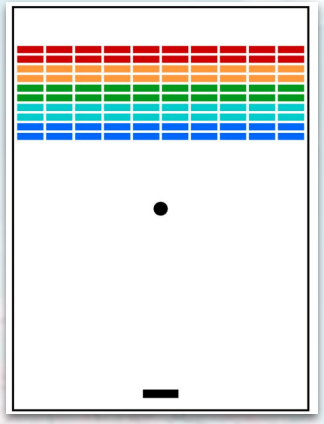
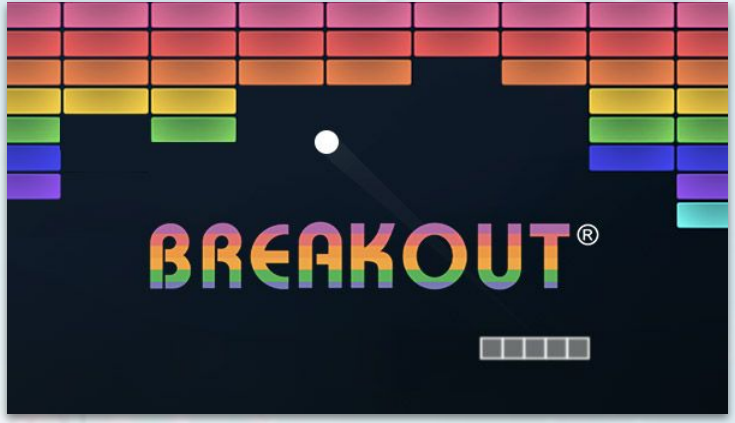
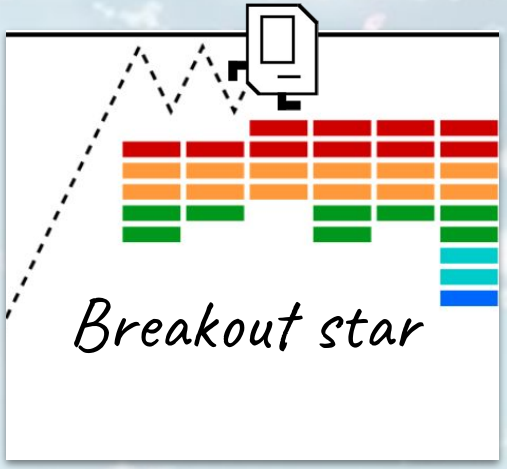
Slides at cs106ax.stanford.edu/assignments.html







YEAH Hours: Assign2

Breakout!

Autumn 2025

cs106ax.stanford.edu

Stanford | ENGINEERING
Computer Science

Welcome to Week 2/3 of 106AX!

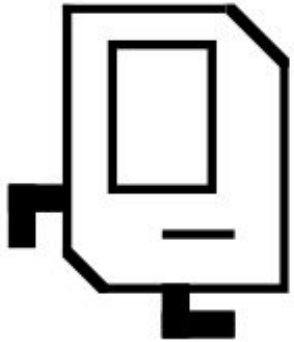
Hope that your weekend is going well!

Autumn

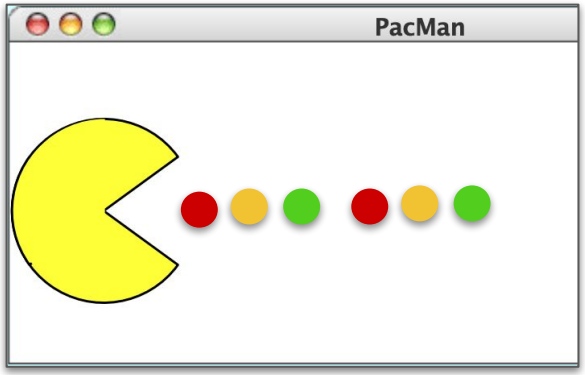
Week 3

Winter

The Map For Today



1. Getting setup
2. Recap of lecture, **graphics + animation**
3. Assignment overview & tips
4. Questions and office hours



Slides adapted from materials by Anand Shankar, Ryan Eberhardt, CS106 staff

Downloading Assign2 Code

- Go ahead and download the **Assign2** code from the course website or **this link**.
- Then, open the files in your text editor!

Test **Breakout** game here (open in web browser)

Write code here

- When editing the code file, make sure to both (1) **save the code**, and (2) **refresh the browser tab** so the changes are reflected!
- Any runtime error messages will pop up in the browser's JavaScript console

Breakout.js

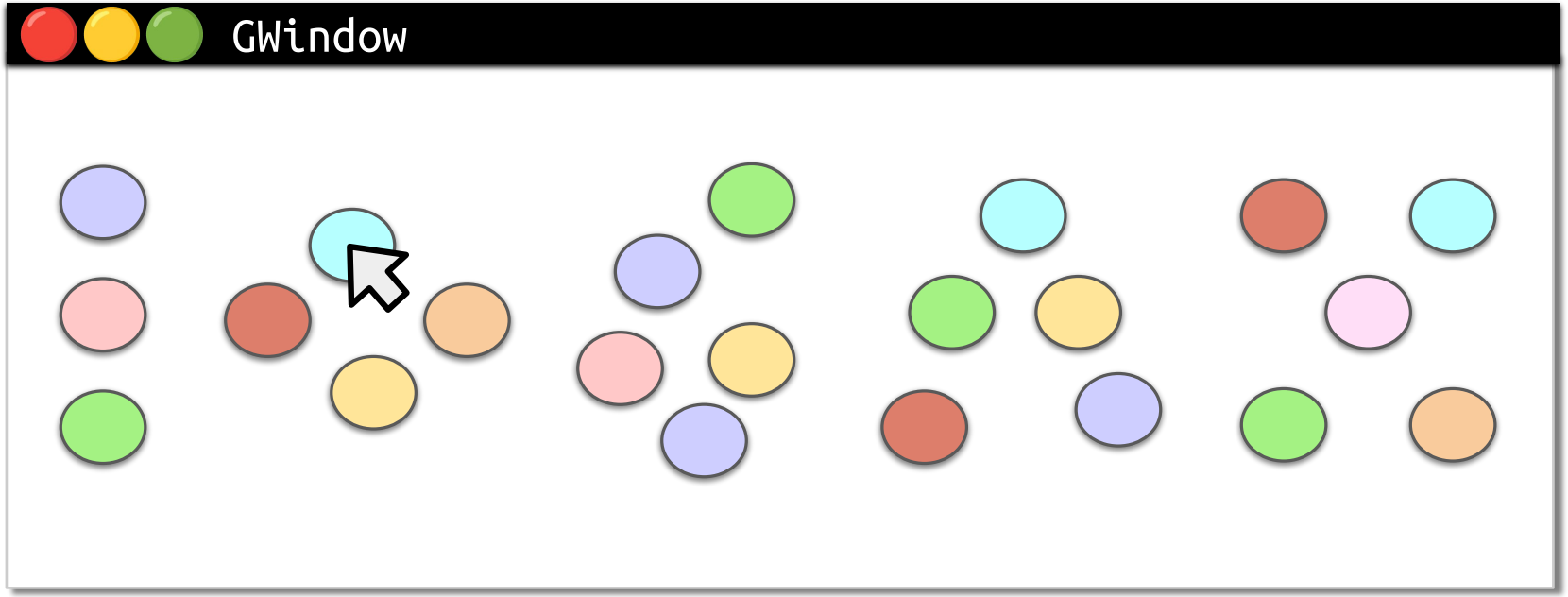
```
/*
* File: Breakout.js
* -----------------
* This program implements the
Breakout game.
*/
"use strict";

/* Constants */
const GWINDOW_WIDTH = 360;
const GWINDOW_HEIGHT = 600;
const N_ROWS = 10;
const N_COLS = 10;
more constants defined here

/* Main program */
function Breakout() {
 // You fill this in along
with any helper and callback
functions.
}
```

Lecture Recap!

⏳ ~10 minutes

JavaScript: Nested Functions / Closure

Consider the lecture example on drawing dots on a **`GWindow`** where it's clicked.



JavaScript: Nested Functions / Closure

Consider the lecture example on drawing dots on a **`GWindow`** where it's clicked.

DrawDots.js

```
function clickAction(e){
 let dot = GOval(e.getX() - DOT_SIZE / 2,
   e.getY() - DOT_SIZE / 2, DOT_SIZE, DOT_SIZE);
 dot.setFilled(true);
 gw.add(dot);
}

function DrawDots() {
 let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
 gw.addEventListener("click", clickAction);
}
```

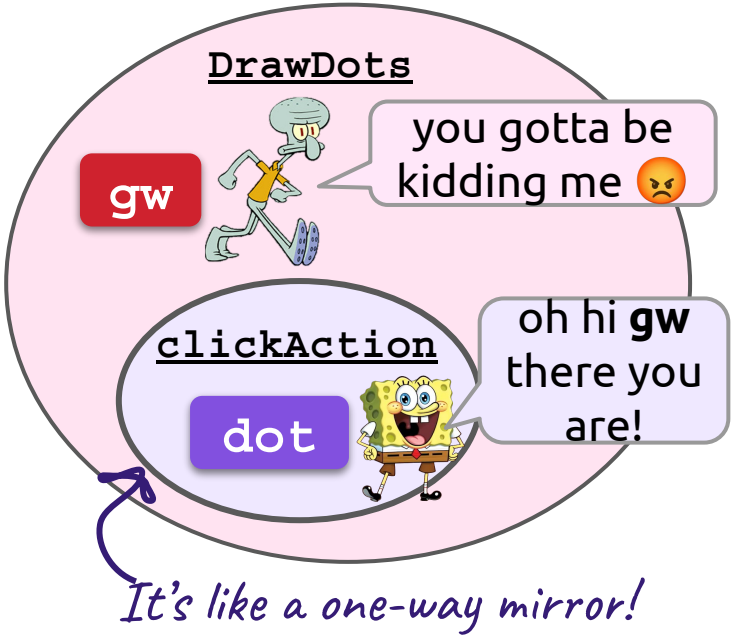
This doesn't work as **`clickAction()` doesn't have access to window** `gw`, which is defined inside a separate function **`DrawDots()`**

JavaScript: Nested Functions / Closure

Instead, we need to **nest the event listener function** so that it inherits its parent function's scope (i.e. has access to its variables such as the **`GWindow`**)

DrawDots.js

```
function DrawDots() {
 let gw = GWindow(GWINDOW_WIDTH,
GWINDOW_HEIGHT);
 let clickAction = function(e) {
   let dot = GOval(e.getX() - DOT_SIZE / 2,
                   e.getY() - DOT_SIZE / 2,
                   DOT_SIZE, DOT_SIZE);
   dot.setFilled(true);
   gw.add(dot);
 };
 gw.addEventListener("click", clickAction);
}
```



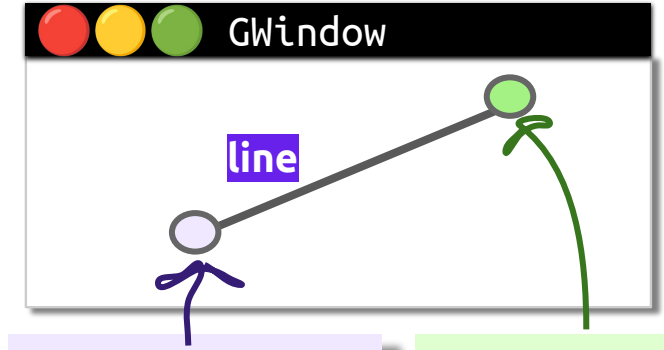
Note: **`clickAction` can access** gw, though **`DrawDots`** can't access the local dot

JavaScript: Nested Functions / Closure

Takeaway: If a graphics variable, e.g., a ball, paddle, needs to be easily accessible later on by callback / event-listener functions, usually declare it first in the parent function.

DrawLines.js

```
function DrawLines() {
 let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
 let line = null;
 let mousedownAction = function(e) {
   line = GLine(e.getX(), e.getY(),
                e.getX(), e.getY());
   gw.add(line);
 };
 let dragAction = function(e) {
   line.setEndPoint(e.getX(), e.getY());
 };

 gw.addEventListener("mousedown", mousedownAction);
 gw.addEventListener("drag", dragAction);
}
```



mousedownAction sets the start point of `line`

dragAction sets the end point of `line`

Note: Both callback functions need access to `line`, so it's declared in the parent function `DrawLines`

JSGraphics: Event Listeners

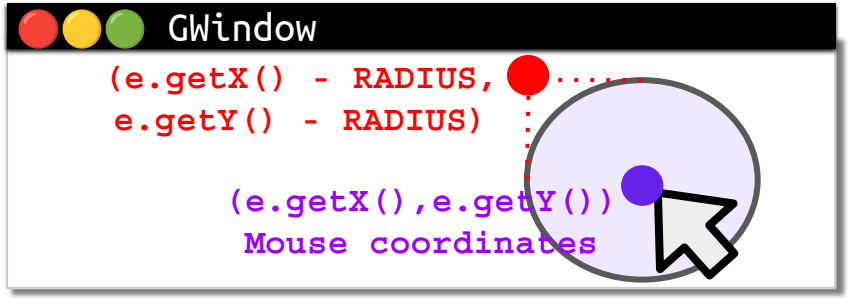
To handle and respond to user mouse interactions, we use **event listeners –** or functions that are **only called when a specific user action** happens.

Mouse events are affiliated with a variable **e**, where **`e.getX()`**, **`e.getY()`** are the mouse's x and y coordinates. This is given to the callback function as information.

The main types of event listeners are:

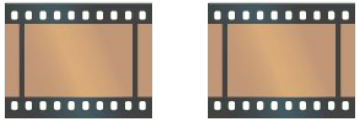
- ❑ **"click"**
- ❑ "dbclk"
- ❑ "mousedown"
- ❑ "mouseup"
- ❑ **"mousemove"**
- ❑ "drag"

Relevant for `Breakout!` game




```
let clickAction = function( e ) {
   let dot = GOval(e.getX() - DOT_SIZE / 2,
     e.getY() - DOT_SIZE / 2, DOT_SIZE, DOT_SIZE);
   dot.setFilled(true);
   gw.add(dot);
}; // callback function, responds upon click
gw.addEventListener("click", clickAction);
```

JSGraphics: Animation

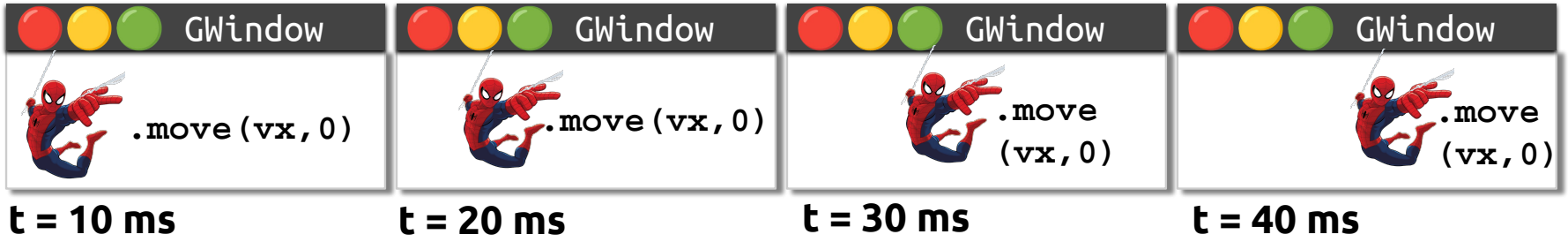


What is animation?

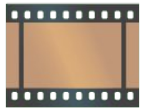
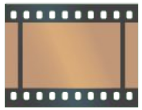
Sequence of frames / images with gradual changes from one to the next, giving the illusion of motion

How we create animation in JS?

Also through frames! Kind of. We use a callback / step function that fires off every few milliseconds to gradually move objects on the screen, e.g., `ball.move(vx,vy)`



JSGraphics: Animation Example

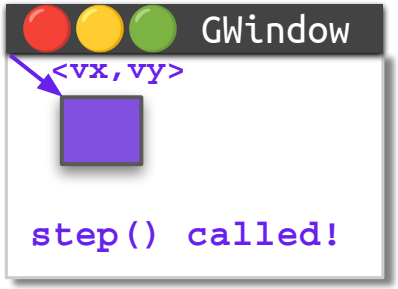



AnimatedSquare.js

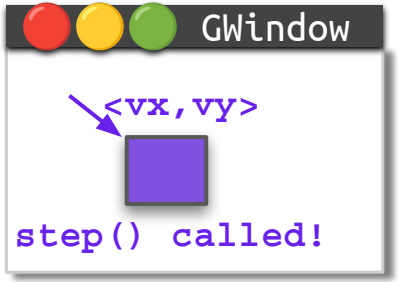
```
function AnimatedSquare() {
  // code to initialize GWindow, GRect square,
  // and x and y velocities dx and dy
  let stepCount = 0;
  let step = function() {
     square.move(dx, dy);
     stepCount++;
     if (stepCount === N_STEPS) clearInterval(timer);
  };
  let timer = setInterval(step, TIME_STEP);
} // say TIME_STEP is 10 milliseconds
```

We can think of ***setInterval*** as a "scheduler" which calls the ***step*** function to move the square every TIME_STEP ms.

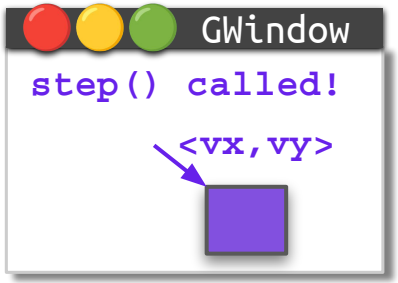
The scheduler is halted when ***clearInterval*** is called 🛑



t=10 ms

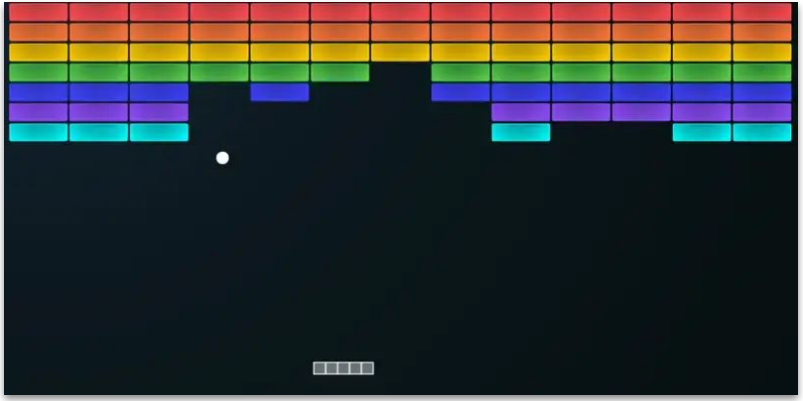


t=20 ms



t=30 ms

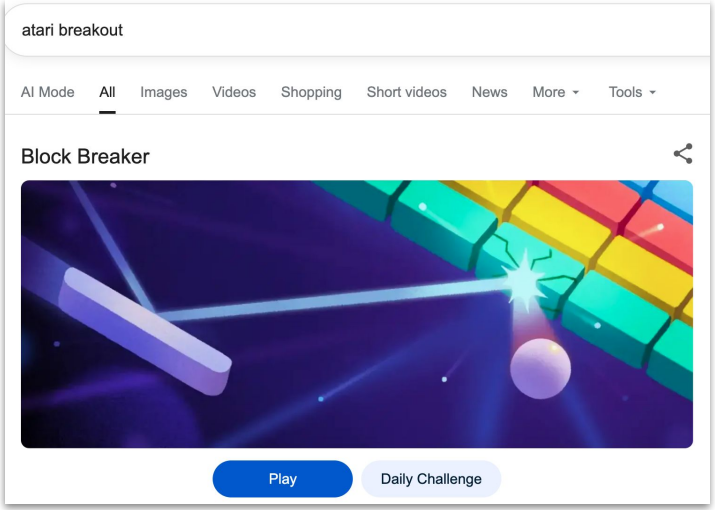
Happy to discuss any questions!



🎮 *Next Up: The Breakout assignment yay!*

Atari Breakout Game (1976)

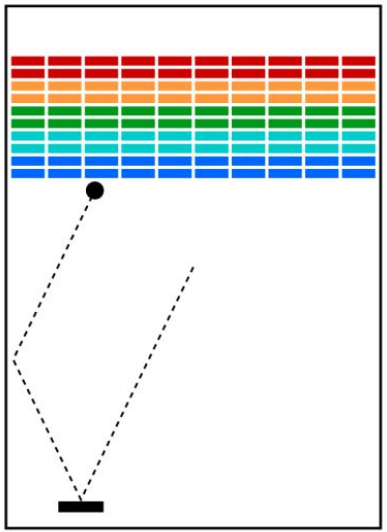
- Classic **arcade video game** – several rows of colorful bricks aligned at the top, which the player has to destroy by **bouncing a ball into them** via a paddle!
- The game was conceptualized by Nolan Bushnell and Steve Bristow, and was physically designed by Steve Wozniak with assistance from Steve Jobs.

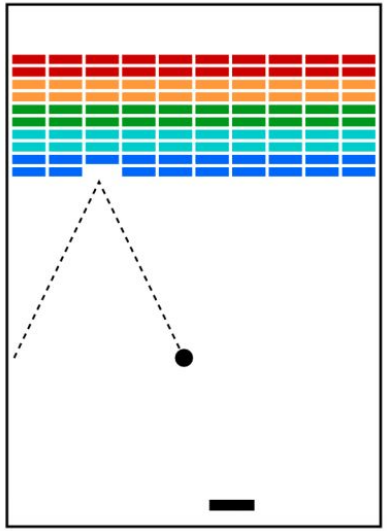


If you want to try playing online for fun / research!

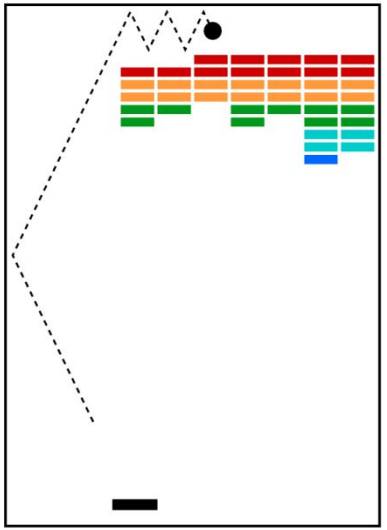
Breakout in CS 106AX

- You'll leverage your terrific graphics knowledge + event-driven programming to **implement this legendary video game** — working is playing! 🕹️
- This assignment is broken up into five smaller, manageable **milestones** .





The magic is really all earthly physics!



Best moment: Breaking out!

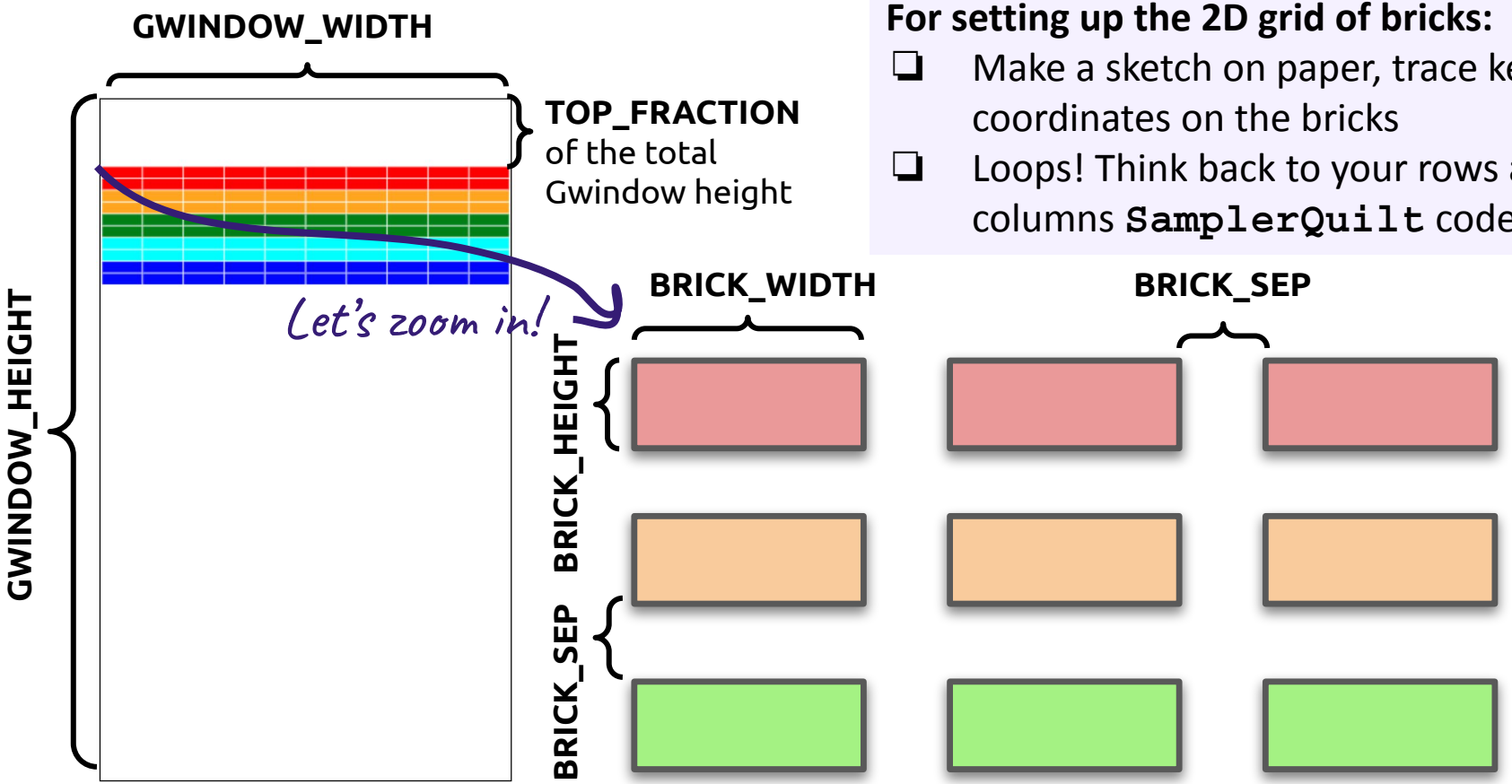
Breakout Constants

These are already defined in the file for you, and we encourage using them!

Breakout.js

```
/* Constants */
const GWINDOW_WIDTH = 360;            /* Width of the graphics window      */
const GWINDOW_HEIGHT = 600;           /* Height of the graphics window     */
const N_ROWS = 10;                    /* Number of brick rows              */
const N_COLS = 10;                    /* Number of brick columns           */
const BRICK_ASPECT_RATIO = 4 / 1;     /* Width to height ratio of a brick  */
const BRICK_TO_BALL_RATIO = 3 / 2;    /* Ratio of brick width to ball size */
const BRICK_TO_PADDLE_RATIO = 2 / 3;  /* Ratio of brick to paddle width    */
const BRICK_SEP = 2;                  /* Separation between bricks         */
const TOP_FRACTION = 0.1;             /* Fraction of window above bricks   */
const BOTTOM_FRACTION = 0.05;         /* Fraction of window below paddle   */
const N_BALLS = 3;                    /* Number of balls in a game         */
const TIME_STEP = 10;                 /* Time step in milliseconds         */
const INITIAL_Y_VELOCITY = 3.0;       /* Starting y velocity downward      */
const MIN_X_VELOCITY = 1.0;           /* Minimum random x velocity         */
const MAX_X_VELOCITY = 3.0;           /* Maximum random x velocity         */

/* Derived constants */
const BRICK_WIDTH = (GWINDOW_WIDTH - (N_COLS + 1) * BRICK_SEP) / N_COLS;
const BRICK_HEIGHT = BRICK_WIDTH / BRICK_ASPECT_RATIO;
const PADDLE_WIDTH = BRICK_WIDTH / BRICK_TO_PADDLE_RATIO;
const PADDLE_HEIGHT = BRICK_HEIGHT / BRICK_TO_PADDLE_RATIO;
const PADDLE_Y = (1 - BOTTOM_FRACTION) * GWINDOW_HEIGHT - PADDLE_HEIGHT;
const BALL_SIZE = BRICK_WIDTH / BRICK_TO_BALL_RATIO;
```

Milestone 1: Setup the Bricks

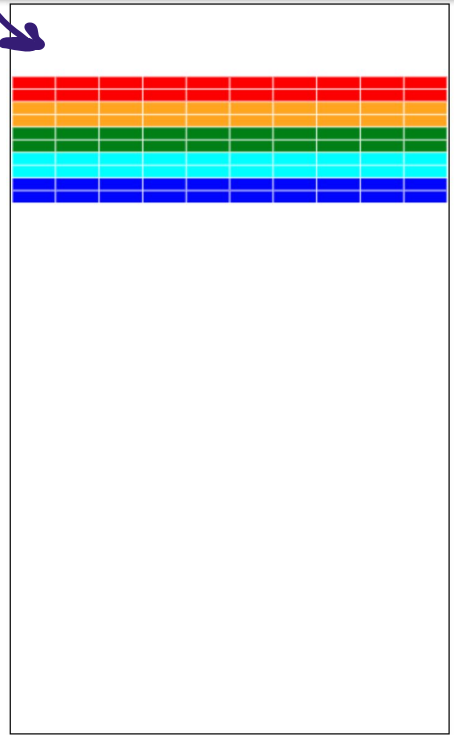


For setting up the 2D grid of bricks:

- ❑ Make a sketch on paper, trace key coordinates on the bricks
- ❑ Loops! Think back to your rows and columns `SamplerQuilt` code!

Milestone 1: Setup the Bricks

GWindow in *Breakout.html* should look like this after **Milestone 1!**



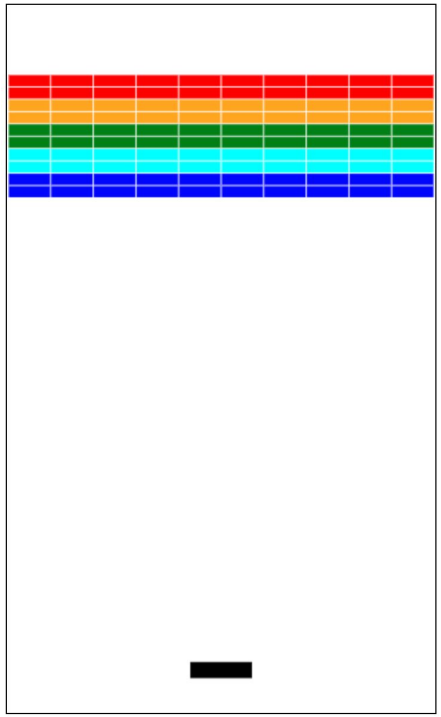
Note: There should `BRICK_SEP` spacing between each brick and each row. This constant is pre-computed so that all bricks fit snugly.

🎨 **Also, to color each pair of rows:**

- We might write a **helper function** that returns a color (or colors in a specific brick) given a row number
- **Colors:** `red, red, orange, orange, green, green, cyan, cyan, blue, blue,` then cycles back to `red`!

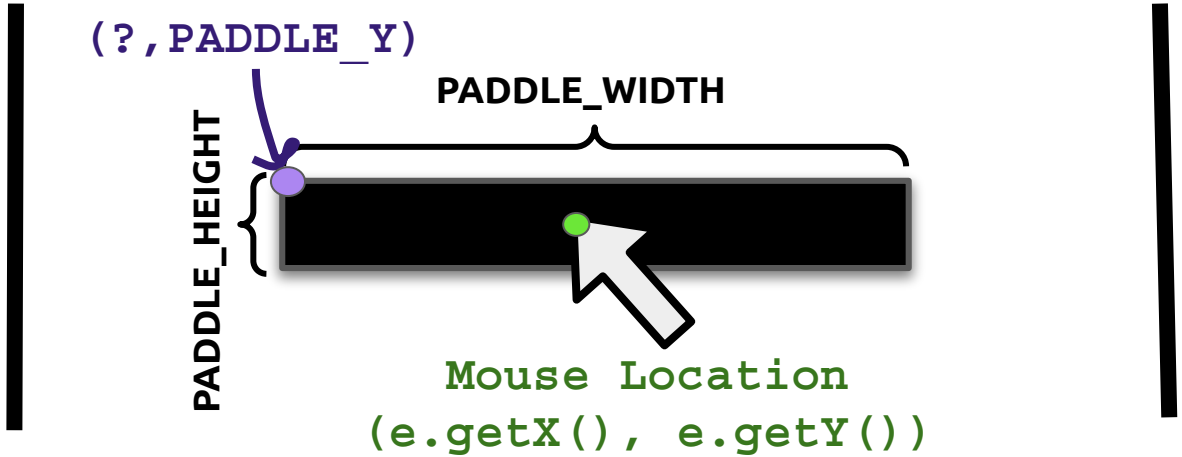
Milestone 2: Create the paddle

Mouse event-listener!



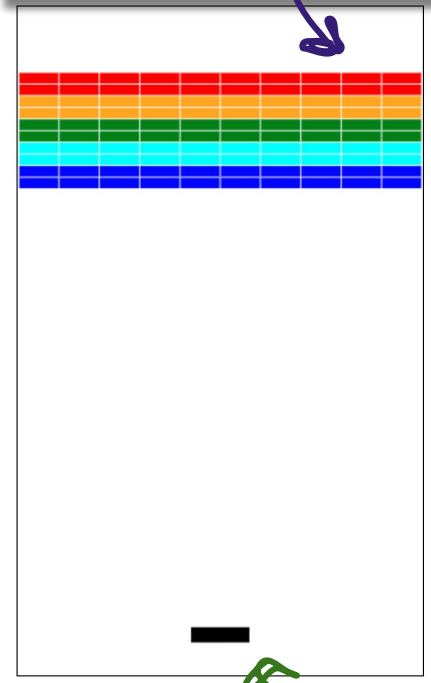
For creating a paddle responsive to *mouse moves*:

- ❏ The paddle should be a simple filled **`GRect`**
- ❏ The paddle's middle should stay anchored to the mouse's location – use **`paddle.setLocation(x,y)`** instead of **`paddle.move`**, it's more convenient :)
- ❏ The paddle has a fixed y-coordinate, **`PADDLE_Y`**



Milestone 2: Create the paddle

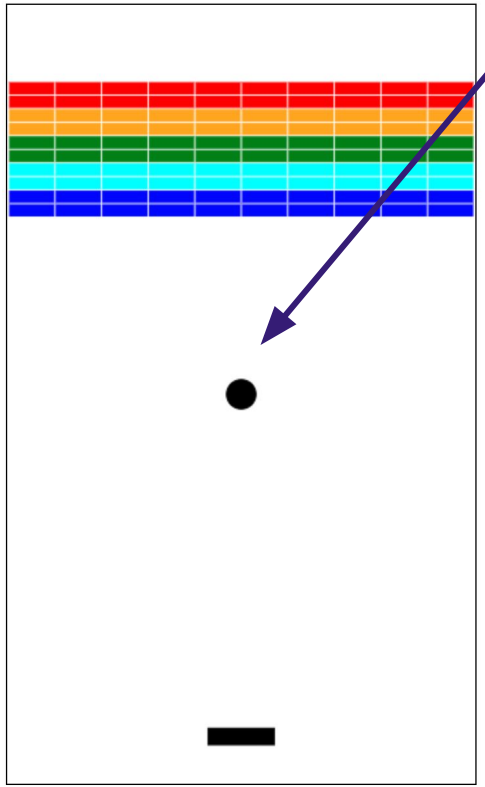
GWindow in *Breakout.html* should look like this after **Milestone 2**!



Note: Even when the mouse moves off screen, the paddle should be forced to stay on the screen *(think about how you might use conditional logic to make this happen!)*

By gliding your mouse, the paddle should slide around **horizontally** within the window, and **no part should leave it!**

Milestone 3: Create a ball and bounce it around 🎾



Draw a ball (filled, circular `GOval`) in the screen's center

Wait for user to **click the screen** (anywhere):

- ❑ Set up a `"click"` event listener 🔊
- ❑ When clicked, the ball animation should begin – but further clicks should be ignored (hint: use a boolean!)

To animate the ball to move:

- ❑ Like the `AnimateSquare` example in class, write a *step* function and use `setInterval()` to call it repeatedly
- ❑ The *step* function should move the ball by `vx` and `vy`

Note: The code to initialize random starting x and y direction velocities `vx` and `vy` is provided in the **assignment handout**!

Milestone 3: Create a ball and bounce it around

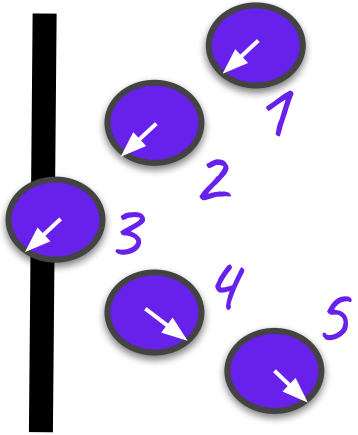
Physics 100 with Ben!
What could go wrong

I was a ~~failed~~ former physics major – so probably take anything I say about physics with a grain of salt lol

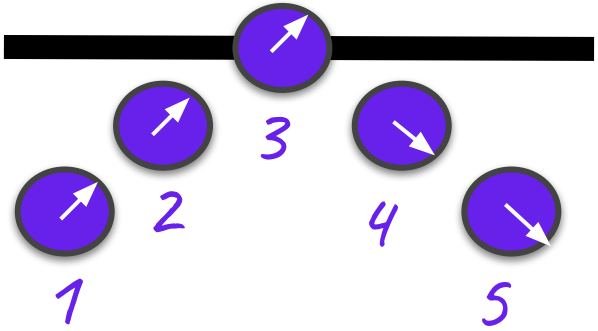
Anyway, let's ball

To bounce the ball around, in the *step* function, you'll want to **check for and handle collisions with walls** – that is, ensure the ball does not leave the window

Left/Right Wall Hit
Reverse the sign of **`vx`**



Top/Bottom Wall Hit
Reverse the sign of **`vy`**



Feel free to ignore acceleration / how gravity really works lol

Is the ball out?

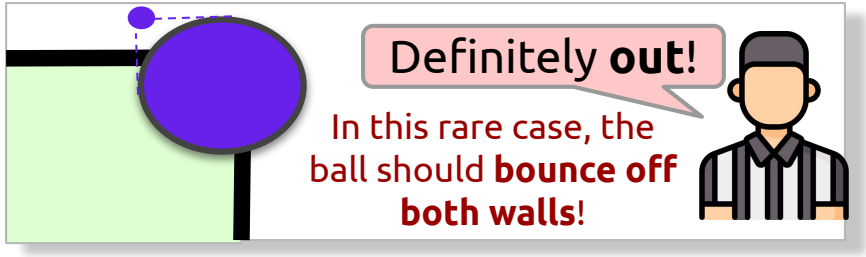
Ball is out if any part exceeds the GWindow's dimensions

- ❑ Ball should bounce off the *right wall* if coordinate of its right edge is greater than the window width
- ❑ Other 3 dimensions are treated similarly

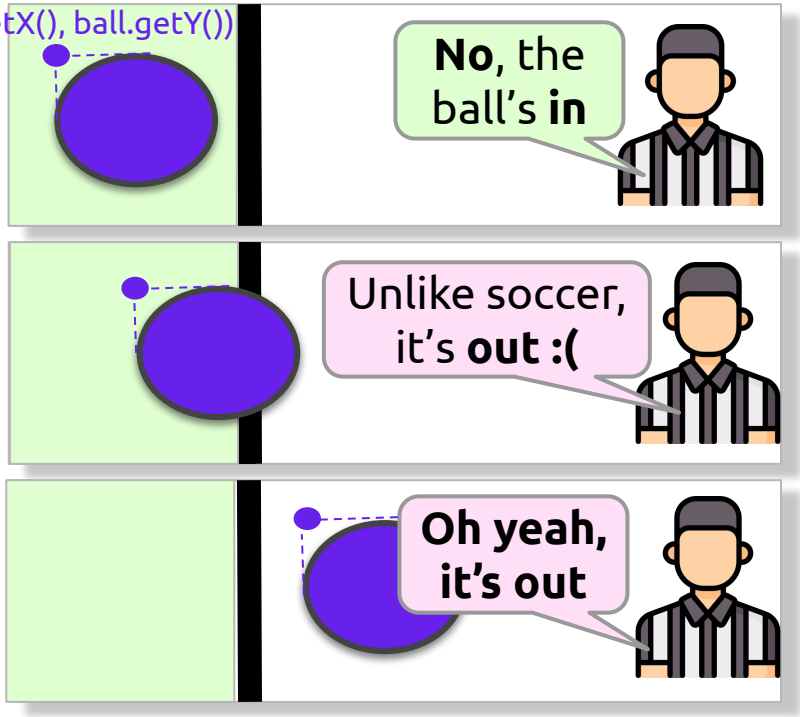
Tip: I recommend sketching each wall's case so you know which x or y coordinates to check!

Spain vs Japan, World Cup 2022

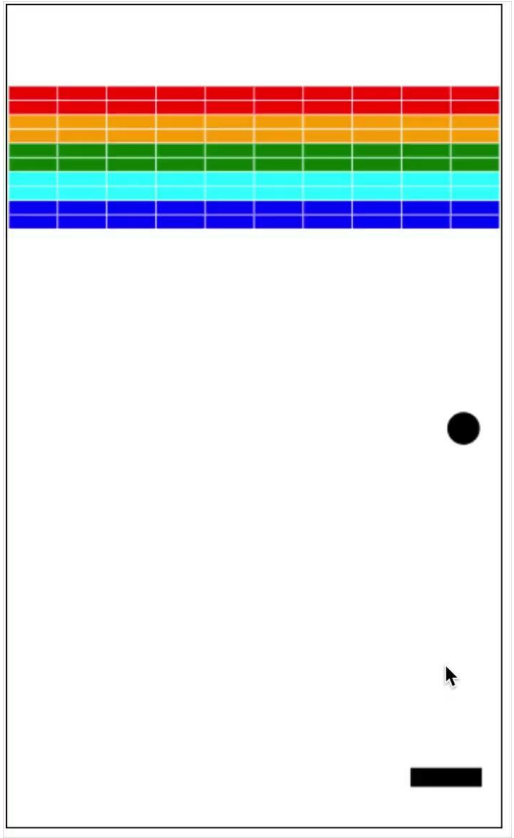
(ball.getX(), ball.getY())



(ball.getX(), ball.getY())



Milestone 3: Create a ball and bounce it around 🎾



After this milestone, **the ball should just bounce around the walls freely** (ignoring the paddle and bricks).

It should be kinda like the bouncing DVD logo on old TVs.

Milestone 4: Checking for object collisions

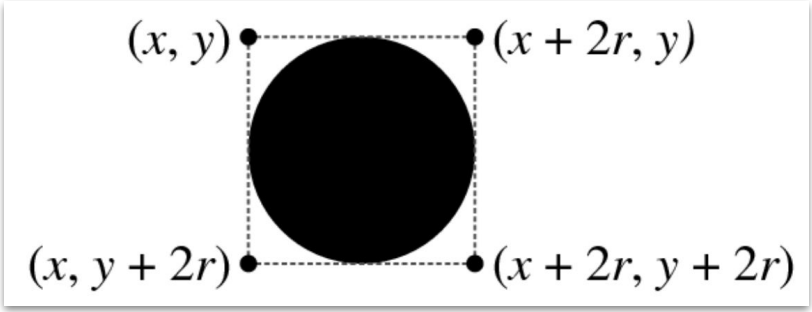
Now, we'll want to check for collisions with on-screen objects, e.g., a brick, or paddle.

- We can use the **`gw.getElementAt(x,y)`** method, which returns the GObject covering a particular point (or **`null`** if there's no object there)

"Geometry" with math major Ben!

A circle has 4 corners, apparently.

Proof is left as an exercise to reader



Breakout.js

```
function getCollidingObject(...) {
   // should have access to gw, ball
   // should return object the ball
collides with (checking its 4
corners), or NULL if no collisions
}
```

But a ball is more than just a single pixel!

- So we model it as the constellation or set of **its 4 corners – a collision with any corner is a collision with the ball.**
- We should write a helper function **`getCollidingObject()`**

Milestone 4: Checking for object collisions

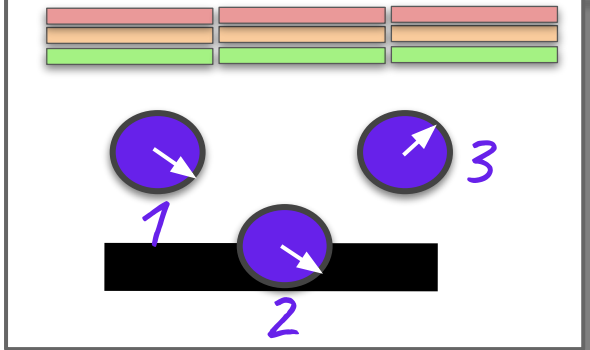
In our animation *step* function, we should call **`getCollidingObject`** to check if the ball is colliding with an onscreen **`GObject`**: must be a paddle or brick.

How do we know if it's a paddle or a brick?

`GObjects` are still variables, so
`if (collidingObject === paddle){ … }`
`else { // it has to be a brick! }`

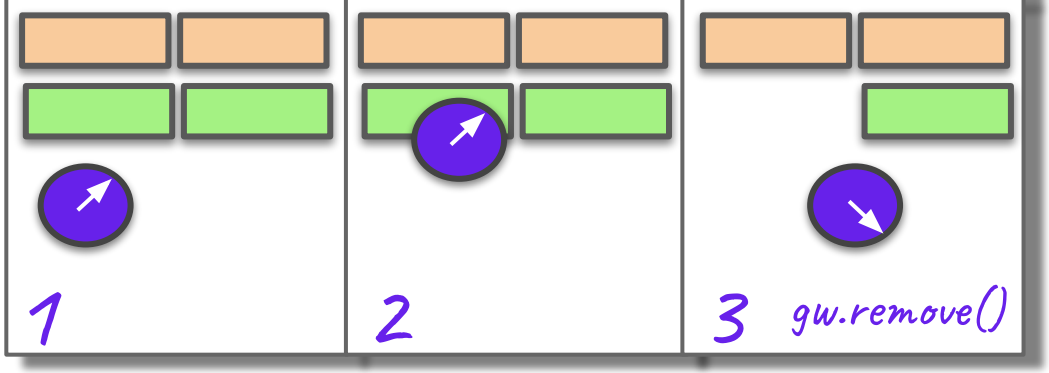
Colliding with Paddle

*Change **`vy`** so ball goes up*



Colliding with Brick

*Reverse the sign of **`vy`**, remove brick from screen*



Milestone 4: Ball stuck to paddle edge case

One tricky bug you might witness is the **“sticky paddle”**.

- When the ball comes down, try swiping the paddle at it quickly and very horizontally, almost like a tennis swerve.
- Ball may “stick” to the paddle, **repeatedly bouncing up / down**

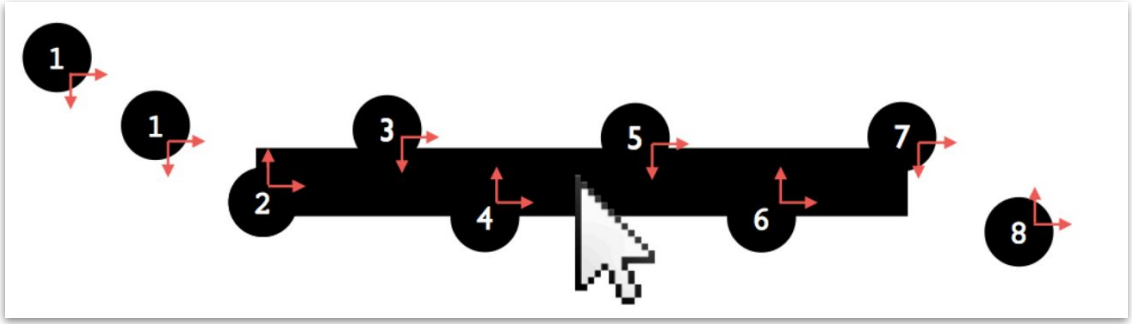
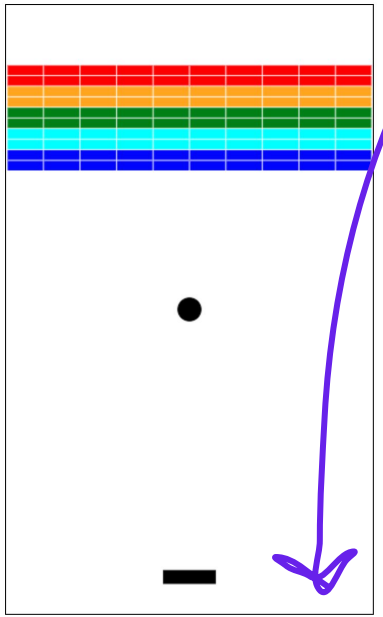


Image from Nick Troccoli’s CS106A YEAH slides back in 2014

Hint: Think about what direction you (always) want the ball to go **when hitting the paddle**, and how you might force the ball to move in that direction

Milestone 5: Getting to the finish line!



When the ball hits the **bottom of the screen**, instead of bouncing up, we should:

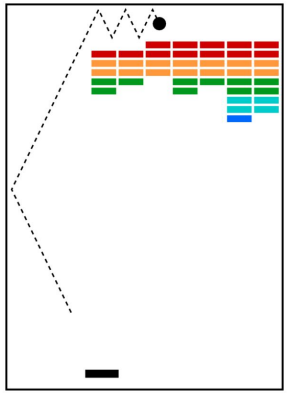
- ❑ Stop the animation / ball from moving
- ❑ Reset the ball, wait for user to click to start next turn

User has **3 "lives" / balls** in total.

We should end the animation entirely when:

- ❑ The user is out of lives / balls (`GAME OVER`),
- ❑ Or when all bricks are deleted (`GAME WIN`)

Thus, we should track of **# of lives**, and we need a way to **recognize when all bricks are gone** 🤔.



Tip: To test the win condition quickly, you can change `N_ROWS = 1` at the top lol

Final Tips and Recommendations

- ❑ Write and test your code incrementally
  - ❑ **Set a milestone schedule**, knocking them down over the course of the week
  - ❑ Make sure each part is working before moving completely on to next phase
- ❑ **Draw plenty of sketches** of Breakout! screen, ball collision scenarios; add comments in code if that helps organize the different cases
- ❑ Come see us in **office hours** if you have questions, get stuck, or want testing tips!
  - ❑ Test and test! It's **playing a video game** :)
  - ❑ Consequently, don't forget to have fun!

Sample Schedule

Sun: Read handout, pull up to YEAH hours! Work on (1), (2)

Mon: Work on (2), (3)

Tues: Math 51 grind 😭

Wed: Work on (4), sweet treat as a reward

Thurs: Work on (5), pull up to OH for debugging / group emotional support

By Fri: Submit! Relax

Have an awesome week 3! :)

Please feel free to reach out if you have any questions:

📬 Contact – **bbyan@stanford.edu**

it’s not a bad time to exchange contact info, find study partners / groups!