csci-e28
Assignment 4: pong

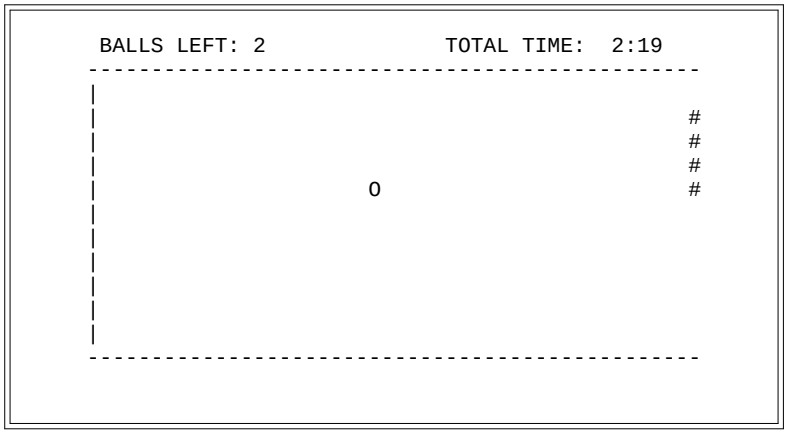
Introduction

For this assignment you will write a video game based on the classic coin-operated (and home console) computer game `pong.´ This is a one-person ping-pong game ; the object of the game is to keep the ball in play as long as possible. The main new idea in this project is to think about and work with techniques for multi-tasking. In writing this game, you will also work with screen management, signal and timer management, and the random number function.

After getting familiar with the structure of this program, you will be better prepared to study servers, sockets, and X-windows programming.

The Pong Board

The pong playing screen looks like:



The pong screen consists of a paddle (the line of #'s), a ping pong ball (the O), and a three-sided court. The size of the court depends on the size of the terminal window. The court is surrounded by a three-unit border. That is, the bottom edge of the court is three rows above the bottom of the screen, the left edge of the court is three columns from the left of the screen, etc. The height of the paddle is approximately one third of the height of the court. The paddle always stays in the same column. The top of the paddle may not rise above the top of the court, and the bottom of the padddle may not drop below the bottom of the court.

Playing Pong

The ball is served from the middle of the court with a random speed. The ball bounces elastically against the walls. The paddle introduces some uncertainty to the game. Each time the ball bounces off the paddle the speed of the ball is changed by a small, random amount.

The player moves the paddle up and down the screen, one row at a time, by pressing the 'k' and 'm' keys, respectively. Each press of the key moves the paddle one row. The paddle will not move beyond its top and bottom limits.

The goal of the game is to play for as long as possible. The amount of playing time (in minutes and seconds) is displayed on the screen.

If the ball gets past the paddle, the ball is out of play. After three balls, the game is over. The number of balls left, not including the one in play, is displayed on the screen.

The player may press the Q key to quit the game.

A Suggested Outline

You may solve this problem almost any way you like. The following steps describe the required modular part and also outline some ideas and skills you may find helpful.

Step One: Putting Up Walls

The bounce2d program from class can serve as a basis for your pong game. Study how it uses curses to draw the ball and move it along. The important functions are `move()`, `addch()`, `refresh()`. `move(y,x)` moves the cursor to a specified point on the screen. `addch(c)` puts a character at the current cursor position and advances the cursor one place right. `refresh()` brings the terminal screen up to date with all your requests.

The bounce program limits the ball to the exact region specified by the pong board, but has invisible walls on all four sides. Your version must have visible walls on three sides and be open on the right. To get familiar with curses, modify bounce2d.c so that it draws the three visible walls. Remember, the size of the pong court depends on the size of the terminal window unlike the fixed-size court of the bounce2d program.

Step Two: Random Serve

The bounce program always serves the ball with the same speed and direction. First, study how the motion is timed and controlled. The x_delay and y_delay members* are counter-intuitive. One usually thinks of speed as distance per unit of time not time per unit of distance. Using delay values ensure the ball moves smoothly from one character position to an adjacent position.

To implement the random serve feature, read the manual page on srand() and rand(). You can initialize, at the start of your program, the random number generator with `srand(getpid())` and create random numbers in the range 0..MAXNUM-1 as needed by calling `num = ( rand() % MAXNUM )`.

Generate a random number in a sensible range for the x_delay and a random number in a sensible range for the y_delay. See the bounce code for a starting value for your experiments.

Step Three: The Paddle - a required implementation

The paddle is defined by a top y-coordinate, a bottom y-coordinate, and an x-coordinate. The x-coordinate is fixed. The top and bottom y-coordinates are controlled by user input. You are *required* to implement the paddle using the C approach to object oriented design: a separate C file and an associated header file. Create a file called paddle.c and define static data to store the state of the paddle. You might use:

```
struct pppaddle {    int     pad_top, pad_bot, pad_col;
                     char    pad_char;
       };
```

In that C file, write a few simple operations for the paddle: `paddle_init()`, `paddle_up()`, `paddle_down()`, and `paddle_contact(y,x)`. The `paddle_init()` function must initialize the paddle data structure and draw it on the screen. The up and down functions should check if the paddle has reached the limits of its path, and if not, to adjust the data structure and then adjust the screen representation of the paddle.

The header file, paddle.h, will contain declarations for the non-static (i.e. public) functions in paddle.c

The interesting one is `paddle_contact(y,x)`. The (y,x) position of the ball is passed to this function. This function tells if the ball at the given position is in contact with the paddle. The calling function can then bounce the ball appropriately.

Other functions used by the paddle must be declared static to make them private to `paddle.c`.

* In the text, these members are called x_ttm and y_ttm. The counters are called x_ttg and y_ttg.

Step Four: Bounce or Lose

The main part of the game is controlled by the `bounce_or_lose()` function. In the original bounce program, the ball bounces against all four walls. In your version, the ball bounces against three walls. At the right side, though, you need to determine if the ball is hitting the paddle, or missing it.

In bounce2d.c, `bounce_or_lose` returns 0 for no contact, or 1 for a bounce. Change the function so it returns -1 for `lose.´ This can help the calling program figure out what to do next. Even better, #define symbolic constants to make the code more readable.

How do you handle the value returned by `bounce_or_lose`? The bounce2d.c program has code for bouncing. What about losing? If the ball sails past the paddle, take it out of play, and if there is still at least one ball left, serve another ball. That will become the current ball in play. When there are no more to play, the `balls_left` variable will hit zero and the game will end.

Finally, modify the program so the speed is slightly, randomly, modified when it hits the paddle. The speed is controlled by the `x_delay` and `s-1y_delay` members of the struct. Where in your program do you add this? Should you change the return value of `bounce_or_lose()` to distinguish bouncing off the paddle from bouncing off the walls?

Step Five: The Glue

The pieces are now ready to assemble. The main loop is only slightly changed from the bounce program. It will look something like

```
main()
{
     set_up();            /* init all stuff */
     serve();             /* start up ball  */

     while ( balls_left > 0 && ( c = getch() ) != 'Q' ){
          if ( c == 'k' )
               up_paddle();
          else if ( c == 'm' )
               down_paddle();
     }
     wrap_up();
}
```

The `up_paddle()` and `down_paddle()` functions should check `bounce_or_lose()`;a paddle could intercept the ball. Draw a few pictures to see how this case works. It could save the game sometimes.

The Big Picture

This initial design solves the problem of multi-tasking by blocking on user input and running the animation with signal-driven animation. The two significant events are character input and timer ticks.

In addition to these two events, there are four objects in the game: the ball, the paddle, the court, and the clock. For each of these objects, consider what variables define the object, how those variables change, and how the objects interact.

One object is the paddle, it has a position, and the position of the paddle is changed by user input. The user types a 'm' and the state of the paddle changes: it moves down one space. The user types a 'k' and the state of the paddle changes again: it moves up one space.

The ball is another object, it has a position and a velocity. This state is defined by six internal values. The position of the ball is changed by timer ticks, and the direction of the ball is changed when it encounters the walls or the paddle.

The function called `bounce_or_lose()` compares the position and speed of the ball to the walls and paddle and modifies the state of the ball or the game under certain conditions. The program has to call this function each time the ball or the paddle moves.

The clock is an object. The clock is controlled by timer ticks. The speed of the clock, unlike the speed of the ball, does not change.

Rather than look for a clear flow of control, look for the interactions of the ball, the walls, the paddle, the clock, the human, and time. These shifting values define the game.

You could take an object-oriented approach to this problem and define the paddle, the ball, the walls, and the out-of-bounds region as objects. When each object moves, you could loop through all the other objects asking what interaction there is.

Alternate Model I -- Short Signal Handlers

A timer tick occurs and control jumps to your timer handler. What does the timer handler do? It adjusts the counters for the x and y motion, moves the ball if those counters run out, and it might also update the elapsed time clock. The handler for the timer signal might, if the ball moves and the elpased time clock has to be updated, take quite a bit of time.

What if the time required to service one timer tick is longer than the intervals between timer ticks? The next timer tick will occur and be blocked. A backlog of timer ticks might even develop. What happens to user input? If the system is busy figuring out whether to move the ball and update the clock, the program might never get around to reading user input. User control of the paddle will be sluggish; the game ceases to be fun. Customers will demand refunds.

The Unix/Linux kernel faces the same problem. There are timer ticks going off every $1/100^{th}$ or every $1/1000^{th}$ of a second; there are disk drives reporting the delivery of requested data, there are network cards announcing arrival of new packets of data. If the kernel spent a lot of time in any one of these event handlers, it would miss other events and would be slow to respond to user keystrokes or mouse clicks.

The kernel addresses this problem by taking the approach of keeping all signal handlers as short as possible. It does so by changing the handler from doing the action to putting the action on a to-do-list. Now the handler can be very short. It adds the event (got a timer tick, got a keypress) to a queue and then returns. The main loop is then modified to handle keypresses and also to handle any events in the queue. By doing so, the event handlers can so quick that they are unlikely to running when the next signal arrives.

If you want to try this approach, you have to set the signal handling for the alarm signal to NOT restart getchar().

Alternate Model II -- Signals for Input and Timers

The bounce2d.c example implements multi-tasking using blocking for user input and signals for timer ticks. The text explains a method for using signals for user input and also for timer ticks. If you want to try this approach, use aio_read() and compile with -lrt (as shown in the text.)

Special Topic: Race Condition

The design of this game includes a race condition. The problem can be explained by the following example: imagine you press a key to move the paddle. The paddle-draw function then tells the terminal to move the cursor to a particular cell on the screen and then tells the terminal to draw some text on the screen. But what if something happens between the move operation and the draw operation. Imagine a timer tick occurs mid-sequence, and imagine that timer tick causes the the ball to move.

The ball_move function moves the cursor to a different location then draws the ball. When control returns to the paddle drawing function, the cursor is not where that function imagines it to be. Therefore, part of the paddle may appear next to the ball.

These two screen operations, move then draw, are analogous to the two file operations, lseek then write. For file descriptors, the kernel provides a solution with the `O_APPEND` flag. Curses does not provide a comparable feature. Avoiding data confusion when two parts of the program want to update the screen at once, therefore, must be solved by the programmer.

In designing your program, consider how you can prevent data confusion reliably. Occasional avoidance of race conditions is not a solution; an important feature of race conditions is that they are only occasional problems.

Please describe in your design document your approach to preventing race condition problems and why you believe it works.

What To Hand In

Required: (using `~lib215/handin pong`)

a) A README cover page
b) A design document
c) Source Code
d) Makefile for your project
e) A script of your program compiled with gcc -Wall

If you cannot get the entire thing working, get some subset working and explain what it is supposed to do.