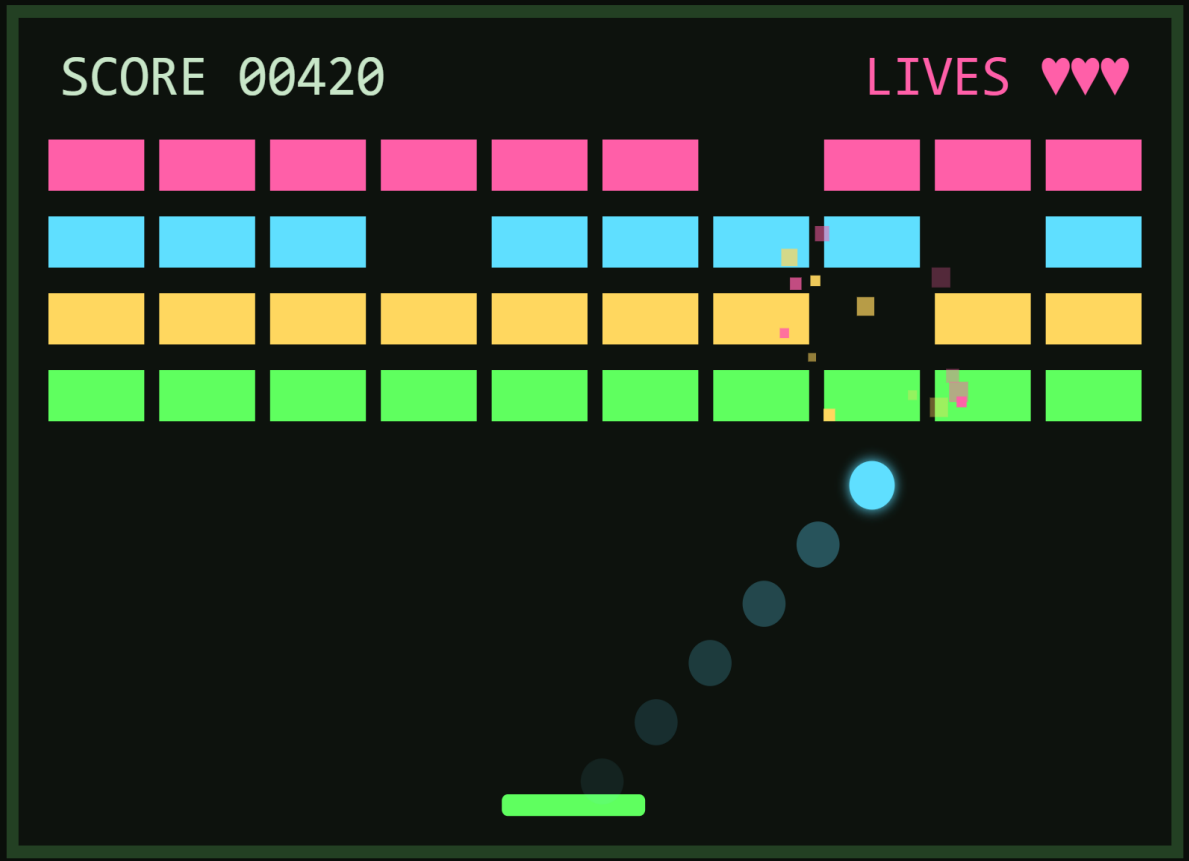
$ open game.html

CANVAS ACADEMY

BUILD 2D GAMES WITH THE HTML5 CANVAS

NO ENGINE · NO FRAMEWORK · JUST A BROWSER



Canvas Academy

Build real 2D games with the HTML5 Canvas — no engine, no framework, no build step

Ali Reza Hassan Sagor

This book is available at https://leanpub.com/canvas-academy

This version was published on 2026-08-17

The author generated this text in part with AI. Upon generating draft language, the author reviewed, edited, and revised the language to their own liking and takes ultimate responsibility for the content of this publication.

Contents

Preface

You are going to learn 2D game development the way it actually sticks: by shipping one small thing per chapter until, somewhere around the middle of this book, you look up and realize you've built a complete game.

The tool is the HTML5 canvas — a drawing surface that lives in every browser you already have. No engine, no framework, no build step, no install. A text editor and a browser are the entire toolchain. That's not a limitation; it's the point. Engines are excellent at hiding the ideas this book exists to teach: the game loop, delta time, collision, state machines, the update/render split. Learn them once at this level and every engine you touch afterwards — Unity, Godot, Phaser — becomes "oh, it's that, with buttons."

What you'll build

The book is in four parts:

- **Part I — Fundamentals.** Drawing, motion, time, and input. Eight short chapters, each one runnable.
- **Part II — Building Breakout.** A paddle, a ball, bricks, collision, game states, a score. By the end of this part you have a real, playable game.
- **Part III — Polish.** The difference between "technically a game" and "feels good": particles, screen shake, easing, sound, sprites.
- **Part IV — Capstone.** You finish the full Breakout, then take the training wheels off in a sandbox with starter prompts for Pong, Snake, and Asteroids.

How to read this book

Do not read this book. *Type* it. Nearly every chapter has a code listing meant to run in the workbench file you'll create in the setup chapter, and ends with a **Try it** challenge — a small modification that proves you understood the chapter

rather than just nodded at it. Worked solutions are in Appendix A, but the challenges are where the learning happens; wrestle first, peek second.

The chapters build on each other deliberately. Skipping ahead works less well than it does in most programming books, because each listing reuses the muscle memory of the last. Pace-wise: a chapter is a comfortable evening, and the book is a month of evenings — it survives bingeing just fine, but it's built for the steady route.

What you need

- Any modern browser (Chrome, Firefox, Edge, Safari).
- Any text editor (VS Code if you want a recommendation, Notepad if you want to prove a point).
- Basic JavaScript: variables, functions, loops, objects. You do not need to know anything about graphics, games, or the DOM.

Spotted a typo, a bug in a listing, or an explanation that fought you? Email **reza054@gmail.com** — errata reports shape the next revision, and book updates are free.

That's it. Let's set up your workbench.

Your Workbench

Every chapter in this book runs inside one HTML file that you are about to create. Build it once, keep it forever — it is the blank cartridge every lesson gets written onto.

Create a folder anywhere you like, and inside it a file called `game.html` with exactly this content:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Canvas Workbench</title>
<style>
  body { margin: 0; height: 100vh; display: grid; place-items: center;
  ↪  background: #182218; }
  canvas { background: #0a0e0a; outline: 1px solid #2a3a2a; }
</style>
</head>
<body>
<canvas id="game" width="480" height="320"></canvas>
<script>
const canvas = document.getElementById('game');
const ctx = canvas.getContext('2d');

// --- Keyboard: keys['ArrowLeft'] is true while that key is held.
//     (You'll start using this in Chapter 7.)
const keys = {};
addEventListener('keydown', e => {
  keys[e.key] = true;
  if (['ArrowUp','ArrowDown','ArrowLeft','ArrowRight',' '].includes(e.key))
  ↪  e.preventDefault();
});
addEventListener('keyup', e => { keys[e.key] = false; });

// --- Mouse: canvas-relative x/y plus a "button held" flag.
//     (You'll start using this in Chapter 8.)
const mouse = { x: 0, y: 0, down: false };
canvas.addEventListener('mousemove', e => {
  const r = canvas.getBoundingClientRect();
  mouse.x = (e.clientX - r.left) * (canvas.width / r.width);
  mouse.y = (e.clientY - r.top)  * (canvas.height / r.height);
});
```

```
canvas.addEventListener('mousedown', () => { mouse.down = true; });
addEventListener('mouseup', () => { mouse.down = false; });

// --- Touch: fingers act like the mouse, so games work on phones too.
function touchXY(e) {
  const t = e.touches[0], r = canvas.getBoundingClientRect();
  mouse.x = (t.clientX - r.left) * (canvas.width / r.width);
  mouse.y = (t.clientY - r.top)  * (canvas.height / r.height);
}
canvas.addEventListener('touchstart', e => { mouse.down = true; touchXY(e);
↪  e.preventDefault(); }, { passive: false });
canvas.addEventListener('touchmove',  e => { touchXY(e); e.preventDefault(); },
↪  { passive: false });
addEventListener('touchend', () => { mouse.down = false; });

// ======= LESSON CODE — replace everything BELOW this line =======

ctx.fillStyle = '#5fff5f';
ctx.fillRect(190, 110, 100, 100);
</script>
</body>
</html>
```

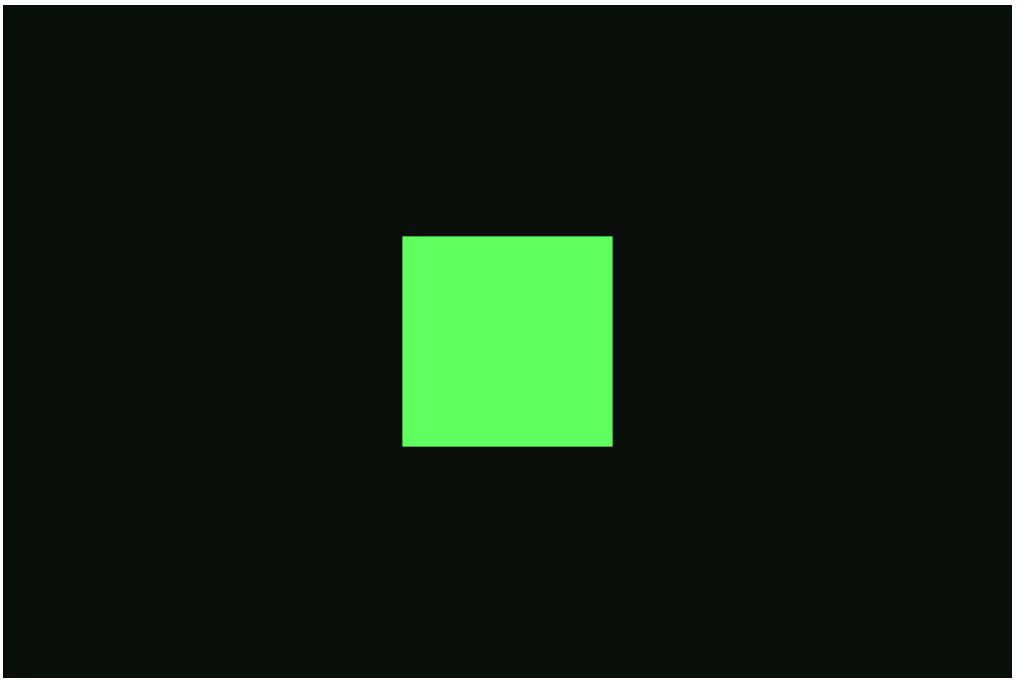


Figure 1. The workbench working: one green square on the dark canvas.

Double-click the file (or drag it into a browser window). If you see a green square on a dark rectangle, your workbench works. That green square is also your first canvas drawing – congratulations, you're one chapter ahead already.

The routine

From here on, every chapter works the same way:

1. Open `game.html` in your editor.
2. Delete everything below the `LESSON CODE` line and paste in the chapter's listing.
3. Save, switch to the browser, refresh.

Everything *above* the line stays put for the whole book. Edit, save, refresh – that three-beat loop is your entire development environment.

What the boilerplate actually does

You'll understand every line of this file properly as the book goes on, but here's the tour so nothing feels like magic:

- The `<canvas id="game" width="480" height="320">` element is the drawing surface. The `width` and `height` attributes set its resolution in pixels – every listing in this book assumes 480×320.
- `canvas.getContext('2d')` hands you `ctx`, the 2D rendering context. Think of the canvas as the paper and `ctx` as the hand holding the brush. Almost every line of game code you write will start with `ctx.`.
- The **keyboard block** keeps a `keys` object up to date: while you hold the left arrow, `keys['ArrowLeft']` is `true`; release it and it flips back to `false`. The `preventDefault` line stops the arrow keys and space bar from scrolling the page while you play. Chapter 7 explains why this "polling" style beats reacting to individual key events.
- The **mouse block** keeps a `mouse` object up to date with canvas-relative coordinates – `(0, 0)` is the canvas's top-left corner, not the page's – plus a `down` flag. Chapter 8 explains the coordinate conversion.
- The **touch block** routes taps and finger-drags into that same `mouse` object, so anything you build with the mouse works on a phone or tablet unmodified. Its `preventDefault` calls stop a drag from scrolling the page, the same favor the keyboard block does for arrow keys.

If something breaks

The canvas fails silently: a typo usually gets you a blank rectangle, not an error dialog. Press **F12** (or right-click □ Inspect) and click the **Console** tab — every JavaScript error appears there with a line number. Keep that panel open while you work through this book. Professional game developers do exactly the same thing, just with more monitors.

Part I — Fundamentals

Eight short chapters: drawing, style, motion, time, and input. Each one is a single idea with a single runnable listing. By the end of this part you'll have every tool needed to build an actual game — which is exactly what Part II does with them.

Chapter 1: Hello, Canvas

The canvas is a grid of pixels, and `ctx` is your brush. That one sentence is most of this chapter; the rest is the two details that trip everyone up on day one.

Detail one: the origin is the top-left corner. Coordinates start at `(0, 0)` in the top-left of the canvas. The x-axis grows to the right, which is fine, and the y-axis grows *downward*, which offends everyone who has ever taken a math class. There's a historical reason (screens draw text top-to-bottom, and graphics memory inherited the layout), but no amount of history makes it intuitive. You will write `y - 10` meaning "move up" for the rest of your game-dev life. Yes, down is positive. It's confusing forever; the only cure is repetition.

Detail two: `ctx` is a state machine. You don't tell the canvas "draw a green rectangle." You tell it "the fill color is now green" (`fillStyle`), and then "fill a rectangle" (`fillRect`). The color sticks until you change it — set it once, and every fill afterwards is green. Most beginner bugs of the form "why is everything pink?" are just a `fillStyle` set three drawings ago and never changed back.

Replace your lesson code with this:

```
// fillStyle = the color used for fill operations
ctx.fillStyle = '#5fff5f';
ctx.fillRect(50, 50, 100, 100);  // x, y, width, height

ctx.fillStyle = '#ff5fa8';
ctx.fillRect(200, 80, 60, 140);
```

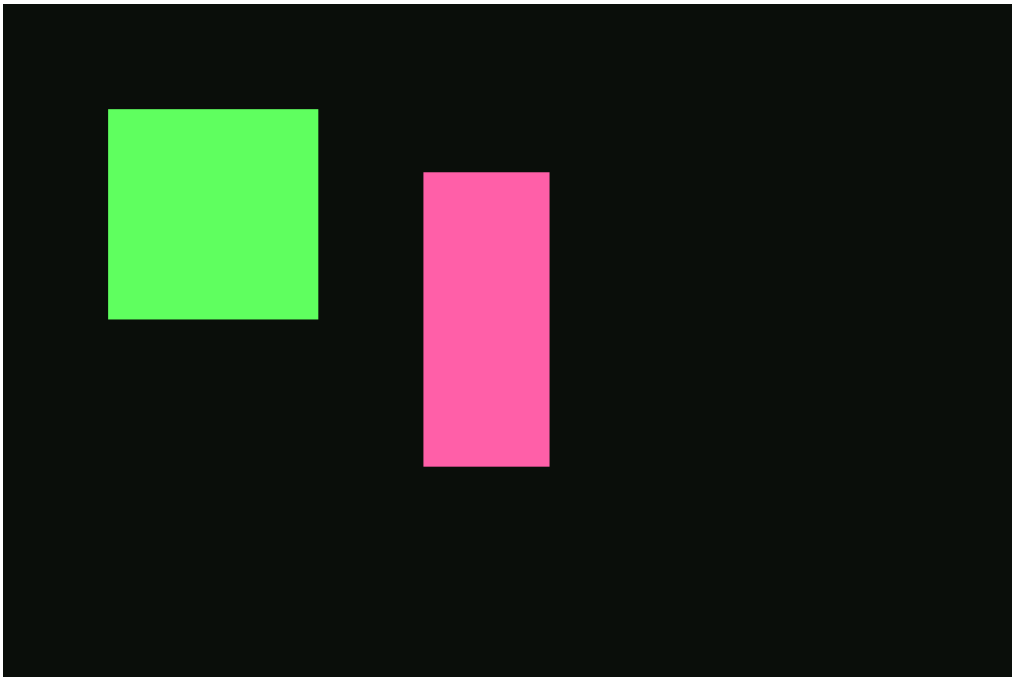


Figure 2. Two fillRect calls: the green square at (50, 50), the pink rectangle at (200, 80).

`fillRect(x, y, width, height)` reads as: starting `x` pixels from the left edge and `y` pixels from the *top* edge, fill a box `width` wide and `height` tall. The first call paints a 100×100 green square whose top-left corner sits at `(50, 50)`. The second paints a tall pink rectangle — 60 wide, 140 tall — starting at `(200, 80)`.

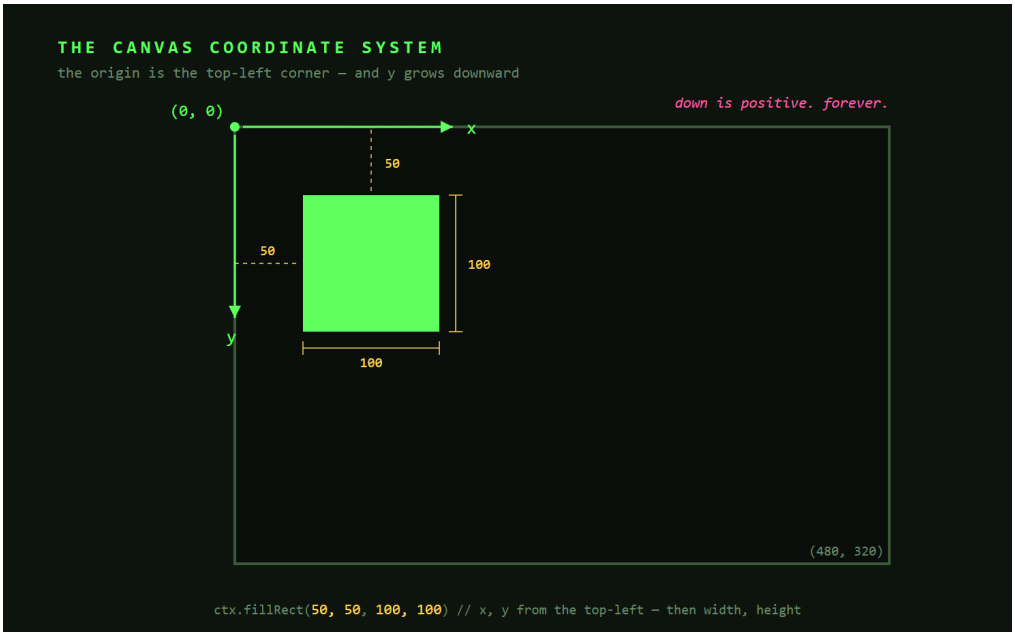


Figure 3. Canvas coordinates: the origin sits top-left, y grows downward, and fillRect measures from there.

Refresh and look at the two rectangles for a second. Everything else in this book — the ball, the bricks, the particles, the explosions — is going to be built out of calls exactly this simple, issued sixty times a second. But that's Chapter 5. First, shapes that aren't rectangles.

Try it

Add a third rectangle in cyan (`#5fdfff`) anywhere on the canvas. Before you refresh, predict where it will appear — the prediction is the actual exercise, because it forces you to read coordinates the canvas's way, top-left down.

A worked solution is in Appendix A.

Chapter 2: Shapes & Paths

Rectangles get a one-line shortcut because games and user interfaces are mostly rectangles. Everything else — circles, triangles, stars, ship hulls — goes through a **path**: you describe an outline point by point, then tell the canvas what to do with it.

The pattern is always the same three beats:

1. `beginPath()` — start a fresh outline.
2. Describe the shape — `moveTo`, `lineTo`, `arc`.
3. `fill()` it solid, or `stroke()` just the outline.

Replace your lesson code with this:

```
// A circle: arc(cx, cy, radius, startAngle, endAngle)
ctx.beginPath();
ctx.arc(120, 160, 50, 0, Math.PI * 2);
ctx.fillStyle = '#5fff5f';
ctx.fill();

// A triangle from line segments
ctx.beginPath();
ctx.moveTo(320, 80);
ctx.lineTo(400, 220);
ctx.lineTo(240, 220);
ctx.closePath();
ctx.strokeStyle = '#ff5fa8';
ctx.lineWidth = 4;
ctx.stroke();
```

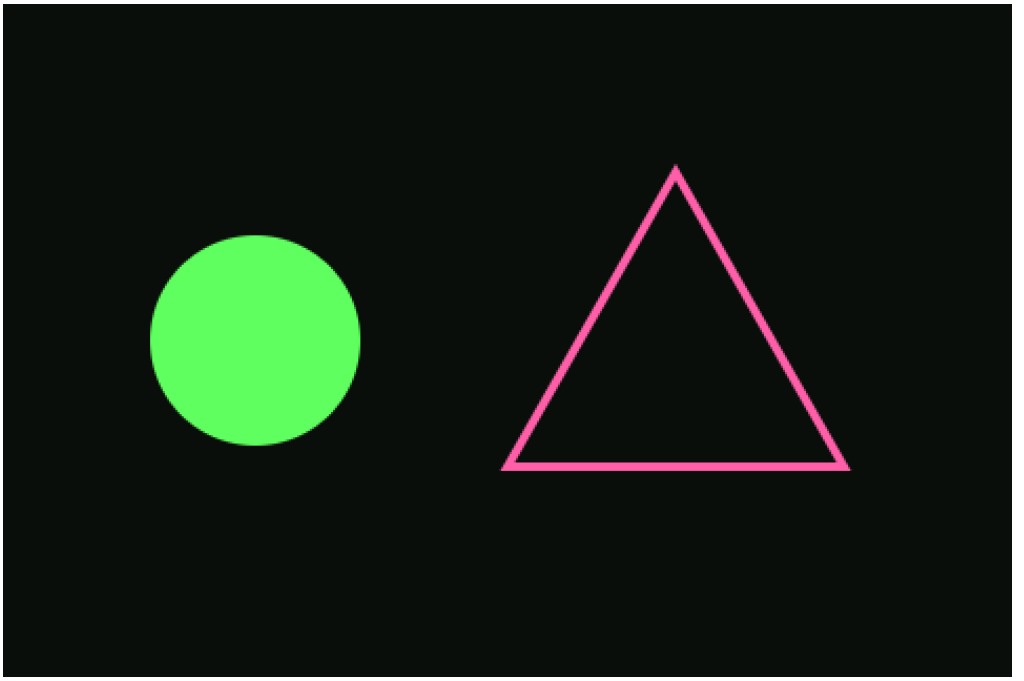


Figure 4. A filled green circle and a stroked pink triangle.

The circle first. `arc(cx, cy, radius, startAngle, endAngle)` sweeps an arc around center `(cx, cy)`. Angles are in **radians**, not degrees: a full circle is `Math.PI * 2`, a half circle is `Math.PI`, a quarter is `Math.PI / 2`. If radians are new to you, the only conversion worth memorizing is π = *half a turn*. Sweeping from `0` to `Math.PI * 2` gives the full circle, and `fill()` paints it solid green.

The triangle shows the manual style. `moveTo` lifts the pen and puts it down at the starting point; each `lineTo` drags it to the next corner. `closePath()` connects the last point back to the first — without it, your triangle is a "V" with a missing bottom edge, which becomes very visible when stroking. Then instead of filling, we `stroke()`: draw just the outline, using `strokeStyle` for color and `lineWidth` for thickness. Fill and stroke are independent — a shape can have both, which is how you get the classic "colored shape with a darker border" look.

One warning that will save you a bad afternoon: **forgetting `beginPath()` is the classic path bug.** Paths accumulate. If you skip `beginPath()`, your new arc gets welded onto the previous shape, and `fill()` repaints *everything described so far* — shapes smear together and old ones mysteriously change color. If your shapes ever look "melted together," the missing `beginPath()` is almost always why.

Try it

Make a half-circle. An arc from `0` to `Math.PI` is a semicircle — but *which* half you get is the interesting part. Remember which way the y-axis points, make a prediction, then check it.

A worked solution is in Appendix A.

Chapter 3: Colors & Style

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 4: Transforms

How do you draw a rotated rectangle? The honest answer: you don't. You rotate the *world*, draw a normal rectangle, and rotate the world back. That mental flip — move the paper, not the pen — is the whole chapter.

Three tools move the paper:

- `translate(x, y)` — slide the origin. After `translate(240, 160)`, the point `(0, 0)` *is* the center of a 480×320 canvas, and everything you draw is measured from there.
- `rotate(angle)` — spin the coordinate system around the current origin, in radians.
- `scale(sx, sy)` — stretch it. `scale(2, 2)` doubles everything; `scale(1, 0.5)` squashes vertically.

And one pair of tools puts the paper back: `save()` snapshots the current coordinate system onto a stack, `restore()` pops back to the snapshot. Transforms are state — they stack up and they *leak* into every draw that follows, exactly like `fillStyle` and `globalAlpha` did. The pattern that keeps you sane, worth typing until it's reflex:

```
ctx.save();      // remember the normal world
// translate / rotate / scale
// draw
ctx.restore();   // put the world back
```

Replace your lesson code with this:

```
ctx.fillStyle = '#5fff5f';

// Draw 6 rectangles in a circular spoke pattern
for (let i = 0; i < 6; i++) {
  ctx.save();
  ctx.translate(240, 160);              // move origin to center
  ctx.rotate((i / 6) * Math.PI * 2);    // rotate by an angle
  ctx.fillRect(40, -8, 80, 16);         // draw at the new origin
  ctx.restore();                         // undo translate + rotate
}
```

Figure 5. Six rectangles fanned into a spoke wheel by translate and rotate.

Walk through one iteration. `translate(240, 160)` puts the origin at the canvas center. `rotate((i / 6) * Math.PI * 2)` turns the world by i-sixths of a full circle — 0°, 60°, 120°, and so on. Then `fillRect(40, -8, 80, 16)` draws a plain, boring, axis-aligned rectangle: 40 units out from the origin along the x-axis, 8 units up so it's vertically centered. No trigonometry, no rotated-corner math — the rectangle thinks it's lying flat; the world underneath it is what turned. `restore()` resets everything for the next spoke.

Order matters, and it's worth burning in now: **translate then rotate** orbits shapes around a point (what we did here); **rotate then translate** spins the whole

world first and then slides along the *tilted* axes — a completely different result from the same two calls. When a transform does something baffling, the order is the first thing to check.

This exact pattern — save, translate to a thing's position, rotate to its heading, draw it as if unrotated, restore — is how every rotated sprite in every 2D game gets drawn. The Asteroids ship you might build in the book's final part is this loop with a triangle in it.

Try it

Change the `6` to `12` — both of them — and refresh. Then add `ctx.scale(1, 0.5)` right after the rotate and try to explain the squashed result to yourself before reading the explanation in Appendix A: is it squashing each spoke, or the whole wheel?

Chapter 5: The Game Loop

Everything so far has been still-life painting: run once, draw, done. Games are movies — sixty paintings a second, each slightly different from the last. This chapter is where your canvas starts moving, and it introduces the single most important piece of structure in this entire book.

The browser gives you exactly the tool for it: `requestAnimationFrame(fn)` says "call `fn` right before you next repaint the screen." That's typically 60 times per second on most monitors. Call it again *inside* `fn` and you've built a loop that runs once per frame, forever, perfectly synced to the display.

Every game you have ever played is, at its heart, this loop:

1. **Clear** — erase last frame.
2. **Update** — change the world a little.
3. **Draw** — paint the new world.
4. **Repeat.**

Replace your lesson code with this:

```
let x = 0;

function loop() {
  // 1. CLEAR — paint over last frame
  ctx.fillStyle = '#0a0e0a';
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  // 2. UPDATE — change state
  x += 2;
  if (x > canvas.width) x = -40;

  // 3. DRAW — render new state
  ctx.fillStyle = '#5fff5f';
  ctx.fillRect(x, 140, 40, 40);

  // 4. REPEAT
  requestAnimationFrame(loop);
}
loop();
```

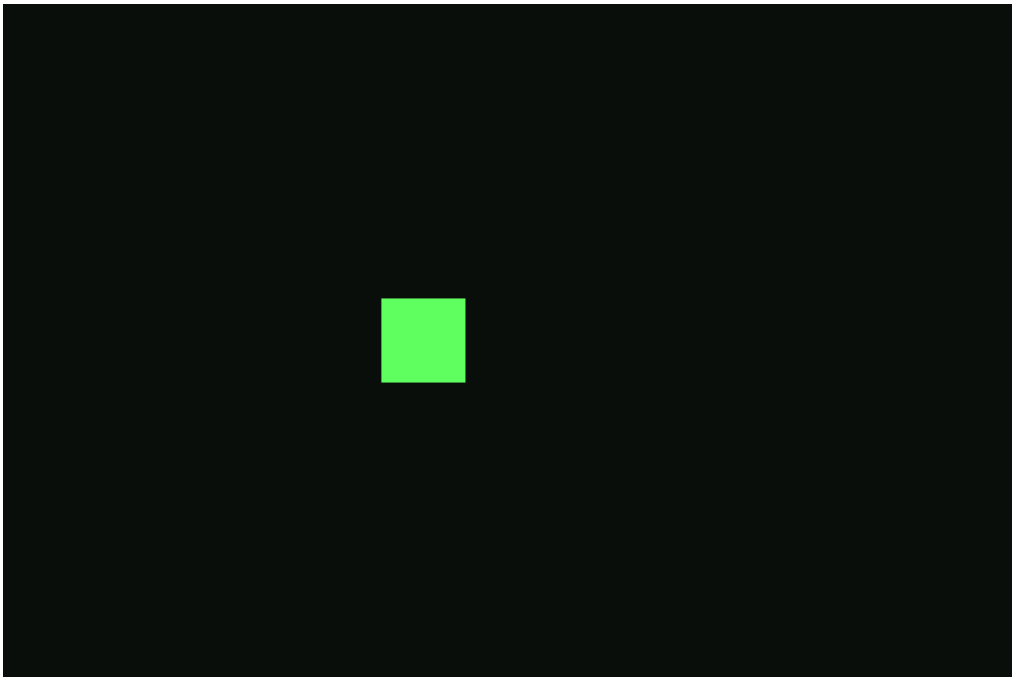


Figure 6. The green box mid-glide — one frame of the flip-book.

A green box glides across the screen and wraps around. Nothing is actually *moving* — that's worth sitting with for a moment. Sixty times a second, the code erases everything and paints a box two pixels further right than last time. Motion on a screen is a flip-book, and you just drew your first one. (And if you're looking at a blank, unmoving rectangle instead: this is the moment the setup chapter's F12 advice stops being optional. A typo in a loop fails sixty times a second, silently — the Console names the line.)

The **clear** step deserves respect. The canvas doesn't erase itself between frames — pixels stay until painted over. Comment out the clearing `fillRect` and refresh: the box becomes a smeared green streak, every frame stacked on the last. (Ugly here — but hold that thought, because Chapter 8 turns deliberate partial clearing into a genuinely pretty trick.)

The **update** step here is two lines: slide right, and if the box has left the right edge, teleport it to just off the left edge (`-40`, so it re-enters smoothly rather than popping into view). Notice that update touches only *variables* — no drawing. Draw touches only the canvas — no logic. That separation looks like fussiness in a ten-line program. By Part II it's how you'll find bugs in seconds instead of hours, and Chapter 16 is entirely about why.

One mechanical note: the state (`x`) lives *outside* the loop function. Anything declared inside `loop` would be recreated from scratch sixty times a second and forget everything. State persists; the loop visits it.

Try it

Make the box **bounce** off the right edge instead of teleporting. You'll need the box to remember which direction it's traveling — that's a hint that one variable is no longer enough.

A worked solution is in Appendix A.

Chapter 6: Delta Time

Chapter 5's box moves 2 pixels per frame. Quick question: how fast is that?

At 60 frames per second: 120 pixels per second. On a 144Hz gaming monitor, where `requestAnimationFrame` runs 144 times a second: 288 pixels per second — the same code, well over twice as fast. On a laptop throttling to 30fps: 60 pixels per second, molasses. "Pixels per frame" isn't a speed; it's a speed *multiplied by whatever screen the player happens to own*. Players with expensive monitors get a harder game. This is a real bug that shipped in real commercial games, and the fix is one idea and three lines.

The idea: measure how much time actually passed since the last frame — the **delta time**, `dt` — and move everything by `velocity × dt`. Speeds become real units (pixels per *second*), and the frame rate stops mattering: a 144Hz machine takes many small steps, a 30fps machine takes fewer big ones, and the box crosses the screen in the same wall-clock time on both.

Replace your lesson code with this:

```
const SPEED = 120;       // pixels per second (a real unit!)
let x = 0;
let last = performance.now();

function loop(now) {
  const dt = (now - last) / 1000;   // seconds since last frame
  last = now;

  ctx.fillStyle = '#0a0e0a';
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  x += SPEED * dt;                  // velocity × time = distance
  if (x > canvas.width) x = -40;

  ctx.fillStyle = '#5fff5f';
  ctx.fillRect(x, 140, 40, 40);

  requestAnimationFrame(loop);
}
requestAnimationFrame(loop);
```

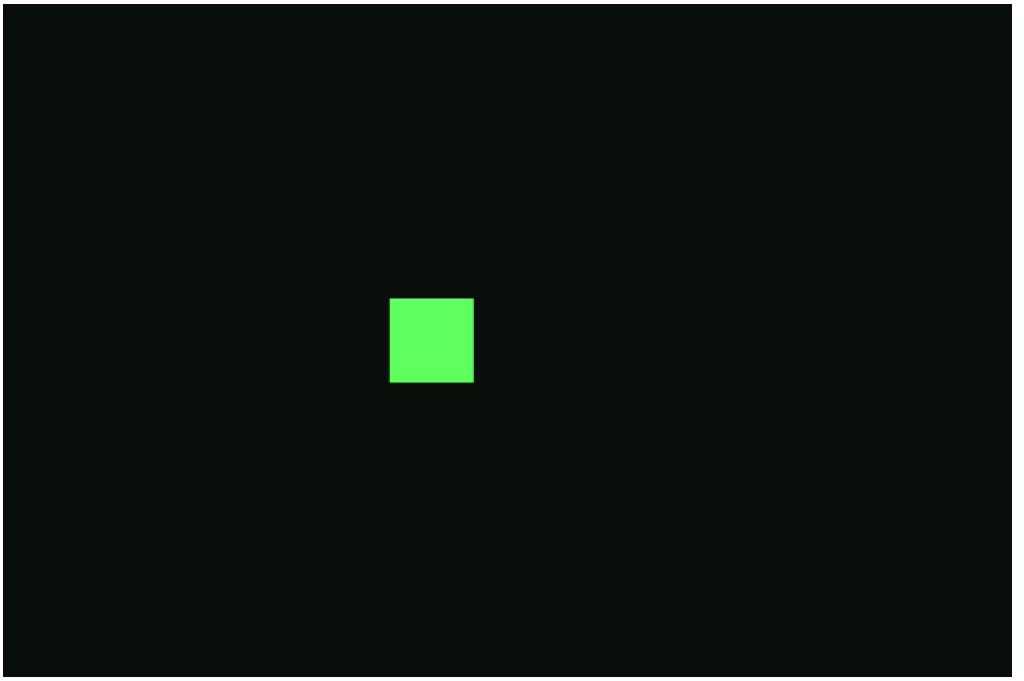


Figure 7. The same box, now crossing at 120 pixels per second on every machine.

The mechanics: `requestAnimationFrame` passes your function a high-precision timestamp in milliseconds — that's the `now` parameter. (`performance.now()`, which seeds `last`, reads the same clock — so frame one measures time since the script started, and every frame after measures time since the frame before.) Subtract the previous frame's timestamp (`last`), divide by 1000, and you have `dt` in seconds: at 60fps about `0.0167`, at 144Hz about `0.007`. Then `SPEED * dt` is distance in pixels — 120 px/sec × 1/60 sec = 2 px, exactly Chapter 5's step, but now *derived* instead of hard-coded.

Two details worth noticing. First, we start the loop with `requestAnimationFrame(loop)` rather than calling `loop()` directly — the function needs a timestamp argument, and the browser is the one who supplies it. Second, `last = now` runs *every* frame; forget it and `dt` measures time since the loop *started*, which sends everything accelerating off-screen within a second. (An educational bug to cause on purpose. Once.)

One caveat that will save a playtester's afternoon: `requestAnimationFrame` stops entirely while a tab is in the background. Switch away mid-game, come back, and `now - last` isn't sixteen milliseconds — it's *seconds*, delivered as one giant `dt`. Everything jumps by `velocity × several seconds` in a single frame: the box teleports, and in a game with collision, a ball can pass clean through a wall it should have bounced off. Real games clamp the delta — `const dt = Math.min(0.05, (now - last) / 1000);` — accepting one imperceptibly slow frame after a tab switch in exchange for never teleporting. The listings in this book stay unclamped to keep the pattern bare; add the clamp to anything you ship.

And if you're on an ordinary 60Hz monitor, feeling skeptical, wanting to *see* the disease this chapter cures: simulate a faster display. Run the update twice per frame, handing each pass half the measured time —

```
const half = dt / 2;
x += SPEED * half;
x += SPEED * half;
```

The `dt` version doesn't blink — two half-steps make one whole step. Now double up Chapter 5's version instead (`x += 2; x += 2;`, no `dt` in sight): the box doubles its speed, because per-frame code counts *passes*, and you just changed how many happen each frame. That, in four lines, is the entire class of bug — reproduced on whatever screen you own.

From this chapter on, every listing that moves something by a speed uses the `dt` pattern, and every speed is in per-second units. It's three extra lines that make your game run identically on every machine that will ever play it — the cheapest insurance in game development.

Try it

Change `SPEED` to `300` and refresh. The point of the exercise is what you *don't* have to do: no other math changes, nothing else is touched, the number means "300 pixels per second" and that's simply now true. Tuning becomes turning real dials with real units — that's what you bought with those three lines.

A worked solution (well, a worked observation) is in Appendix A.

Chapter 7: Keyboard Input

Time to hand the player a controller. The keyboard tracking has been sitting in your workbench since the setup chapter; this is the chapter where it earns its place.

Recall what the boilerplate does: on `keydown` it sets `keys[e.key] = true`, on `keyup` it sets it back to `false`. The result is a live map of the keyboard — at any instant, `keys['ArrowLeft']` tells you whether that key is *currently held*. This is called **polling**, and it's worth understanding why games poll instead of reacting to key events directly.

Browser key events are built for typing, and they behave like typing: press and hold a key in a text box and you get one letter… pause… then a machine-gun repeat, at whatever rate the OS keyboard settings say. Drive a paddle with that and it lurches, hiccups, then stutters across the screen. Worse, events arrive *between* frames, on their own schedule. The polling pattern converts those messy, typing-shaped events into clean state that your loop reads on *its* schedule: every frame, mid-update, exactly when it needs to know. Events write the map; the game loop reads it. (File that sentence away — it's Chapter 16 in miniature.)

Replace your lesson code with this:

```
const p = { x: 220, y: 140, w: 40, h: 40, speed: 250 };
let last = performance.now();

function loop(now) {
  const dt = (now - last) / 1000; last = now;

  if (keys['ArrowLeft'])  p.x -= p.speed * dt;
  if (keys['ArrowRight']) p.x += p.speed * dt;
  if (keys['ArrowUp'])    p.y -= p.speed * dt;
  if (keys['ArrowDown'])  p.y += p.speed * dt;

  ctx.fillStyle = '#0a0e0a';
  ctx.fillRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = '#5fff5f';
  ctx.fillRect(p.x, p.y, p.w, p.h);

```

```
  ctx.fillStyle = '#6a8a6a';
  ctx.font = '11px monospace';
  ctx.fillText('ARROWS to move', 10, 20);

  requestAnimationFrame(loop);
}
requestAnimationFrame(loop);
```

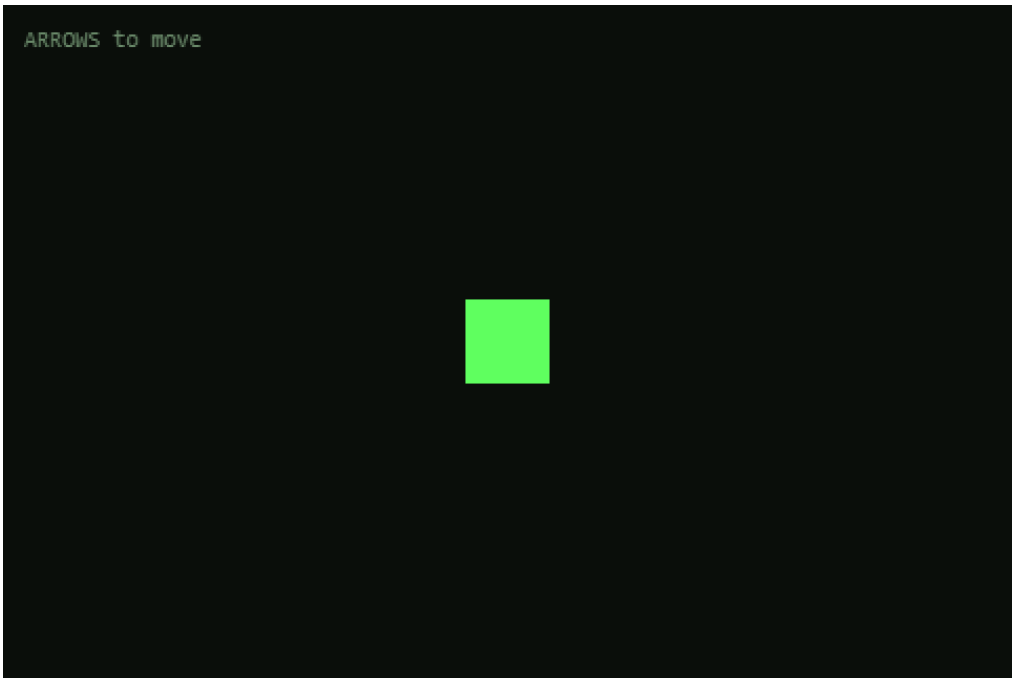


Figure 8. The keyboard-driven box, waiting for arrows.

Refresh, and drive. Movement is smooth because every frame asks "is the key down *right now*?" and moves by `speed * dt` – the delta-time pattern from last chapter, now steered by a human. Press up and right together and you move diagonally, for free, because the four checks are independent. And notice the player is now an *object* – `x`, `y`, size, speed, bundled together. As games grow, "a thing in the game" becomes "an object with properties" so naturally you'll stop noticing you're doing it; this is the first of many.

Two supporting details. `fillText(string, x, y)` makes its debut here – text drawing is that simple (a font, a color, a position), and Chapter 15 builds a full HUD out of it. And remember why arrow keys don't scroll the page while you play: the `preventDefault` line in your workbench. Your workbench page

is too short to scroll, so you won't catch it misbehaving here — but embed a canvas in a real, scrollable page without that line and every press of down drags the page out from under the game.

Try to escape. Drive off the left edge and keep going — the box exits reality, because nothing says it can't. `p.x` will happily reach -3000. Constraints on movement are your job, and they're the challenge.

Try it

Clamp the player so it can't leave the canvas. The idiom: `p.x = Math.max(0, Math.min(canvas.width - p.w, p.x))` — read it inside-out: "no bigger than the right limit, and no smaller than 0." Apply the same treatment to `p.y`. This one-liner guards every screen edge in Part II's Breakout, so make friends with it now.

A worked solution is in Appendix A.

Chapter 8: Mouse Input

The last fundamental. Your workbench keeps a `mouse` object current — `mouse.x` and `mouse.y` in canvas coordinates, `mouse.down` while a button is held, the touch block feeding finger-drags into the same object — and this chapter puts it to work, plus one genuinely pretty rendering trick as a bonus.

First, the coordinate conversion in the boilerplate deserves its explanation, because everyone who skips it pays later:

```
1  const r = canvas.getBoundingClientRect();
2  mouse.x = (e.clientX - r.left) * (canvas.width / r.width);
```

Mouse events report *viewport* coordinates — pixels from the browser window's top-left. Your game thinks in *canvas* coordinates — pixels from the canvas's top-left. Subtracting the canvas's position in the window (`r.left`, `r.top`) fixes that. The multiplication handles a sneakier case: CSS can display a canvas at a different size than its actual pixel resolution, and if the two differ, clicks land where the cursor *isn't* — a bug that reliably appears the day you make your game responsive. The ratio `canvas.width / r.width` converts between the two. With your current setup the ratio is 1 and does nothing; it's there so it never becomes your problem.

It also hands you a free upgrade whenever you want one. Add `width: 960px; height: 640px; image-rendering: pixelated;` to the workbench's `canvas` CSS rule and the game displays at double size — the canvas still *thinks* in 480×320, every listing in this book still works, and clicks still land, because the ratio line is finally earning its keep. (`image-rendering: pixelated` keeps the blown-up pixels chunky instead of smeared.) One CSS edit, twice the game.

Replace your lesson code with this:

```
function loop() {
  // Fade trail (instead of full clear)
  ctx.fillStyle = 'rgba(10, 14, 10, 0.15)';
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  ctx.beginPath();
  ctx.arc(mouse.x, mouse.y, mouse.down ? 28 : 14, 0, Math.PI * 2);
  ctx.fillStyle = mouse.down ? '#ff5fa8' : '#5fff5f';
  ctx.fill();

  requestAnimationFrame(loop);
}
loop();
```

Figure 9. The comet tail: a bright dot trailing its own fading history.

Refresh and move the mouse: a dot follows the cursor, trailing a comet tail. Hold the button: it swells and turns pink. The circle is Chapter 2, the loop is Chapter 5, and `mouse.down ? 28 : 14` is the entire input handling – polling, exactly like the keyboard, just reading a mouse instead.

The comet tail is the trick worth stealing. Chapter 5 said clearing every frame is non-negotiable – here's the loophole. Instead of erasing with an opaque color, we "clear" with the background at 15% alpha (`rgba(10, 14, 10,`

`0.15)`). Each frame dims what came before rather than deleting it; older frames fade a step further into the dark with every pass, and recent history hangs in the air as a glowing streak. One changed line — full clear to translucent clear — and you get motion blur for free. In Part III this same line does the work behind particle streaks, and it's doing exactly what it does here.

That's the fundamentals, all eight. Drawing, shapes and paths, style, transforms, the loop, real time, keyboard, mouse. Part II spends them: next chapter starts Breakout, and it opens with a rectangle you steer along the bottom of the screen — which, you might notice, you essentially built two chapters ago.

Try it

Two experiments. First, raise the fade alpha to `0.5`, then drop it to `0.03`, and watch the tail shorten and stretch — you're tuning how fast history dies. Second: make particles of ink — while `mouse.down` is held, *don't* clear at all, and you've built the world's smallest drawing app.

A worked solution is in Appendix A.

Part II — Building Breakout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 9: The Paddle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 10: The Ball

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 11: AABB Collision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 12: Bricks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 13: Brick Collision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 14: Game States

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 15: Score & Lives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 16: The Shape of a Game

No new code this chapter. Instead, the most valuable thing this book has to give you — the idea you'll still be using when everything else here has become muscle memory.

Look back at your Chapter 13 listing. Fifty-odd lines, one `loop()` — paddle logic, ball physics, collision, drawing, all braided together. It works, and for a Breakout it's honestly fine. But it would *collapse* under an Asteroids: add enemies that shoot, a ship that respawns, saved high scores, a second level — and every new feature has to thread itself through everyone else's lines. There's a reason almost every 2D game ever shipped converges on the same internal shape, and there's a reason the unfindable bugs in big projects almost always trace back to someone breaking it.

Here is the shape:

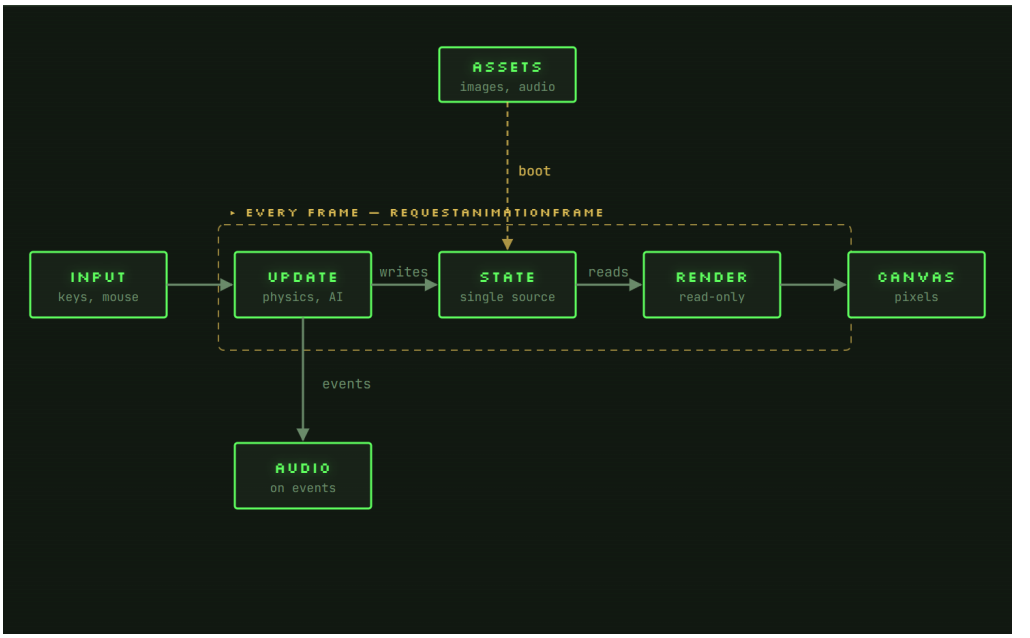


Figure 10. The shape of a game: input flows through update into state; render reads state and draws to the canvas. Assets load once at boot; audio fires on events.

Each frame, data moves through four stations in order. **Input** collects the raw facts — which keys are down, where the mouse is. **Update** reads input and the current state, runs physics and game logic, and *writes* the new state. **State** is the single source of truth: one place where the world lives — ball, paddle, bricks, score. **Render** reads that state — read-only! — and draws it to the canvas. Off to the side, **audio** fires on events that update emits, and **assets** (images, sounds) load once at boot, before the loop ever starts.

Data flows in one direction: left to right, through state. And that one-way flow buys you the cheapest debugging deal in software. Something looks wrong on screen? There are exactly two suspects: *the state is wrong* (update's fault) or *the state is right but drawn wrong* (render's fault). Log the state and you know which within a minute. Two places to look — never one of fifty.

The shape teaches best through its violations. Here are the three classic ones — each with its trademark symptom, so you can recognize them in the wild.

Failure: render writes to state

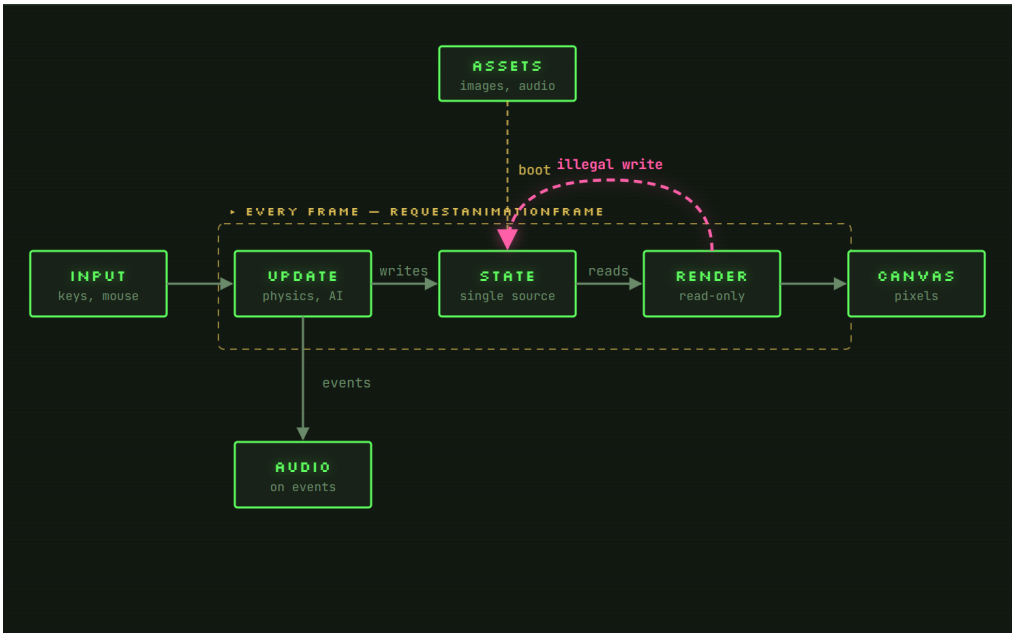


Figure 11. Render reaching backward to write state — the illegal arrow.

Drawing code that mutates the world: adjusting a position "just for this frame," animating a value it doesn't own, caching a measured text width into the state. It feels harmless — the code is *right there*. Three things break. Rendering bugs become **gameplay** bugs. Replays and networked games diverge across machines, because rendering-side float rounding differs. And pause stops working, because drawing keeps mutating things even when update is switched off. The trademark symptom: *"the bug only appears when the window is a certain size."* If a gameplay bug depends on anything visual, a render is writing somewhere it shouldn't.

Failure: update draws to the canvas

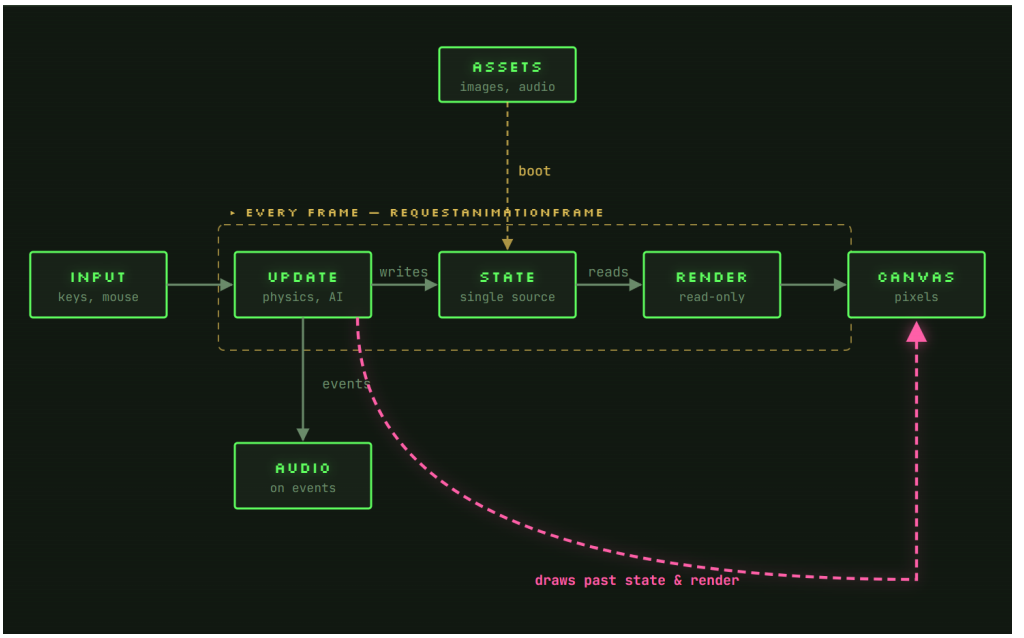


Figure 12. Update sneaking a draw call past state and render.

A `fillRect` in the middle of physics code — usually born as a debug visualization that never left. Now update *requires* a canvas to exist: headless tests become impossible, you can't run two update ticks back-to-back to catch up after a slow frame without double-drawing, and pause-and-rewind is broken because the draw happened *during* the simulation step. The trademark symptom: *"I can't add a server / AI bot / test runner to my game."* Update should be able to run in the dark.

Failure: no single source of truth

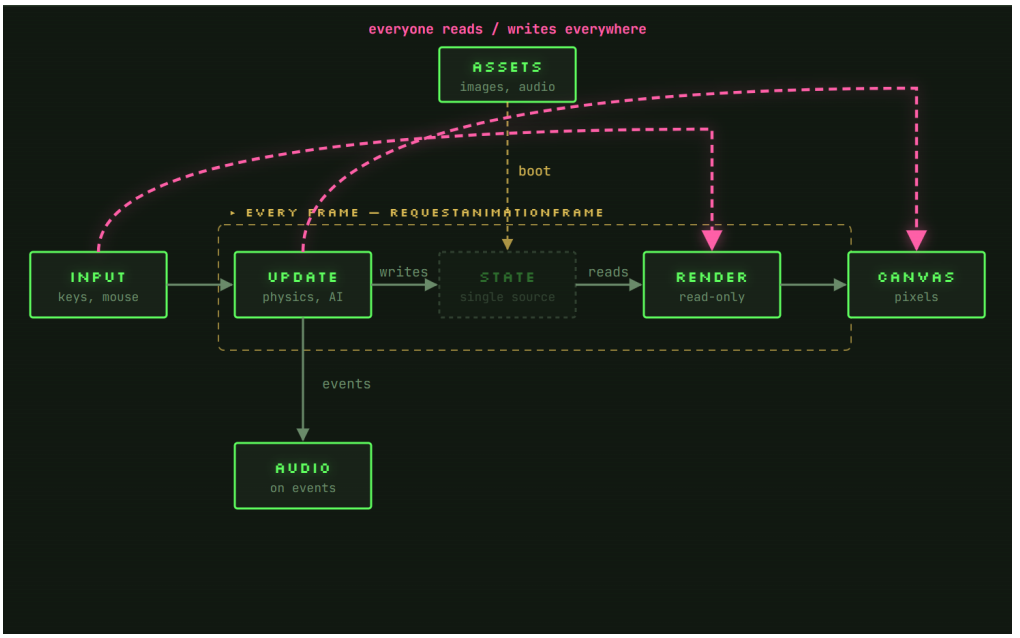


Figure 13. No state node: every system holding private copies, arrows everywhere.

The subtlest one, because nothing crashes. Every system keeps private variables — input holds its own coordinates, update its own positions, render its own copies — and they drift. Save/load is impossible (what would you even save?). Pause is impossible (there's no *world* to freeze). Replays, multiplayer rollback, time-travel debugging — all off the table. The trademark symptom: "*two parts of the game disagree about where the player is.*"

You've been doing it all along

Here's the part I owe you a confession about: this book has been sneaking the shape into your hands since Chapter 5. "Update touches only variables; draw touches only the canvas" — that was update/render. The workbench's key listeners *write* the `keys` map and your loop *reads* it — that's the input system, events on one side, polling on the other (Chapter 7 said so in miniature). Chapter 14 kept transitions and drawing in separate blocks. Chapter 15's HUD read the score and touched nothing. And two chapters from now, Chapter

18 will tell you to shake the world but not the scoreboard. None of that was fussiness — it was this diagram, arriving one arrow at a time.

The discipline costs nearly nothing in a small game and pays off precisely when the game stops being small — which is always the moment you least want to refactor. Keep the shape from line one and you never have to.

Try it

Open your Chapter 13 listing and label every line: **Input**, **Update**, or **Render** — and mark where **State** lives. Two questions to answer while you're in there: which lines *write* state, and does anything violate the shape? Appendix A has the full labeling, including one line that's easy to misfile and the answer to whether Chapter 13 was already clean.

Part III — Polish

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 17: Particles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 18: Screen Shake

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 19: Easing & Tweening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 20: Sound (Web Audio)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 21: Sprite Sheets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Part IV — Capstone

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 22: Finish the Breakout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 23: Sandbox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Try it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 24: What You Built

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Drawing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Motion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Collision & State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Polish

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Where to go next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Appendix A: Challenge Solutions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 1: Hello, Canvas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 2: Shapes & Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 3: Colors & Style

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 4: Transforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 5: The Game Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 6: Delta Time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 7: Keyboard Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 8: Mouse Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 9: The Paddle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 10: The Ball

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 11: AABB Collision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 12: Bricks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 13: Brick Collision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 14: Game States

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 15: Score & Lives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 16: The Shape of a Game

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 17: Particles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 18: Screen Shake

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 19: Easing & Tweening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 20: Sound (Web Audio)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 21: Sprite Sheets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 22: Finish the Breakout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Chapter 23: Sandbox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Appendix B: The Helper Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

The API at a glance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.

Glossary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/canvas-academy.