[Game concept: 8-ball pool]

For this assessment I had originally taken upon the idea of creating a snooker game; however, with further consideration into the game's rules and logic complexion, I had decided to adapt to a pool game concept, which seemed more comprehensive. Choosing a pool game enabled me to aspire to a concurrent mobile game '8 Ball Pool', and had enabled me to better understand game mechanics, the types of physics involved, and the ratio of physic forces applied to objects. Relative to physics and game mechanics, my 8-ball pool game project incorporates the 'Box2D' physics engine and the 'SFML library' to achieve a full functioning pool game; with graphical, auditory and physical aspects. I had chosen this engine and library due to academic familiarity and popularity of external implementation.
Regarding object-orientated (OO) design, I had many considerations into fully abstracting my project, by means of creating a base class, such as a game controller; alternatively, I had used the 'main' class to achieve all my functionality. I had originally used the main class as a network to test my implementation of key press and release processing, this was relative to the transform of Box2D bodies (b2Body). However, as I continued to implement game mechanics, my code had become more advanced and more difficult to abstract class functionality into a base class. As a restraint, my unit testing is limited, to only accessing functionality outside of the main class; and reuse is not as coherent as it could potentially be. On the other hand, my project code is easily interpretable, I have fluently abided by naming standards for all my variables and incorporated 'pragma regions' to section my code. Also, computational performance proves to be efficient (see performance profile testing), and I have attempted to keep loading files into memory (such as textures) at a minimum for memory expense considerations.
Meanwhile, relating back to abstraction, each class initialises and instantiates variables and functions relative to itself and is accessed in the main class via creating game objects corresponding to each class. In doing so, enables my publicly assigned variables and functions from my classes to be accessible within the main body. Within the main class, an instance of b2World is created, all my sprite and shape objects are created and have their parameters set via passing by value and some instances by reference, my textures are loaded from file into memory, text objects and sound objects are set or called by function, the game logic is instantiated, a ray is casted from the cue ball, key pressed and release processes are assigned to key specific operations, b2Body data and contact listener are set as well as object update and b2World step functions to update my objects and Box2D physics, per passing frame. Inevitably I have implemented a vast amount of game-appropriate mechanics.
Relating to inheritance, my project incorporates inheritance on a superficial level. For contact listening to function between two fixtures, my 'BallEdgeListener' class had to inherit from 'b2Contact' which operates to manage contact occurring between two fixture bodies. Integrating contact listening, enabled my project to precisely output sound objects on a begin-contact basis. In which, 'BallEdgeListener' also inherits from the 'UI' class, enabling all sound object functions to be accessible and called upon, in instances of contact during program runtime. To achieve this, the contact listener utilises a combination of body-to-body contact as well as body-to-sensor contact. I did not require an end-contact function. Meanwhile, all of my text, sprite and shape objects are drawn in the main class (window.draw()) and do not require to be rendered to a target; the sprite and or shape can be returned to draw, my project does not inherit from 'sf::Drawable', it includes 'SFML graphics' to achieve such.

[Performance profile testing] See performance profile reports in GitHub repository:

CW-xGliff/Unit Test/Peformance Profiler/CPU profiler files (CPU usage)
CW-xGliff/Unit Test/Peformance Profiler/CPU profiler files (Memory usage)

Overall my performance profile reports conclude efficient use of CPU and memory (RAM) usage from program runtime testing. CPU usage maximised at 37% at render window launch and fluctuated between 24-27% for the remaining runtime (five minutes). Meanwhile memory usage maximised and maintained a 142-143mb consumption during the render window launch and remaining runtime.

[Blackbox testing]

Below are the Blackbox styled tests which I had conducted, these tests purposed to test functionality not possible by using unit testing (see grid below).

Case	Summary	Process	Actual result(s)	Expected result(s)
1	Cue sprite pullback, using 'Down' key.	Down key pressed (and held).	Cue sprite retracts from cue ball parallel to time held down, before reaching maximum pullback (stops).	Cue sprite retracts from cue ball and stops retracting when the set maximum pullback value is met.
2	Cue sprite rotates, using 'Left' and 'Right' keys.	Left and right keys pressed (and held), separately and simultaneously.	Separate: cue sprite rotates at an increment relative to the time pressed and held. Simultaneously: cue doesn't rotate.	Separate: cue sprite rotates in either direction. Simultaneously: cue doesn't rotate.
3	Playable balls can be potted and appear in allocated UI holes for either player.	Strike balls with cue sprite when pulled back to apply linear velocity to cue ball, aiming for balls targeted at table pocket sprites.	Balls collide with table pockets, position of balls is set correctly for each player.	Balls collide with table pockets, balls position based on player ball type assignment.
4	Cue ball transform resets when potted (collides with table pockets).	Strike cue ball with cue sprite when pulled back to apply linear velocity to cue ball, aiming for table pocket sprites.	Cue ball collides with table pocket(s), position of cue ball reverts to original position.	Cue ball has a new set position and appears at its original position.
5	Textures are correctly loaded from file and are applied to set shape and sprite objects.	Implement output stream of text when !LoadFromFile, and execute project code, checking for output to console. Check render window for visual confirmation.	Textures load into render window, no output to console. Textures move relative to object position(s).	Textures load from file. Textures display in render window. Textures transform with objects.
6	Font style is loaded correctly from file and is applied to set text objects.	Implement output stream of text when !LoadFromFile, and execute project code, checking for output to console. Check render window for visual confirmation.	Font loads into render window, no output to console. Font style applied to text objects.	Font load from file. Font displays in render window. Font styles text objects.
7	Sounds are played ball to edge contact (contact listener).	Strike cue ball with cue sprite when pulled back to apply linear velocity to cue ball, aiming for table edge shapes.	Cushion collide sound outputs per ball to table edge collision.	Cushion collide sound plays when ball(s) collide with table edge(s).
8	Sounds are played ball to ball contact (contact listener).	Strike cue ball with cue sprite when pulled back to apply linear velocity to cue ball, aiming for other ball sprites.	Balls collide sound outputs per ball to ball collision.	Balls collide sound plays when ball(s) collide with other ball(s).
9	Player pots corresponding ball of his type, player has another turn.	Strike cue ball with cue sprite when pulled back to apply linear velocity to cue ball, aiming for balls targeted at table pocket sprites. When collision occurs, ball(s) repositions to UI.	Corresponding player ball type is potted, player turn counter doesn't increment. Same player turn.	Player turn will occur again, meaning that the player turn counter value will not increase.
10	Player names set in console, displayed in allocated namespaces on UI.	Input characters into console window for each player name when prompted upon program execution, press Enter key to proceed (console window displays names).	Player names compare precise to console window input and set positions in UI.	Player names will display correctly. Relative to set positions and output exact console input.

[Whitebox testing (unit testing)] See unit test files in GitHub repository:

CW-xGliff/Unit Test/

Unit testing purposes to test individual components within software, in which, in consideration of my design implementation I was restraint from accessing any functions and variable from being tested within my main class (int main()); I therefore conducted tests for functionality and variables that are test accessible. My initial nine test cases test for the loading of sound files (.wav) into memory, and for the loading of one instance. This was validated using non-equivalating Boolean values, which both return the current value of a Boolean variable within my UI class. My next five test cases test for pointer deallocation for when destructors are called during program termination, similarly these test cases were conducted using non-equivalating Boolean values. Lastly, my remaining three test cases test for view, player name text and game over text instantiation from function calling; and are validated via non-equivalating unsigned integer and Vector2f variables. All nineteen (19) test cases 'passed' (see unit test files in repo) and proves proper code functionality.