

The HTML5 Canvas Using JavaScript

Computer Science and Information Technology
School of Mathematical & Computational Sciences
Massey University (AKLI, DISD & MTUI)

Learning Objectives

At the end of this topic, you should be able to:

- Use the canvas element in HTML5 to create some graphics on a Web page;
- Define the graphic content of the canvas by calling the various drawing methods using JavaScript;
- Use the markup within the canvas element for appropriate fallback content when the browser does not support canvas; and
- Differentiate between the canvas and SVG.

The HTML5 canvas element

- The main thing youâ€™ll notice is that canvas is completely implemented in JavaScript;
- The only part that is in the markup is a simple element definition like this:

```
<canvas id="myCanvas" width="400" height="400">  
    Canvas is not supported on this browser  
</canvas>
```

- Youâ€™ll define the content by calling the various drawing methods using JavaScript;
- The text of markup within the canvas element is used when the browser does not support canvas. You can use this to provide the appropriate fallback content.

The Nature of HTML5 Canvas

- The `<canvas>` element provides an area that you can use to draw on.
- When you create a `<canvas>` element, you define its **size** using the *height* and *width* attributes.
- Other attributes can be specified using markup or CSS to specify the margin, padding, and border — affecting where the element is positioned within the page.
- `<canvas>` element — simply defines a blank area on which you can create your masterpiece so, you cannot modify any of the content within it.
- You should always assign an **id** attribute to `<canvas>` element so you can access it in JavaScript using the **getElementById()** method.

- You may also access <canvas> element using the **getElementsByTagName()** method or use the new query selectors you learned in this course.

Working with the HTML5 Canvas

(1) Defining the HTML5 **canvas** element:

```
<canvas id="canvas1" width="200" height="200"></canvas>
```

(2) **Referencing** the canvas element in JavaScript (like any other HTML element):

```
var canvas = document.getElementById("canvas1");
```

(3) The **canvas context** — where drawing and canvas and state of canvas is located:

```
var context = canvas.getContext("2d");
```

(4) Separation of *canvas* from *context* — necessary because canvas is a DOM

element while context describes kind of drawing to be done on canvas.

(5) Two types of contexts: **2D** (default) and **3D** (accessed using "webgl")

Canvas Fallback HTML

An arbitrary block of HTML inside the `<canvas>` element:

- Displayed when browser doesn't support HTML5 `<canvas>` element.
- Also accessed by assistive technologies.

```
<!-- Canvas Fallback HTML -->

<canvas width="300" height="300">
  <p>
    You can't see these paragraphs if your browser supports HTML5 canvas.
  </p>
  <p>
    If you can see them, you should download a newer browser!
  </p>
  <p>
    This is where you might place a fallback embedded Flash object,
    or just an image, or provide a link to Chrome Frame.
  </p>
</canvas>
```


HTML Page with a Canvas

```
<!-- canvas-example-1.html - A Complete HTML Page with a Canvas -->
<!doctype html>
<html>
  <body>
    <canvas id="myCanvas1" width="500" height="500"
      style="border: 1px solid black;">
      This text is displayed if your browser
      does not support HTML5 canvas
    </canvas>

    <script type="text/javascript">
      // We'll attach both of these to the
      // JavaScript window object
      // so we can easily access them in the console
      window.canvas = document.getElementById('myCanvas1');
      window.ctx = canvas.getContext('2d');
      ctx.fillRect(50, 60, 70, 80);
    </script>
  </body>
</html>
```



Drawing on the canvas involves calling a set of commands that apply pixels or

drawing operation on the canvas. The following JavaScript command draws a rectangle starting at coordinates (50, 60) with width 70 and height 80.

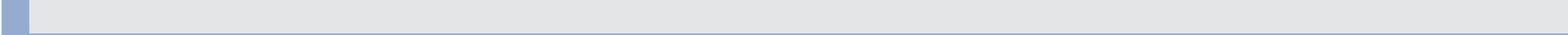
```
ctx.fillRect(50, 60, 70, 80);
```

fillStyle and strokeStyle

In previous slide, the drawn rectangle is black, thus the default **fillStyle** is "black"

To draw a "red" rectangle, you have to change the state of the context by changing the value of **fillStyle** as shown below:

```
ctx.fillStyle="red";  
// identical to "red" but represented as hex value  
ctx.fillStyle="#FF0000";  
// also identical to red but represent as an RGB value, 0 to 255  
ctx.fillStyle="rgb(255,0,0)";  
// also identical to red but represented as HSL value  
ctx.fillStyle="hsl(0,100%,50%)";  
// also red but semi-transparent  
ctx.fillStyle="rgba(255,0,0,0.5)";
```



The context attribute **strokeStyle** is set in the exact same way with the same possible values.

Paths

Context methods *fillRect()* and *strokeRect()* are high-level methods that perform common **path** operations for creating a rectangle and stroking (as in paint brush stroke) or filling it.

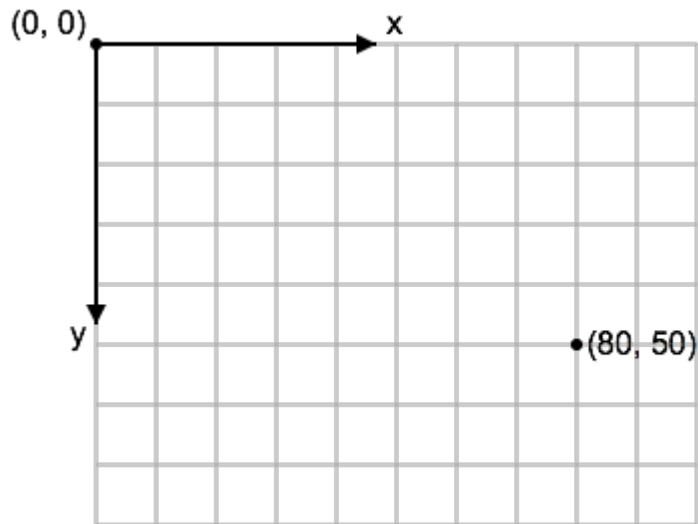
Other shapes are drawn by constructing **paths** made up of *lines* and *curves* drawn on the canvas by the *stroke* or *fill* method.

Example: Drawing the plus (+) sign — "*draw two lines crossing*"

```
ctx.beginPath();  
// The following specify the path your  
// paint-brush would follow to draw the plus sign  
ctx.moveTo(50,0);  
ctx.lineTo(50,100);  
ctx.moveTo(0,50);  
ctx.lineTo(100,50);  
ctx.stroke(); // Draw the path just constructed above
```

The Canvas Coordinate System

The **canvas coordinate system** is a rectangular grid with top left corner being the origin (0,0) and x-axis is from origin to the right and y-axis going down from the origin.

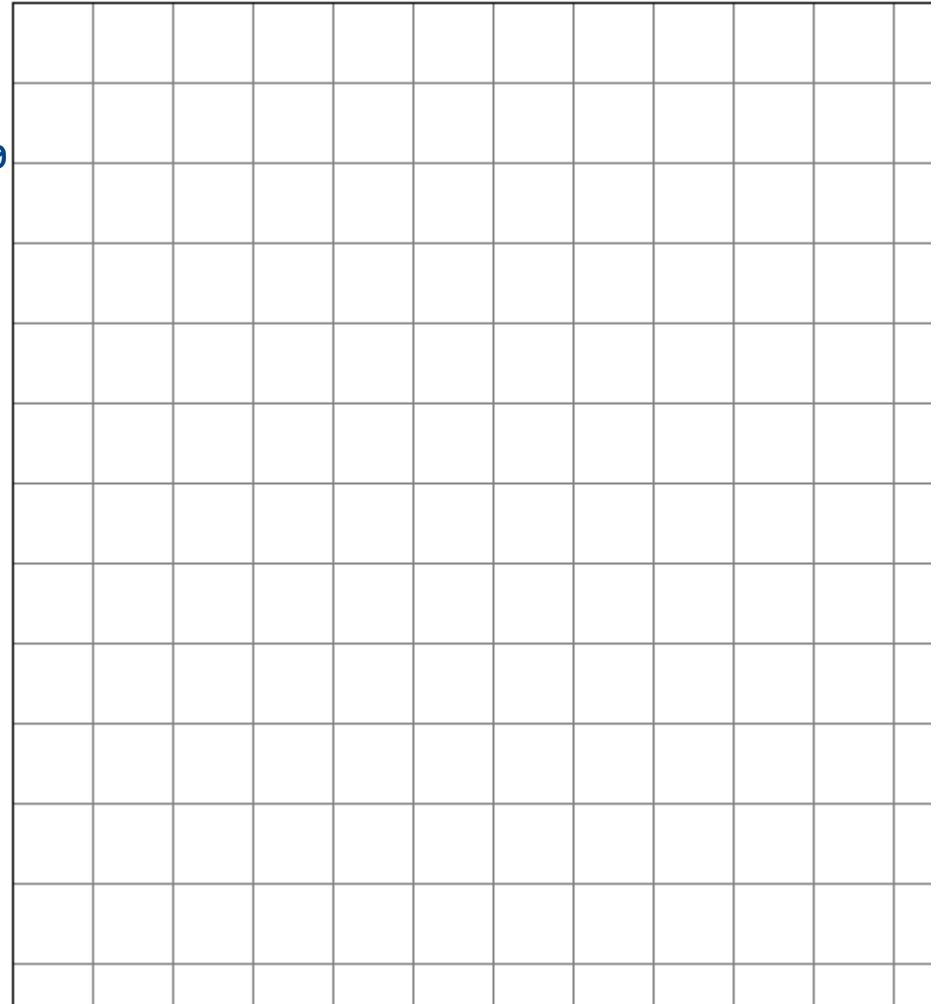


Drawing Simple grid lines

```
<!doctype html>
<html>
  <body>
    <canvas id="myCanvas" width="500" height="500"
      style="border: 1px solid black;">
      This text is displayed if your
      browser does not support HTML5 canvas
    </canvas>

    <script type="text/javascript">
      // We'll attach both of these to
      // the JavaScript window object
      // so we can easily access
      // them in the console
      window.canvas = document.getElementById('myCanvas');
      window.ctx = canvas.getContext('2d');

      // Drawing Simple Gridlines
      var width = canvas.width;
      var height = canvas.height;
      for (var i = 0; i < width; i += 40) {
        ctx.moveTo(i - 0.5, 0);
        ctx.lineTo(i - 0.5, height);
      }
      for (var i = 0; i < height; i += 40) {
        ctx.moveTo(0, i - 0.5);
        ctx.lineTo(width, i - 0.5);
      }
      ctx.strokeStyle = 'gray';
    </script>
  </body>
</html>
```



```
ctx.stroke();  
</script>  
</body>  
</html>
```

Line Styles

The canvas context provides four properties to style stroked points:

- **lineWidth** — number attribute to customise width of stroked paths;
- **lineCap** — string attribute to control the looks of any open subpaths. Valid values are: *butt*, *round*, and *square*;
- **lineJoin** — string attribute to control how corners look where 2 lines meet;
- **miterLimit** — positive number that controls the mitering ratio with default value 10.0.

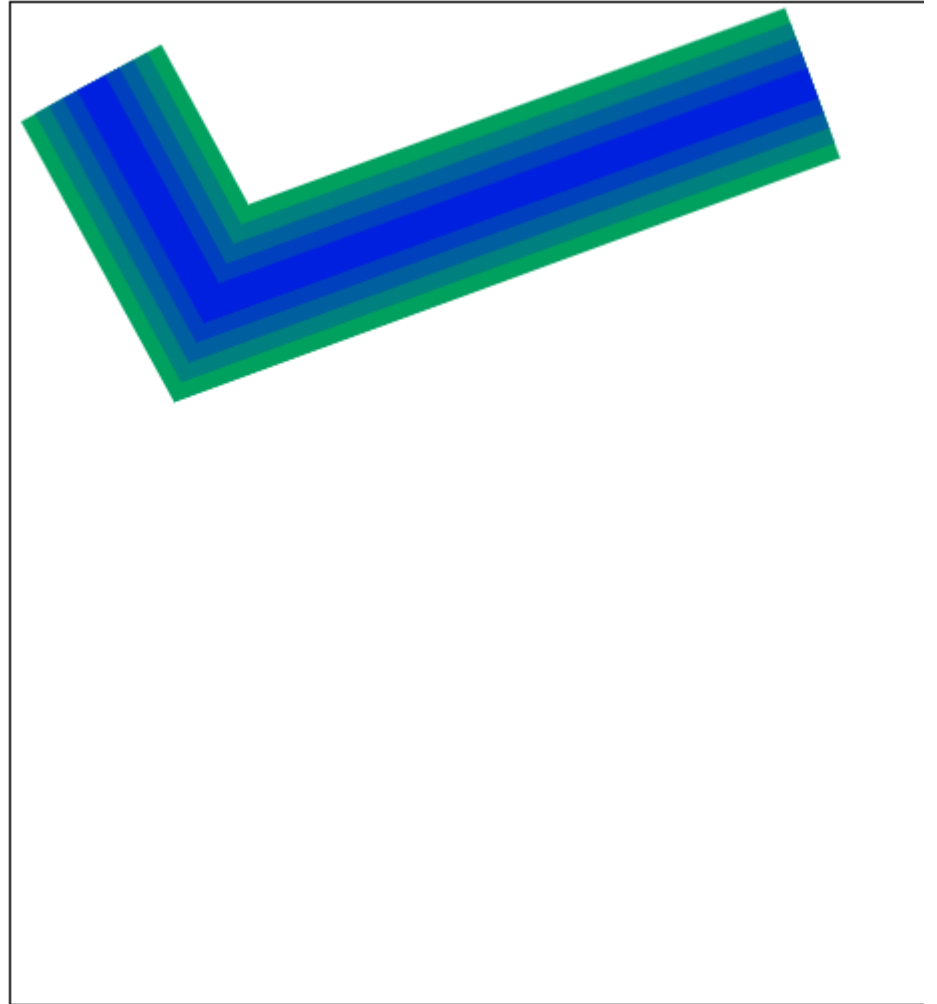
Line Styles: Stroking Repeatedly

```
// Stroking the Same Path Multiple Times
// #####
// If you're following along in the developer
// console and ran prior code,
// Remember to clear the canvas with
// canvas.width = canvas.width
// Or a page reload

// Construct a path
ctx.moveTo(40,40);
ctx.lineTo(100, 150);
ctx.lineTo(400, 40);

// stroke several times with a smaller line width
// and different color each time
for (var i = 5; i > 0; i--) {
  ctx.lineWidth = i*16;
  var blueValue = 255 - (i*32);
  var greenValue = i*32;
  ctx.strokeStyle = 'rgb(0, ' + greenValue + ', '
    + blueValue + ')';
  ctx.stroke();
}
```

Here is code that uses repeated stroking to produce the canvas drawing seen here.



The JavaScript code has been taken out of the HTML context on this slide.

Example I: A Simple Chessboard on a Web Page

The web page for the chess board containing the HTML5 <canvas> element

```
<!DOCTYPE html>
<!-- filename: 09-javascript-canvas-chess1.html -->
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>The HTML5 Canvas and JavaScript - Chess</title>
  </head>
  <body>
    <!-- setup the canvas -->
    <canvas id="board" width ="600" height ="600">
      <!-- accessed when browser does not support the canvas element -->
        Not supported
    </canvas>
    <!-- Javascript to draw the chessboard -->
    <script src="js/chess1.js" defer></script>
  </body>
</html>
```

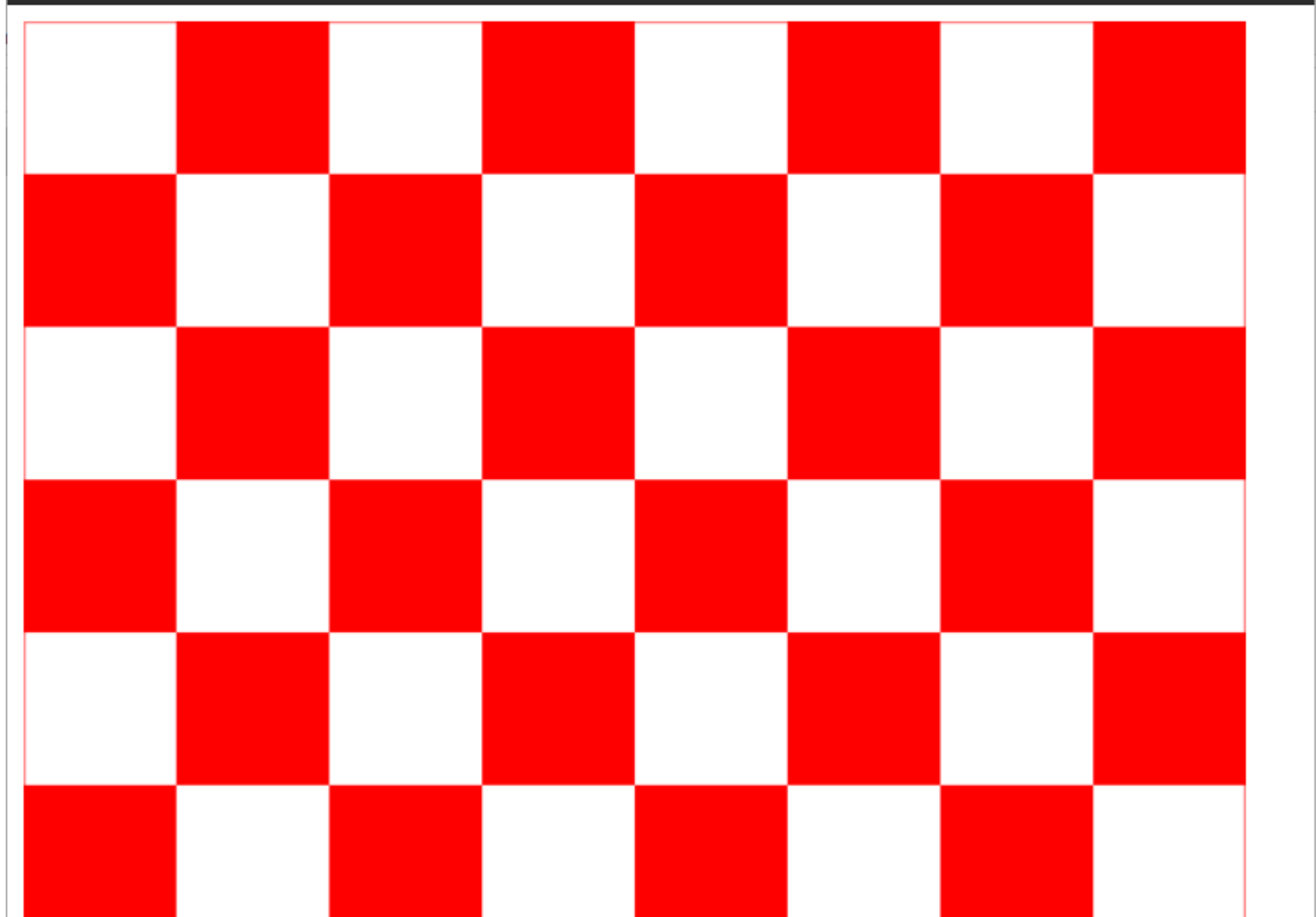
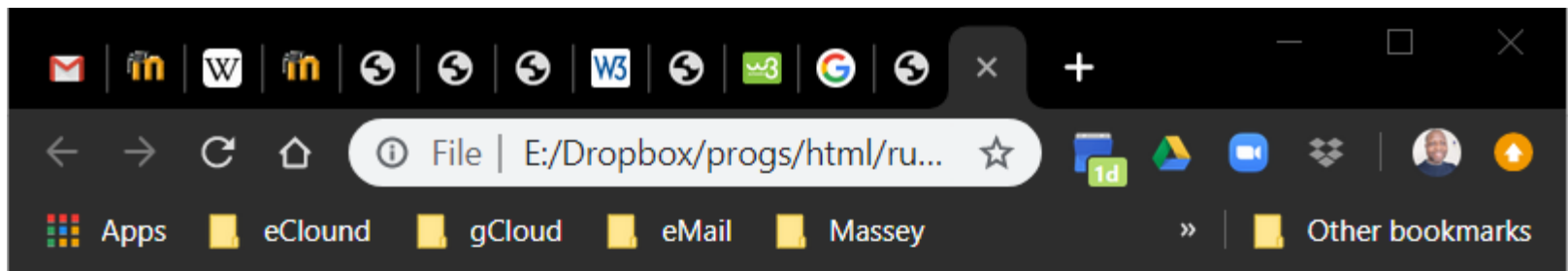
Example I: Drawing the Chessboard

```
// filename: chess1.js
"use strict";

// Get the canvas context
let chessCanvas = document.getElementById("board");
let chessContext = chessCanvas.getContext("2d");

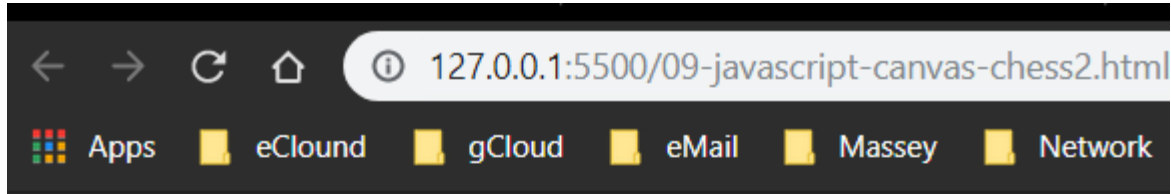
// Draw the chess board
function drawBoard() {
    chessContext.clearRect(0, 0, 600, 600);
    chessContext.fillStyle = "red";
    chessContext.strokeStyle = "red";

    // Draw the alternating squares
    for (let x = 0; x < 8; x++) {
        for (let y = 0; y < 8; y++) {
            if ((x + y) % 2) {
                chessContext.fillRect(75 * x,
                                        75 * y,
                                        75, 75);
            }
        }
    }
    // Add a border around the entire board
    chessContext.strokeRect(0, 0, 600, 600);
}
drawBoard();
```



Example 2: Chessboard with Chess Pieces

```
// Define a class to store the piece properties
function ChessPiece() {
    this.image = null;
    this.x = 0;
    this.y = 0;
    this.height = 0;
    this.width = 0;
    this.killed = false;
}
// Draw a chess piece
function drawPiece(p) {
    if (!p.killed)
        chessContext.drawImage(p.image,
                               (75 - p.width) / 2 + (75 * p.x),
                               75 - p.height + (75 * p.y),
                               p.width,
                               p.height);
}
// Draw all of the chess pieces
function drawAllPieces() {
    for (let i = 0; i < 32; i++) {
        if (pieces[i] != null) {
            drawPiece(pieces[i]);
        }
    }
}
```



Curves in the Canvas Context

The canvas context provides several methods for augmenting paths with curves, which are as follows:

```
(1) arc(x, y, radius, startAngle, endAngle, counterClockwise);  
(2) arcTo(x1, y1, x2, y2, radius);  
(3) bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y);  
(4) quadraticCurveTo(cpx, cpy, x, y);
```

These are used within the context of path construction and drawing shown in previous slides.

Ellipses in the Canvas Context

The `CanvasRenderingContext2D.ellipse()` method of the Canvas 2D API adds an elliptical arc to the current sub-path.

```
ctx.ellipse(x, y, radiusX, radiusY, rotation, startAngle, endAngle [, anticlockwise]);
```

The **ellipse()** method creates an elliptical arc centered at **(x, y)** with the radii **radiusX** and **radiusY**. The path starts at **startAngle** and ends at **endAngle**, and travels in the direction given by **anticlockwise** (defaulting to **clockwise**).

Image Drawing on the Canvas

```
drawImage(image, dx, dy[, dw, dh] );
```

The above code provides the syntax for rendering an image to the canvas context at location **(dx,dy)**. The two optional arguments represent a destination width and height for optional scaling. The image can be an image element (), video element(<video>), or canvas element (<canvas>) not attached to the DOM tree (may be constructed in JavaScript).

```
drawImage(image,  sx, sy, sw, sh, dx, dy, dw, dh);
```

The above code provides the syntax for rendering a portion of an image described by the source (s-) arguments to a rectangle of the context described by the destination (d-) arguments.

Transformations on the Canvas

- **translate(x, y)** — Translates the origin of the context;
- **scale(x, y)** — Scales the context;
- **rotate(angle)** — Rotates the context by angle in radians;
- **transform(m11, m12, m21, m22, dx, dy)** — Applies an arbitrary transformation to the current transformation matrix; and
- **transform(m11, m12, m21, m22, dx, dy)** — Replaces the current transformation matrix with the specified matrix.

Topic Summary

- (1) Canvas allow dynamic content in similar way as Flash and Silverlight.
- (2) It is expected to replace browser plugins for graphics rendering on the Web.
- (3) This topic covered the basic methods and attributes of HTML5 Canvas and its 2D context — its capabilities.
- (4) The topic has created a basic foundation for you to learn a lot more about the HTML5 canvas that we did not cover within the limited time of one topic in a course with many topics.