

Assignment #2—Breakout!

This assignment is one of many masterpieces by Eric Roberts.

Due: Monday, October 16th, 5:00 P.M.

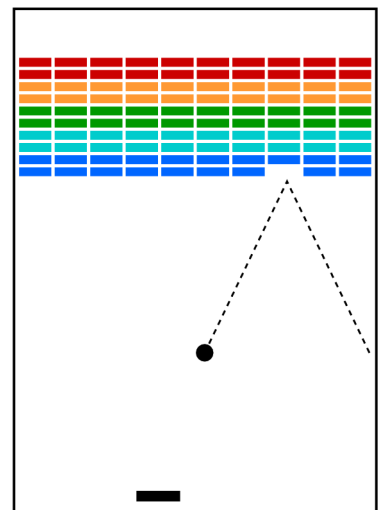
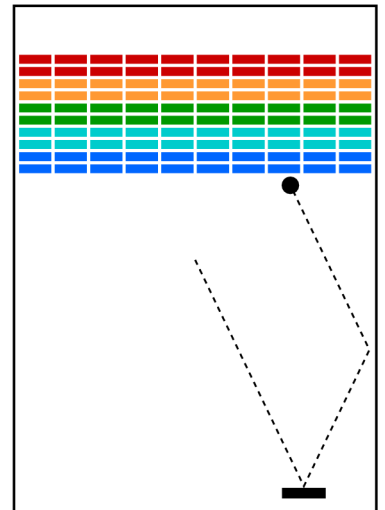
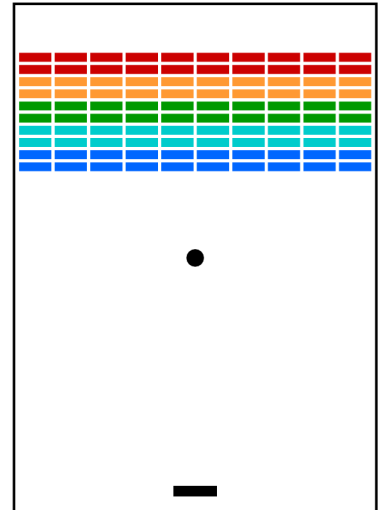
Your job for this assignment is to implement the classic arcade game of Breakout, which was invented by Steve Wozniak before he founded Apple with Steve Jobs. It is a large assignment, but well within your capabilities, provided you break the problem up into manageable pieces. The decomposition is discussed in this handout, and there are several suggestions for staying on top of things in the "Strategy and tactics" section later in this handout.

The Breakout Game

In Breakout, the initial configuration of the world appears as shown on the right. The colored rectangles in the top part of the screen are bricks, and the slightly larger rectangle at the bottom is the paddle. The paddle is in a fixed position in the vertical dimension but moves back and forth across the screen along with the mouse until it reaches the edges of the window.

A complete game consists of three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. That ball bounces off the paddle and the walls of the world, in accordance with the physical principle generally expressed as "the angle of incidence equals the angle of reflection" (which turns out to be very easy to implement as discussed later in this handout). Thus, after two bounces—one off the paddle and one off the right wall—the ball might have the trajectory shown in the second diagram. (Note that the dotted line is there only to show the ball's path and does not actually appear on the screen.)

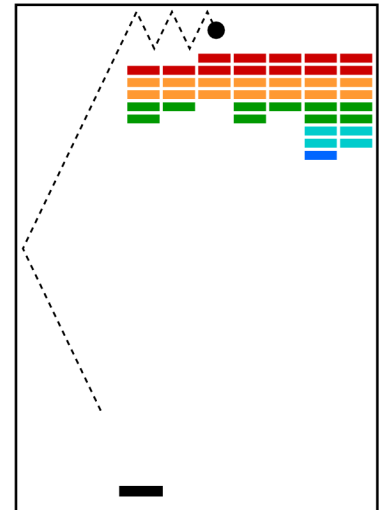
As you can see from the second diagram, the ball is about to collide with one of the bricks on the bottom row. When that happens, the ball bounces just as it does on any other collision, but the brick disappears. The third diagram shows what the game looks like after that collision and after the player has moved the paddle to put it in line with the oncoming ball.



The play on a turn continues in this way until one of two conditions occurs:

1. The ball hits the lower wall, which means that the player must have missed it. In this case, the turn ends, and the next ball is served if the player has any turns left. If not, the game ends in a loss.
2. The last brick is eliminated. In this case, the player wins, and the game ends immediately.

After all the bricks in a particular column have been cleared, a path will open to the top wall. When this delightful situation occurs, the ball will often bounce back and forth several times between the top wall and the upper line of bricks without the user ever having to worry about hitting the ball with the paddle. This condition is a reward for "breaking out" and gives meaning to the name of the game. The diagram on the right shows the situation shortly after the first ball has broken through the wall. That ball will go on to clear several more bricks before it comes back down the open channel.



It is important to note that, even though breaking out is a very exciting part of the player's experience, you don't have to do anything special in your program to make it happen. The game is simply operating by the same rules it always follows: bouncing off walls, clearing bricks, and otherwise obeying the laws of physics.

The starter files

The starter project for this assignment has a little more in it than Assignment 1's did, but none of the important parts of the program are included. The contents of the initial `Breakout.js` file appear in Figure 1. This file takes care of the following details:

- It defines the constants that control the game parameters, such as the dimensions of the various objects. Your code should use these constants internally so that changing them in your file changes the behavior of your program accordingly. Note that some of these constants are specified directly but that others are derived from constants specified earlier.
- It includes a skeleton for the `Breakout` function. Your job is to add the code for the Breakout game along with the definition of helper and callback functions, all of which are easiest to define as local functions inside `Breakout` so that they can share its variables.

The companion `breakout.html` file is already configured to reference a shared copy of our graphics library (at <https://cs106ax.stanford.edu/jslib/JSGraphics.js>), our random number library (<https://cs106ax.stanford.edu/jslib/RandomLib.js>), and your `Breakout.js` file.

Success in this assignment will depend on breaking up the problem into manageable pieces and getting each one working before you move on to the next. The next few sections describe a reasonable staged approach to the problem.

Figure 1. The starter file for the Breakout game

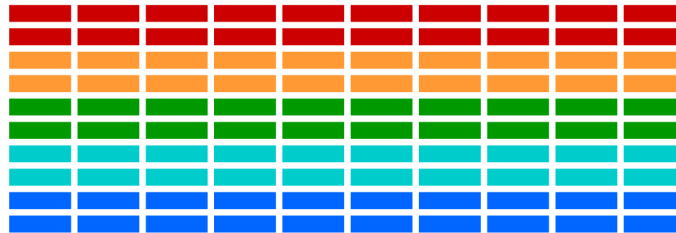
```

Breakout.js — ~/Desktop/assign-2
Breakout.js
1  /*
2  * File: Breakout.js
3  * -----
4  * This program implements the Breakout game.
5  */
6  "use strict";
7
8  /* Constants */
9  const GWINDOW_WIDTH = 360;           /* Width of the graphics window */
10 const GWINDOW_HEIGHT = 600;         /* Height of the graphics window */
11 const N_ROWS = 10;                  /* Number of brick rows */
12 const N_COLS = 10;                  /* Number of brick columns */
13 const BRICK_ASPECT_RATIO = 4 / 1;   /* Width to height ratio of a brick */
14 const BRICK_TO_BALL_RATIO = 3 / 2;  /* Ratio of brick width to ball size */
15 const BRICK_TO_PADDLE_RATIO = 2 / 3; /* Ratio of brick to paddle width */
16 const BRICK_SEP = 2;                /* Separation between bricks */
17 const TOP_FRACTION = 0.1;           /* Fraction of window above bricks */
18 const BOTTOM_FRACTION = 0.05;       /* Fraction of window below paddle */
19 const N_BALLS = 3;                  /* Number of balls in a game */
20 const TIME_STEP = 10;               /* Time step in milliseconds */
21 const INITIAL_Y_VELOCITY = 3.0;     /* Starting y velocity downward */
22 const MIN_X_VELOCITY = 1.0;         /* Minimum random x velocity */
23 const MAX_X_VELOCITY = 3.0;         /* Maximum random x velocity */
24
25 /* Derived constants */
26 const BRICK_WIDTH = (GWINDOW_WIDTH - (N_COLS + 1) * BRICK_SEP) / N_COLS;
27 const BRICK_HEIGHT = BRICK_WIDTH / BRICK_ASPECT_RATIO;
28 const PADDLE_WIDTH = BRICK_WIDTH / BRICK_TO_PADDLE_RATIO;
29 const PADDLE_HEIGHT = BRICK_HEIGHT / BRICK_TO_PADDLE_RATIO;
30 const PADDLE_Y = (1 - BOTTOM_FRACTION) * GWINDOW_HEIGHT - PADDLE_HEIGHT;
31 const BALL_SIZE = BRICK_WIDTH / BRICK_TO_BALL_RATIO;
32
33 /* Main program */
34
35 function Breakout() {
36     // You fill this in along with any helper and callback functions.
37 }
~/Desktop/assign-2/Breakout.js* 40:1  LF  UTF-8  JavaScript  master  No remote  GitHub  Git (

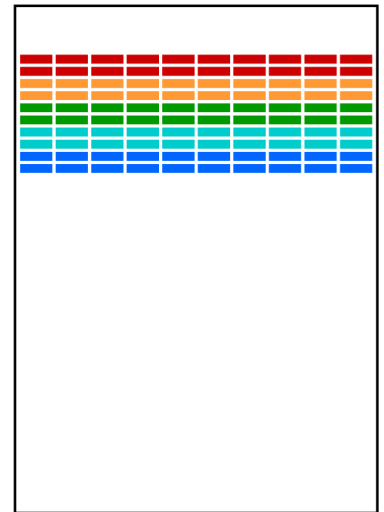
```

Milestone 1: Set up the bricks

Before you start playing the game, you have to set up the various components. Thus, it makes sense to call a function to set up the game before you create the timer that launches the animation. An important part of the setup consists of creating the rows of bricks at the top of the graphics window, which look like this:



The number, dimensions, and spacing of the bricks are specified using constants in the starter file, as is the distance from the top of the window to the first line of bricks. The only value you need to compute is the x coordinate of the first column, which should be chosen so that the bricks are centered in the window, with the leftover space divided equally on the left and right sides. The colors of the bricks remain constant for two rows and run in the following rainbow-like sequence: "Red", "Orange", "Green", "Cyan" and "Blue". If you change the definition of `N_ROWS` in the starter file, any additional rows should simply repeat this sequence, starting over from "Red".



Milestone 2—Create the paddle

The next step is to create the paddle. At one level, this is considerably easier than creating the bricks. There is only one paddle, which is a filled `GRect`. You even know its position relative to the bottom of the window.

The challenge comes with creating a paddle that tracks the mouse. The technique is similar to that discussed in Chapter 6 for dragging an object around within the window. Here, however, you only need to pay attention to the x coordinate of the mouse because the y position of the paddle is fixed. An additional wrinkle is that you should not let the paddle move off the edge of the window. Thus, you'll have to check to see whether the x coordinate of the mouse would make the paddle extend beyond the boundary and, if necessary, change that x coordinate to ensure that the entire paddle is visible.

Milestone 3—Create a ball and get it to bounce off the walls

At one level, creating the ball is easy, given that it's just a filled `GOval`. The interesting part lies in getting it to move and bounce appropriately. You are now past the "setup" stage and into the "play" phase of the game. To start, create a ball and put it in the center of the window. As you do so, keep in mind that the coordinates of the `GOval` do not specify the location of the center of the ball but rather its upper left corner. The math is not any more difficult, just less intuitive.

The program needs to keep track of the velocity of the ball, which consists of two

separate components—`vx` and `vy` are the names used in this handout—which you will presumably declare as local variables inside `Breakout`.

The values of the `vx` and `vy` variables represent the change in position that occurs on each time step. Initially, the ball should be heading downward, and you might try a starting value of 3.0 for `vy`, which is included among the constants as `INITIAL_Y_VELOCITY`. The game would be boring if every ball took the same course, so you should choose the `vx` component randomly. If you use the `RandomLib.js` library discussed during last week's lectures, the best way to set the x component of the velocity is:

```
vx = randomReal(MIN_X_VELOCITY, MAX_X_VELOCITY);
if (randomChance()) vx = -vx;
```

This code sets `vx` to be a random real number in the range 1.0 to 3.0 (given the provided definitions of the constants) and then makes it negative half the time. This strategy works much better than calling `randomReal(-MAX_X_VELOCITY, MAX_X_VELOCITY)`, which might generate a ball going straight down. That would make life too easy for the player.

You then need to get the ball moving by creating a timer that fires every `TIME_STEP` milliseconds and updates the position of the ball each time by moving it `vx` pixels in the x direction and `vy` pixels in the y direction. The model to use for this part of the program is the Chapter 6's `AnimatedSquare.js` program, which we covered in lecture. To give the user a chance to get ready, you want the ball to start moving only after the user clicks the mouse. You therefore need to listen for a `"click"` event that itself starts a timer.

Once you've gotten the ball moving, your next challenge is to get it to bounce around the world, ignoring—at least for the moment—the paddle and the bricks entirely. To do so, you need to check to see if the coordinates of the ball have moved outside the window. Thus, to see if the ball has bounced off the right wall, you need to see whether the coordinate of the right edge of the ball has become greater than the width of the window; the other three directions are treated similarly. For now, have the ball bounce off the bottom wall so that you can watch it make its path around the world. You can change that test later so that hitting the bottom wall signals the end of a turn.

Computing what happens after a bounce is extremely simple. If a ball bounces off the top or bottom wall, all you need to do is reverse the sign of `vy`. Symmetrically, bounces off the side walls simply reverse the sign of `vx`.

Milestone 4—Checking for collisions

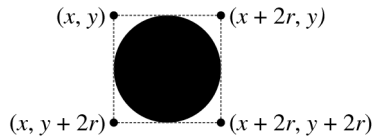
In order to make `Breakout` a real game, you have to be able to tell whether the ball is colliding with another object in the window. As scientists often do, it helps to begin by making a simplifying assumption and then relaxing that assumption later. Suppose the ball were a single point rather than a circle. In that case, how could you tell whether it had collided with another object?

If you look in Chapter 6 at the methods that are defined for the `GWindow` class, you will discover that there is a method `getElementAt(x, y)` that takes a location in the window and returns the graphical object at that location, if any. If there are no graphical objects that cover that position, `getElementAt` returns `null`. If there is more than one,

`getElementAt` always chooses the one closest to the top of the stacking order, which is the one that appears to be in front on the display.

So far, so good. But, unfortunately, the ball is not a single point. It occupies physical area and therefore may collide with something on the screen even though its center does not. The easiest thing to do—which is in fact typical of the simplifying assumptions made in real computer games—is to check a few carefully chosen points on the outside of the ball and see whether any of those points has collided with anything. As soon as you find something at one of those points, you can declare that the ball has collided with that object.

In your implementation, the easiest thing to do is to check the four corner points on the square in which the ball is inscribed. Remember that a `GOval` is defined in terms of its bounding rectangle, so that if the upper left corner of the ball is at the point (x, y) , the other corners will be at the locations shown in this diagram:



These points have the advantage of being outside the ball—which means that `getElementAt` can't return the ball itself—but nonetheless close enough to make it appear that collisions have occurred. Thus, for each of these four points, you need to:

1. Call `getElementAt` on that location to see whether anything is there.
2. If the value you get back is not `null`, then you need look no further and can take that value as the `GObject` with which the collision occurred.
3. If `getElementAt` returns `null` for a particular corner, go on and try the next corner.
4. If you get through all four corners without finding a collision, no collision exists.

It would be very useful to decompose this part of the program into a separate function called `getCollidingObject` that returns the `GObject` colliding with the ball, if any, and `null` otherwise. You could then use it in a declaration like

```
let collider = getCollidingObject();
```

which assigns that value to a variable called `collider`.

From here, the only remaining thing you need to do is decide what to do when a collision occurs. There are only two possibilities. First, the object you get back might be the paddle, which you can test simply by checking whether `collider` is equal to the `GObject` for the paddle, which you've presumably stored in a local variable, defined using a `let` statement—as part of Stage 2.

If the colliding object is the paddle, you need to bounce the ball so that it starts traveling up. If it isn't the paddle, then it must be a brick, since those are the only other objects in the world. Once again, the ball should reverse its course in the y direction, but you also need to take the brick away. To do so, all you need to do is remove it from the screen by

calling the `remove` method on the `GWindow` object.

Stage 5—Finishing up

If you've gotten to here, you've done all the hard parts. There are, however, a few more details you need to manage if you're to arrive at a working program:

- You have to take care of the case when the ball hits the bottom wall. In the prototype you've been building, the ball just bounces off this wall like all the others, but that makes the game difficult to lose. You'll need to modify your program structure so that it tests for hitting the bottom wall and clears the interval timer.
- You have to give the user three chances to remove the bricks. When the ball falls through the bottom of the window, you should remove it from the window, create a new ball just as you did at the beginning, and then wait for another click to start things up. If the user has already had three chances, the game is over.
- You have to check for the other terminating condition, which is hitting the last brick. How do you know when you've done so? Although there are other ways to do it, one of the easiest is to have your program keep track of the number of bricks remaining. Every time you hit one, subtract one from that counter. When the count reaches zero, you've clearly won!
- You've got to test your program to see that it works. Play for a while and make sure that as many parts of it as you can check are working. If you think everything is working, here is something to try: Just before the ball is going to pass the paddle level, move the paddle quickly so that the paddle collides with the ball rather than vice-versa. Does everything still work, or does your ball seem to get tethered to the paddle? If you get this error, try to understand why it occurs and how you might fix it.

Strategy and tactics

Here are some survival hints for this assignment:

- *Start as soon as possible.* This assignment is due a week from today, which will be here before you know it. If you wait until the day before this assignment is due, you will have a very hard time getting it all together.
- *Implement the program in stages, as described in this handout.* Don't try to get everything working all at once. Implement the various pieces of the project one at a time and make sure that each one is working before you move on to the next phase.
- *Set up a milestone schedule.* Work to complete the suggested milestones over the course of the week instead of waiting until Thursday. If you do, you'll have lots of time for the more interesting parts of the assignment and for implementing extensions.
- *Don't try to extend the program until you get the basic functionality working.* The following section describes several ways in which you could extend the implementation. Several of them are lots of fun. Don't start them, however, until the basic assignment is working. If you add extensions too early, you'll find that the debugging process gets really difficult.

Possible extensions

This assignment is perfect for those of you who are looking for + or ++ scores because there are so many possible extensions. Here are a few:

- *Add messages.* The sample application on the web site waits for the user to click the mouse before serving each ball and announces whether the player has won or lost at the end of the game. These are just `GLabel` objects (as described in Chapter 4) that you can add and remove at the appropriate time.
- *Improve the user control over bounces.* The program gets rather boring if the only thing the player has to do is hit the ball. It is far more interesting if the player can control the ball by hitting it at different parts of the paddle. In the old arcade game, the ball would bounce in both the x and y directions if you hit it on the edge of the paddle from which the ball was coming.
- *Add in the "kicker".* The arcade version of Breakout lured you in by starting off slowly. But, as soon as you thought you were getting the hang of things, the program sped up, making life just a bit more exciting.
- *Keep score.* You could easily keep score, generating points for each brick. In the arcade game, bricks were more valuable higher up in the array, so that you got more points for red bricks than cyan bricks.
- *Use your imagination.* What else have you always wanted a game like this to do?