Problem Set 4: Breakout

This is CS50. Harvard University. Fall 2013.

Table of Contents

due ~~Thu 10/10~~ Fri 10/11 at noon

with thanks to Eric Roberts of Stanford

No need to submit any work early (i.e., by Wed) this week in order to get an extra day. Everyone gets an extra day this week; everyone's deadline is Fri 10/11!

Questions? Head to cs50.net/discuss[1] or join classmates at office hours[2]!

Objectives

- Learn an API.
- Build a game with a real GUI.
- Acquaint you with event handling.
- Impress your friends.

[1] https://www.cs50.net/discuss
[2] https://www.cs50.net/ohs

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on quizzes is not permitted at all. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course's heads. Acts considered not reasonable by the course are handled harshly. If the course refers some matter to the Administrative Board and the outcome is Admonish, Probation, Requirement to Withdraw, or Recommendation to Dismiss, the course reserves the right to impose local sanctions on top of that outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at Office Hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past semesters' quizzes and solutions thereto.

- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on CS50 Discuss or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Redeeming or attempting to redeem someone else's code for a late day.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.

- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Scores

Your work on this problem set will be evaluated along four axes primarily.

Scope
: To what extent does your code implement the features required by our specification?

Correctness
: To what extent is your code consistent with our specifications and free of bugs?

Design
: To what extent is your code written well (i.e., clearly, efficiently, elegantly, and/or logically)?

Style
: To what extent is your code readable (i.e., commented and indented with variables aptly named)?

All students, whether taking the course SAT/UNS or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a satisfactory grade unless granted an exception in writing by the course's heads.

Shorts

- Head to https://www.cs50.net/shorts/4 and watch the shorts on pointers and strings. Then head to https://www.cs50.net/shorts/5 and watch the short on the CS50 Library.

We may have a few questions for you when it comes time to submit this problem set's form!

Backstory

One day in the late summer of 1975, Nolan Bushnell [founder of Atari and, um, Chuck E. Cheese's], defying the prevailing wisdom that paddle games were over, decided to develop a single-player version of Pong; instead of competing against an opponent, the player would volley the ball into a wall that lost a brick whenever it was hit. He called [Steve] Jobs into his office, sketched it out on his little blackboard, and asked him to design it. There would be a bonus, Bushnell told him, for every chip fewer than fifty that he used. Bushnell knew that Jobs was not a great engineer, but he assumed, correctly, that he would recruit [Steve] Wozniak, who was always hanging around. "I looked at it as a two-for-one thing," Bushnell recalled. "Woz was a better engineer."

Wozniak was thrilled when Jobs asked him to help and proposed splitting the fee. "This was the most wonderful offer in my life, to actually design a game that people would use," he recalled. Jobs said it had to be done in four days and with the fewest chips possible. What he hid from Wozniak was that the deadline was one that Jobs had imposed, because he needed to get to the All One Farm to help prepare for the apple harvest. He also didn't mention that there was a bonus tied to keeping down the number of chips.

"A game like this might take most engineers a few months," Wozniak recalled. "I thought that there was no way I could do it, but Steve made me sure that I could." So he stayed up four nights in a row and did it. During the day at HP, Wozniak would sketch out his design on paper. Then, after a fast-food meal, he would go right to Atari and stay all night. As Wozniak churned out the design, Jobs sat on a bench to his left implementing it by wire-wrapping the chips onto a breadboard. "While Steve was breadboarding, I spent time playing my favorite game ever, which was the auto racing game Gran Trak 10," Wozniak said.

Astonishingly, they were able to get the job done in four days, and Wozniak used only forty-five chips. Recollections differ, but by most accounts Jobs

> simply gave Wozniak half of the base fee and not the bonus Bushnell paid for saving five chips. It would be another ten years before Wozniak discovered (by being shown the tale in a book on the history of Atari titled *Zap*) that Jobs had been paid this bonus….
>
> — Walter Isaacson *Steve Jobs*

Getting Started

Your challenge for this problem set is to implement the same game that Steve and Steve did, albeit in software rather than hardware. That game is Breakout.

Now, Problem Set 3 was also a game, but its graphical user interface (GUI) wasn't exactly a GUI; it was more of a textual user interface, since we essentially simulated graphics with `printf`. Let's give Breakout an actual GUI by building atop the Stanford Portable Library (SPL), which is similar in spirit to the CS50 Library but includes an API (application programming interface) for GUI programming and more.

Let's get you started.

- As always, first open a terminal window and execute

  ```
  update50
  ```

  to make sure your appliance is up-to-date.

- Next execute

  ```
  cd ~/Dropbox
  ```

  followed by

  ```
  wget http://cdn.cs50.net/2013/fall/lectures/5/m/src5m.zip
  ```

  to download some source code from Week 5. If you instead see

  ```
  unable to resolve host address
  ```

your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

When ready, unzip the file with

```
unzip src5m.zip
```

at which point you should find yourself with a directory called `src5m` in `~/Dropbox`. Navigate your way into it with

```
cd src5m
```

and then execute

```
ls
```

and you should see the below.

```
bounce.c  checkbox.c  cursor.c  Makefile  spl      text.c
button.c  click.c     label.c   slider.c  spl.jar  window.c
```

Go ahead and compile all of these programs at once (thanks to the `Makefile` in there) by executing the below.

```
make
```

Then execute the simplest of those programs as follows.

```
./window
```

A window quite like the below should appear and then disappear after 5 seconds.

Neat, eh? Open up `window.c` with `gedit`. Let's now take a tour.

http://www.youtube.com/watch?v=gjirq3MzqBY

How did we know how to call `newGWindow` like that? Well, there aren't `man` pages for SPL, but you can peruse the relevant header file (`gwindow.h`). In fact, notice that inside of `src5m` is a subdirectory called `spl`. Inside of that is another subdirectory called `include`. If you take a look there, you'll find `gwindow.h`. Open it up with `gedit` and look around. (Alternatively, you can see it at http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/include/gwindow.h.) Hm, a bit overwhelming. But because SPL's author has commented the code in a standard way, it turns out that you can generate more user-friendly, web-based documentation as a result! Indeed, take a look now at http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gwindow.html, and you'll see a much friendlier format. Click `newGWindow` under **Functions**, and you'll see its prototype:

```
GWindow newGWindow(double width, double height);
```

That's how we knew! See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/index.html for an index into SPL's documentation, though we'll point out more specific places to look.

Now, in the interests of full disclosure, we should mention that SPL is still in beta, so there may be some bugs in its documentation. When in doubt, best to consult those raw header files instead!

- Next open up `click.c` with `gedit`. This one's a bit more involved but it's representative of how to "listen" for "events", quite like those you could "broadcast" in Scratch. Let's take a look.
  http://www.youtube.com/watch?v=BStiekPFKWI
  See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gevents.html for SPL's documentation of `GEvent`.
- Now open up `cursor.c` with `gedit`. This program, too, handles events, but it also responds to those events by moving a circle (well, a `GOval`) in lockstep. Let's take a look.
  http://www.youtube.com/watch?v=xsB0v8GtVMw
  See http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gobjects.html for SPL's documentation of `GOval` and other types of objects.
- Next open `bounce.c` with `gedit`. This one uses a bit of arithmetic to bounce a circle back and forth between a window's edges. Let's take a look.
  http://www.youtube.com/watch?v=8RMHJe1ZpKM
- Now take a look at `button.c`, `checkbox.c`, `label.c`, `slider.c`, and `text.c` in any order with `gedit`. No videos for those, but do try running them each to see what more you can do with SPL. If you've any questions on usage, reach out via CS50 Discuss[3]!
- Okay, now that you've been exposed to a bit of GUI programming, let's turn our attention to this problem set's own distribution code. In a terminal window, execute

```
cd ~/Dropbox
```

if not already there, and then execute

[3] https://www.cs50.net/discuss

```
wget http://cdn.cs50.net/2013/fall/psets/4/pset4/pset4.zip
```

to download a ZIP of this problem set's distro into your appliance. You should see a bunch of output followed by:

```
'pset4.zip' saved
```

As before, if you instead see

```
unable to resolve host address
```

your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

Ultimately, confirm that you've indeed downloaded `pset4.zip` by executing:

```
ls
```

Then, run

```
unzip pset4.zip
```

to unzip the file. If you then run `ls` again, you should see that you have a newly unzipped directory called `pset4` as well. Proceed to execute

```
cd pset4
```

followed by

```
ls
```

and you should see that the directory contains three files and one subdirectory:

- `Makefile`, which will let `make` know how to compile your program;
- `breakout.c`, which contains a skeleton for your program;
- `spl`, a subdirectory containing SPL; and
- `spl.jar`, a "Java archive" that contains code (written in a language called Java) atop which SPL's C library is written.

- Let's see what the distribution code does. Go ahead and execute

```
make breakout
```

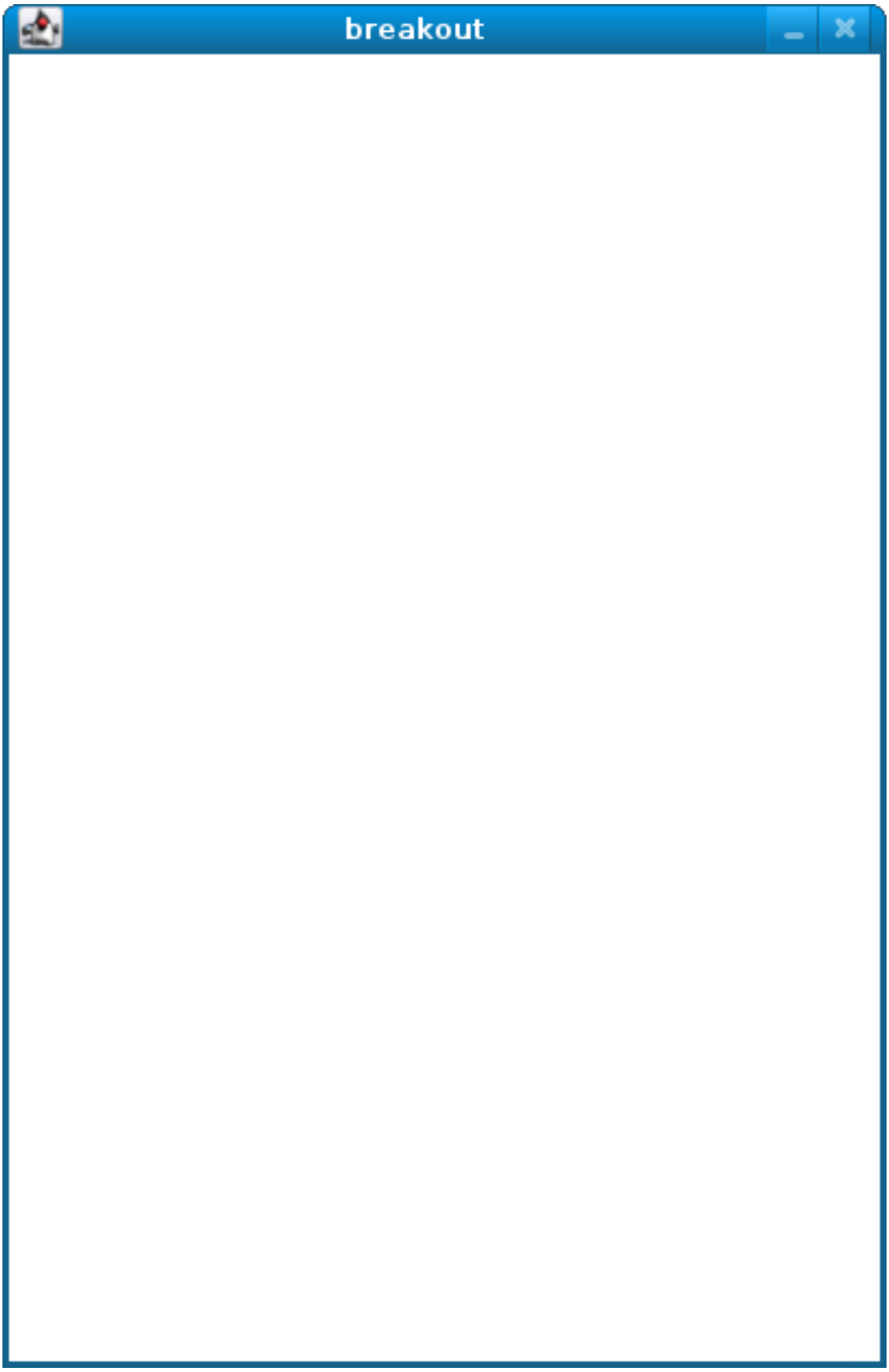
or, more simply,

```
make
```

to compile the distro. Then execute

```
./breakout
```

to run the program as is. A window like the below should appear.

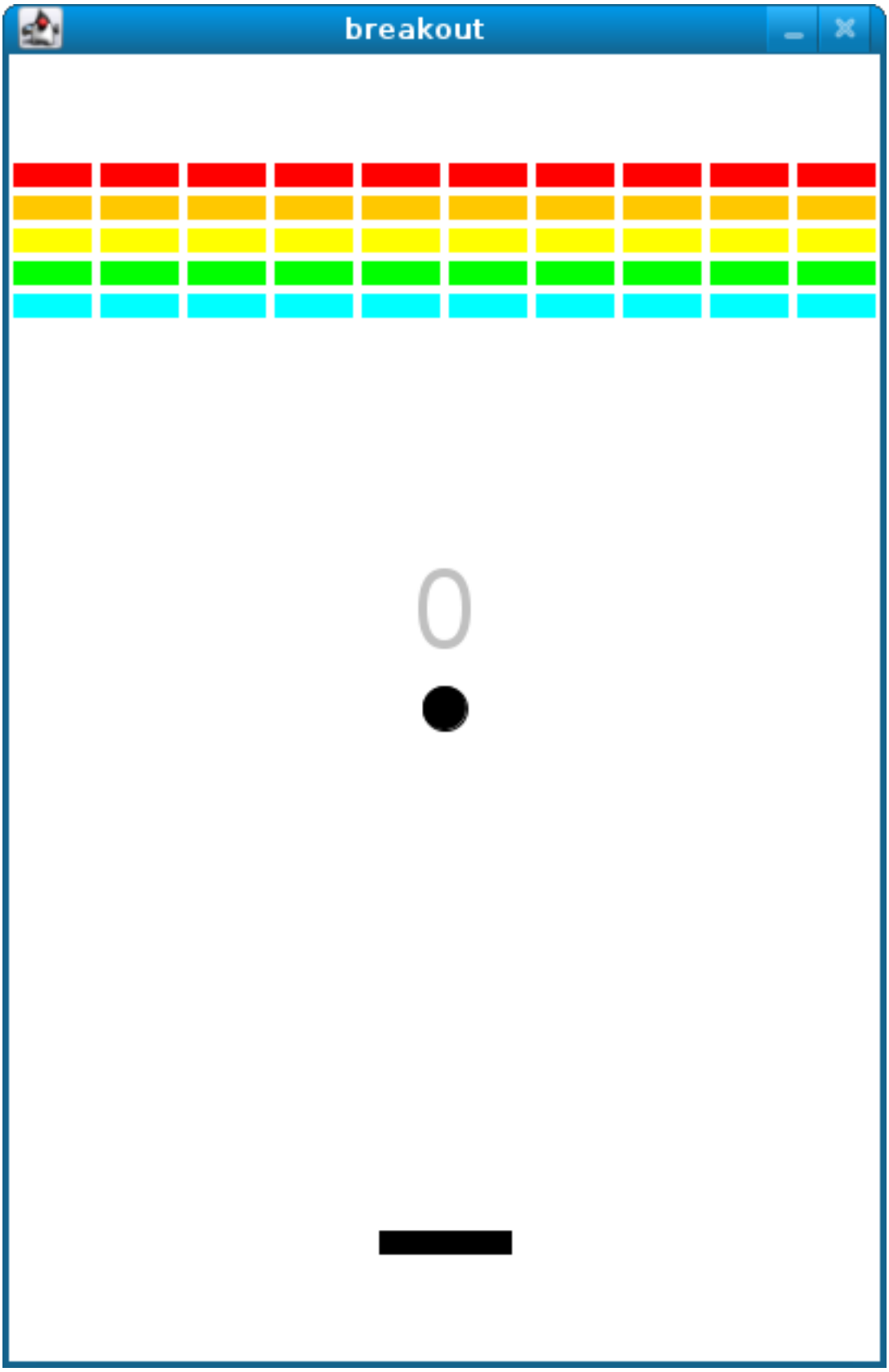

breakout

Hm, not much of a game. Yet!

- Next try out the staff's solution by executing the below from within your own `~/Dropbox/pset4` directory.

```
~cs50/pset4/breakout
```

A window like the below should appear.


breakout
0

How fun! Go ahead and click somewhere inside that window in order to play. The goal, quite simply, is to bounce the ball off of the paddle so as to hit bricks with it. If you break all the bricks, you win! But if you miss the ball three times, you lose! To quit the game, hit control-c back in the terminal window.

Nice. Let's make your implementation look more like that one. But, first, a tour!

- Open up `breakout.c` with `gedit` and take a moment to scroll through it to get a sense of what lies ahead. It's a bit reminiscent of the skeleton for Game of Fifteen, no? But definitely some new functions in there, most from SPL. Let's walk through it from top to bottom.
    - Atop the file you'll see some familiar header files. We've also included `time.h` so that you have access to a "pseudorandom number generator" (PRNG), a function that can generate random (well, technically not-quite-random) numbers. We've also included some header files from SPL. Because those files are included in this problem set's distribution code (in `pset4/spl/include`), we've used, as is required by C, double quotes (`"`) around their filenames instead of the usual angled brackets (`<` and `>`) because they're not installed deep in the appliance itself.
    - Next up are some constants, values that you don't need to change, but because the code we've written (and that you'll write) needs to know these values in a few places, we've factored them out as constants so that we or you could, theoretically, change them in one convenient location. By contrast, hard-coding the same number (pejoratively known as a "magic number") into your code in multiple places is considered bad practice, since you'd have to remember to change it, potentially, in all of those places.
    - Below those constants are a bunch of prototypes for functions that are defined below `main`. More on each of those soon.
    - Next up is our old friend, `main`. It looks like the first thing that `main` does is "seed" that so-called PRNG with the current time. (See `man srand48` and `man 2 time` if curious.) To seed a PRNG simply means to initialize it in such a way that the numbers it will eventually spit out will appear to be random. It's deliberate, then, that we're initializing the PRNG with the current time: time's always changing. Were we instead to initialize the PRNG with some hard-coded value, it'd always spit out the same sequence of "random" numbers.

After that call to `srand48`, it looks like `main` calls `newGWindow`, passing in a desired `WIDTH` and `HEIGHT`. That function "instantiates" (i.e., creates) a new graphical window, returning some sort of reference thereto. (It's technically a pointer, but that detail, and the accompanying `*`, is, again, hidden from us by SPL.) That function's return value is apparently stored in a variable called `window` whose type is `GWindow`, which happens to be declared in a `gwindow.h` header file that you may have glimpsed earlier.

Next, `main` calls `initBricks`, a function written partly by us (and, soon, mostly by you!) that instantiates a grid of bricks atop the game's window.

Then `main` calls `initBall`, which instantiates the ball that will be used to play Breakout. Passed into that function is `window` so that the function knows where to "place" (i.e., draw) the ball. The function returns a `GOval` (graphical oval) whose width and height will simply be equal (ergo a circular ball).

Called by `main` next is `initPaddle`, which instantiates the game's paddle; it returns a `GRect` (graphical rectangle).

Then `main` calls `initScoreboard`, which instantiates the game's scoreboard, which is simply a `GLabel` (graphical label).

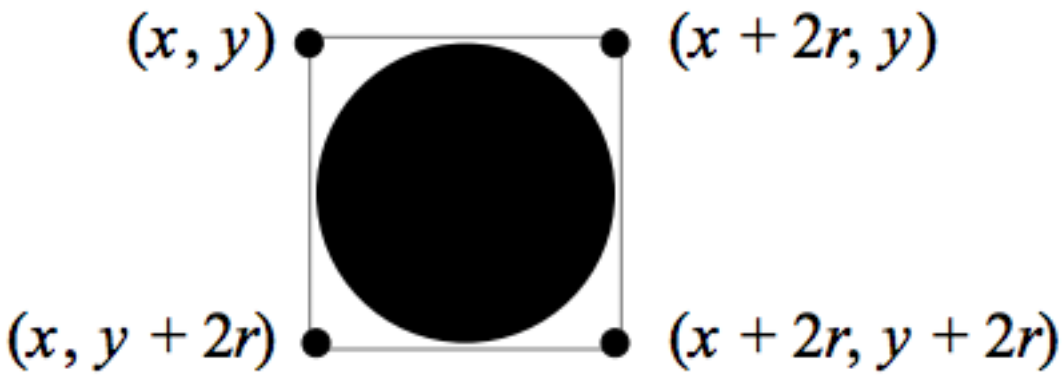
Below all those function calls are a few definitions of variables, namely `bricks`, `lives`, and `points`. Below those is a loop, which is meant to iterate again and again so long as the user has lives left to live and bricks left to break. Of course, there's not much code in that loop now!

Below the loop is a call to `waitForClick`, a function that does exactly that so that the window doesn't close until the user intends.

Not too bad, right? Let's next take a closer look at those functions.

- In `initBricks`, you'll eventually write code that instantiates a grid of bricks in the window. Those constants we saw earlier, `ROWS` and `COLS`, represent that grid's dimensions. How to draw a grid of bricks on the screen? Well, odds are you'll want to employ a pair of `for` loops, one nested inside of the other. And within that innermost loop, you'll likely want to instantiate a `GRect` of some width and height (and color!) to represent a brick.

- In `initBall`, you'll eventually write code that instantiates a ball (that is, a circle, or really a `GOval`) and somehow center it in the window.
- In `initPaddle`, you'll eventually write code that instantiates a paddle (just a `GRect`) that's somehow centered in the bottom-middle of the game's window.
- Finally, in `initScoreboard`, you'll eventually write code that instantiates a scoreboard as, quite simply, a `GLabel` whose value is a number (well, technically, a `char*`, which we once knew as a `string`).
- Now, we've already implemented `updateScoreboard` for you. All that function does, given a `GWindow`, a `GLabel`, and an `int`, is convert the `int` to a `string` (okay, `char*`) using a function called `sprintf`, after which it sets the label to that value and then re-centers the label (in case the `int` has more digits than some previous `int`). Why did we allocate an array of size `12` for our representation of that `int` as a `string`? No worries if the reason's non-obvious, but give some though as to how wide the most positive (or most negative!) `int` might be. You're welcome to change this function, but you're not expected to.
- Last up is `detectCollision`, another function that we've written for you. (Phew!) This one's a bit more involved, so do spend some time reading through it. This function's purpose in life, given the ball as a `GOval`, is to determine whether that ball has collided with (i.e., is overlapping) some other object (well, `GObject`) in the game. (A `GRect`, `GOval`, or `GLabel` can also be thought of and treated as a `GObject`, per http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gobjects.html.) To do so, it cuts some corners (figuratively but also kind of literally) by checking whether any of the ball's "corners," as defined by the ball's "bounding box", per the below (wherein *x* and *y* represent coordinates, and *r* represents the ball's radius) are touching some other `GObject` (which might be a brick or a paddle or even something else).



Alright, ready to break out Breakout?

Breakout

Alright, if you're like me, odds are you'll find it easiest to implement Breakout via some baby steps, each of which will get you closer and closer to a great outcome. Rather than try to implement the whole game at once, allow me to suggest that you proceed as follows:

1. Try out the staff's solution again (via `~cs50/pset4/breakout` from within your own `~/Dropbox/pset4` directory) to remind yourself how our implementation behaves. Yours doesn't need to be identical. In fact, all the better if you personalize yours. But playing with our implementation should help guide you toward yours.
2. Implement `initPaddle`. Per the function's return value, your paddle should be implemented as a `GRect`. Odds are you'll first want to decide on a width and height for your paddle, perhaps declaring them both atop `breakout.c` with constants. Then calculate coordinates (*x* and *y*) for your paddle, keeping in mind that it should be initially aligned in the bottom-middle of your game's window. We leave it to you to decide exactly where. Odds are some arithmetic involving the window's width and height and the paddle's width and height will help you center it. Keep in mind that *x* and *y* refer to a `GRect's top-left corner, not its own middle. Your paddle's size and location doesn't need to match the staff's precisely, but it should be perfectly centered, near the window's bottom. You're welcome to choose a color for it too, for which `setColor` and `setFilled` might be of interest. Finally, instantiate your paddle with `newGRect`. (Take note of that function's prototype at http://cdn.cs50.net/2013/fall/psets/4/pset4/pset4/spl/doc/gobjects.html.) Then return the `GRect` returned by `newGRect` (rather

than `NULL`, which the distribution code returns only so that the program will compile without `initPaddle` fully implemented).

3. Now, `initPaddle's purpose in life is only to instantiate and return a paddle (i.e., `GRect`). It shouldn't handle any of the paddle's movement. For that, turn your attention to the `TODO` up in `main`. Proceed to replace that `TODO` with some lines of code that respond to a user's mouse movements in such a way that the paddle follows the movements, but only along its (horizontal) x-axis. Look back at `cursor.c` for inspiration, but keep in mind that `cursor.c` allowed that circle to move along a (vertical) y-axis as well, which we don't want for Breakout, else the paddle could move anywhere (which might be cool but not exactly Breakout).

4. Now turn your attention to the `TODO` in `initBricks`. Implement that function in such a way that it instantiates a grid of bricks (with `ROWS` rows and `COLS` columns), with each such brick implemented as a `GRect`. Drawing a `GRect` (or even a bunch of them) isn't all that different from drawing a `GOval` (or circle). Odds are, though, you'll want to instantiate them within a `for` loop that's within a `for` loop. (Think back to `mario`, perhaps!) Be sure to leave a bit of a gap between adjacent bricks, just like we did; exactly how many pixels is up to you. And we leave it to you to select your bricks' colors.

5. Now implement `initBall`, whose purpose in life is to instantiate a ball in the window's center. (Another opportunity for a bit of arithmetic!) Per the function's prototype, be sure to return a `GOval`.

6. Then, back in `main`, where there used to be a `TODO`, proceed to write some additional code (within that same `while` loop) that compels that ball to move. Here, too, take baby steps. Look to `bounce.c` first for ideas on how to make the ball bounce back and forth between your window's edges. (Not the ultimate goal, but it's a step toward it!) Then figure out how to make the ball bounce up and down instead of left and right. (Closer!) Then figure out how to make the ball move at an angle. Then, utilize `drand48` to make the ball's initial velocity random, at least along its (horizontal) x-axis. Note that, per its `man` page, `drand48` returns "nonnegative double-precision floating-point values uniformly distributed between [0.0, 1.0)." In other words, it returns a `double` between 0.0 (inclusive) and 1.0 (exclusive). If you want your velocity to be faster than that, simply add some constant to it and/or multiply it by some constant! Ultimately, be sure that the ball still bounces off edges, including the window's bottom for now.

7. When ready, add some additional code to `main` (still somewhere inside of that `while` loop) that compels the ball to bounce off of the paddle if it collides with it on its way downward. Odds are you'll want to call that function we wrote, `detectCollision`, inside that loop in order to detect whether the ball's collided with something so that, if so, you can somehow handle such an event. Of course, the ball could collide with the paddle or with any one of those bricks. Keep in mind, then, that `detectCollision` could return any such `GObject`; it's left to you to determine what has been struck. Know, then, that if you store its return value, as with

```
GObject object = detectCollision(window, ball);
```

you can determine whether that `object` is your game's paddle, as with the below.

```
if (object == paddle)
{
    // TODO
}
```

More generally, you can determine if that `object` is a `GRect` with:

```
if (strcmp(getType(object), "GRect") == 0)
{
    // TODO
}
```

Once it comes time to add a `GLabel` to your game (for its scoreboard), you can similarly determine if that `object` is `GLabel`, in which case it might be a collision you want to ignore. (Unless you want your scoreboard to be something the ball can bounce off of. Ours isn't.)

```
if (strcmp(getType(object), "GLabel") == 0)
{
    // TODO
}
```

8. Once you have the ball bouncing off the paddle (and window's edges), focus your attention again on that `while` loop in `main` and figure out how to detect if the

ball's hit a brick and how to remove that brick from the grid if so. Odds are you'll find `removeGWindow` of interest, per http://cdn.cs50.net/2013/fall/lectures/5/m/src5m/spl/doc/gwindow.html. **SPL's documentation incorrectly refers to that function as `remove`, but it's indeed `removeGWindow` you want, whose prototype, to be clear, is the below.**

```
void removeGWindow(GWindow gw, GObject gobj);
```

9. Now decide how to determine whether the ball has zoomed past the paddle and struck the window's bottom edge, in which case the user should lose a life and gameplay should probably pause until the user clicks the mouse button, as in the staff's implementation. Odds are detecting this situation isn't all that different from the code you already wrote for bouncing; you just don't want to bounce off that bottom edge anymore!
10. Lastly, implement `initScoreboard` in such a way that the function instantiates and positions a `GLabel` somewhere in your game's window. Then, enhance `main` in such a way that the text of that `GLabel` is updated with the user's score anytime the user breaks a brick. Indeed, be sure that your program keeps track of how many lives remain and how many bricks remain, the latter of which is inversely related to how many points you should give the user for each brick broken; our solution awards one point per brick, but you're welcome to offer different rewards. A user's game should end (i.e., the ball should stop moving) after a user runs out of lives or after all bricks are broken. We leave it to you to decide what to do in both cases, if anything more!

Because this game expects a human to play, no `check50` for this one! Best to invite some friends to find bugs!

How to Submit

Step 1 of 2

- When ready to submit, open up a Terminal window and navigate your way to `~/Dropbox`. Create a ZIP (i.e., compressed) file containing your entire `pset4` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset4`, including any subdirectories (or even subsubdirectories!).

```
zip -r pset4.zip pset4
```

If you type `ls` thereafter, you should see that you have a new file called `pset4.zip` in `~/Dropbox`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset4.zip`, then create a new ZIP file as before.)

- Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit cs50.net/submit[4], logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **pset4** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `pset4.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset4.zip`, click it once to select it, then click **Open**.
- Click **Start upload** to upload your ZIP file to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to cs50.net/submit[5] and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://forms.cs50.net/2013/fall/psets/4/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!
  This was Problem Set 4.

[4] https://www.cs50.net/submit
[5] https://www.cs50.net/submit