Module 12 Resources

Canvas

Please see below for resources related to this module's content. Resources include readings, videos, and attachments related to the module's content (and could show up in one of the quizzes).

**Reading:**

**1.**[Drawing on Canvas](https://westernwyoming.instructure.com/courses/16676/pages/drawing-on-canvas)

**2.** [HTML Canvas ReferenceLinks to an external site.](https://www.w3schools.com/tags/ref_canvas.asp)

**3.**[Canvas lineTo() MethodLinks to an external site.](https://www.w3schools.com/tags/canvas_lineto.asp)

Drawing on Canvas

**Module 12 Reading: Drawing on Canvas**

Browsers give us several ways to display graphics. The simplest way is to use styles to position and color regular DOM elements. This can get us quite far, as the app in the previous chapter showed. By adding partially transparent background images to the nodes, we can make them look exactly the way we want. It is even possible to rotate or skew nodes with the transform style.

But we would be using the DOM for something that it was not originally designed for. Some tasks, such as drawing a line between arbitrary points, are extremely awkward to do with regular HTML elements.

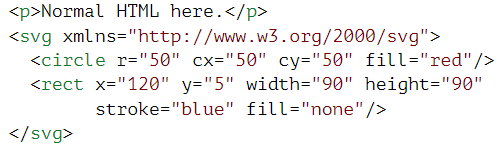
There are two alternatives. The first is DOM based but utilizes Scalable Vector Graphics (SVG) rather than HTML. Think of SVG as a document-markup dialect that focuses on shapes rather than text. You can embed an SVG document directly in an HTML document or include it with an <img> tag.

The second alternative is called a canvas. A canvas is a single DOM element that encapsulates a picture. It provides a programming interface for drawing shapes onto the space taken up by the node. The main difference between a canvas and an SVG picture is that in SVG the original description of the shapes is preserved so that they can be moved or resized at any time. A canvas, on the other hand, converts the shapes to pixels (colored dots on a raster) as soon as they are drawn and does not remember what these pixels represent. The only way to move a shape on a canvas is to clear the canvas (or the part of the canvas around the shape) and redraw it with the shape in a new position.

SVG

This module won’t go into SVG in detail, but I will briefly explain how it works. I will also come back to the trade-offs that you must consider when deciding which drawing mechanism is appropriate for a given application.

This is an HTML document with a simple SVG picture in it:



The xmlns attribute changes an element (and its children) to a different XML namespace. This namespace, identified by a URL, specifies the dialect that we are currently speaking. The <circle> and <rect> tags, which do not exist in HTML, do have a meaning in SVG—they draw shapes using the style and position specified by their attributes.

These tags create DOM elements, just like HTML tags, that scripts can interact with. For example, this changes the <circle> element to be colored cyan instead:



The canvas Element

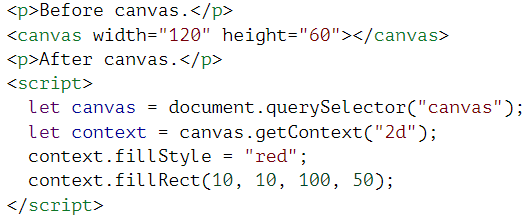
Canvas graphics can be drawn onto a <canvas> element. You can give such an element width and height attributes to determine its size in pixels.

A new canvas is empty, meaning it is entirely transparent and thus shows up as empty space in the document.

The <canvas> tag is intended to allow different styles of drawing. To get access to an actual drawing interface, we first need to create a context, an object whose methods provide the drawing interface. There are currently three widely supported drawing styles: "2d" for two-dimensional graphics, "webgl" for three-dimensional graphics through the OpenGL interface, and "webgpu", a more modern and flexible alternative to WebGL.

We won’t discuss WebGL or WebGPU—we will stick to two dimensions. But if you are interested in three-dimensional graphics, I do encourage you to look into WebGPU. It provides a direct interface to graphics hardware and allows you to render even complicated scenes efficiently, using JavaScript.

You create a context with the getContext method on the <canvas> DOM element.



After creating the context object, the example draws a red rectangle that is 100 pixels wide and 50 pixels high, with its upper-left corner at coordinates (10, 10).

Just like in HTML (and SVG), the coordinate system that the canvas uses puts (0, 0) at the upper-left corner, and the positive y-axis goes down from there. This means (10, 10) is 10 pixels below and to the right of the upper-left corner.

Lines and Surfaces

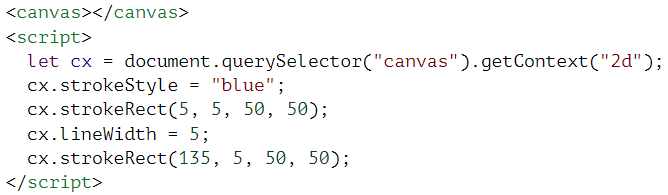
In the canvas interface, a shape can be filled, meaning its area is given a certain color or pattern, or it can be stroked, which means a line is drawn along its edge. SVG uses the same terminology.

The fillRect method fills a rectangle. It takes first the x- and y-coordinates of the rectangle’s upper-left corner, then its width, and then its height. A similar method called strokeRect draws the outline of a rectangle.

Neither method takes any further parameters. The color of the fill, thickness of the stroke, and so on, are not determined by an argument to the method, as you might reasonably expect, but rather by properties of the context object.

The fillStyle property controls the way shapes are filled. It can be set to a string that specifies a color, using the color notation used by CSS.

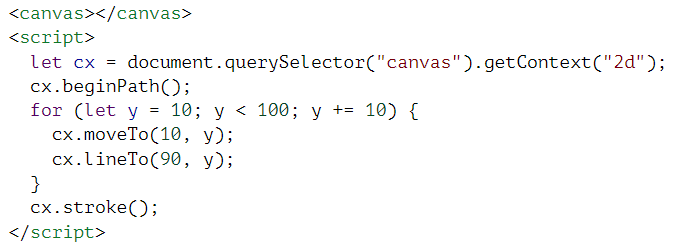
The strokeStyle property works similarly but determines the color used for a stroked line. The width of that line is determined by the lineWidth property, which may contain any positive number.



When no width or height attribute is specified, as in the example, a canvas element gets a default width of 300 pixels and height of 150 pixels.

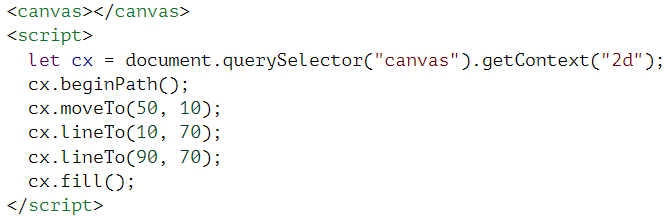
Paths

A path is a sequence of lines. The 2D canvas interface takes a peculiar approach to describing such a path. It is done entirely through side effects. Paths are not values that can be stored and passed around. Instead, if you want to do something with a path, you make a sequence of method calls to describe its shape.



This example creates a path with a number of horizontal line segments and then strokes it using the stroke method. Each segment created with lineTo starts at the path’s current position. That position is usually the end of the last segment, unless moveTo was called. In that case, the next segment would start at the position passed to moveTo.

When filling a path (using the fill method), each shape is filled separately. A path can contain multiple shapes—each moveTo motion starts a new one. But the path needs to be closed (meaning its start and end are in the same position) before it can be filled. If the path is not already closed, a line is added from its end to its start, and the shape enclosed by the completed path is filled.



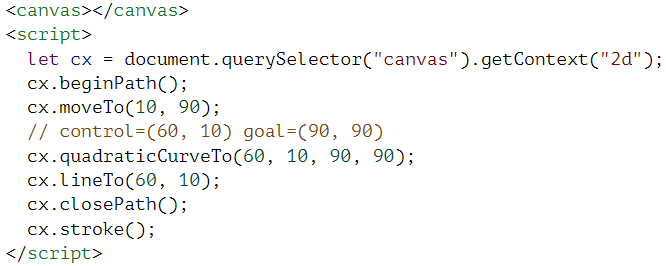
This example draws a filled triangle. Note that only two of the triangle’s sides are explicitly drawn. The third, from the lower-right corner back to the top, is implied and wouldn’t be there if you stroked the path.

You could also use the closePath method to explicitly close a path by adding an actual line segment back to the path’s start. This segment is drawn when stroking the path.

Curves

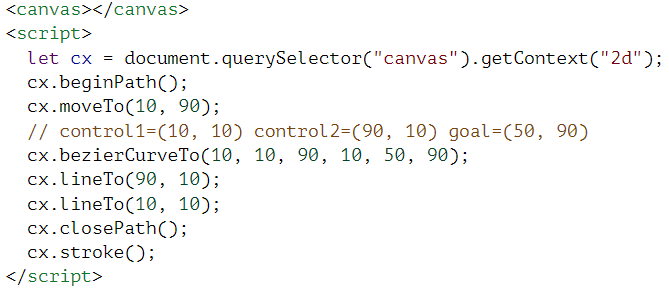
A path may also contain curved lines. These are unfortunately a bit more involved to draw.

The quadraticCurveTo method draws a curve to a given point. To determine the curvature of the line, the method is given a control point as well as a destination point. Imagine this control point as attracting the line, giving it its curve. The line won’t go through the control point, but its direction at the start and end points will be such that a straight line in that direction would point toward the control point. The following example illustrates this:



We draw a quadratic curve from the left to the right, with (60, 10) as the control point, and then draw two line segments going through that control point and back to the start of the line. The result somewhat resembles a Star Trek insignia. You can see the effect of the control point: the lines leaving the lower corners start off in the direction of the control point and then curve toward their target.

The bezierCurveTo method draws a similar kind of curve. Instead of a single control point, this method has two—one for each of the line’s end points. Here is a similar sketch to illustrate the behavior of such a curve:

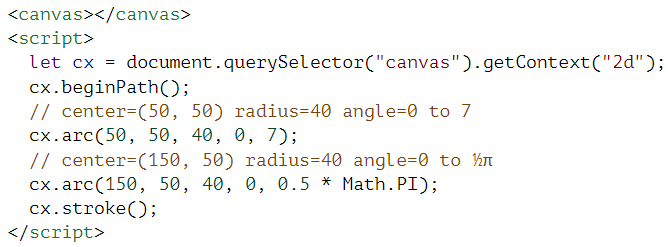


The two control points specify the direction at both ends of the curve. The farther they are away from their corresponding point, the more the curve will “bulge” in that direction.

Such curves can be hard to work with—it is not always clear how to find the control points that provide the shape you are looking for. Sometimes you can compute them, and sometimes you will just have to find a suitable value by trial and error.

The arc method is a way to draw a line that curves along the edge of a circle. It takes a pair of coordinates for the arc’s center, a radius, and then a start angle and end angle.

Those last two parameters make it possible to draw only part of the circle. The angles are measured in radians, not degrees. This means a full circle has an angle of 2π, or 2 \* Math.PI, which is about 6.28. The angle starts counting at the point to the right of the circle’s center and goes clockwise from there. You can use a start of 0 and an end bigger than 2π (say, 7) to draw a full circle.



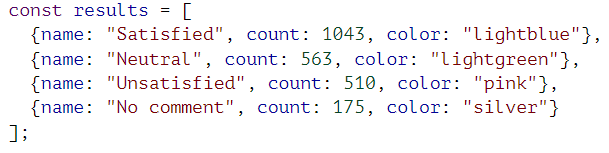
The resulting picture contains a line from the right of the full circle (first call to arc) to the right of the quarter-circle (second call).

Like other path-drawing methods, a line drawn with arc is connected to the previous path segment. You can call moveTo or start a new path to avoid this.

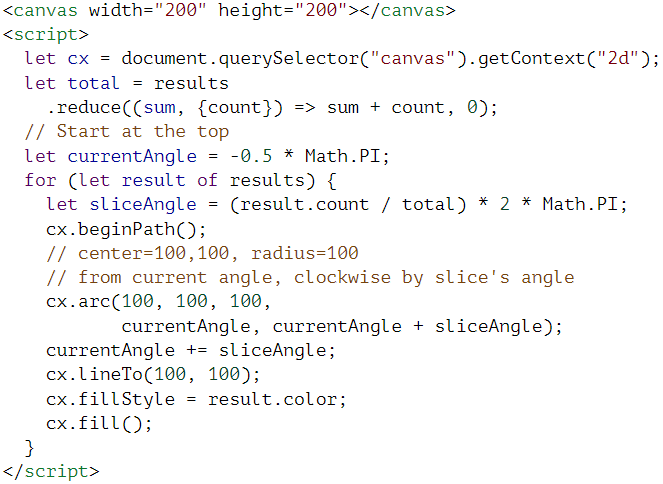
Drawing a Pie Chart

Imagine you have just taken a job at EconomiCorp, Inc. Your first assignment is to draw a pie chart of its customer satisfaction survey results.

The results binding contains an array of objects that represent the survey responses.



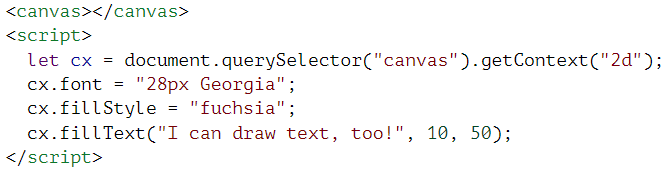
To draw a pie chart, we draw a number of pie slices, each made up of an arc and a pair of lines to the center of that arc. We can compute the angle taken up by each arc by dividing a full circle (2π) by the total number of responses and then multiplying that number (the angle per response) by the number of people who picked a given choice.



But a chart that does not tell us what the slices mean is not very helpful. We need a way to draw text to the canvas.

Text

A 2D canvas drawing context provides the methods fillText and strokeText. The latter can be useful for outlining letters, but usually fillText is what you need. It will fill the outline of the given text with the current fillStyle.



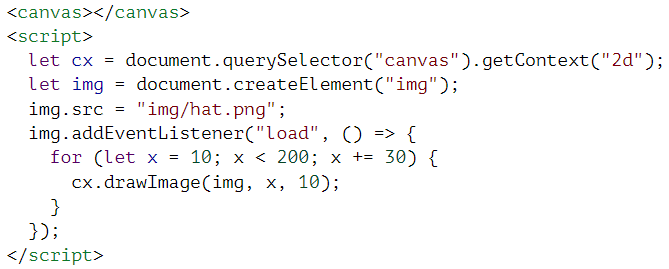
You can specify the size, style, and font of the text with the font property. This example just gives a font size and family name. It is also possible to add italic or bold to the start of the string to select a style.

The last two arguments to fillText and strokeText provide the position at which the font is drawn. By default, they indicate the position of the start of the text’s alphabetic baseline, which is the line that letters “stand” on, not counting hanging parts in letters such as j or p. You can change the horizontal position by setting the textAlign property to "end" or "center" and the vertical position by setting textBaseline to "top", "middle", or "bottom".

Images

In computer graphics, a distinction is often made between vector graphics and bitmap graphics. The first is what we have been doing so far in this chapter—specifying a picture by giving a logical description of shapes. Bitmap graphics, on the other hand, don’t specify actual shapes but rather work with pixel data (rasters of colored dots).

The drawImage method allows us to draw pixel data onto a canvas. This pixel data can originate from an <img> element or from another canvas. The following example creates a detached <img> element and loads an image file into it. But the method cannot immediately start drawing from this picture because the browser may not have loaded it yet. To deal with this, we register a "load" event handler and do the drawing after the image has loaded.



By default, drawImage will draw the image at its original size. You can also give it two additional arguments to specify the width and height of the drawn image, when those aren’t the same as the origin image.

When drawImage is given nine arguments, it can be used to draw only a fragment of an image. The second through fifth arguments indicate the rectangle (x, y, width, and height) in the source image that should be copied, and the sixth to ninth arguments give the rectangle (on the canvas) into which it should be copied.

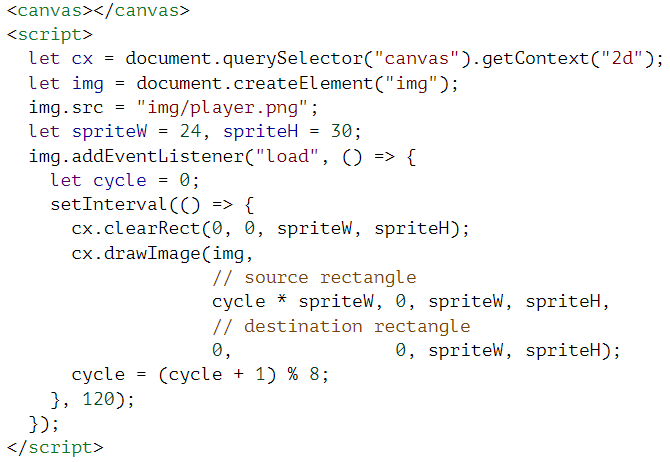
This can be used to pack multiple sprites (image elements) into a single image file and then draw only the part you need. For example, this picture contains a game character in multiple poses:

Pixel art showing a computer game character in 10 different poses. The first 8 form its running animation cycle, the 9th has the character standing still, and the 10th shows him jumping.

By alternating which pose we draw, we can show an animation that looks like a walking character.

To animate a picture on a canvas, the clearRect method is useful. It resembles fillRect, but instead of coloring the rectangle, it makes it transparent, removing the previously drawn pixels.

We know that each sprite, each subpicture, is 24 pixels wide and 30 pixels high. The following code loads the image and then sets up an interval (repeated timer) to draw the next frame:

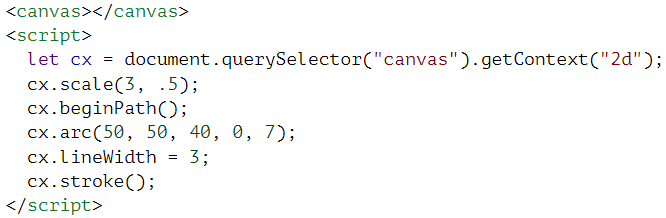


The cycle binding tracks our position in the animation. For each frame, it is incremented and then clipped back to the 0 to 7 range by using the remainder operator. This binding is then used to compute the x-coordinate that the sprite for the current pose has in the picture.

Transformation

What if we want our character to walk to the left instead of to the right? We could draw another set of sprites, of course. But we could also instruct the canvas to draw the picture the other way around.

Calling the scale method will cause anything drawn after it to be scaled. This method takes two parameters, one to set a horizontal scale and one to set a vertical scale.

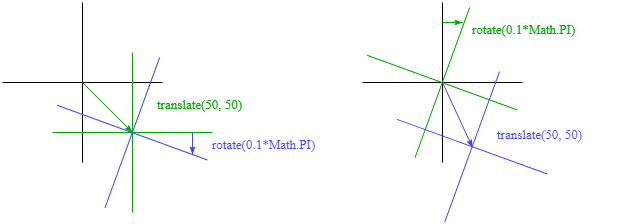


Scaling will cause everything about the drawn image, including the line width, to be stretched out or squeezed together as specified. Scaling by a negative amount will flip the picture around. The flipping happens around point (0, 0), which means it will also flip the direction of the coordinate system. When a horizontal scaling of -1 is applied, a shape drawn at x position 100 will end up at what used to be position -100.

To turn a picture around, we can’t simply add cx.scale(-1, 1) before the call to drawImage. That would move our picture outside of the canvas, where it won’t be visible. We could adjust the coordinates given to drawImage to compensate for this by drawing the image at x position -50 instead of 0. Another solution, which doesn’t require the code doing the drawing to know about the scale change, is to adjust the axis around which the scaling happens.

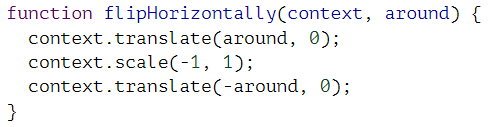
There are several other methods besides scale that influence the coordinate system for a canvas. You can rotate subsequently drawn shapes with the rotate method and move them with the translate method. The interesting—and confusing—thing is that these transformations stack, meaning that each one happens relative to the previous transformations.

If we translate by 10 horizontal pixels twice, everything will be drawn 20 pixels to the right. If we first move the center of the coordinate system to (50, 50) and then rotate by 20 degrees (about 0.1π radians), that rotation will happen around point (50, 50).

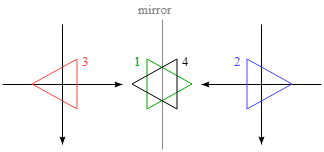


But if we *first* rotate by 20 degrees and *then* translate by (50, 50), the translation will happen in the rotated coordinate system and thus produce a different orientation. The order in which transformations are applied matters.

To flip a picture around the vertical line at a given x position, we can do the following:

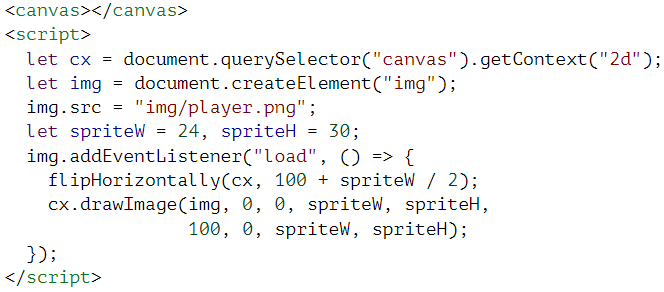


We move the y-axis to where we want our mirror to be, apply the mirroring, and finally move the y-axis back to its proper place in the mirrored universe. The following picture explains why this works:



This shows the coordinate systems before and after mirroring across the central line. The triangles are numbered to illustrate each step. If we draw a triangle at a positive x position, it would, by default, be in the place where triangle 1 is. A call to flipHorizontally first does a translation to the right, which gets us to triangle 2. It then scales, flipping the triangle over to position 3. This is not where it should be, if it were mirrored in the given line. The second translate call fixes this—it “cancels” the initial translation and makes triangle 4 appear exactly where it should.

We can now draw a mirrored character at position (100, 0) by flipping the world around the character’s vertical center.



Storing and Clearing Transformations

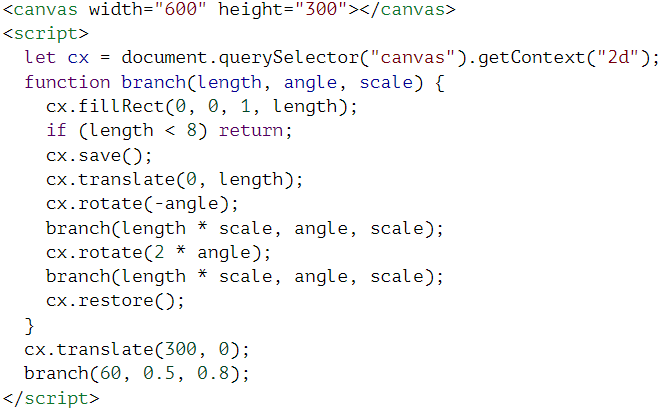
Transformations stick around. Everything else we draw after drawing that mirrored character would also be mirrored. That might be inconvenient.

It is possible to save the current transformation, do some drawing and transforming, and then restore the old transformation. This is usually the proper thing to do for a function that needs to temporarily transform the coordinate system. First, we save whatever transformation the code that called the function was using. Then the function does its thing, adding more transformations on top of the current transformation. Finally, we revert to the transformation we started with.

The save and restore methods on the 2D canvas context do this transformation management. They conceptually keep a stack of transformation states. When you call save, the current state is pushed onto the stack, and when you call restore, the state on top of the stack is taken off and used as the context’s current transformation. You can also call resetTransform to fully reset the transformation.

The branch function in the following example illustrates what you can do with a function that changes the transformation and then calls a function (in this case itself), which continues drawing with the given transformation.

This function draws a treelike shape by drawing a line, moving the center of the coordinate system to the end of the line, and calling itself twice—first rotated to the left and then rotated to the right. Every call reduces the length of the branch drawn, and the recursion stops when the length drops below 8.



If the calls to save and restore were not there, the second recursive call to branch would end up with the position and rotation created by the first call. It would be connected not to the current branch but rather to the innermost, rightmost branch drawn by the first call. The resulting shape might also be interesting, but it is definitely not a tree.