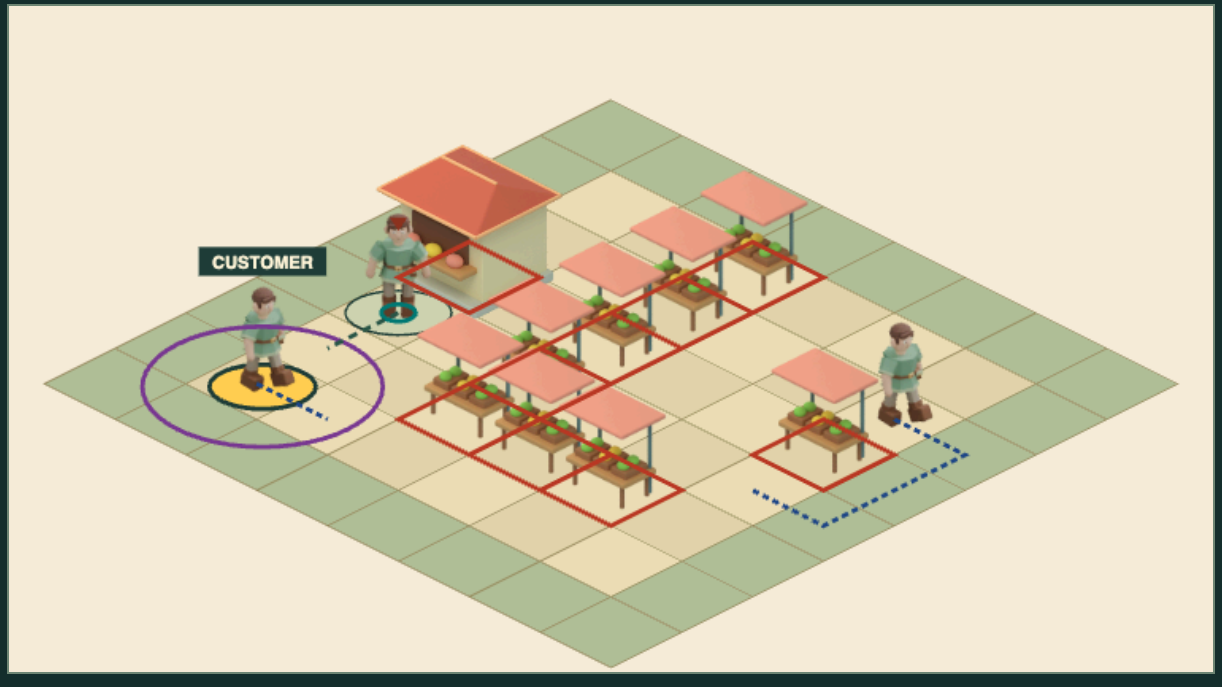
AN ILLUSTRATED PROGRAMMING BOOK

Build a Game Engine from Scratch in JavaScript

From one moving sprite to Rainy Market



Six labs · Three playable shifts · A worked fourth-shift capstone

GETTING STARTED

Run the reader edition

Extract the matching source ZIP. Open a terminal in the folder that contains `index.html` and `README.md`. Start a local server with one of these commands.

```
macOS / Linux
python3 -m http.server 8080 --bind 127.0.0.1

Windows (Python launcher)
py -m http.server 8080 --bind 127.0.0.1
```

Open **http://localhost:8080/**. Start a shift and try a delivery. Return to the title or reload the page when you're ready to read the code. The browser fetches local modules, images and level data over HTTP. It needs the terminal window to stay open. Stop the server with Ctrl+C when you're done.

Check your setup. If the browser presents a directory listing, find the directory containing the game's index file. If you see a missing image, inspect its request in the browser Network panel. A 404 means the server received the request but could not find the file at that path. If you see “address already in use,” switch 8080 to 8081 in both places.

Your learning path. The links below the game open six labs: movement, animation, depth, first delivery, replay and camera. Start with movement. In the delivery lab, edit the starter and compare it to the finished version. Keep a clean ZIP so you can recover any file after an experiment.

01 · TIME

Build along · start from a blank canvas

Create a folder called `workbench` beside `engine` inside your working copy. Leave the lab files in place as reference checkpoints. Create `index.html` in your new folder with this complete content and open `/workbench/` through your server.

```
<!doctype html>
<meta charset="utf-8">
<title>Movement workbench</title>
<canvas id="view" width="640" height="320"></canvas>
<script type="module" src="./boot.js"></script>
```

Create `boot.js` next to it. Start with a single observable result.

```
const view = document.querySelector('#view');
const ctx = view.getContext('2d');
let x = 40;
function draw() {
  ctx.fillStyle = '#e5e9d9';
  ctx.fillRect(0, 0, view.width, view.height);
  ctx.fillStyle = '#993f25';
  ctx.fillRect(x, 130, 16, 16);
}
draw();
```

You should see a rust-colored square 40 pixels from the left. Change x to 80 and reload. This first experiment separates a drawing problem from a scheduling problem. If the square is missing, the loop cannot be responsible: none exists yet. Check the URL and the console before adding code.

01 · TIME

Build along · implement a tiny loop

Keep your drawing code and remove the final `draw()` call. Add the following to the end of `boot.js`. The event handlers keep track of a held key. This small version intentionally handles one direction so you can read the entire trip from event to pixel.

```
let right = false;
addEventListener('keydown', e => {
  if (e.code === 'ArrowRight') { e.preventDefault(); right = true;
}
});
addEventListener('keyup', e => {
  if (e.code === 'ArrowRight') right = false;
});
addEventListener('blur', () => { right = false; });

function update(dt) {
  if (right) x = Math.min(view.width - 16, x + 120 * dt);
}
```

There is no motion yet. The new code knows what it should do for one small interval, but nothing calls it. If you run it in the console with a held key you can change x without seeing a new picture. Drawing and updating really are separate operations.

Predict. If x starts at 40 and you call update with the right key held for three ticks of 1/60 second, what should x contain? The answer is 46 unless the square reaches the right boundary. The next page supplies those ticks.

01 · TIME

Build along · schedule the steps

Append this scheduler to your workbench module. Hold the right key and watch the square travel.

```
let previous, budget = 0;
function frame(now) {
  if (previous !== undefined) {
    budget += Math.min((now - previous) / 1000, 0.1);
  }
  previous = now;
  while (budget >= 1 / 60) {
    update(1 / 60);
    budget -= 1 / 60;
  }
  draw();
  requestAnimationFrame(frame);
}
requestAnimationFrame(frame);
```

Extract the engine. Replace everything from `let previous` through the final call with the two lines below. Put the import at the top if you prefer.

```
import { GameLoop } from '../engine/loop.js';
new GameLoop(update, draw).start();
```

Try the same key again. You just moved timing into a reusable module while keeping the input, motion rule and drawing in your program. Compare with the first lab, which also supports diagonals, pause and a tick counter.

01 · TIME

What happens when you press an arrow?

Start the movement lab. Press and hold the right arrow, release it, then repeat the experiment with two arrows together. The browser receives key events, but a held key can influence many updates. Treating movement as a sequence of keyboard repeat events would make the game depend on a setting of the user's operating system.

Instead, keep the current input intent. At each simulation step read that intent and update the position. When the browser is ready for another frame, draw the latest position. It is useful to name the three different facts involved: “right is held,” “the marker is at x,” and “a pixel at this screen coordinate has this color.” Each belongs to a different part of the program.

Suppose the speed is 120 units per second. With an update of 1/60 second, one step advances two units. Holding the key for 15 steps advances 30 units. If the browser draws twice during that interval, it shows two snapshots. If it draws 15 times, the final position is still 30 units farther along.

> **Try it.** Find the movement speed in the first lab. Double it, refresh, and hold the same key. Explain which quantity changed. Then find the place that schedules the next browser frame. Explain why changing speed required no edit there.

Read alongside: `labs/01-movement.js` and `engine/loop.js`. In the latter, find where the constructor receives the update and render functions. That pair of arguments lets the same loop drive a marker or a delivery game.

01 · TIME

Read the accumulator

The browser supplies a timestamp each time it runs the frame callback. Two consecutive timestamps yield an elapsed interval in seconds. In `GameLoop.frame`, the accumulator keeps elapsed time the simulation has yet to consume. The following is the scheduling decision in the edition's loop.

```
while (this.accumulator >= 1 / 60) {
  this.update(1 / 60);
  this.accumulator -= 1 / 60;
  this.tick++;
}
```

Imagine an accumulator of 0.040 seconds. Two updates consume about 0.03333 seconds. The remaining 0.00667 stays available for the next frame. Rendering once after the loop avoids drawing intermediate states that would be overwritten before the browser displays them. This also explains why a fast display can show the same position in two consecutive frames.

The loop caps one elapsed interval at 0.1 seconds. Without that choice, returning to a tab after a long interruption could trigger a burst of updates. With the cap the simulation deliberately gives up some elapsed wall time. A game clock advanced by simulation steps therefore measures *played time*. A real-world auction deadline would call for a different clock.

> **Quick check.** A frame contributes 0.010 seconds while the accumulator is empty. How many updates run? Zero. The render still runs. Another 0.010 seconds is enough for one update, with about 0.00333 seconds left over.

Debug order. First inspect the input intent, then the position before and after update, then the draw coordinates. Finding the first wrong value is more informative than changing all three.

01 · TIME

A movement experiment

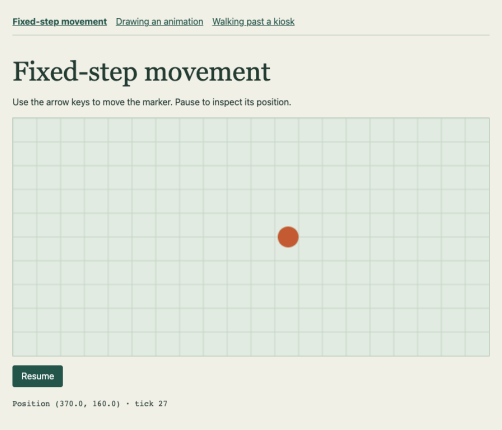


Figure 1. The movement checkpoint. Run it yourself and compare a straight move with a diagonal move.

Two perpendicular input components of magnitude one form a vector of length √2. If you multiply each component by speed, a diagonal covers about 41 percent more ground than a straight move. `normalize` divides each component by the vector's length and returns a zero vector when the input length is zero.

```
input (1, 1) → length 1.414 → direction (0.707, 0.707)
direction × 120 units/second × 1/60 second
            → step (1.414, 1.414) → distance 2 units
```

Exercise. Log the distance moved over one simulated second in two unobstructed directions. A wall or edge shortens the measured distance, so choose your starting position before interpreting the result. Explain why both distances should agree. Use your edition's actual lab speed when you compare numbers.