

Rich Spencer
CSCI-120-70
Project 2: AI-Powered Project Genie
July 29, 2024

ChatGPT Prompt

I am a student in a college level introduction to Computer Science course currently studying C++ programming. I am looking for creative and effective ways to deepen my understanding of specific C++ topics. My interests include games, Dungeons and Dragons, and heavy metal music, and my preferred learning style is multimodal with a combination of kinesthetic learning with visual learning. Please suggest some tailored programming project ideas that incorporate the following C++ topics: function and class templates, lists, hash tables, object-oriented programming, file I/O, and algorithms.

ChatGPT Results

Here are some tailored project ideas that incorporate your interests and the specified C++ topics:

1. Dungeons and Dragons Character Manager

Create a comprehensive program to manage characters for a Dungeons and Dragons campaign. This project can include:

- **Function and Class Templates:** Use templates to create flexible functions and classes for different types of characters and items.
- **Lists and Hash Tables:** Store and retrieve character attributes, inventory items, and spells using lists or hash tables.
- **Object-Oriented Programming:** Design classes for characters, items, spells, and quests with appropriate inheritance and polymorphism.
- **File I/O:** Save and load character data, campaign progress, and inventory to and from files.
- **Algorithms:** Implement algorithms for sorting characters by different attributes, finding the shortest path in a quest, or calculating combat outcomes.

2. Heavy Metal Music Playlist Organizer

Develop a music playlist organizer tailored for heavy metal tracks. Features could include:

- **Function and Class Templates:** Create templates for handling different types of media files and playlists.
- **Maps and Hash Tables:** Organize and quickly retrieve songs based on metadata such as artist, album, genre, and release year.
- **Object-Oriented Programming:** Design classes for songs, playlists, and user profiles with proper inheritance and relationships.
- **File I/O:** Save and load playlists, user preferences, and playback history.
- **Algorithms:** Implement algorithms for shuffling songs, recommending similar tracks, and optimizing playback order.

3. Console-Based Adventure Game

Build an engaging console-based adventure game where players navigate through different locations, collect items, and encounter NPCs. Include:

- **Function and Class Templates:** Use templates for different game entities like players, enemies, and items.
- **Maps and Hash Tables:** Manage the game world, including locations and items, using maps or hash tables for fast access.
- **Object-Oriented Programming:** Create classes for game elements with inheritance for different types of entities.
- **File I/O:** Save and load game states, player progress, and inventory.
- **Algorithms:** Implement algorithms for decision-making, pathfinding, and random event generation.

4. Fantasy Creature Database

Design a database for cataloging fantasy creatures, complete with information about their abilities, habitats, and lore. Incorporate:

- **Function and Class Templates:** Use templates to handle different types of creatures and attributes.
- **Maps and Hash Tables:** Efficiently store and retrieve creature information using maps or hash tables.
- **Object-Oriented Programming:** Develop classes for creatures, habitats, and abilities with hierarchical relationships.
- **File I/O:** Save and load creature data, user annotations, and database states.
- **Algorithms:** Implement search and filtering algorithms to find creatures based on various criteria.

5. Battle Simulator

Create a battle simulator where different characters or creatures fight each other, with stats and abilities affecting the outcome. Features can include:

- **Function and Class Templates:** Templates for various combatants and their abilities.
- **Maps and Hash Tables:** Store combatant stats, abilities, and battle logs for quick access.
- **Object-Oriented Programming:** Design classes for combatants, battles, and abilities with proper relationships.
- **File I/O:** Save and load battle scenarios, combatant profiles, and battle logs.

- **Algorithms:** Implement combat algorithms for turn-based battles, damage calculations, and strategy planning.

Tips for Multimodal Learning:

- **Visual Learning:** Use diagrams and flowcharts to plan your projects. Visualize data structures, class hierarchies, and algorithm flow.
- **Kinesthetic Learning:** Implement the projects by writing code, testing, and debugging. Engage with hands-on coding exercises and build tangible parts of the projects step by step.

These projects should help deepen your understanding of C++ while also incorporating your interests and preferred learning style.

Chosen Project

With the help of my friendly neighborhood ChatGPT I had a lot of great ideas to choose from for this project. But what caught my attention was the **console-based adventure game** (surprise, surprise, Rich wants to make a game).

The console-based adventure game suggested by the GPT is a great way for me to tackle something I've been wanting to do for a while, which is make a game engine. So, this project will be to build a basic game engine that can create Windows-console based games in the style of an old-school console adventure game. I'll be using Nethack (one of my favorite games) as inspiration.

Implementation

The application's source code has been divided into two distinct groups: Core and Game. Core classes are more like engine classes where I've implemented basic functionality that the game objects and classes can derive from.

I use branches to direct the flow of code execution all over the codebase. For example, in the Application's Run and TickInternal functions I use if checks to only perform operations if the given conditions are met. In Run(), I use an if statement to check if the AcculatedTime is greater than the TargetDeltaTime to see if it is time to render the current frame. Likewise, in TickInternal, we only want to tick if the Word is valid, so I put the World's Tick behind an if statement.

Strings are used heavily in this application as it's primary data format. User Interface and Map files are all saved as strings and parsed through the FileHandler to generate game objects from the strings. Strings are also used as the "sprite" for Actor objects to render in the World/Level.

The main game loop in Application::Run is a good example of how loops are implemented in this project. The game loop will execute while bAppRunning is true. Another example of a loop in the application is in the Delegate class where we loop over all of the function Callbacks when a delegate is broadcasted to its subscribers. In this case, the for-loop loops over the List of functions and executes the Callback.

Templated functions are a key aspect of the architecture of the application. I think the best example of templated functions is in the World class with the `SpawnActor<T>` and `SpawnHUD<T>` functions. These are core methods that create instances of the referenced class in the game World and holds a reference to the new object in a List of objects in the World.

File I/O is handled by the FileHandler class which loads and converts the text data files into game objects. For example, the FileHandle class reads a .map file from disk and simply makes one long string out of it before passing the new string off to the StringToMap function that fills in the Map object. The same is true for loading and populating UI elements from disk. HUD layouts are stored in .layout files that generate the overall HUD layout.

Journal

Daily work done on the project.

7/16/2024

Created project files and git repository.

7/17/2024

Created requirements document with minimum requirements to make a functional game. Began R&D on how to approach the user interface. There's a few low-level libraries that can do the arrow key movement, but I want to see if I can do it with native C++ Standard Library modules first. It seems [Windows.h](#) has some functions that control the terminal window directly, so need to explore that. Got some initial ideas down for the Map class.

7/18/2024

Did a ton of research on ways to work with the Windows console window without using a library. I think I'm going to have to use the Windows Console API to work with the console directly. Pretty much all the libraries I found were just wrappers for the console api anyway.

In the console api, [WriteConsole\(\)](#) writes a character string to a console screen buffer beginning at the current cursor location. I can specify a location in the console with [SetConsoleCursorPosition\(\)](#), which takes in a HANDLE to the console screen buffer, and COORD struct that specifies the new cursor position, in characters. The coordinates are the column and row coordinates of a cell in the console screen buffer.

[ReadConsole\(\)](#) gets keyboard input, passing in a handle to the console's input buffer. It returns keyboard events that can be translated into ANSI or Unicode characters.

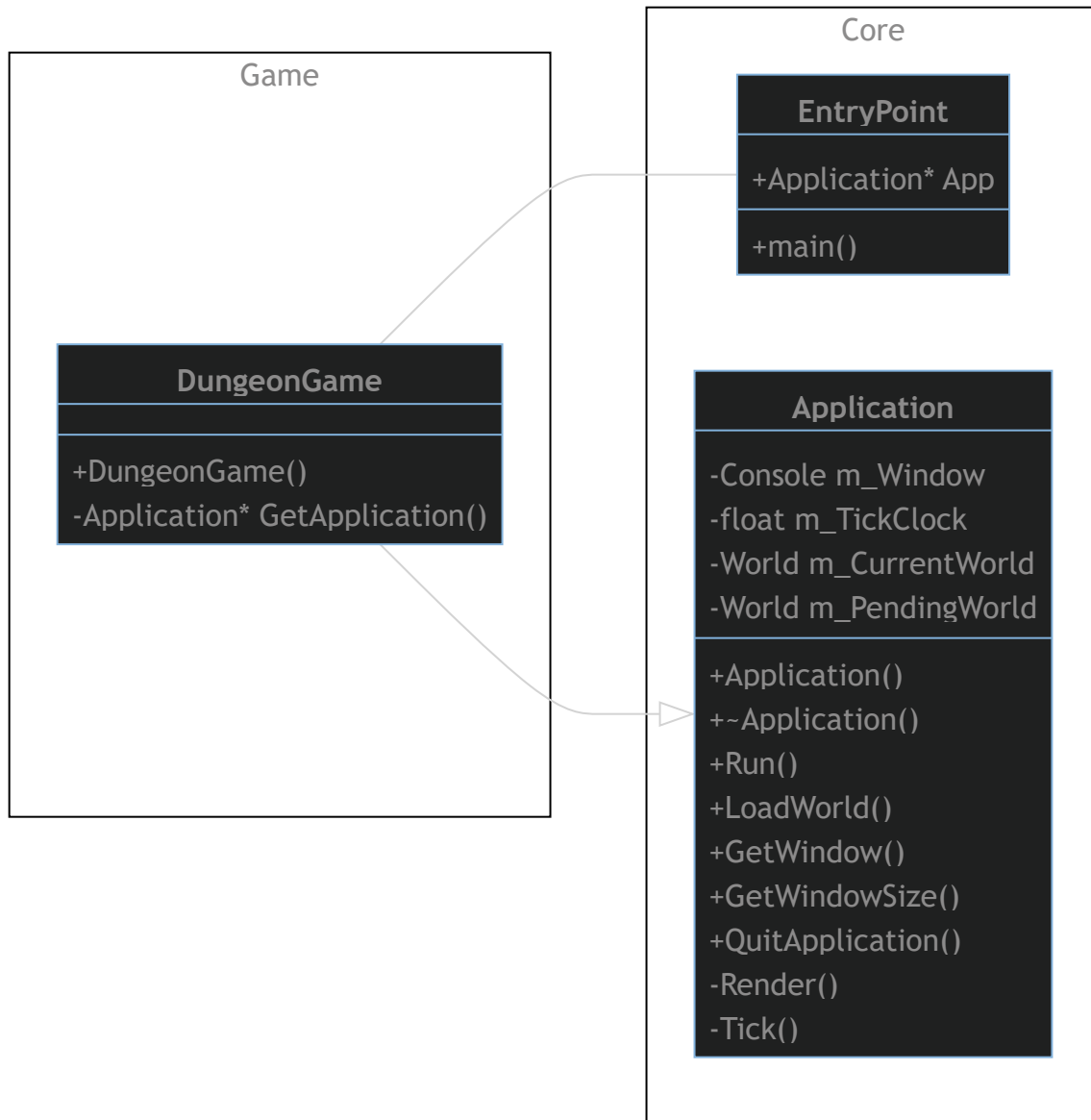
This is manageable, I think. It's very similar to how OpenGL handles vertex buffers, so approach it from that angle. Next step is to block it out; get a screen loaded and try and draw the title screen. If I can do that tomorrow, I should be fine to proceed.

Ok, for the application design itself...

I want to take a Game Engine approach and separate the Core logic from the Game logic. So building the *engine* as a static library and link it into the *game*:

7/19/2024

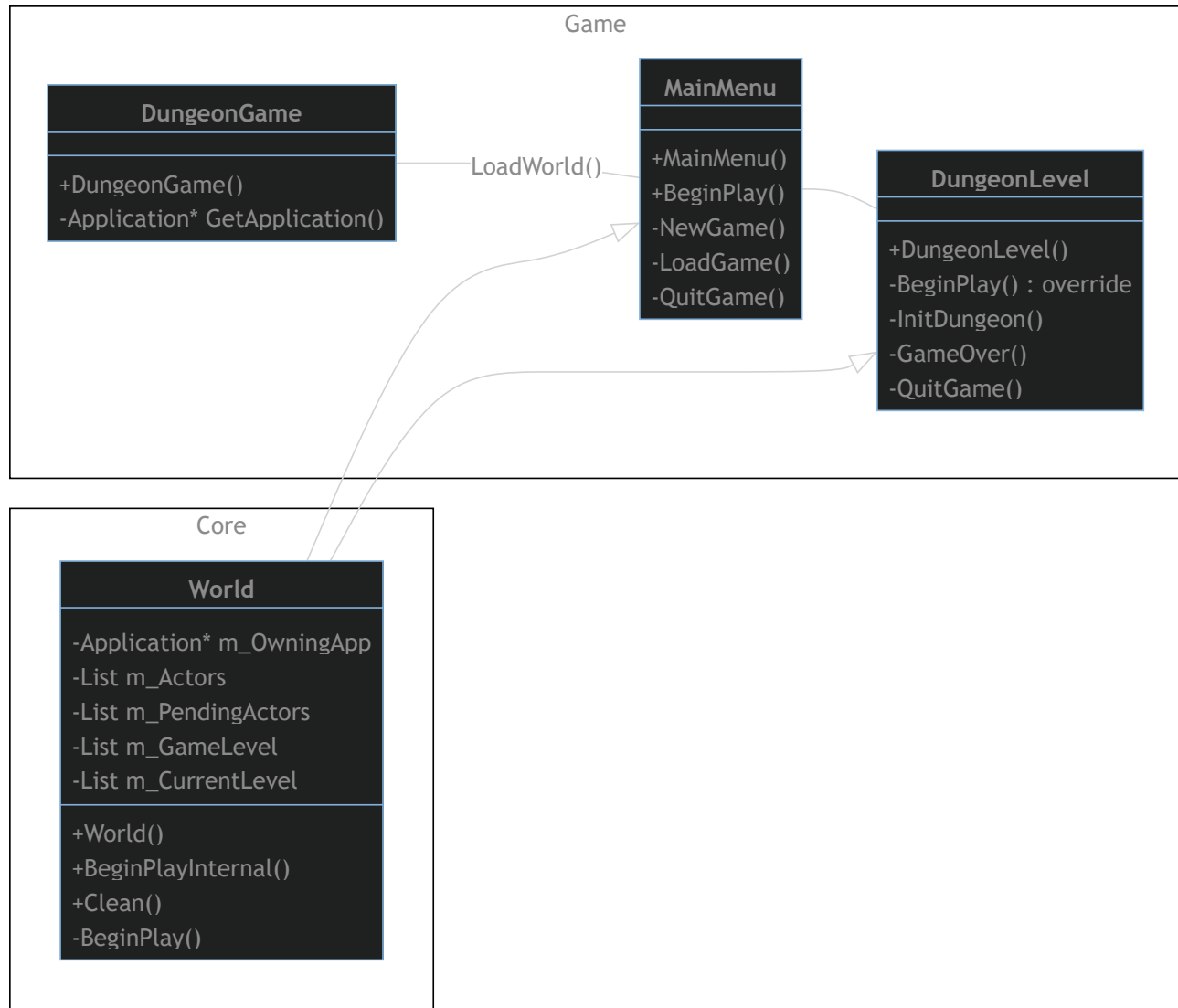
Today I worked mostly on design and setting up the application/game framework. I'm not going to do any static library linking, instead just going to separate the core logic into a folder to keep it separate from gameplay.



EntryPoint::main() in Core calls **GetApplication()** that is defined in DungeonGame.cpp (will probably rename that file soon), which creates a Game object on the heap and returns a pointer to EntryPoint which immediately calls **Run()** on the Game.

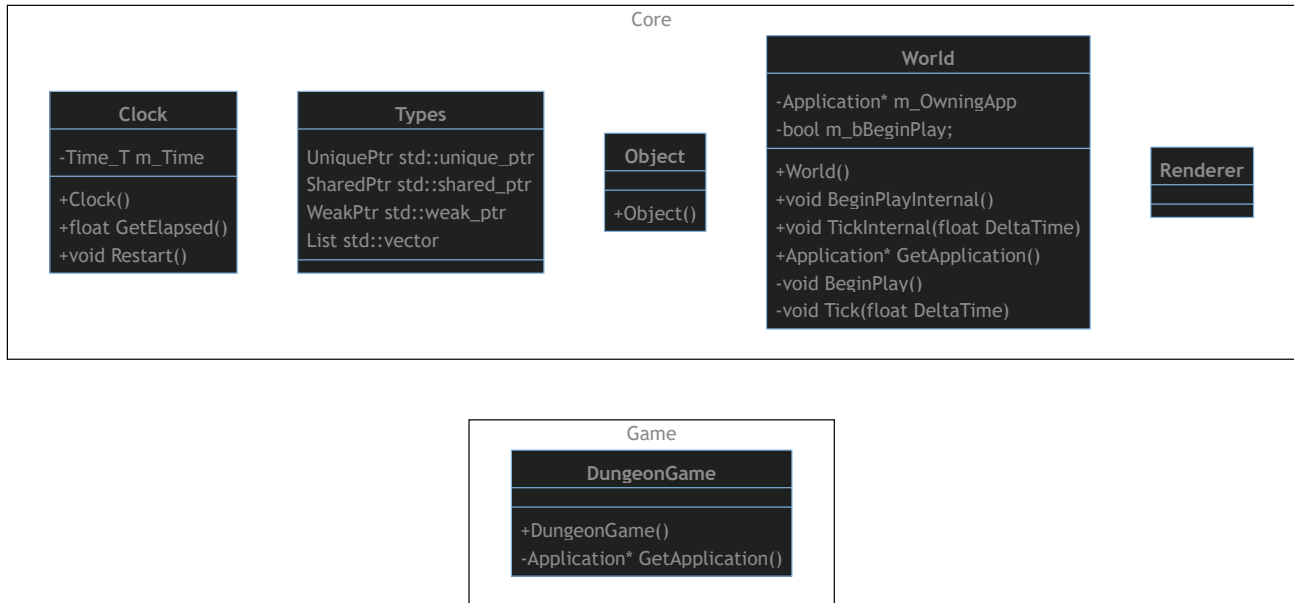
7/20/2024

Got a basic drawing working on the console. I can move the cursor to a specific coordinate in the console window and draw text into the console buffer. Need to make sure I only update if something has changed, and if possible, only updated the cells that have changed. Implemented basic control to only render if we need to, i.e. `RenderIsDirty` equals true.



7/21/2024

Created a Clock class to handle render timings and as a base class for any other timers that might be needed in-game. Created a Types.h files that defines datatypes. Created base Object class that all game objects will inherit from. Currently, it's only assigning a unique id to the object. Created World class that manages the loading, unloading, time management for all objects in the world. Created a Renderer class the handle visual rendering to the console.



7/24/2024

Working some problems

Render Pipeline

Need to figure out the render pipeline and how to get the Actor's display properties up to the Renderer to be displayed in the console.

Step back a second... the way we draw in the console is to first define the render space size as a 2D grid, then move the cursor to a location on the grid with a coordinate (x, y), use `std::cout` to print a character to the console. That's really all that's involved in writing characters to the console.

Since I've already set up Tick with a target draw rate of 60 fps, I can cache all the draw calls from the game objects in the world into a 2D display buffer that fills up as much as it can in 1/60 of a second, then draws it all in one go. The display buffer would be a representation of the console display and each Actor passes their Render parameters off for the Renderer to handle.

The Renderer should define the window size and use that to view only the part of the World that is within the bounds of the Render view. So if the World is 200x100 tiles and the Render view is 80x30 we only see that section. This might be a nice-to-have as I can see moving the Render view could be complex. Do it if you have time.

So the Actor holds the character that represents it.

- . : empty tile
- # : corridor
- @ : player
- \$: pile of gold

- ! : potion
- ^ : trap
- a-z, A-Z : monsters

The Actor should track if its state has changed this frame and needs to re-render. Actors should have a `MarkRenderDirty()` function to call at the end of anything that changes the Actor's display state.

So the pipeline would be:

- (Application)
 - End of Run loop call `RenderInternal()`
 - `RenderInternal` clears the screen
 - Calls `Render()`
 - `Render` calls `Render()` on the current World
- (World)
 - `World::Render` loops over the Actors in the world
 - Calls `Render()` on each Actor
 - World renders the HUD/UI
 - .
- (Actor)
 - `Actor::Render` returns if `IsPendingDestroy` or `!IsRenderable`
 - Actor calls `Draw` on the `Renderer`, adding itself with a payload (location, character, color) to the `Render buffer`
- (Application)
 - Calls `Display` on the `Renderer` which writes the buffer to the console
- (Renderer)
 - iterates through the 2D buffer and either draws each character in sequence, or find a way to draw the buffer at once

Input

I don't think `std::cin` is going to work. Looks like Windows console has a library in the Standard C++ Library called `conio.h` that includes methods to fetch input from the console. I think this is the way to poll user input every frame.

7/25/2024

Did a ton today. Created a base class for UI Widgets and derived a `TextWidget` class from `Widget` base class. Created a virtual `HUD` base class to manage UI functionality and derived `MainMenuHUD`, implemented a basic main menu. And by basic I mean non-functional, just for looks at the moment. Input will be finished tomorrow. Updated the `MainMenuLevel` to spawn the `MainMenuHUD` to draw the main menu. Added a `Constants` header file to hold game-specific constants, specifically for the screen width and height.

Created a `Vector2i` helper class for 2D integer Vectors to help with positioning elements on the screen. Also added a `Transform` class to hold all the transform data for game objects. Currently, it only holds

position, but I plan on extending it to scale to account for larger objects and UI elements.

A big step forward was finishing the base implementation of the `Renderer` class which holds a render buffer in memory until the frame is complete at which point it draws the buffer to the console screen.

Added an `Actor` class, derived from `Object`, to represent any game object that can be displayed on the screen. Updated the `World` class to hold a pointer to all Actors in the World and the ability to spawn actors into the World.

Updated the `Application` and `DungeonGame` classes to implement the `Renderer`.

Working some problems

Still left to do...

Movement

Input is polled every frame and if any of the movement keys are pressed (W, A, S, D or arrow keys), the players token moves one tile in that direction if it is allowed to. To keep things simple, all Game Objects are blocking objects, meaning you can't move into a tile that's occupied.

Interaction

If an object is interactable. Open doors. Pick up items. Within a radius if 1 tile, interaction HUD pops up.

Player

Classes: Warrior, mage, rouge. Inventory:

Map

Map is loaded from file. World reads the map and spawns in required game objects.

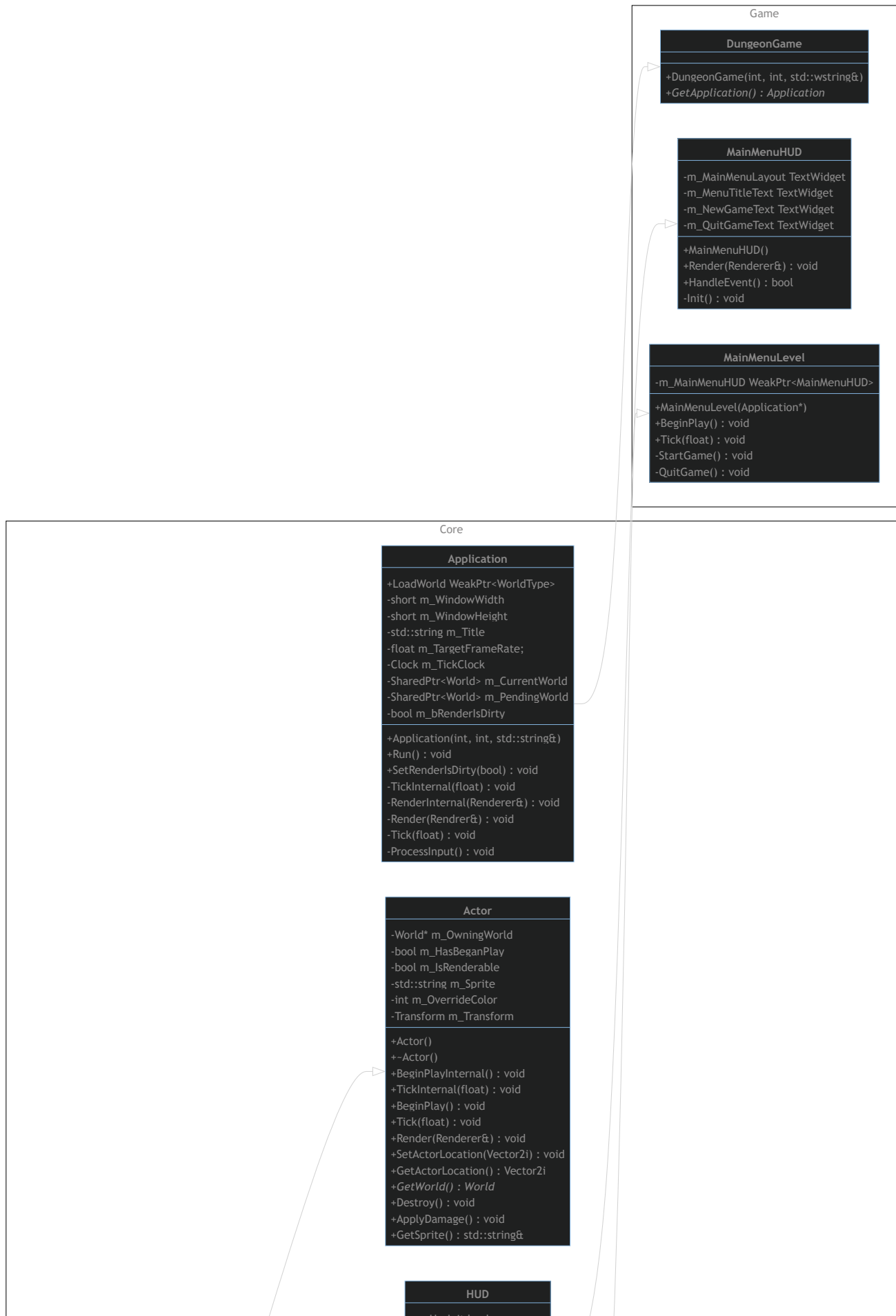
7/26/2024

Going to include an updated, full class diagram each update. Easier for me to keep track of.

Added a `MainMenu.layout` file to draw the menu background at once rather than one cell at a time. Added a `FileHandler` class to load files from disk. Specifically addressed loaded `.layout` files for UI layouts.

Updated the `Widget` and `Render` classes to handle multiline text by adding a boolean parameter to the function. With that, `MainMenuHUD` loads the `MainMenu.layout` file and writes it to the console.

Class Diagram





7/27/2024

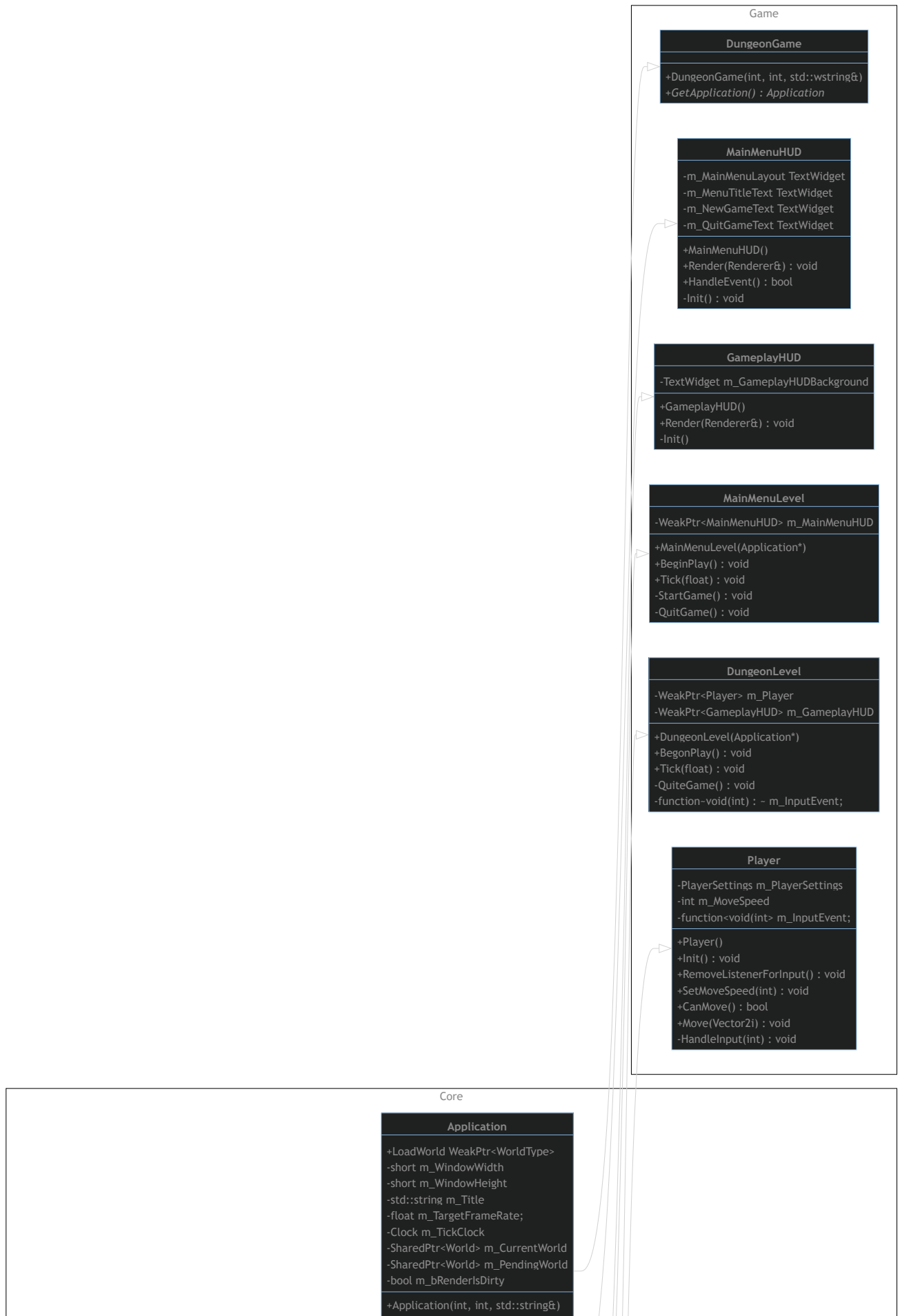
Completed the Input class and callback functionality and hooked it up to the main menu. HUD classes now receive input notifications and can switch between levels. Currently, main menu can quit the game and start a new game by loading the DungeonLevel.

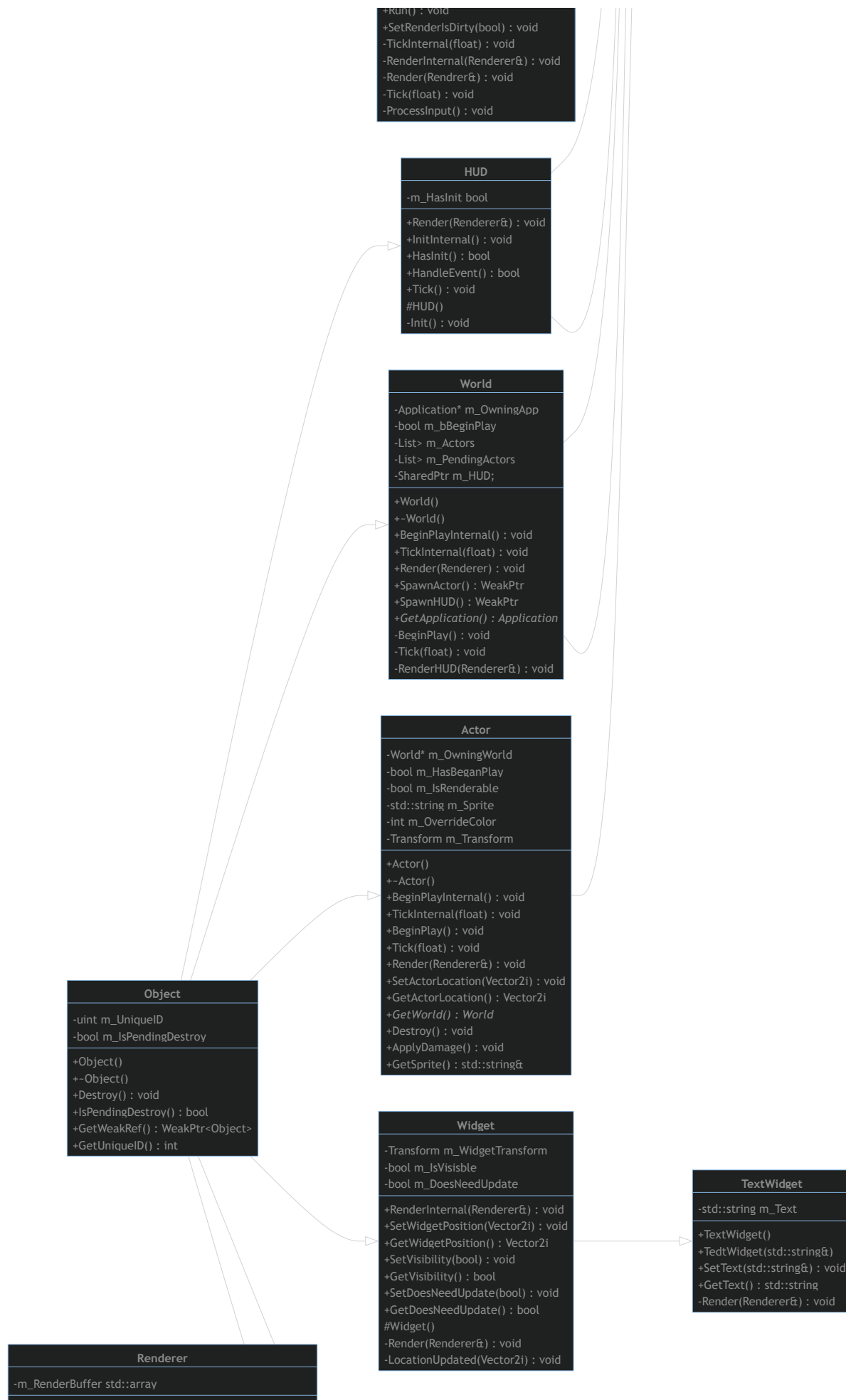
Updated the Renderer to draw Actors to the console screen. The Renderer can also take in a color override to change the color of the sprite. The DungeonLevel now draws the HUD to console from the DungeonHUD layout file.

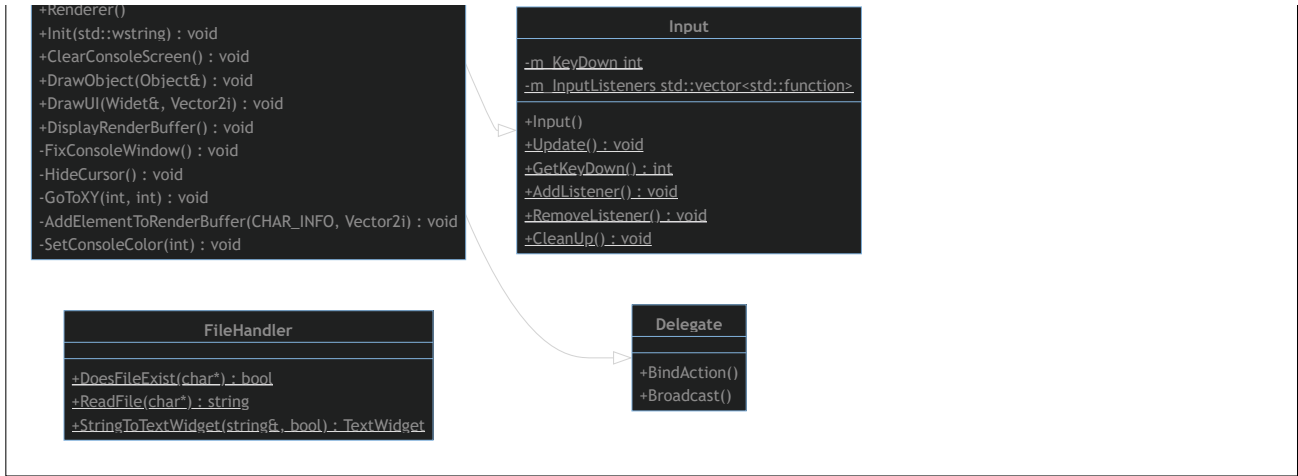
Also implemented the Player class with player movement, created a Player Settings struct to hold the player's settings, and the Dungeon level spawns the player after it loads from the main menu.

Lastly, created a Delegate class to handle event notification and subscription. This will primarily be used to drive the user interface, but I want to use this in the Input system as well. Currently, the Input system is a static library and doesn't have an instance in the world, so I'll need to refactor the Input class if I want to use the Delegate system. That'll be a nice to have, if I have time.

Class Diagram





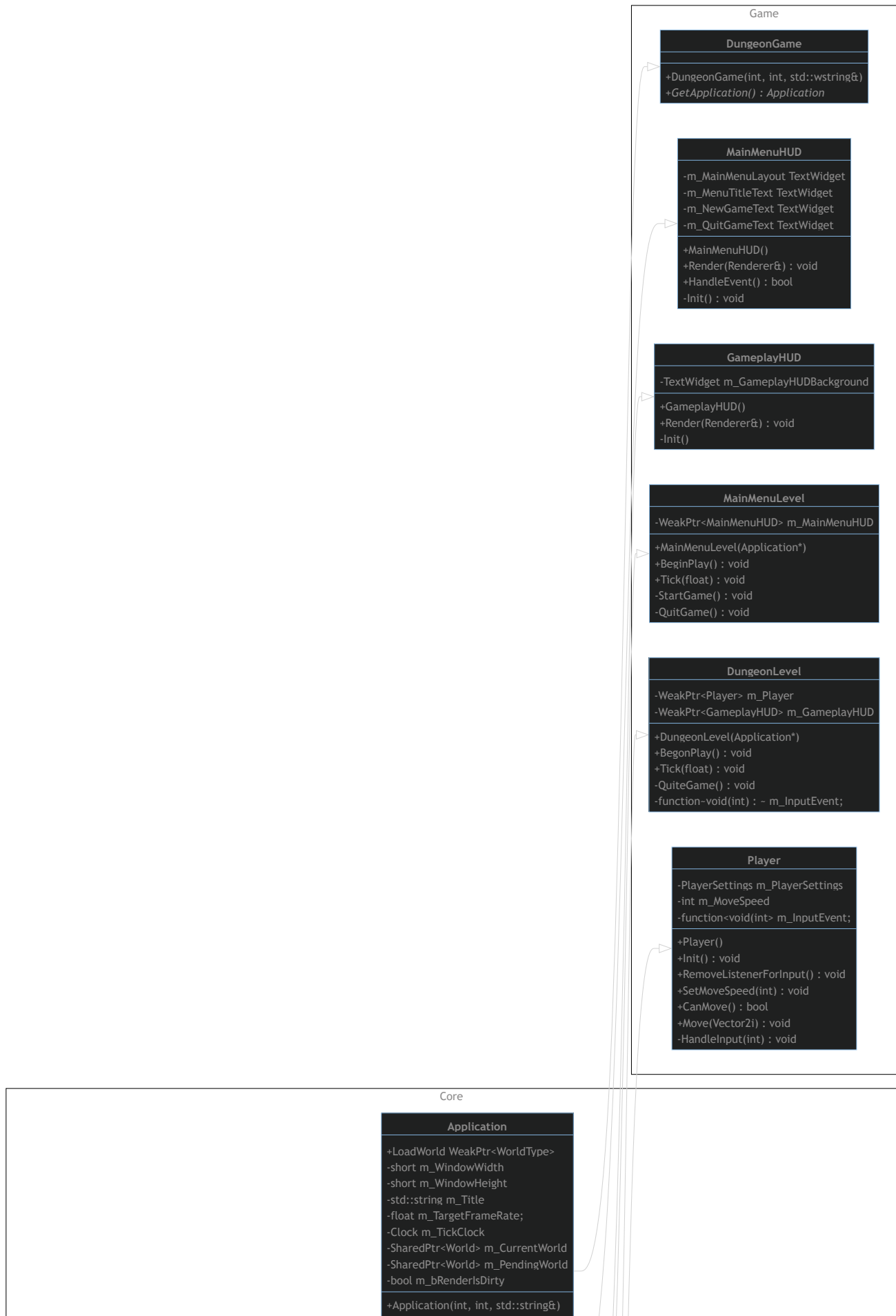


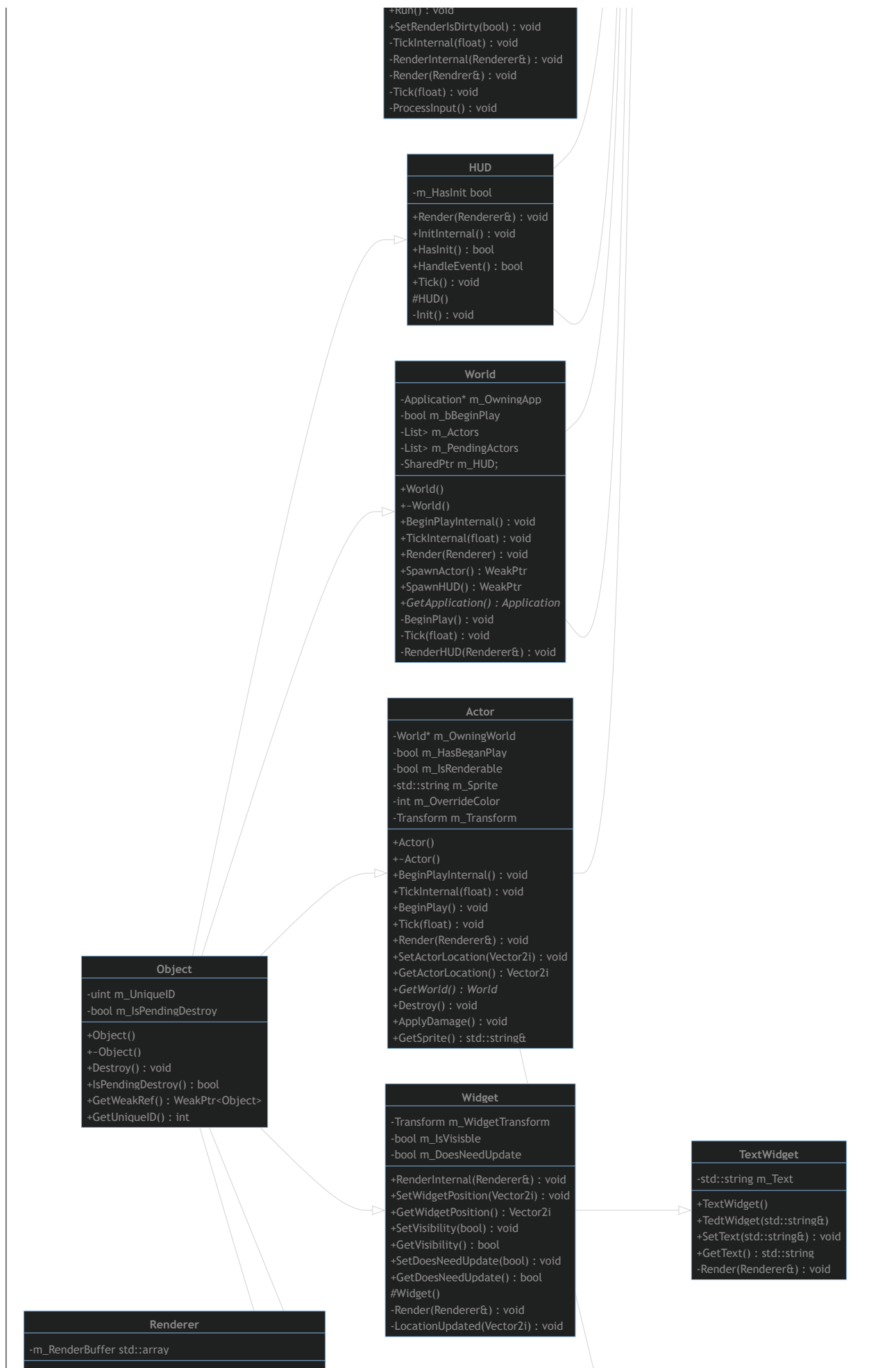
7/28/2024

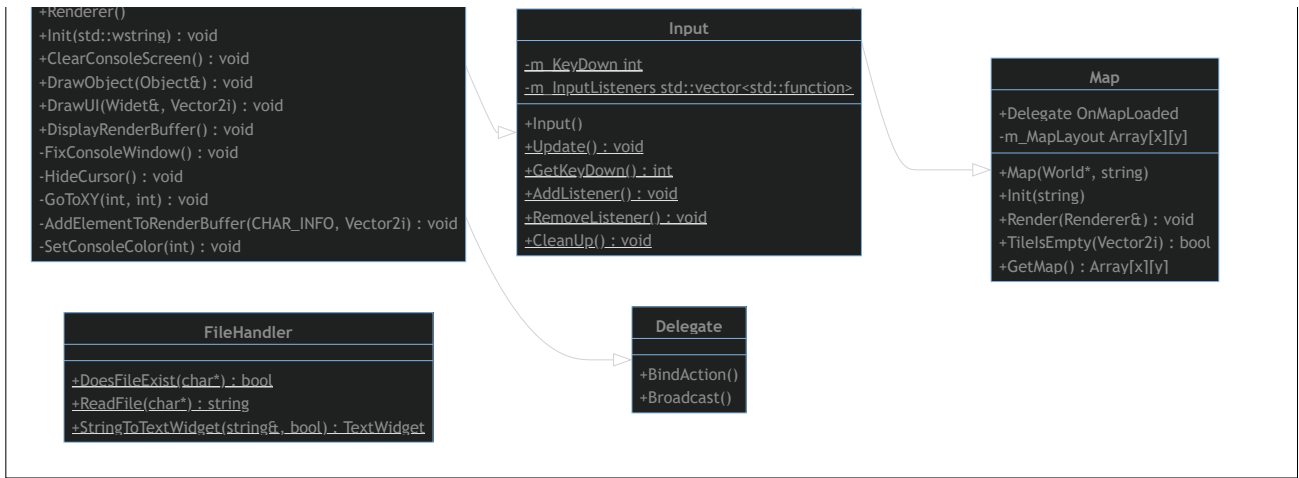
Completed the `Map` class (derived from `Actor`) and supporting updates to `FileHandler` to load map files from disk and display in the console. Also implemented movement blocking by occupied map tiles and sets the player's starting position in the level. Ideally this would be loaded from the map, but for now I'm just hardcoding the player start in the level.

Made updates to the `HUD` classes and layout files to use the full buffer size when writing to the console.

Class Diagram







7/29/2024

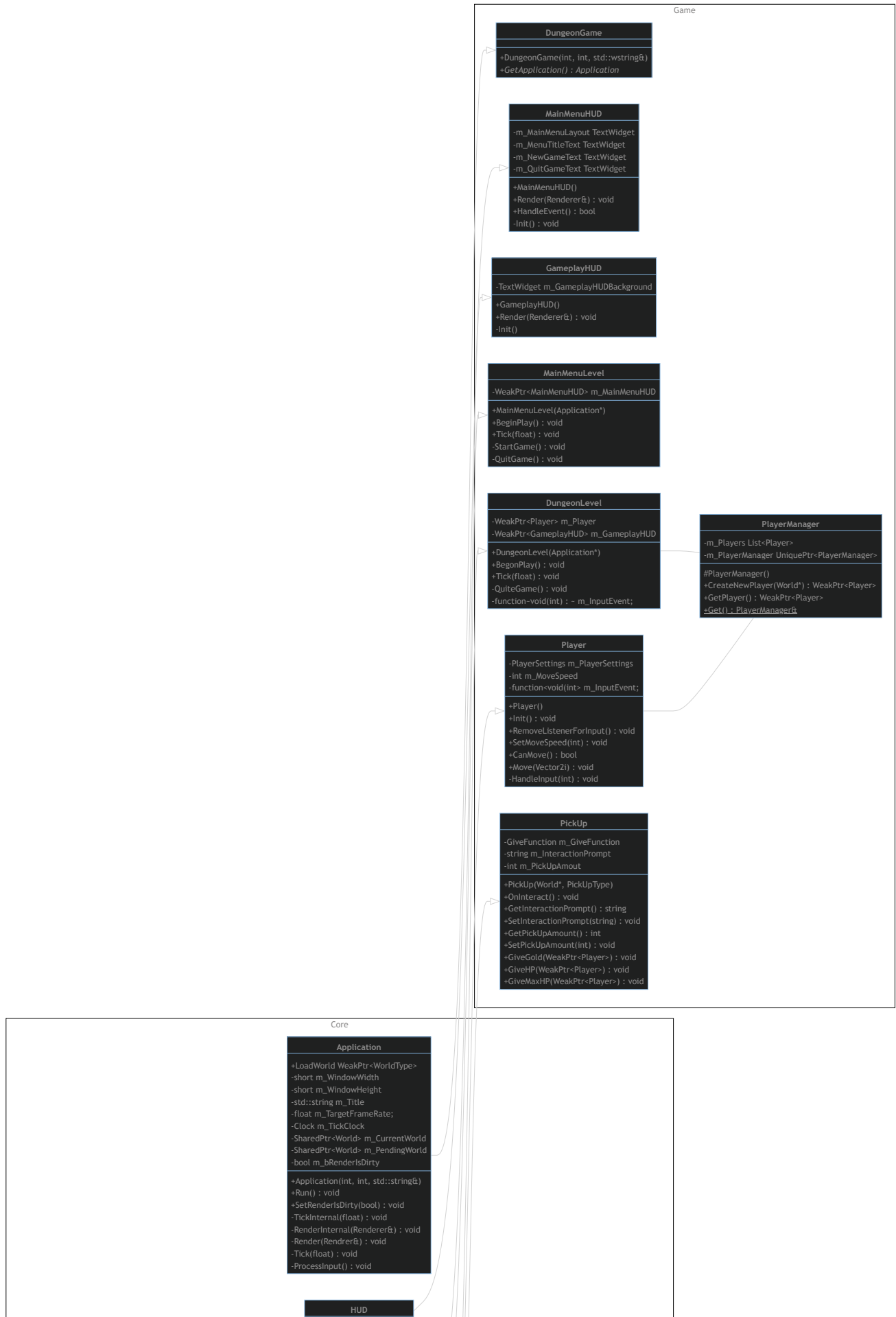
Added a `PlayerManager` class to handle spawning in and holding a reference to the `Player`. `DungeonLevel` uses the `PlayerManager` to spawn the actor into the level.

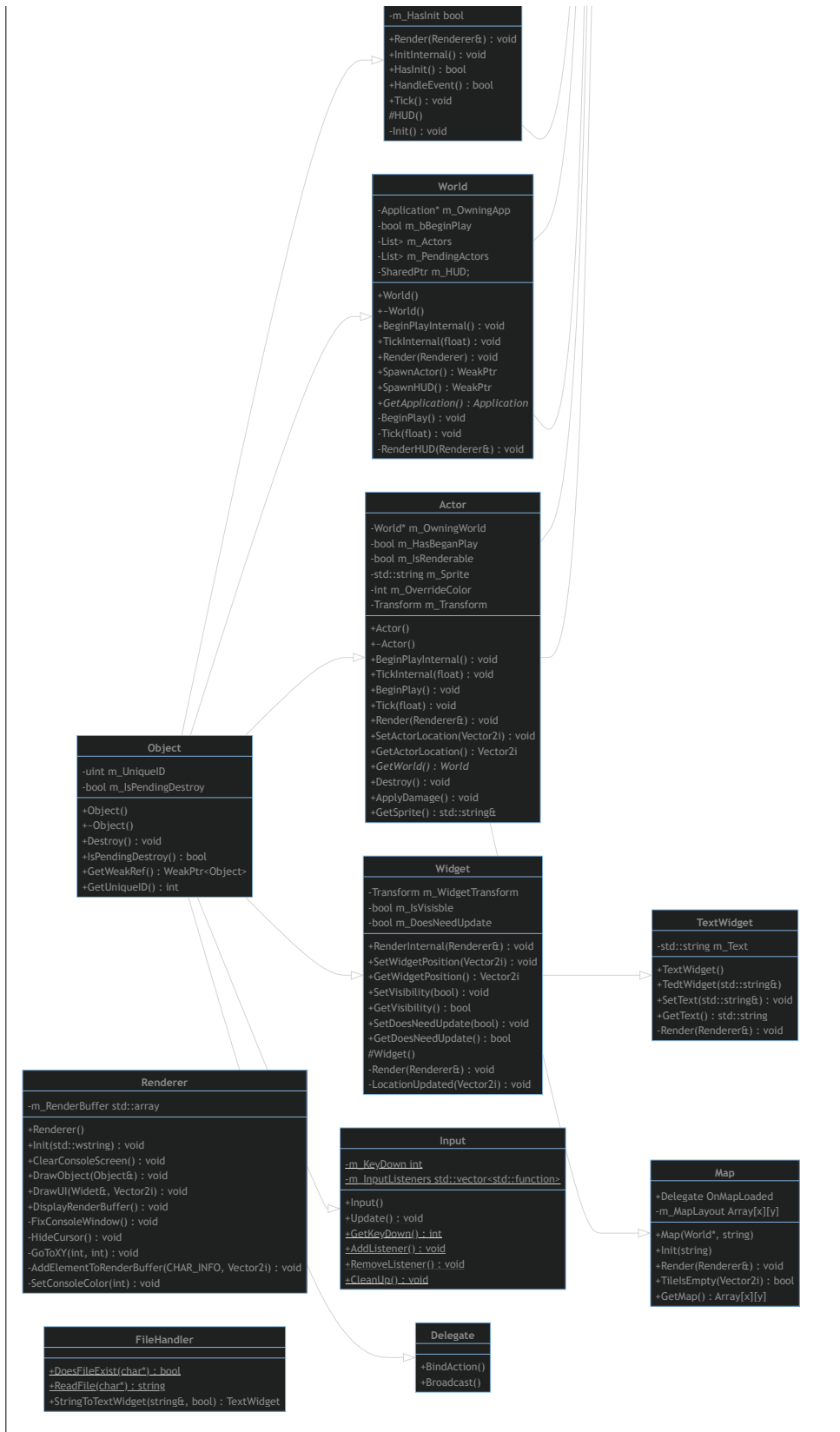
Also set up `Player` attributes and related `Delegate` broadcasts. `GameplayHUD` now binds to these `Player` delegates to update the UI when appropriate.

Added a `PickUp` class to define pick up styles with an enum `PickUpTypes` defined on the `PickUp` class. Updated the `World` to spawn in `PickUp` Actors and add them to the `Map`. I ran out of time and wasn't able to implement the interactions.


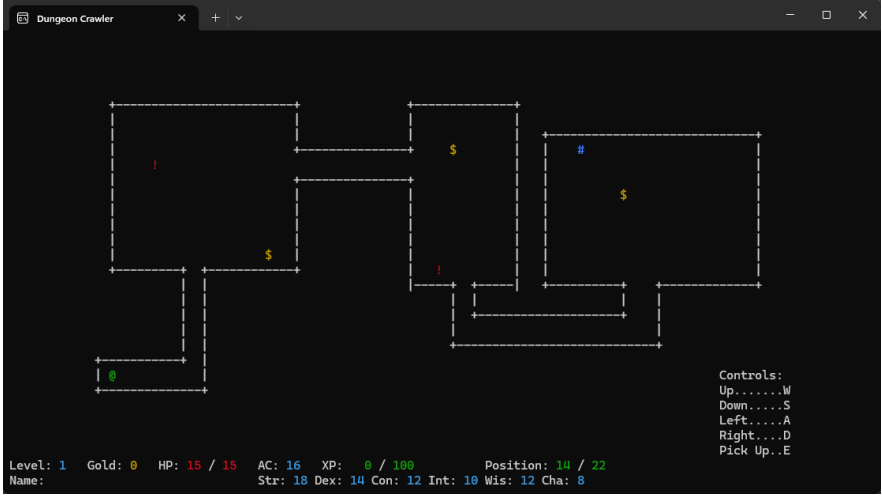
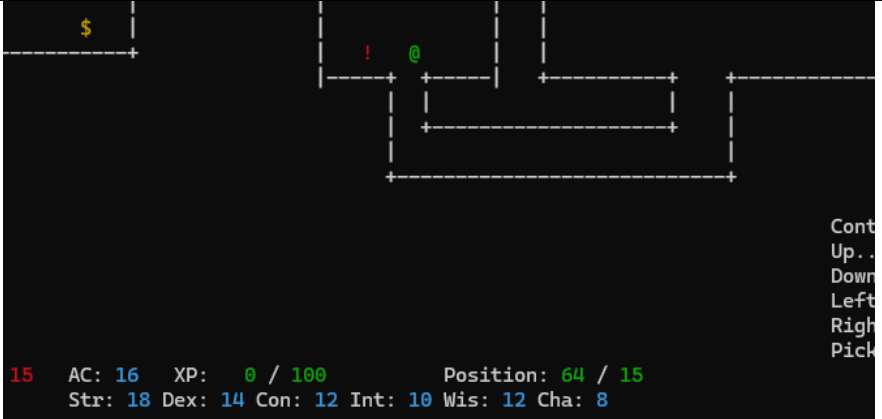
Time to wrap up the report and turn this beast in. I think I'm going to keep working on this as I had WAY too much fun working on this and scratched the itch I had to make a game engine. Ultimately, I didn't finish the game, but I got a super cool engine started that can be extended to make games like [Nethack](#), the inspiration for this project.

Class Diagram





Screenshots

<p>On launch, the main menu is displayed.</p> <p>Press N for New Game ...Or... Q to Quit</p>	
<p>The dungeon loads and places the player at the start of the dungeon, bottle left.</p> <p>The World spawns in pick-ups like health potions (!) and piles of gold (\$).</p> <p>Player movement is restricted by walls.</p>	
<p>The UI updates the player's position through a delegate broadcast to the Game HUD.</p> <p>All UI is driven by delegate broadcasts from the Player class to the Game HUD class.</p>	

Reflection

Well, I definitely bit off a lot for this project. I mean, who thought it was a good idea to try and make a console game engine in 2 weeks?! Oh, right, it was my idea.

I think the biggest thing I learned from this project is how important it is to try and keep functionality contained within a class. For example, when I was building the UI / Widget system for this project, it was really easy to fall into the habit of putting UI functionality all over the place, like in the Player class, because it was easier to get the data I needed. So, I had to repeatedly take a step back and think about what the class's responsibilities are and think in terms of how to provide the needed data.

It was that methodology that led me down the road of researching and making the Delegate class and functionality. Being able to simply "broadcast data" when needed and have unrelated classes subscribe to those notifications was a pretty cool problem to tackle.

The other HUGE challenge on this project was manipulating the Windows console. I think I spent too much time trying to figure that out, if I'm honest, but in the end, I think the application looks pretty close to what I was going for in terms of visual design. But again, there was an "ah-ha" moment when it clicked for me. Over the last few weeks, I've been learning OpenGL and some of that experience really helped me on this project by utilizing a render buffer, very similar to OpenGL's vertex buffers, to draw to the console in a batch, rather than drawing to each cell individually.