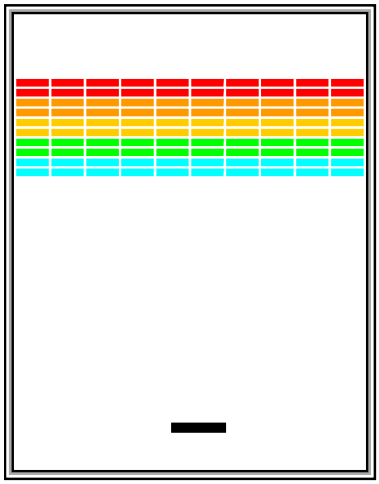
Mehran Sahami
CS 106A

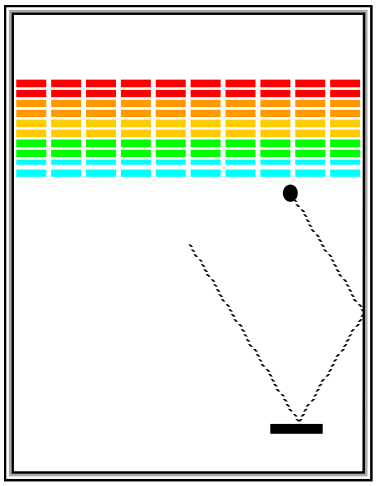
Handout #19
October 15, 2007

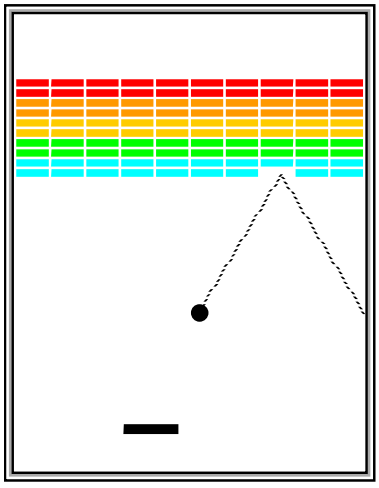
Assignment #3—Breakout!
Due: 3:15pm on Wednesday, October 24th

Based on a handout by Eric Roberts

Your job in this assignment is to write 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 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.







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 edge of its space.

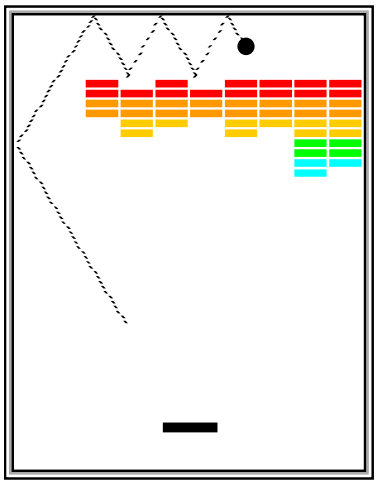
A complete game consists of three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. That ball bounces off the paddle and the walls of the world, in accordance with the physical principle generally expressed as "the angle of incidence 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 to show the ball's path and won't 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 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 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. 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 an 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 applies: bouncing off walls, clearing bricks, and otherwise obeying the laws of physics.

The starter file

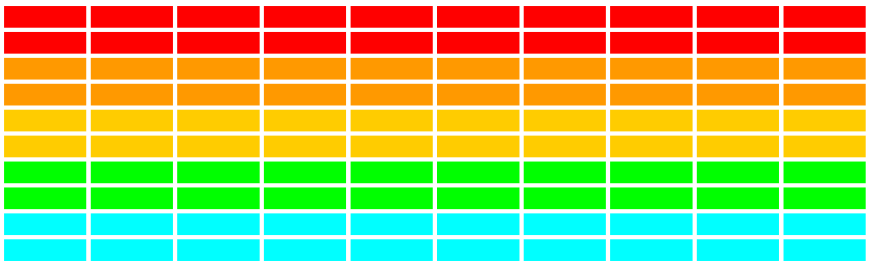
The starter project for this assignment has a little more in it than it has in the past, but none of the important parts of the program. The starting contents of the **`Breakout.java`** file appear in Figure 1 (on the next page). This file takes care of the following details:

- It includes the imports you will need for writing the game.
- It defines the named 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.

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.

Set up the bricks

Before you start playing the game, you have to set up the various pieces. Thus, it probably makes sense to implement the **`run`** method as two method calls: one that sets up the game and one that plays it. An important part of the setup consists of creating the rows of bricks at the top of the game, which look like this:

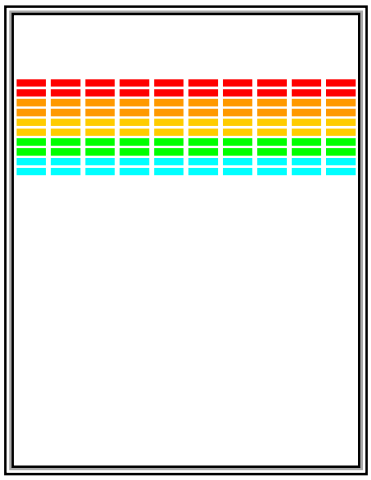


The number, dimensions, and spacing of the bricks are specified using named constants in the starter file, as is the distance from the top of the window to the first line of bricks. The only value you need to compute is the *x* coordinate of the first column, which should be chosen so that the bricks are centered in the window, with the leftover space divided equally on the left and right sides. The color of the bricks remain constant for two rows and run in the following rainbow-like sequence: **`RED`**, **`ORANGE`**, **`YELLOW`**, **`GREEN`**, **`CYAN`**.

Figure 1. The `Breakout.java` starter file

```
/*
 * File: Breakout.java
 * -------------------
 * This file will eventually implement the game of Breakout.
 */

import acm.graphics.*;
import acm.program.*;
import acm.util.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Breakout extends GraphicsProgram {

/** Width and height of application window in pixels */
   public static final int APPLICATION_WIDTH = 400;
   public static final int APPLICATION_HEIGHT = 600;

/** Dimensions of game board (usually the same) */
   private static final int WIDTH = APPLICATION_WIDTH;
   private static final int HEIGHT = APPLICATION_HEIGHT;

/** Dimensions of the paddle */
   private static final int PADDLE_WIDTH = 60;
   private static final int PADDLE_HEIGHT = 10;

/** Offset of the paddle up from the bottom */
   private static final int PADDLE_Y_OFFSET = 30;

/** Number of bricks per row */
   private static final int NBRICKS_PER_ROW = 10;

/** Number of rows of bricks */
   private static final int NBRICK_ROWS = 10;

/** Separation between bricks */
   private static final int BRICK_SEP = 4;

/** Width of a brick */
   private static final int BRICK_WIDTH =
     (WIDTH - (NBRICKS_PER_ROW - 1) * BRICK_SEP) / NBRICKS_PER_ROW;

/** Height of a brick */
   private static final int BRICK_HEIGHT = 8;

/** Radius of the ball in pixels */
   private static final int BALL_RADIUS = 10;

/** Offset of the top brick row from the top */
   private static final int BRICK_Y_OFFSET = 70;

/** Number of turns */
   private static final int NTURNS = 3;

   public void run() {
      /* You fill this in, along with any subsidiary methods */
   }
}
```

This part of the assignment is almost exactly like the pyramid problem from Assignment #2. The parts that are only a touch more difficult are that you need to fill and color the bricks. This extra complexity is more than compensated by the fact that there are the same number of bricks in each row, and you don't have to change the coordinate calculation from row to row.



Here's a suggestion: Why don't you get this part of the program working by Wednesday the 17th so that you can produce just the diagram at the right? The display has most of what you see on the final screen and will give you considerable confidence that you can get the rest done. And you'll be well on your way before time gets short.

Create the paddle

The next step is to create the paddle. At one level, this is considerably easier than 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 in creating the paddle is to make it track the mouse. The technique is similar to that discussed in Chapter 9 for dragging an object around in the window. Here, however, you only have to pay attention to the *x* coordinate of the mouse because the *y* position of the paddle is 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 boundary and change it if necessary to ensure that the entire paddle is visible in the window.

Here's a soon-to-become-boring suggestion: Why don't you get this part of the program working by Friday the 19th, so that you can follow the mouse with the paddle? This entire part of the program takes fewer than 10 code lines, so it shouldn't take so long. The hard part lies in reading the Graphics chapter and understanding what you need to do.

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" phase 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 mathematics is not any more difficult, but may be a bit less intuitive.

The program needs to keep track of the velocity of the ball, which consists of two separate components, which you will presumably declare as instance variables like this:

```
private double vx, vy;
```

The velocity components represent the change in position that occurs on each time step. Initially, the ball should be heading downward, and you might try a starting velocity of +3.0 for **`vy`** (remember that *y* values in Java 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.

In line with our discussion of generating random numbers last week, you should simply do the following:

1. Declare an instance variable **rgen**, which will serve as a random-number generator:

   ```
   private RandomGenerator rgen = RandomGenerator.getInstance();
   ```

   Remember that instance variables are declared outside of any method but inside the class.

2. Initialize the **vx** variable as follows:

   ```
   vx = rgen.nextDouble(1.0, 3.0);
   if (rgen.nextBoolean(0.5)) vx = -vx;
   ```

   This code sets **vx** to be a random **double** in the range 1.0 to 3.0 and then makes it negative half the time. This strategy works much better for Breakout than calling

   ```
   nextDouble(-3.0, +3.0)
   ```

   which might generate a ball going more or less straight down. That would make life far too easy for the player.

Once you've done that, your next challenge is to get the ball to bounce around the world, ignoring entirely the paddle and the bricks. To do so, you need to check to see if the coordinates of the ball have gone beyond the boundary, taking into account that the ball has a nonzero size. 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 signifies 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**.

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 look in Chapter 9 (page 299) at the methods that are defined at the **GraphicsProgram** level, you will discover that there is a method

```
public GObject getElementAt(double x, double y)
```

that takes a position 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 the special constant **null**. If there is more than one, **getElementAt** always chooses the one closest to the top of the stack, which is the one that appears to be in front on the display.

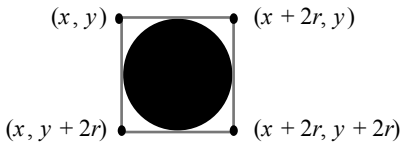
What happens if you call

```
getElementAt(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 **null**.

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 farther 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, then no collision exists.

It would be very useful to write this section of code as a separate method

```
private GObject getCollidingObject()
```

that returns the object involved in the collision, if any, and **null** otherwise. You could then use it in a declaration like

```
GObject 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 by checking

```
if (collider == paddle) . . .
```

If it 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.

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 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'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 must be done. In terms of the requirements of the assignment, you can simply stop at that point, but it would be nice to give the player a little feedback that at least indicates whether the game was won or lost.
- You've got to 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: 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.

Strategy and tactics

Here are some survival 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.

- *Set up a milestone schedule.* In the handout, I've suggested that you get the brick display up and running by this coming Wednesday and the paddle moving by Friday. 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 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 + or (dare I say it) ++ 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 of for possible extensions (of course, we encourage you to use your imagination to come up with other ideas as well):

- *Add sounds.* The version that is running as an applet on the CS 106A assignment page plays a short bounce sound every time the ball collides with a brick or the paddle. This extension turns out to be very easy. The starter project contains an audio clip file called **`bounce.au`** that contains that sound. You can load the sound by writing

  ```
  AudioClip bounceClip = MediaTools.loadAudioClip("bounce.au");
  ```

  and later play it by calling

  ```
  bounceClip.play();
  ```

  The Java libraries do make some things easy.
- *Add messages.* The web version 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 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. The web version implements this feature.
- *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. The applet version implements 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 got 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?