Architecture

"Mathochist Studios" Cohort 4, Team 11

Euan Cottam
Charlie Thoo-Tinsley
Harri Thorman
Will King
Zach Moussallati
Aiden Turner
Marcus Williamson
Joshua Zacek

Introduction

Our game “All-Nigher” is a 2D Maze game set in a university. To prioritise our developer resources, we’ll be using the **LibGDX** library, saving us time by not needing to re-invent the wheel. Our architecture will be main **event-driven**, choosing to go this route due to LibGDX’s lack of a main loop[1].

Design Languages and Tools

To create our class diagrams, we used the Unified Modeling Language (UML), using the PlantUML language to create these diagrams using text. By using a programming language and text to generate these diagrams instead of visual tools and editors, we can create class diagrams much more efficiently. To render the diagrams we used PlantText, an online PlantUML text editor. Alternatively, we could’ve used an offline application to render these diagrams as a safety measure in case we did not have internet access.

Initial Architecture

Screens

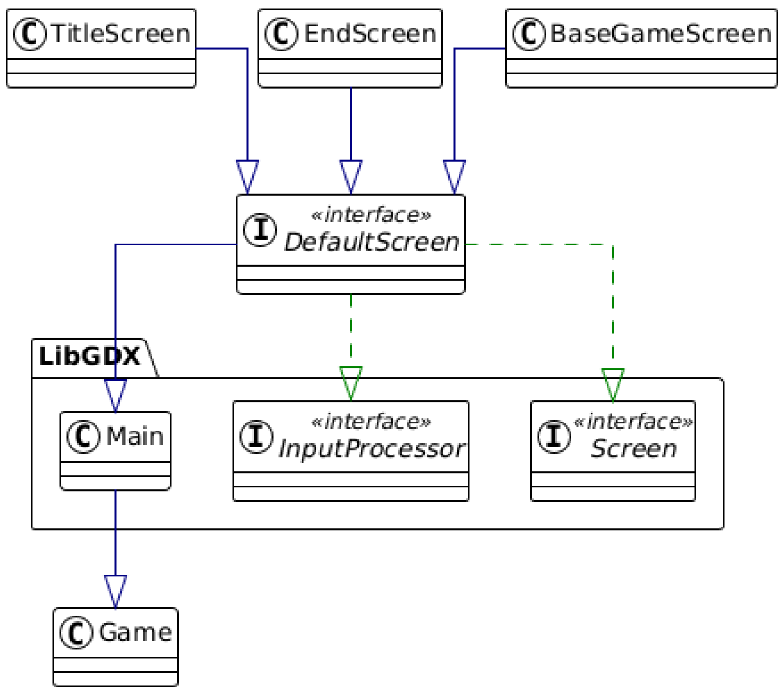


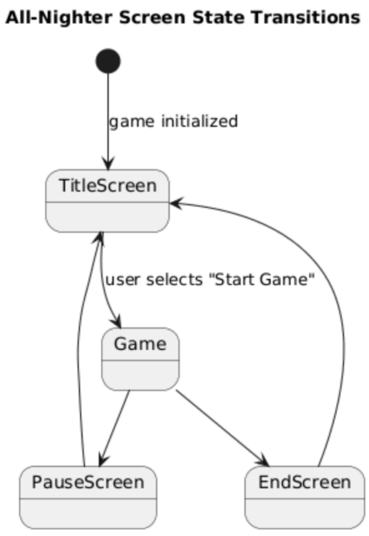
Figure 1: Simple UML class diagram of our game’s architecture

We chose to start off with an abstract representation of our game, which we would be able to expand as our needs during development evolved. At the core of our game lies an interface containing key methods and data used across our game’s screens. Inside this interface, we placed many of LibGDX’s provided classes, such as input processing to detect key presses and mouse clicks, and the standard Screen class. Meanwhile, the Main class, a direct child of LibGDX’s Game class, is the heart and entry point of our game.

The composite relationship between the Main class (entry point of the game) and the DefaultScreen interface shows how interlinked these two components are, i.e. without the interface containing basic functions like input processing, the game wouldn't be able to exist.

Flow

Figure 2 displays the transitions between each of our game screens. Our game has an **event-driven** architecture, so each of these screens effectively acts as a separate state for our game, making a state diagram a good fit. Further verifying the event-driven nature of our game, LibGDX itself is also event-driven i.e. there is no "main loop" that renders each frame, instead rendering every time an event occurs[1].

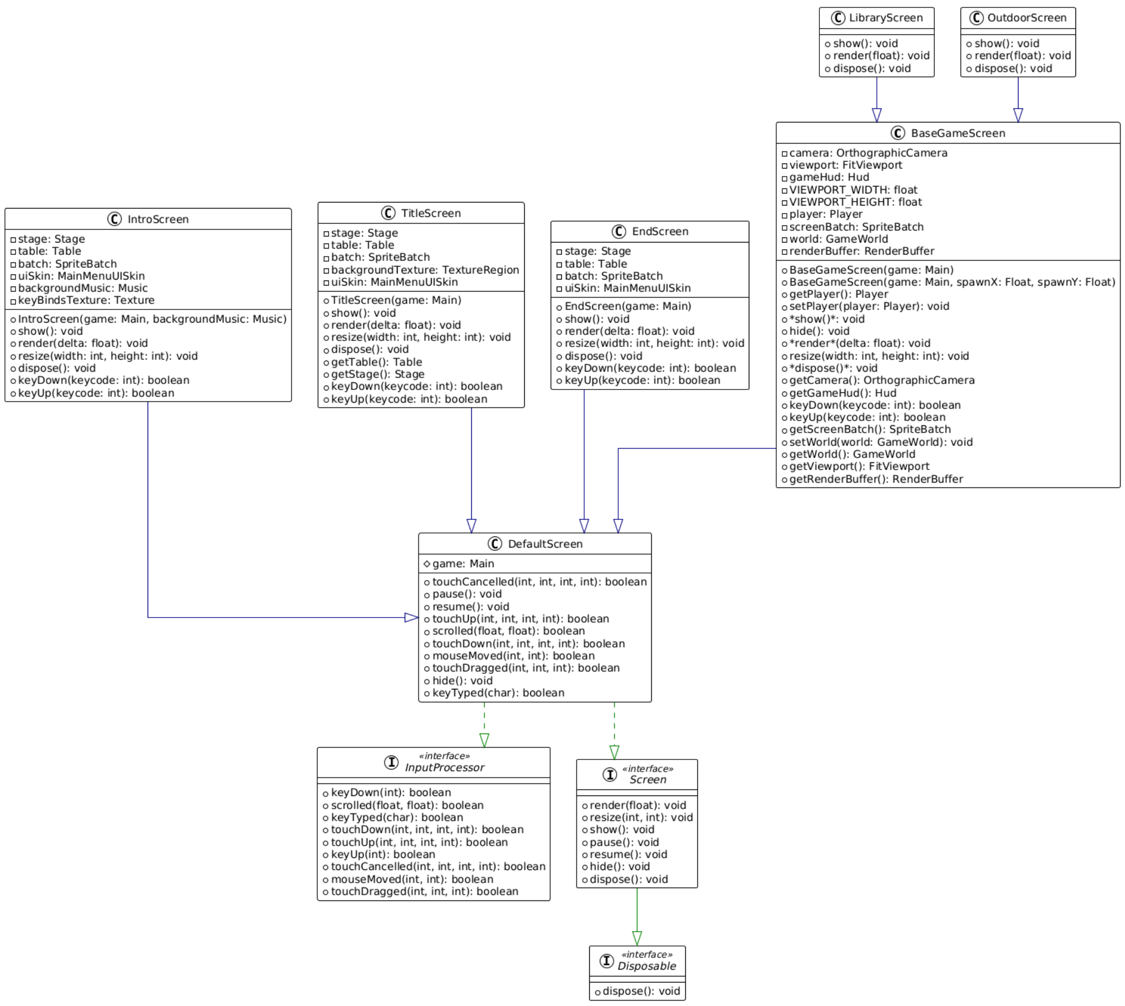


All-Nighter Screen State Transitions
game initialized
TitleScreen
user selects "Start Game"
Game
PauseScreen
EndScreen

Figure 2: UML state diagram

Final Architecture

Screens



LibraryScreen
show(): void
render(float): void
dispose(): void
OutdoorScreen
show(): void
render(float): void
dispose(): void
BaseGameScreen
camera: OrthographicCamera
viewport: FitViewport
gameHud: Hud
VIEWPORT_WIDTH: float
VIEWPORT_HEIGHT: float
player: Player
screenBatch: SpriteBatch
world: GameWorld
renderBuffer: RenderBuffer
BaseGameScreen(game: Main)
BaseGameScreen(game: Main, spawnX: Float, spawnY: Float)
getPlayer(): Player
setPlayer(player: Player): void
show(): void
hide(): void
render(delta: float): void
resize(width: int, height: int): void
dispose(): void
getCamera(): OrthographicCamera
getGameHud(): Hud
keyDown(keycode: int): boolean
keyUp(keycode: int): boolean
getScreenBatch(): SpriteBatch
setWorld(world: GameWorld): void
getWorld(): GameWorld
getViewport(): FitViewport
getRenderBuffer(): RenderBuffer
IntroScreen
stage: Stage
table: Table
batch: SpriteBatch
uiSkin: MainMenuUISkin
backgroundMusic: Music
keyBindsTexture: Texture
IntroScreen(game: Main, backgroundMusic: Music)
show(): void
render(delta: float): void
resize(width: int, height: int): void
dispose(): void
keyDown(keycode: int): boolean
keyUp(keycode: int): boolean
TitleScreen
stage: Stage
table: Table
batch: SpriteBatch
backgroundTexture: TextureRegion
uiSkin: MainMenuUISkin
TitleScreen(game: Main)
show(): void
render(delta: float): void
resize(width: int, height: int): void
dispose(): void
getTable(): Table
getStage(): Stage
keyDown(keycode: int): boolean
keyUp(keycode: int): boolean
EndScreen
stage: Stage
table: Table
batch: SpriteBatch
uiSkin: MainMenuUISkin
EndScreen(game: Main)
show(): void
render(delta: float): void
resize(width: int, height: int): void
dispose(): void
keyDown(keycode: int): boolean
keyUp(keycode: int): boolean
DefaultScreen
game: Main
touchCancelled(int, int, int, int): boolean
pause(): void
resume(): void
touchUp(int, int, int, int): boolean
scrolled(float, float): boolean
touchDown(int, int, int, int): boolean
mouseMoved(int, int): boolean
touchDragged(int, int, int): boolean
hide(): void
keyTyped(char): boolean
«interface»
InputProcessor
keyDown(int): boolean
scrolled(float, float): boolean
keyTyped(char): boolean
touchDown(int, int, int, int): boolean
touchUp(int, int, int, int): boolean
keyUp(int): boolean
touchCancelled(int, int, int, int): boolean
mouseMoved(int, int): boolean
touchDragged(int, int, int): boolean
«interface»
Screen
render(float): void
resize(int, int): void
show(): void
pause(): void
resume(): void
hide(): void
dispose(): void
«interface»
Disposable
dispose(): void

Figure 3: Final class UML diagram of our game screens

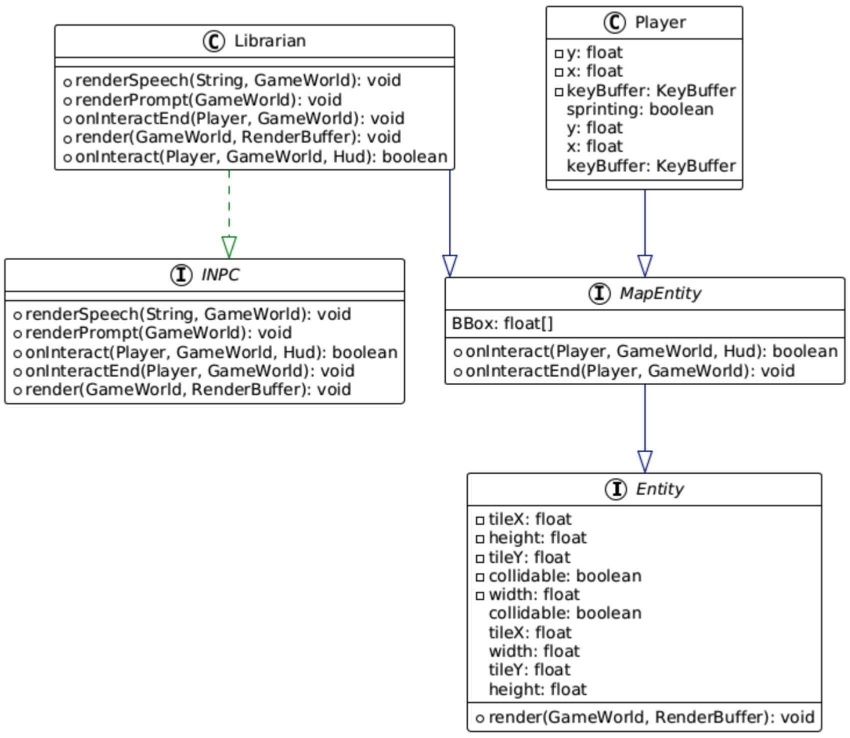
We implemented new screens such as BaseGameScreen which is now a central class of shared attributes (camera, player, world, viewport, HUD, batch) as well as a common setup logic for gameplay. This extends LibraryScreen and OutdoorScreen and they inherit its reusable methods and shared functionality. We also removed PauseScreen as it was a redundant class in implementation due to pause logic being handled by UI overlays of existing screens such as LibraryScreen and OutdoorScreen. Now that pausing the game is built directly into gameplay screens, the system is less complicated, and the player benefits from immediate and seamless transitions.

Moreover, we added the Disposable interface necessitating every screen implements cleanup logic, which avoids memory leaks and ensures proper removal of assets when screen is off or changed, subsequently improving memory efficiency and performance.

Further to the aforementioned, we have expanded the use of interfaces, such as Screen and InputProcessor, so that every screen consistently responds to user actions and manages states effectively. This **event-driven** architecture allows for different screen types to be extended or swapped with ease, while keeping code organised and future testing/improvement simple.

In conclusion, these improvements have made the system more maintainable and modular, as well as making screens more efficient and flexible. The system can be expanded in future and can be relied on for stable and smooth gameplay, as it has adopted a standardised **event-driven** approach, ensured effective resource management, and organised a shared logic across screens[1].

Entity



Librarian
renderSpeech(String, GameWorld): void
renderPrompt(GameWorld): void
onInteractEnd(Player, GameWorld): void
render(GameWorld, RenderBuffer): void
onInteract(Player, GameWorld, Hud): boolean
Player
y: float
x: float
keyBuffer: KeyBuffer
sprinting: boolean
y: float
x: float
keyBuffer: KeyBuffer
INPC
renderSpeech(String, GameWorld): void
renderPrompt(GameWorld): void
onInteract(Player, GameWorld, Hud): boolean
onInteractEnd(Player, GameWorld): void
render(GameWorld, RenderBuffer): void
MapEntity
BBox: float[]
onInteract(Player, GameWorld, Hud): boolean
onInteractEnd(Player, GameWorld): void
Entity
tileX: float
height: float
tileY: float
collidable: boolean
width: float
collidable: boolean
tileX: float
width: float
tileY: float
height: float
render(GameWorld, RenderBuffer): void

Figure 4: UML class diagram of our entity classes

In addition to our **event-driven** architecture, we implemented some elements of **Entity-Component system (ECS)** architecture.

MapEntity acts as a component for entities that exist inside the world/game map, whilst INPC acts as a separate component to entities which are *not* the player. By using MapEntity and INPC as components, it allows for future expandability to our game through new characters that exist on the map, and crucially, other non-player characters (NPC).

We didn’t include ECS in our initial architecture, initially just having a Player class which would react to events. Over time however, it became more visible to us that using an ECS architecture instead of an event-driven architecture would be more viable, once again for its expandability, which would end up proving useful as we added a librarian NPC to our game, which would’ve been more complicated had we used an independent Player event-driven class[2].

Requirements

ID	Architectural Link
UR_SCREEN_SCALABILITY	FitViewport, OrthographicCamera, and resolution-aware screen classes such as BaseGameScreen with its inheriting screens ensure the game scales appropriately and remains legible on different screen sizes.
UR_FAMILY_FRIENDLY	The content itself is inoffensive and neutral by nature, aligning with BBFC PG rating, and is suitable for young children. This is reflected in the architectural implementation across the game classes/interfaces.
UR_UNIVERSITY_ACCURATE/NFR_UNIVERSITY_STUDENT_RELATABLE	There is encapsulation of game state and data of world in GameWorld class, as well as event logic in screen classes, enabling an accurate representation of university environments. Assets are all university themed e.g bookcases, vending machines, etc.. - see Objects in World.
UR_ASSESSMENT1_EVENTS/FR_EVENTS	There is event-driven handling with InputProcessor and the centralised game state/event management inside BaseGameScreen - both supporting event tracking and assessment features - code ensures that at least one type of each event (positive, negative, hidden) occurs.

UR_TIME_TRACKER/FR_TIME_TRACKER	We have the TimerManager class which centralises all timer states, limit enforcements, and HUD time rendering logic for every play session.
UR_MAXIMUM_TIME_LIMIT	TIME_LIMIT attribute set to 300 seconds (5 mins) in TimerManager class.
UR_EVENT_COUNTER/FR_EVENT_COUNTER	Implemented the EventsCounter class to effectively count and track the number of events that occur in a game session, while adjusting HUD accordingly.
UR_COLOURBLIND_FRIENDLY	The Hud class handles UI rendering supporting customisable colouring, ensuring high contrast and accessibility for users with colour blindness.
UR_MULTI_OPERATING_SYSTEM_COMPATIBLE	All classes, game logic, and UI rendering are built using LibGDX, which natively supports MacOS, Windows, and Linux.
UR_BEGINNER_DIFFICULTY	Architecture of the game is designed in such a way where 70% of responses of initial public game testing found the game balanced in regard to difficulty, satisfying the requirement.
UR_LEGIBLE_TEXT	The Hud class displays all on-screen text using scalable, high-contrast fonts using BitmapFont rendering - this ensures legibility across different resolutions and display sizes throughout the game session.
UR_FILE_SIZE	The Disposable interface ensures that resources, such as fonts, textures, and music, are only loaded when needed and are properly removed. This minimises memory leaks and reduces unnecessary memory usage, subsequently keeping the overall file size of the game down.

Bibliography

[1] libGDX, “The life cycle - libGDX,” 2025. [Online]. Available:
https://libgdx.com/wiki/app/the-life-cycle

[2] Wikipedia, “Entity Component System,” 2025. [Online]. Available:
https://en.wikipedia.org/wiki/Entity_component_system