Simulation Billiards (Pool) Game

Report

Jak Waterfall

N0906911

SOFT10101

Computer Science Programming

Coursework 2020/21

Specification

This project is a game that simulates the classic pub game Billiards (pool as its sometimes known). The game uses the SDL2 framework to render all the game objects to a window and capture input events. The menu system is done via the command prompt as there is no easy widget capabilities in SDL and I wished for the project to be based on the game of Billiards itself not a custom-made menu system.

Key features:

* Simulate Billiards.
* Control game via mouse.
* Different difficulty settings (change size of pockets).
* Keep track of player information (2 player game).
* Play sound effects.
* Save and resume game from file.

Aims and objectives:

* The game will simulate all the balls on the snooker table.
* The game will be rendered in a top-down view of the table, all the balls will be visible and different colours shown.
* The mouse will act as the pool cue to aim and direct your shot with the white ball.
* The power of your shot will be determined by how far away your mouse curser is to the white ball.
* The game will simulate the balls colliding with each other, the bounds of the table and pockets of the table (play sound effects on each collision).
* The game will keep track of whose turn it is, how many shots they have left, and which colour ball is theirs to hit.
* The game will present the current players information on screen to indicate who’s turn it is, which colour ball they need to sink, how many shots they have left, and which balls have been potted and in what order (display balls).
* The game will intrepid when a foul ball/shot have been taken and resolve accordingly (ending the current players turn and giving the opposing player 2 shots).
* The game will allow the player to place the white ball anywhere within the droppable area (behind the line) as the rules state, at the beginning of the game and whenever the white ball is pocketed.
* The game will follow all the rules of Billiards / Pool.
* The game will allow the player to save a game in progress and resume the game later. (the state of the table, positions of all the balls and player state will be recorded to a file)
* The game will end once the black ball is pocketed.
* The simulation will run at a constant rate of 60 FPS to ensure the game runs at the same speed on any computer.
* A vector class will be made to hold all the logic and mathematics of the simulation.
* All elements required by the project spec will be implemented to create this game such as loops, arrays, classes, file handling etc.
* A class hieratic will be created for the sphere objects that will inherit from one another. White ball inherits from the base Ball class etc.

Design and Implementation

File Structure

The file structure used to save and resume the game are two separate txt files, ‘balls.txt’ and ‘player.txt’.

‘balls.txt’ saves the position and colours of all the balls on the table in this pattern:

* x coordinate
* y coordinate
* colour

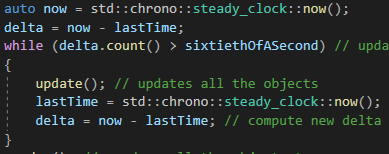
‘player.txt’ saves the variables required by the Players object that manages the game state, such as shots left and who’s turn it is and the colour of each potted ball for the display. They are saved in this pattern:

* isPlayer1Turn
* arePlayerColoursSetup
* player1 Shots Left
* player2 Shots Left
* player1 Colour
* player2 Colour
* Display ball colour 1
* Display ball colour 2
* Display ball colour 3… etc

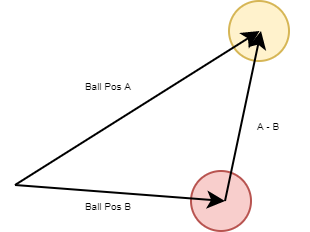
60 Updates Per Second

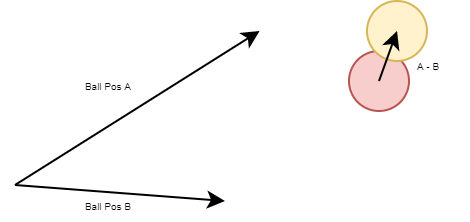
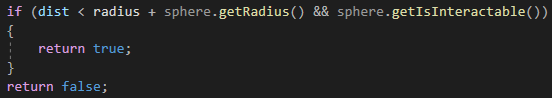
To keep the simulation speed equal on all machines and not tied to how fast a computer runs, I have tied the update speed to the clock. All the objects in the game are updated 60 times a second, this was achieved by taking a timepoint when the update function has been called. I called this “lastTime” as it was the last time the update function was called:

Then the program continues, and on the next cycle just before we get to the update function it creates another timepoint called “now” and calculates the difference (delta time) between them. If the delta time between the last time it called the function and now is greater than 1/60th of a second, then the update function is called and a new delta time is calculated until the delta time is less than 1/60th of a second. This process repeats every cycle.



Pool Ball Collision

The main process that makes this game possible is how the pool balls interact with each other when they collide. To give the illusion of pool balls bouncing off each other, first it creates a vector which points from the centre of 1 ball to the centre of another ball that you wish to check for a collision. This is done by taking the position vector from one of the balls away from the position vector of another ball.

Then the magnitude (or length) of that vector is calculated. We can postulate that the two balls are currently colliding with each other if the distance between their centres (magnitude of vector A - B) is less than the radius of each ball added together.

If the balls are colliding it takes that vector (A - B) and sets its magnitude to a to a constant size (“ballCollisionStrength”) that I tested to be a good fit for collisions and adds it to the ball’s velocity vector.

As the vector from before (A -B) is still pointing in the direction of ball A, when it is added to ball A’s velocity vector, ball A is pushed in the direction it was hit from giving the illusion of pool balls bouncing off each other. Many collisions can happen at a time so that if two balls only graze each other, this may result in being pushed once by the constant value of “ballCollisionStrength” but if two balls smash into each other and collide for many cycles, “ballCollisionStrength” is added to their velocity vector many times over giving the illusion of the balls being smashed very far when hit by a fast ball.

Class Diagram

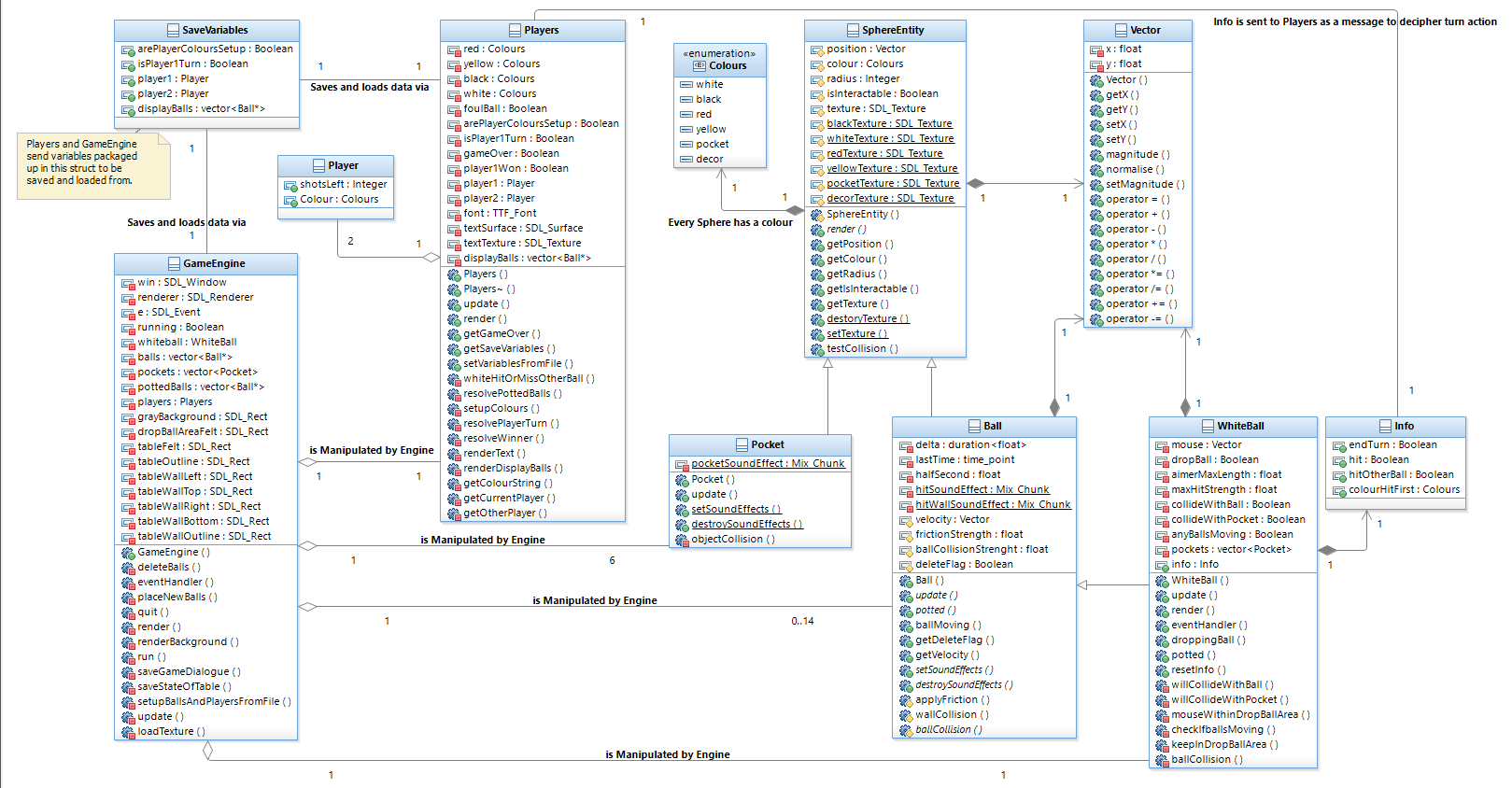
The class diagram shows the relationship between all the classes that make up the game. The game contains 7 classes, 3 structs and 1 enumeration. At the heart of the simulation lives the sphere hierarchy that models all the spherical objects.

The SphereEntity class is an abstract class which cannot be instantiated by itself and is a base for all other derived classes; it holds variables such as the position, colour, texture and radius of each sphere and functions that render it too screen or test if two spheres collide (which is very important in the game of pool).

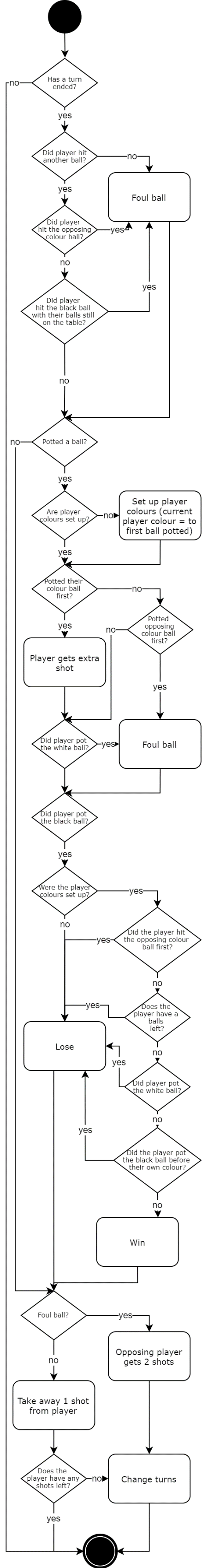
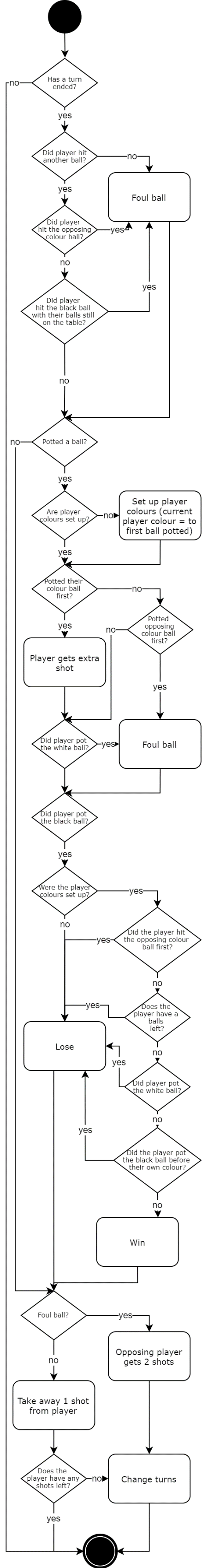
From SphereEntity, the Pocket and Ball classes are derived, the Pocket class holds all the logic for the side pockets in the game and when a ball collides with it; removes that ball from play. The Ball class holds all the logic and algorithms for simulating pool balls on the table, including colliding with walls, other balls and general movement via a velocity vector which is added to its position and slowed down over time via simulating friction.

The WhiteBall class is derived from the Ball class and has all the methods required for taking a turn in the game as a game of pool is played by hitting the white ball with a cue. The white ball reads the input events from the mouse and allows the player to strike and drop the ball as needed. Over the course of a turn the white ball builds an “Info” struct and sends it off to the players object for interpreting what happened each turn.

The Players class keeps track of the game state such as whose turn it is and how many shots are left. Finally, the GameEngine class holds the main game loop, which is handles all the input events, updates all the objects (at 60 updates a second) and renders all the objects to the screen.



End Turn Activity Diagram

The activity diagram outlines what happens at the end of each turn. All the logic is located within the players object which first determines if the current player has made a foul ball or if they had potted their own ball. Secondly, if the player has potted the black ball, it resolves who just won the game based on a few variables such as what balls are left on the table and which balls and in what order were potted that turn. Finally based on its conclusions it determines whether to give the opposing player 2 shots and if to change turns.

Problems Encountered

Some of the bugs I encountered and had to squash were a bug with dropping the white ball straight into the pocket which would mess up the player turn order. I resolved this by giving the white ball a copy of the vector of pockets so that it could test against colliding with them before dropping the ball. I had a bug were giving the save game prompt a big string of characters resulted in printing the “Incorrect answer chosen.” message multiple times. This was resolved by using a std::string instead of char\* as well as clearing cin function.

Another bug I had was if a player had no balls left on the table and they hit the black ball, 2 shots was given to the opposing player. I resolved this by looping through all the balls on the table and checking if the player has any balls left. I had a bug where you could strike the white ball while other balls were moving which you cannot do in the rules of pool, so I resolved this by looping through all the balls on the table and testing if their velocity vector was (0,0) on each attempt.

Finally, I had a similar bug to the pockets bug where a player could drop the white ball straight onto another ball which would cause the balls to not react properly and is not allowed in the rules. This was resolved by looping through all the balls on the table and testing if the white ball collides with any of them.

Test Plan

|  |  |  |
| --- | --- | --- |
| Test | Method | Success Criteria |
| Resume game from file (if saved game exists). | Enter ‘y’ when asked. | Last saved game should be resumed from file. |
| Start a new game. | Enter ‘n’ when asked if old save is present. | Balls should be in the starting triangle arrangement with white ball in hand. |
| Exit game. | Press the red ‘X’ button on the window. | Game should close and if the game was over, the option to save the game should be present. |
| Save the game. | After exiting chose the option ‘y’ to save the game to file. | The state of the game should be saved to file. To test this open game again and choose to resume the game. |
| 3 different pocket sizes for difficulty modification. | Choose 1,2 or 3 when asked on start up. | 1 gives small pockets.  2 gives medium pockets.  3 give large pockets. |
| White ball can only be dropped behind the line. | Mouse click to drop the ball. | White ball is placed onto the table but only behind the line. |
| White ball cannot be placed onto another ball or pocket. | Mouse click to drop the ball over a pocket or another ball. | White ball is will remain on the cursor and not placed on the table. |
| Strike the white ball. | Mouse click to strike ball in direction of aimer. The longer the aimer the more force is used. | White ball travels in direction of aimer line. |
| First ball potted becomes that players colour. | Pot a ball at start of the game when you have a free colour. | The colour of the ball you pot will become your colour. |
| If the player hits the opponent’s ball first, they give them 2 shots. | Player with colour red hits a yellow ball first or vice versa. | Opposing player gets 2 shots. |
| Cannot place white ball when other balls are moving | Mouse click to drop the ball when other balls on the table are moving. | White ball cannot be dropped. |
| Cannot strike white ball while other balls are moving | Mouse click to strike ball when other balls on the table are moving. | The white ball does not move. |
| Potting own coloured ball first. | When colours are set up pot the colour ball that is associated with the player. (so red player pots a red ball first) | The player will get 1 extra shot. |
| Potting opponents coloured ball first. | When colours are set up pot the colour ball that is associated with the opposing player. (so red player pots a yellow ball first) | The opposing player will get 2 shots. |
| Not hitting another ball with the white ball. | Strike the white ball in a way that will miss all other balls. (suggest low power) | The opposing player will get 2 shots. |
| Hitting the black ball first unless you have potted all of your colours gives the opponent 2 shots. | Hit the black ball with the white ball first while having at least one of your coloured balls left on the table. | The opposing player will get 2 shots. |
| Potting the black ball before you have a colour. | Pot the black ball while the player has a free colour. | The opposing player wins. |
| Potting the black ball with your colour balls still on the table gives you a loss. | Pot the black ball while at least one of your coloured balls are still on the table. | The opposing player wins. |
| Potting the black ball after potting all your coloured balls gives you a win. | Pot the black ball after potting all your coloured balls. | The player wins. |
| Potting the white ball on the same turn you pot the black gives you a loss. | Pot the black and white ball in the same turn. | The opposing player wins. |
| Potting the black and then potting your last colour ball in the same turn gives you a loss. | Pot the black ball before potting your last-coloured ball in the same turn. | The opposing player wins.  The black ball must always we potted last. |
| Any non-white balls potted should be displayed in the correct order they were potted in. | Pot any non-white coloured ball. | The ball should appear in the display. (white balls potted should not) |