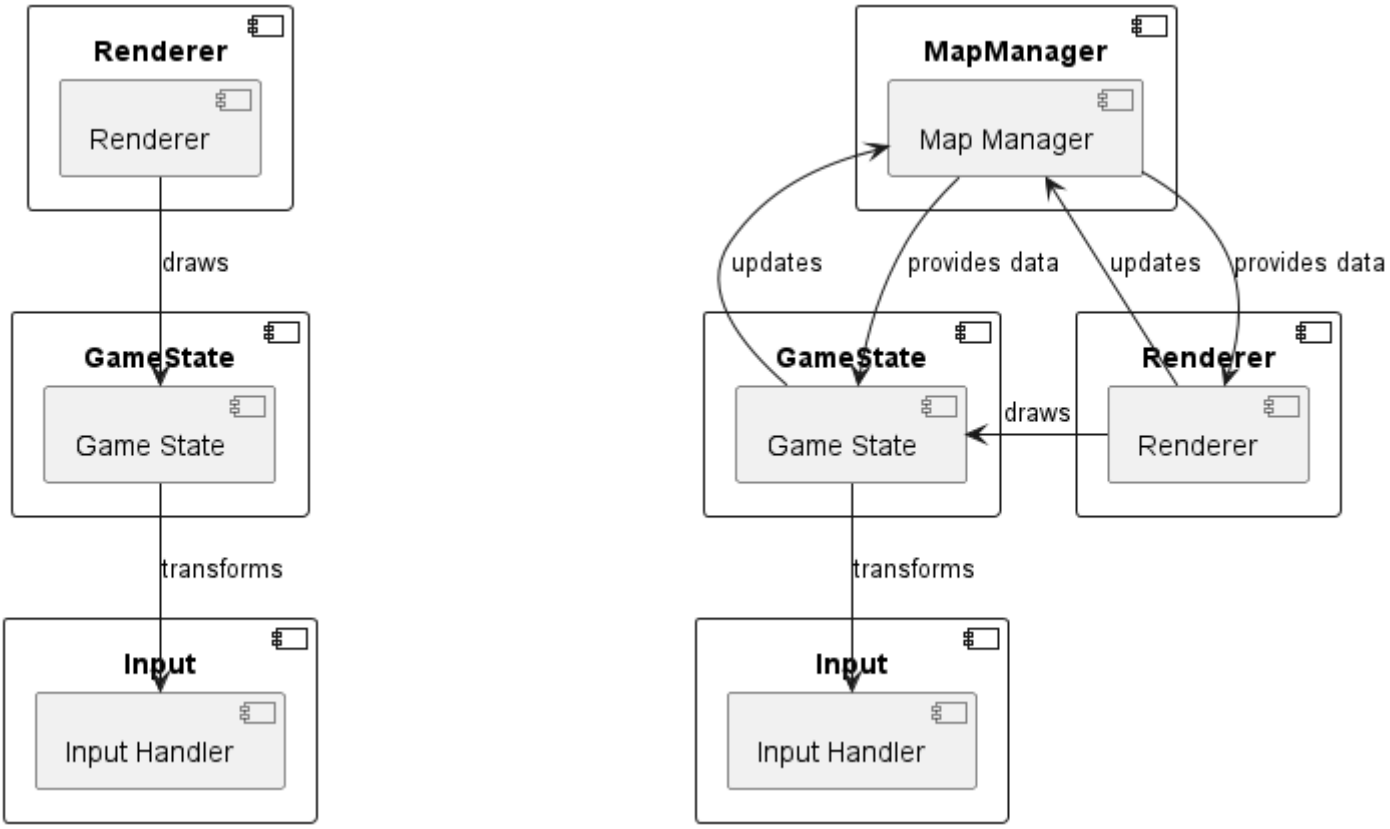
Architecture
Group 19, “Piazza Pitstop Crew”
By: Noah Forryan, Lewis Morgan, Naufal Tun Thamanian, Tom Owen, Dan Manby, José Fernandes



UML component diagrams depicting a high-level overview of the system's architecture.
(Created using PlantUML [1] and PlantUML Integration [2]).
Left: Original concept. Right: Revised concept.

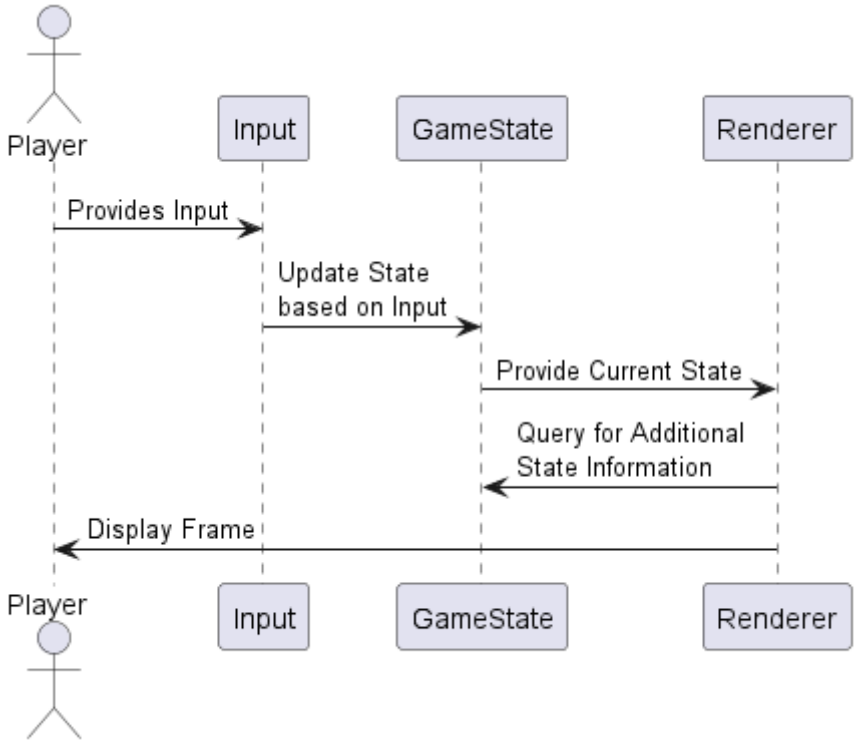
This architecture is almost universal in games, due to their real-time nature. We made some assumptions, including the target frame rate of 60, the processing of a single input per frame, and the sequential execution of each part of the component diagram.
As the project evolved, we decided to add a new component to facilitate communication about maps between the Renderer and the GameState.

The Input component transforms raw user input into 'actions' that the GameState understands.

The GameState component contains all of the data concerning the logical state, or simulation, of the game:

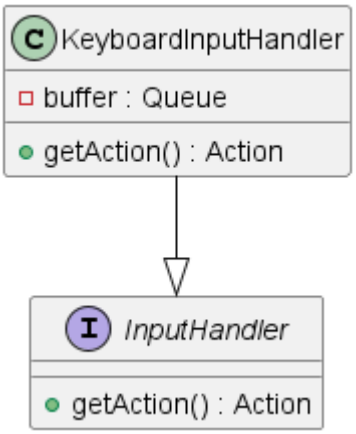
- Player position
- Map
- Interactable entities
- Energy
- Time
- Score

The Renderer component queries the GameState in order to draw it to the screen. In this conceptualisation, the state is entirely independent of the renderer; no change to the renderer will have an effect on the game state. In some sense, this is a simplified MVC architecture wherein the GameState is the data model, the Renderer the view, and the Input the controller.



UML sequence diagram depicting the game loop. (Created using PlantUML [1] and PlantUML Integration [2]).

Input



UML class diagram showing an overview of input handling.
(Created using PlantUML [1] and PlantUML Integration [2]).

We chose to use a simple InputHandler interface and implement it with a concrete class that handled keyboard input and used a small buffer to keep track of the player's recent actions.
The advantage of this approach is the flexibility of being able to easily implement the interface for any kind of input, using any underlying technology. Our concrete class extended LibGDX's input class, but the rest of the system doesn't care about any implementation details.

Input is responsible for functional requirements: FR_MENU_NAVIGATION_CONTROLS, FR_MENU_SELECT_CONTROLS, FR_START, FR_MOVEMENT_CONTROLS, FR_INTERACT_CONTROLS.

GameState

Initially, we considered inheritance-based entity approaches that are common in games; the state maintains a collection of all entities in the game, and each frame calls a virtual update() method on each entity. Creating new game objects entails extending or implementing an Entity class.
This was our first choice due to simplicity and flexibility, but over the course of the project this changed; due to the nature of the game, there was no requirement for distinct entities to update every frame. We chose to do away with the entity abstraction completely and keep the codebase as simple as possible.

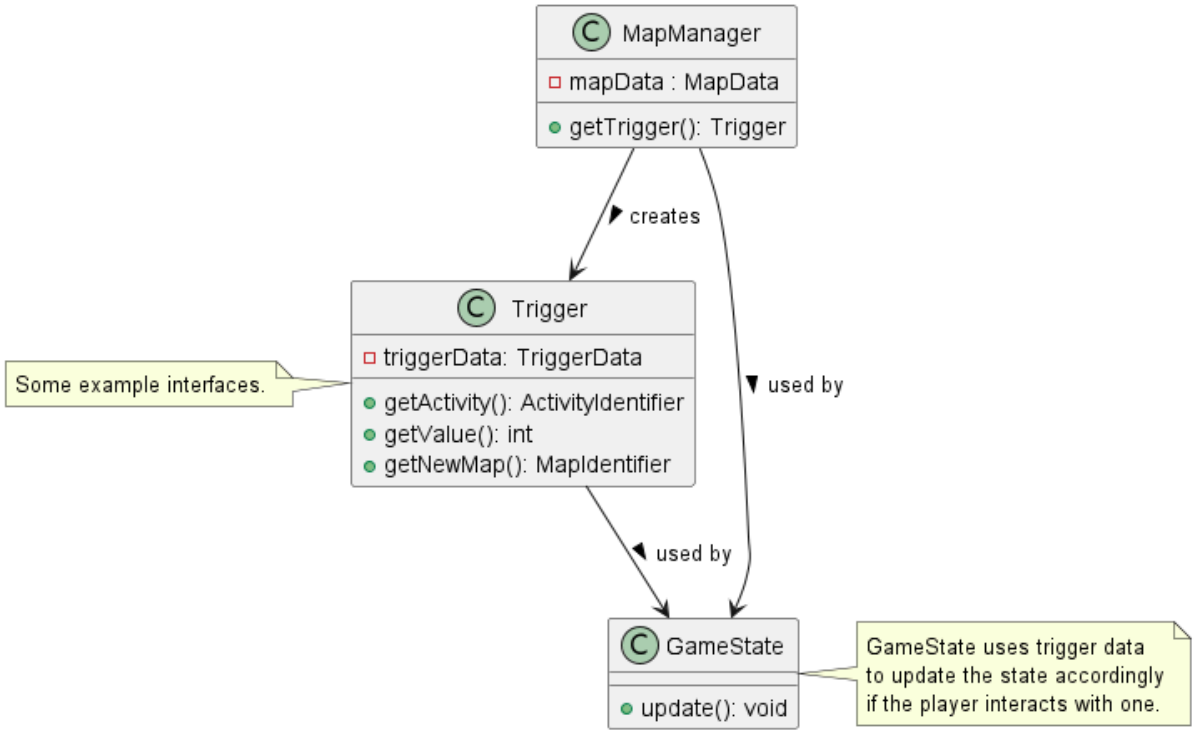
We also considered the popular ECS design paradigm, but decided against it for a number of reasons; this complicates the codebase significantly (perhaps requiring an external library which would cost developer time to learn), and enforces a particular way of doing things which limits flexibility. The advantages of ECS are numerous, but none seemed particularly important for our project:

- *Composition*: The scope of the project is small enough that we felt there was little benefit to limiting ourselves in this way; more than adequate composition is achievable without adopting the whole ECS paradigm.
- *Separation of data and logic*: Less intuitive to reason about in this context compared to object-oriented design, where instances both own their data and act upon their logic. We didn't consider this to be a problem in our project, as it doesn't require features such as modding or scripting support.
- *Performance*: This is solving a problem that we don't have; this game will never conceivably have a number of entities which would cause performance degradation. Given that the project uses a memory-managed language, choosing a performant ECS library or otherwise implementing one would be a difficult task. Performance issues are more likely to arise from incorrectly managing the lifecycle of objects provided by the underlying library we chose.

UML class diagram of the GameState. (Created using PlantUML [1] and PlantUML Integration [2]).
The GameState essentially acts as the coordinator of the game, keeping track of the player, the time and advancing the day, as well as all of the different activities the player can do

(along with their limits). When the update() method is called, the GameState uses the provided action to transform itself, ensuring all game rules are followed. For example, if the player interacts with a place of study, the game will ensure that this activity hasn't been performed more than once in the current day, and then update the data accordingly.

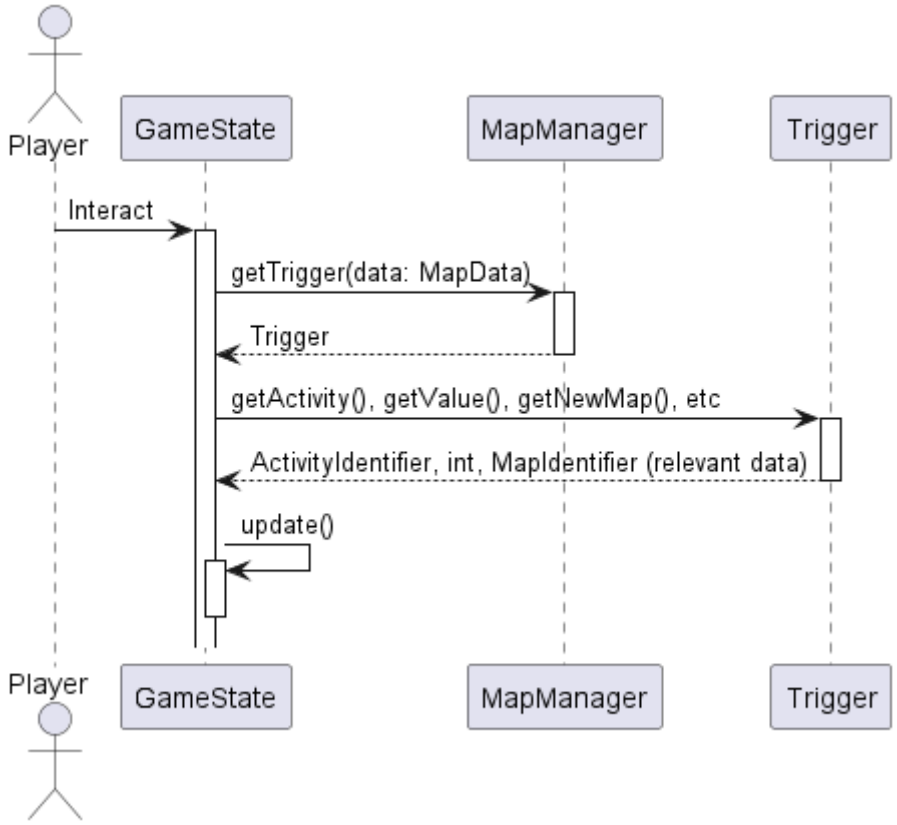
In the evolution of the project, we found that integrating the map (which we loaded from an external editor) was quite involved. Rather than having a separate construct for a logical map, we implemented a new MapManager component which provided interfaces to both the GameState and the Renderer to work with the same data. Although this added a level of coupling to the system, it was a more convenient system to implement and use and maintain encapsulation of the map data. Furthermore, it solved questions of who was responsible for managing the lifetime of the map-related objects.



UML diagram of the interaction between MapManager, Trigger, and GameState. (Created using PlantUML [1] and PlantUML Integration [2]).

Another interesting development that came about late in the development cycle was our Trigger system. When creating the map using the external editor, the designer can add triggers which correspond to different interactable activities and game events. This data is represented in the game state as the Trigger class, which holds the trigger data that the GameState uses to update the state. This design ensures the data is encapsulated and the GameState is not concerned with the underlying implementation of triggers and maps.

For example, a trigger may allow the player to travel to a different map and would contain the data for which map to move to and the coordinates of the map. A different trigger might have data detailing which activity it is, and how many points it gives.



UML sequence diagram depicting Trigger system. (Created using PlantUML [1] and PlantUML Integration [2]).

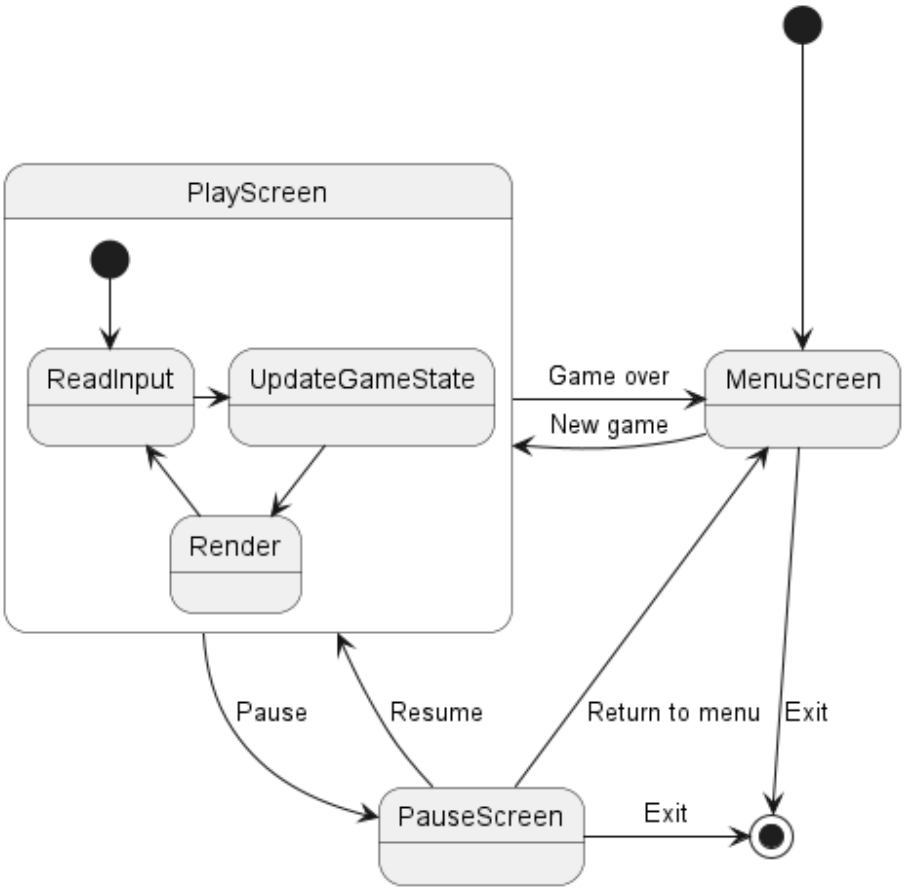
This system is easily extendable and allows for more team members with different proficiencies to effectively collaborate on the project.

The GameState handles functional requirements: FR_INTERACT_DISTANCE, FR_DOUBLE_STUDY, FR_END_DAY, FR_EAT_ENERGY, FR_ACTIVITY_ENERGY_REQ, FR_FAILURE, FR_SUCCESS, FR_TIME_LIMIT
And is partially responsible for some control-scheme requirements such as FR_MOVEMENT_CONTROLS.

Renderer

The Renderer component is conceptually very simple; it is concerned only with drawing the current state of the game to the screen. It interfaces heavily with the underlying LibGDX library code. Its responsibilities are split into sections for drawing the game world and drawing the HUD, with supporting classes for holding asset data.

The Renderer is responsible for functional requirements: FR_ANIMATION, and also participates in those which engage the HUD or dialogue boxes.



UML state diagram showing the different screens in the game. (Created using PlantUML [1] and PlantUML Integration [2]).

This diagram shows how our game loop is contained within the PlayScreen, and how each of the screens in our game relate to one another.

References

[1] PlantUML. "PlantUML" Accessed March 20, 2024. https://plantuml.com/.
[2] JetBrains. "PlantUML Integration" by Vojtěch Krása, JetBrains Marketplace. Accessed March 20, 2024. https://plugins.jetbrains.com/plugin/7017-plantuml-integration.