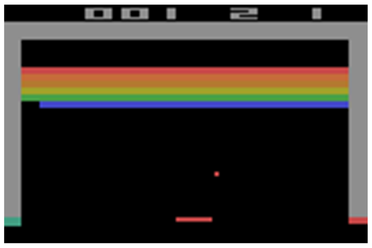
ECE 71 – Introduction to Computational Programming
Fall 2017 – Dr. Gregory Kriehn

FINAL PROJECT: Pong Breakout – It's a Fight for Survival!
Due By: Tuesday, December 12 (8:00 am – 5:00 pm)



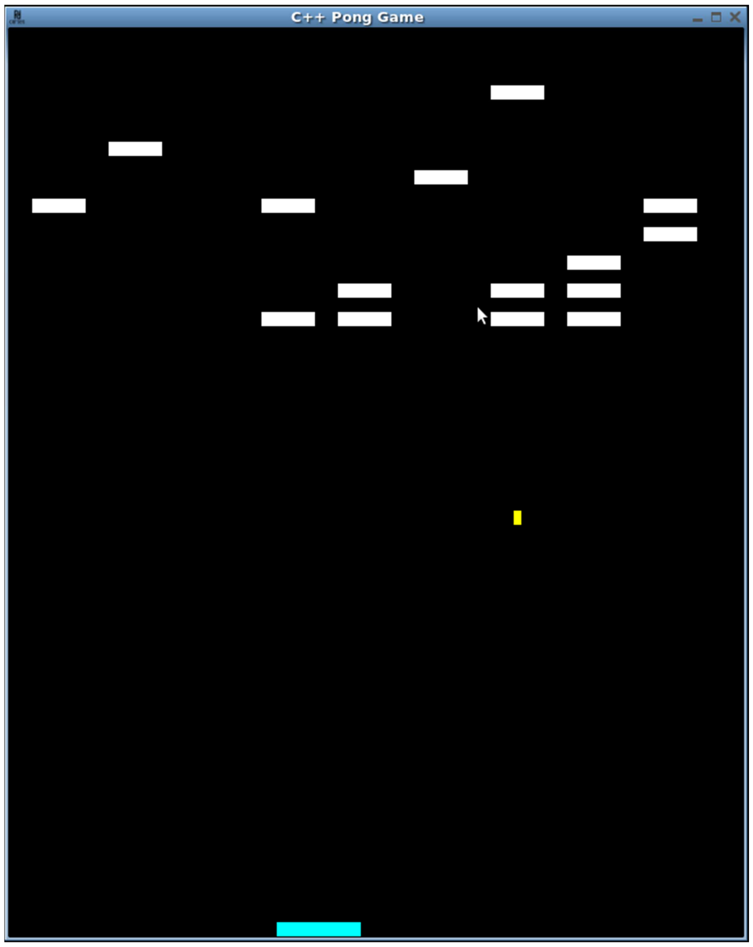
Expectations:

1. Good Attitude!
2. Fully Commented Code!!
3. Live Project Demonstration!!!

Teams:

2 Students/Team (Required)

It's the Holiday Season, and you desperately need a gift for your significant other. But being an engineering student, you have no money, no time, and no social life. But you do know how to code in C++! So what better way to provide love, joy, and holiday cheer than by creating your own Pong Breakout Game? It looks like your prayers have just been answered!



NOTES: <u>If you cheat on the project, you will fail the class!</u>

Breakout Rules:
Breakout involves deflecting a ball into bricks located at the top of the screen (thereby destroying them), using a paddle that moves back and forth along the bottom of the screen. If the ball hits the paddle, the ball is deflected upward where it has a chance to hit one or more bricks. The side walls, the top wall, and the bricks themselves deflect the ball as well. The bottom wall however, does not, which why is the paddle is required to keep the ball in play.

The core of the program is a simple loop that moves the paddle and ball each turn, and checks to see if the ball has collided with an object (the paddle, a wall, or a brick). If a collision occurs, the direction of the ball should be changed appropriately, and if a brick is hit, the brick should be removed from the game board. If all bricks are removed from the board while keeping the ball in play, the game is won. If a user fails to deflect the ball with the paddle at any point, the game is lost.

When using the keyboard to play the game, the left and right arrow keys can be used to move the paddle. Once a direction is chosen, the paddle will continue to move on its own unless it hits a wall, whereby it stops. The user can also stop the paddle by pressing the down arrow key.

If you are unsure about the mechanics of the game, watch the “ECE 71 Breakout Video” on the website. You can always play a game of Breakout typing “Atari Breakout” while performing an image search on Google. No, really. Try it.

Game Play Mechanics:
In the original Atari version of the game, the further the ball hit toward the edge of the paddle, the sharper the angle of deflection. If the ball hit in the exact center of the paddle, then the angle of deflection was set to 0° relative to the normal of the paddle. This deflection behavior defies the laws of physics, but in keeping with the spirit of the game, I suggest implementing this feature into your code so that your game follows the original arcade game as closely as possible. I chose to limit my angles of deflection to be between 0° and 75° relative to the normal, based upon how far away the ball hit from the center of the paddle.

If the ball hits the right, left, or upper wall, the ball should be reflected appropriately – meaning that only the direction (sign) of the movement in the `x` or `y` should be changed. To start the game, an initial angle may be chosen at random for the ball. Each turn, the ball should be moved incrementally by an amount `dy` in the `y` direction and `dx` in the `x` direction such that the angle is $\tan^{-1}(dy/dx)$. The overall magnitude of the movement should be 1, such that $\text{sqrt}(dx^2 + dy^2) = 1$. However, because the character used to represent the ball is not square, the ball will appear to travel in the vertical direction much faster than the horizontal direction due to the character being taller than its width.

Once you determine how to move the paddle and ball, the calculation of a collision with the paddle, walls, or a brick is not terribly difficult. However, determining the appropriate angle of deflection off the top, bottom, or side of a brick can be very confusing. Therefore, it is not expected that your deflection physics perfectly work for your project, but they should work most of the time. The better your deflection physics, the more extra credit you will receive on your project grade.

Specifications:
Develop a program that allows a user to play a game of PONG BREAKOUT. We will be using a rudimentary graphics library called “PDCurses” to support ASCII graphics for the project (start reading

through the documentation file immediately). To begin, use the following global constants, enumerated data type, and structure to store some relevant game information:

```
const int NROWS = 64;             // Number of Rows in Game Board
const int NCOLS = 96;             // Number of Cols in Game Board
const int padLen = 11;            // Paddle Length (Can be Varied)

enum Dir {LEFT, DOWN, RIGHT};     // Keyboard Directions

struct info
{
     int boardSize[NROWS][NCOLS];       // Stores Block Placement Game Data
     int paddleLength;                  // Stores Paddle Length
     int speed;                         // Stores Game Speed
     ...                                // More Variables May Be Added
};
```

If necessary, you may add additional data members to the structure, depending on what grade you are trying to achieve (such as extra credit). These declarations should be placed in "`init.h`", which is to be used as an initialization header file.

You will need a structure object to make use of the "`info`" structure you just created. Let's allow the structure variable to be called "`gameInfo`" and to be globally accessible across all source files, meaning that wherever your global constants are declared (in the "`init.h`" header file), you should also include the following declaration:

```
extern info gameInfo;
```

This will tell the compiler that the `gameInfo` object has been declared elsewhere (in one of your source files), and has the properties of an external (global) variable. I chose to formally declare `gameInfo` in `main.cpp` using the following declaration outside of the main function:

```
info gameInfo;
```

Since `gameInfo` is declared in `main.cpp`, but is referred to via the `extern` keyword in init.h, a simple directive:

```
#include "init.h"
```

is all that is necessary in each source or header file requiring access to `gameInfo`, the declaration of the structure data type, the enumeration, or the declared constants.

Your `main()` function should perform the following tasks:

1. Declare the structure object `gameInfo` with file scope.
2. Open a file for saving relevant game information.
3. Initialize the XCurses library using a function.
4. Initialize the colors using a function (if needed).
5. Initialize the `gameInfo` object using a function.
6. Print a Welcome Screen using a function.

7. Declare `paddle`, `ball`, and `block` class objects.
8. Fill the game screen with bricks using a class member function.
9. Create a loop to play the game:
    a. Print the paddle location to the screen with a class member function.
    b. Move the paddle with a class member function.
    c. Print the ball location to the screen with a class member function.
    d. Move the ball with a class member function.
    e. Sleep for a certain number of microseconds before the next iteration begins.
10. Print the results of the game to the screen using a function.
11. Print the game information to a file (if needed).
12. End the program.

Regardless of whether you are coding to receive and A, B, C, or D, the code base is significantly complex that it warrants breaking the program into separate files. (Therefore, attending the last lectures is critical toward the successful completion of the project.) Header files will need to be created to store the following information, as necessary:

1. Inclusion of Additional Header Files
2. Constants and/or Macros
3. Global Data Types
4. Enumerations and Type Definitions
5. Function Prototypes
6. Structure Data Type Definitions
7. Class Data Type Definitions

Personally, I used the following header files for my program:

```
balls.h       // Declares a "balls" class to store relevant ball information
blocks.h      // Declares a "blocks" class to store relevant brick information
color.h       // Stores color code constants and color function prototypes
display.h     // Stores display function prototypes
init.h        // Stores global constants, structure data types, enumerations,
              // global data types, and function prototypes for initialization
paddles.h     // Declares a "paddles" class to store relevant paddle information
```

NOTE: `init.h` should be used to define all global constants (like the variables for storing the number of rows and columns on the screen, the structure definition, enumerations, global constants/variables, etc.), and several initialization function prototypes.

Function definitions and class member definitions should be placed in appropriate source files:

```
balls.cpp           // Contains all "balls" class member function definitions
blocks.cpp          // Contains all "blocks" class member function definitions
color.cpp           // Contains all color functions (optional)
display.cpp         // Contains all display functions
init.cpp            // Contains all initialization functions
main.cpp            // Contains the main function
paddles.cpp         // Contains all "paddles" class member function definitions
```

Be sure to include the appropriate library and header files in each source file, as needed.

NOTE: The PDCurses graphics library is contained in <xcurses.h>. Therefore, whenever access to the functions supported by the library is required, the XCurses library header file should be included with:

```
#include <xcurses.h>
```

I used the following functions to support my gameplay:

BALLS.CPP CLASS MEMBER FUNCTIONS:

```
Public Members
   // Default Constructor
   balls::balls()

   // Print Function used to Print the Ball to the Screen
   void balls::print()

   // Move Position Function used to Incrementally Move the Ball
   void balls::movePos()
```

Private members of the class include variables to store the (`y, x`) location of the ball, variables to store the previous location, variables to store the `dy` and `dx` increments, and a variable to store the current ball angle. You may add to the class definition, as appropriate. Recall that the class definition occurs in `balls.h`, and contains the declarations for the function prototypes and the data members only. The class member function definitions are placed in `balls.cpp`.

BLOCKS.CPP CLASS MEMBER FUNCTIONS:

```
Public Members
   // Default Constructor
   blocks::blocks()

   // Function to fill the game board with bricks
   void blocks::fillBlocks()
```

Private members of the class include variables to store the brick width, spacing, number of rows, and margin spacing. You may add to the class definition, as appropriate. Recall that the class definition occurs in `blocks.h`, and contains the declarations for the function prototypes and the data members only. The class member function definitions are placed in `blocks.cpp`.

COLOR.CPP FUNCTIONS:

```
// Initialize Color Attributes
void initColors(void);

// Set Color Function used to Set the Color of an Object
void color(int x);
```

DISPLAY.CPP FUNCTIONS:

```
// Welcome Screen
void welcome(void);

// Win Screen
void win(void);

// Loss Screen
void loss(void);
```

```
// Game Over
void gameOver();

// "Press any Key" Function
void keypress();
```

INIT.CPP FUNCTIONS:

```
// Initialize XCurses
void initXCurses(int argc, char** argv);

// Initialize gameInfo Structure Data
void initGameInfo(ostream &fout);
```

NOTE: To initialize the PDCurses library to use XCurses (since the server uses Linux and creates X graphics windows), the first line of your `init.cpp` source file **<u>must</u>** be:

```
#define XCURSES
```

NOTE: When initializing the XCurses library with the `initXCurses()` function, you will need to:

1. Initialize the X Display Screen via `Xinitscr()`
2. Set a Screen Title via `PDC_set_title()`
3. Retain Ctrl-C behavior via `cbreak()`
4. Adjust the cursor display via `curs_set()`
5. Turn on the keypad via `keypad()`
6. Turn off echoing of the input via `noecho()`
7. Start NCURSES colors (optional) via `start_color()`
8. Adjust the delay of the screen via `nodelay()`
9. Determine if the screen has been resized (optional) via `resize_term()`

MAIN.CPP FUNCTIONS:

```
// Main Function
int main(int argc, char** argv);
```

PADDLES.CPP CLASS MEMBER FUNCTIONS:

<u>`Public Members`</u>

```
// Default Constructor
paddles::paddles()

// Print paddle to the screen
void paddles::print()

// Print paddle information to cout/file
void paddles::info(ostream &out);

// Move Paddle Function
void paddles::movePos()
```

Private members of the class include variables to store the full paddle position, in the `x` and `y` directions, the paddle width, and the paddle direction. If the paddle direction is `LEFT` (defined by the enumeration), the paddle should continuously move to the left. If the paddle direction is `RIGHT`, the paddle should continuously move to the right. If the paddle direction is `DOWN`, the paddle should stop. You may add to the class definition, as appropriate. Recall that the class definition occurs in `paddles.h`, and contains the declarations for the function prototypes and the data members only. The class member function definitions are placed in `paddles.cpp`.

NOTE: If you need additional functions to support the expansion of your game, please add them! However, the base functions noted here must be used "as is" if your code requires their use.

NOTE: Several of the header/source files may need to declare external global or static variables, constants, macros, etc. to keep track of appropriate information.

NOTE: All block, paddle, and ball characters may be printed to the screen using the `ACS_BLOCK` character, which is supported by the PDCurses graphics library.

Beyond what is outlined here, you are free to expand upon the complexity of the game in whatever way you see fit, but the core functionality of the game play must remain intact. It is also strongly recommended that you print a significant amount of debugging information to `cout` to help you see what is happening while the ball is in play.

GRADING REQUIREMENTS

D Level Program:
1. No Welcome or Win/Loss Screen is required.
2. No information needs to be written to a file.
3. Quitting the game at any time is not required.
4. Collision physics is poor, but the ball should always reflect off the walls and paddle correctly.

C Level Program:
1. Meet all D Level Requirements, and ensure the program works.
2. Include a Welcome and Win/Loss Screen.
3. Include an option 'Q' for quitting the game at ANY time.
4. The collision of the ball and paddle should include variable deflection angles.

B Level Program:
1. Meet all C Level Requirements, and ensure the code works and is easy to follow.
2. Write relevant information to a `savegame.txt` file.
3. Improve the collision physics such that the ball reflects off the bricks correctly most of the time.
4. Clean up the game presentation and user interface.

A Level Program:
1. Meet all B Level Requirements, and ensure the code works, is easy to follow, and is pretty.
2. Include color for your welcome screen, win/loss screen, and for the game itself.
3. Improve the collision physics such that the ball reflection physics is almost always correct.
4. Polish the game presentation and user interface so that it is aesthetically pleasing.

Extra Credit (Include One or More of the Following):
1. Allow the user to play multiple games. Include a game score and a high score.
2. Allow the user to forcibly increase/decrease the ball speed using the keyboard.
3. Perfect the deflection physics of the ball off of all objects. You must consider all heights/widths.

4. Introduce temporary power pellets for the paddle at random times, which provide buffs such as:
    a. Introducing 2 paddles, which use the “ASD” and “JKL” characters for movement.
    b. Introducing 2 or more balls into play.
    c. Allowing the paddle to fire at the bricks.
    d. Slowing down the ball speed.
    e. Etc.
5. Introduce temporary debuffs pellets for the paddle, such as:
    a. Shortening the paddle length
    b. Increasing the ball speed
    c. Randomly introducing new bricks into the game.
    d. Etc.
6. Create ASCII animations for your Welcome and Win/Loss Screens.
7. Create an ASCII Configuration Menu to adjust game play options (Not for the Faint of Heart).
8. Optimize the game presentation and make it gorgeous.
9. Etc.

Note: Any Extra Credit functionality added to your program will contribute points to your **OVERALL, FINAL GRADE**.

MOST IMPORTANTLY: Start designing and coding your solution NOW.