CSC3221 Project 2

Welcome to the practical classes for CSC3221. These classes are built to complement and drive the lectures. The practicals are very important for a programming module. Things which can seem clear during the lectures may just appear not to work at all when in front of a machine. Conversely concepts which seem intractable when described are obvious when seen.

Module Assessment

There are 3 pieces of assessment for this module:

1. Project 1 which is submitted to NESS. (20)
2. Project 2 which is submitted to NESS. (30)
3. An exam in January. (50)

Project 2 Ultimate Chess (Deadline: 4pm, Wednesday 11th December, 2019)

Aims

To practice implementation of an inheritance hierarchy and use of polymorphism in C++ and introduce ideas of collision detection in computer games.

Specification

An important topic in many computer games is determining when 2 object collide with one another. Games looks poor if a car appears to drive into a wall or appears to collide with thin air. Physics Engines devote considerable effort to solving this problem. In this project we will start to consider this issue.

Implement an abstract base class `Piece` to represent moving chess pieces in a game. Derive classes `Rook`, `Bishop` and `Queen in an` appropriate inheritance hierarchy from `Piece` with appropriate information to store the positions on a flat, square 2_D surface. The classes should provide methods to move the position of the piece. A `Rook` moves vertically or horizontally across the grid. A `Bishop` moves diagonally (at 45 degrees to the movement of the Rook) in any direction. A `Queen` can move either like a Rook or a Bishop (as in real chess). Distances moved need not be integers.

In this game pieces capture by colliding with another piece during the course of their move. For the purposes of capture, pieces have a 2 dimensional shapes (a circle for Bishops and Queen and a Square for rooks). All pieces have a radius of 1 unit or a side length of 2 units in the case of the rook. You will need to provide functions to detect when two pieces overlap whereupon the non-moving piece is removed from the board. You will need to deal with overlap of 2 circles, 2 squares and circle and square separately.

Write a test game program that creates several pieces and stores them in an array or list or other data structure of your choosing. The program should then iterate through the data structure, moving each piece by a small random amount (but keeping them within a grid of a given size) and check to see if there are any captures. Keep a score of the number of captures made by each of the 3 types of piece. If a capture is detected then the non-moving piece should be removed from the data structure and the moving piece should stop at the position where the capture is detected. The game should continue for a given number of moves by each piece (say 100 or whatever you deem appropriate) but should terminate earlier if at most one piece remains.

NO GRAPHICAL OUTPUT is expected or required and providing some will not score any extra marks (this isn't a graphics course!)

You may choose whichever method you want for keeping the shapes within a grid but a piece should always move some distance at each turn. You should output (using operator<< appropriately) information about each capture in the game and present the final scores for each category of piece at the end of the game.

Note that, unlike real chess, there are no squares on the board but simply distances.

Deliverables

A zip file of the relevant project in the projects directory of Visual Studio containing the following classes:

Piece.h
Piece.cpp
Rook.h
Rook.cpp
Bishop.h
Bishop.cpp
Queen.h
Queen.cpp
Game.cpp
Any other classes you wish to use
Sample output

Mark Scheme

Pieces (8)	
Basic Piece, Rook, Bishop and Queen classes:	5
Inheritance Hierarchy:	3
Collision Detection (8)	
Circle/Circle	2
Square/Square	2
Circle/Square (or Square/Circle)	4
Game (14)	
Creation deletion and storage of pieces	3
Correct removal of pieces	2
Finding the type of a shape	2
Outputting Piece information	2
Overall correctness of solution	2
Sensible and convincing testing	3
Total:	30