

Lecture 5: Timer-Based Animation

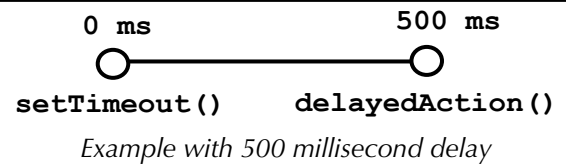
Written by Ben Yan, from materials by Jerry Cain

JavaScript Timers

Last week, we discussed *interactive* programs, which have “callback” functions that respond to mouse events. These events can occur at sudden, irregular times (depending on the user’s actions), or not at all.

Today, we’ll go over *animated* programs, with callback functions that respond to time-scheduled events. There are two types of timers you’ll want to know. The first is a *one-shot timer*, which has this structure

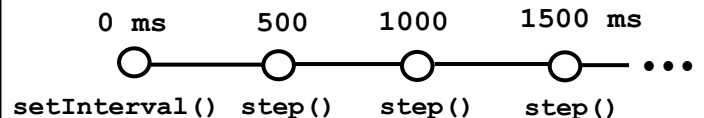
```
let delayedAction = function() {
    block of code that runs after the delay
}
setTimeout(delayedAction, delay);
```



This invokes its callback function, e.g., `delayedAction`, once after the input `delay` in milliseconds.

The second—and more widely used in timer-based animation—is the *interval timer*, with code template

```
let step = function() {
    block of code that runs
    repeatedly at regular intervals
}
let timer = setInterval(step, delay);
```



This timer invokes its callback function, e.g., `step`, not once, but repeatedly and regularly at time intervals of `delay` milliseconds. For instance, if `delay = 500`, then `step` is executed for the 1st time after 500 ms, a 2nd time after 1000 ms, a 3rd time after 1500 ms, and so on. **Note the `setInterval` function returns a `timer` value you can use to terminate it, i.e. by calling `clearInterval(timer)`.**

A metaphor is setting alarms for the morning, in fact many alarms to guarantee you wake up for class ☺. The alarm, of course, doesn’t go off right away when you set it—that would be very *alarming*. Instead, **it’s delayed to ring at 7 AM, then every 10 minutes**, i.e. at 7:10 AM, 7:20 AM, 7:30 AM, so on, until you finally get up and turn off the darn thing. The ring is the `step` function, the timespan between rings is the `delay`, the turn off is `clearInterval`. Here’s this (hopefully not too relatable) metaphor in code.

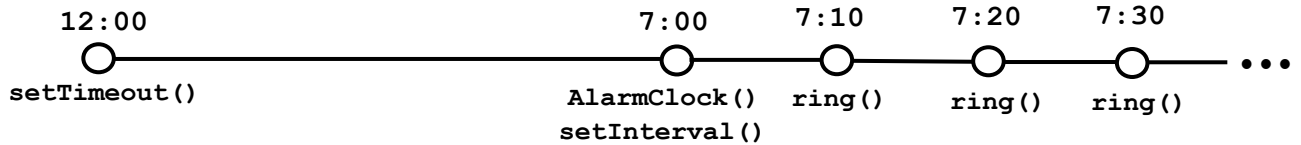
```
const SLEEP_DELAY = 7 * 60 * 60 * 1000; // 7 hours of beauty sleep
const ALARM_DELAY = 10 * 60 * 1000; // time between rings, in milliseconds

function AlarmClock() { // executes only after we sleep 7 hours
    let ring = function() { // executes every 10 min after timeout
        console.log("BEEP BEEP BEEP BEEP!! GET UP BEN!!");
    }
    let timer = setInterval(ring, ALARM_DELAY);
}

setTimeout(AlarmClock, SLEEP_DELAY);
```



The above example uses both the *one-shot* and *interval* timers, each to different effect (one to delay the alarm until morning, the other to ring the alarm repeatedly after the delay), but is absent of any graphics. A caveat is that the alarm first rings at 7:10, not 7:00, due to the initial interval delay, but oh well ☺.

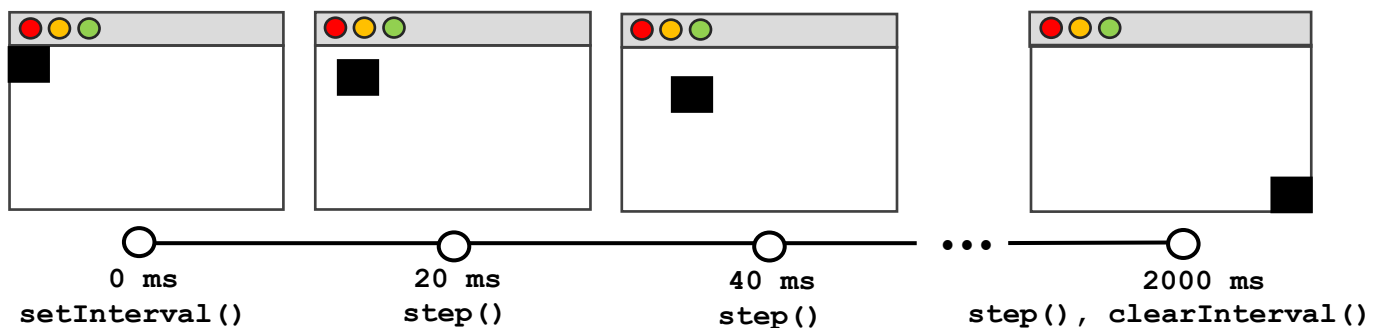


The following programs will focus on **graphics**—and how, by setting a fast-firing **timer**, and having the callback function repeatedly make small updates to **GObjects** on a canvas, we can produce **animation** 🙌. Indeed, in animated films, it's the sequence of evolving frames that creates the illusion of movement.

If the time interval is short enough (typically 20-30 milliseconds), the animation will appear smooth to the naked eye. For cinema or video game enthusiasts, this is equivalent to 33 – 50 frames per second ☺.

Animated Square

Here is an introductory example that animates a square to move diagonally from the top-left corner of the window to the bottom-right corner, as displayed in frames below. It's like the old gliding DVD logo.



Assume the following constants are defined. Note we require the square to take exactly **N_STEPS** = 100 steps from start to end, and that steps occur at regular intervals of **TIME_STEP** = 20 milliseconds.

```
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const N_STEPS = 100;
const TIME_STEP = 20;
const SQUARE_SIZE = 50;
```

First, we'll create the graphics window, and a square with the specified dimensions. The square is placed so that it begins in the top-left corner of the window, as visualized above in the leftmost frame.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let square = GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
square.setFilled(true);
gw.add(square);
```

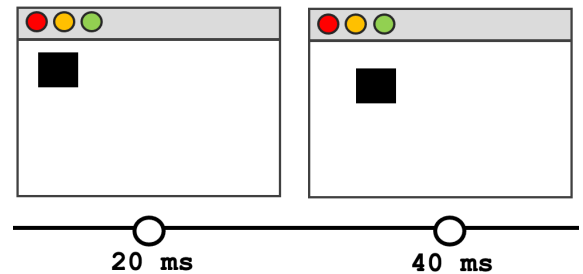
More interestingly, we'll compute **dx** and **dy**, how far the square needs to move per step in the **x** and **y** directions. From start to end, the square's top-left corner goes from (0, 0) to point (**w - s**, **h - s**),

for canvas height **h** and width **w**, and square size **s**, all in pixels. To accomplish this in **n** steps, then **dx** = **(w - s) / n**, and **dy** = **(h - s) / n**. In JavaScript, using the above constants, we can write

```
let dx = (gw.getWidth() - SQUARE_SIZE) / N_STEPS;
let dy = (gw.getHeight() - SQUARE_SIZE) / N_STEPS;
```

Here's where the fun begins. We implement the callback **step**, which is called regularly on a schedule of every **TIME_STEP = 20**. Each time step, we move the square by **dx** and **dy**, as calculated above.

```
let stepCount = 0;
let step = function() {
    square.move(dx, dy);
    stepCount++;
    if (stepCount === N_STEPS) {
        clearInterval(timer);
    }
};
let timer = setInterval(step, TIME_STEP);
```



We also keep track of **stepCount** which is declared outside so that all calls to **step** access the same variable in memory. This counter is used to terminate the animation once **N_STEPS** is reached, which means the square has arrived at its destination in the window's bottom-right corner.

Putting this all together, we get the following implementation of **AnimatedSquare** ☺.

```
function AnimatedSquare() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    let dx = (gw.getWidth() - SQUARE_SIZE) / N_STEPS;
    let dy = (gw.getHeight() - SQUARE_SIZE) / N_STEPS;

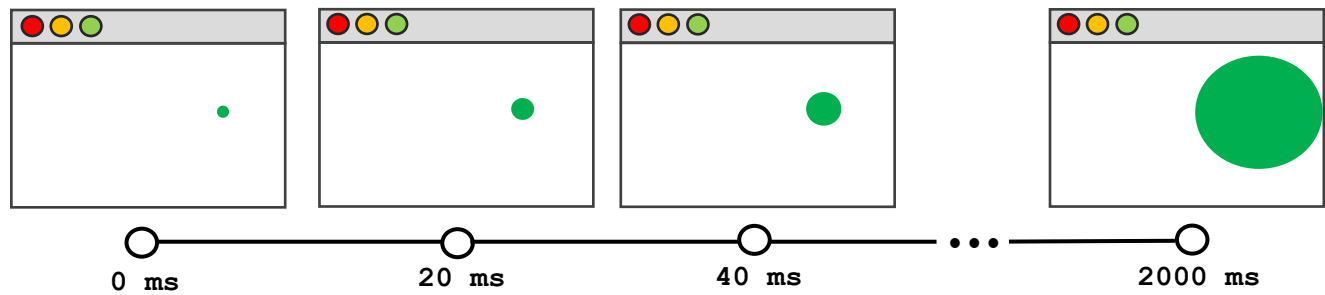
    let square = GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
    square.setFilled(true);
    gw.add(square);

    let stepCount = 0;
    let step = function() {
        square.move(dx, dy);
        stepCount++;
        if (stepCount === N_STEPS) {
            clearInterval(timer);
        }
    };
    let timer = setInterval(step, TIME_STEP);
}
```

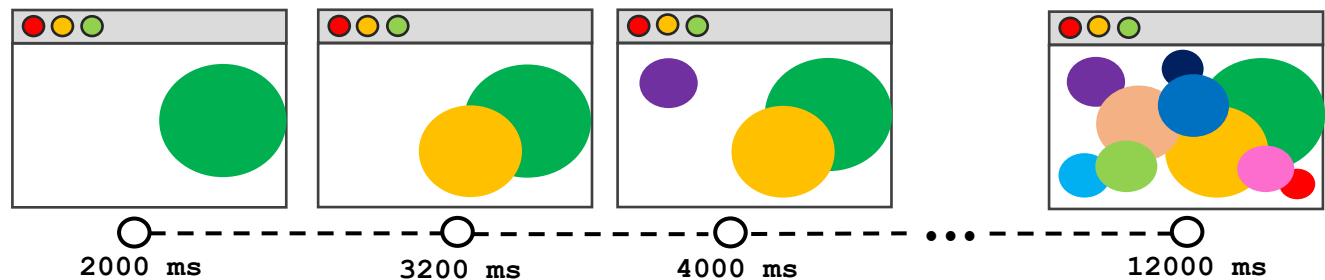
Exploding Circles

We'll write a program **GrowingCircles** that **draws 10 filled, randomly shaded circles at various locations within a graphics window**. The program is ornamented with the following animated features.

For the first circle, its center coordinate and full radius are randomly generated so it fits entirely within the canvas. Initially, we draw the circle with radius zero, but the radius gradually increases over several time steps until the radius matches what we generated previously. An example is displayed below for a circle that grows **1 pixel in diameter** every **20 ms**, until it becomes a fully-fledged **50 pixels in radius**.



Each of the other circles is introduced in the same “exploding” manner as above, but the second circle is only placed after the first circle has fully expanded, the third circle is only placed after the second has fully expanded, and so forth. The animation ends when the tenth and final circle is fully expanded. The timeline below displays a 2nd circle with a 30-pixel radius, a 3rd circle with a 20-pixel radius, and so on.



Assume the following constants are defined. Each circle’s radius is a randomly and independently chosen value between **MIN_RADIUS = 15** and **MAX_RADIUS = 50** pixels. The currently “exploding” circle grows by **DELTA_SIZE = 1** pixels in diameter every **TIME_STEP = 20** milliseconds.

```
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const N_CIRCLES = 10;
const MIN_RADIUS = 15;
const MAX_RADIUS = 50;
const TIME_STEP = 20;
const DELTA_SIZE = 1;
```

First, we create the canvas, and a variable to keep track of how many circles we’ve fully created so far.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
let circlesCreated = 0;
```

We'll also want to store—and make accessible to callback functions—the current **circle** (a **GOval** object) we're expanding, as well as its current diameter, and the full diameter it should be expanded to.

```
let desiredSize = 0; /* all are initialized to temporary zero values! */
let currentSize = 0;
let circle = null;
```

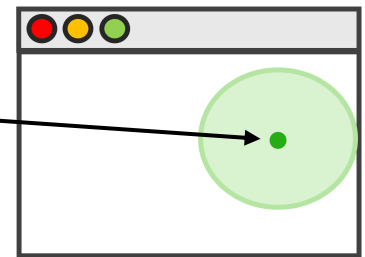
Next, we'll write a helper function that **generates a circle with initially zero radius, but with a random future radius r when fully grown**. The circle's color and center coordinate (x, y) are also random. We assign the **GOval** object to **circle** above, as well as update variables **desiredSize**, **currentSize**.

```
let createNewCircle = function() {
  let r = randomReal(MIN_RADIUS, MAX_RADIUS); // future radius!

  let x = randomReal(r, GWINDOW_WIDTH - r);
  let y = randomReal(r, GWINDOW_HEIGHT - r);

  circle = GOval(x - 0, y - 0, 0, 0);
  circle.setFilled(true);
  circle.setColor(randomColor());

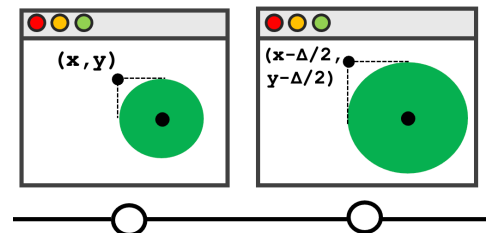
  desiredSize = 2 * r; // note size is diameter
  currentSize = 0;     // not the radius!
  return circle;
}
```



Note for the fully-fledged circle to fit entirely within the window, we must have $r \leq x \leq w - r$ and $r \leq y \leq h - r$, where w and h are the width and height of the graphics window, respectively.

To proceed, we focus on just one circle at a time ☺, and draft a **step** function which grows the current **circle**'s diameter by **DELTA_SIZE** = 1 pixel, as long as the circle hasn't reached its fully grown size.

```
let step = function() { // this is called every TIME_STEP milliseconds!
  if (currentSize < desiredSize) {
    currentSize += DELTA_SIZE;
    let x = circle.getX() - DELTA_SIZE / 2;
    let y = circle.getY() - DELTA_SIZE / 2;
    circle.setBounds(x, y, currentSize, currentSize);
  }
  /* more code to implement later! */
};
let timer = setInterval(step, TIME_STEP);
```



Note that a **GOval**'s location is specified by its top-left coordinate. Thus, **when the circle's diameter increases by Δ pixels**, to maintain the same center, its **top-left coordinate actually moves**—up and to the left by $\Delta / 2$ pixels. We then use **.setBounds($x, y, width, height$)** to alter the circle's location and dimensions to effectively grow the circle outward from its center.

Next, we'll need to handle the **else** condition, i.e. when **currentSize** reaches **desiredSize**. In this case, we should check if there are more circles to grow, or if the animation should be terminated.

```

let step = function() {
  if (currentSize < desiredSize) {
    /* same code as above to grow the circle's diameter! */
  }
  /* we can add a new circle, and there are more circles to grow */
  else if (circlesCreated < N_CIRCLES) {
    gw.add(createNewCircle()); // start growing a new circle
    circlesCreated++;
  } else { // when all circles are fully expanded, stop animation
    clearInterval(timer);
  }
}

```

If we haven't created all circles yet, we enlist `createNewCircle` to generate a new `circle`, and reset `currentSize`, `desiredSize` so the next call to `step()` enters the `if` loop, growing that new circle.

Constellating all the above jigsaw pieces together, with some style ironing, we have our final solution ☺.

```

function GrowingCircles() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let circlesCreated = 0;
  let desiredSize = 0;
  let currentSize = 0;
  let circle = null;

  let createNewCircle = function() {
    let r = randomReal(MIN_RADIUS, MAX_RADIUS);
    let x = randomReal(r, GWINDOW_WIDTH - r);
    let y = randomReal(r, GWINDOW_HEIGHT - r);
    circle = GOval(x, y, 0, 0);
    circle.setFilled(true);
    circle.setColor(randomColor());
    desiredSize = 2 * r;
    currentSize = 0;
    return circle;
  };

  let step = function() {
    if (currentSize < desiredSize) {
      currentSize += DELTA_SIZE;
      let x = circle.getX() - DELTA_SIZE / 2;
      let y = circle.getY() - DELTA_SIZE / 2;
      circle.setBounds(x, y, currentSize, currentSize);
    } else if (circlesCreated < N_CIRCLES) {
      gw.add(createNewCircle());
      circlesCreated++;
    } else {
      clearInterval(timer);
    }
  };

  let timer = setInterval(step, TIME_STEP);
}

```

