Architecture

Team 10: Hard G For GIFs

Dragos Stoican

Rhys Milling

Samuel Plane

Quentin Rothman

Bowen Lyu

Jack Gerhard

Updated by The 8-Team:

Charlie Hayes

David Kayode

Matilda Garcia

Joshua Stafford

Ionut Manasia

Matthew Tomlinson

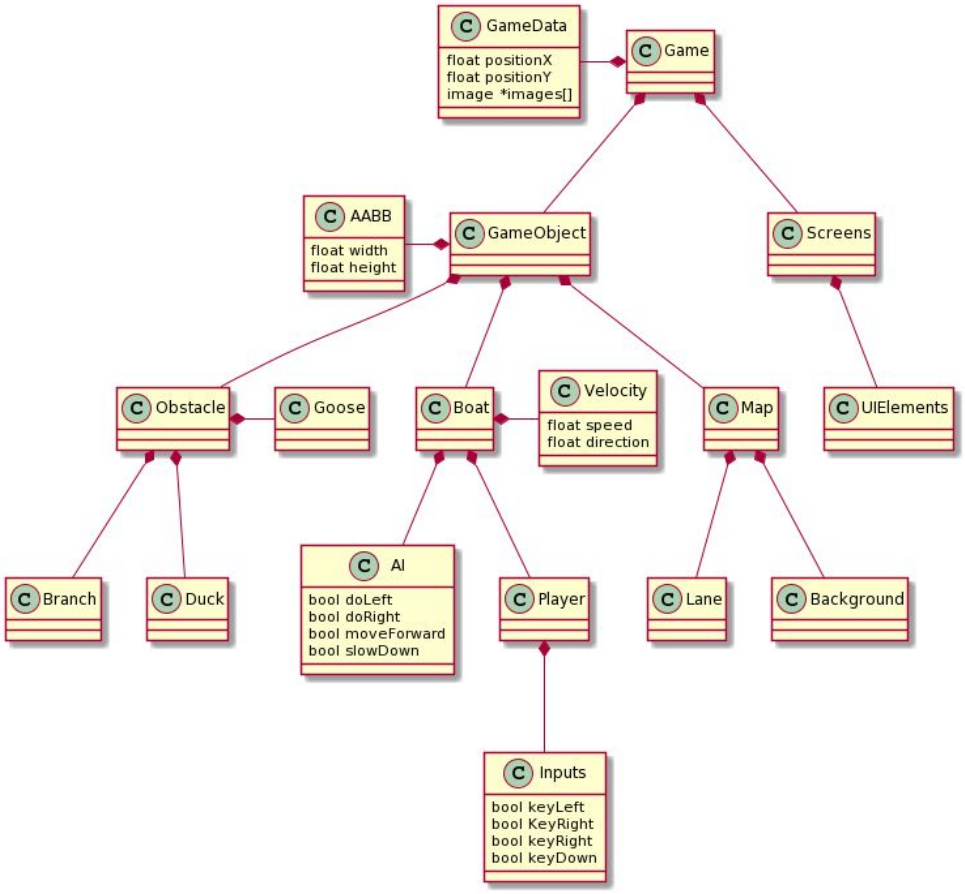
Changes are highlighted as such

Architecture (a)

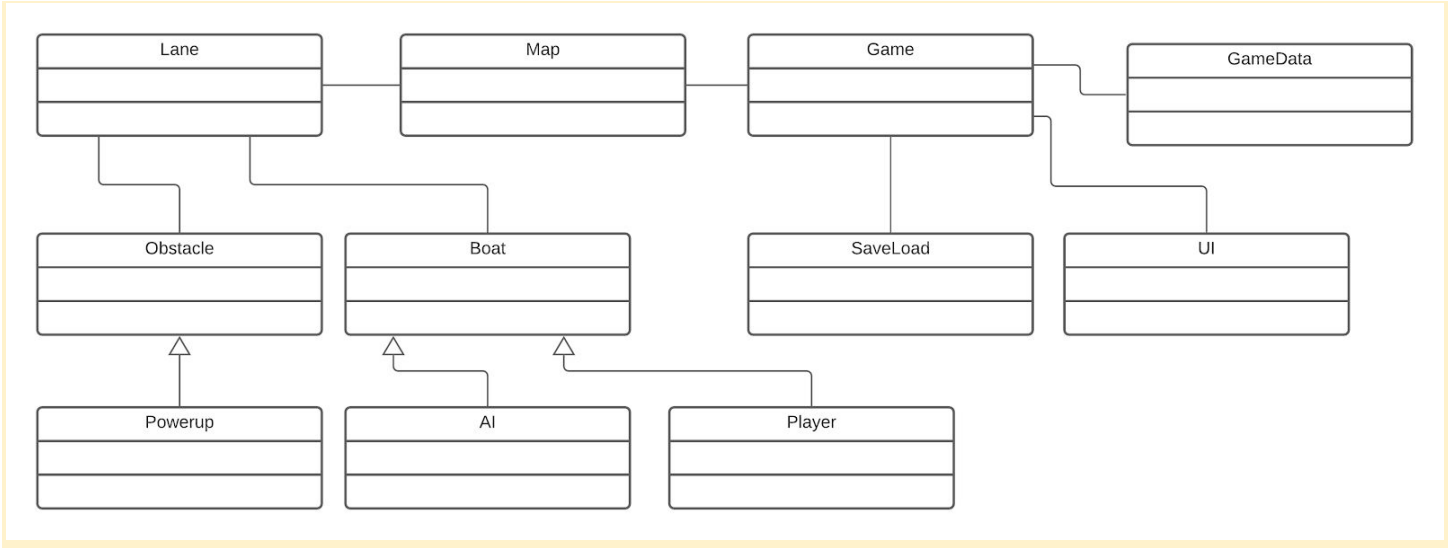
- We used Lucidchart to represent the two architectures.
- We created an entity component system and alternative class diagram for the Abstract architecture, allowing us to visualise the system in 2 distinct ways.
- We designed a UML Class Diagram for the Concrete architecture, using an OOP Entity Hierarchy with inheritance and polymorphism. UML was chosen as it allows consistent and readable diagrams that are standardised, so can be recognised by other developers and understand the system at a glance, enhancing maintainability.

Abstract Architecture

State diagram replaced with class diagrams as we felt they better visualise the structure of the system.



Inheritance is shown by the hollow arrow heads, where one class uses another a straight line is used.



Concrete Architecture

The concrete architecture was extended from the abstract architecture and involves implementation details such as framework specificities (Application Adapter) from the chosen framework, LibGDX. Adaptations were made to fully cover the requirements outlined:

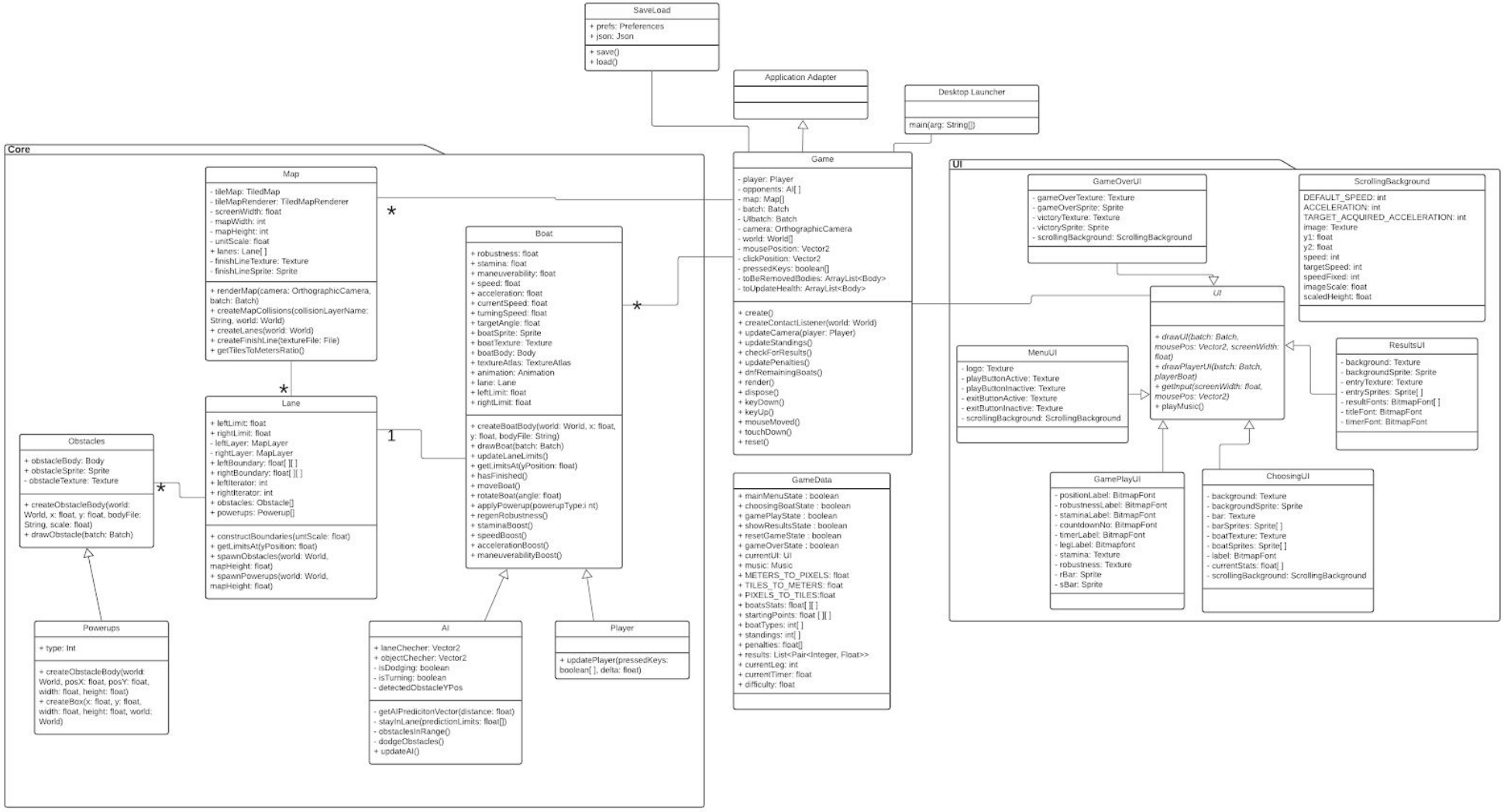
- **Desktop Launcher**, the settings to apply for the desktop distribution
- **Game,** the main game class for logic and containing the map instance and game structure related methods
- **GameData**, a class containing static attributes which track current game information
- **UI**, an abstract class from which the other UI classes are extended:
    - **GamePlayUI**, the UI elements displayed while the player is racing
    - **MenuUI**, the class for UI elements (e.g. buttons) for the main menu when the game first starts
    - **GameOverUI**, the class for the end game screens such as a victory or DNQ
    - **ResultsUI**, the class for UI elements pertaining to showing the results at the end of each leg
    - **ChoosingUI**, the class for the UI elements such as stat bars and selectable boat sprites
    - **ScrollingBackground**, a class that contains methods to generate a scrolling background for menus
- **Map**, holds methods and attributes for loading a tiled map and rendering it, as well as creating lanes which are contained within an array in Map
- **Lane**, a class containing methods to create ‘lanes’ in which the Player and AI race, also contains the methods for spawning obstacles and stores them in arrays
- **Boat**, a parent class which contains methods for constructing a boat and Box2D physics body and methods for updating a boat’s movement. Holds attributes relating to a boat’s performance e.g. stamina, acceleration etc.
    - **Player**, inherits the methods and attributes from the Boat class but adds an additional method for updating the boat’s movement by player input
    - **AI**, inherits the methods and attributes from Boat while adding methods relevant to enabling automated movement and obstacle evasion
- **Obstacle**, a class representing objects that can collide with the player using a method to construct a Box2D physics body

Each of these build on their abstract counterparts, with the UI class being extended into many child classes inheriting from the main UI class, containing relevant attributes and methods to form the complete system.

Additional classes methods were added when extending the architecture of the system to fulfill new requirements:

- **Powerup**, inherits the methods and attributes from the Obstacle class and adds the additional attribute type and methods to return the type, so different effects can be applied to Boat attributes
- **SaveLoad**, holds static methods to save and load the game
- **Boat.applyPowerup(int type)**, takes a powerup type and determines a below method to apply
    - **Boat.regenRobustness()**
    - **Boat.<>Boost()**
- **Lane.spawnPowerups()**, randomly spawns powerups on a lane

Game.reset(), resets the game or just the leg


SaveLoad
+ prefs: Preferences
+ json: Json
+ save()
+ load()
Application Adapter
Desktop Launcher
main(arg: String[])
Core
Map
- tileMap: TiledMap
- tileMapRenderer: TiledMapRenderer
- screenWidth: float
- mapWidth: int
- mapHeight: int
- unitScale: float
+ lanes: Lane[ ]
- finishLineTexture: Texture
- finishLineSprite: Sprite
+ renderMap(camera: OrthographicCamera, batch: Batch)
+ createMapCollisions(collisionLayerName: String, world: World)
+ createLanes(world: World)
+ createFinishLine(textureFile: File)
+ getTilesToMetersRatio()
Boat
+ robustness: float
+ stamina: float
+ maneuverability: float
+ speed: float
+ acceleration: float
+ currentSpeed: float
+ turningSpeed: float
+ targetAngle: float
+ boatSprite: Sprite
+ boatTexture: Texture
+ boatBody: Body
+ textureAtlas: TextureAtlas
+ animation: Animation
+ lane: Lane
+ leftLimit: float
+ rightLimit: float
+ createBoatBody(world: World, x: float, y: float, bodyFile: String)
+ drawBoat(batch: Batch)
+ updateLaneLimits()
+ getLimitsAt(yPosition: float)
+ hasFinished()
+ moveBoat()
+ rotateBoat(angle: float)
+ applyPowerup(powerupType:i nt)
+ regenRobustness()
+ staminaBoost()
+ speedBoost()
+ accelerationBoost()
+ maneuverabilityBoost()
Lane
+ leftLimit: float
+ rightLimit: float
- leftLayer: MapLayer
- rightLayer: MapLayer
+ leftBoundary: float[ ][ ]
+ rightBoundary: float[ ][ ]
+ leftIterator: int
+ rightIterator: int
+ obstacles: Obstacle[]
+ powerups: Powerup[]
+ constructBoundaries(untScale: float)
+ getLimitsAt(yPosition: float)
+ spawnObstacles(world: World, mapHeight: float)
+ spawnPowerups(world: World, mapHeight: float)
Obstacles
+ obstacleBody: Body
+ obstacleSprite: Sprite
- obstacleTexture: Texture
+ createObstacleBody(world: World, x: float, y: float, bodyFile: String, scale: float)
+ drawObstacle(batch: Batch)
Powerups
+ type: Int
+ createObstacleBody(world: World, posX: float, posY: float, width: float, height: float)
+ createBox(x: float, y: float, width: float, height: float, world: World)
AI
+ laneChecher: Vector2
+ objectChecher: Vector2
- isDodging: boolean
- isTurning: boolean
- detectedObstacleYPos
- getAIPredicitonVector(distance: float)
- stayInLane(predictionLimits: float[])
- obstaclesInRange()
- dodgeObstacles()
+ updateAI()
Player
+ updatePlayer(pressedKeys: boolean[ ], delta: float)
Game
- player: Player
- opponents: AI[ ]
- map: Map[]
- batch: Batch
- UIbatch: Batch
- camera: OrthographicCamera
- world: World[]
- mousePosition: Vector2
- clickPosition: Vector2
- pressedKeys: boolean[]
- toBeRemovedBodies: ArrayList<Body>
- toUpdateHealth: ArrayList<Body>
+ create()
+ createContactListener(world: World)
+ updateCamera(player: Player)
+ updateStandings()
+ checkForResults()
+ updatePenalties()
+ dnfRemainingBoats()
+ render()
+ dispose()
+ keyDown()
+ keyUp()
+ mouseMoved()
+ touchDown()
+ reset()
GameData
+ mainMenuState : boolean
+ choosingBoatState : boolean
+ gamePlayState : boolean
+ showResultsState : boolean
+ resetGameState : boolean
+ gameOverState : boolean
+ currentUI: UI
+ music: Music
+ METERS_TO_PIXELS: float
+ TILES_TO_METERS: float
+ PIXELS_TO_TILES:float
+ boatsStats: float[ ][ ]
+ startingPoints: float [ ][ ]
+ boatTypes: int[ ]
+ standings: int[ ]
+ penalties: float[]
+ results: List<Pair<Integer, Float>>
+ currentLeg: int
+ currentTimer: float
+ difficulty: float
UI
GameOverUI
- gameOverTexture: Texture
- gameOverSprite: Sprite
- victoryTexture: Texture
- victorySprite: Sprite
- scrollingBackground: ScrollingBackground
ScrollingBackground
DEFAULT_SPEED: int
ACCELERATION: int
TARGET_ACQUIRED_ACCELERATION: int
image: Texture
y1: float
y2: float
speed: int
targetSpeed: int
speedFixed: int
imageScale: float
scaledHeight: float
UI
+ drawUI(batch: Batch, mousePos: Vector2, screenWidth: float)
+ drawPlayerUI(batch: Batch, playerBoat)
+ getInput(screenWidth: float, mousePos: Vector2)
+ playMusic()
MenuUI
- logo: Texture
- playButtonActive: Texture
- playButtonInactive: Texture
- exitButtonActive: Texture
- exitButtonInactive: Texture
- scrollingBackground: ScrollingBackground
ResultsUI
- background: Texture
- backgroundSprite: Sprite
- entryTexture: Texture
- entrySprites: Sprite[ ]
- resultFonts: BitmapFont[ ]
- titleFont: BitmapFont
- timerFont: BitmapFont
GamePlayUI
- positionLabel: BitmapFont
- robustnessLabel: BitmapFont
- staminaLabel: BitmapFont
- countdownNo: BitmapFont
- timerLabel: BitmapFont
- legLabel: Bitmapfont
- stamina: Texture
- robustness: Texture
- rBar: Sprite
- sBar: Sprite
ChoosingUI
- background: Texture
- backgroundSprite: Sprite
- bar: Texture
- barSprites: Sprite[ ]
- boatTexture: Texture
- boatSprites: Sprite[ ]
- label: BitmapFont
- currentStats: float[ ]
- scrollingBackground: ScrollingBackground
*
*
*
*
1

Architecture (b)

- The abstract architecture represents the overall structure of the system rather than the exact implementation details, allowing a concrete architecture to be extended from it.
- The abstract architecture details the classes which will be necessary in the development of the game and also which classes will inherit from others.
- Using an abstract architecture allow the concrete architecture to be more easily developed through extending the abstract classes to include methods and adding implementation details (such as those enforced by the framework or libraries)

Concrete architecture:

- The concrete architecture adopts an inheritance-based style, focused on easy implementation of the abstract architecture.
- Our project is divided into 3 main parts, represented by the two packages we created, and the main game class. Additionally, we use a static class to store information about the game that is necessary for all the other sections, and a SaveLoad class for the game save functionality.

GameData class

- The static class, GameData stores the data needed for the game, including the current state of the game, the UI drawn on the screen, the difficulty of the game (UR_DIFFICULTY), hardcoded specs of each boat type, and the player's boat decision.
- On top of that, as we use Box2D to handle the game's physics, and Tiled to create the maps, all of which use a different unit of measurement, GameData stores the ratios between pixels, meters, and tiles as necessary.

Game class

- The main game class, called Game, handles the creation of the world physics and the maps, this is also where we update objects after collisions, the boat standings, and results. (UR/FR_DAMAGE, UR_LANE, FR_PENALTY)
- Here we decide what is rendered on the screen, based on the game state found in the GameData class, implementing movement through all the UI states.
- The main class also handles the player's input, passing those parameters to the other methods.

Core Package

- The Boat class stores all the specs, handles the movement, rotation, physical properties, and rendering of each particular boat. We also include here the ties of the boat with the lane it is located in. (UR/FR_CONTROLS, UR_CHOOSING_BOATS, FR_STAMINA, FR_STATS, FR_ASPECT, FR_VARIABLE_CONTROLS)
- All the race contestants are an instance of a boat, thus we can control both the player and the AI by re-using methods with different parameters. (UR/FR_CONTROLS)
- The Player and the AI class extend Boat and add to it a updatePlayer or updateAI method which call the inherited moveBoat and rotateBoat methods, which are necessary for the gameplay state of the game
- The Map object is used to create an instance of a map. This includes the necessary functions for creating physical objects in the world, the lane boundaries, and the finish line. (UR_MAP)
- Every map object has an array of Lanes which is created in the createLanes method. Every lane is filled with obstacles and its boundaries are created and made available for the boats to use. This allows for the detection of collisions with obstacles and passing outside the lane as required by the abstract architecture. (UR/FR_LANE, FR_OBSTACLES)
- When creating an obstacle, we randomise its type and create its body using the Obstacle class methods. (FR_OBSTACLES)
- The Powerup class inherits from the Obstacle class but contains an attribute 'type' that determines the colour of the sprite and what boost will be applied to a boat that collides with it (UR_POWERUPS).

SaveLoad class

- This class does not inherit from any other classes and contains utilities to save and load the game
- Specifically, the save() and load() methods, implementing LibGDX preferences
- The methods are called from the Game or MenuUI classes
- This class and methods are needed to fulfill the requirement UR_SAVE

UI package

- The abstract UI class is used as the declared type of a variable which will store the current UI to be displayed. It is instantiated with its subclasses so that each state of UI can be implemented in a separate class.
- The UI class has a getInput method to respond to user input. The method is declared here as several sub-classes will need to implement it, each in a different manner based on what game state they are displaying.
- UI declares two abstract methods, drawUI is responsible for drawing static components to the screen, such as the menu, whilst drawPlayerUI is responsible for player-related elements, such as the bar representing the player's stamina in a race.
- The playMusic method is called by all the inherited classes to play the current soundtrack from the GameData class.
- MenuUI and ChoosingUI display static elements that transition the user from the start of the game to the beginning of the gameplay, choosing a boat in the process and saving it in the GameData class (UR_MENU, UR_CHOOSING_BOATS)
- GamePlayUI is displayed during races, showing the player's remaining robustness and stamina, their current time in the leg, and their position in the race. As this UI is constantly changing, we use the drawPlayerUI method, allowing us to access the current stats of the player's boat. (UR_HUD, FR_HEALTHBAR)
- ResultsUI displays the results of each leg, showing each team's position and the time they took to complete the leg.
- GameOverUI is displayed when the player finishes their legs, showing a victory screen if the player won, or a game over screen otherwise.