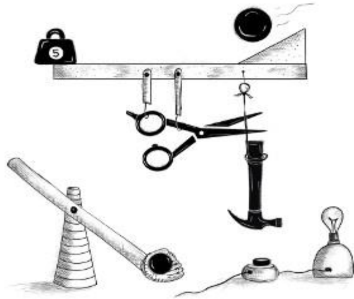
UNIT -IV

Handling Events

You have power over your mind—not outside events. Realize this, and you will find strength.

Marcus Aurelius, *Meditations*



Some programs work with direct user input, such as mouse and keyboard actions. That kind of input isn't available as a well-organized data structure—it comes in piece by piece, in real time, and the program is expected to respond to it as it happens.

Event handlers

Imagine an interface where the only way to find out whether a key on the keyboard is being pressed is to read the current state of that key. To be able to react to keypresses, you would have to constantly read the key's state so that you'd catch it before it's released again. It would be dangerous to perform other time-intensive computations since you might miss a keypress.

Some primitive machines do handle input like that. A step up from this would be for the hardware or operating system to notice the keypress and put it in a queue. A program can then periodically check the queue for new events and react to what it finds there.

Of course, it has to remember to look at the queue, and to do it often, because any time between the key being pressed and the program noticing the event will cause the software to feel unresponsive. This approach is called *polling*. Most programmers prefer to avoid it.

A better mechanism is for the system to actively notify our code when an event occurs. Browsers do this by allowing us to register functions as *handlers* for specific events.

```html
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

The window binding refers to a built-in object provided by the browser. It represents the browser window that contains the document. Calling its addEventListener method registers the second argument to be called whenever the event described by its first argument occurs.

Events and DOM nodes

Each browser event handler is registered in a context. In the previous example we called addEventListener on the window object to register a handler for the whole window. Such a method can also be found on DOM elements and some other types of objects. Event listeners are called only when the event happens in the context of the object they are registered on.

```html
<button>Click me</button>
<p>No handler here.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Button clicked.");
  });
</script>
```

That example attaches a handler to the button node. Clicks on the button cause that handler to run, but clicks on the rest of the document do not.

Giving a node an onclick attribute has a similar effect. This works for most types of events—you can attach a handler through the attribute whose name is the event name with on in front of it.

But a node can have only one onclick attribute, so you can register only one handler per node that way. The addEventListener method allows you to add any number of handlers so that it is safe to add handlers even if there is already another handler on the element.

The removeEventListener method, called with arguments similar to addEventListener, removes a handler.

```
<button>Act-once button</button>
<script>
 let button = document.querySelector("button");
 function once() {
  console.log("Done.");
  button.removeEventListener("click", once);
 }
 button.addEventListener("click", once);
</script>
```

The function given to removeEventListener has to be the same function value that was given to addEventListener. So, to unregister a handler, you'll want to give the function a name (once, in the example) to be able to pass the same function value to both methods.

Event objects

Though we have ignored it so far, event handler functions are passed an argument: the *event object*. This object holds additional information about the event. For example, if we want to know *which* mouse button was pressed, we can look at the event object's button property.

```
<button>Click me any way you want</button>
<script>
 let button = document.querySelector("button");
 button.addEventListener("mousedown", event => {
  if (event.button == 0) {
   console.log("Left button");
  } else if (event.button == 1) {
   console.log("Middle button");
  } else if (event.button == 2) {
   console.log("Right button");
  }
 });
```

```
</script>
```

The information stored in an event object differs per type of event. We'll discuss different types later in the chapter. The object's type property always holds a string identifying the event (such as "click" or "mousedown").

Propagation

For most event types, handlers registered on nodes with children will also receive events that happen in the children. If a button inside a paragraph is clicked, event handlers on the paragraph will also see the click event.

But if both the paragraph and the button have a handler, the more specific handler—the one on the button—gets to go first. The event is said to *propagate* outward, from the node where it happened to that node's parent node and on to the root of the document. Finally, after all handlers registered on a specific node have had their turn, handlers registered on the whole window get a chance to respond to the event.

At any point, an event handler can call the stopPropagation method on the event object to prevent handlers further up from receiving the event. This can be useful when, for example, you have a button inside another clickable element and you don't want clicks on the button to activate the outer element's click behavior.

The following example registers "mousedown" handlers on both a button and the paragraph around it. When clicked with the right mouse button, the handler for the button calls stopPropagation, which will prevent the handler on the paragraph from running. When the button is clicked with another mouse button, both handlers will run.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
 let para = document.querySelector("p");
 let button = document.querySelector("button");
 para.addEventListener("mousedown", () => {
  console.log("Handler for paragraph.");
 });
 button.addEventListener("mousedown", event => {
```

```
    console.log("Handler for button.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

Most event objects have a target property that refers to the node where they originated. You can use this property to ensure that you're not accidentally handling something that propagated up from a node you do not want to handle.

It is also possible to use the target property to cast a wide net for a specific type of event. For example, if you have a node containing a long list of buttons, it may be more convenient to register a single click handler on the outer node and have it use the target property to figure out whether a button was clicked, rather than register individual handlers on all of the buttons.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
      console.log("Clicked", event.target.textContent);
    }
  });
</script>
```

Default actions

Many events have a default action associated with them. If you click a link, you will be taken to the link's target. If you press the down arrow, the browser will scroll the page down. If you right-click, you'll get a context menu. And so on.

For most types of events, the JavaScript event handlers are called *before* the default behavior takes place. If the handler doesn't want this normal behavior to happen, typically because it has already taken care of handling the event, it can call the preventDefault method on the event object.

This can be used to implement your own keyboard shortcuts or context menu. It can also be used to obnoxiously interfere with the behavior that users expect. For example, here is a link that cannot be followed:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

Try not to do such things unless you have a really good reason to. It'll be unpleasant for people who use your page when expected behavior is broken.

Depending on the browser, some events can't be intercepted at all. On Chrome, for example, the keyboard shortcut to close the current tab (CONTROL-W or COMMAND-W) cannot be handled by JavaScript.

Key events

When a key on the keyboard is pressed, your browser fires a "keydown" event. When it is released, you get a "keyup" event.

```
<p>This page turns violet when you hold the V key.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
      document.body.style.background = "";
```

```
    }
  });
</script>
```

Despite its name, "keydown" fires not only when the key is physically pushed down. When a key is pressed and held, the event fires again every time the key *repeats*. Sometimes you have to be careful about this. For example, if you add a button to the DOM when a key is pressed and remove it again when the key is released, you might accidentally add hundreds of buttons when the key is held down longer.

The example looked at the key property of the event object to see which key the event is about. This property holds a string that, for most keys, corresponds to the thing that pressing that key would type. For special keys such as ENTER, it holds a string that names the key ("Enter", in this case). If you hold SHIFT while pressing a key, that might also influence the name of the key—"v" becomes "V", and "1" may become "!", if that is what pressing SHIFT-1 produces on your keyboard.

Modifier keys such as SHIFT, CONTROL, ALT, and META (COMMAND on Mac) generate key events just like normal keys. But when looking for key combinations, you can also find out whether these keys are held down by looking at the shiftKey, ctrlKey, altKey, and metaKey properties of keyboard and mouse events.

```
<p>Press Control-Space to continue.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("Continuing!");
    }
  });
</script>
```

The DOM node where a key event originates depends on the element that has focus when the key is pressed. Most nodes cannot have focus unless you give them a tabindex attribute, but things like links, buttons, and form fields can. We’ll come back to form fields in Chapter 18. When nothing in particular has focus, document.body acts as the target node of key events.

When the user is typing text, using key events to figure out what is being typed is problematic. Some platforms, most notably the virtual keyboard on Android phones, don’t fire key events. But even when you have an old-fashioned keyboard, some types of text input don’t match key presses in a straightforward way, such as *input method editor* (IME) software used by people whose scripts don’t fit on a keyboard, where multiple key strokes are combined to create characters.

To notice when something was typed, elements that you can type into, such as the <input> and <textarea> tags, fire "input" events whenever the user changes their content. To get the actual content that was typed, it is best to directly read it from the focused field. Chapter 18 will show how.

Pointer events

There are currently two widely used ways to point at things on a screen: mice (including devices that act like mice, such as touchpads and trackballs) and touchscreens. These produce different kinds of events.

Mouse clicks

Pressing a mouse button causes a number of events to fire. The "mousedown" and "mouseup" events are similar to "keydown" and "keyup" and fire when the button is pressed and released. These happen on the DOM nodes that are immediately below the mouse pointer when the event occurs.

After the "mouseup" event, a "click" event fires on the most specific node that contained both the press and the release of the button. For example, if I press down the mouse button on one paragraph and then move the pointer to another paragraph and release the button, the "click" event will happen on the element that contains both those paragraphs.

If two clicks happen close together, a "dblclick" (double-click) event also fires, after the second click event.

To get precise information about the place where a mouse event happened, you can look at its clientX and clientY properties, which contain the event’s coordinates (in pixels) relative to the top-left corner of the window, or pageX and pageY, which are relative to the top-left corner of the whole document (which may be different when the window has been scrolled).

The following implements a primitive drawing program. Every time you click the document, it adds a dot under your mouse pointer. See Chapter 19 for a less primitive drawing program.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* rounds corners */
    background: blue;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

Mouse motion

Every time the mouse pointer moves, a "mousemove" event is fired. This event can be used to track the position of the mouse. A common situation in which this is useful is when implementing some form of mouse-dragging functionality.

As an example, the following program displays a bar and sets up event handlers so that dragging to the left or right on this bar makes it narrower or wider:

```
<p>Drag the bar to change its width:</p>
```

```
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  let lastX; // Tracks the last observed mouse X position
  let bar = document.querySelector("div");
  bar.addEventListener("mousedown", event => {
    if (event.button == 0) {
      lastX = event.clientX;
      window.addEventListener("mousemove", moved);
      event.preventDefault(); // Prevent selection
    }
  });

  function moved(event) {
    if (event.buttons == 0) {
      window.removeEventListener("mousemove", moved);
    } else {
      let dist = event.clientX - lastX;
      let newWidth = Math.max(10, bar.offsetWidth + dist);
      bar.style.width = newWidth + "px";
      lastX = event.clientX;
    }
  }
</script>
```

Note that the "mousemove" handler is registered on the whole window. Even if the mouse goes outside of the bar during resizing, as long as the button is held we still want to update its size.

We must stop resizing the bar when the mouse button is released. For that, we can use the buttons property (note the plural), which tells us about the buttons that are currently held down. When this is zero, no buttons are down. When buttons are held, its value is the sum of the codes for those buttons—the left button has code 1, the right button 2, and the middle one 4. With the left and right buttons held, for example, the value of buttons will be 3.

Note that the order of these codes is different from the one used by button, where the middle button came before the right one. As mentioned, consistency isn't really a strong point of the browser's programming interface.

Touch events

The style of graphical browser that we use was designed with mouse interfaces in mind, at a time where touchscreens were rare. To make the Web "work" on early touchscreen phones, browsers for those devices pretended, to a certain extent, that touch events were mouse events. If you tap your screen, you'll get "mousedown", "mouseup", and "click" events.

But this illusion isn't very robust. A touchscreen works differently from a mouse: it doesn't have multiple buttons, you can't track the finger when it isn't on the screen (to simulate "mousemove"), and it allows multiple fingers to be on the screen at the same time.

Mouse events cover touch interaction only in straightforward cases—if you add a "click" handler to a button, touch users will still be able to use it. But something like the resizeable bar in the previous example does not work on a touchscreen.

There are specific event types fired by touch interaction. When a finger starts touching the screen, you get a "touchstart" event. When it is moved while touching, "touchmove" events fire. Finally, when it stops touching the screen, you'll see a "touchend" event.

Because many touchscreens can detect multiple fingers at the same time, these events don't have a single set of coordinates associated with them. Rather, their event objects have a touches property, which holds an array-like object of points, each of which has its own clientX, clientY, pageX, and pageY properties.

You could do something like this to show red circles around every touching finger:

```
<style>
  dot { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Touch this page</p>
```

```
<script>
  function update(event) {
    for (let dot; dot = document.querySelector("dot");) {
      dot.remove();
    }
    for (let i = 0; i < event.touches.length; i++) {
      let {pageX, pageY} = event.touches[i];
      let dot = document.createElement("dot");
      dot.style.left = (pageX - 50) + "px";
      dot.style.top = (pageY - 50) + "px";
      document.body.appendChild(dot);
    }
  }
  window.addEventListener("touchstart", update);
  window.addEventListener("touchmove", update);
  window.addEventListener("touchend", update);
</script>
```

You'll often want to call preventDefault in touch event handlers to override the browser's default behavior (which may include scrolling the page on swiping) and to prevent the mouse events from being fired, for which you may *also* have a handler.

Scroll events

Whenever an element is scrolled, a "scroll" event is fired on it. This has various uses, such as knowing what the user is currently looking at (for disabling off-screen animations or sending spy reports to your evil headquarters) or showing some indication of progress (by highlighting part of a table of contents or showing a page number).

The following example draws a progress bar above the document and updates it to fill up as you scroll down:

```
<style>
  #progress {
    border-bottom: 2px solid blue;
```

```
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progress"></div>
<script>
  // Create some content
  document.body.appendChild(document.createTextNode(
    "supercalifragilisticexpialidocious ".repeat(1000)));

  let bar = document.querySelector("#progress");
  window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight;
    bar.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>
```

Giving an element a position of fixed acts much like an absolute position but also prevents it from scrolling along with the rest of the document. The effect is to make our progress bar stay at the top. Its width is changed to indicate the current progress. We use %, rather than px, as a unit when setting the width so that the element is sized relative to the page width.

The global innerHeight binding gives us the height of the window, which we have to subtract from the total scrollable height—you can't keep scrolling when you hit the bottom of the document. There's also an innerWidth for the window width. By dividing pageYOffset, the current scroll position, by the maximum scroll position and multiplying by 100, we get the percentage for the progress bar.

Calling preventDefault on a scroll event does not prevent the scrolling from happening. In fact, the event handler is called only *after* the scrolling takes place.

Focus events

When an element gains focus, the browser fires a "focus" event on it. When it loses focus, the element gets a "blur" event.

Some events, like these two and "scroll", do not propagate. A handler on a parent element is not notified when a child element gains or loses focus.

The following example displays help text for the text field that currently has focus:

```
<p>Name: <input type="text" data-help="Your full name"></p>
<p>Age: <input type="text" data-help="Your age in years"></p>
<p id="help"></p>

<script>
  let help = document.querySelector("#help");
  let fields = document.querySelectorAll("input");
  for (let field of Array.from(fields)) {
    field.addEventListener("focus", event => {
      let text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    field.addEventListener("blur", event => {
      help.textContent = "";
    });
  }
</script>
```

The window object will receive "focus" and "blur" events when the user moves from or to the browser tab or window in which the document is shown.

Load event

When a page finishes loading, the "load" event fires on the window and the document body objects. This is often used to schedule initialization actions that require the whole document to have been built. Remember that the content of <script> tags is run immediately when the tag is encountered.

This may be too soon, for example when the script needs to do something with parts of the document that appear after the <script> tag.

Elements such as images and script tags that load an external file also have a "load" event that indicates the files they reference were loaded. Like the focus-related events, loading events do not propagate.

When a page is closed or navigated away from (for example, by following a link), a "beforeunload" event fires. The main use of this event is to prevent the user from accidentally losing work by closing a document. If you prevent the default behavior on this event *and* set the returnValue property on the event object to a string, the browser will show the user a dialog asking if they really want to leave the page. That dialog might include your string, but because some malicious sites try to use these dialogs to confuse people into staying on their page to look at dodgy weight loss ads, most browsers no longer display them.

Events and the event loop

In the context of the event loop, as discussed in Chapter 11, browser event handlers behave like other asynchronous notifications. They are scheduled when the event occurs but must wait for other scripts that are running to finish before they get a chance to run.

The fact that events can be processed only when nothing else is running means that, if the event loop is tied up with other work, any interaction with the page (which happens through events) will be delayed until there’s time to process it. So if you schedule too much work, either with long-running event handlers or with lots of short-running ones, the page will become slow and cumbersome to use.

For cases where you *really* do want to do some time-consuming thing in the background without freezing the page, browsers provide something called *web workers*. A worker is a JavaScript process that runs alongside the main script, on its own timeline.

Imagine that squaring a number is a heavy, long-running computation that we want to perform in a separate thread. We could write a file called code/squareworker.js that responds to messages by computing a square and sending a message back.

```
addEventListener("message", event => {
```

```
  postMessage(event.data * event.data);
});
```

To avoid the problems of having multiple threads touching the same data, workers do not share their global scope or any other data with the main script's environment. Instead, you have to communicate with them by sending messages back and forth.

This code spawns a worker running that script, sends it a few messages, and outputs the responses.

```
let squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", event => {
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

The postMessage function sends a message, which will cause a "message" event to fire in the receiver. The script that created the worker sends and receives messages through the Worker object, whereas the worker talks to the script that created it by sending and listening directly on its global scope. Only values that can be represented as JSON can be sent as messages—the other side will receive a *copy* of them, rather than the value itself.

Timers

We saw the setTimeout function in Chapter 11. It schedules another function to be called later, after a given number of milliseconds.

Sometimes you need to cancel a function you have scheduled. This is done by storing the value returned by setTimeout and calling clearTimeout on it.

```
let bombTimer = setTimeout(() => {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
```

```
  clearTimeout(bombTimer);
}
```

The cancelAnimationFrame function works in the same way as clearTimeout—calling it on a value returned by requestAnimationFrame will cancel that frame (assuming it hasn't already been called).

A similar set of functions, setInterval and clearInterval, are used to set timers that should *repeat* every *X* milliseconds.

```
let ticks = 0;
let clock = setInterval(() => {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

Debouncing

Some types of events have the potential to fire rapidly, many times in a row (the "mousemove" and "scroll" events, for example). When handling such events, you must be careful not to do anything too time-consuming or your handler will take up so much time that interaction with the document starts to feel slow.

If you do need to do something nontrivial in such a handler, you can use setTimeout to make sure you are not doing it too often. This is usually called *debouncing* the event. There are several slightly different approaches to this.

In the first example, we want to react when the user has typed something, but we don't want to do it immediately for every input event. When they are typing quickly, we just want to wait until a pause occurs. Instead of immediately performing an action in the event handler, we set a timeout. We also clear the previous timeout (if any) so that when events occur close together (closer than our timeout delay), the timeout from the previous event will be canceled.

```
<textarea>Type something here...</textarea>
```

```
<script>
  let textarea = document.querySelector("textarea");
  let timeout;
  textarea.addEventListener("input", () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => console.log("Typed!"), 500);
  });
</script>
```

Giving an undefined value to clearTimeout or calling it on a timeout that has already fired has no effect. Thus, we don't have to be careful about when to call it, and we simply do so for every event.

We can use a slightly different pattern if we want to space responses so that they're separated by at least a certain length of time but want to fire them *during* a series of events, not just afterward. For example, we might want to respond to "mousemove" events by showing the current coordinates of the mouse but only every 250 milliseconds.

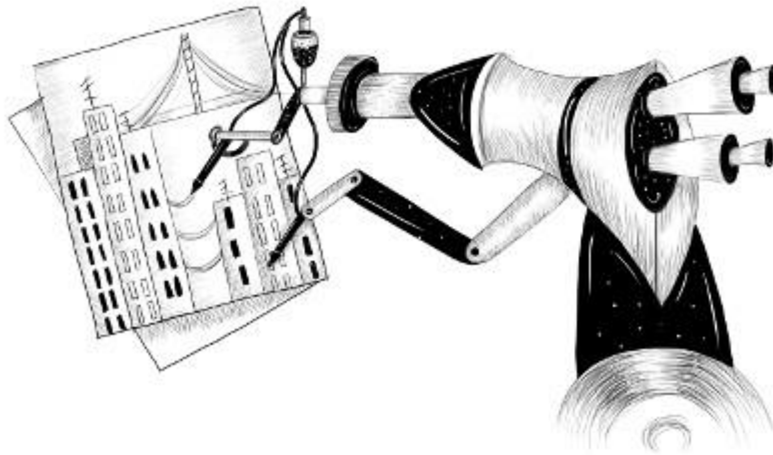
```
<script>
  let scheduled = null;
  window.addEventListener("mousemove", event => {
    if (!scheduled) {
      setTimeout(() => {
        document.body.textContent =
          `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`;
        scheduled = null;
      }, 250);
    }
    scheduled = event;
  });
</script>
```

.

Drawing on Canvas

Drawing is deception.

M.C. Escher, *cited by Bruno Ernst in The Magic Mirror of M.C. Escher*



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 you quite far, as the game 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'd be using the DOM for something that it wasn't 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 book will not go into SVG in detail, but I will briefly explain how it works. At the end of the chapter, I'll 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:

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

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:

```
let circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

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 two widely supported drawing styles: "2d" for two-dimensional graphics and "webgl" for three-dimensional graphics through the OpenGL interface.

This book won't discuss WebGL—we'll stick to two dimensions. But if you are interested in three-dimensional graphics, I do encourage you to look into WebGL. 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.

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

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

Just like in HTML (and SVG), the coordinate system that the canvas uses puts (0,0) at the top-left corner, and the positive y-axis goes down from there. So (10,10) is 10 pixels below and to the right of the top-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. The same terminology is used by SVG.

The fillRect method fills a rectangle. It takes first the x- and y-coordinates of the rectangle's top-left corner, then its width, and then its height. A similar method, 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.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.strokeStyle = "blue";
  cx.strokeRect(5, 5, 50, 50);
  cx.lineWidth = 5;
  cx.strokeRect(135, 5, 50, 50);
</script>
```

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.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
```

```
  cx.beginPath();
  for (let y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

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.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>
```

This example draws a filled triangle. Note that only two of the triangle's sides are explicitly drawn. The third, from the bottom-right corner back to the top, is implied and wouldn't be there when you stroke 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:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control=(60,10) goal=(90,90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
  cx.stroke();
</script>
```

We draw a quadratic curve from the left to the right, with (60,10) as 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 one has two—one for each of the line's endpoints. Here is a similar sketch to illustrate the behavior of such a curve:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
```

```
  cx.beginPath();
  cx.moveTo(10, 90);
  // control1=(10,10) control2=(90,10) goal=(50,90)
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
  cx.lineTo(90, 10);
  cx.lineTo(10, 10);
  cx.closePath();
  cx.stroke();
</script>
```

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’s 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’ll 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.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // center=(50,50) radius=40 angle=0 to 7
  cx.arc(50, 50, 40, 0, 7);
  // center=(150,50) radius=40 angle=0 to ½π
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

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've just taken a job at EconomiCorp, Inc., and 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.

```
const results = [
  {name: "Satisfied", count: 1043, color: "lightblue"},
  {name: "Neutral", count: 563, color: "lightgreen"},
  {name: "Unsatisfied", count: 510, color: "pink"},
  {name: "No comment", count: 175, color: "silver"}
];
```

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.

```
<canvas width="200" height="200"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let total = results
    .reduce((sum, {count}) => sum + count, 0);
  // Start at the top
  let currentAngle = -0.5 * Math.PI;
  for (let result of results) {
    let sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
    // center=100,100, radius=100
    // from current angle, clockwise by slice's angle
```

```
    cx.arc(100, 100, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(100, 100);
    cx.fillStyle = result.color;
    cx.fill();
  }
</script>
```

But a chart that doesn't tell us what the slices mean isn't 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.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.font = "28px Georgia";
  cx.fillStyle = "fuchsia";
  cx.fillText("I can draw text, too!", 10, 50);
</script>
```

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".

We'll come back to our pie chart, and the problem of labeling the slices, in the exercises at the end of the chapter.

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).

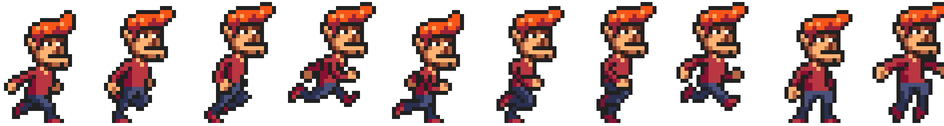
The drawImage method allows us to draw pixel data onto a canvas. This pixel data can originate from an <img> element or from another canvas. The following example creates a detached <img> element and loads an image file into it. But it cannot immediately start drawing from this picture because the browser 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.

```
<canvas></canvas>
<script>
 let cx = document.querySelector("canvas").getContext("2d");
 let img = document.createElement("img");
 img.src = "img/hat.png";
 img.addEventListener("load", () => {
  for (let x = 10; x < 200; x += 30) {
    cx.drawImage(img, x, 10);
  }
 });
</script>
```

By default, drawImage will draw the image at its original size. You can also give it two additional arguments to set a different width and height.

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, we have this picture containing a game character in multiple poses:



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:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    let cycle = 0;
    setInterval(() => {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                   // source rectangle
                   cycle * spriteW, 0, spriteW, spriteH,
                   // destination rectangle
                   0,             0, spriteW, spriteH);
      cycle = (cycle + 1) % 8;
    }, 120);
  });
</script>
```

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

But 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 can also instruct the canvas to draw the picture the other way round.

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.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>
```

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.

So to turn a picture around, we can't simply add cx.scale(-1, 1) before the call to drawImage because that would move our picture outside of the canvas, where it won't be visible. You 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 that does 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.

So 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:

```
function flipHorizontally(context, around) {
  context.translate(around, 0);
  context.scale(-1, 1);
  context.translate(-around, 0);
}
```

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.

```
<canvas></canvas>
```

```
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                 100, 0, spriteW, spriteH);
  });
</script>
```

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.

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```

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 wouldn't be connected 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.

Back to the game

We now know enough about canvas drawing to start working on a canvas-based display system for the game from the previous chapter. The new display will no longer be showing just colored boxes. Instead, we'll use drawImage to draw pictures that represent the game's elements.

We define another display object type called CanvasDisplay, supporting the same interface as DOMDisplay from Chapter 16, namely, the methods syncState and clear.

This object keeps a little more information than DOMDisplay. Rather than using the scroll position of its DOM element, it tracks its own viewport, which tells us what part of the level we are currently looking at. Finally, it keeps a flipPlayer property so that even when the player is standing still, it keeps facing the direction it last moved in.

```
class CanvasDisplay {
  constructor(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");

    this.flipPlayer = false;

    this.viewport = {
      left: 0,
      top: 0,
      width: this.canvas.width / scale,
      height: this.canvas.height / scale
    };
  }

  clear() {
    this.canvas.remove();
  }
}
```

The syncState method first computes a new viewport and then draws the game scene at the appropriate position.

```
CanvasDisplay.prototype.syncState = function(state) {
  this.updateViewport(state);
  this.clearDisplay(state.status);
  this.drawBackground(state.level);
```

```
  this.drawActors(state.actors);
};
```

Contrary to DOMDisplay, this display style *does* have to redraw the background on every update. Because shapes on a canvas are just pixels, after we draw them there is no good way to move them (or remove them). The only way to update the canvas display is to clear it and redraw the scene. We may also have scrolled, which requires the background to be in a different position.

The updateViewport method is similar to DOMDisplay's scrollPlayerIntoView method. It checks whether the player is too close to the edge of the screen and moves the viewport when this is the case.

```
CanvasDisplay.prototype.updateViewport = function(state) {
  let view = this.viewport, margin = view.width / 3;
  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5));

  if (center.x < view.left + margin) {
    view.left = Math.max(center.x - margin, 0);
  } else if (center.x > view.left + view.width - margin) {
    view.left = Math.min(center.x + margin - view.width,
                         state.level.width - view.width);
  }
  if (center.y < view.top + margin) {
    view.top = Math.max(center.y - margin, 0);
  } else if (center.y > view.top + view.height - margin) {
    view.top = Math.min(center.y + margin - view.height,
                        state.level.height - view.height);
  }
};
```

The calls to Math.max and Math.min ensure that the viewport does not end up showing space outside of the level. Math.max(x, 0) makes sure the resulting number is not less than zero. Math.min similarly guarantees that a value stays below a given bound.

When clearing the display, we'll use a slightly different color depending on whether the game is won (brighter) or lost (darker).

```
CanvasDisplay.prototype.clearDisplay = function(status) {
  if (status == "won") {
    this.cx.fillStyle = "rgb(68, 191, 255)";
  } else if (status == "lost") {
    this.cx.fillStyle = "rgb(44, 136, 214)";
  } else {
    this.cx.fillStyle = "rgb(52, 166, 251)";
  }
  this.cx.fillRect(0, 0,
                   this.canvas.width, this.canvas.height);
};
```

To draw the background, we run through the tiles that are visible in the current viewport, using the same trick used in the touches method from the previous chapter.

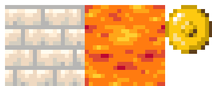
```
let otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function(level) {
  let {left, top, width, height} = this.viewport;
  let xStart = Math.floor(left);
  let xEnd = Math.ceil(left + width);
  let yStart = Math.floor(top);
  let yEnd = Math.ceil(top + height);

  for (let y = yStart; y < yEnd; y++) {
    for (let x = xStart; x < xEnd; x++) {
      let tile = level.rows[y][x];
      if (tile == "empty") continue;
      let screenX = (x - left) * scale;
      let screenY = (y - top) * scale;
      let tileX = tile == "lava" ? scale : 0;
```

```
      this.cx.drawImage(otherSprites,
                        tileX,         0, scale, scale,
                        screenX, screenY, scale, scale);
    }
  }
};
```

Tiles that are not empty are drawn with drawImage. The otherSprites image contains the pictures used for elements other than the player. It contains, from left to right, the wall tile, the lava tile, and the sprite for a coin.



Background tiles are 20 by 20 pixels since we will use the same scale that we used in DOMDisplay. Thus, the offset for lava tiles is 20 (the value of the scale binding), and the offset for walls is 0.

We don't bother waiting for the sprite image to load. Calling drawImage with an image that hasn't been loaded yet will simply do nothing. Thus, we might fail to draw the game properly for the first few frames, while the image is still loading, but that is not a serious problem. Since we keep updating the screen, the correct scene will appear as soon as the loading finishes.

The walking character shown earlier will be used to represent the player. The code that draws it needs to pick the right sprite and direction based on the player's current motion. The first eight sprites contain a walking animation. When the player is moving along a floor, we cycle through them based on the current time. We want to switch frames every 60 milliseconds, so the time is divided by 60 first. When the player is standing still, we draw the ninth sprite. During jumps, which are recognized by the fact that the vertical speed is not zero, we use the tenth, rightmost sprite.

Because the sprites are slightly wider than the player object—24 instead of 16 pixels to allow some space for feet and arms—the method has to adjust the x-coordinate and width by a given amount (playerXOverlap).

```
let playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
const playerXOverlap = 4;
```

```
CanvasDisplay.prototype.drawPlayer = function(player, x, y,
                                              width, height){
  width += playerXOverlap * 2;
  x -= playerXOverlap;
  if (player.speed.x != 0) {
    this.flipPlayer = player.speed.x < 0;
  }

  let tile = 8;
  if (player.speed.y != 0) {
    tile = 9;
  } else if (player.speed.x != 0) {
    tile = Math.floor(Date.now() / 60) % 8;
  }

  this.cx.save();
  if (this.flipPlayer) {
    flipHorizontally(this.cx, x + width / 2);
  }
  let tileX = tile * width;
  this.cx.drawImage(playerSprites, tileX, 0, width, height,
                                   x,     y, width, height);
  this.cx.restore();
};
```

The drawPlayer method is called by drawActors, which is responsible for drawing all the actors in the game.

```
CanvasDisplay.prototype.drawActors = function(actors) {
  for (let actor of actors) {
    let width = actor.size.x * scale;
    let height = actor.size.y * scale;
    let x = (actor.pos.x - this.viewport.left) * scale;
```

```
    let y = (actor.pos.y - this.viewport.top) * scale;
    if (actor.type == "player") {
      this.drawPlayer(actor, x, y, width, height);
    } else {
      let tileX = (actor.type == "coin" ? 2 : 1) * scale;
      this.cx.drawImage(otherSprites,
                        tileX, 0, width, height,
                        x,     y, width, height);
    }
  }
};
```

When drawing something that is not the player, we look at its type to find the offset of the correct sprite. The lava tile is found at offset 20, and the coin sprite is found at 40 (two times scale).

We have to subtract the viewport's position when computing the actor's position since (0,0) on our canvas corresponds to the top left of the viewport, not the top left of the level. We could also have used translate for this. Either way works.

This document plugs the new display into runGame:

```
<body>
  <script>
    runGame(GAME_LEVELS, CanvasDisplay);
  </script>
</body>
```

Choosing a graphics interface

So when you need to generate graphics in the browser, you can choose between plain HTML, SVG, and canvas. There is no single *best* approach that works in all situations. Each option has strengths and weaknesses.

Plain HTML has the advantage of being simple. It also integrates well with text. Both SVG and canvas allow you to draw text, but they won't help you position that text or wrap it when it takes up more than one line. In an HTML-based picture, it is much easier to include blocks of text.

SVG can be used to produce crisp graphics that look good at any zoom level. Unlike HTML, it is designed for drawing and is thus more suitable for that purpose.

Both SVG and HTML build up a data structure (the DOM) that represents your picture. This makes it possible to modify elements after they are drawn. If you need to repeatedly change a small part of a big picture in response to what the user is doing or as part of an animation, doing it in a canvas can be needlessly expensive. The DOM also allows us to register mouse event handlers on every element in the picture (even on shapes drawn with SVG). You can’t do that with canvas.

But canvas’s pixel-oriented approach can be an advantage when drawing a huge number of tiny elements. The fact that it does not build up a data structure but only repeatedly draws onto the same pixel surface gives canvas a lower cost per shape.

There are also effects, such as rendering a scene one pixel at a time (for example, using a ray tracer) or postprocessing an image with JavaScript (blurring or distorting it), that can be realistically handled only by a pixel-based approach.

In some cases, you may want to combine several of these techniques. For example, you might draw a graph with SVG or canvas but show textual information by positioning an HTML element on top of the picture.

For nondemanding applications, it really doesn’t matter much which interface you choose. The display we built for our game in this chapter could have been implemented using any of these three graphics technologies since it does not need to draw text, handle mouse interaction, or work with an extraordinarily large number of elements.