16

PROJECT: A PLATFORM GAME

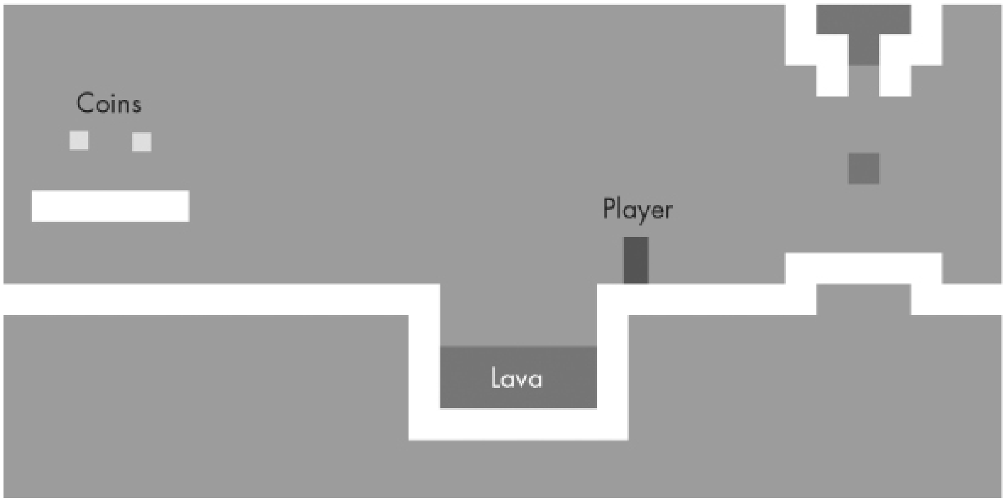
Much of my initial fascination with computers, like that of many nerdy kids, had to do with computer games. I was drawn into the tiny simulated worlds that I could manipulate and in which stories (sort of) unfolded—more, I suppose, because of the way I projected my imagination into them than because of the possibilities they actually offered.

I don't wish a career in game programming on anyone. As with the music industry, the discrepancy between the number of eager young people wanting to work in it and the actual demand for such people creates a rather unhealthy environment. But writing games for fun is amusing.

This chapter will walk through the implementation of a small platform game. Platform games (or "jump and run" games) are games that expect the player to move a figure through a world, which is usually two-dimensional and viewed from the side, while jumping over and onto things.

The Game

Our game will be roughly based on Dark Blue (*https://www.lessmilk.com/dark-blue/*) by Thomas Palef. I chose that game because it is both entertaining and minimalist and because it can be built without too much code. It looks like this:



The dark box represents the player, whose task is to collect the yellow boxes (coins) while avoiding the red stuff (lava). A level is completed when all coins have been collected.

The player can walk around with the left and right arrow keys and can jump with the up arrow. Jumping is this game character's specialty. It can reach several times its own height and can change direction in midair. This may not be entirely realistic, but it helps give the player the feeling of being in direct control of the on-screen avatar.

The game consists of a static background, laid out like a grid, with the moving elements overlaid on that background. Each field on the grid is either empty, solid, or lava. The moving elements are the player, coins, and certain pieces of lava. The positions of these elements are not constrained to the grid—their coordinates may be fractional, allowing smooth motion.

The Technology

We will use the browser DOM to display the game, and we'll read user input by handling key events.

The screen- and keyboard-related code is only a small part of the work we need to do to build this game. Since everything looks like colored boxes, drawing is uncomplicated: we create DOM elements and use styling to give them a background color, size, and position.

We can represent the background as a table, since it is an unchanging grid of squares. The free-moving elements can be overlaid using absolutely positioned elements.

In games and other programs that should animate graphics and respond to user input without noticeable delay, efficiency is important. Although the DOM was not originally designed for high-performance graphics, it is actually better at this than you would expect. You saw some animations in Chapter 14. On a modern machine, a simple game like this performs well, even if we don't worry about optimization very much.

In the next chapter, we will explore another browser technology, the `<canvas>` tag, which provides a more traditional way to draw graphics, working in terms of shapes and pixels rather than DOM elements.

Levels

We'll want a human-readable, human-editable way to specify levels. Since it is OK for everything to start out on a grid, we could use big strings in which each character represents an element—either a part of the background grid or a moving element.

The plan for a small level might look like this:

```
let simpleLevelPlan = `
......................
..#................#..
..#..............=.#..
..#.........o.o....#..
..#.@......#####...#..
..#####............#..
......#++++++++++++#..
......##############..
......................`;
```

Periods are empty space, hash (`#`) characters are walls, and plus signs are lava. The player's starting position is the at sign (`@`). Every O character is a coin, and the equal sign (`=`) at the top is a block of lava that moves back and forth horizontally.

We'll support two additional kinds of moving lava: the pipe character (`|`) creates vertically moving blobs, and `v` indicates *dripping* lava—vertically moving lava that doesn't bounce back and forth but only moves down, jumping back to its start position when it hits the floor.

A whole game consists of multiple levels that the player must complete. A level is completed when all coins have been collected. If the player touches lava, the current level is restored to its starting position, and the player may try again.

Reading a Level

The following class stores a level object. Its argument should be the string that defines the level.

```
class Level {
  constructor(plan) {
    let rows = plan.trim().split("\n").map(l => [...l])
    this.height = rows.length;
    this.width = rows[0].length;
    this.startActors = [];

    this.rows = rows.map((row, y) => {
      return row.map((ch, x) => {
        let type = levelChars[ch];
        if (typeof type != "string") {
          let pos = new Vec(x, y);
          this.startActors.push(type.create(pos, ch));
          type = "empty";
        }
        return type;
      });
    });
  }
}
```

The `trim` method is used to remove whitespace at the start and end of the plan string. This allows our example plan to start with a newline so that all lines are directly below each other. The remaining string is split on new-line characters, and each line is spread into an array, producing arrays of characters.

So `rows` holds an array of arrays of characters, the rows of the plan. We can derive the level's width and height from these. But we must still separate the moving elements from the background grid. We'll call moving elements *actors*. They'll be stored in an array of objects. The background will be an array of arrays of strings, holding field types such as "`empty`", "`wall`", or "`lava`".

To create these arrays, we map over the rows and then over their content. Remember that `map` passes the array index as a second argument to the mapping function, which tells us the x- and y-coordinates of a given character. Positions in the game will be stored as pairs of coordinates, with the upper left being (0, 0) and each background square being 1 unit high and wide.

To interpret the characters in the plan, the `Level` constructor uses the `levelChars` object, which, for each character used in the level descriptions, holds a string if it is a background type, and a class if it produces an actor. When `type` is an actor class, its static `create` method is used to create an object, which is added to `startActors`, and the mapping function returns "`empty`" for this background square.

The position of the actor is stored as a `Vec` object. This is a two-dimensional vector, an object with `x` and `y` properties, as seen in the exercises of Chapter 6.

As the game runs, actors will end up in different places or even disappear entirely (as coins do when collected). We'll use a `State` class to track the state of a running game.

```
class State {
  constructor(level, actors, status) {
    this.level = level;
    this.actors = actors;
    this.status = status;
  }
  static start(level) {
    return new State(level, level.startActors, "playing
  }

  get player() {
    return this.actors.find(a => a.type == "player");
  }
}
```

The `status` property will switch to "`lost`" or "`won`" when the game has ended.

This is again a persistent data structure—updating the game state creates a new state and leaves the old one intact.

Actors

Actor objects represent the current position and state of a given moving element (player, coin, or mobile lava) in our game. All actor objects conform to the same interface. They have `size` and `pos` properties holding the size and the coordinates of the upper-left corner of the rectangle representing this actor, and an `update` method.

This `update` method is used to compute their new state and position after a given time step. It simulates the thing the actor does—moving in response to the arrow keys for the player and bouncing back and forth for the lava— and returns a new, updated actor object.

A `type` property contains a string that identifies the type of the actor—"`player`", "`coin`", or "`lava`". This is useful when drawing the game—the look of the rectangle drawn for an actor is based on its type.

Actor classes have a static `create` method used by the `Level` constructor to create an actor from a character in the level plan. It is given the coordinates of the character and the character itself, which is necessary because the `Lava` class handles several different characters.

This is the `Vec` class that we'll use for our two-dimensional values, such as the position and size of actors.

```
class Vec {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  plus(other) {
    return new Vec(this.x + other.x, this.y + other.y);
  }
  times(factor) {
    return new Vec(this.x * factor, this.y * factor);
  }
}
```

The `times` method scales a vector by a given number. It will be useful when we need to multiply a speed vector by a time interval to get the distance traveled during that time.

The different types of actors get their own classes, since their behavior is very different. Let’s define these classes. We’ll get to their `update` methods later.

The player class has a `speed` property that stores its current speed to simulate momentum and gravity.

```
class Player {
  constructor(pos, speed) {
    this.pos = pos;
    this.speed = speed;
  }

  get type() { return "player"; }

  static create(pos) {
    return new Player(pos.plus(new Vec(0, -0.5)),
                      new Vec(0, 0));
  }
}

Player.prototype.size = new Vec(0.8, 1.5);
```

Because a player is one-and-a-half squares high, its initial position is set to be half a square above the position where the `@` character appeared. This way, its bottom aligns with the bottom of the square where it appeared.

The `size` property is the same for all instances of `Player`, so we store it on the prototype rather than on the instances themselves. We could have used a getter like `type`, but that would create and return a new `Vec` object every time the property is read, which would be wasteful. (Strings, being immutable, don’t have to be re-created every time they are evaluated.)

When constructing a `Lava` actor, we need to initialize the object differently depending on the character it is based on. Dynamic lava moves along at its current speed until it hits an obstacle. At that point, if it has a `reset` property, it will jump back to its start position (dripping). If it does not, it will invert its speed and continue in the other direction (bouncing).

The `create` method looks at the character that the `Level` constructor passes and creates the appropriate lava actor.

```
class Lava {
  constructor(pos, speed, reset) {
    this.pos = pos;
    this.speed = speed;
    this.reset = reset;
  }

  get type() { return "lava"; }

  static create(pos, ch) {
    if (ch == "=") {
      return new Lava(pos, new Vec(2, 0));
    } else if (ch == "|") {
      return new Lava(pos, new Vec(0, 2));
    } else if (ch == "v") {
      return new Lava(pos, new Vec(0, 3), pos);
    }
  }
}

Lava.prototype.size = new Vec(1, 1);
```

`Coin` actors are relatively simple. They mostly just sit in their place. But to liven up the game a little, they are given a “wobble,” a slight vertical back-and-forth motion. To track this, a coin object stores a base position as well as a `wobble` property that tracks the phase of the bouncing motion. Together, these determine the coin’s actual position (stored in the `pos` property).

```
class Coin {
  constructor(pos, basePos, wobble) {
    this.pos = pos;
    this.basePos = basePos;
    this.wobble = wobble;
  }

  get type() { return "coin"; }
  static create(pos) {
    let basePos = pos.plus(new Vec(0.2, 0.1));
    return new Coin(basePos, basePos,
                    Math.random() * Math.PI * 2);
  }
}

Coin.prototype.size = new Vec(0.6, 0.6);
```

In Chapter 14, we saw that `Math.sin` gives us the y-coordinate of a point on a circle. That coordinate goes back and forth in a smooth waveform as we move along the circle, which makes the sine function useful for modeling a wavy motion.

To avoid a situation where all coins move up and down synchronously, the starting phase of each coin is randomized. The period of `Math.sin`'s wave, the width of a wave it produces, is 2π. We multiply the value returned by `Math.random` by that number to give the coin a random starting position on the wave.

We can now define the `levelChars` object that maps plan characters to either background grid types or actor classes.

```
const levelChars = {
  ".": "empty", "#": "wall", "+": "lava",
  "@": Player, "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

That gives us all the parts needed to create a `Level` instance.

```
let simpleLevel = new Level(simpleLevelPlan);
console.log(`${simpleLevel.width} by ${simpleLevel.heig
// → 22 by 9
```

The task ahead is to display such levels on the screen and to model time and motion inside them.

Drawing

In the next chapter, we'll display the same game in a different way. To make that possible, we put the drawing logic behind an interface and pass it to the game as an argument. That way, we can use the same game program with different new display modules.

A game display object draws a given level and state. We pass its constructor to the game to allow it to be replaced. The display class we define in this chapter is called `DOMDisplay` because it uses DOM elements to show the level.

We'll be using a style sheet to set the actual colors and other fixed properties of the elements that make up the game. It would also be possible to directly assign to the elements' `style` property when we create them, but that would produce more verbose programs.

The following helper function provides a succinct way to create an element and give it some attributes and child nodes:

```
function elt(name, attrs, ...children) {
  let dom = document.createElement(name);
  for (let attr of Object.keys(attrs)) {
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

A display is created by giving it a parent element to which it should append itself and a level object.

```
class DOMDisplay {
  constructor(parent, level) {
    this.dom = elt("div", {class: "game"}, drawGrid(lev
    this.actorLayer = null;
    parent.appendChild(this.dom);
  }
```

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

The level's background grid, which never changes, is drawn once. Actors are redrawn every time the display is updated with a given state. The `actorLayer` property will be used to track the element that holds the actors so that they can be easily removed and replaced.

Our coordinates and sizes are tracked in grid units, where a size or distance of 1 means one grid block. When setting pixel sizes, we will have to scale these coordinates up—everything in the game would be ridiculously small at a single pixel per square. The `scale` constant gives the number of pixels that a single unit takes up on the screen.

```
const scale = 20;

function drawGrid(level) {
  return elt("table", {
    class: "background",
    style: `width: ${level.width * scale}px`
  }, ...level.rows.map(row =>
    elt("tr", {style: `height: ${scale}px`},
        ...row.map(type => elt("td", {class: type})))
  ));
}
```

The `<table>` element's form nicely corresponds to the structure of the `rows` property of the level—each row of the grid is turned into a table row (`<tr>` element). The strings in the grid are used as class names for the table cell (`<td>`) elements. The code uses the spread (triple dot) operator to pass arrays of child nodes to `elt` as separate arguments.

The following CSS makes the table look like the background we want:

```
.background    { background: rgb(52, 166, 251);
                 table-layout: fixed;
                 border-spacing: 0;              }
.background td { padding: 0;                     }
```

```
.lava          { background: rgb(255, 100, 100); }
.wall          { background: white;              }
```

Some of these (`table-layout`, `border-spacing`, and `padding`) are used to suppress unwanted default behavior. We don’t want the layout of the table to depend upon the contents of its cells, and we don’t want space between the table cells or padding inside them.

The `background` rule sets the background color. CSS allows colors to be specified both as words (`white`) or with a format such as `rgb(R, G, B)`, where the red, green, and blue components of the color are separated into three numbers from 0 to 255. In `rgb(52, 166, 251)`, the red component is 52, green is 166, and blue is 251. Since the blue component is the largest, the resulting color will be bluish. In the `.lava` rule, the first number (red) is the largest.

We draw each actor by creating a DOM element for it and setting that element’s position and size based on the actor’s properties. The values must be multiplied by `scale` to go from game units to pixels.

```
function drawActors(actors) {
  return elt("div", {}, ...actors.map(actor => {
    let rect = elt("div", {class: `actor ${actor.type}`
    rect.style.width = `${actor.size.x * scale}px`;
    rect.style.height = `${actor.size.y * scale}px`;
    rect.style.left = `${actor.pos.x * scale}px`;
    rect.style.top = `${actor.pos.y * scale}px`;
    return rect;
  }));
}
```

To give an element more than one class, we separate the class names by spaces. In the following CSS code, the `actor` class gives the actors their absolute position. Their type name is used as an extra class to give them a color. We don’t have to define the `lava` class again because we’re reusing the class for the lava grid squares we defined earlier.

```
.actor  { position: absolute;            }
.coin   { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64);   }
```

The `syncState` method is used to make the display show a given state. It first removes the old actor graphics, if any, and then redraws the actors in their new positions. It may be tempting to try to reuse the DOM elements for actors, but to make that work, we would need a lot of additional book-keeping to associate actors with DOM elements and to make sure we remove elements when their actors vanish. Since there will typically be only a handful of actors in the game, redrawing all of them is not expensive.

```
DOMDisplay.prototype.syncState = function(state) {
  if (this.actorLayer) this.actorLayer.remove();
  this.actorLayer = drawActors(state.actors);
  this.dom.appendChild(this.actorLayer);
  this.dom.className = `game ${state.status}`;
  this.scrollPlayerIntoView(state);
};
```

By adding the level's current status as a class name to the wrapper, we can style the player actor slightly differently when the game is won or lost by adding a CSS rule that takes effect only when the player has an ancestor element with a given class.

```
.lost .player {
  background: rgb(160, 64, 64);
}
.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

After touching lava, the player turns dark red, suggesting scorching. When the last coin has been collected, we add two blurred white shadows— one to the upper left and one to the upper right—to create a white halo effect.

We can't assume that the level always fits in the *viewport*, the element into which we draw the game. That is why we need the `scrollPlayerIntoView` call: it ensures that if the level is protruding outside the viewport, we scroll that viewport to make sure the player is near its center. The following CSS gives the game's wrapping DOM element a maximum size and ensures that anything that sticks out of the element's box is not visible. We also give it a relative position so that the actors inside it are positioned relative to the level's upper-left corner.

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

In the `scrollPlayerIntoView` method, we find the player's position and update the wrapping element's scroll position. We change the scroll position by manipulating that element's `scrollLeft` and `scrollTop` properties when the player is too close to the edge.

```
DOMDisplay.prototype.scrollPlayerIntoView = function(st
  let width = this.dom.clientWidth;
  let height = this.dom.clientHeight;
  let margin = width / 3;

  // The viewport
  let left = this.dom.scrollLeft, right = left + width;
  let top = this.dom.scrollTop, bottom = top + height;

  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5))
                         .times(scale);
  if (center.x < left + margin) {
    this.dom.scrollLeft = center.x - margin;
  } else if (center.x > right - margin) {
    this.dom.scrollLeft = center.x + margin - width;
  }
  if (center.y < top + margin) {
    this.dom.scrollTop = center.y - margin;
  } else if (center.y > bottom - margin) {
    this.dom.scrollTop = center.y + margin - height;
  }
};
```

The way the player's center is found shows how the methods on our `Vec` type allow computations with objects to be written in a relatively readable way. To find the actor's center, we add its position (its upper-left corner) and half its size. That is the center in level coordinates, but we need it in pixel coordinates, so we then multiply the resulting vector by our display scale.

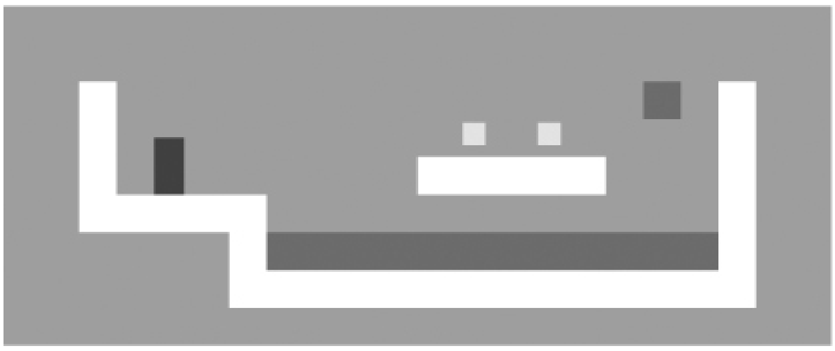
Next, a series of checks verifies that the player position isn't outside of the allowed range. Note that sometimes this will set nonsense scroll coordinates that are below zero or beyond the element's scrollable area. This is OK—the DOM will constrain them to acceptable values. Setting `scrollLeft` to `-10` will cause it to become `0`.

While it would have been slightly simpler to always try to scroll the player to the center of the viewport, this creates a rather jarring effect. As you are jumping, the view will constantly shift up and down. It's more pleasant to have a "neutral" area in the middle of the screen where you can move around without causing any scrolling.

We are now able to display our tiny level.

```
<link rel="stylesheet" href="css/game.css">

<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLev
  display.syncState(State.start(simpleLevel));
</script>
```



The `<link>` tag, when used with `rel="stylesheet"`, is a way to load a CSS file into a page. The file *game.css* contains the styles necessary for our game.

Motion and Collision

Now we're at the point where we can start adding motion. The basic approach taken by most games like this is to split time into small steps and, for each step, move the actors by a distance corresponding to their speed multiplied by the size of the time step. We'll measure time in seconds, so speeds are expressed in units per second.

Moving things is easy. The difficult part is dealing with the interactions between the elements. When the player hits a wall or floor, they should not simply move through it. The game must notice when a given motion causes an object to hit another object and respond accordingly. For walls, the motion must be stopped. When hitting a coin, that coin must be collected. When touching lava, the game should be lost.

Solving this for the general case is a major task. You can find libraries, usually called *physics engines*, that simulate interaction between physical objects in two or three dimensions. We'll take a more modest approach in this chapter, handling only collisions between rectangular objects and handling them in a rather simplistic way.

Before moving the player or a block of lava, we test whether the motion would take it inside of a wall. If it does, we simply cancel the motion altogether. The response to such a collision depends on the type of actor—the player will stop, whereas a lava block will bounce back.

This approach requires our time steps to be rather small, since it will cause motion to stop before the objects actually touch. If the time steps (and thus the motion steps) are too big, the player would end up hovering a noticeable distance above the ground. Another approach, arguably better but more complicated, would be to find the exact collision spot and move there. We will take the simple approach and hide its problems by ensuring the animation proceeds in small steps.

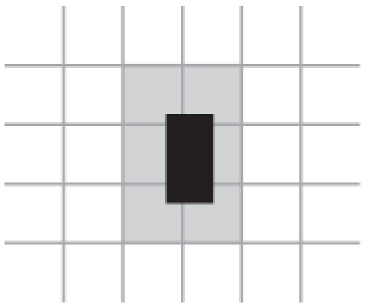
This method tells us whether a rectangle (specified by a position and a size) touches a grid element of the given type.

```
Level.prototype.touches = function(pos, size, type) {
  let xStart = Math.floor(pos.x);
```

```
    let xEnd = Math.ceil(pos.x + size.x);
    let yStart = Math.floor(pos.y);
    let yEnd = Math.ceil(pos.y + size.y);

    for (let y = yStart; y < yEnd; y++) {
      for (let x = xStart; x < xEnd; x++) {
        let isOutside = x < 0 || x >= this.width ||
                        y < 0 || y >= this.height;
        let here = isOutside ? "wall" : this.rows[y][x];
        if (here == type) return true;
      }
    }
    return false;
  };
```

The method computes the set of grid squares that the body overlaps with by using `Math.floor` and `Math.ceil` on its coordinates. Remember that grid squares are 1 by 1 units in size. By rounding the sides of a box up and down, we get the range of background squares that the box touches.



We loop over the block of grid squares found by rounding the coordinates and return `true` when a matching square is found. Squares outside of the level are always treated as "`wall`" to ensure that the player can't leave the world and that we won't accidentally try to read outside of the bounds of our `rows` array.

The state `update` method uses `touches` to figure out whether the player is touching lava.

```
  State.prototype.update = function(time, keys) {
    let actors = this.actors
      .map(actor => actor.update(time, this, keys));
    let newState = new State(this.level, actors, this.sta

    if (newState.status != "playing") return newState;

    let player = newState.player;
    if (this.level.touches(player.pos, player.size, "lava
      return new State(this.level, actors, "lost");
    }

    for (let actor of actors) {
      if (actor != player && overlap(actor, player)) {
        newState = actor.collide(newState);
      }
    }
    return newState;
  };
```

The method is passed a time step and a data structure that tells it which keys are being held down. The first thing it does is call the `update` method on all actors, producing an array of updated actors. The actors also get the time step, the keys, and the state so that they can base their update on those. Only the player will actually read keys, since that's the only actor that's controlled by the keyboard.

If the game is already over, no further processing has to be done (the game can't be won after being lost, or vice versa). Otherwise, the method tests whether the player is touching background lava. If so, the game is lost and we're done. Finally, if the game really is still going on, it sees whether any other actors overlap the player.

Overlap between actors is detected with the `overlap` function. It takes two actor objects and returns `true` when they touch—which is the case when they overlap both along the x-axis and along the y-axis.

```
  function overlap(actor1, actor2) {
    return actor1.pos.x + actor1.size.x > actor2.pos.x &&
           actor1.pos.x < actor2.pos.x + actor2.size.x &&
           actor1.pos.y + actor1.size.y > actor2.pos.y &&
           actor1.pos.y < actor2.pos.y + actor2.size.y;
  }
```

If any actor does overlap, its `collide` method gets a chance to update the state. Touching a lava actor sets the game status to "`lost`". Coins vanish when you touch them and set the status to "`won`" when they are the last coin of the level.

```
Lava.prototype.collide = function(state) {
  return new State(state.level, state.actors, "lost");
};

Coin.prototype.collide = function(state) {
  let filtered = state.actors.filter(a => a != this);
  let status = state.status;
  if (!filtered.some(a => a.type == "coin")) status = '
  return new State(state.level, filtered, status);
};
```

Actor Updates

Actor objects' `update` methods take as arguments the time step, the state object, and a `keys` object. The one for the `Lava` actor type ignores the `keys` object.

```
Lava.prototype.update = function(time, state) {
  let newPos = this.pos.plus(this.speed.times(time));
  if (!state.level.touches(newPos, this.size, "wall"))
    return new Lava(newPos, this.speed, this.reset);
  } else if (this.reset) {
    return new Lava(this.reset, this.speed, this.reset)
  } else {
    return new Lava(this.pos, this.speed.times(-1));
  }
};
```

This `update` method computes a new position by adding the product of the time step and the current speed to its old position. If no obstacle blocks that new position, it moves there. If there is an obstacle, the behavior depends on the type of the lava block—dripping lava has a `reset` position, to which it jumps back when it hits something. Bouncing lava inverts its speed by multiplying it by `-1` so that it starts moving in the opposite direction.

Coins use their `update` method to wobble. They ignore collisions with the grid, since they are simply wobbling around inside of their own square.

```
const wobbleSpeed = 8, wobbleDist = 0.07;
```

```
Coin.prototype.update = function(time) {
  let wobble = this.wobble + time * wobbleSpeed;
  let wobblePos = Math.sin(wobble) * wobbleDist;
  return new Coin(this.basePos.plus(new Vec(0, wobblePo
                  this.basePos, wobble);
};
```

The `wobble` property is incremented to track time and then used as an argument to `Math.sin` to find the new position on the wave. The coin's current position is then computed from its base position and an offset based on this wave.

That leaves the player itself. Player motion is handled separately per axis because hitting the floor should not prevent horizontal motion, and hitting a wall should not stop falling or jumping motion.

```
const playerXSpeed = 7;
const gravity = 30;
const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {
  let xSpeed = 0;
  if (keys.ArrowLeft) xSpeed -= playerXSpeed;
  if (keys.ArrowRight) xSpeed += playerXSpeed;
  let pos = this.pos;
  let movedX = pos.plus(new Vec(xSpeed * time, 0));
  if (!state.level.touches(movedX, this.size, "wall"))
    pos = movedX;
  }

  let ySpeed = this.speed.y + time * gravity;
  let movedY = pos.plus(new Vec(0, ySpeed * time));
  if (!state.level.touches(movedY, this.size, "wall"))
    pos = movedY;
  } else if (keys.ArrowUp && ySpeed > 0) {
    ySpeed = -jumpSpeed;
  } else {
    ySpeed = 0;
  }
  return new Player(pos, new Vec(xSpeed, ySpeed));
};
```

The horizontal motion is computed based on the state of the left and right arrow keys. When there's no wall blocking the new position created by this motion, it is used. Otherwise, the old position is kept.

Vertical motion works in a similar way but has to simulate jumping and gravity. The player's vertical speed (`ySpeed`) is first accelerated to account for gravity.

We check for walls again. If we don't hit any, the new position is used. If there *is* a wall, there are two possible outcomes. When the up arrow is pressed *and* we are moving down (meaning the thing we hit is below us), the speed is set to a relatively large, negative value. This causes the player to jump. If that is not the case, the player simply bumped into something, and the speed is set to zero.

The gravity strength, jumping speed, and other constants in the game were determined by simply trying out some numbers and seeing which ones felt right. You can try experimenting with them.

Tracking Keys

For a game like this, we do not want keys to take effect once per keypress. Rather, we want their effect (moving the player figure) to stay active as long as they are held.

We need to set up a key handler that stores the current state of the left, right, and up arrow keys. We will also want to call `preventDefault` for those keys so that they don't end up scrolling the page.

The following function, when given an array of key names, will return an object that tracks the current position of those keys. It registers event handlers for "`keydown`" and "`keyup`" events and, when the key code in the event is present in the set of codes that it is tracking, updates the object.

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys =
  trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);
```

The same handler function is used for both event types. It looks at the event object's `type` property to determine whether the key state should be updated to true ("`keydown`") or false ("`keyup`").

Running the Game

The `requestAnimationFrame` function, which we saw in Chapter 14, provides a good way to animate a game. But its interface is quite primitive—using it requires us to track the time at which our function was called the last time around and call `requestAnimationFrame` again after every frame.

Let's define a helper function that wraps all that in a convenient interface and allows us to simply call `runAnimation`, giving it a function that expects a time difference as an argument and draws a single frame. When the frame function returns the value `false`, the animation stops.

```
  function runAnimation(frameFunc) {
    let lastTime = null;
    function frame(time) {
      if (lastTime != null) {
        let timeStep = Math.min(time - lastTime, 100) / 1
        if (frameFunc(timeStep) === false) return;
      }
      lastTime = time;
      requestAnimationFrame(frame);
    }
    requestAnimationFrame(frame);
  }
```

I have set a maximum frame step of 100 milliseconds (one-tenth of a second). When the browser tab or window with our page is hidden, `request`

`AnimationFrame` calls will be suspended until the tab or window is shown again. In this case, the difference between `lastTime` and `time` will be the entire time in which the page was hidden. Advancing the game by that much in a single step would look silly and might cause weird side effects, such as the player falling through the floor.

The function also converts the time steps to seconds, which are an easier quantity to think about than milliseconds.

The `runLevel` function takes a `Level` object and a display constructor and returns a promise. It displays the level (in `document.body`) and lets the user play through it. When the level is finished (lost or won), `runLevel` waits one more second (to let the user see what happens) and then clears the display, stops the animation, and resolves the promise to the game's end status.

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.syncState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

A game is a sequence of levels. Whenever the player dies, the current level is restarted. When a level is completed, we move on to the next level. This can be expressed by the following function, which takes an array of level plans (strings) and a display constructor:

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level])
                                Display);
    if (status == "won") level++;
  }
  console.log("You've won!");
}
```

Because we made `runLevel` return a promise, `runGame` can be written using an `async` function, as shown in Chapter 11. It returns another promise, which resolves when the player finishes the game.

There is a set of level plans available in the `GAME_LEVELS` binding in this chapter's sandbox (*https://eloquentjavascript.net/code#16*). This page feeds them to `runGame`, starting an actual game.

```
<link rel="stylesheet" href="css/game.css">

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

Exercises

Game Over

It's traditional for platform games to have the player start with a limited number of *lives* and subtract one life each time they die. When the player is out of lives, the game restarts from the beginning.

Adjust `runGame` to implement lives. Have the player start with three. Output the current number of lives (using `console.log`) every time a level starts.

Pausing the Game

Make it possible to pause (suspend) and unpause the game by pressing ESC. You can do this by changing the `runLevel` function to set up a keyboard event handler that interrupts or resumes the animation whenever ESC is hit.

The `runAnimation` interface may not look like it is suitable for this at first glance, but it is if you rearrange the way `runLevel` calls it.

When you have that working, there’s something else you can try. The way we’ve been registering keyboard event handlers is somewhat problematic. The `arrowKeys` object is currently a global binding, and its event handlers are kept around even when no game is running. You could say they *leak* out of our system. Extend `trackKeys` to provide a way to unregister its handlers, then change `runLevel` to register its handlers when it starts and unregister them again when it is finished.

A Monster

It is traditional for platform games to have enemies that you can defeat by jumping on top of them. This exercise asks you to add such an actor type to the game.

We’ll call this actor a monster. Monsters move only horizontally. You can make them move in the direction of the player, bounce back and forth like horizontal lava, or have any other movement pattern you want. The class doesn’t have to handle falling, but it should make sure the monster doesn’t walk through walls.

When a monster touches the player, the effect depends on whether the player is jumping on top of them or not. You can approximate this by checking whether the player’s bottom is near the monster’s top. If this is the case, the monster disappears. If not, the game is lost.