Nick Bowman, Sonja Johnson-Yu, Kylie Jue
CS 106AP

Assignment #5
July 30, 2019

Assignment #5: Breakout

On-time deadline: 11:59 PM on Wednesday, August 7
Extended deadline: 11:59 PM on Thursday, August 8
This assignment can optionally be done in pairs.

This assignment consists of a pair of warmups and one larger application that you will build in several steps. As in previous assignments, you can download the starter code for this project under the "Assignments" tab on the CS106AP website.

As in the previous two assignments, you will be running all your code from the command line in this assignment. We will not be using doctests in this assignment, as doctests are difficult to implement for programs with graphical output.

This assignment may be done in **pairs** or may be done individually. **You may only pair up with someone in the same section time and location**. If you work as a pair, **comment both members' names** at the top of every `.py` file. Make **only one** assignment submission; do not turn in two copies. If you choose to work in a pair, you should make sure to read this Pair Programming handout before beginning the assignment.

Your job in this assignment is to write the classic arcade game Breakout, which was invented by Steve Wozniak before he founded Apple with Steve Jobs. It is a large assignment, but it's entirely manageable as long as you break the problem up into 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.

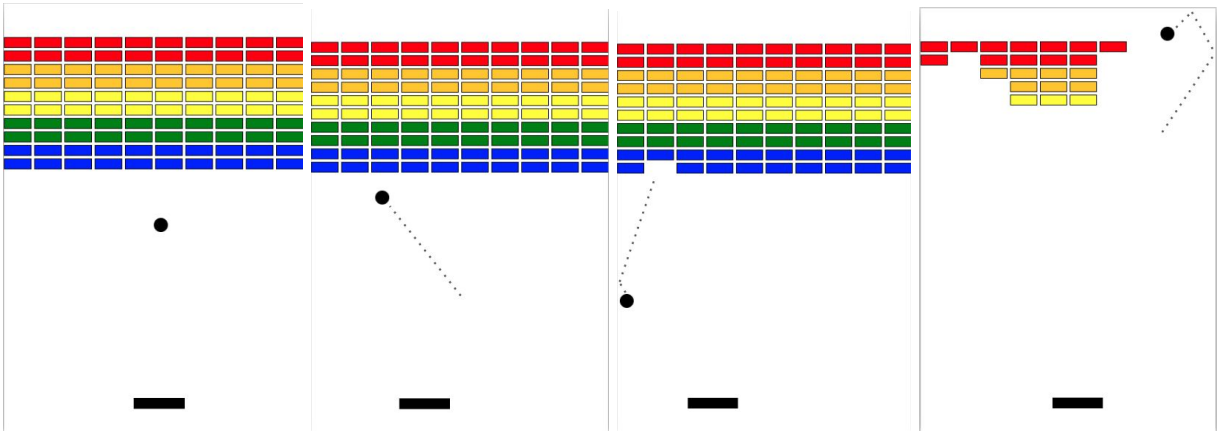

Figure 1: The first image (leftmost) shows the starting state for the game before the user clicks to launch the ball. The two middle pictures show what happens after the ball has bounced off the user's paddle, moves to hit a brick, and then bounces back towards the left wall. The last image (rightmost) shows the "breaking out" phenomenon, as described in "The Breakout game" section of the handout. Note that the lines drawn on each image are only to clarify gameplay and will not be present in the real game itself.

Warmup: Designing breakout

Since this is the first program in which we will not be providing explicit decomposition in the starter code, your warmup for this assignment will be to go through this handout and outline the

BreakoutGraphics class (in **`breakoutgraphics.py`**), as well as how the main file (**`breakout.py`**) will interact with your class interface. In particular, you should aim to answer two of the three class design questions we discussed in class (since the constructor is already given to you):

1. Attributes (instance variables): What subvariables make up your BreakoutGraphics class?
2. Methods: What functions will you need to call on your BreakoutGraphics class?

In order to answer these questions, you should first read through this entire assignment handout and keep track of potential attributes and methods that you might need to implement. While we've provided the constructor for the **`BreakoutGraphics`** class, taking a look at the starter code might also give you some ideas for methods and attributes. Then, synthesize your findings into an outline that will help guide you in your implementation of the entire Breakout program. Put your work in the **`breakout_design.txt`** file included in the starter code.

At minimum, your outline should answer the two questions above. But you should also feel free to include a more extensive plan and/or pseudocode for the program – whatever you feel would be most helpful before jumping into code. Since you will be working in two different files for this assignment, you should also ask yourself, what does the interface for the **`BreakoutGraphics`** class look like, and how will **`breakout.py`** use that interface?

While it can be tempting to complete the warmup after you've finished implementing the assignment (or to go back and edit your answers afterward), **we do not want you to do this**. We will not be grading the warmup on correctness or on how closely your design/pseudocode matches your final project. We will only be evaluating whether or not the **`.txt`** file is present (and has contents), and **we are far more interested in how you actually thought through the planning of the assignment before getting started**.

The Breakout game

In the game Breakout, your goal is to eliminate all of the colorful rows of bricks at the top of the window by using the paddle (the larger black rectangle at the bottom of the window) to bounce a ball against the bricks. Each time the ball hits a brick, it removes that brick from the screen. The game ends when one of two conditions is met:

- All of the bricks have been removed from the screen, and the player wins.
- Three **turns** are completed, and the player loses.

A **turn** begins when the user clicks within the window, and the ball launches from the center of the window toward the bottom of the screen at a random angle. A turn ends when the user misses the ball with the paddle, causing the ball to hit the bottom "wall" of the window. After the first and second turns, the ball gets reset in the middle of the screen so that the player can click to start the next turn.

The initial configuration of the world appears as shown in the first (leftmost) window in Figure 1 on the previous page. You can move the paddle horizontally, but its vertical position (its y-coordinate) is fixed. The paddle will not move beyond the left and right edges of the window.

The ball bounces in accordance with the physical principle generally expressed as "the angle of incidence equals the angle of reflection" (which we'll explain how to implement later in this handout). The second and third windows in Figure 1 show the ball's trajectory before and after bouncing off a brick. As you can see, the way the ball bounces is consistent no matter what object it collides with (wall, paddle, or brick). (**Note that the dotted lines in Figure 1 are there to show the ball's path and won't appear on the screen in your game.**)

The last window in Figure 1 shows the origin of the game's name, "Breakout." After all the bricks in a particular column have been cleared, a path will open to the top wall. When this 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 called "breaking out" and gives meaning to the name of the game. 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 as always: bouncing off the walls, clearing bricks, and otherwise obeying the laws of physics.

The starter files

We have provided two starter files for you in the project folder. The first file, **`breakout.py`**, will contain the **`main()`** function that actually implements the game logic and is responsible for running the animation loop that drives the game. The second file, **`breakoutgraphics.py`**, defines a class called **`BreakoutGraphics`** that will eventually handle all of the important graphical tasks that bring the game to life. Both files already take care of the following details:

- They include the imports you will need for writing the game.
- They define the named constants that control the game parameters, such as the dimensions of the various objects. Your code should use the provided constants within your program so that changing the constants in the files changes the program's behavior accordingly.

The starter code includes both a constructor with various arguments in the **`breakoutgraphics.py`** file and also the creation of an instance of the **`BreakoutGraphics`** class in the **`main()`** function in **`breakout.py`**. You should not modify the arguments provided to the constructor. 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 the following milestones that you should achieve on your way to building the final game:

- Filling in the constructor
- Drawing the bricks
- Connecting the paddle to your mouse movements
- Getting the ball to bounce off the walls
- Checking for collisions between the ball and other objects
- Implementing end conditions and finishing the game

Each milestone will indicate which file it should take place in.

Milestone 1: Fill in the BreakoutGraphics constructor (`breakoutgraphics.py`)

The function definition for the constructor for the BreakoutGraphics class has been provided to you inside **`breakoutgraphics.py`**. We've also already created a GWindow object of the appropriate size and set it to the instance variable **`self.window`**. However, it's your job to fill in the rest of the constructor based on the guiding comments inside the method. The goal of the constructor is to draw all of the static components that make up the game. Do not worry about animating the movement of any of the components, as this will be done in later milestones.

You should first:

- Create a paddle at the correct starting location using the given constants (**`PADDLE_OFFSET`** is the distance from the top of the paddle to the bottom of the window)

- Create a ball that whose center is in the middle of the window
- Set default values for the ball's initial x and y velocities (referred to as **`vx`** and **`vy`** throughout the rest of this handout)
- Call the appropriate campy mouse listener functions to set up any mouse listeners that we might need (the functions you pass in can be placeholders for now since you haven't written them yet)

The last step (drawing the bricks) will take place in Milestone 2, but you should call the method for drawing the bricks inside your constructor.

Milestone 2: Draw the bricks (`breakoutgraphics.py`)

In Milestone 1, we started setting up the various graphical components that make up the game. Before you start playing, you have to draw the final missing component: the rows of bricks at the top of the game. These are shown in Figure 2.

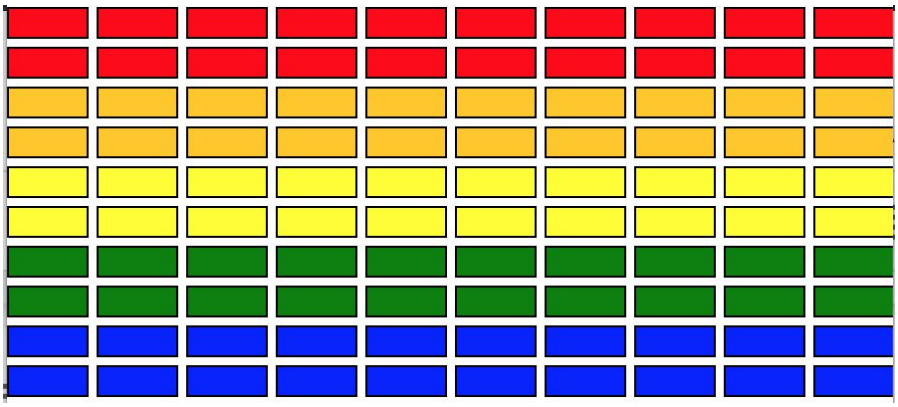

Figure 2: The 10 rows of colored bricks that exist at the top of the Breakout window

The number, dimensions, and spacing of the bricks are specified using named constants in the **`breakoutgraphics.py`** starter file. The **`BRICK_OFFSET`** constant is the distance from the top of the window to the top edge of the first line of bricks. The left side of the very first brick in a row should start at the very left edge of the window, and the remaining bricks should be drawn at appropriate locations to produce the pattern detailed above. If done correctly, the right side of the very last brick in a row will be even with the right edge of the window. The color of the bricks remain constant for two rows and run in the following rainbow-like sequence: **`RED`**, **`ORANGE`**, **`YELLOW`**, **`GREEN`**, **`BLUE`**, which is defined in a provided constant in **`breakoutgraphics.py`**.

Milestone 3: Connect the paddle to the mouse (`breakoutgraphics.py`)

The next step is to bring the paddle to life. Remember that we initially created the paddle, which is a filled **`GRect`**, in Milestone 1. The paddle moves along a fixed horizontal line on the bottom of the screen, and you are provided with a **`PADDLE_OFFSET`** constant that is the distance from the top of the paddle to the bottom of the window.

The challenge here is to make the paddle track the mouse. **Note that the *center* of your paddle should follow your mouse when it moves.** Mouse tracking makes use of the event-driven model discussed in Lecture 21. Here, you only have to pay attention to the *x* coordinate of the mouse because the *y* position of the paddle is fixed. An additional constraint 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 change it if necessary to ensure that the entire paddle is visible in the window.

Milestone 4: Get the ball to bounce off the walls (`breakoutgraphics.py` and `breakout.py`)

The ball is just a filled **`GOval`**, but the interesting part lies in getting it to move and bounce appropriately. You are now past the "setup" phase and into the "play" phase of the game. If you didn't already do this in Milestone 1, you should create a ball and place it in the center of the window. 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 program needs to keep track of the velocity of the ball, which consists of two separate components that you can declare as class attributes. The velocity components represent the change in position that occurs at each time step. Initially, the ball should be heading downward, and we provide you with a starting speed constant for **`vy`** (remember that *y* values increase as you move down the screen). The game would be boring if every ball took the same course, so you should choose the **`vx`** component randomly. You can use the function **`random.uniform(-MAX_SPEED, MAX_SPEED)`** to select the **`vx`** component.

Once you've done that, your next challenge is to get the ball to bounce around the world, ignoring the paddle and the bricks for now. To do so, you need to check to see if the coordinates of the ball have gone beyond the boundary of the window, taking into account that the ball has a particular width and height. Thus, to see if the ball has bounced off the right wall, you'll need to see if the x coordinate of the right edge of the ball has become greater than the width of the window. The other three walls should be handled 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 it later so that hitting the bottom wall signifies the end of a turn.

Next, you'll need to compute what happens after a bounce. If a ball bounces off the top or bottom wall, all you need to do is reverse the sign of **`vy`**. Similarly, bounces off the side walls should reverse the sign of **`vx`**.

Finally, you must think about when the ball should actually start bouncing. In order to make the game more realistic, you should wait until the user indicates that they are ready to play before you start bouncing the ball. In our version of the game, we will say that the user indicates that they are ready to play by clicking the mouse. Therefore, you should wait until a mouse click event is received before setting the initial velocity of the ball. The **`onmouseclicked()`** campy mouse listener function might be useful here!

Note that in a given turn, subsequent mouse clicks after the initial one **should not** reset the velocity of the ball. You'll need to think about how to write your mouse listener function in a way that checks whether or not the game has already started.

Milestone 5: Check for collisions (`breakoutgraphics.py` and `breakout.py`)

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. To do so, in each iteration of your animation loop, you'll need to check if the ball has collided with an object or a wall. Since collisions with objects (thinks bricks and the paddle) behave differently than collisions with the walls (which we implemented in the previous milestone), you should have a separate method to handle collisions with objects. In order to implement this method, we first have to think about what constitutes a collision with another object.

To think through this problem, it helps to begin by making a simplifying assumption and then later change that assumption. 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? What happens if you call

```
window.get_object_at(x, y)
```

where **x** and **y** are the coordinates of the ball? If the point **(x, y)** is underneath an object, this call returns the graphical object with which the ball has collided. If there are no objects at the point **(x, y)**, you'll get the value **None**.

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:

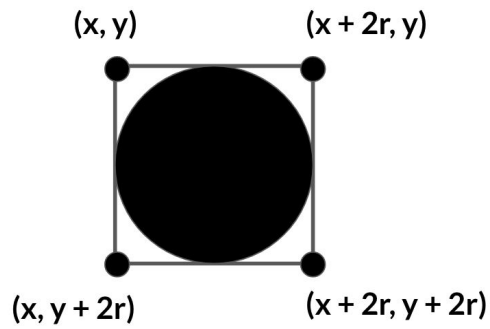

Figure 3: The four coordinates that mark the ball's bounding square

These points have the advantage of being outside the ball (which means that **get_object_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_object_at()** on that location to see whether anything is there.
2. If the value you get back is not **None**, then you don't need to look any farther. You can simply return that **GObject**, since it's the object that the ball hit.
3. If **get_object_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 collision exists.

Hint: It would be very useful to write this section of code as a separate method that returns the object involved in the collision, if any, and **None** otherwise.

Once you have completed the task above, the one step remaining is deciding what to do when a collision occurs. There are only two possibilities. The first possibility is that the object you get back is the paddle, in which case you should bounce the ball so that it starts traveling up. The second possibility is that the object you get back 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. You can remove it from the screen by calling the **window.remove()** method on the object.

Milestone 6: Implement the end conditions (`breakout.py` and `breakoutgraphics.py`)

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

- You've got 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 pretty hard to lose. You've got to modify your loop structure so that it tests for hitting the bottom wall as one of its terminating conditions. You may want to refer to Lecture 20 for ideas about how to implement a method that checks for this terminating condition.
    - When the ball hits the bottom wall, you should reset the ball to its starting position (waiting for a click to set the ball into motion). It may be helpful to decompose out a method that's responsible for resetting the ball. You should not reset the bricks at all.
    - You should implement logic that allows the player to have three "turns", or attempts, at breaking all of the bricks. The third time that the ball hits the bottom wall, you should remove the ball from the screen, signifying that the player has lost the game. Once the player loses the game, you should display some centered text (think **`GLabel`**) in the window letting them know that the game is over.
- You've got 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 are done. Once the user has reached this point, they have won the game! As before, you should remove the ball from the screen at this point and display some centered text (again, a **`GLabel`** might be useful here) in the window letting the user know that they won the game.
- You can experiment with the settings that control the speed of your program. How long should you pause in the loop that updates the ball? Do you need to change the velocity values to get better play action?
- 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: First, change the appropriate constant to make the paddle twice or three times as tall as usual. Then, 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.

Congratulations! Once you've worked through all the steps above, you have officially implemented your very own arcade game! Share the game widely with your family and friends to test their gaming prowess.

Strategy and tactics

Here are some implementation hints for this assignment:

- *Start as soon as possible.* This assignment is due in just over a week, 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.

- *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 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 + scores because there are so many possible extensions. Remember that if you are going to create a version of your program with extensions, you should submit two versions of the assignment: the basic version that meets all the assignment requirements and the extended version. Here are a few ideas for possible extensions (of course, we encourage you to use your imagination to come up with other ideas as well):

- *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. The way the old arcade game worked was that 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. As one example of this, you might consider adding this feature by doubling the horizontal velocity of the ball the seventh time it hits the paddle, figuring that's the time the player is growing complacent.
- *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 have more points for red bricks than cyan bricks. You could display the score underneath the paddle, since it won't get in the way there.
- *Use your imagination.* What else have you always wanted a game like this to do?