**Data Structures**

This program was first adapted from the CarDemo workshop files written by Jon Macey. One of the first changes was replacing the game loop with a single function in the main application loop for clarity, and moving it to GameLoop.cpp/GameLoop.h, along with the SDLErrorExit() and createOpenGLContext() functions originally written by Jon Macey. The first original classes that were added to the program were the Button and Menu classes, the former of which proved very useful later on in the Game class when creating texture buffers from SDL TTF text surfaces for the in-game UI.

It's safe to say my original class diagram from the initial research report hardly represents this final structure. The original was made with very little knowledge of how the drawing of the 3D context would actually work and what classes I would actually be building and should focus on designing. While some of the data structures were created out of immediate necessity while building the program without much foresight (mainly the Menu classes), some of the classes were still based off of the original report design (Button, HighScore, and somewhat Projectiles). Others, namely those important ones used by the Game class (Ship, Projectiles, GameEnv) as well as the Game class itself were designed after the structure of John Macey's SDL JoyPad Demo, with each class planned to carry out a similar role to the Game, Rocket, and Starmap classes of that project.

The idea of using a base class for the projectile types was something that was planned but not fully implemented until much after simple projectiles of a class Projectile were implemented. Simply adding another attribute to the Projectile class to store the type which would control a bunch of conditional statements in its member functions seemed tempting at first, but being able to successfully use class inheritance and polymorphism for the draw and update functions was worth it in the end and a good exercise.

**General Application (Menus)**

**Game Loop**

The actual OpenGL context took a while to get comfortable with but felt fairly intuitive after the workshops and especially after the Principles of Rendering project. The foundation for the game loop first provided in the CarDemo was kept fairly intact with the SDLErrorExit() and createOpenGLContext() functions. More things were taken out of the loop than added in.

A notable exception is the method for implementing smoother movement control of the player Ship. This was done by means of keeping variables that stored both keyboard up and down information for each separate direction and then adding all the information together for a final movement vector. This allowed for responsive changes in direction when multiple keys were pressed at the same time as well as diagonal movement, as opposed to formerly having a single translational case statement for each arrow key down-press that would cause sudden halts in movement where multiple keys were pressed. This method was also based off of the SDL JoyPad Demo.

**Collision Detection**

There were two different implementations of collision detection in this project. The case of the collisions between the ship and the projectiles which used bounding spheres as suggested in the initial feedback, and the case of the ship colliding with the edges of the player movement area, using a simple bounding box. I noticed in my original report that though I had an algorithm for AABB collision detection, essentially the same algorithm used for the player movement bounds, I did not include an algorithm for bounding spheres, an extremely simple method of collision, which proved more than adequate for the purpose of colliding with projectiles and any inaccuracy is masked by the speed of the movement. This algorithm is shown below.

//distance between two points, two dimensions is appropriate as we are moving on a plane

**Distance = square\_root((x1-x2)2+(y1-y2)2)**

**If Distance < bounding\_sphere\_radius1+bounding\_sphere\_radius2**

**Collision = true;**

**Shaders**

To shade the objects in the scene, I used an instance of the ngl::ShaderLib class to manage different shader programs, used to render objects to different texture buffers. These texture buffers (m\_\*\*\*TexId) are created and linked to a framebuffer (m\_fboId) in the Game::initFBO() function, which is called once. The code in this function is adapted from the Principles of Rendering exercises by Richard Southern. Using texture buffers allowed me to perform texture lookups to create effects like blur and chromatic aberration to give the game a retro look, as well as allowed me to blend layers of textures in the fragment shader.

**Updating the score display**

Unfortunately I was not able to figure out how to get ngl::Text to work with my program, where the program would produce a segmentation fault where the class would call Qfont. As a work-around and a way to use my own font used in the menus with the previously created Button class, I decided to store the pixel data of the SDL\_Surface that is created with SDL\_TTF to another texture buffer. This seemed all right at first, especially considering that now that text was on a texture buffer (something I would pass to the shader as a sampler2D) I could use sampling methods to apply the same blur and chromatic aberration effects from the game texture to the text as well. However, I was not considering the fact that the initFBO() function, where I create all the texture buffers and link them to the framebuffer, is only called once at the beginning of the game, meaning that if I was to render my score as text on the texture buffer it would just be zero, even if the actual score variable increments as the game plays. I did test out calling initFBO() on draw() and this did increment the score display but also to no surprise tanked the frame rate considerably before an inevitable crash.

To get around this issue I decided to render digits 0-9 on the texture buffer instead and implement a lookup algorithm in the fragment shader that would be driven by a uniform containing an array of the digits of the current score, which could be resent to the GPU on every draw(). This ended up being an unnecessarily long process but in the end it did work. The algorithm used to perform the lookup for every digit of the score was originally designed as described below:

However, after trial and error, this ended up being the following algorithm below, due to the frustrating fact that the origin in SDL\_Surfaces is located in the top-left corner as opposed to the bottom-left in OpenGL textures, meaning that normal texture2D() lookups end up upside down. This is something was fairly easy to keep track of and fix in the previous SDL\_Surface texture buffer lookups, but with this algorithm I wasn't able to keep up.

The algorithm used for the recursive function made to create an array of digits from an integer (see below) was very nicely explained by Broam on StackOverflow.com (2009).

Endgame Screen

References

<https://stackoverflow.com/questions/5289447/using-sdl-ttf-with-opengl>

<https://stackoverflow.com/questions/1860983/convert-integer-to-array>