

    for (var i=1;i<6;i++){ // Loop through rings (from inside to out)
        ctx.save();
        ctx.fillStyle = 'rgb('+(40*i)+','+(255-40*i)+',255)';

        for (var j=0;j<i*6;j++){ // draw individual dots
            ctx.rotate(Math.PI*2/(i*6)); ctx.beginPath();
            ctx.arc(0,i*12.5,5,0,Math.PI*2,true);
            ctx.fill();
        }

        ctx.restore();
    }
}
```

The outer for loop counts for each of the six circles. The inner loop draws the individual dots. The innermost ring has six dots and thus the angular distance between each dot is 360/6 that is 60 degrees. The canvas is rotated by that amount incrementally with every execution of the inner loop. With each circle growing outwards, the number of dots doubles and thus the angular distance decreases by half.

Before moving to the next loop the canvas is restored to the original state. Note that in this case since each circle completes an entire turn the state restoration does not do anything. Confirm that. This will **not** be true in general.

Scaling

The last transformation that we will look at is the scaling transformation. The function to achieve scaling is

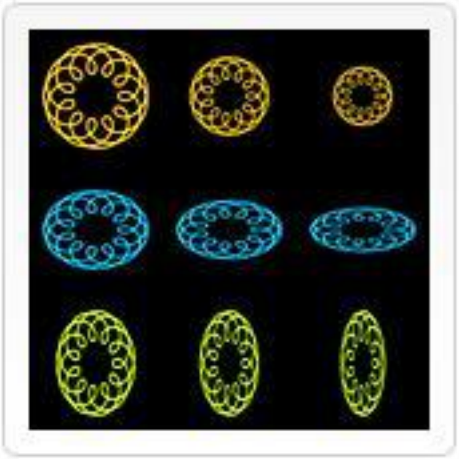
scale(x, y)	Scales the canvas by x-axis scale factor specified by the parameter x and the y-axis scale factor specified by y. The scale factors must be real numbers and they can be negative.

Scale factor values larger than 1.0 result in larger sizes whereas
positive values less than 1.0 result in shrinkage. Negative numbers result in axis mirroring. In the following example we draw one Spiro graph nine times with different scaling values. Instead of using a for loop we have drawn each one separately so that you can understand better. There are multiple transformations in play at once. Play with the code and try to understand what each transformation is doing. Keep track of the context save and restores.

It looks like a lot of code but do not worry. Once you get the hang of it, it will be simple.

Only the draw function is included here. You will need to include the drawSpirograph function from the previous section.

```
function draw() {
    var ctx = document.getElementById('canvas').getContext('2d');
    ctx.strokeStyle = "#fc0";
    ctx.lineWidth = 1.5;
    ctx.fillRect(0,0,300,300);

    // Uniform scaling
    ctx.save()
    ctx.translate(50,50);
    drawSpirograph(ctx,22,6,5);        //  no
scaling

    ctx.translate(100,0);
    ctx.scale(0.75,0.75);
    drawSpirograph(ctx,22,6,5);

    ctx.translate(133.333,0);
    ctx.scale(0.75,0.75)
```



```
        drawSpirograph(ctx,22,6,5);
        ctx.restore();

        // Non-uniform scaling (y direction)
        ctx.strokeStyle = "#0cf";
        ctx.save()
        ctx.translate(50,150);
        ctx.scale(1,0.75);
        drawSpirograph(ctx,22,6,5);

        ctx.translate(100,0);
        ctx.scale(1,0.75);
        drawSpirograph(ctx,22,6,5);

        ctx.translate(100,0);
        ctx.scale(1,0.75);
        drawSpirograph(ctx,22,6,5);
        ctx.restore();

        // Non-uniform scaling (x direction)
        ctx.strokeStyle = "#cf0";
        ctx.save()
        ctx.translate(50,250);
        ctx.scale(0.75,1);
        drawSpirograph(ctx,22,6,5);

        ctx.translate(133.333,0);
        ctx.scale(0.75,1);
        drawSpirograph(ctx,22,6,5);

        ctx.translate(177.777,0);
        ctx.scale(0.75,1);
        drawSpirograph(ctx,22,6,5);
        ctx.restore();

    }
```

That brings us to the end of the transformations. Now onto the last section ☺

Basic Animations

In this section we will introduce the very basics of animation in the canvas. Most of the skills you have picked up till now will be used in this section.

There are few basic steps involved in the animation given here. Please note that there are many ways to animate on the canvas and you should explore different ways to animate on the canvas. In this example

1. You need to clear the canvas. The easiest way to do this is to use the clearRect method.
2. You need to save the canvas state before you apply any transformations. This is so that you can restore the original canvas state so that the next cycle in the animation can be started on a clean slate.
3. Your rest of the draw() method comes next. This is the heart of the animation. Each frame of the animation is rendered using this function. Various transformations and images are drawn here.
4. The original canvas state is restored.
5. The animation is started by calling the init() function. The init() function has a method called setInterval which takes as parameter a custom function which draws the animation content and a time interval which dictates the interval with which the animation is to be drawn.

setInterval(function, time interval)	the function every x milliseconds specified by the 'time interval' parameter

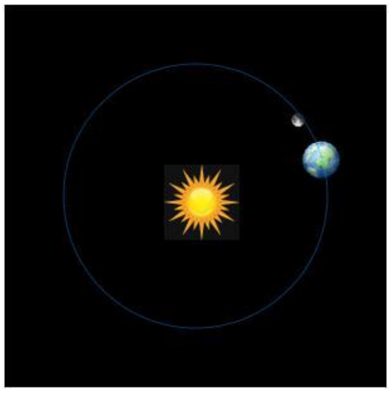
The following code should be included in the <script> </script> section

```
//Create the JS image objects for the earth, moon and sun
var sun = new Image(); var
moon = new Image(); var
earth = new Image();

function init(){
     sun.src = 'images/sun.png';
     moon.src = 'images/moon.png';
     earth.src = 'images/earth.png';
     setInterval(draw,10 );
}

function draw() {
     var ctx = document.getElementById('canvas').getContext('2d');

     ctx.clearRect(0,0,300,300); // clear canvas

     ctx.fillStyle = 'rgba(0,0,0,1)';
     ctx.fillRect(0,0,300,300);

     ctx.fillStyle = 'rgba(0,0,0,0.3)';
     ctx.strokeStyle = 'rgba(0,153,255,0.4)';
     ctx.save();

     //Draws the Sun
     ctx.translate(150,150);
     ctx.drawImage(sun,-25,-25,60,60);
```

```
        // Draws the Earth
        var time = new Date();
        ctx.rotate( ((2*Math.PI)/60 )*time.getSeconds() +
((2*Math.PI)/60000)*time.getMilliseconds() );
        ctx.translate(105,0);
        ctx.fillRect(0,-12,50,24); // Shadow
        ctx.drawImage(earth,-12,-12);

        // Draws the Moon
        ctx.save();
        ctx.rotate( ((2*Math.PI)/6 )*time.getSeconds() +
((2*Math.PI)/6000)*time.getMilliseconds() );
        ctx.translate(0,28.5);
        ctx.drawImage(moon,-3.5,-3.5,15,15);
        ctx.restore();

        ctx.restore();


        ctx.beginPath();
       // draws the Earth orbit
        ctx.arc(150,150,105,0,Math.PI*2,false);
        ctx.stroke();
    }
    init(); //Starts the animation
```

The rotation transformations control how long it will take the earth to complete one rotation around the sun and the moon around the earth. In this scenario it takes the earth 60 seconds to complete one revolution around the earth. In the same time the moon completes 10 revolutions around the earth.

The translation transformation is applied after the rotation transformation and controls how far out the earth is drawn from the sun and the moon from the earth. You will notice that the earth's orbit arc has a radius of 105 and the translation method for the earth has a x component of 105 as well.



Changing the time interval in the setInterval method will make the animation finer or coarser. Increase the time interval to 1000 milliseconds and see what how it affects the animation.

Change the different values of the animation and try to determine what each of the parameters is doing.

Partner Technologies and Trends

We have just scratched the surface of the plethora of possibilities made available by HTML5. But that's not all. The experience can be made much richer by coupling the power of HTML5 with other bleeding edge technologies like the ones we will introduce in this section. This is simply an overview of other technologies you would have to be familiar with if web design and development interests you.

CSS3

Most of you will be familiar with the idea of Cascading Style Sheets or CSS. CSS is to HTML what butter (or whatever is your favorite condiment) is to bread. CSS3 is the newest specification of CSS and brings with it amazing abilities. CSS3 follows a modular structure that gives it a significant advantage over its predecessors. There are over 40 modules in the CSS3 specification such as borders, shadows, transformations and translations, media queries (introduced next).

Many of the modules are close to being certified as web standards but there are others that are still experimental, although widely supported. You should understand that even though experimental there is wide scale adoption of CSS3 since Apple and Google spearhead it. You should also be mindful of testing your applications extensively on different operating systems and browsers if you are using CSS3 to ensure compatibility. Fallback content must be provided for users who are on older browsers otherwise your application would not be available for them.

To give you a glimpse of the amazing things one can do with CSS3, go over to http://www.apple.com/iphone/ and look at the animation happening on the page. This is all done using HTML5, Javascript but mainly CSS3. It uses a cubic Bezier curve we introduced to you before. Now go over to http://johnbhall.com/iphone-4s/ to see how the animation really works (screenshot below).

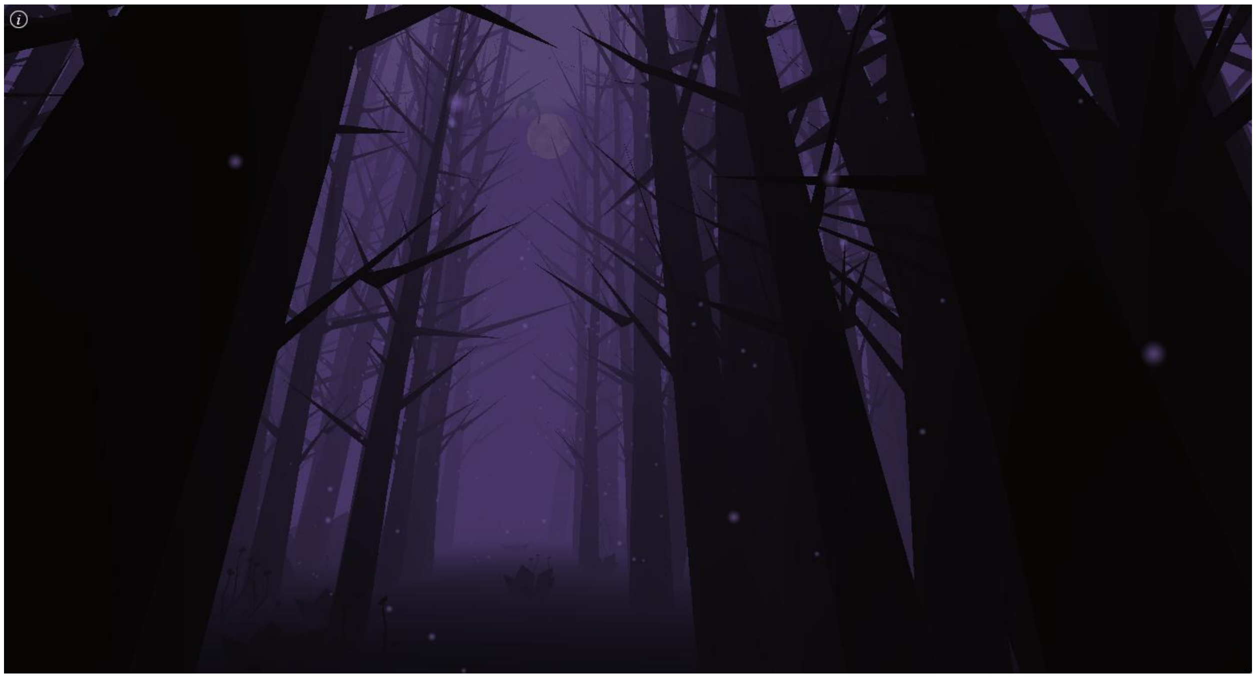
WebGL

If you remember, when we introduced the 2D rendering context we mentioned that there exists an experimental 3D context that used WebGL. WebGL is based on OpenGL and is optimized for browsers. It opens up a fascinating realm of possibilities. You are encouraged to explore WebGL and use it in your projects if you want to be bold.

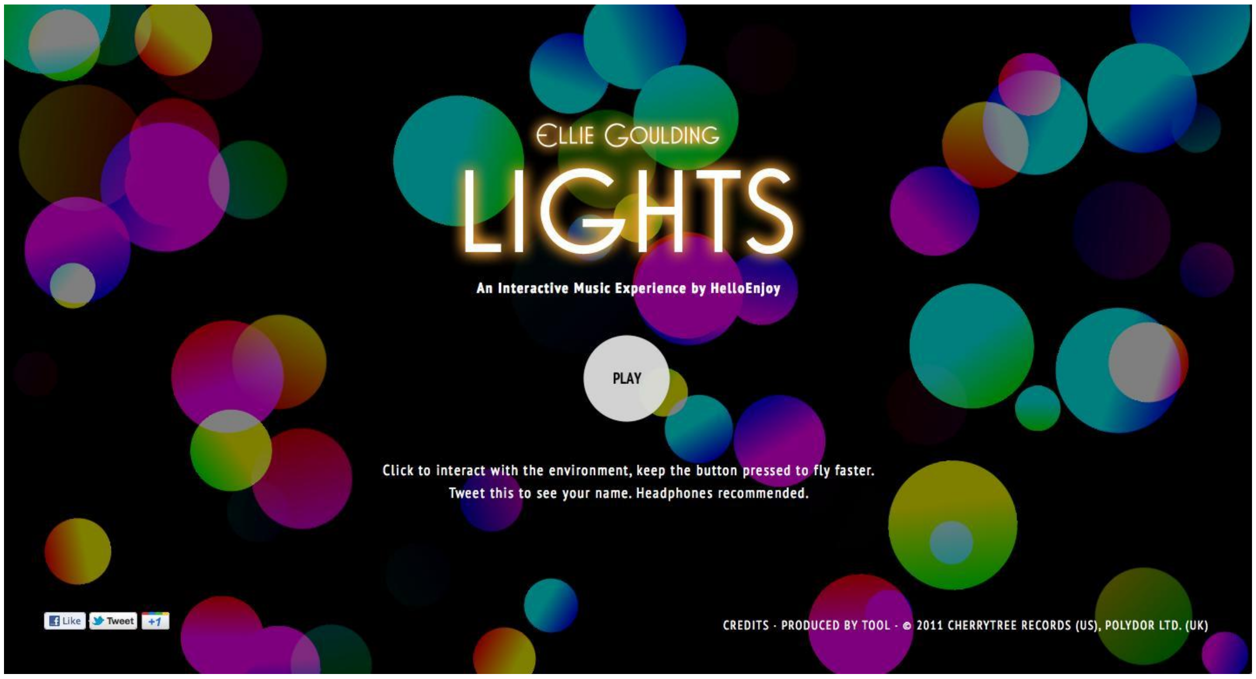
One of the biggest adoption of WebGL came only a few weeks ago when Google updated Google Maps to appear more 3 dimensional without the need of any sort of plugin. The technology behind it is something Google calls MapGL and is based on WebGL.

Here are some other cool examples of WebGL in action. You will need a modern browser to view most of them (latest builds of chrome of firefox recommended). Click on the name to go to the web page

Endless Forest (http://oos.moxiecode.com/js_webgl/forest/index.html)



Lights (http://lights.elliegoulding.com/)


ELLIE GOULDING
LIGHTS
An Interactive Music Experience by HelloEnjoy
PLAY
Like
Tweet
+1
CREDITS · PRODUCED BY TOOL · © 2011 CHERRYTREE RECORDS (US), POLYDOR LTD. (UK)

Click to interact with the environment, keep the button pressed to fly faster.
Tweet this to see your name. Headphones recommended.

Responsive Web Design

The final technology, or I should say idea, we will introduce is Responsive Web Design. This is a practice that started out as a fad but is rapidly becoming a centerpiece of modern web design. It dictates ideas and practices in order to make a website available ubiquitously on different browsers with varying sizes. You will understand that this is very significant since computers aren't the only devices today with a browser. Most significantly there are tablets and smartphones that come with a number of different screen dimensions. Having to build a website for each of these individually is a very tedious task.

Responsive web design solves this problem using media queries, a module under CSS3. If you design your website using responsive ideas then it will automatically resize and rearrange elements on your webpage to fit sizes of any dimensions. Too see this in action go to any of the following sites and slowly resize your browser window

- http://thinkvitamin.com
- http://www.bostonglobe.com/
- http://teegallery.com/
- http://www.space150.com/

Other Resources

There are many good resources available on the Internet free of charge for anyone who wants to learn HTML5 and other related technologies. Some of them are listed below

http://thinkvitamin.com/

http://www.w3schools.com

http://diveintohtml5.info/

http://html5boilerplate.com/

http://cssdeck.com/

To find out level of adoption of HTML5 and CSS3 by different browsers go here

http://html5readiness.com/

Please approach me if you want information on something specific and I will be glad to help.