Canvas

Canvas:

If we want to draw onto Canvas we need to define a Canvas element onto our page. We do so by inserting

```
<canvas id="drawingSurface" width="600px"
height="600px" style="border-style: solid"> </canvas>
```

into the body of our HTML file.
Notice the 4 attributes/properties of the tag. The **width** and **height** attributes are visual properties and defines how big it appears on the page. The **style** property, defines how the canvas will appear on the page. In this case, we gave it a border so we can easily locate it on the page.
The **id** will be the name we will use to refer to the canvas element in our JavaScript code. The **id** in this case is `drawingSurface`. If we had more than one Canvas, each Canvas should have an unique **id**.

Scripts:

It is good practice to include JavaScript code in separate files with the extension **.js**. We can include our script into our HTML files, using the **<script>** tag.

```
<script src="lab3.js"></script>
```

The above tag will insert the script **`lab3.js`** into the body of the HTML file. The preceding element should appear after the canvas element and before `</body>`. The web browser need to create the canvas element before we can reference it, therefore the script needed to be inserted after the canvas element.

Drawing on Canvas Via JavaScript and built-in functions

JavaScript has a lot of code that have been prewritten for us. This code is organized into logical units called functions. Each function does one task and we can use many functions to complete bigger tasks. Each of these functions has a name and we can refer to the functions through their names. These pre-existing functions are known as built-in functions. In a couple of weeks, we will learn to write our own functions. Similarly, there are built-in values in JavaScript. We have previously learned to define our own variables; there are pre-existing variables that hold special values, which we can refer to by their name. An example is `Math.PI`, with holds the special mathematical constant π.

To draw to Canvas, we must get what is known as the **render context** and assign it to a variable, so we can refer to it throughout our program. The render context is an object, which is a special **type** in JavaScript different than that of a String or Number.

Getting the render context

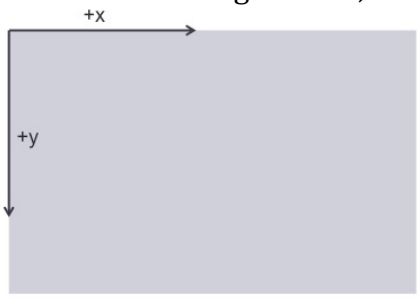
```
var surface = document.getElementById("drawingSurface");
var renderContext = surface.getContext("2d");
```

In the above line of code, we refer to `"drawingSurface"`. We have given the canvas the **id** `"drawingSurface"` in our webpage. From the drawing surface we get the **render context.** Each Canvas has it's own render context. In the above example, getContext is a built-in function associated with the Canvas. Because each of the function is associated with an object, the function is called a method, rather than a function. We will explain more about functions in a couple of weeks.

For a complete reference of the render context:
https://developer.mozilla.org/en/docs/Web/API/CanvasRenderingContext2D

Drawing on the Canvas:

With the rendering context, we can draw to the canvas.



The canvas coordinate system has the origin in the upper left hand corner. The positive x-direction is to the right, and the positive y-direction points downward.

fillRect

The `fillRect` method on the render context is used to draw a rectangle; this is filled with the current color.

`fillrect(x,y,w,h)`	
`x`	The x-coordinate of the upper left corner
`y`	The y-coordinate of the upper left corner
`w`	The width of the rectangle
`h`	The height of the rectangle

As an example, to draw a filled rectangle we can use cause

```
renderContext.fillStyle = "red";
renderContext.fillRect(0,0, 100,100);
```

The above code will draw a red rectangle with the upper right coordinate at (0,0). The first line

```
renderContext.fillStyle = "red";
```

changes the current color. `fillStyle` is a property so it's used like a variable, we can assign it values or access it's values. `fillrect` is a method, so it's used differently than `fillStyle`.

Remember that the Canvas coordinate system is upside down, that is the positive y direction is pointed downwards.

Drawing polygons

Polygons are drawn as paths. To draw a path:

1. Construct the path inside the computer memory
2. Tell the javaScript to draw the path in memory.

To draw a triangular path, that is a polygon with no fill.

```
renderContext.beginPath();
renderContext.lineTo(0,0);
renderContext.lineTo(100,0);
renderContext.lineTo(0,100);
renderContext(0,0);
renderContext.stroke();
```

The **beginPath** resets the drawing path. Inside the rederingContext is a path or list of points. The **beginPath** command empties this list. If you do not use **beginPath**, your program may appear to work, but it may have strange behavior at some point.

lineTo adds a point to the list of points inside the rendering context, and specifies that that point is connected to the previous point.

`lineTo(x,y)`	
x	x-coordinate to add
y	y-coordinate to add

The **stroke** method causes the path defined so far to be drawn.

Alternatively if we wanted to draw a red triangle with a blue border we can write:

```
renderContext.beginPath();
renderContext.lineTo(0,0);
renderContext.lineTo(100,0);
```

```
renderContext.lineTo(0,100);
renderContext(0,0);
renderContext.fillStyle = "red";
renderContext.fill();
renderContext.strokeStyle = "blue";
renderContext.stroke();
```

We've added a couple of commands. **strokeStyle** property allows us to control the color of the path drawn with the **stroke** method. The **fillStyle** property determines what color the **fill** method will use. The **fill** method draws the polygon in memory as a filled polygon.

Drawing a circle

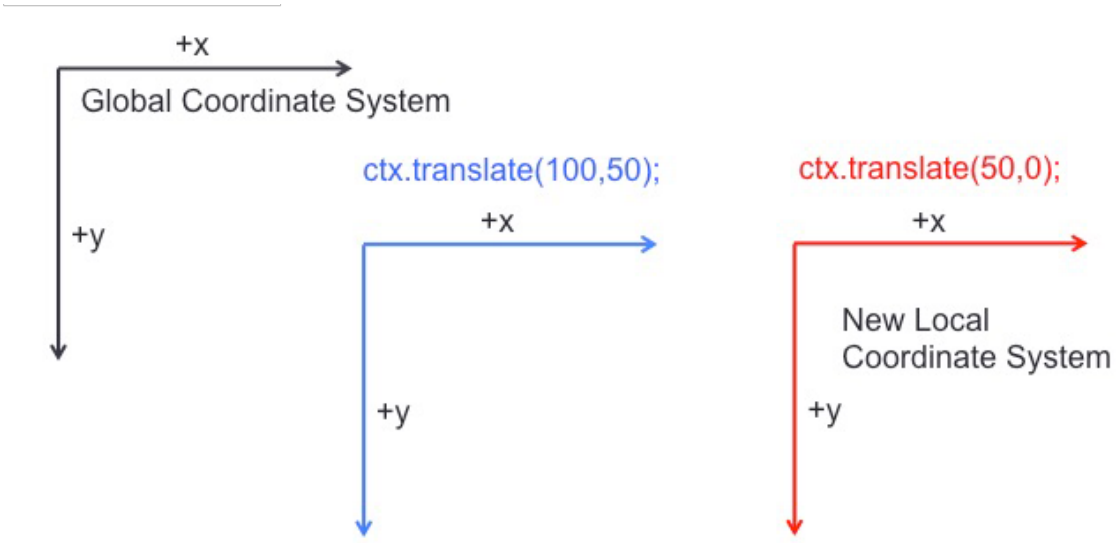
To draw a circle we need the arc method. The **arc** method

`arc(x,y,r,startAngle, endAngle)`	
`x`	x-coordinate of the center of the arc
`y`	y-coordinate of the center of the arc
`r`	radius of the arc
`startAngle`	starting angle in radians
`endAngle`	ending angle in radians

Translate, save, restore

For the canvas it is possible to move the origin and have subsequent commands use the new origin location.

`translate(x,y)`	
x	The amount to move the origin in the x direction.
y	The amount to move the origin in the y direction.

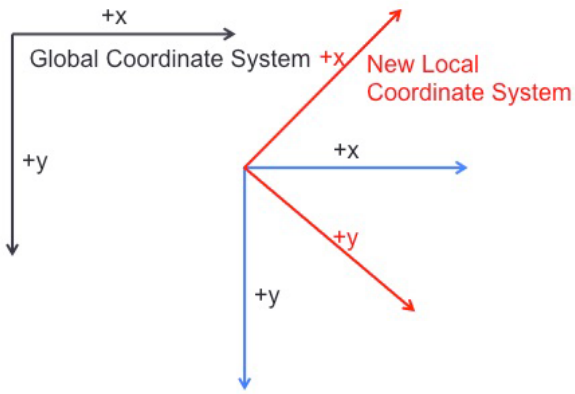


Rotate

Rotate is a bit more abstract to understand. Rotate, does exactly the name suggest. It rotates the coordinate system. Rotate allows use to draw circularly symmetric shapes or patterns easily.

`rotate(angle)`	
angle	angle to rotate the coordinate system in radians.

```
ctx.translate(100,50);
ctx.rotate(-45*Math.PI/180);
```



In the above diagram, first a translate is performed followed by a rotate. Notice how the x-axis and y-axis is not longer parallel to the original axes, because we have rotated the coordinate system.

Save and Restore

Both translate and rotate changes the coordinate system. Often we only want these commands to have an effect over a small number of drawing commands. The **save**

method then saves the location of the current origin and it's orientation to a stack. The **restore** command, removes the last saved origin and orientation from the stack and set it to the current origin and orientation. In theory, every save should be matched by a restore.

resetTransform

In addition to **save** and **restore** there is **resetTransform**. resetTransform resets the origin to the upper left and corner, with the positive x direction point right and the positive y direction pointing down.

Comments:

It is important to add comments to you code.
JavaScript comments are preceded by a double slash, //, in which anything after the double slash on the same line is ignored by the interpreter. Comments can be used to label the purpose each section of code, or provide clarification on what each line of code does.

Example 1 demonstrates the drawing of a triangle. The code demonstrates **save** and restore. Notice that save comes before the **translate** command and restore comes after the **stroke** command.

Example 1

Lab3demo.html:

```
<!DOCTTYPE html>
<html>
<head>
   <title> Canvas Drawing Sample </title>
   <meta charset="UTF-8">
   <script src="Lab3Sample.js" defer></script>
</head>
<body>
   <h1> Lab 3: HTML Canvas drawing example File </h1>
   <canvas id="drawingSurface"
           width="400" height="400"></canvas>
</body>
```

Lab3Sample.js:

```
//Lab 3 Sample
//Author : Kim Lam
//Get the reference to the canvas and drawing context
var drawingSurface=document.getElementById("drawingSurface");
var ctx = drawingSurface.getContext("2d");

//draw a simple triangle
ctx.save();
ctx.translate(100,100);
ctx.beginPath();
ctx.lineTo(100,100);
ctx.lineTo(100,0);
```

```
ctx.lineTo(0,100);
ctx.lineTo(100,100);
ctx.stroke();
ctx.restore();
```

Example 2

Demonstrates how to draw circles, as well as translate.

filename: circleExample.html

```
<!DOCTYPE html>
<head>
   <title>Example of using Circle</title>
   <meta charset="UTF-8">
   <script src="circleExample.js" defer></script>
</head>
<body>
      <h1>Circle Drawing Example</h1>
      <canvas id="drawSurface" width="400px" height="400px"
                  style="border : solid"></canvas>
</body>
```

filename: circleExample.js

```
//Example for drawing Circles
//Author: Kim Lam

//Get the render context
var canvas = document.getElementById("drawSurface");
var ctx = canvas.getContext("2d");


//Example of drawing circles to
//position them.

ctx.save();
//Move the origin to the center of the canvas.
ctx.translate(canvas.width/2, canvas.height/2);

//draws a circle of radius 100
//With the current color
ctx.beginPath();
ctx.arc(0,0,100,0,2*Math.PI);
ctx.stroke();

//Move the coordinate system to the right by 10
ctx.translate(10,0);
//Draw a red circle with radious 90
ctx.beginPath();
ctx.strokeStyle = "red";
ctx.arc(0,0,90,0,2*Math.PI);
ctx.stroke();

//Move the coordinate system again by 45
ctx.translate(45,0);
//Draw a blue filled circle with radius 45
ctx.beginPath();
ctx.fillStyle = "blue";
ctx.arc(0,0,45,0,2*Math.PI);
```

```
ctx.fill();

ctx.restore();
```

Example 3:Rotate example

Demonstrates rotation

filename: rotateExample.html

```
<!DOCTYPE html>
<head>
      <title>Rotate Example</title>
      <meta charset="UTF-8">
      <script src="rotateExample.js" defer></script>
</head>
<body>
      <h1>Rotate Example</h1>
      <canvas id="drawSurface" width="400px" height="400px"
                  style="border : solid"></canvas>
</body>
```

filename: rotateExample.js

```
//Example of rotation
//Author: Kim Lam

//Get the render context
var canvas = document.getElementById("drawSurface");
var ctx = canvas.getContext("2d");

ctx.save();
//Move the origin to the center of the canvas.
ctx.translate(canvas.width/2, canvas.height/2);

//Draw the first set of squares
ctx.save();
ctx.fillStyle = "black";
ctx.fillRect(0,0,10,10);
ctx.fillRect(20,0,10,10);
ctx.fillRect(40,0,10,10);
ctx.restore(); //Undo the rotation

ctx.save();
//Rotate the coordiante so it's on a
//45 degree angle
ctx.rotate(45*Math.PI/180)
ctx.fillStyle = "red";
ctx.fillRect(0,0,10,10);
ctx.fillRect(20,0,10,10);
ctx.fillRect(40,0,10,10);
ctx.restore(); //Undo the rotation

ctx.save();
//Rotate the coordiante so it's on a
// 110 degree angle
ctx.rotate(110*Math.PI/180)
ctx.fillStyle = "blue";
ctx.fillRect(0,0,10,10);
```

```
ctx.fillRect(20,0,10,10);
ctx.fillRect(40,0,10,10);
ctx.restore(); //Undo the rotation

ctx.restore(); //Undo the initial translate
```