

chapter one

The Game Loop and the Canvas

An animated sketch updates its state and draws a new frame repeatedly. For this book, p5.js calls a `draw()` function while the sketch runs. We will use it to clear a canvas, update a dot's position, and draw the dot again. The loop is a useful model for a small game, though larger games may organize updates and rendering into separate systems.

At a steady 60 fps, frame starts are about 16.7 ms apart; the actual interval varies.

Two functions to start a sketch

The p5.js Web Editor supplies a sketch and a Play button in a browser. The same code can also run in a local project that loads p5.js.

These examples declare two functions that p5.js calls:

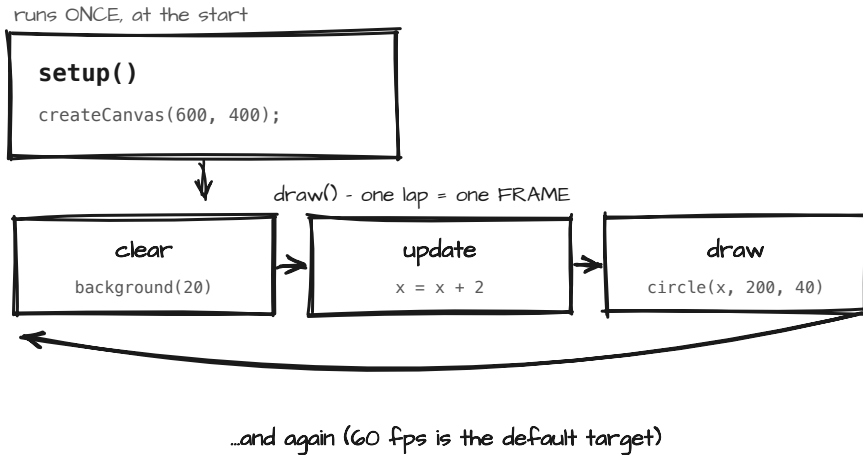
```
function setup() {  
  createCanvas(600, 400);  
}  
  
function draw() {  
  background(20);  
  // everything else happens here  
}
```

`setup()` runs once when the sketch starts. Here it creates our 600×400-pixel canvas. Later examples also initialize their objects there.

`draw()` is called repeatedly while the sketch loops. p5.js normally targets about 60 calls per second, but actual timing depends on the browser and work done each frame. One call produces one frame in these examples.

p5.js invokes `draw()` while looping; `noLoop()` can stop those calls.

`background(20)` paints a dark gray over the canvas at the start of each call. In the default grayscale range, 20 is close to black. Painting the background removes the previous frame's visible marks before the new state is drawn. Omit it when you deliberately want a trail.



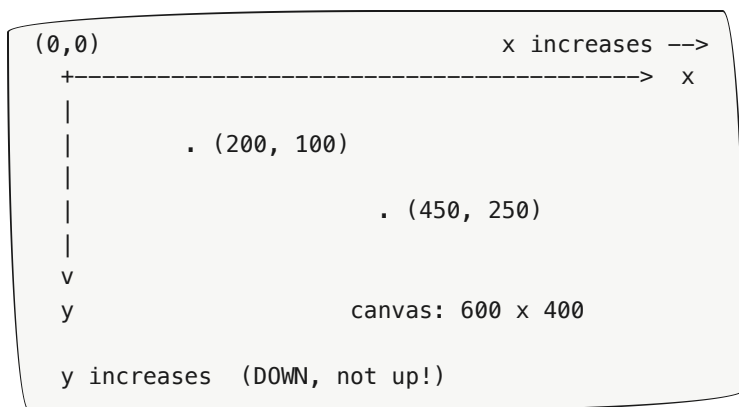
`setup()` initializes the sketch. Repeated calls to `draw()` clear, update, and render this example; 60 fps is a target, not a guarantee.

note

For this sketch: clear, update position, draw. The repetition continues while the p5.js loop is running.

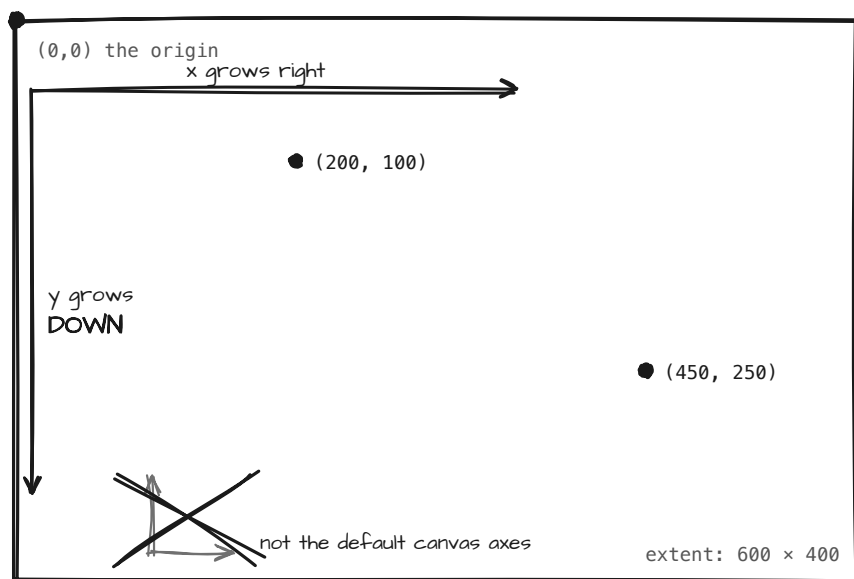
Canvas coordinates

In the default 2D canvas coordinates, the origin `(0, 0)` is at the top-left. x increases to the right and y increases downward. Coordinates need not be whole numbers, even though the rendered image is made of pixels.



Moving up decreases y ; moving down increases it. For a falling object in our default coordinate system, y grows. Canvas transformations can change the coordinate system, so read the current transform when porting or extending a sketch.

In this book's untransformed canvas, downward motion increases y .



'up' means SUBTRACT from y

On the untransformed p5.js canvas, x grows right and y grows down. A downward step adds to y .

Drawing your first shapes

p5.js has commands for simple shapes. This sketch uses their default 2D drawing modes:

```
function setup() {
  createCanvas(600, 400);
}

function draw() {
  background(20);

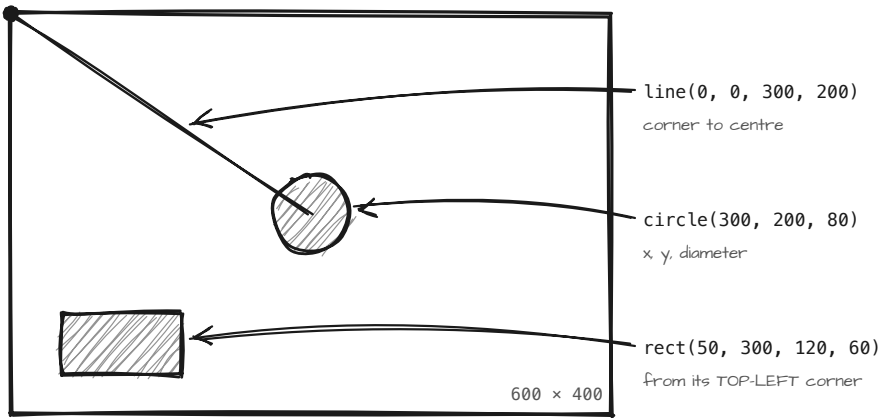
  fill(255, 100, 100); // reddish
  circle(300, 200, 80); // x, y, diameter

  stroke(100, 200, 255); // light blue outline
  line(0, 0, 300, 200); // corner to center

  noStroke();
  fill(120, 220, 120);
  rect(50, 300, 120, 60); // x, y, width, height
}
```

Circle centers and rectangle corners differ in the default drawing modes.

`circle(300, 200, 80)` draws a circle centered on the canvas, with diameter 80. In the default RGB color mode, `fill()` sets the interior color and `stroke()` sets the outline color; `noStroke()` disables that outline. The default `rect()` mode takes the upper-left corner, width, and height. Change a coordinate or color and rerun the sketch to check each argument's effect.



for these shapes: x first, then y

The three shapes use their default 2D modes: a centered circle, a line between endpoints, and a rectangle from its upper-left corner.

Because the coordinates are fixed, the same shapes appear in the same places on each frame. Motion will require changing state between calls to `draw()`.

Frame rate and elapsed time

p5.js normally aims for about 60 frames per second (fps). Calling `frameRate()` without an argument gives an estimate of the current rate; `frameCount` counts calls to `draw()`.

```
function draw() {
  background(20);
  fill(255);
  textSize(20);
  text("fps: " + round(frameRate()), 20, 30);
  text("frame: " + frameCount, 20, 60);
}
```

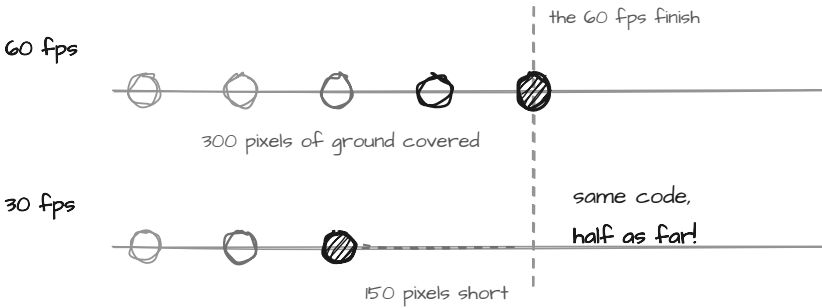
The target may not be reached. If a dot advances five pixels on every call, 60 calls move it 300 pixels while 30 calls

A per-frame step changes real-time speed when the frame rate changes.

move it 150 pixels in the same second. That makes its real-time speed depend on how often `draw()` runs.

```
x = x + 5; // 5 pixels per FRAME
```

if they draw 60 vs 30 frames in one second:



At a fixed five pixels per frame, 60 frames cover 300 pixels and 30 frames cover 150 pixels over one second.

Delta time for constant-speed motion

Specify the dot's speed in pixels per second, then multiply by the elapsed seconds for each frame. p5.js provides `deltaTime` in milliseconds for the previous frame interval. At a steady 60 fps it is about 16.7 ms; a delayed frame normally yields a larger value.

Convert
`deltaTime` from
milliseconds to
seconds before
multiplying by pixels
per second.

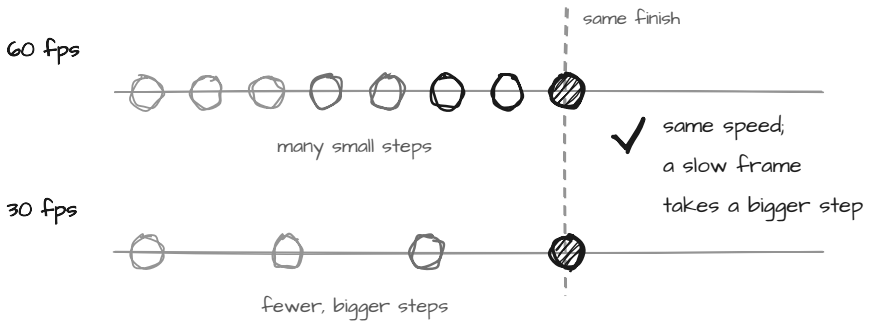
```
let x = 0;
let speed = 120; // pixels per SECOND

function setup() {
  createCanvas(600, 400);
}

function draw() {
  background(20);
  let dt = deltaTime / 1000; // ms -> seconds
  x = x + speed * dt;
  fill(255);
  circle(x, 200, 40);
}
```

```
x = x + speed * dt; // 120 px per SECOND
```

constant speed over the same elapsed second:



For constant speed, smaller or larger time steps sum to the same displacement over the same elapsed time, apart from timing and rounding details.

Here `speed * dt` has units of pixels. For a constant speed, the increments sum to approximately 120 pixels per elapsed second. The frame on which the dot appears at a particular position may differ. Games with collision detection or acceleration need more care: a very large time step can skip over an obstacle or make a numerical update unstable.

Elapsed-time scaling fixes constant-speed motion, not every variable-step simulation problem.

Many later sketches use per-frame steps to keep the arithmetic visible. Treat those numbers as “per update,” not guaranteed real-time speeds. For a game that must behave consistently across frame rates, choose a timing strategy and test slow frames as well as normal ones.

Your first moving dot

For a minimal per-frame example, store the dot's horizontal position in `x`. Each call clears the canvas, advances `x`, and draws the dot at its new position:

```

let x = 0;

function setup() {
  createCanvas(600, 400);
}

function draw() {
  background(20);

  x = x + 2; // pixels per update
  if (x > 600) {
    x = 0; // wrap to the left
  }

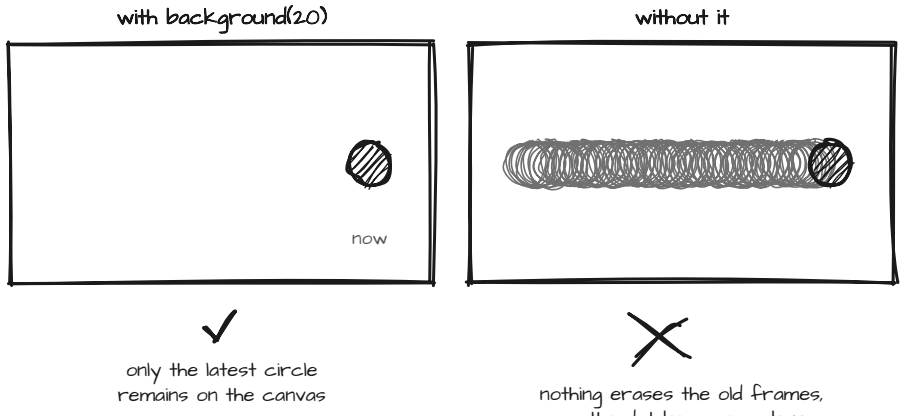
  fill(255, 200, 80);
  noStroke();
  circle(x, 200, 40);
}

```

`x` must persist between calls; a local `let x = 0` inside `draw()` would restart it every frame.

`x` is declared outside `draw()`, so its value survives between calls. The sketch adds two pixels per update and wraps the center back to zero after it passes `x = 600`. The dot is partly clipped at the edges because its center, not its radius, controls that wrap. The previous drawing is covered by `background(20)`; the next call draws the circle at a different coordinate.

Comment out `background(20)`. The earlier circles remain, so overlapping positions form a bar. Restore the call to show only the latest position.



Clearing each frame leaves one current circle. Without the background call, previous circles remain visible and overlap into a bar.

note

Try changing the update to move upward, adding a second dot with another speed, or changing the fill color with `x`. Predict the result first, then run the sketch.

From one coordinate to two

The dot uses one position coordinate and one per-update step. Two-dimensional motion needs `x` and `y` components for both position and velocity. Chapter 2 introduces vectors to hold and operate on those pairs. We will also keep the distinction between per-frame and per-second units as the updates become more involved.

A vector bundles related components; its meaning still depends on units and use.

Exercises

Exercise 1.1. The moving-dot sketch adds 2 to `x` every frame and wraps back to 0 at 600. Change that step to `x = x + 5` and change nothing else. Before you run it, pred-

ict two things: how the dot's speed changes, and how often it wraps around. Then run it and check your guess.

Exercise 1.2. Take the moving-dot sketch and make it travel straight *up* the screen instead of rightward, reappearing at the bottom when it leaves the top. Remember that the origin is the top-left corner and that "up" means subtracting from `y`. Write the new position update and the new wrap condition.

Exercise 1.3. A friend deletes the `background(20)` line "to save time." Now the dot smears into a solid bar across the canvas. Explain in one sentence why the smear happens, and name the single line that fixes it.

Exercise 1.4. Rewrite the moving dot with `deltaTime` and a speed in pixels per second. Compare its approximate displacement over one second at 30 and 60 fps. What remains different about the individual steps?