UNIVERSITY OF YORK

DEPARTMENT OF COMPUTER SCIENCE

Architecture
Cohort 2 - Group 16 (Skloch)

Group Members:

Charlotte MacDonald
Hollie Shackley
Luis Benito
Kaustav Das
Sam Hartley
Owen Gilmore

Originally by: Noah Forryan, Lewis Morgan, Naufal Tun Thamanian, Tom Owen, Dan Manby, José Fernandes of Group 19

Tools Used

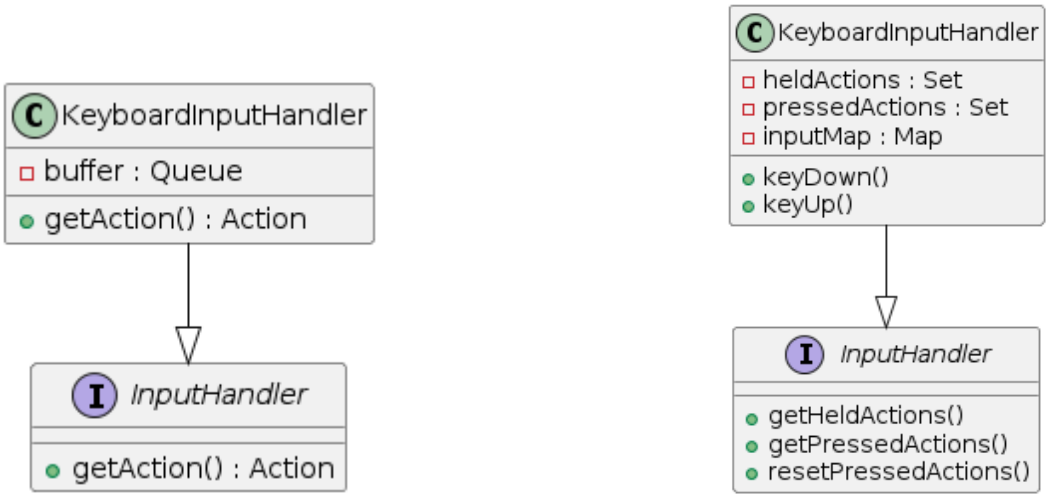
We used plantUML to create the structural and behavioural diagrams needed as we could use it in many different formats. As we are using Google Docs for our reports, plantUML Gizmo is a helpful extension that can be used to embed diagrams into a Google Doc. This means if we were to change them in the future it would be easy to do so. There is also an IntelliJ IDEA plugin [3] which goes alongside our choice of that IDE for implementation and was used by the original group working on this.

When considering the current architectural design it was clear some parts were unnecessary, overcomplicated or handled too much of the project. This caused us to redesign the architecture behind the game. The general Renderer and Trigger classes have been removed and the other classes will be restructured to function without them. The aim is to better divide up the responsibilities of the classes so they work in smaller, more cohesive units.

The game's classes (other than HeslingtonHustleGame) are split into these categories:

- Input - Responsible for taking in the user's raw input and translating it into actions the game can understand.
- Map - Can load the maps, check which rectangles are colliding with the player and find the nearest object to the Player.
- Renderer - Renders the map, character and HUD.
- Screens - Allows the player to move through different screens for the main menu, game screen, leaderboard, options, credits and pause screen.
- Sound - Responsible for managing all music and sound effects.
- State - The main game logic

Input



UML class diagram showing an overview of input handling.

(Created using PlantUML [1] and PlantUML Integration [2]).
Left: Previous design, Right: Current design.

The original KeyboardInputHandler transformed the user's raw keyboard input into a game Action. It also maintained a small FIFO buffer of actions. We deemed this buffer to be unnecessary so it was removed from the current implementation. It does however still translate the user's inputs into Actions the game can understand. The ability to store pressed and held actions has now been added. The set of held actions includes all the keys currently being held down. The set of pressed actions contains Actions for the current frame which should be wiped afterwards. This allows us to distinguish between what inputs are continuously being used and which do not need to be used beyond a frame. Now instead of just getting the Action, when a key is pressed down, the input map will be used to map the key pressed to an Action that needs to be performed. When a key is released, the relevant Actions need to be removed from held and pressed Actions. The InputHandler interface extends LibGDX's InputProcessor which is convenient and neat (although unnecessary). Now that there are held and pressed actions the functionality will change as it can no longer just get general Actions.

There was an initial Input-Update-Render sequence diagram [REPLACE] but this has now become obsolete due to the removal of the Renderer class. Currently, keyboard inputs are translated into actions using a map which maps the key pressed to an Action. Then this action will be added to pressed and held actions. The PlayScreen is then responsible for checking for any Actions in pressed and held actions. There is an Action class which contains all the possible actions that can be taken in the game. Methods are then called by the PlayScreen from other classes. For example, the Player class has a move() method which is called when a moving related Action is detected.

Map

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.

The MapManager can check what current map is loaded and load new ones. It is also responsible for checking which Rectangles (collision boxes) are in the same space as the user as this causes a collision. PlayScreen uses the functionality of the MapManager to ensure the user does not end up inside an object at any time. The MapManager can also check where the nearest interactable object is and how far away it is. This is used by the PlayScreen to inform the player when they are able to interact with an object and what.

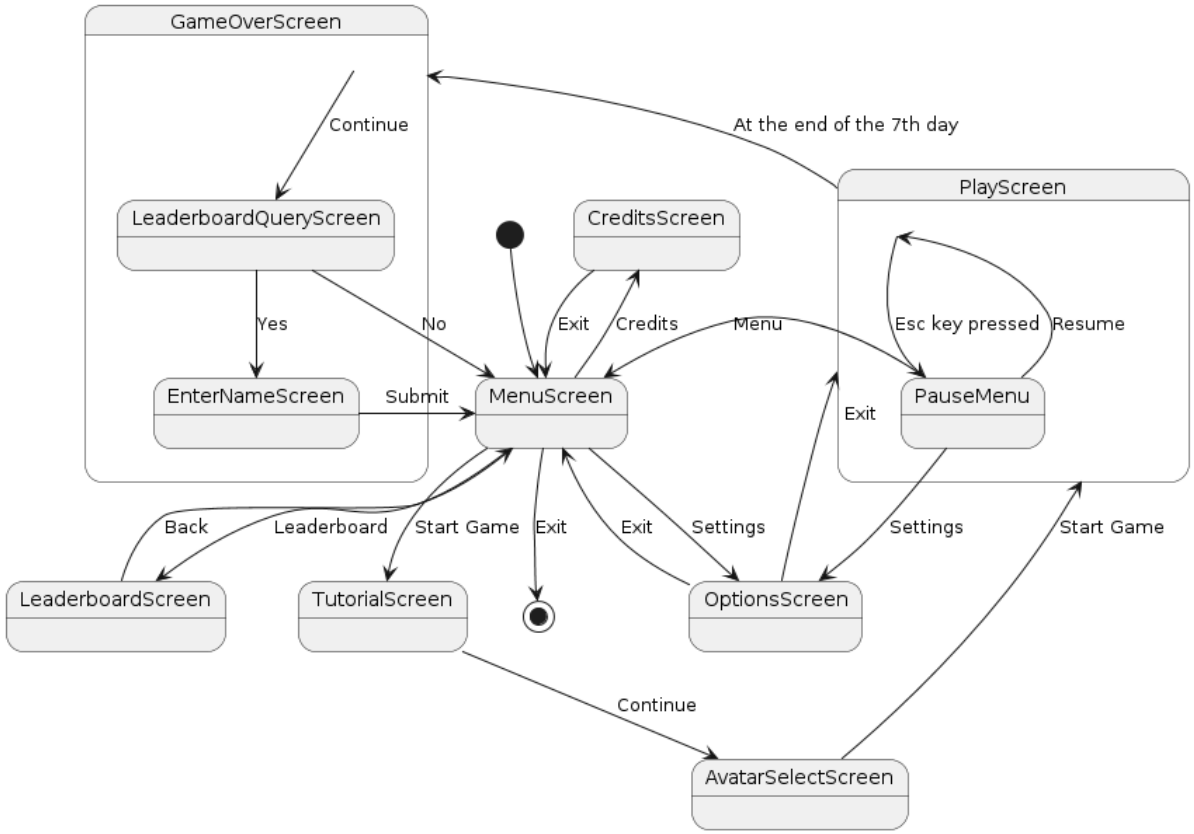
Renderer

The original Renderer component was conceptually very simple; it was concerned only with drawing the current state of the game to the screen. It interfaced heavily with the underlying LibGDX library

code. Its responsibilities were split into sections for drawing the game world and drawing the HUD, with supporting classes for holding asset data.

When updating the architecture, it became clear the generic Renderer class wasn't really necessary and each of the separate rendering classes could be called directly from the PlayScreen. First, the PlayScreen will use the MapManager class to find the MapRenderer and use that to render the map. Then the CharacterRenderer class can be used to render any characters that are needed in the game including the main character (player) and any NPCs. When rendering characters this class checks which direction they are facing to ensure the correct asset is shown. Finally, the HudRenderer will render the HUD (heads-up display). This is all the static information on the screen the player will need including the time, date, energy level and a label for when interactions are possible. The TextureManager is responsible for storing textures and animations to make them easier to retrieve when necessary.

Screens



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

The initial version [REPLACE WITH LINK TO WEBSITE] had just 3 screens: the Menu screen, Pause screen and Play screen. The player will begin in the Menu and can either exit or begin a game. While in the game, they can click the Esc key to exit to the Pause screen. From there they can rejoin the game or exit to the main menu. This diagram also displayed ReadInput, UpdateGameState and Render which are actions and not states.

The current state diagram of the different screens in the game has removed the actions as they are not states and shouldn't be included. We have also added lots of new screens including the CreditsScreen, OptionsScreen, LeaderboardScreen and GameOverScreen. From the Menu screen you can now also access a Leaderboard, Credits and some Settings that allow you to change the music and sfx volume. At the end of the game it now displays the GameOverScreen where the user will see their results and can add themselves to the Leaderboard.

The most major change has been to the PlayScreen class. This class is now responsible for checking for inputs, allowing the player to move and ensuring they remain inside the map and outside of objects. It also calls the map, player and HUD rendering classes to render the game as well as being in charge of telling the game state to pass time and notifying other classes of the game being over.

Due to feedback from the user evaluation, we added a TutorialScreen class which appears after the user decides to start the game. This will then take you to an AvatarSelectScreen where the user can select between two avatars. They can then continue to the PlayScreen. The original implementation of this game had no avatar selection so this was added as it is one of the user requirements from our client interview.

Sound

There was no Sound functionality in the original game so it was added from scratch. The SoundController class was designed to manage the playing, switching, loading and disposing of sounds and music in the game. The ability to set the volume of music and sound effects separately was also added and these will be changeable by the user in the options screen.

State

State contains most of the game logic. In the initial design, the State class acted 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 State class 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.

The Achievement class is responsible for tracking the user's progress towards streaks and achievements. The Activity class contains information about all the activities that the user can perform and assigns them a score, an amount of energy they use up, the time it takes to complete them and how many times they have been completed. The Clock class stores the day and time and is responsible for increasing them when necessary. The State class uses the Clock to pass days and check when 7 days have passed so it can move to a game over state.

The DialogueManager handles the dialogue delivered to the player and their selection (when there is one to make). The LeaderboardManager is used when a user wants to add their score to the leaderboard. It will check if their name is valid and will write their name and score to the correct position on the leaderboard. The NPC class stores the position, type and direction of an NPC. It can reposition them onto the screen and uses the CharacterRenderer to render them onto the screen.

The Player class stores data about the player and can set its position on the screen. It is also responsible for moving the player and checking if it is colliding with any objects.

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

HeslingtonHustleGame Class

This class extends LibGDX's Game class and originally was just used to change the screen. We now allow it to store the height and width of the game as this could be helpful in the future when altering the aspect ratio. As the OptionsScreen is available from both the PlayScreen and MenuScreen, it was necessary to store which screen the Options had been accessed from as you don't want the game to end when accessing the options and being sent back to the menu screen to start again. It also passes on information about the user's score when switching to the GameOverScreen. We chose to implement the new leaderboard requirement by using text files so this class now has the ability to read text files.

Original Architecture Ideas

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:

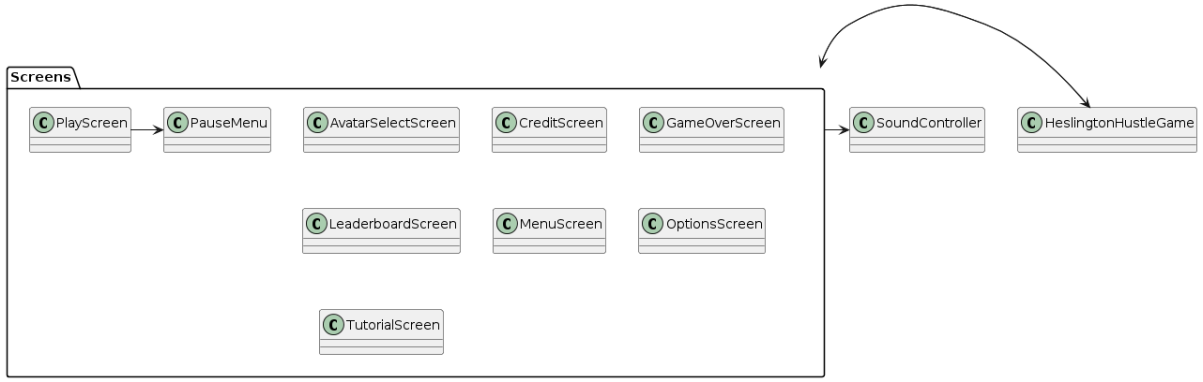
- *Composition*: The scope of the project is small enough that we felt there was little benefit to limiting ourselves in this way; more than adequate composition is achievable without adopting the whole ECS paradigm.
- *Separation of data and logic*: Less intuitive to reason about in this context compared 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.

In the end, we centred the game around the game state which coordinates the whole game depending on the player's action. However the architecture was changed during assessment 2 to focus more on the PlayScreen class. This class is responsible for checking for inputs, moving the player, rendering the game and knowing when the game is over. It interacts with many other classes in order to ensure it knows the current state of the game.

Class Diagrams

An overall class diagram was created with PlantUML but it was very large and confusing so it was broken down into smaller class diagrams focussing on certain parts of the functionality.

Overall Screen Class Diagram



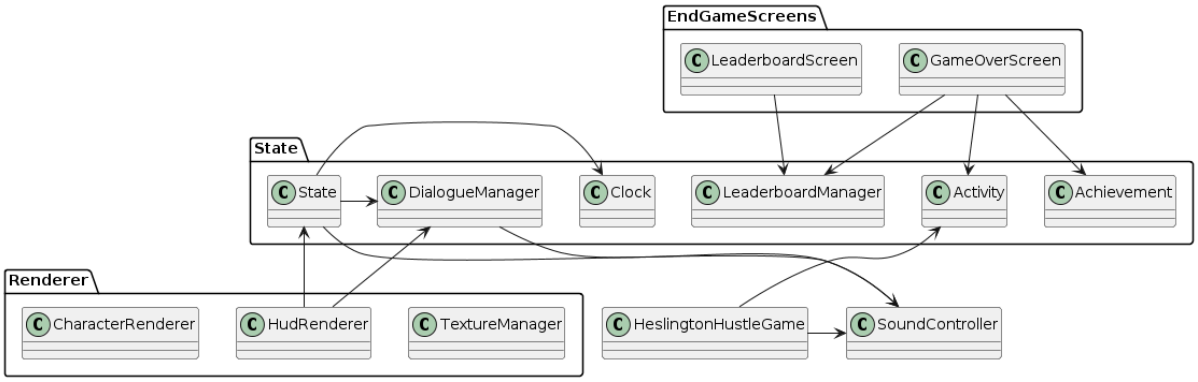
Most of the screens only use the SoundController and HeslingtonHustleGame. The HeslingtonHustleGame class uses the Screens as it is responsible for switching between them. The PlayScreen, GameOverScreen and LeaderboardScreen use a lot more classes which will be explored in the following diagrams.

Main Game Class Diagram

This diagram (Figure 1: Main Game Class Diagram) included all classes that interact with the PlayScreen class as it is one of the major parts of the game. As the PauseMenu is only accessible inside the PlayScreen, this has also been included. PlayScreen is responsible for moving the player so must be able to interact with the Input. It is also responsible for rendering the map, character and HUD so uses the MapManager, CharacterRenderer and HudRenderer. PlayScreen uses State when it wants to pass time, handle actions and handle interactions. It also passes the State to the HudRenderer so the correct date, time and energy are displayed on the screen. For this reason, HudRenderer also uses State. HudRenderer and PlayScreen also both use DialogueManager as the HudRenderer is responsible for rendering the correct dialogue boxes on the screen and the PlayScreen will call the HudRenderer to do so when an action has been performed. The Player class and Action enumerable are used by PlayScreen to handle moving the player and passing the Player's Actions to the relevant classes/methods. State, HeslingtonHustleGame and DialogueManager all use SoundController to ensure the correct sound effects are played at the correct times. State uses DialogueManager as when the game state changes to an interaction dialogue needs to be loaded. HeslingtonHustleGame uses Activity to pass the completed activities to the GameOverScreen when switching to it at the end of the game. State uses Clock to pass the time and day as well as resetting it

when the game ends. The NPC class uses the CharacterRenderer to render itself onto the screen. It also uses the Facing enumerable to store which direction it should be facing in.

End of Game Class Diagram



This diagram focuses on the GameOverScreen and LeaderBoardScreen. It includes how the non-screen classes interact with each other as well but this will not be explained as they were included in the previous diagram. The LeaderboardScreen must use the LeaderboardManager as this is the class which manages reading from and writing to the leaderboard. GameOverScreen uses the LeaderboardManager as well because at the end of the game the player will be able to submit their score to the leaderboard. LeaderboardManager is called by GameOverScreen to check if their name is valid and to update the leaderboard with their name and score. GameOverScreen uses Activity to calculate the Player's score based on what activities they have done and when. It also uses Achievement to alter the score if any achievements have been completed. These achievements are also displayed on the GameOverScreen as well as the score.

Relating Architecture to Requirements

User Requirements

ID	Architecture
UR_MENU	The MenuScreen class is opened when the game is launched and has buttons leading to different parts of the game. These include: Start game, Leaderboards, Settings, Credits, Exit. Start game leads to the PlayScreen, Leaderboards leads to the LeaderboardScreen, Settings leads to OptionsScreen, Credits leads to CreditScreen and Exit will close the application.
UR_MAP	The MapManager class facilitates communication between the State and renderer classes.
UR_MOVEMENT	The KeyboardInputHandler takes key presses as input and maps them to an action. For the key presses W, A, S and D they are mapped to the MOVE_UP, MOVE_LEFT, MOVE_DOWN and MOVE_RIGHT Actions respectively. The PlayScreen will check for this input and use CharacterRenderer to render the player onto the screen with their updated direction, location and if they are moving or not.
UR_OBJECTS	Using the CharacterRenderer both the player and the NPCs can be rendered into the game. The PlayScreen will call MapManager's renderNPCs() method and CharacterRenderer directly for the player. The MapManager will load the map which contains buildings and other decorations.
UR_INTERACT	KeyboardInputHandler will map a key press to an INTERACT Action. PlayScreen checks if this type of interaction is needed. The State class handles interactions by checking the closest trigger (interactable object). Then it is handled depending on if it is a sign, dialogue, sleep or

	other activity. If a new map is needed (interaction at a bus stop) then this will be rendered with the MapManager.
UR_ENERGY	Energy is stored in the [REPLACE] class. The HudRenderer class, which is responsible for rendering the heads up display, will render an energy bar onto the screen with the current value of energy.
UR_TIME	The Clock class is responsible for storing the time and day, as well as increasing them. When the time gets to 24 hours it will be reset to 0 hours while also increasing the day. The State class creates an instance of the Clock class and when advancing the day, it will use the getDay() method to get the current day. If this is 7 (or more), the game will then enter the game over stage.
UR_GAME_END	When the State class checks the current day from the Clock class, if it is 7 or more then the game will end. The GameOverScreen will be opened. This class is responsible for calculating the player's score from the activities it has completed which are passed when it is instantiated. The data is then displayed on the screen for the player to read.
UR_ANIMATION	The player assets are available in moving and non-moving states. The player can also face left, right, up and down. The CharacterRenderer class will check if the player is moving (or not) and which direction they are facing. It will then set the character sprite to appear as it should.
UR_PAUSE	When the player pressed the Esc key, the KeyboardInputHandler class will map this to the PAUSE action. The PlayScreen checks if a PAUSE action needs to be handled. PlayScreen creates an instance of the PauseMenu class and calls the showPauseMenu() method from that class which will display the buttons (options) for the Player to choose between. From the pause menu you can resume the game, access settings or exit back to the main menu.
UR_SCORING	The GameOverScreen class is responsible for calculating the player's score. It does this based on the activities it has completed during the game. The PlayScreen class will check the State class to see if the game is over. The HeslingtonHustleGame class is then responsible for handling the end of the game by switching to the GameOverScreen and passing it the completed activities and achievements.
UR_TUTORIAL	The TutorialScreen class exists to display information for a tutorial to inform the player on their objective and how to play the game. There will also be a button for the player to continue when they wish to do so, which will then call HeslingtonHustleGame's switchScreen() method to be able to switch to the AvatarSelectScreenClass.
UR_ADDITIONAL_MAP	The PlayScreen class creates an instance of the MapManager class to load the Campus East map initially. If the player interacts with an object that requires a new map, this will be loaded by the MapManager.
UR_LEADERBOARD	The GameOverScreen class will form a popup window using the LeaderboardManager class which will enable the player to add themselves to the leaderboard. It will then call HeslingtonHustleGame to switch to the MenuScreen. From the MenuScreen the player can select the Leaderboard button which will make HeslingtonHustleGame switch to the LeaderboardScreen. This class is responsible for drawing and rendering the leaderboard.
UR_STREAKS	When GameOverScreen is instantiated by the HeslingtonHustleGame class, it will calculate the player's score from the activities the player has completed and their achievements. The Achievement class is responsible for tracking the player's progress on each achievement. Each Achievement has a score and if the player has completed it the GameOverScreen will get the score of the achievement and add it to the total score.
UR_AVATARS	The TutorialScreen class will call HeslingtonHustleGame to switch to the AvatarSelectScreen. This will display the avatars the player can choose between and pass their choice to the

	HeslingtonHustleGame with the startGame() method.
UR_CREDITS	From the Main Menu the player can access the Credits screen. The HeslingtonHustleGame class will switch the game to the CreditScreen class. The CreditScreen class will load the text for the credits onto the screen. There will be a button for the player to exit back to the Main Menu. The HeslingtonHustleGame class is also responsible for loading the text file with the credit data.
UR_MAP_OVERVIEW	When the player pressed the M key, the KeyboardInputHandler will map this to a MAP action. The PlayScreen will check for the existence of a MAP action in the pressedActions and if it is there will zoom the map out (or in if it is already out).
UR_SOUND	The SoundController class will load the music and sound effects and give them a key. When its playSound() method is called, it will check what sound type is needed and map that to the relevant sound which is then played. Music and sfx volume can be changed in the OptionsScreen.

Functional System Requirements

ID	**Architecture**
FR_MENU_NAVIGATION	The KeyboardInputHandler maps key presses to actions. W, A, S and D map to up, left, down and right respectively. The enter key, space key and E key all map to an Interaction Action which is used when selecting buttons.
FR_MENU_SELECT_CONTROLS	The MenuScreen class draws 4 buttons, Start game, Leaderboard, Settings and Exit. Start game leads to the TutorialScreen, AvatarSelectScreen then the PlayScreen. The leaderboard button leads to the LeaderboardScreen. The Settings button leads to the OptionsScreen and the Exit button will close the game. The HeslingtonHustleGame switchScreen() method is used to swap between screens when a button is pressed.
FR_START_GAME	After selecting the MenuScreen class, HeslingtonHustleGame switches the current screen to the TutorialScreen. The player can then press the Continue button drawn by this class, which will call HeslingtonHustleGame to switch the current screen to the AvatarSelectScreen. When the player has chosen their avatar this is passed to the CharacterRenderer so the correct one is shown on screen. Then the PlayScreen is responsible for calling the Map, Character and Hud Renderers to render the map, player and NPCs onto the screen.
FR_ADDITIONAL_MAP and FR_MAP_BUS_STOP	A bus stop is included in the maps when rendered by the MapManager. Initially, PlayScreen will call it to render the campus east map. If an Action from KeyboardInputHandler requires the map to be switched, PlayScreen will call the MapManager to render this new screen. The Player's location is also moved to the bus stop location on the other map.
FR_INTERACTION_CONTROLS	The KeyboardInputHandler class maps Enter, Space and E key presses to an INTERACTION Action. When one of these Actions is detected by the PlayScreen class, the closest interactable will be interacted with.
FR_INTERACT_PROMPT	The State class stores the closest trigger (interactable object). When the closest trigger is not equal to null, text saying the player can interact will appear on the screen. When it is null, the player is not near a trigger so there will be no text rendered.
FR_INTERACT_DISTANCE	When an interact key (space, enter, or e) is pressed, KeyboardInputHandler maps this to an interaction Action. PlayScreen checks if this type of interaction is needed. The State class handles interactions by checking the closest trigger (interactable object). This means the Player will only interact with the closest object. The closest trigger is always updated so if the Player is not near any interactable objects then no interaction will be possible as

	the closest trigger will be null.
FR_READ_BOOK	There will be a bench which is a trigger rendered with the map by the MapManager. When interacted with, the DialogueManager will pop up dialogue about reading a book.
FR_PLAY_SPORT	There will be a sports village and gym which are triggers rendered with the map by the MapManager. When interacted with, the DialogueManager will pop up dialogue about playing sports.
FR_FEED_DUCKS	There will be some ducks which are triggers rendered with the map by the MapManager. When interacted with, the DialogueManager will pop up dialogue about reading a book.
FR_STUDY	The Study activity uses 25 energy points, as you start with 100, this is a reduction of 25%. It will also increase the timesCompletedToday and timesCompleted for this type of activity.
FR_DOUBLE_STUDY	The maxPerDay attribute will be set to 2 for the Study activity. When querying whether an activity can be completed, it will check timesCompletedToday against maxPerDay and if it is more than or equal to, the user will not be able to complete the activity.
FR_EAT	The Eat activity uses 10 energy points, as you start with 100, this is a reduction by 10%. It will also increase the timesCompletedToday and timesCompleted for this type of activity.
FR_SLEEP	When the user presses an interact key (space, enter or e) the KeyboardInputHandler maps this to an interact Action. When the closest interactable is a “sleep” type, the Clock will be used to increment the day. The State will replenish energy to 100 energy points (100%).
FR_ACTIVITY_ENERGY	When the user presses an interact key (space, enter or e) the KeyboardInputHandler maps this to an interact Action. When the closest trigger is anything other than eating, sleeping or clubbing, energy is subtracted by 10 (which is a 10% reduction as initial energy is 100). This is done by the State class. When the closest trigger is a club, energy is reduced by 50.
FR_ACTIVITY_ENERGY_REQ	The State class will get the amount of energy the relevant Activity will use up. If the amount of energy the Player has left is less than this, they will not be allowed to complete the activity.
FR_SCORING	The activities the user completes are stored by the State class until the end of the game. They are then passed to the GameOverScreen which can calculate the score by getting the score of each Activity (with Activity’s getScore() method) and adding it to the total score.
FR_SCORING_GAME_OVER	The score is only calculated in the GameOverScreen class so this will be unavailable until it is drawn on the game over screen.
FR_PAUSE	PlayScreen calls HeslingtonHustle to switchScreen() to the PauseMenu screen. This will be drawn and rendered by the PauseMenu class itself.
FR_PAUSE_CONTROLS	When a user presses the Esc key, the KeyboardInputHandler maps this to a Pause Action. When the current actions that need handling are checked by the PlayScreen, it will call the handlePauseAction() method. This will check if it is a pause action, and the screen is not currently paused, it will show the PauseMenu. If it is currently paused, it will unpause the menu by going back to the PlayScreen.
FR_PAUSE_FREEZE	During a pause, the dontMove() method in the Player class means that no matter if W,A,S and D are pressed, the Player will not be able to move. The attribute of the PlayScreen class isPaused() will be set to true. This means when an action occurs, PlayScreen will first check if the screen is paused and this will determine if it can happen or not.
FR_TIME_LIMIT	The doActiivty() method in the State class will check whether the time is too late. If it is too late, it will add dialogue to the DIalogueManager saying this activity is unavailable/the building is not open etc.
FR_ENERGY	The HudRenderer is responsible for rendering the energy bar assets onto the screen. It will add an energy bar to the screen and will check the Player’s current energy level. It will then scale the bar according to this amount.

FR_TIME	The HudRenderer is responsible for rendering the current day and time onto the screen. PlayScreen will call HudRenderer's updateValues() method to update the time, day and energy. It will set the text of the time button to be the current time which can be retrieved from the Clock class. It will also set the text of the day button to the current day, which can also be retrieved from the Clock class.
FR_SCORE_CALC	The Activity class has a score attribute. Each type of activity will therefore have a score. When the score is calculated in the GameOverScreen, it will go through all the activities that have been completed, check their score and add that to the total score. If the score is negative, adding it will reduce the score. The achievement class can also be used to add points for good achievements, and remove points if necessary.
FR_LEADERBOARD	The LeaderboardScreen is switched to using the switchScreen() method in HeslingtonHustleGame. It is called by the MenuScreen when the Leaderboard button is pressed. It draws a table, fills it with the correct information which is read from a file, then renders it onto the screen. When adding a Player's score to the leaderboard, the LeaderBoardManager class's writeScore() method is called by the GameOverScreen. This will rewrite all the current scores to the leaderboard, but if the new score is higher it will write that first so the scores are in descending order.
FR_STREAKS	When GameOverScreen is instantiated by the HeslingtonHustleGame class, it will calculate the player's score from the activities the player has completed and their achievements. The Achievement class is responsible for tracking the player's progress on each achievement. Each Achievement has a score and if the player has completed it the GameOverScreen will get the score of the achievement and add it to the total score. Only 3 achievements are instantiated as described by FR_STREAKS.
FR_STREAK_GYM_BRO	The GameOverScreen instantiates an Achievement with title "Gym bro", description "Go to the gym at least 3 times per week" and a score of 500.
FR_STREAK_LOTD	The GameOverScreen instantiates an Achievement with title "Duck duck go", description "Feed the ducks 6 times" and a score of 300.
FR_STREAK_JOGGER	The GameOverScreen instantiates an Achievement with title "Walker", description "Walk 200 steps each day" and a score of 200.
FR_MAP_OVERVIEW_CONTROLS	The KeyboardInputHandler class maps M key presses to a MAP Action. The PlayScreen will check for the existence of a MAP action in the pressedActions and if it is there will zoom the map out (or in if it is already out).
FR_MAP_OVERVIEW_MOVEMENT	The KeyboardInputHandler maps key presses to actions. W, A, S and D map to up, left, down and right respectively. When these actions are detected by PlayScreen, it will call the move() method of the Player class to reposition the player. The CharacterRenderer will then render the Player's new location. It will also consider whether the player is moving and which direction they are facing with the Facing enumeration.
FR_CREDITS	From the Main Menu the player can access the Credits screen. The HeslingtonHustleGame class will switch the game to the CreditScreen class. The CreditScreen class will load the text for the credits onto the screen. There will be a button for the player to exit back to the Main Menu. The HeslingtonHustleGame class is also responsible for loading the text file with the credit data.
FR_TUTORIAL	The TutorialScreen is switched to using the switchScreen() method in HeslingtonHustleGame. It is called by the MenuScreen when the Play Game button is pressed. The screen will draw and render all the necessary information.
FR_AVATARS	The TutorialScreen class will call HeslingtonHustleGame to switch to the AvatarSelectScreen. This will display the avatars the player can choose between and pass their choice to the HeslingtonHustleGame with the startGame() method.

Non-Functional System Requirements will not be included as they are unrelated to the system's architecture.

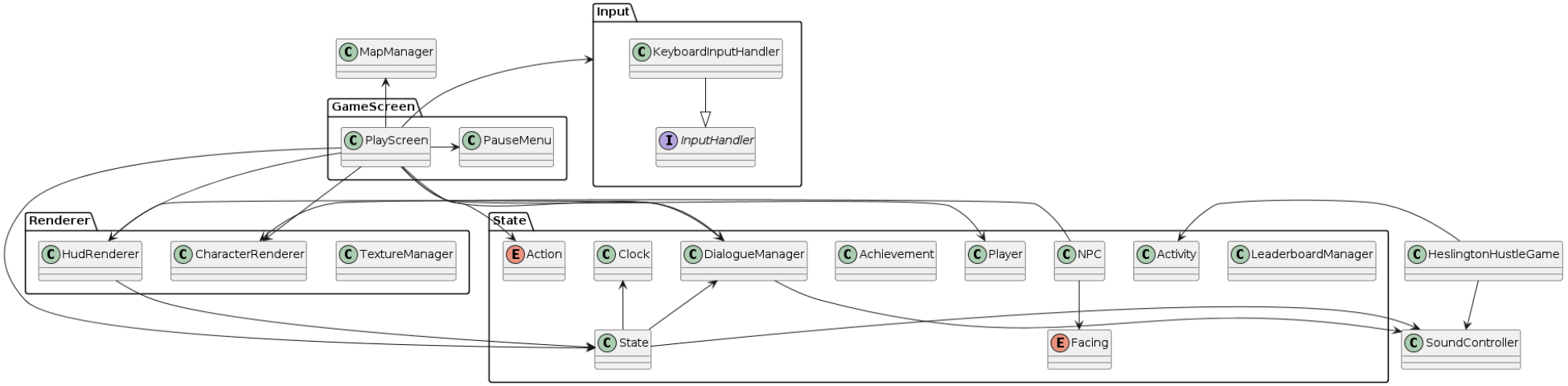


Figure 1: Main Game Class Diagram

References

[1] PlantUML. "PlantUML". [Online]. Available: https://plantuml.com/.
[2] fuhrmanator. PlantUML Gizmo. [Online]. Available: https://workspace.google.com/marketplace/app/plantuml_gizmo/950520042571
[3] V. Krása. "PlantUML Integration", JetBrains Marketplace. [Online]. Available: https://plugins.jetbrains.com/plugin/7017-plantuml-integration.