|  |  |  |  |
| --- | --- | --- | --- |
| **Unit:** | 6G4Z2101 Introduction to Web Design & Development: HTML5/JavaScript Instructions & Assignment | Weeks | 10-11 |
| **Learning goals:** | * Understand how shapes are drawn on the HTML5 <canvas> element * Use event-driven JavaScript to draw a range of regular shapes on the HTML5 canvas | | |

|  |
| --- |
| **Tools required:** |
| - Web Browser(s)  - text/web editor (e.g. Notepad++, Dreamweaver [code mode], TextWrangler) |
|

These notes will enable you to understand the basics of HTML5’s Canvas API via exploration and practice. When you have finished, you will be able to draw a variety of regular shapes directly onto the canvas element of a web page using JavaScript: lines, boxes, balls, triangles, diamonds, flags. You should bring questions and problems to your tutor in the lab session.

You need to build your understanding rather than just key in code correctly. Before you follow step-by-step instructions, you **must** carefully read the sections headed *Explanation*. Then when you key in code, you should try to relate it to the explanation. And write comments that reflect this.

If you are reading a section headed ‘**Explanation’** and it includes code, **wait** until you get to the section headed ***‘Putting into Practice’*** before keying in any code.

## Introducing the canvas element

**Explanation** [i.e. *read only: don’t start typing until section 2!]*

As well as video, it is HTML5’s ability to draw graphics dynamically onto a drawing surface known as the ‘Canvas’ that has seen it touted as a ‘Flash-killer’. The graphics are also normally drawn using JavaScript. This tutorial will show you how to draw static 2D shapes directly onto a web page, creating different shapes and setting colours and styles. Canvas, unlike XML-based Scalable Vector Graphics (SVG), is able to accelerate performance via the GPU.

The basic canvas element that goes into our markup, takes the following simple form:

<canvas> </canvas>

To do anything with it, though, we will need to be able to identify it in our JavaScript. You are already familiar with the different means of getting elements from the DOM, but as we saw in the video controls tutorial and assessment, the most natural way to identify an element in JavaScript is to get the element by its id. So we will normally add the id attribute, and give it a value. For example:

<canvas id="myCanvas"> </canvas>

As canvas is a visual element, it also has two specific optional attributes to determine its size. For example:

<canvas id="myCanvas" width="120" height="160"></canvas>

If you do not give width and height values in your HTML, the canvas will default to **300 pixels** wide and **150 pixels** high.

To provide for browsers that don’t support the <canvas> element, we simply place the alternate content inside the tag. This could be a text description of the canvas content, or an image representing the content:

<canvas id="clock" width="150" height="150">

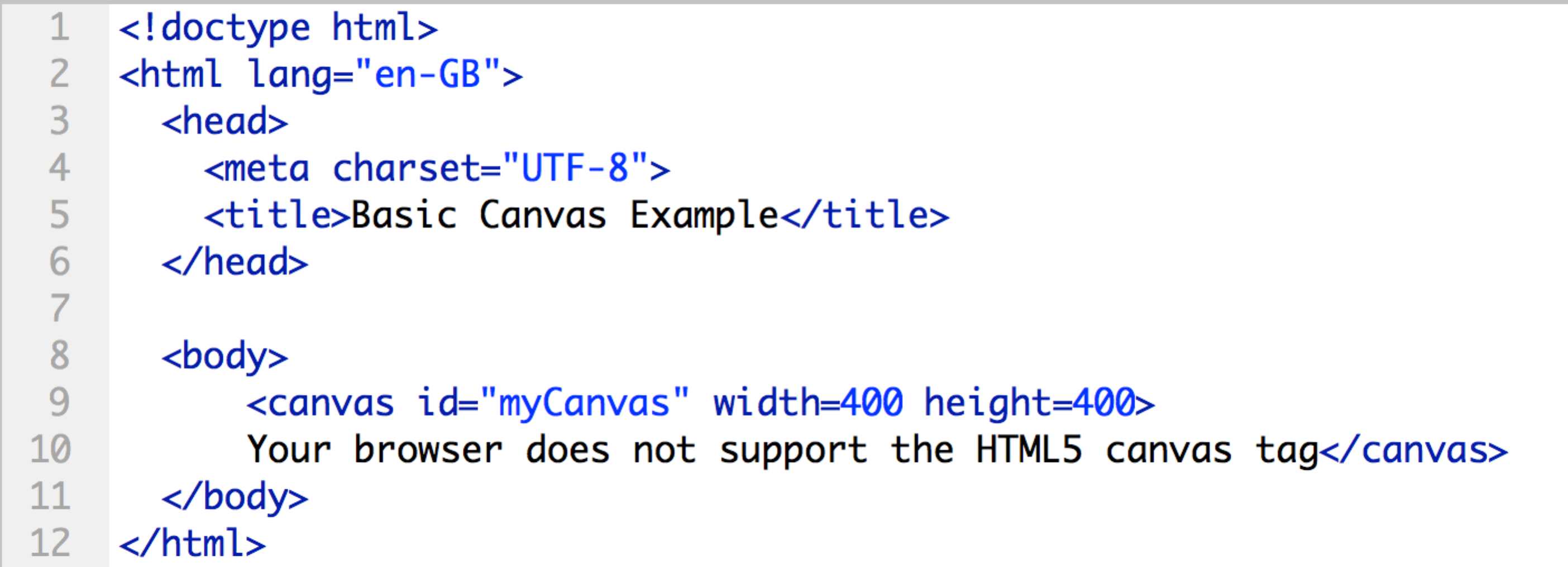
<img src="images/clock.png" width="150"

height="150" alt="clock drawing"/>

</canvas>

## Putting into Practice: writing the <canvas> tag in HTML [start typing!]

*a. Setting up the basic html*Using a tool of your choice (e.g. NotePad++, Dreamweaver, TextWrangler, Brackets, Sublime), **create a new text file, containing the basic markup below, and save it to your disk as a html file** (call it something meaningful):

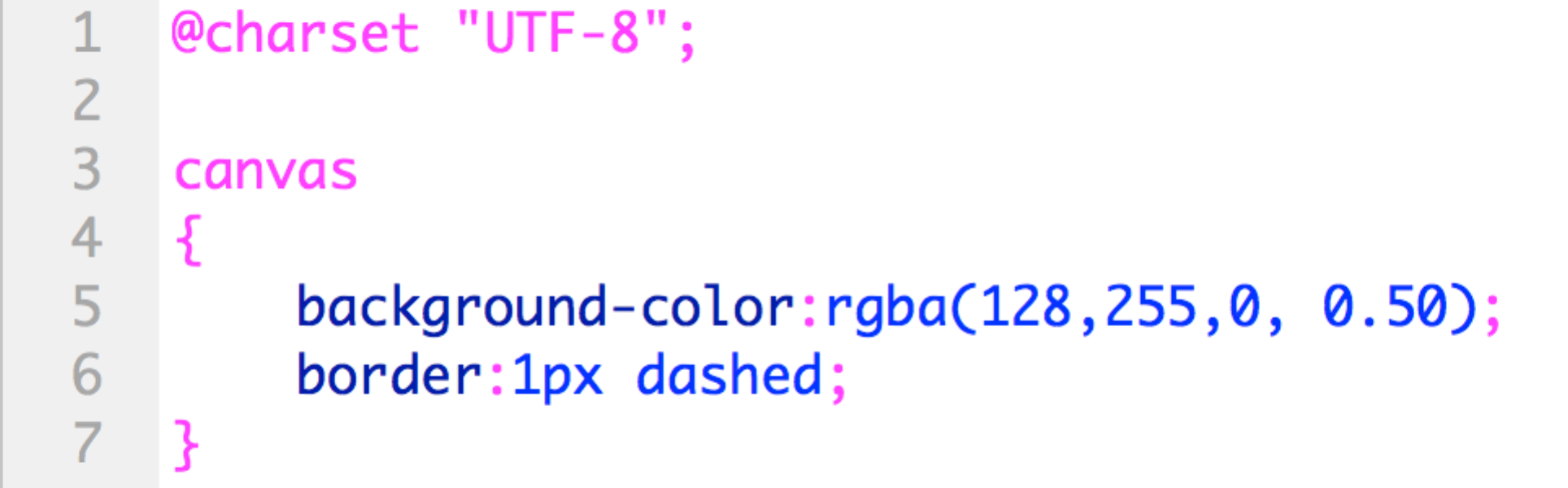


This is all basic markup that you are already familiar with. When you load the page in a browser, you will see absolutely nothing because there is nothing in the canvas as yet! We will need to code graphic content in JavaScript.

*b. add styling so we can see the perimeter of the canvas*

Before we start coding, we can add a style to the canvas element so that we can see it and confirm that it is there.

As you know, styles should be contained in a separate .css file: **create a new text or css file as below, and save it with a .css extension, in the same folder for now as your .html file**:



We now have two files – a .html file, and a .css file. We need to edit our html file so as to include a link to this .css file. In your **html** file, **go to the line immediately after the title tag, and enter the following link element**:

<link rel="stylesheet" href="simpleCanvas.css">

If you named your .css file something else, then change the href string accordingly.

When you have done this and saved the html document, **load it in Chrome**. You should see a rectangle with a dotted-line perimeter and a light green background.

Check: If you cannot see your canvas, go back over the instructions to check your markup and stylesheet, then check with your lab tutor.

Modify any of the stylesheet values as you wish.

**Explanation**

This css creates a border around the canvas. The background colour (given here in rgba, but could also be specified in hexadecimal or by name) is added just as a further means of making the blank canvas visible.

The rel attribute will give the specified linked document’s relationship to the current document by setting it to stylesheet, and the **href** attribute to the URL of our .css file.

(Note that if we set the width and the height here, it would affect not just the canvas itself but also anything we draw on it – hence the use instead of the canvas width and height attributes in the markup)

## JavaScript: the rendering context:

**Explanation**

The canvas is a drawing area, similar to the Flash stage, that is initially blank. To display content we need JavaScript to do three main things first:

1. identify the element from our HTML5 document;
2. obtain its rendering context; and then
3. draw on this context.
4. We are already familiar with the getElementById JavaScript method for getting an element from the markup: here you will transfer your understanding of that from the video controls tasks, to the task of creating a canvas drawing.   
     
   Once we have the element stored in a variable, we can draw on it via its 2D rendering context.
5. To get the context, we can use the getContext() JavaScript method on the variable storing the element reference, and supply it simply with the ‘2d’ rendering context.
6. The rendering context possesses an extensive range of drawing functions that we can then call on to create and manipulate graphics. These are functions that are primarily provided through the user interface in Flash, but which must be hand-coded in HTML5. And, unlike in Flash where each drawn object is retained and can be referred to subsequently, these functions draw on the canvas in immediate mode: the canvas does not have its own child objects. In other words, we cannot refer back to any individual shapes we have drawn, only to the canvas. We can save and restore the state of the context at particular points, but not individual elements within it.

We have created the <canvas> element in our HTML5 above, and we have visually checked that it is present by making it visible. Now we will hook up the JavaScript to the element, getting it by its ID.

As you already know, the DOM represents the structure of web pages and, as an API, provides the means to traverse and manipulate it. The simplest method for getting the document node in JavaScript is getElementById(). To obtain the node by id, we simply provide it with the id name that we used in HTML. If you check your html file, you’ll see that we gave it the id myCanvas. Our JavaScript therefore would be getElementById ( ‘myCanvas’ ). We will assign this to a variable for ease of future use, then get its 2d context and assign that to a further variable.

In the example that follows, for you to copy, we use ‘canvas’ and ‘context’ as identifiers for the variables used to hold the canvas and context respectively, but the names can be anything as long as they are meaningful.

## Putting into Practice: drawing a box

Just as we decouple stylesheets, so we will once again write our JavaScript in a separate file: **create a new text file and save it to the same folder blank for now with the name canvasScript.js. Then in your html document immediately before the css link (or, if you prefer, immediately before the end of the <body> tag), add a link to your .js file:**

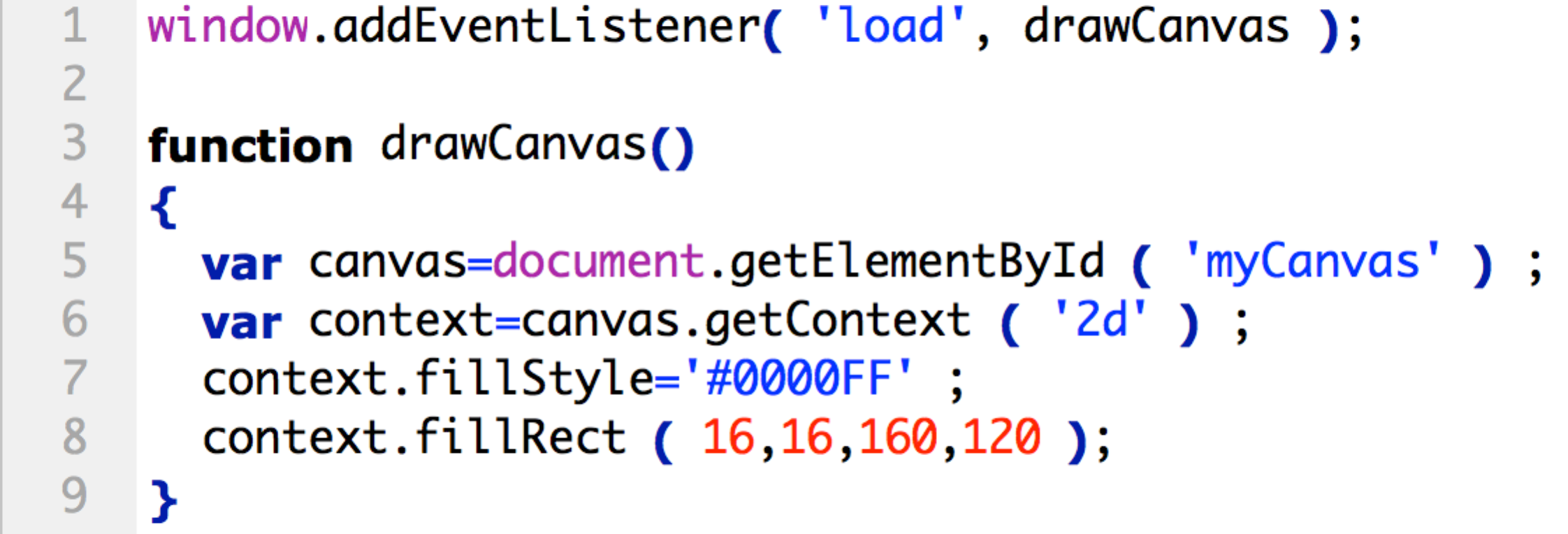
<script src="canvasScript.js"></script>

**Save your html document.**

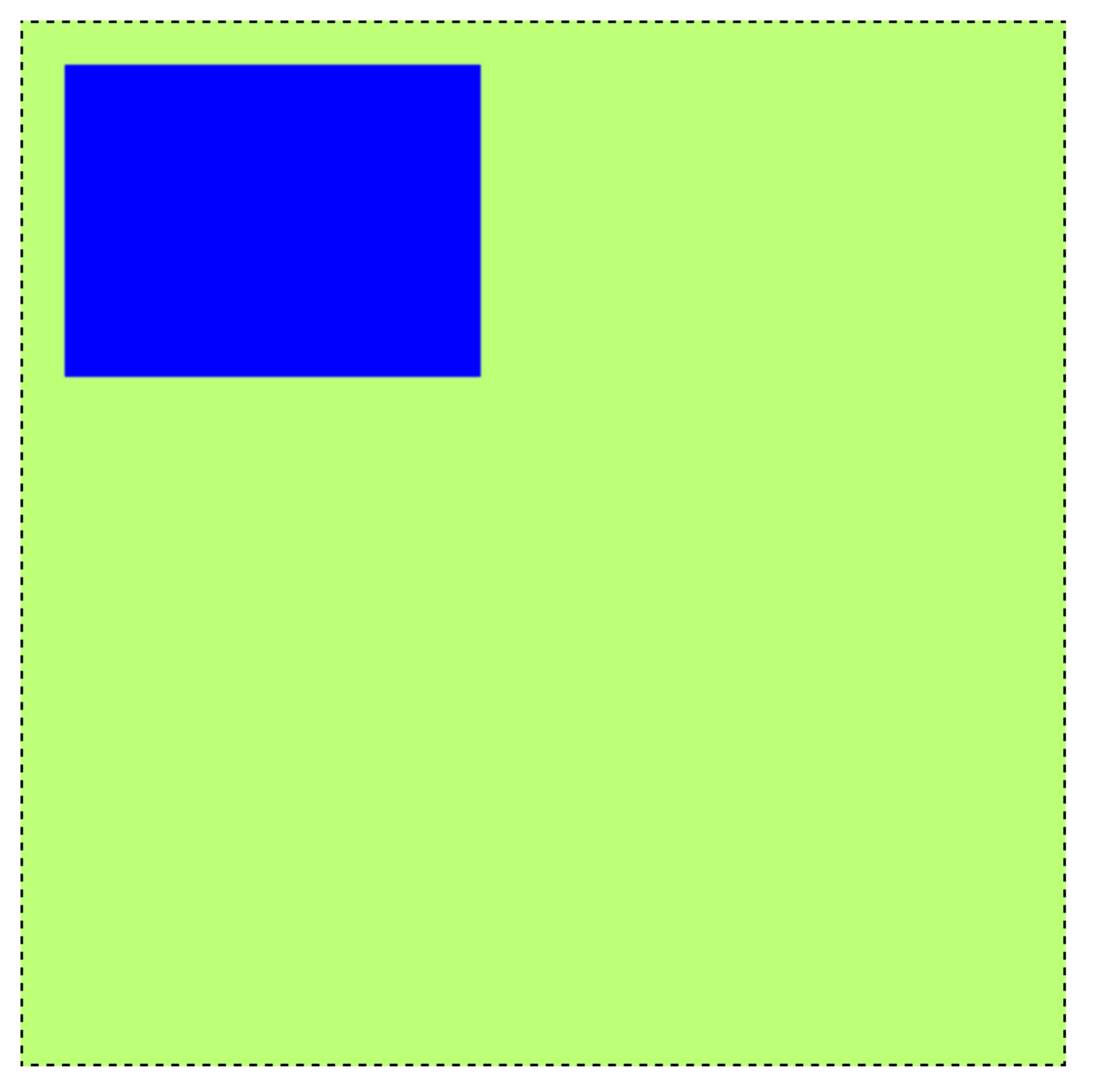
Note that how you handle multiple linked files will depend on your editing tool. In Dreamweaver for example you will only have the html file open: Dreamweaver automatically places clickable links to all linked files within the single tab for the open html file:

Now to get down to business!

In your canvasScript.js file, **key in the code below. Draft your own comments as you go along, to explain what you think is happening. Refer to the week 11 lecture to check your understanding of canvas coordinates.**

**

**Save your JavaScript file and then open the html file in a browser.** *It should display a 160x120 blue rectangle at position 16, 16 of the canvas (ie near the top left corner):*

**

*If there is no blue box as above, check your code and then check with your tutor.*

Explore the four parameters of the fillRect function by changing each one at a time and then reloading the page to check the difference.

**Explanation**

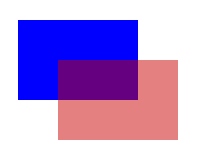
Line 1 listens for the window to trigger the load event, and handles it by calling the function we have named drawCanvas. We are therefore calling the function that begins at line 3 only when the document has fully loaded. Recall from the video tutorial that we could alternatively listen to the document for the DOMContentLoaded event.

For handling events we will again use **event listeners** – in this case one that will ‘listen’ to the window for its load event and then handle the event by running a **named** function. This is discussed in detail in the week 10 Lecture, and is mandatory: **never** use code inline or DOM-zero events!

These are all key concepts and practices! Go over the material above again if you do not understand any aspect. And speak to your tutor! It is not enough simply to copy the code out and get it to work.

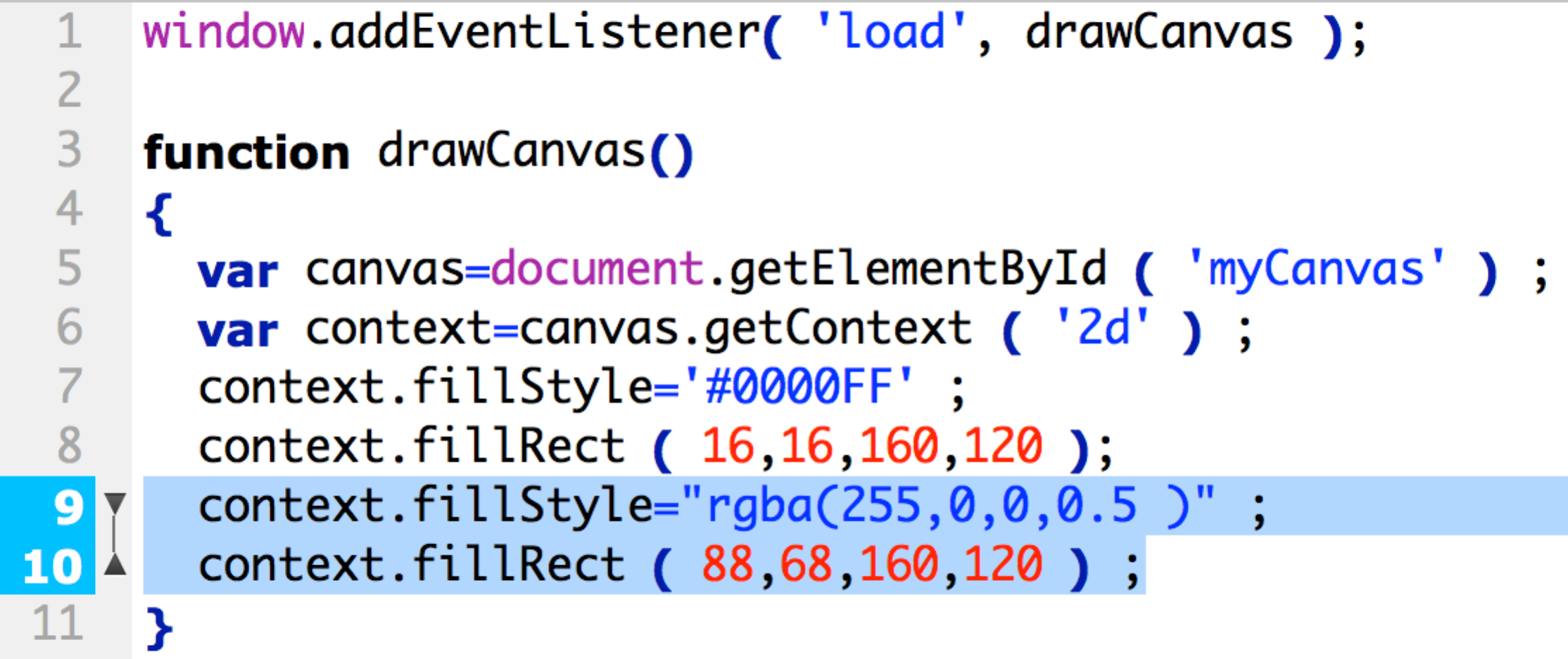
## Adding more shapes:

Now we know how to draw a rectangle, it is simple to add another rectangle to our previous rectangle, overlapping the first one. All we need to do is repeat the steps in the previous section, but with different coordinates. If you played around with the parameters of fillRect then you should have a feel for the coordinates and dimensions. This time we will make the second rectangle semi-transparent, like so:

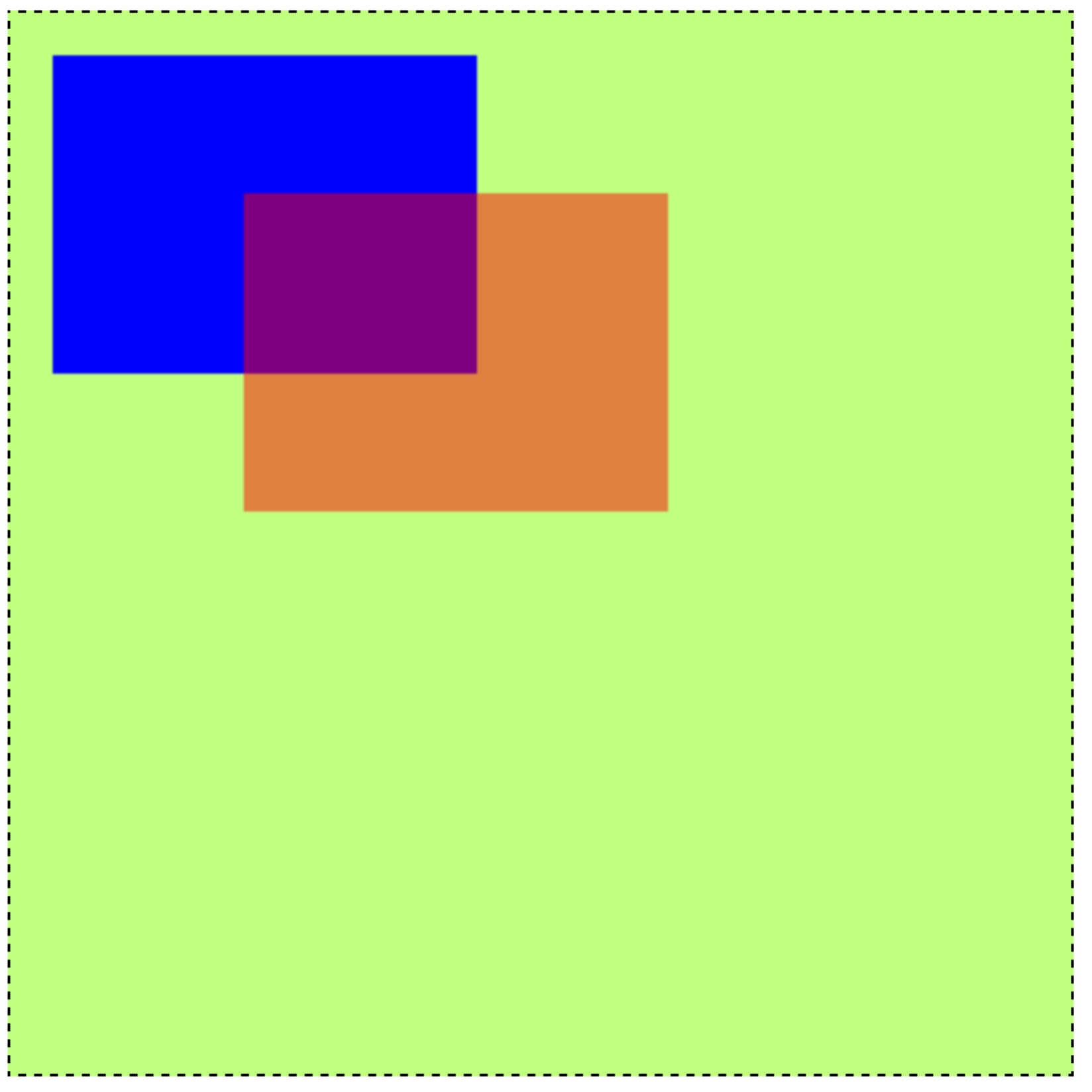
****

Putting into Practice: adding an overlapping rectangle:

To do this, we simply repeat the **fillStyle** and **fillRect** method calls, working out the coordinates necessary to offset the second rectangle from the first. In this example the offset is 40 pixels, further across and down than the first. Add the two highlighted lines 9 and 10 below to your existing JavaScript code, along with your own comment.



**Test in Chrome.** You should see this:



Now see what happens when you comment out ( // at the start of the line) the second fillStyle (line 9 in the code listing above). You should notice that without setting the style again, the new rectangle would keep the previous colour.

**Explanation**

Each drawing is a new layer, which will appear on top of previous layers, in the order drawn.

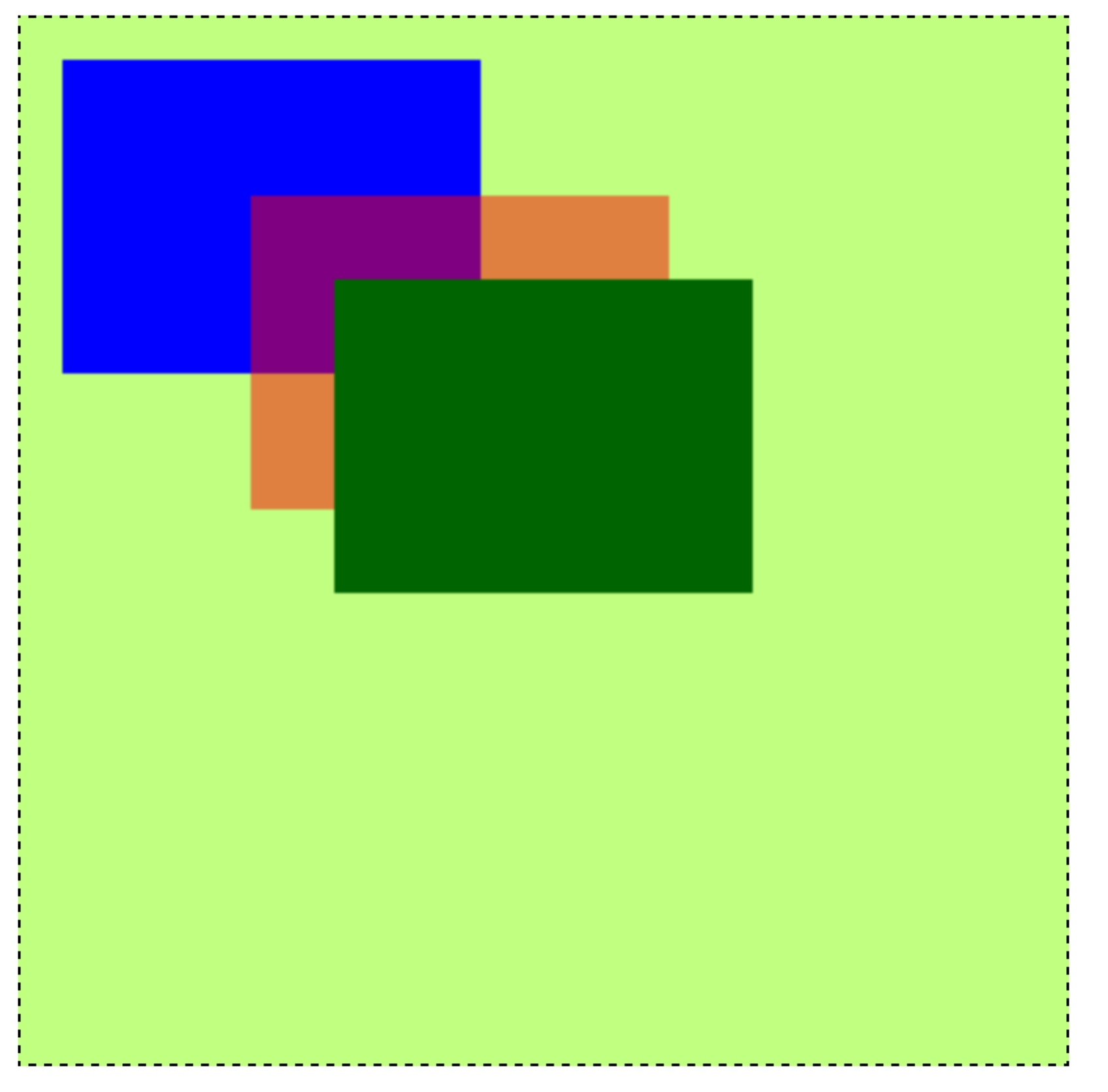
To consolidate your understanding of the coordinates, experiment by changing the coordinate parameters above and reloading. Refer to the lecture coverage of the coordinate space of the canvas element.

Note again the different methods for specifying colour (and of course hexadecimal, e.g. ctx.fillStyle = ‘#0000FF’) – but also that, as with so much of HTML5 at present, these are not necessarily supported by all browsers. If in doubt, use Chrome yourself, but do also check other browsers.

*Note: If using rgb decimal values, don’t use a space between rgb or rgba and the opening parenthesis for the arguments.* Some colours (red, green, blue, gray, purple, silver..) can be specified by name, in the form of a string, e.g. context.fillStyle= "yellow". This can be more readable, though it’s worth checking browser support for more obscure colours (papayawhip etc).

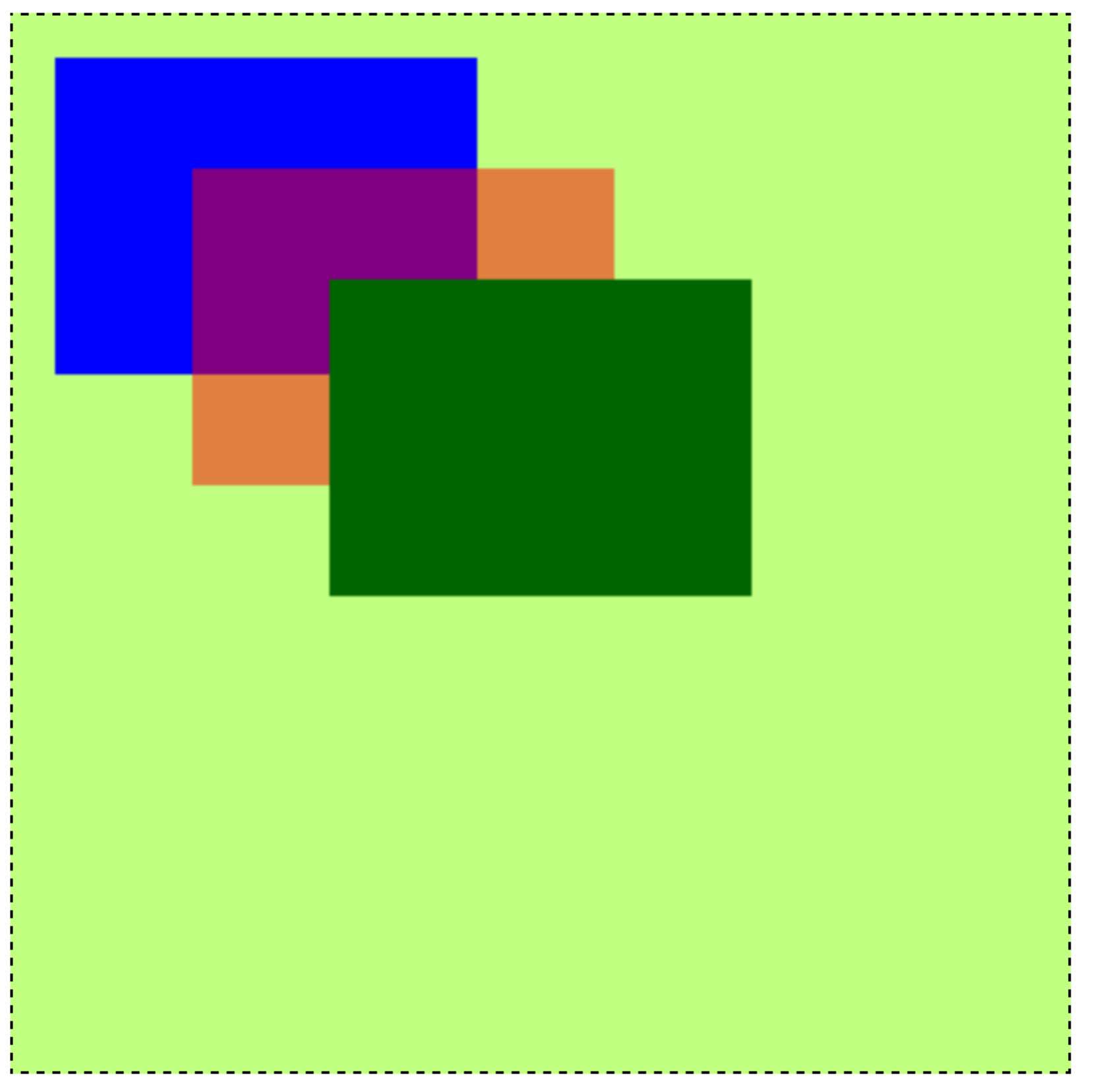
1. ***Transfer-of-Understanding Task.***Now add a third rectangle of the same dimensions, coloured dark green, opaque, and offset 32 pixels vertically and horizontally from the second.

|  |  |
| --- | --- |
| [6. Three rectangles] Done |  |



1. ***Transfer-of-Understanding Task:*** Now change the coordinates of the middle rectangle so it is spaced evenly or equidistantly as below:

|  |  |
| --- | --- |
| [7. Three distributed rectangles] Done |  |



1. ***Transfer-of-Understanding Task:*** create a new document and draw some national flags: Ireland, Italy, Germany, Holland, Hungary, Romania, or Nigeria, Ivory Coast, or Yemen are all simple. Refer to above and week 11 lecture notes concerning canvas coordinates, and see Wikipedia for precise flag details (including rgb or hex colours, and proportions).

|  |  |
| --- | --- |
| [8. Simple flag] Done |  |

## Show all these completed tasks to your lab tutor and discuss.

## Drawing lines

**Explanation**

As explained in the lecture, there is no simple drawLine method, so we need to make our own. Think about the properties that a line has.

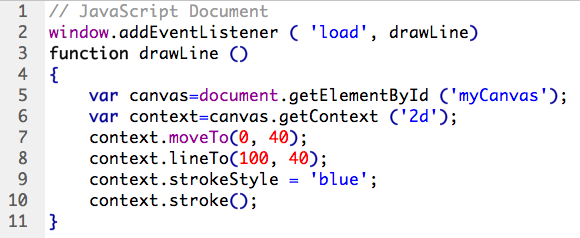
The most basic properties that a line needs, are a start point and an end point. Each of these will be a set of x, y coordinates. We will need to move to the start point, then map out a line to the end point; and then ink in the line with a given colour. The relevant drawing context methods for these steps are: moveTo, lineTo, and stroke; and to pick the colour, we can use the context’s strokeStyle property.

We will stick with these four steps for now, but bear in mind that if we want to subsequently draw different lines using different colours, we will also need to open and close the paths of the line to decouple it from other lines.

**Putting into Practice**

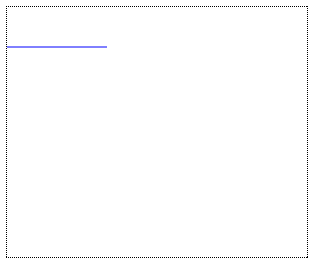
**Create another new JavaScript file, hooking it up as before with a new basic HTML file using a <script> tag that points to a JavaScript file.**

**Enter the following code in your new JavaScript file:**

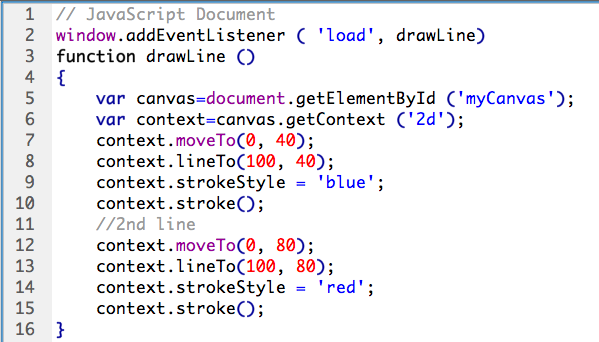


**Explanation**

This specifies a starting-point 40 pixels down on the side of the canvas (0,40) and an end-point 100 pixels across (100, 40). The strokeStyle property is set as blue, before we draw the line (note here yet another way of setting colour). The stroke() function will therefore then draw a horizontal blue line that is 40 pixels down the canvas, from the left-hand edge. Test in a browser. Your output should look like this:



**Putting into Practice**

**Now add a second line further down (lines 12-15 in the code below), using a different colour: copy the drawing code you have just written, and change the y coordinate:**

**Now test in a browser.** You will notice that both colours have become confused, even though we changed the strokeStyle! They are not blue and red.

As already mentioned, if we want a line to be treated differently from others, then we need to open and close its path. So to fix this colour problem, **add context.beginPath() before each moveTo (lines 7 and 12 above); and context.closePath() after each lineTo (or after each stroke() – lines 9 and 14 or lines 11 and 16).** Once the blue line is closed, it will keep its colour when the next strokeStyle is set. Also **make the line thicker by setting the context.lineWidth property to 4** (the logical place to do this would be between the moveTo and lineTo). Your lines should now be blue and red.

1. ***Transfer Understanding Task:*** use a for loop to draw a set of horizontal lines 1 pixel wide and 10 pixels apart.

|  |  |
| --- | --- |
| [10. Parallel lines] Done |  |

**Explanation**

The for loop will take the form

for (var y = startValue; y < stopValue; y += step)

{

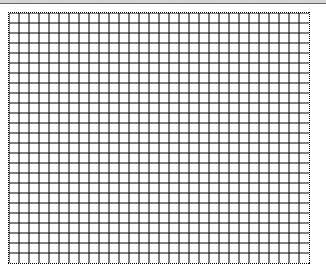
//code here

}

(Note that the single-letter variable identifier is acceptable here, as it represents the y coordinate). In this case, as the grid lines will share the same style/colour, the code repeated will simply be two lines: the moveTo and lineTo. You will then set the strokeStyle property to ‘black’ and call the stroke() method just the once, after both for loops are complete, to ink in all the lines.

1. **Transfer Understanding Task: parts a, b, and c:**

|  |
| --- |
| a. use another for loop to draw a second set of vertical lines 10 pixels apart, to form a grid.   Set your canvas to 250x300 in the HTML and use these as your stop values. This will produce a grid as below. |

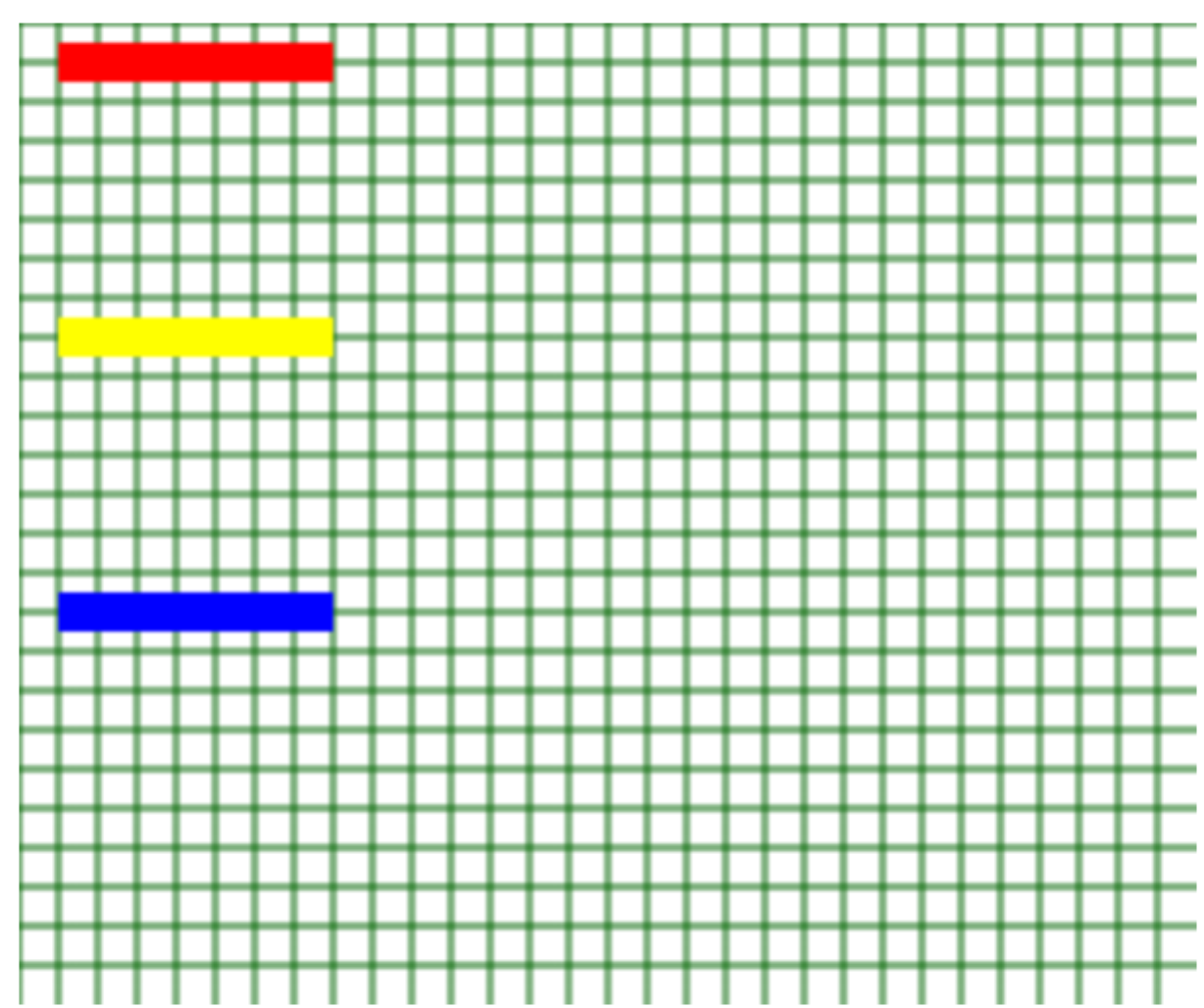


b. When this is complete, change the canvas width in your html to 600x400 and test again. The grid will be incomplete, because we have only drawn the lines to 250 across and 300 down. Instead of 250x300 it is better if we reference the canvas.width and canvas.height properties in the for loops.

Substitute these instead of the numbers in the for loops and reload to check that the grid takes up the entire canvas, whatever its dimensions might be.

c. Now draw three horizontal bars on the grid, coloured red, yellow and blue respectively (see below), and spaced 60 or 70 pixels apart. Make the line width thicker so you can see the coloured lines more clearly. Note that the grid is a separate set of paths: to create the coloured lines you will need to begin and close a new path each time.

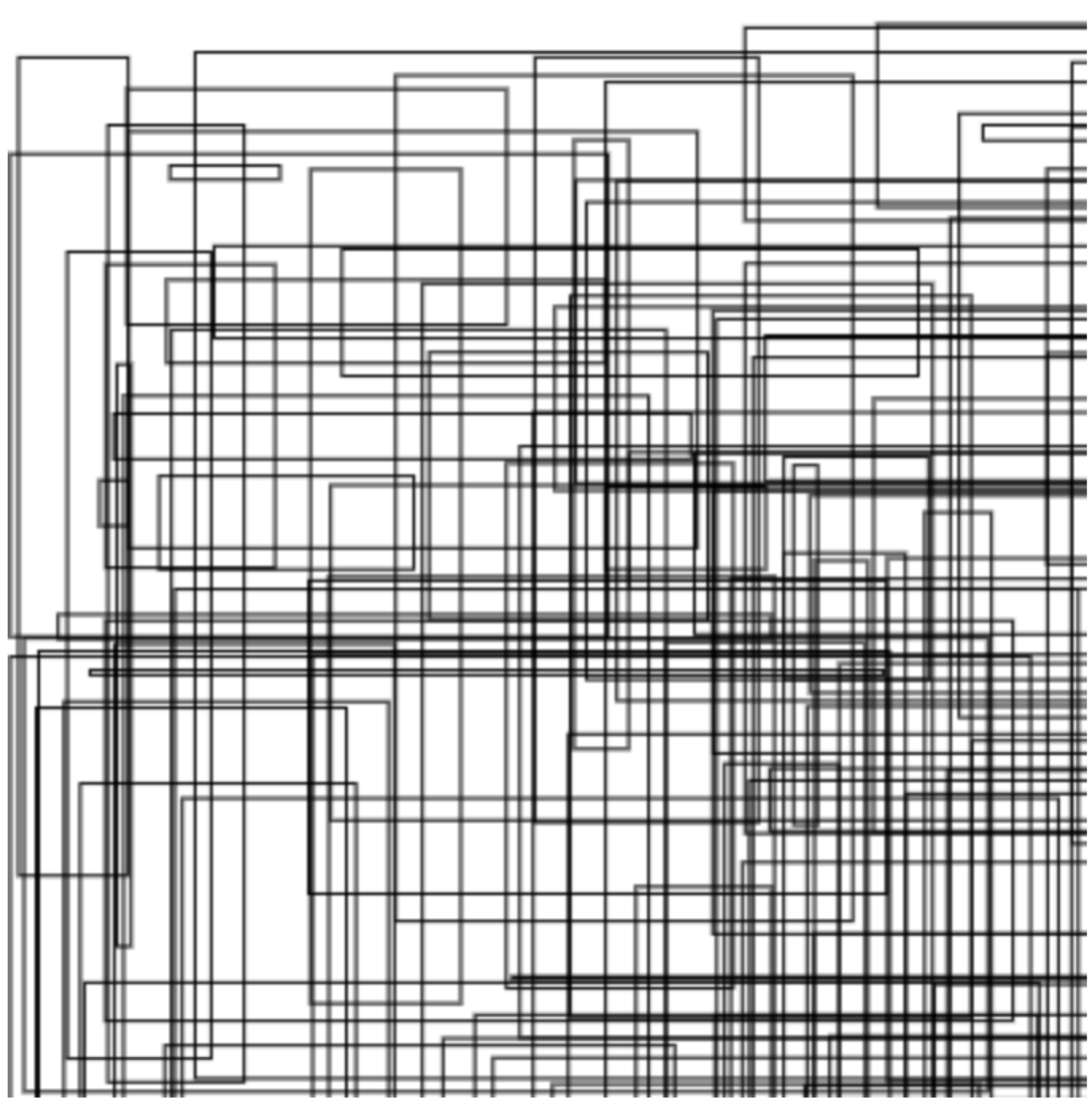
|  |  |
| --- | --- |
| [11. Grid with bars] Done |  |



1. **Assessed Task:** **After Mondrian**

|  |
| --- |
| To be done **without assistance**, and using the techniques learned above.  Write code to generate 100 randomly sized rectangles at random locations within a canvas. See example and hint below.  **Getting it marked:**   1. Fill out your Assignment Checklist/Self-Review Form. 2. Get your work checked against the checklist by your lab tutor, and act on feedback 3. Ask your lab tutor to complete the Summative Assessment Form and award your marks |

Hint: use the canvas width and height with the Math.random() function.

****

|  |  |
| --- | --- |
| [12. Mondrian] Done |  |

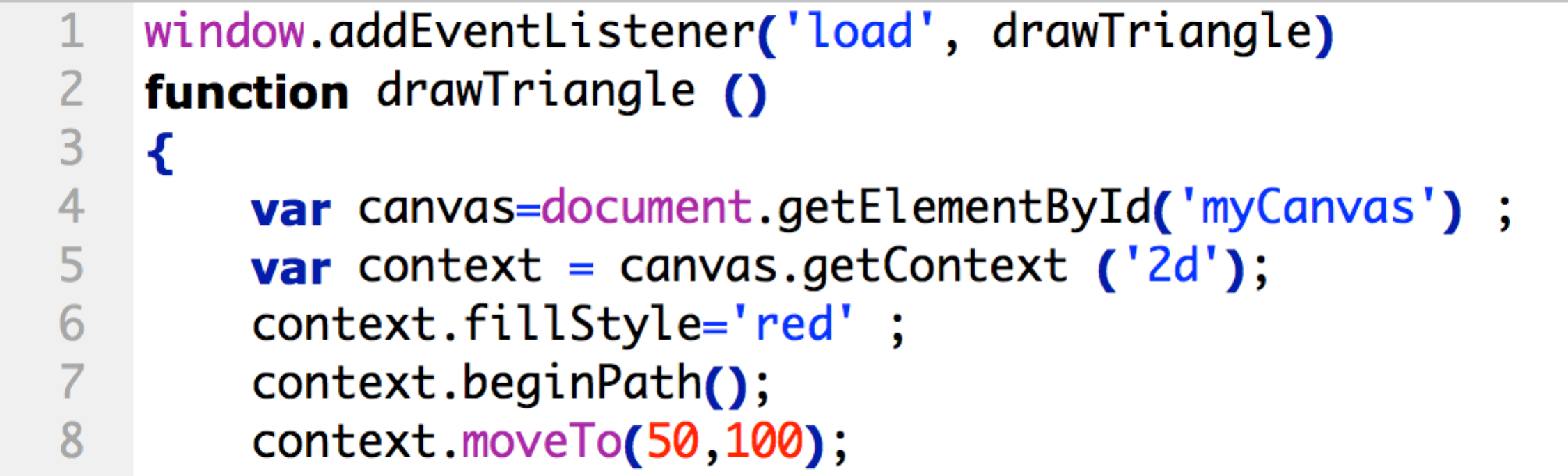
## Drawing a triangle

There is no ready-made drawTriangle method in canvas – again, we will create our own by drawing two connected lines then closing the path.

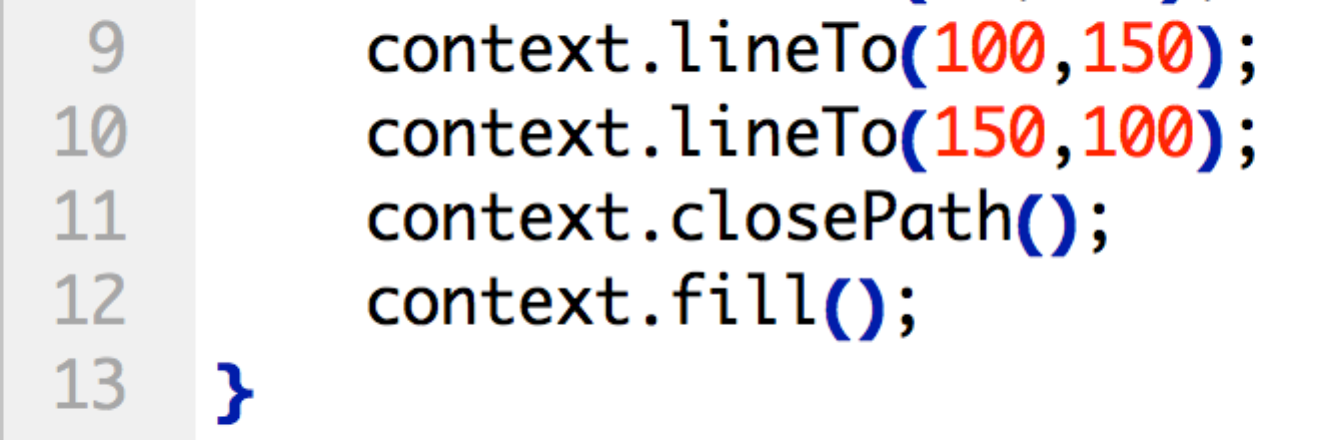
**Putting into Practice**

**Create and save a new basic html file (or copy/Save As your grid document), include a script element linking to a file called drawTriangle.js, then create and save drawTriangle.js.**

**We can use the lineTo method to draw a triangle. First we set the fill colour, begin a path, and then move to the start position as follows:**



Now that we are in position, the line-drawing can begin. If we draw two lines in a v-shape, the third line can be completed by closing the path. **Type the following immediately after context.moveTo(50,100):**



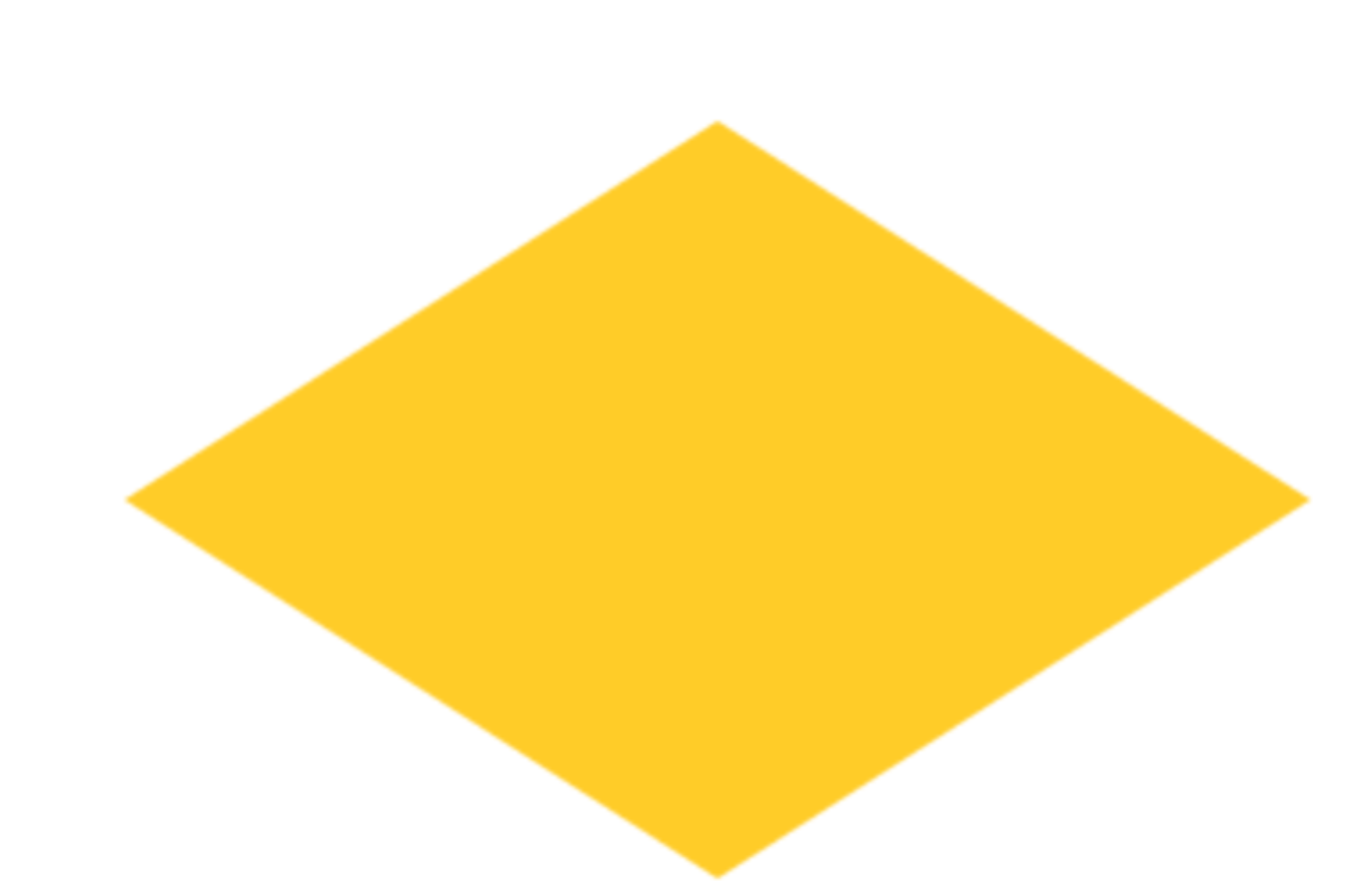
A triangle should now be drawn.

1. To be sure you understand what’s happening here, sketch out the drawing from line 8 to line 10 below (reproduce the grid on paper if you don’t have a printout):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| [14. Triangle coordinates] Done | | | | | |  |
|  | 50 | 100 | 150 | 200 |
| 50 |  |  |  |  |
| 100 |  |  |  |  |
| 150 |  |  |  |  |
| 200 |  |  |  |  |

|  |
| --- |
| 1. *Transfer-of-Understanding Task: Draw an equilateral diamond (rhombus) as below. The top should be at 50,100, bottom at 250, 100, with the sides at 150,50 and 150,150.* |

|  |  |
| --- | --- |
| [15. Rhombus] Done |  |



## Drawing a circle

There is no drawCircle or drawEllipse method – yet again, we need to make our own!

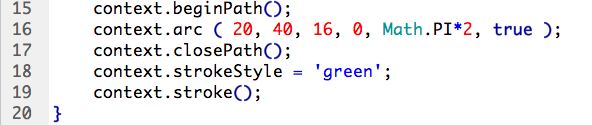
What we **do** have though is an arc method, which we can use to create a circle: as an arc is part of a circle, we can create a circle via a full 360 degree arc.

The arc method takes 6 parameters as follows:

|  |  |
| --- | --- |
| **x** | x coordinate (across) for the centre of the circle |
| **y** | Y coordinate (down) for the centre of the circle |
| **radius** | The radius of the circle of which the arc will be part |
| **startAngle** | Radian point in the circle where the arc starts |
| **endAngle** | Radian point in the circle where the arc stops |
| **anticlockwise** | Drawing direction defaults as anticlockwise=true |

The starting angle 0 is east (right side). To determine the endAngle, we need to consider radians. An arc that is the length of the radius of a circle, is a radian. A full circle will consist of a number of radians that is two times pi (each radian is just under 57.3 degrees), so setting the endAngle to 2 times pi in radians will describe a full circle. The radius will determine the circle’s size. Explore the excellent interactive tutorial at scienceprimer.com/drawing-circles-JavaScript-html5-canvas-element, also observe how the fill(), stroke() and closePath() methods are used with the arc() method.

The following code will therefore draw a simple circle (you can either add this to your existing code, or replace the triangle code):



1. Modify this code to draw a filled circle. Show the circle to your lab tutor.

|  |  |
| --- | --- |
| [17. Filled Circle] Done |  |

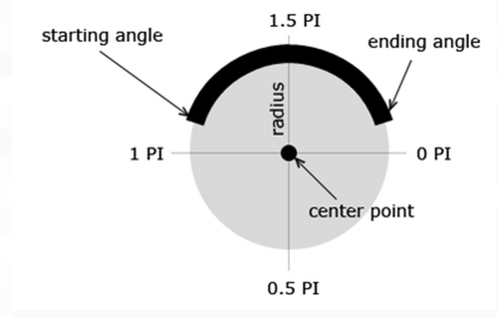


Figure 1: http://www.html5canvastutorials.com/tutorials/html5-canvas-arcs/

1. **Assessed Task:** **Draw an outline flag of Brazil.**

|  |
| --- |
| To be done **without assistance**, and using the techniques learned above.  The outline of the flag of Brazil that you should create, is represented below. Precise details of the flag, including aspect ratio, colours, and positioning, should be followed: these are available at <https://en.wikipedia.org/wiki/Flag_of_Brazil>  **Getting it marked:**   1. Fill out your Assignment Checklist/Self-Review Form. 2. Get your work checked against the checklist by your lab tutor, and act on feedback 3. Ask your lab tutor to complete the Summative Assessment Form and award your marks |



|  |  |
| --- | --- |
| [18. Map of Brazil] Done |  |

*Shapes summary*

We have drawn some basic shapes on the HTML5 canvas. You should now be familiar with basic drawing, and with the canvas coordinate space. Bear in mind that drawing more sophisticated shapes will (unlike in Flash, where shapes can simply be drawn with interface tools) require some abstract mathematical thinking.

Higher-level tools exist that can support the drawing process. For example, you can explore shape drawing code using the code generator at <http://www.htmlcanvasstudio.com/painterpage.aspx>

Another difference is that shapes are drawn in immediate mode – that is, they cannot be manipulated subsequently as separate objects. This is important when it comes to coding animation. Adobe provide an alternative to coding animation from scratch - Edge Animate is like Flash for HTML5, and is available on the lab computers.

## Adding text.

The drawing context’s method for drawing a rectangle was fillRect; the method for drawing text on the canvas is fillText.

fillText takes three parameters: the first is the actual text to display, and the other two are the x and y coordinates where you want to place the text. So the basic syntax would be, for example, context.fillText (“Hello World!”, 70,80).

This is straightforward: **add this line to your triangle code to see**.

However, it is good practice to control the font by specifying its style, size and typeface. We can **specify the typeface along with its style and size using the font property of our drawing context object**:

context.font = “Bold 32px Georgia”

We can additionally **set the alignment of the text**:

context.textAlign = “center”

And **add the colour by using fillStyle**:

context.font = “Bold 32px Georgia” ;

context.textAlign = “center” ;

context.fillStyle = “#FF00FF” ;

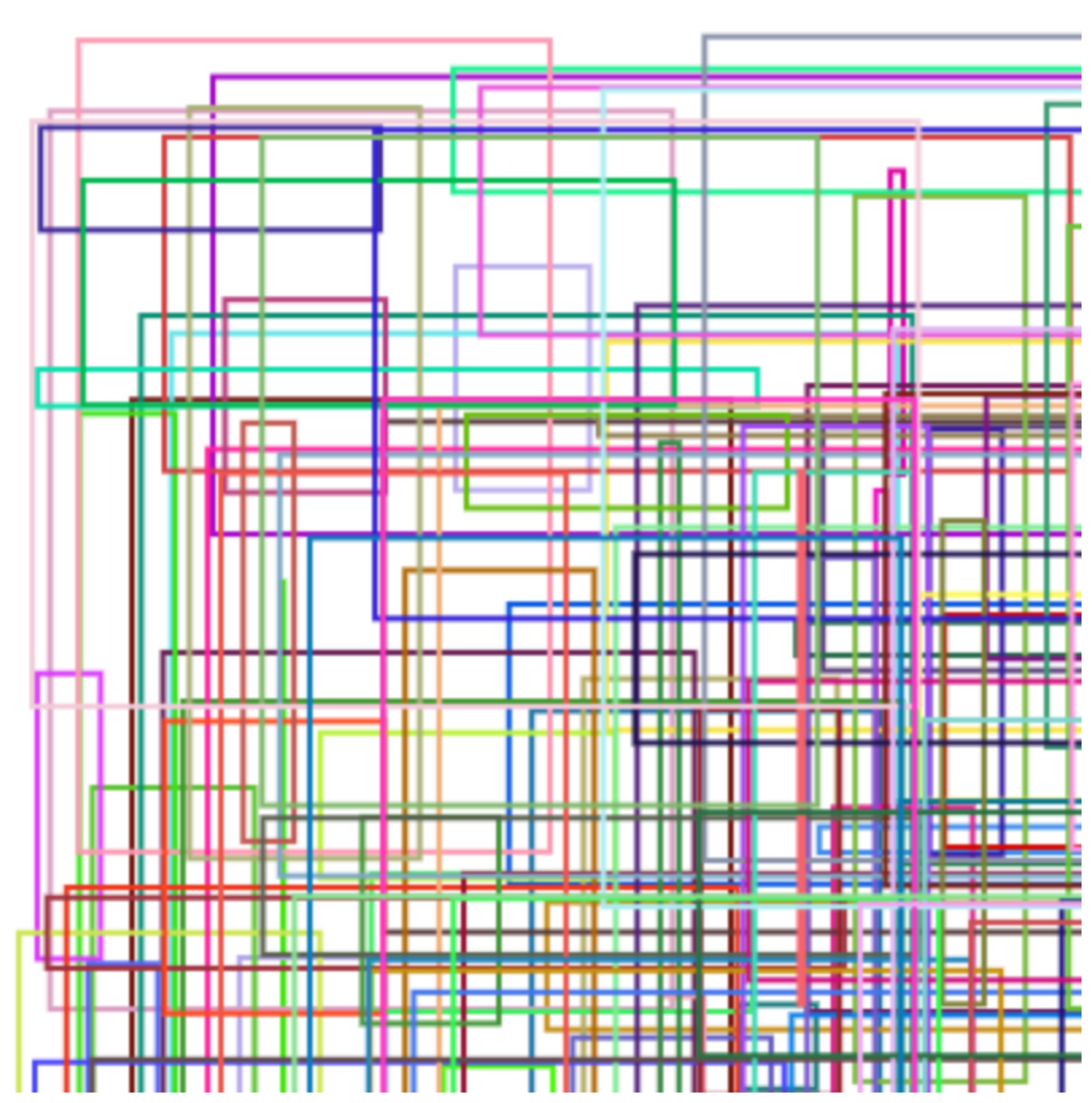
If you run this using your existing x and y coordinates you will realise that centering simply moves the registration point of the text to the centre and 70 pixels does not allow enough room on the left. If you want to centre it on the canvas, use canvas.width/2 instead of 70 as your x coordinate.

**Further work:**

*Extension Tasks: not assessed*

**1. More flags:**Draw the flags of the Czech Republic, Jamaica, and Palestine.

**2. More Mondrian**Extend the ‘Mondrian’ drawing by allocating random colours to the rectangles, as below:



3. Create a **business e-card** for yourself with canvas: include a logo.

4. **Reusability**: Write your own drawEllipse function that you can call to draw a circle by supplying it with a context variable, coordinates, and width/height.

5. **Animation**: View the [HTML5: Graphics and Animation with Canvas with Joe Marini](http://www.lynda.com/HTML-tutorials/Creating-basic-animation/80782/85071-4.html?org=mmu.ac.uk) tutorial section on Lynda.com. This is section 8.3, “Creating a basic animation” [5 min 42 secs]: stop at the end of 8.3, then rewrite the code in line with best-practice.

*Assignment Checklist/Self-Review Form:*

As with the video controls, implementation is evaluated on the basis of code quality criteria as well as functionality. You must achieve these to pass and clearly demonstrate your understanding of what you have learned specifically from the tutorial and the lectures. You can use the same quality criteria checklist for both your video and canvas work:

|  |  |  |  |
| --- | --- | --- | --- |
| Code quality criteria and checklist | | | If No, what is the room for improvement and what needs to be done? {complete following review\*, and advice from tutor} |
|  | Yes | No |  |
| Code is in separate .js file(s) |  |  |  |
| JavaScript code uses DOM Level 2 event listeners. Contains **no** assignment to html event attributes or properties, e.g. onclick=”playVideo”, or playBtn.onclick = “playVideo” |  |  |  |
| **All** identifiers use camelCase where appropriate |  |  |  |
| **All** identifiers clearly indicate purpose to anyone unfamiliar with the code |  |  |  |
| **No** code is copied or adapted from the internet or other sources. |  |  |  |
| Code structure is made clear by systematic use of indentation |  |  |  |
| **Comments** communicate clearly, effectively, and accurately what is happening in the code USD L4.3 |  |  |  |
| Comments don’t just repeat the code |  |  |  |
| Comments demonstrate **understanding** (theory and implementation) |  |  |  |
| Comments are **concise**: notes, not sentences |  |  |  |
| Solutions are complete: in the case of the video, include the guided solutions from the tutorial; in the case of the flag, colours and dimensions are exact. |  |  |  |
|  | | | |

*\* You should self-review prior to checking with your lab tutor on the specified checkpoint date.*

You should achieve all of the above; if you have, and have done all of the components in the weighted list below, you will be awarded **full marks** for each component. This is the objective for the unit – that you independently apply what you have learned to new situations, and achieve the learning outcomes and obtain full marks.

If you have not achieved all of the above, and/or have not done all the components, you will be awarded a partial mark based on the extent to which you have met the criteria in relation to the weighting below. The formative feedback Checkpoint gives you an opportunity to improve your work based on the feedback given.

Significant copying of material (from the internet, books, or fellow students), and other indications that you have not learned what has been taught, is likely to attract a **zero mark**.

**Summative Assessment form and weighting of components**

|  |  |  |  |
| --- | --- | --- | --- |
|  | *If <100%, unresolved room for improvement* | *mark* | *weight* |
| *Video* |  | | |
| 1. *volume slider* |  |  | *5* |
| 1. *current time* |  |  | *5* |
| 1. *playback speed changer* |  |  | *10* |
| 1. *fast-forward + reverse control* |  |  | *10* |
| *Checkpoint :* | *Lab in week commencing 21 November* |  |  |
| *Canvas* |  | | |
| 1. *map of Brazil* |  |  | *10* |
| 1. *Mondrian* |  |  | *10* |
| *Checkpoint:* | *Lab in w.c. 5 December* |  |  |
| *total* | |  | */50* |

**Graduate Outcomes**

Many of the specific criteria above map onto the following Graduate Outcomes and accompanying Standard Descriptors. Graduate Outcome 3 and its grade descriptors reflect in-code comments and identifiers that communicate clearly, effectively, and accurately to other coders what is happening in the code; Graduate Outcome 6 your ability to apply understanding based on learning and reference material.

|  |  |  |
| --- | --- | --- |
| Graduate  Outcome  Grade  Grade range | 3 | 6 |
| Express ideas effectively and communicate information appropriately and accurately using a range of media including ICT | Find, evaluate, synthesise and use information from a variety of sources |
| 86%-100% | Ideas are presented creatively to an audience of peers using a defined range of strategies and media. | Information from primary and secondary sources is carefully collected, analysed, interpreted and applied to new problems under supervision, bringing insight to the analysis. |
| 70%-85% | Ideas are presented fluently to an audience of peers using a defined range of strategies and media. | Information from primary and secondary sources is meticulously collected, analysed, interpreted and applied to specific problems under supervision, bringing insight to the analysis. |
| 60%-69% | Ideas are presented convincingly to an audience of peers using a defined range of strategies and media. | Information from primary and secondary sources is thoroughly collected, analysed, interpreted and applied to specific problems under supervision. |
| 50%-59% | Ideas are presented confidently to an audience of peers using a defined range of strategies and media. | Information from primary and secondary sources is carefully collected, analysed, interpreted and applied to specific problems under supervision. |
| 40%-49% | Ideas are presented adequately to an audience of peers using a defined range of strategies and media. | Information from primary and secondary sources is collected, analysed, interpreted and applied to specific problems under supervision. |
| 35%-39% | Unclear or confused ideas are presented to an audience of peers using a defined range of strategies and media. | Insufficient information from primary and secondary sources is collected, analysed, interpreted and applied to specific problems under supervision, or the analysis of such information is inadequate or incomplete. |
| 20%-34% | Unclear and confused ideas are presented to an audience of peers using a defined range of strategies and media. | Errors are made when information from primary and secondary sources is collected, analysed, interpreted and applied to specific problems under supervision, and the analysis of such information is inadequate and incomplete. |
| 0%-19% | Unstructured or wrong ideas are presented to an audience of peers using a defined range of strategies and media. | Extremely limited or no information from primary and secondary sources is collected, analysed, interpreted and applied to specific problems under supervision. |