When I had started my project developing a game, the first thing I knew I wanted was simplicity. A top-down shooter game seemed perfect since there's an abundance of online resources, tutorials, and code examples available. This would make my project manageable and enjoyable, especially when I was stuck—I often referred to tutorials by Low Level Game Dev on YouTube and numerous GitHub repositories featuring raylib-based projects. These sources provided important knowledge and ready-to-use examples which made the early stages straightforward.

My game's development begins with a simple menu screen. Implementing the main menu was inspired directly by the structured tutorials from Low Level Game Dev, who specializes in raylib game tutorials on YouTube. Their video series on "Creating a Game Menu in raylib" significantly simplified my task.

Early on, I tackled core C++ concepts essential to game development. I started by defining basic structures like 'Entity' for representing game objects (players, enemies, projectiles). Structures are an important concept in C++ allowing me to group related data like position and health and status into neat and reusable units.

Then I introduced another essential concept in C++ programming called Pointers. A pointer is essentially an address reference to data stored elsewhere in memory. They're useful for managing my game objects like spawning new enemies and projectiles during gameplay.

With basic entities and pointers in place, I moved towards classes and inheritance to improve my game's modularity and scalability. I set up a hierarchy beginning with the 'Entity' struct, from which 'Player' and 'Enemy' structures inherited. Inheritance in C++ allowed me to define general behaviors in the base 'Entity' class, then specify unique behaviors in derived classes. For instance, the 'Boss' class inherited from 'Enemy', overriding behaviors to create distinct gameplay challenges.

To handle the dynamic aspect of my game—random enemy movements, projectile spawning, and varying gameplay—I incorporated advanced C++ features. I utilized random number generation (RNG) through the modern and reliable std::mt19937 engine to allow me to introduce unpredictability in enemy behaviors and spawning locations in order to enhance the replayability of my game.

Another advanced feature I integrated was smart pointers, specifically std::unique_ptr. This significantly simplified memory management by automatically freeing memory for objects no longer in use, reducing the chance of memory leaks

The primary data structure used throughout was std::vector. It dynamically managed my game's entities like projectiles and enemies, efficiently resizing itself as objects were added or removed during gameplay. Custom structs like 'Room' helped in organizing the game's level design and organizing enemies and defining the game's progression logic.

Coding these initial features had its challenges like managing collisions and interactions. My first attempts at collision detection lacked precision, prompting me to revisit tutorials by raylib enthusiasts on GitHub and YouTube. After finishing these essential elements I was ready for more advanced features.

I went on developing more advanced features with added complexity which present new challenges that were also learning opportunities. Initially, managing memory with raw pointers resulted in memory leaks and segmentation faults—a common hurdle for Computer Science students. Learning about smart pointers, particularly std::unique_ptr from resources like The Cherno on YouTube, significantly improved my memory management.

Implementing enemy AI was another challenge since Random enemy movements lacked excitement, leading me to introduce an aggro system. This addition enhanced gameplay but increased complexity in debugging and collision detection. Tutorials by Coding Made Easy and community resources on GitHub greatly aided this process.

Structured testing became more important as the code's complexity grew and Early playtests revealed obscure bugs which warranted me to integrate unit tests for collision detection, enemy behaviors, and room transitions. Testing highlighted areas needing refactoring which led to cleaner and more maintainable code.

Adding a boss enemy introduced a little bit of difficulty mainly in implementing specialized behaviors through inheritance and function overriding. Debugging these mechanics was initially challenging, but community forums and resources from developers like Ramon Santamaria proved invaluable.

Gameplay balance was another tricky aspect. Early tests showed the game to be either too easy or too difficult. Adjusting parameters like enemy health and shooting cooldowns through iterative testing and feedback from peers emphasized the importance of playtesting.

Optimizing data structures, such as std::vector and custom structs, became necessary as the game expanded. Profiling code to improve performance became an essential skill which enhanced both gameplay smoothness and my analytical abilities.