Overall Approach

As discussed in Phase 1's documentation, our main approach when creating this game was to develop from backend to frontend—solidifying the code infrastructure and the game engine mechanisms before focusing on features or visuals. The key questions to ask during the game design was not how a feature would be implemented but how the feature would be integrated into the game engine mechanisms in a manageable and trackable way.

To achieve code manageability and trackability, the core game infrastructure and mechanisms need to be well organized. We started with a simple goal of first implementing the core gameplay elements, with a heavy emphasis on designing around localized call structures, hierarchies of locality, and minimum repetitions. Only after the core game infrastructure was solidified did we add the extra game features, unique flavors, and assets. As predicted in Phase 1, we ended up designing around four main levels of game locality. From highest to lowest, they are as follows:

1. **Game Engine**: The highest level of game awareness, having reference to all other objects in the game. The Game Engine's main responsibility is to initialize the Game Manager and to execute the various logics behind rendering and incrementing frames;
2. **Game Manager**: The second highest level of game awareness, having reference to all GameObjects and their respective managers. The Game Manager's main responsibility is to initialize the three GameObject Managers, to track game states, and to run audio assets;
3. **GameObject Managers**: The third highest level of game awareness, having reference only to its respective class of GameObject (i.e. player and enemies). The GameObject Manager's main responsibility is to initialize its respective class of GameObjects, and to handle inter-GameObject logics, interactions, and communications; and
4. **GameObjects**: The lowest level of game awareness, having access only to its respective GameObject Manager. The GameObject holds various self statistics, animations, and movement logic.

With these four levels acting as the primary infrastructure, we were able to keep our code relatively clean and maintainable even when new features were added. Ideally, objects of a locality level can only call methods from the same locality level or from those that are one level lower. We followed this rule as tightly as possible to avoid potential "spaghetti codes". Additionally, while designing the integration of new features, we focused on minimizing the "locality spread" of said feature by minimizing the need for adding extra inter-level communications.

Modifications

The initial plan for most inter-class communications was to create a singleton object, called Map, for other objects to read and write from. This object would act as a communication medium for the other classes; however, this proved to be a big bottleneck and a synchronization hazard. We examined the information required of each class and decided on the following modifications:

1. Removal of GameCore: instead of GameCore rendering all GameObjects, the GameObject Manager classes are now responsible for the rendering and updating of logic for their respective GameObjects. This change also reinforces our locality philosophy and removes the need for communication between GameCore to the various managers; and
2. Inter-manager communication: we found that most, if not all, communication requests can be reduced to purely on a GameObject Manager level of locality. Hence, we decided to create static methods in each of the GameObject Managers for others to retrieve data from instead of through an extra medium.

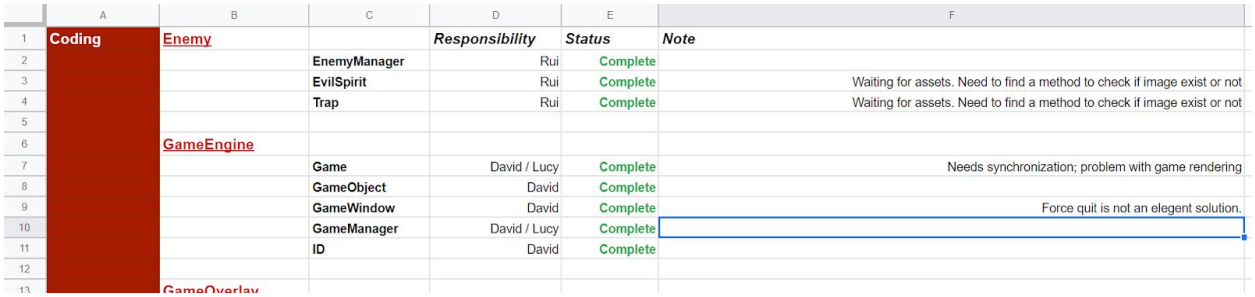
These modifications removed the need for our map object to act as both read and write communication for all objects. Nonetheless, in our final implementation, the map class remains to be a singleton class and is primarily used as a read-only tool for our GameObjects to retrieve positional information and environmental collision logics.

It was never intended for there to be a PlayerManager class since we initially believed that the logic behind the player character would be relatively simple. However, we quickly realized that the player management section of GameManager would become too big as it provides information for the other two GameObject Managers. So we decided to create a PlayerManager class which contains all player object related methods and data. The PlayerManager class now contains the main collision logic for the enemies and rewards, and various static getters for the other managers to retrieve from.

Originally, the enemies would utilize A* search to reach the player character. This was ultimately scrapped due to three main reasons: the game would be too difficult; it's too computationally intense; and it would be difficult to code correctly (especially when trying to lower the difficulty). Running A* Search 60 times per second on a 1080x680p map would be too taxing on the computer and having them always find the most optimal path to the player would render the game to be far too difficult. We believe that trying to remedy these problems through more code is not an efficient usage of our time (on top of the already difficult A* algorithm). Hence, we decided to operate the AI based on purely distance heuristics, but give them the skill to temporarily walk through walls—which proved to be ideal in terms of difficulty for the player.

Management and Quality Control

For discussions and updates, we chose Discord as our primary communication platform because it was easily accessible and had voice chat as well as screen share functions. Additionally, we utilized Google Spreadsheets to keep track of the development progress and responsibilities of each game component, as seen below:



	A	B	C	D	E	F
1	**Coding**	**Enemy**		***Responsibility***	***Status***	***Note***
2			**EnemyManager**	Rui	**Complete**	
3			**EvilSpirit**	Rui	**Complete**	Waiting for assets. Need to find a method to check if image exist or not
4			**Trap**	Rui	**Complete**	Waiting for assets. Need to find a method to check if image exist or not
5						
6		**GameEngine**				
7			**Game**	David / Lucy	**Complete**	Needs synchronization; problem with game rendering
8			**GameObject**	David	**Complete**	
9			**GameWindow**	David	**Complete**	Force quit is not an elegent solution.
10			**GameManager**	David / Lucy	**Complete**	
11			**ID**	David	**Complete**	
12						
13		**GameOverlay**				

Each major component is labeled on the leftmost column (i.e. “Enemy”) with the various classes that make up said component labeled on the right. Responsibilities are typically split by units of components due to the respective classes usually sharing strong coupling. Members put their names beside the classes they are working on to indicate their responsibilities, ensuring no work gets overlapped or forgotten. The status and notes columns of a class are constantly updated by the respective individual in order to help track the game’s overall development progress.

Due to the limited amount of time and the scope of the project, member roles are split into two types: backend developers and feature developers. Backend developers have a full understanding of the game’s infrastructure, mechanisms, and development methodologies; they focus their work across three areas:

1. Developing and optimizing the game engine, the Game Manager, and the high-level GameObject Manager logics;
2. Creating infrastructure for feature components to operate in; and
3. General quality controlling.

Feature developers focus on the development of various highly independent modules that provide additional enhancement to the game. These include features such as the game visual overlays, the audio systems, the GameObject entities, the art assets, and the sound assets.

External Libraries

No external libraries were used.

Challenges

The biggest challenge we faced was the varied levels of software development experience each member had. Some members were well acquainted with the programming language, whereas others have not had as much practice. Learning from each other became increasingly important as new techniques and mechanics were introduced; overall, frequent communication was needed to keep everyone on the same page and overcome this project.

The other challenge we faced was regarding the backend; lacking the foresight in what we needed, the development process was largely a mix of trial and error. Communication structure revolving around the Game Engine, Game Manager, and the GameObject Managers were constantly overhauled for various improvements and optimizations. Methods and access structure were constantly swapped around or modified to accommodate the “newly optimized” backend. And because of major changes being made regularly, frequent meetings were needed to keep the other members up to date. However, as the backend became more stable and optimized, incorporating new features and extensions became increasingly easier to integrate.