Programming Project: pong

Introduction

For this assignment you will write a video game based on the classic coin-operated 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. In implementing this game, you will 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 six #'s), a ping pong ball (the O), and a three-sided court. The positions of the objects are given in 'curses' coordinates: upper left corner is (0,0), the first coordinate is the row, the second coordinate is the column. The court has corners at (4,70), (4,9), (21,9), (21,70). The paddle is exactly six rows tall and stays at column 70. Its top cannot rise above row 5, and its bottom cannot drop below row 20.

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, though, 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.

If the ball gets past the paddle, the ball is out of play. After three balls, the game is over.

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

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 should 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 contains 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.

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 ticks_to_move members is counter-intuitive. One usually thinks of speed as distance per time not time per distance. Using ticks per move ensures that the ball will move 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 )`. This method is not perfect, but it is simple and works well enough for this game.

Generate a random number in a sensible range for the x-timer and a random number in a sensible range for the y-timer. 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 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`.

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 still bounces against three walls. At the right side, though, you need to determine if the ball is hitting the paddle, or missing it.

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

Since `bounce_or_lose` controls the action of the ball, it should also start a new one if one goes out of bounds. 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 `bounce_or_lose` so that the speed is slightly, randomly, modified when it hits the paddle. The speed is controlled by the `ttm` member of the struct.

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 = getchar() ) != 'Q' ){
          if ( c == 'j' )
               down_paddle();
          else if ( c == 'k' )
               up_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 program may seem to be missing something. The main loop controls the paddle only. The ball appears nowhere in the main loop.

To understand what is going on, consider the following discussion. There are three objects in the game: the ball, the paddle, and the court. 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 'j' 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.

Rather than look for a clear flow of control, look for the interactions of the ball, the walls, the paddle, 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.

What To Hand In

Required:
a) Source Code
b) Makefile for your project
c) A design document called Plan

Extra Credit

Up to ten points of extra credit may be earned with a significant addition to the game. A two-player version, a multi-ball version, additional bumpers, and a paddle that moves in two dimensions are all examples of significant additions. Decision of the judges is final, void where prohibited.

Shared Resources, Race Conditions, Locking

This design, based on the bounce program, works pretty well. It works pretty well as long as you don't use it via modem or other slow lines. If you test your program over a slow line, you may find that the screen accumulates random garbage. This is not line noise; it is a symptom of a deeper problem.

This problem is caused by the fact that the paddle functions and the ball functions share the screen but operate independently. Consider the following sequence of events: Imagine you press a key to move a paddle. The paddle-draw function tells the terminal to move the cursor by sending a 'control sequence' to the terminal. Imagine that a timer tick hits in mid-sequence. The timer tick decides to move the ball, so it sends a different sequence to the terminal. This second screen update command appears in the middle of the previous command. The resulting comingled message makes no sense to the terminal. It ignores the muddled commands and prints 'garbage' characters on the screen.

You do not have to address or solve this problem. If you notice garbage when testing over a modem, you now know why. If you want to, you could build a solution into your program from the start. Regardless of what you do with this information, simply understanding the nature of this problem provides a good example of some of the issues in sharing resources among several processes.