Project 1: Breakout!

Your job in this programming project is to write the classic arcade game of Breakout, a mainstay in arcades and other community centers prior to the emergence of personal computing at population level. It is a large project, but well within your capabilities, as long as 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 the handout. As per usual, you will submit this through Github Classroom, and the Gitub link will be distributed via Discord. To help you navigate the project, I've included a table of contents below, but I highly recommend you read everything as you work through the project.

Contents

1 The Breakout game

In Breakout, the initial configuration of the world appears in the leftmost image of Figure 1. 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 direction, 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 later discussed). Thus, after two bounces–one off the paddle and one off the left wall–the ball might have the trajectory shown in the second image in Figure 1. (Note that the dotted line is only there to show the ball's path and does not actually appear on the screen.)

As you can see in the second image, the ball is soon to collide with one of the bricks on the bottom row. When that happens, the ball bounces just at it does on any other collision, but the brick disappears. The third image 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

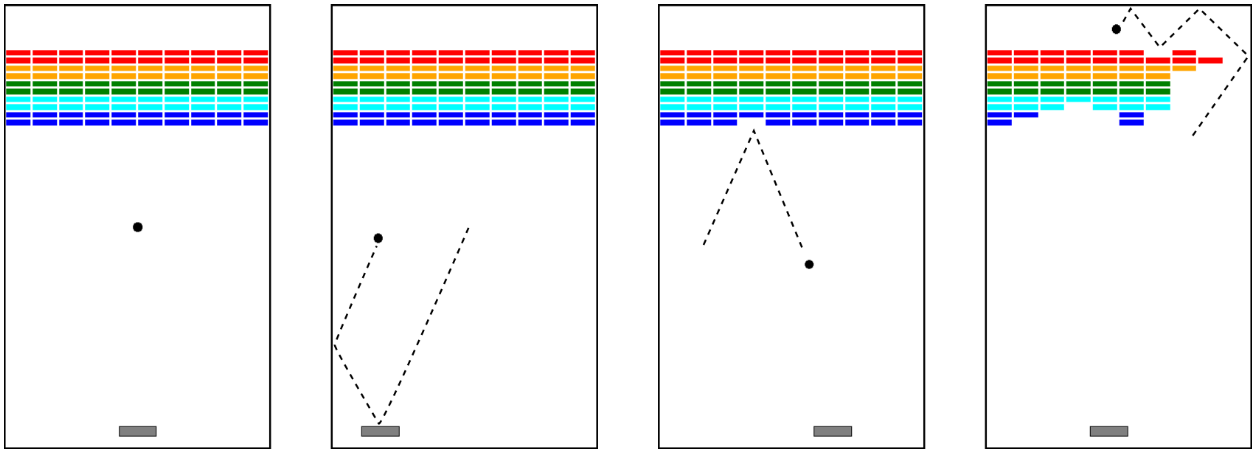

Figure 1: Example of the game Breakout. The ball starts in the center of the screen and proceeds to bounce off both the paddle and the walls. The images here proceed in chronological fashion from left to right.

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 with the paddle. 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 for the player.

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 rightmost image in Figure 1 shows the situation shortly after the 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 simple operating by the same rules it always applies: bouncing off walls, clearing bricks, and otherwise obeying the laws of physics.

2 The starter file

The starter file for this project contains the initial version of the `Breakout.py` file. This file takes care of the following details:

- It include the imports you will need in writing the game.
- 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 the program accordingly. Note that some of the these constants are specified directly, but that others are derived from constants specified earlier.
- It includes a skeleton for the `breakout` function that creates the graphics window. Your job is to add the code for the Breakout game along with the definition of any helper and callback functions, all of which are easiest to define as inner functions inside `breakout` so they can share its variables.
- It includes the startup code to invoke the `breakout` function when the `Breakout.py` module is run from the command line.

3 Milestones

Success in this project will depend on decomposing 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.

3.1 Milestone 1 – Set up the bricks

Before you start playing the game, you need to set up the various pieces. Thus, it makes sense to call a function to set up the game before you create the timer that starts the animation. An important part of the setup consists of creating the rows of bricks at the top of the game, which should resemble Figure 2.

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 evenly divided between 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 simple repeat this sequence, starting over from `"Red"`.

This part of the project will rely largely on constructing the proper loops and getting the spacing worked out. Don't be afraid to sketch out distances and dimensions on a sheet of paper to help you get the necessary spacing.

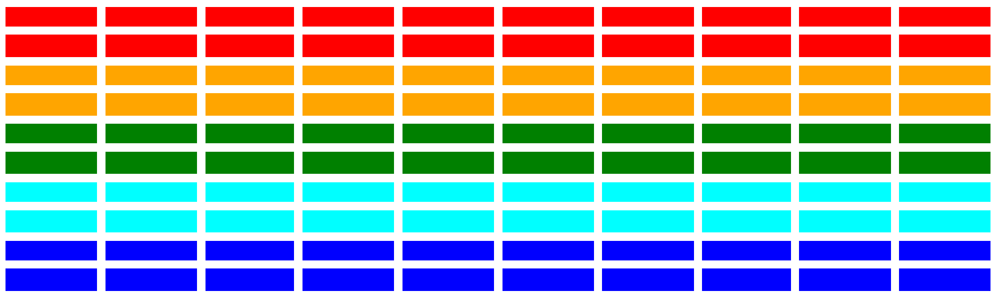

Figure 2: The initial state of the bricks. Colors shift in rows of two as they move up or down the screen.

3.2 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 after all, a single filled `GRect`. You even know its position relative to the bottom of the window.

The challenge in creating the paddle is to make it track the mouse. The technique is similar to that discussed in Chapter 5 and in how you made the eyeballs track the mouse on the most recent Problem Set. Here, however, you only need to pay attention to the x coordinate of the mouse since the y position of the paddle is always fixed. The only 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 window boundary and change that x coordinate if necessary to ensure that the entire paddle is visible in the window at all times.

Of note, this entire part of the program takes fewer than 10 or so lines of code, so it shouldn't take all that long to implement. It is essentially an easier version of what you had to do with the tracking eyeballs.

3.3 Milestone 3 – Create a bouncy ball

On one level, creating the ball is easy, as it is just a filled `GOval`. The interesting part lies in getting it to move and bounce appropriately. To start, create a ball and put it in the center of the window, keeping in mind that the coordinates of the `GOval` do not specify the location of the center of the ball but rather the upper left corner. The math is not any more difficult, but decidedly 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 must be part of the `GState` object created inside the main program. The values of the `vx` and `vy` variables represent the change in the position that occurs on each time step (or the *velocity* in physics parlance). Initially, the ball is heading downward, and you might try a starting value of 3.0 for `vy`, which is included in the constants as `INITIAL_Y_VELOCITY`. The game would be very boring if every ball took the same course, so you should choose the `vx` component randomly using the following code:

```
gs.vx = random.uniform(MIN_X_VELOCITY, MAX_X_VELOCITY)
if random.uniform(0,1) < 0.5:
    gs.vx = -gs.vx
```

This code sets `vx` to be a random real number in the range 1.0 to 3.0 (given the definitions constants) and then makes it negative half the time. This strategy gives better results than just setting it to `random.uniform(-MAX_X_VELOCITY, MAX_X_VELOCITY)`, which might generate a ball going more or less straight down. We can't be making life too easy for the player!

You then need to get the ball moving by creating a timer that triggers 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 problem is likely the `AnimatedSquare.py` program from Chapter 5. But it isn't quite sporting to have the ball start moving the instant that the game starts. To give the player a chance to get ready, what you want to have happen is that the ball starts moving only once the user has clicked the mouse.

There are several strategies you can use to accomplish the goal of waiting for a click before starting the game. One possibility is to start the timer inside the listener for the "click" event. That strategy, however, is difficult to get right. A problem that often shows up is that clicking the mouse again while the ball is moving doubles the speed of the ball, because then there are two timers running simultaneously, each of which advances the ball. A better strategy is to start one timer running at the beginning (just like `AnimatedSquare.py` does), but have the `step` function update the position of the ball only if the ball is moving. You can keep track of whether the ball is moving in a Boolean variable that is set to `True` when a click occurs and is set to `False` initially or if the ball falls through the bottom of the window.

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 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 moving past the bottom wall signifies the end of a turn.

Computing what happens during 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`. Similarly, bouncing off the side walls reverses the sign of `vx`.

3.4 Milestone 4 – Checking for collisions

Now comes the interesting part. In order to make Breakout into 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 review in Chapter 5 the methods that are available for the `GWindow` class, you will be reminded that there is a method `get_element_at(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, `get_element_at` returns the constant `None`. If there is more than one, `get_element_at` 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. In reality, however, the ball is not a single point, which means that its boundary may collide with something even thought its center does not. The easiest thing to do–which is in fact typical of the simplifying assumption made in real computer games–is to check the corner points of the square that encloses the ball. 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 Figure 3.

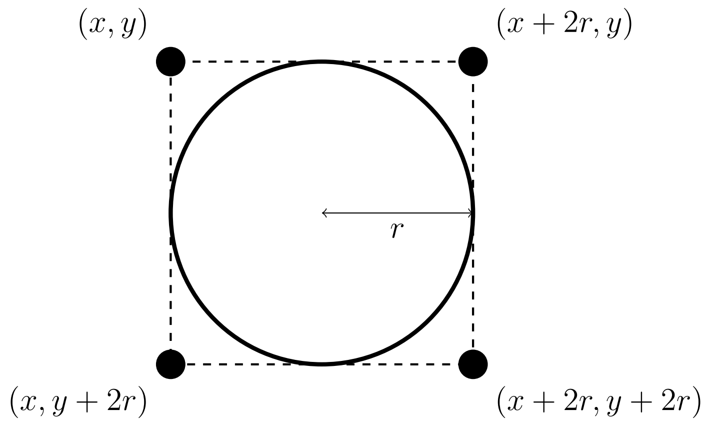

Figure 3: Points at the corners of a `GOval` object located at point (x, y) with radius r.

These points have the advantage of being outside the ball itself–which means that `get_element_at` 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 `get_element_at` on that location to see whether anything is there.
2. If the value you get back is not `None`, then you need to look no further as you have found something and can take that returned object as the `GObject` with which the collision occurred.
3. If `get_element_at` returns `None` for a particular corner, go on and try the next corner.
4. If you get through all four corners without finding a collision, then no collisions exists!

It will be useful to decompose this part of the program into a separate function called `get_colliding_object` that returns the `GObject` colliding with the ball (if any), or `None` otherwise. You can then use it in a statement like

```
collider = get_colliding_object()
```

which assigns the colliding `GObject` 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 store in a local variable as part of Milestone 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, the only other thing it might be is a brick, since those are the only other objects in the world. Once again, you need to cause a bounce in the vertical 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.

3.5 Milestone 5 – Finishing up

If you've gotten here, you've done all the hard parts! There are, however, a few more details you need to take into account.

- 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 impossible to lose. You've got to modify your program structure so that it tests for hitting the bottom wall and stops 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 a 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 must be done!

- 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 "glued" to the paddle? If you get this error, try to understand why it occurs and how you might fix it.

4 Strategy and Tactics

Here are some survival hints for this project:

- *Start as soon as possible!* This assignment is due in a week, which will be here before you know it. If you wait until the day before this project 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.* Consider approaching a milestone a day while being kind to yourself if you get stuck on one. If you keep a solid pace, you'll have more time for the more interesting parts and for implementing potential 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 those are lots of fun. Don't start them, however, until the basic project is working. If you try to add the extensions too early, you'll find that the debugging process gets really difficult.

5 Possible Extensions

There are so many possible extensions. Here are a few, but this list is by no means exhaustive!

- *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 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 some by hitting it at different parts of the paddle. In the old arcade game, the ball would bounce more horizontally if you hit it further from the center of the paddle.

- *Add in a difficulty curve:* 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 broken. In the arcade game, bricks were more valuable higher up in the array, so that you got more points for red bricks than for blue ones.