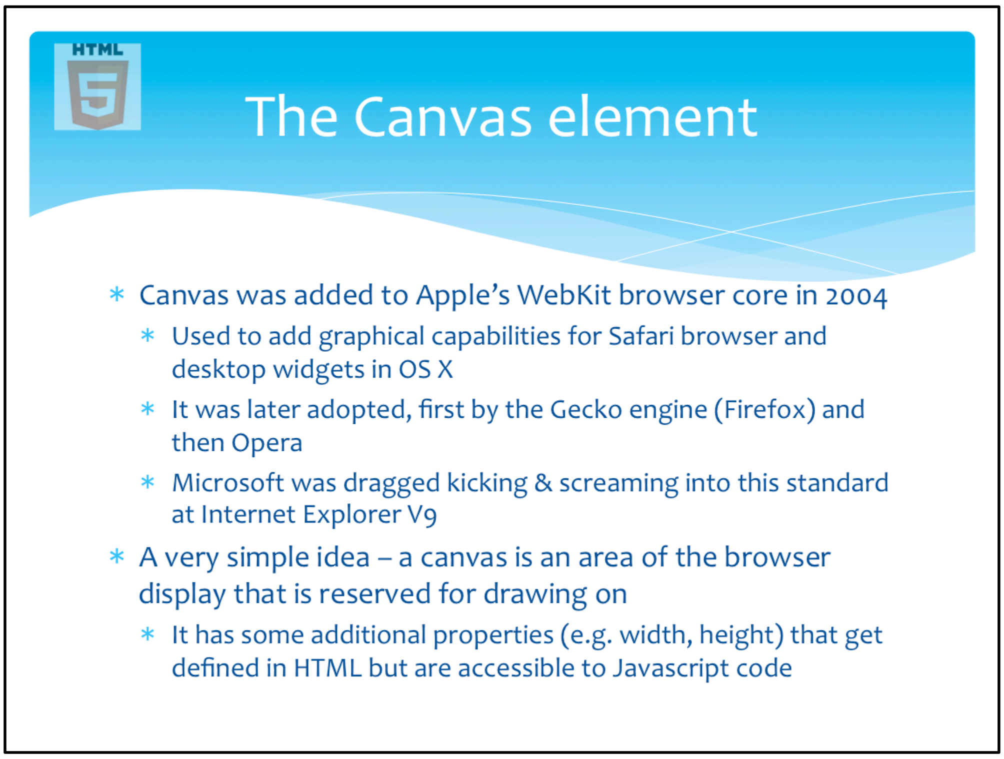
HTML5 & Javascript

HTML5 Graphics

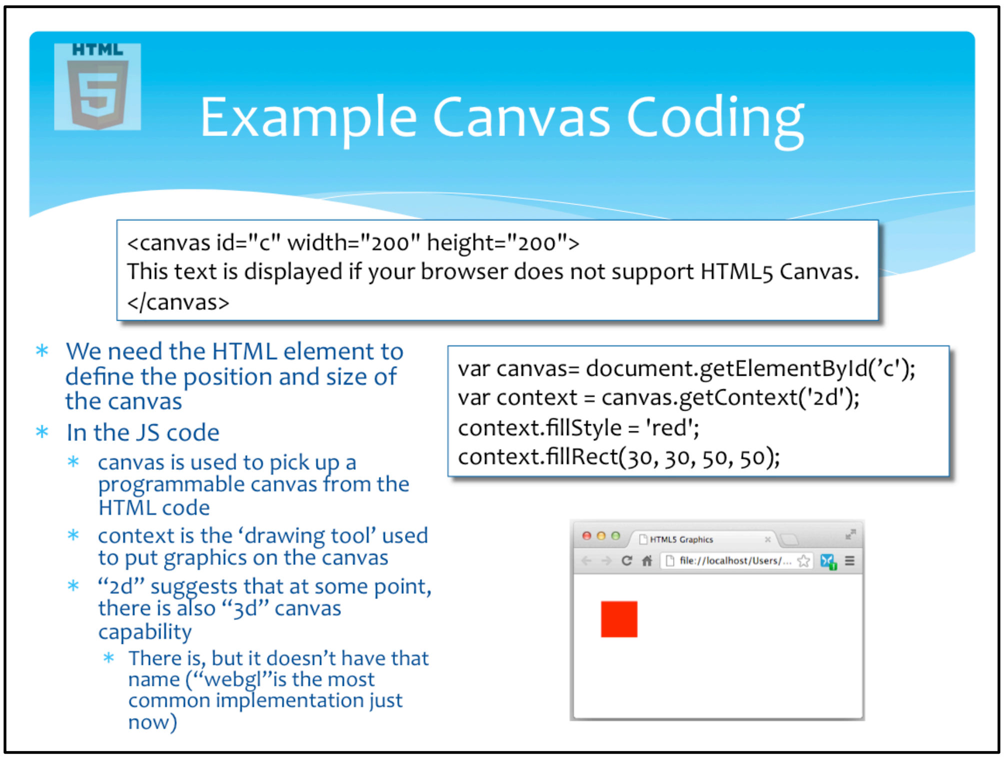
The fun stuff…



The Canvas element

- Canvas was added to Apple's WebKit browser core in 2004
  - Used to add graphical capabilities for Safari browser and desktop widgets in OS X
  - It was later adopted, first by the Gecko engine (Firefox) and then Opera
  - Microsoft was dragged kicking & screaming into this standard at Internet Explorer V9
- A very simple idea – a canvas is an area of the browser display that is reserved for drawing on
  - It has some additional properties (e.g. width, height) that get defined in HTML but are accessible to Javascript code

Canvas itself is not very useful in JS code. It has the width and height properties, which you can get from the graphics context, via context.canvas.width, context.canvas.height

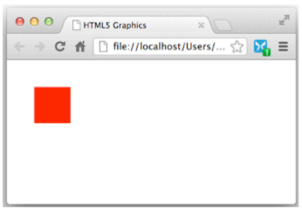


Example Canvas Coding

```
<canvas id="c" width="200" height="200">
This text is displayed if your browser does not support HTML5 Canvas.
</canvas>
```

* We need the HTML element to define the position and size of the canvas
* In the JS code
  * canvas is used to pick up a programmable canvas from the HTML code
  * context is the ‘drawing tool’ used to put graphics on the canvas
  * “2d” suggests that at some point, there is also “3d” canvas capability
    * There is, but it doesn’t have that name (“webgl”is the most common implementation just now)

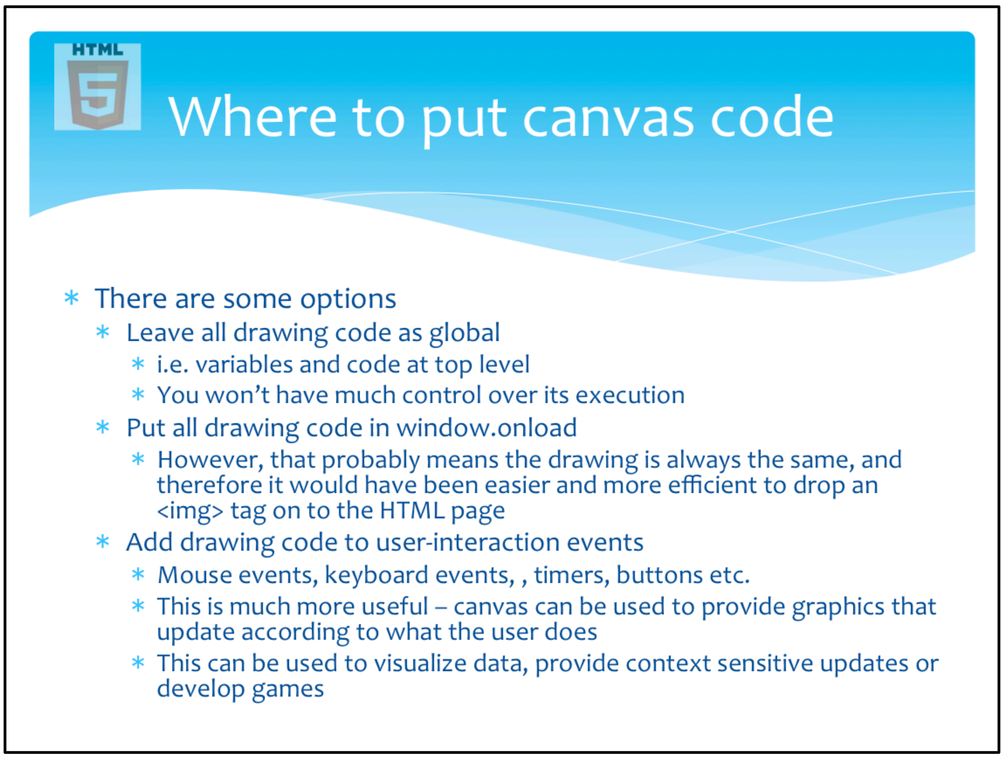
```
var canvas= document.getElementById('c');
var context = canvas.getContext('2d');
context.fillStyle = 'red';
context.fillRect(30, 30, 50, 50);
```



context.fillRect() is just about the only complete drawing function.
Other drawing functions have to be finished with .stroke() or .fill().

There are some experimental “3d” context implementations. See http://www.kevs3d.co.uk/dev/canvask3d/k3d_test.html, but these are fudges, based on the standard 2D canvas with a lot of drawing code.

See
http://www.webglacademy.com/#0 and
https://developer.mozilla.org/en-US/docs/Web/WebGL/
Getting_started_with_WebGL
for good tutorials on WebGL.



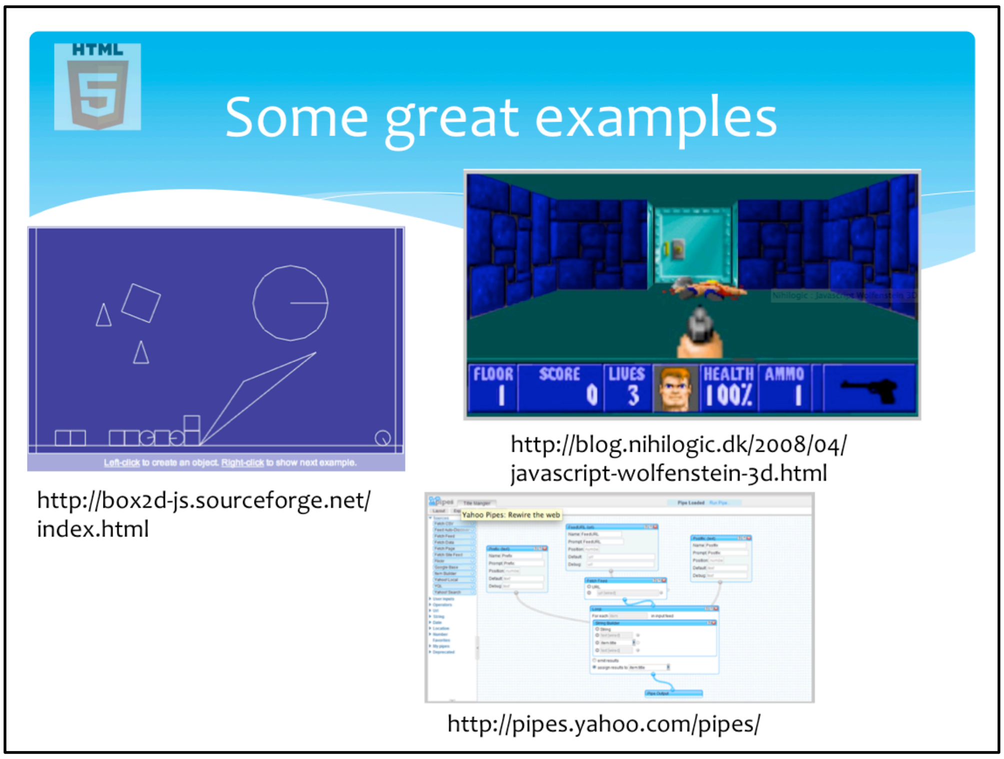
Where to put canvas code

* There are some options
  * Leave all drawing code as global
    * i.e. variables and code at top level
    * You won't have much control over its execution
  * Put all drawing code in window.onload
    * However, that probably means the drawing is always the same, and therefore it would have been easier and more efficient to drop an <img> tag on to the HTML page
  * Add drawing code to user-interaction events
    * Mouse events, keyboard events, , timers, buttons etc.
    * This is much more useful – canvas can be used to provide graphics that update according to what the user does
    * This can be used to visualize data, provide context sensitive updates or develop games

The point to make here is that putting the same drawing code either at the top level of a JS file, or in window.onload will result in a static image. This can save on downloading time compared to embedding an <img> tag, but this is no big issue and most web developers would simply add the <img>

To make some sort of point about live rendering, an app has to draw an image on a canvas based on some variable – e.g. the canvas width, or some other data that may change depending on viewing circumstances.

The best uses of canvas will involve dynamically drawn images – reactions to user-interactions, downloaded data (e.g. a static page with a canvas which accesses a .txt data file (possibly generated) and draws a graph or chart based on the data in it.


HTML
Left-click to create an object. Right-click to show next example.
FLOOR 1
SCORE 0
LIVES 3
HEALTH 100%
AMMO 1
Yahoo Pipes: Rewire the web

Some great examples

http://box2d-js.sourceforge.net/index.html

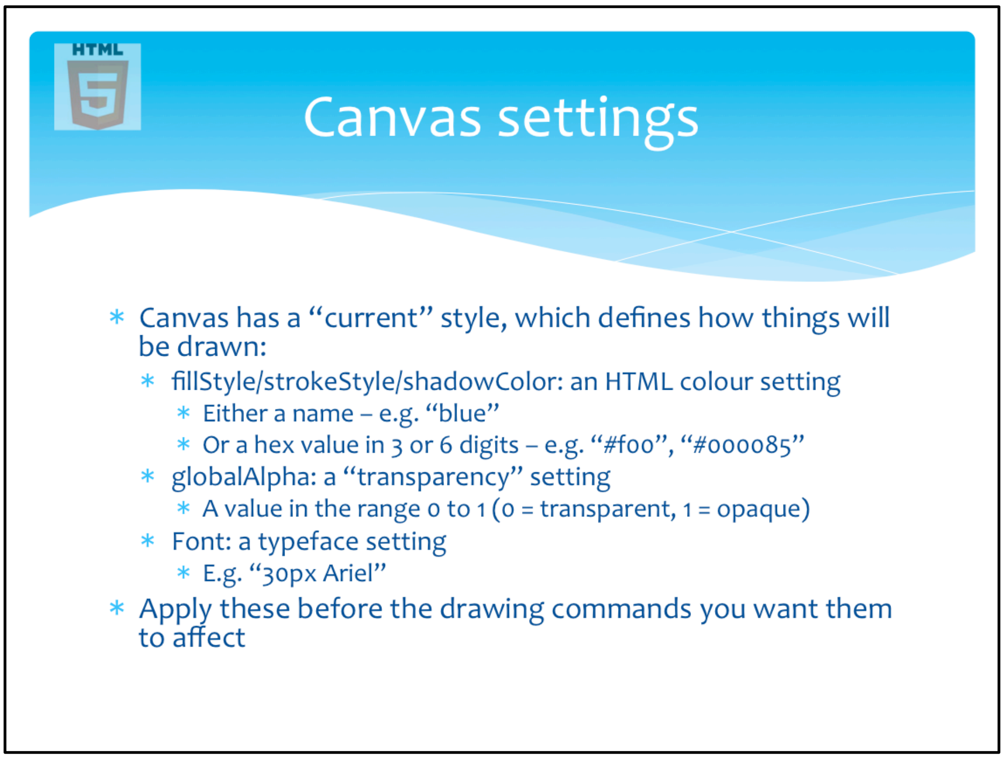
http://blog.nihilogic.dk/2008/04/javascript-wolfenstein-3d.html

http://pipes.yahoo.com/pipes/

Try some of these demos in class:

Cloth experiment: http://www.andrew-hoyer.com/andrewhoyer/experiments/cloth/
Bomomo: http://bomomo.com/
Liquid particles: http://spielzeugz.de/html5/liquid-particles.html
Sinuous (game): http://hakim.se/experiments/html5/sinuous/01/

These are all pretty good to show the class during the lecture. If you don't have a live web connection in the class but can view a laptop screen, try downloading the apps and starting them before the class, and sleeping or hibernating the machine. When you resume, the apps still ought to be operational.



Canvas settings

- Canvas has a "current" style, which defines how things will be drawn:
  - fillStyle/strokeStyle/shadowColor: an HTML colour setting
    - Either a name – e.g. "blue"
    - Or a hex value in 3 or 6 digits – e.g. "#f00", "#000085"
  - globalAlpha: a "transparency" setting
    - A value in the range 0 to 1 (0 = transparent, 1 = opaque)
  - Font: a typeface setting
    - E.g. "30px Ariel"
- Apply these before the drawing commands you want them to affect

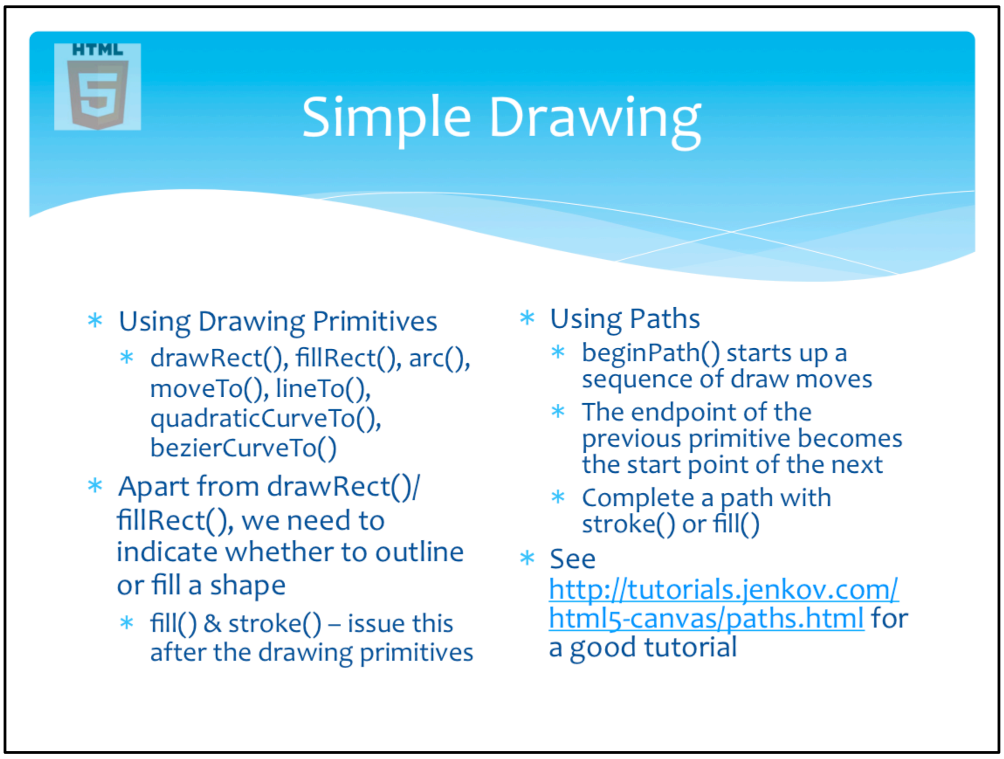
These are actually settings that are applied to the context (technically a HTMLGraphicsContext object).
The usual sequence is…

```
apply setting
draw figures (using moveTo, lineTo, arc etc., possibly with a path)
call stroke() (outline) or fill()
```

e.g.

```
function draw(context) {
   context.fillStyle = "red";
   context.strokeStyle = "black";
   context.arc( 50, 50, 20, 0, Math.PI * 2 );
   context.stroke();
   context.fill();
}
```

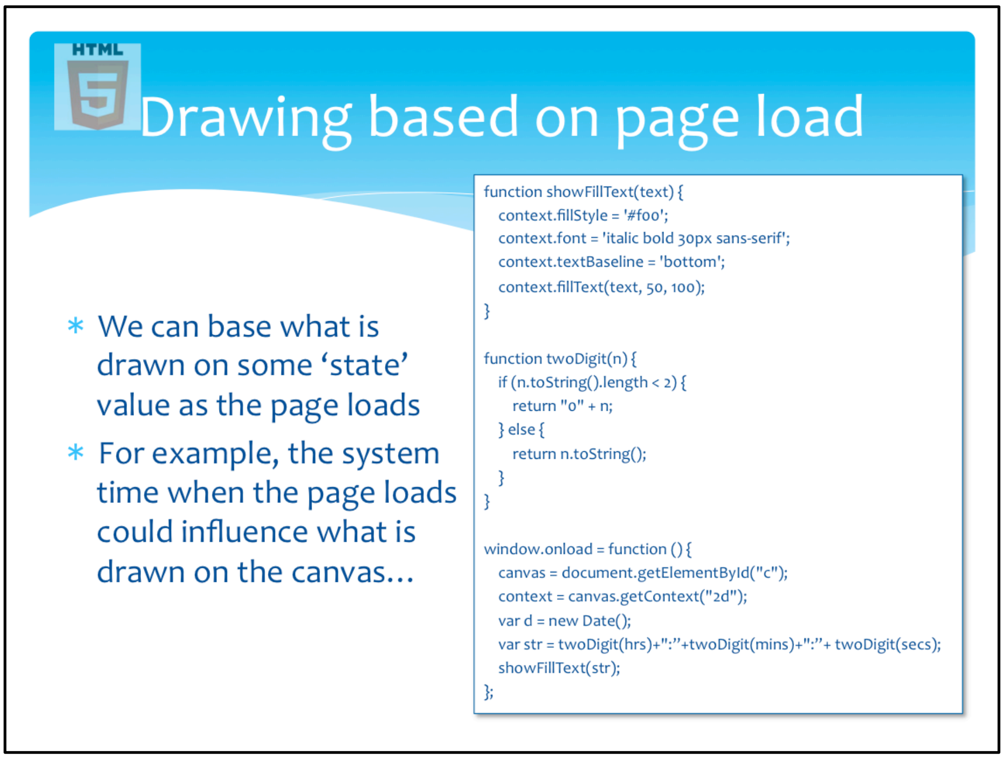
This does a filled/outlined circle.



Simple Drawing

- Using Drawing Primitives
  - drawRect(), fillRect(), arc(), moveTo(), lineTo(), quadraticCurveTo(), bezierCurveTo()
- Apart from drawRect()/ fillRect(), we need to indicate whether to outline or fill a shape
  - fill() & stroke() – issue this after the drawing primitives
- Using Paths
  - beginPath() starts up a sequence of draw moves
  - The endpoint of the previous primitive becomes the start point of the next
  - Complete a path with stroke() or fill()
- See http://tutorials.jenkov.com/html5-canvas/paths.html for a good tutorial

Between calls to beginPath(), all drawing results in a single path, with a single outline and fill colour. .fill() or .stroke() ends a path.

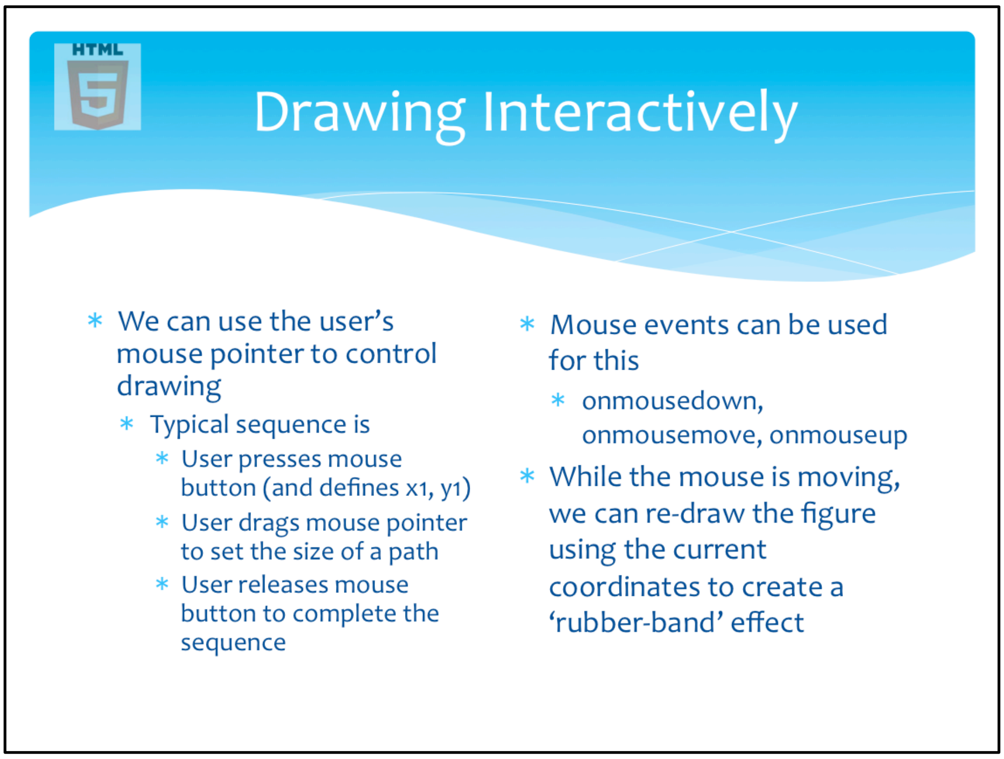


Drawing based on page load

- We can base what is drawn on some 'state' value as the page loads
- For example, the system time when the page loads could influence what is drawn on the canvas…

```
function showFillText(text) {
   context.fillStyle = '#f00';
   context.font = 'italic bold 30px sans-serif';
   context.textBaseline = 'bottom';
   context.fillText(text, 50, 100);
}

function twoDigit(n) {
   if (n.toString().length < 2) {
      return "0" + n;
   } else {
      return n.toString();
   }
}

window.onload = function () {
   canvas = document.getElementById("c");
   context = canvas.getContext("2d");
   var d = new Date();
   var str = twoDigit(hrs)+":"+twoDigit(mins)+":"+ twoDigit(secs);
   showFillText(str);
};
```

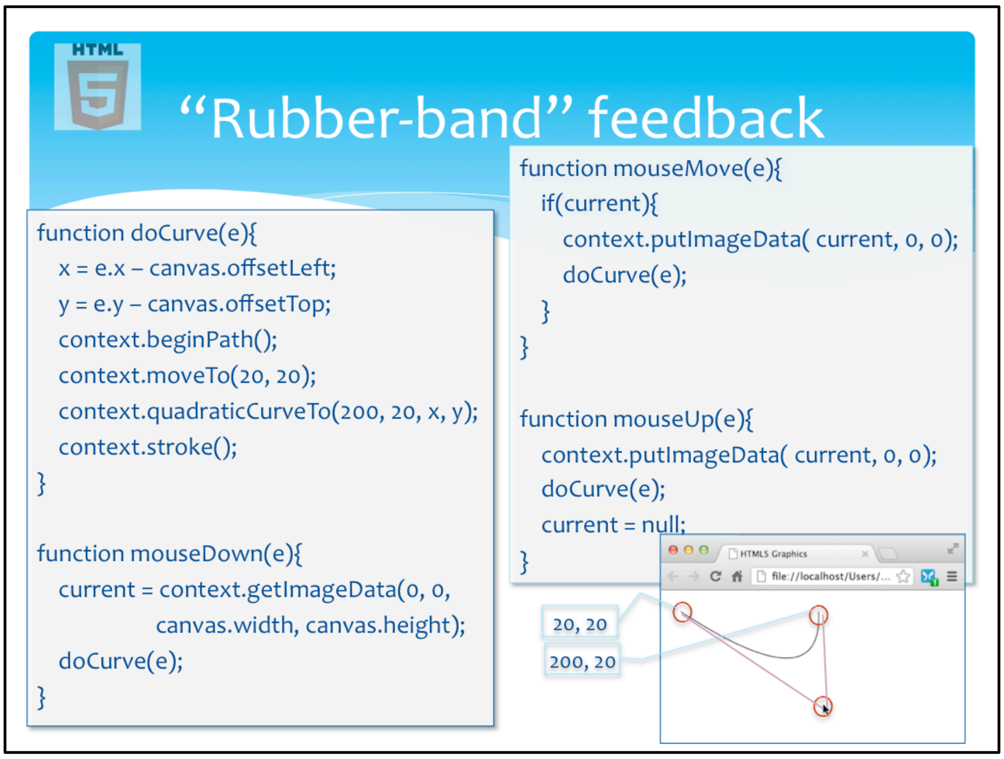
The point of this code is that the current time (i.e. the time the page was loaded into the browser) will be drawn on a canvas – hardly rocket science, but something that requires live drawing.



Drawing Interactively

- We can use the user's mouse pointer to control drawing
  - Typical sequence is
    - User presses mouse button (and defines x1, y1)
    - User drags mouse pointer to set the size of a path
    - User releases mouse button to complete the sequence
- Mouse events can be used for this
  - onmousedown, onmousemove, onmouseup
- While the mouse is moving, we can re-draw the figure using the current coordinates to create a 'rubber-band' effect

This is necessarily more complex. A very easy variant is to use onmousemove to draw continuously (e..g
small rectangles, circles or line segments). However to give the user control, we need state variables to
indicate when the mouse button is pressed. In the next slide, I use the available of background image data
to indicate that we are currently drawing. Could also use a boolean (mousedown = true) and then only draw
on mouseMove() when the boolean is true.

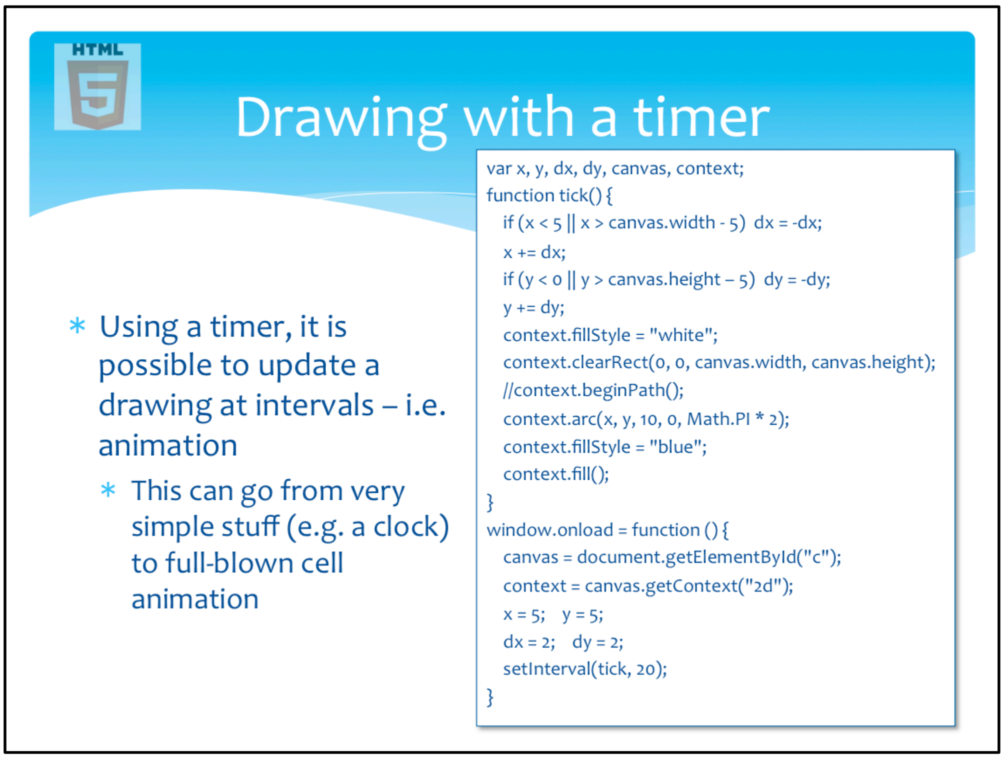


"Rubber-band" feedback

```
function doCurve(e){
  x = e.x – canvas.offsetLeft;
  y = e.y – canvas.offsetTop;
  context.beginPath();
  context.moveTo(20, 20);
  context.quadraticCurveTo(200, 20, x, y);
  context.stroke();
}

function mouseDown(e){
  current = context.getImageData(0, 0,
                canvas.width, canvas.height);
  doCurve(e);
}
```

```
function mouseMove(e){
  if(current){
    context.putImageData( current, 0, 0);
    doCurve(e);
  }
}

function mouseUp(e){
  context.putImageData( current, 0, 0);
  doCurve(e);
  current = null;
}
```

The little diagram shows how the control vertex (at the mouse pointer) controls the quadratic curve. For a bezier curve, we need two control points. The only actual drawn element in the diagram is the black curve – the rest simply show how it is constructed.

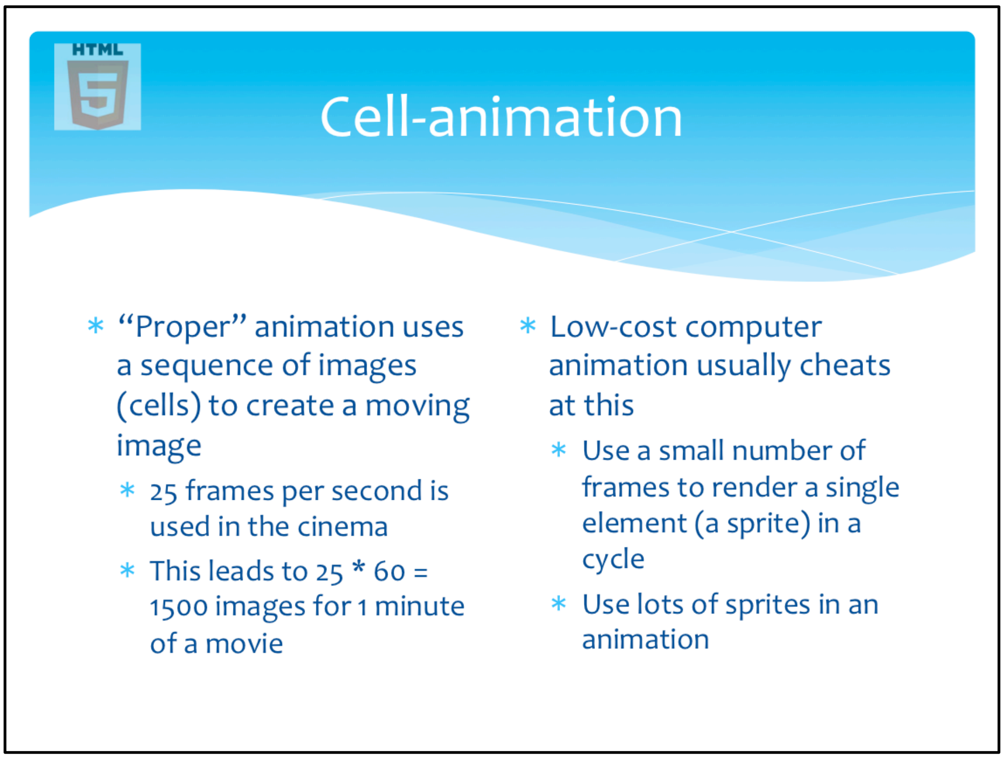


Drawing with a timer

* Using a timer, it is possible to update a drawing at intervals – i.e. animation
  * This can go from very simple stuff (e.g. a clock) to full-blown cell animation

```
var x, y, dx, dy, canvas, context;
function tick() {
   if (x < 5 || x > canvas.width - 5)  dx = -dx;
   x += dx;
   if (y < 0 || y > canvas.height – 5)  dy = -dy;
   y += dy;
   context.fillStyle = "white";
   context.clearRect(0, 0, canvas.width, canvas.height);
   //context.beginPath();
   context.arc(x, y, 10, 0, Math.PI * 2);
   context.fillStyle = "blue";
   context.fill();
}
window.onload = function () {
   canvas = document.getElementById("c");
   context = canvas.getContext("2d");
   x = 5;    y = 5;
   dx = 2;    dy = 2;
   setInterval(tick, 20);
}
```

This does a "bouncing ball" animation. An interesting variation would come by un-commenting the
   beginPath() call

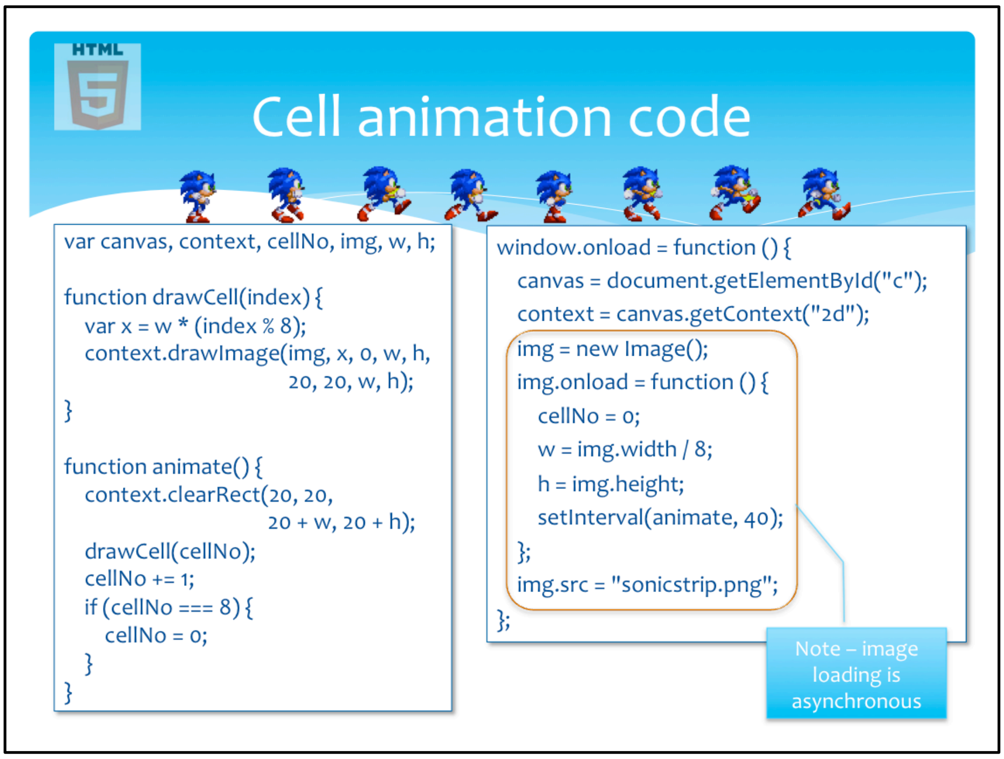


Cell-animation

* "Proper" animation uses a sequence of images (cells) to create a moving image
  * 25 frames per second is used in the cinema
  * This leads to 25 * 60 = 1500 images for 1 minute of a movie
* Low-cost computer animation usually cheats at this
  * Use a small number of frames to render a single element (a sprite) in a cycle
  * Use lots of sprites in an animation

Cell-animation in this context is based on a sequence of images, a bit like flipping the corner of a book to make an animation. We can simplify it in HTML5 by using a composite image containing all of the cells as equally-sized rectangular segments.

Note that full-image cell animation is bound to be expensive, but game-style cell animation
involves a lot of repetition in cells, and lots of simple cells placed on top of a static or near-static
background.



Cell animation code

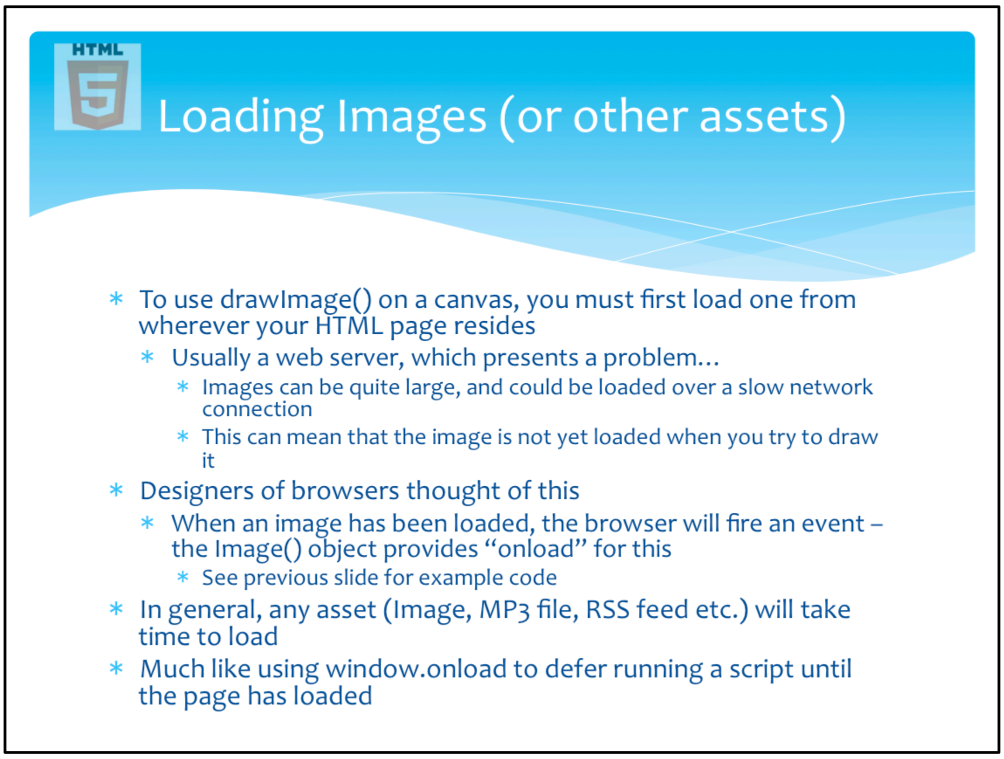
```
var canvas, context, cellNo, img, w, h;

function drawCell(index) {
   var x = w * (index % 8);
   context.drawImage(img, x, 0, w, h,
                              20, 20, w, h);
}

function animate() {
   context.clearRect(20, 20,
                           20 + w, 20 + h);
   drawCell(cellNo);
   cellNo += 1;
   if (cellNo === 8) {
      cellNo = 0;
   }
}
```

```
window.onload = function () {
   canvas = document.getElementById("c");
   context = canvas.getContext("2d");
   img = new Image();
   img.onload = function () {
      cellNo = 0;
      w = img.width / 8;
      h = img.height;
      setInterval(animate, 40);
   };
   img.src = "sonicstrip.png";
};
```

Note – image loading is asynchronous

The problem with loading an image at run-time is that the width and height are only known once the image is fully loaded – hence the use of img.onload on the slide.

Assume the worst – a slow network connection and large images – this is the only safe way to load an image and view it on the browser.



Loading Images (or other assets)

- To use drawImage() on a canvas, you must first load one from wherever your HTML page resides
  - Usually a web server, which presents a problem…
    - Images can be quite large, and could be loaded over a slow network connection
    - This can mean that the image is not yet loaded when you try to draw it
- Designers of browsers thought of this
  - When an image has been loaded, the browser will fire an event – the Image() object provides "onload" for this
    - See previous slide for example code
- In general, any asset (Image, MP3 file, RSS feed etc.) will take time to load
- Much like using window.onload to defer running a script until the page has loaded

See code on previous slide.

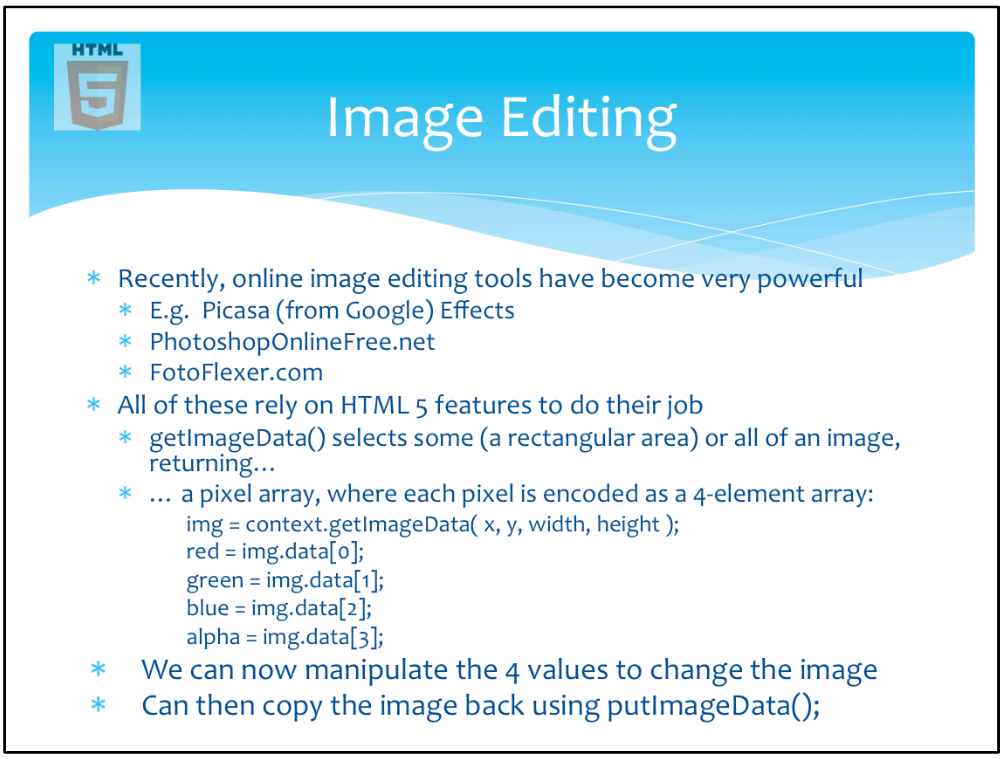


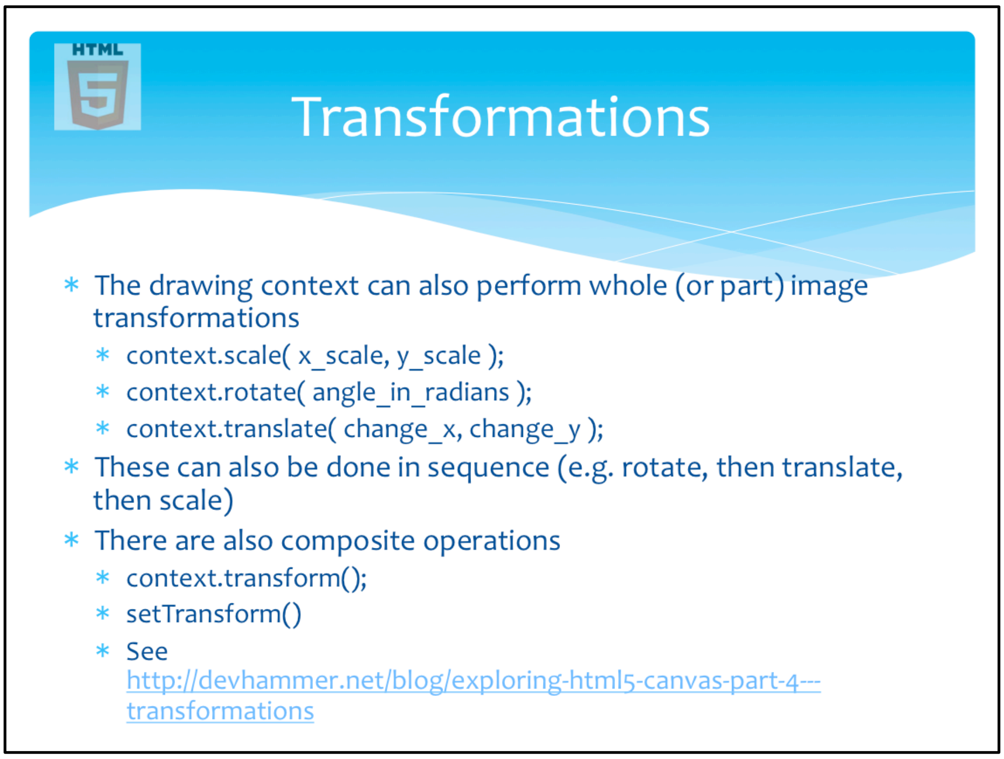
Image Editing

* Recently, online image editing tools have become very powerful
  * E.g. Picasa (from Google) Effects
  * PhotoshopOnlineFree.net
  * FotoFlexer.com
* All of these rely on HTML 5 features to do their job
  * getImageData() selects some (a rectangular area) or all of an image, returning…
  * … a pixel array, where each pixel is encoded as a 4-element array:

```
img = context.getImageData( x, y, width, height );
red = img.data[0];
green = img.data[1];
blue = img.data[2];
alpha = img.data[3];
```

* We can now manipulate the 4 values to change the image
* Can then copy the image back using putImageData();

This is not something we'll try in the lab, although it is easy enough to do. For example
use a small nested loop to remove (set to zero) the green component of each pixel in
an area of an image, then call putImageData() to draw this back on to the canvas.



Transformations

- The drawing context can also perform whole (or part) image transformations
  - context.scale(x_scale, y_scale);
  - context.rotate(angle_in_radians);
  - context.translate(change_x, change_y);
- These can also be done in sequence (e.g. rotate, then translate, then scale)
- There are also composite operations
  - context.transform();
  - setTransform()
  - See http://devhammer.net/blog/exploring-html5-canvas-part-4---transformations

Again, no details and no lab work here, but there is a lot of good material on the web. Try "HTML Canvas transformations" as a Google search term.