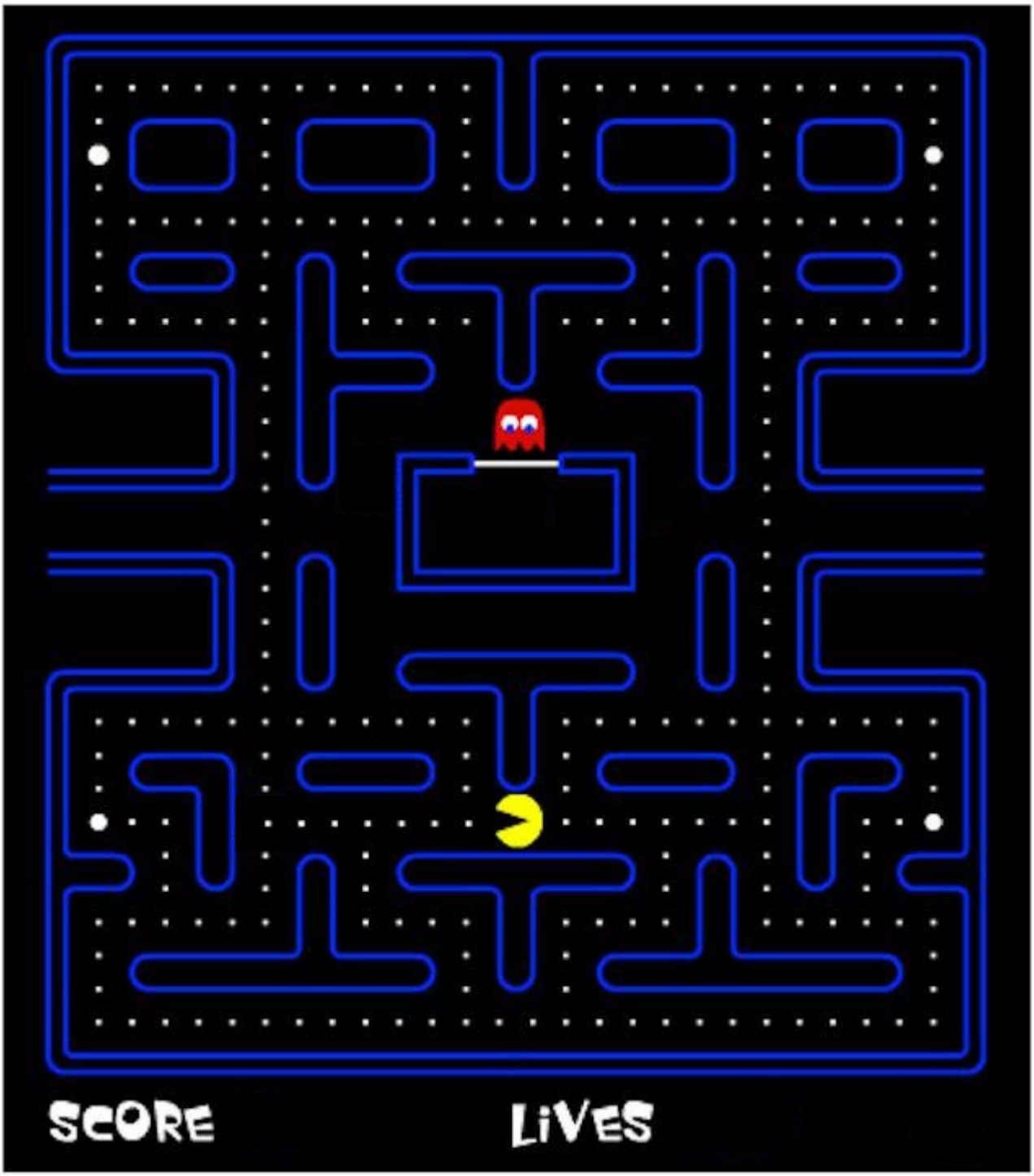
ECE 4122/6122 Lab 2: 2D Graphics & SMFL

(100 pts)

Due: Wednesday Oct 26th, 2022 by 11:59 PM



Problem 1: Pacman Game (100 pts)

Write a C++ application that uses SFML to create a simple pacman game.

Game Rules:

- (~5 pts) Display the provided pac-man.bmp at the start of the game with the extra text
  - PRESS ENTER TO START

  in the middle of the window.
- (~20 pts) Once the user presses the ENTER key the game starts with
  - pacman located as shown above
  - all four ghosts start at the same location as shown above (bitmaps provided)
  - place the four power ups as shown above
  - place the coins as shown above
- (~10 pts) The ghost's initial direction is determined randomly and when a ghost hits a wall its new direction is chosen randomly.
- (~20 pts) Control the location of pacman using the left, right, up, down arrow keys.
- (~10 pts) Determine a game speed that makes the game fun to play (not too slow and not to fast)
- (~10 pts) The game ends when
  - one of the ghosts comes in contact with a non-powered up pacman
  - pacman consumes all the coins
  - user presses the escape key.
- (~5 pts) When pacman eats a powerup he can eat ghosts for 5 seconds. Ghosts that are eaten are gone forever.
- (~15 pts) Pacman and the ghost cannot go through walls.
- (~5 pts) Pacman and the ghost can go through the middle tunnels on each side.
- You do not have to handle resizing the window. Just use the image size.

Suggestions:

- Use invisible rectangles to represent the location of the walls.
- Use circles to represent the coins and power ups.
- Use vectors to hold locations of coin, powerup, and wall objects
- Use the Chapter 5 example from *Beginning-Cpp-Game-Programming-Second-Edition* as a starting point.
  https://github.com/PacktPublishing/Beginning-Cpp-Game-Programming-Second-Edition/tree/master/Chapter05

Turn-in Instructions:

Two methods:

1.
    a. Upload a video(s) of your game running and demonstrate the "Game Rules" above. Multiple smaller videos are fine, just make sure that the name is representative of the game rule being covered.
    b. Put all the code files you created into a zip file called ***Lab3.zip*** and upload to canvas.
2.
    a. The TAs will build and run your code.
    b. Use the Chapter 5 example from *Beginning-Cpp-Game-Programming-Second-Edition* to develop your code. Remove the existing timber.cpp file and place your new source code files in the *code* directory. Once you have finished your development, zip the Chapter 5 folder with your changes into a zip file called ***Lab3.zip*** and upload to canvas.

Grading Rubric

If a student's program runs correctly and produces the desired output, the student has the potential to get a 100 on his or her homework; however, TA's will **randomly** look through this set of "perfect-output" programs to look for other elements of meeting the lab requirements. The table below shows typical deductions that could occur.

AUTOMATIC GRADING POINT DEDUCTIONS PER PROBLEM:

Element	Percentage Deduction	Details
Does Not Compile	30%	Code does not compile on PACE-ICE!
Does Not Match Output	Up to 100%	The code compiles but does not produce the correct outputs. See point values above.
Clear Self-Documenting Coding Styles	10%-25%	This can include incorrect indentation, using unclear variable names, unclear/missing comments, or compiling with warnings. (See Appendix A)

LATE POLICY

Element	Percentage Deduction	Details
Late Deduction Function	score – 0.5 * H	H = number of hours (ceiling function) passed deadline

Appendix A: Coding Standards

Indentation:

When using *if/for/while* statements, make sure you indent 4 spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

```
for (int i; i < 10; i++)
{
    j = j + i;
}
```

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i; i < 10; i++)
{
   if (i < 5)
   {
       counter++;
       k -= i;
   }
   else
   {
       k +=1;
   }
   j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird). This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names:

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of “imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is the idea behind self-documenting code.

File Headers:
Every file should have the following header at the top

```
/*
Author: your name
Class: ECE4122 or ECE6122 (section)
Last Date Modified: date

Description:

What is the purpose of this file?

*/
```

Code Comments:

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.