James Paton (S1111175)

Games Programming 1 (M2I622939-17-A)

I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

Signed: James Paton

**Intro**This Wiki will take you through every class in the game Boss Battler. We will start from the lowest level and work up to the highest and I will explain everything as we progress. The structure roughly follows the order I created each class.  
Note: If a class begins with c, it was written by my lecturer, Bobby. I will highlight any changes I made to his code.

**Setting up SLD**There are two classes that are used to setup SDL. BossBattlerSDL is our entry point, and cSDLWNDManager is where we initialize the window and the renderer. I made changes inside cSDLWNDManager to include the ability for the game to run in full screen. The was important from an aesthetic perspective. I wanted the game to use pixel art and running this game in a window just means you’re playing a very small game. I was able to adjust the size and position of the SDL\_Rect that the renderer uses to create the viewport. The code takes the 1280x720 resolution of the game and stretches it until one of the axes hits the maximum size of the display, then to maintain aspect ratio, the other axes gets black bars if requires. The information on render scale and render excess is then passed to control so that control can handle mouse input correctly (see control).  
GameConstants.h includes all the includes for SDL, global variables for the window title height and width, and a few enumerated types.

**Managers**cTextureMgr, cSoundMgr, and cFontMgr are three of the four managers in the game, the other being scene manager. As you would expect, each manger is responsible for handling their respective class. The texture manager allows you to load textures from files and use them later to render sprites. The font manager allows you to load fonts, you then use these fonts in conjunction with the texture manager to render text.  
The sound manager loads sounds which can then be played on loop or just once. It does not come with any way of pausing sounds but that was not necessary for this game anyway.

**cTexture, cFont, and cSound**These are the classes used within their respective managers.

**cSprite**cSprite is one of the most used classes in the game. Every object in the game either inherits from cSprite or contains multiple sprites. It allows objects to be assigned a texture, a location, and ultimately, rendered on the screen.

**cGame**The first thing cGame does is call initialize(), this initializes a few SDL functions, control, our managers, including the scene manager, and it loads all our textures, sounds, and fonts.  
Next we call run, the contains our main loop and we will not exit it until the bool looping inside Control is set to false.  
I removed many of the redundant functions from the original cGame.

**Control**The control class was the first class in the game that I wrote from the ground up. It was developed out of necessity, I found the way that SDL handled keyboard and mouse input to be far too ‘clunky’ for a game. Character control is key when trying to make a game feel responsive. There is nothing more frustrating to a gamer than feeling like they lost because of bad controls, it makes the games feel like the game is cheating them.  
The main purpose of control is to maintain a list of all the current key presses. These are stored in std::vector<std::string> keys and edited with two functions void add(std::string key) and void remove(std::string key). These functions add or remove items from the list without duplicates.  
The list can be queried with the down functions bool down(std::string key) and bool up(std::string key) to ask whether a given key is up or down.  
Control also handles mouse clicks the same way it handles keys presses. Mouse movement is kept tracked of and can be queried in a similar way.  
The last thing Control does is maintain a “camera”. SDL does not come with a camera, so I had to make one, the solution is simple: Control keeps track of two variables called xCamera and yCamera, these variables can be modified and queried. When any object in the game is being rendered, it’s position should be offset by the camera’s x and y. The only exceptions are the player (who remains in the centre of the screen) and UI elements.

**Scene Manager**Although the scene manager was an extra feature that we were given the option of including in the game, I made it very early on. My reason being that making the game I had in mind without a scene manager would have been more complicated.  
The scene manager can load a new scene with void load(string newScene) this will update the current scene and initialized anything required for the new scene (buttons, backgrounds, etc). The scene manager’s update function updates all other entities in the current scene. For the menus, this is the buttons and it also checks to see if we have tried to quit by pressing the escape key. In the level, it updates the level, the player, and the boss. It also checks to see if we should transition to the win or lose scenes based on the boss’ or player’s health.  
While playing the level, the escape key no longer quits the game but rather saves and returns the player to the main menu. Now, the Start button has been replaced by a continue button and pressing it will load the savegame.dat file that was created, allowing the gamer to continue where they left off. When the game saves or loads it displays a notification to the gamer so they know it has happened.

**Button**Buttons are mostly used to interact with the scene manger, allow the gamer to transition between scenes. A few buttons have additional functionality, such as the continue button from the lose scene reviving the player, and the back button from the win screen resetting all the entities so that they game can be played again from the beginning.  
The buttons will also play an animation if the mouse hovers over them.

**Level**The level is the heart of the game. It maintains a 2D array of tiles which make up the game world. It updates their position based on the camera, which in turn, is based on the player’s movement. It also handles the collision detection between the players and the walls, as well as the bullets and everything they can collide with.

**Tile**Tiles inherited from cSprite and add some additional functionality. They have four boolean attributes: collidable, solid, slow, and animated. These tags allow other classes to interact with the tiles appropriately. The tiles also have an ID which determines which type of tile they will be: wall, grass, water, etc.  
Some of the tiles are also animated, this brings some life into and otherwise still environment.

**Player**The player’s most basic functionality is movement. This is achieved by checking the keyboard input and the adjusting the camera accordingly, our player never actually moves, the world moves beneath him. The code for determining the player’s direction and speed was originally quite simple, but with the extension we were given I made it far more complicated than what would have been passable for this course. By this point in the project I had become attached to my little character and I wanted him to have a particular “feel”. If there is a contradiction between moving up and down, or left and right, the last key pressed will dominate. As the player has only four sprites with no diagonals, the sprite is updated based on which direction was pressed last. And finally, the speed at which he moves when travelling diagonally needs to be adjusted: If an object travels along a grid at one unit per second either up or down, then in order to have the same velocity while traveling diagonally, the object must travel at 0.707 units in each axis.  
His speed can also be adjusted: If he is on water, he’ll move very slowly making the ponds in the game quite hazardous. If you press left shift, we will walk, moving more slowly but his rate of fire will increase allowing the gamer to take advantage of being in a good position to hit the boss.  
His health bar is updated based on his health.  
The goal of the gamer is to control the player and make him shoot at the boss. When the gamer clicks the mouse, a projectile is spawned at the player’s location and travels towards the location of the mouse.

**Boss**The boss moves in a predictable pattern, has a large health bar like the player’s, and shoots projectiles. The boss has two attack patterns, his basic attack which simply shoots towards the player, and his AOE attack which shoots 8 projectiles in a circle around himself. The AOE attack will happen 20% of the time, as well as any time the player is touching the boss, this is to encourage the player to stay back.

**Projectile**Both the player and the boss can shoot projectiles, these share the same class. The only difference is they have different textures. Their differing behaviour (whether the hurt the player or the boss) is handled inside the collision detection of Level.

**Tools**I created a tools class to provide quick and easy access to functions I used throughout the game. It has three basic tools:  
Round – Returns a rounded int for the given float or double.  
Random – Return an int, float, or double between the two given values (inclusive).  
CalcUnitVec – Returns the unit vector between two given points.

**Extension**We were given an extension of four days to submit our coursework. I was already finished a couple of days before the original deadline, so I used this additional time to fix a few things I wasn’t quite happy with. Firstly, the character movement: I have already updated the page titled Player to reflect the changes made.  
I fixed a bug where the boss’ Y position was not being saved properly.  
I added a delta time to support variable refresh rates.  
Lastly, and most importantly, the performance issues: I had always wanted my game to be a bullet hell games like Touhou or Jamestown, but when I tried to increase the number of projectiles the boss fired, the framerate of the game dropped dramatically. After a discussion with one of my lecturers, Warren, we discovered my code for detecting collision between the projectiles and the solid tiles was very inefficient. I rewrote all the collision detection functions found inside level and, happily, I now had plenty of overhead to have more projectiles on the screen. This meant I could turn my casual shooter game into a true bullet hell game! I gave the boss two attack patters: a triple shoot aimed at the player, and a far more aggressive spiral AOE attack.