3^3

Architecture

3^3 Studios | Team 27

James Hooper, Daniel O'Brien, Connor Davies, Joe Krystek-Walton, Mitchell Simpson, Yiyang Ye

3³
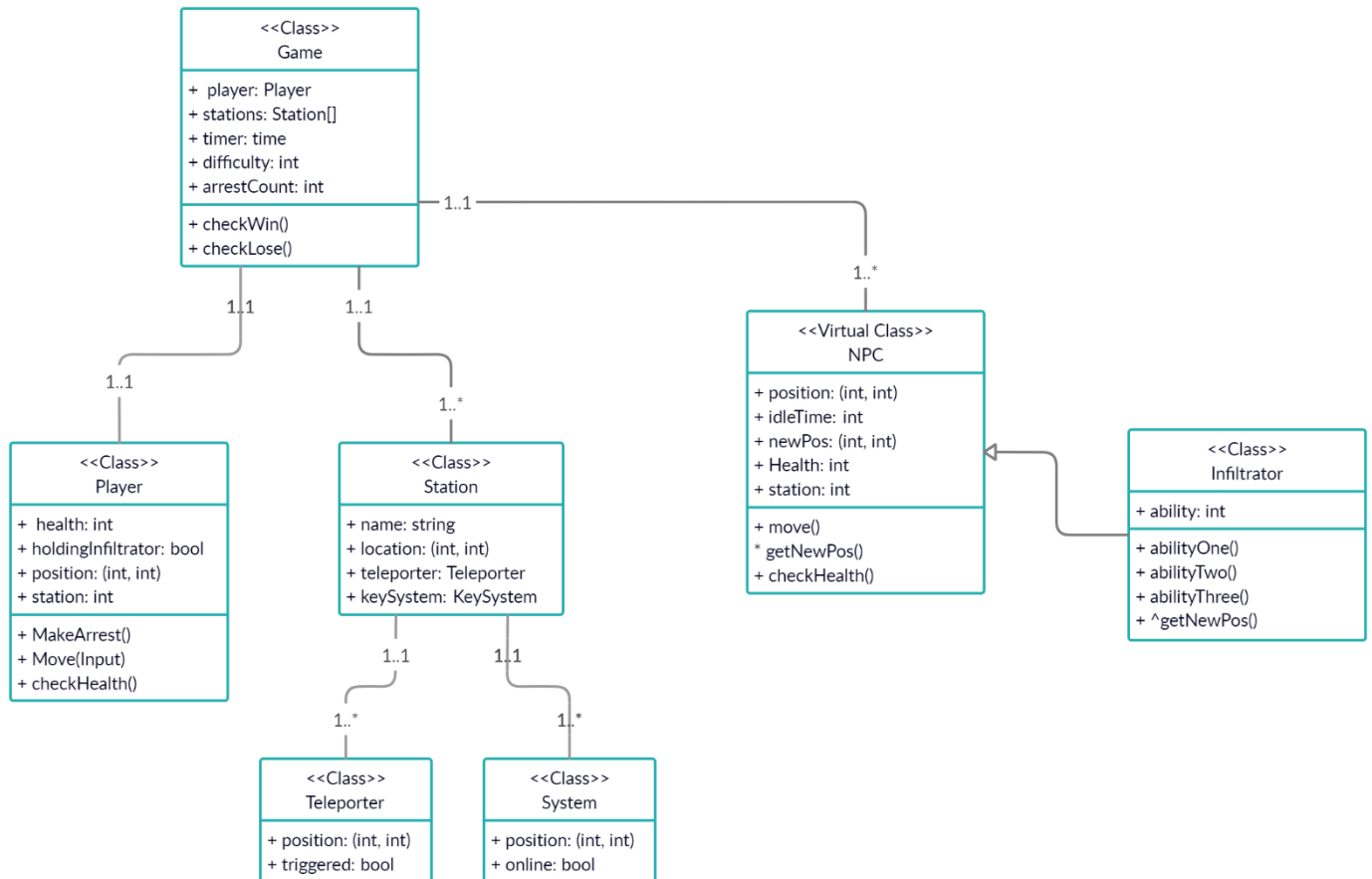
Architecture

Introduction

This document goes into detail about the architecture used for the implementation of the software, with justification as to why it was designed in this manner.
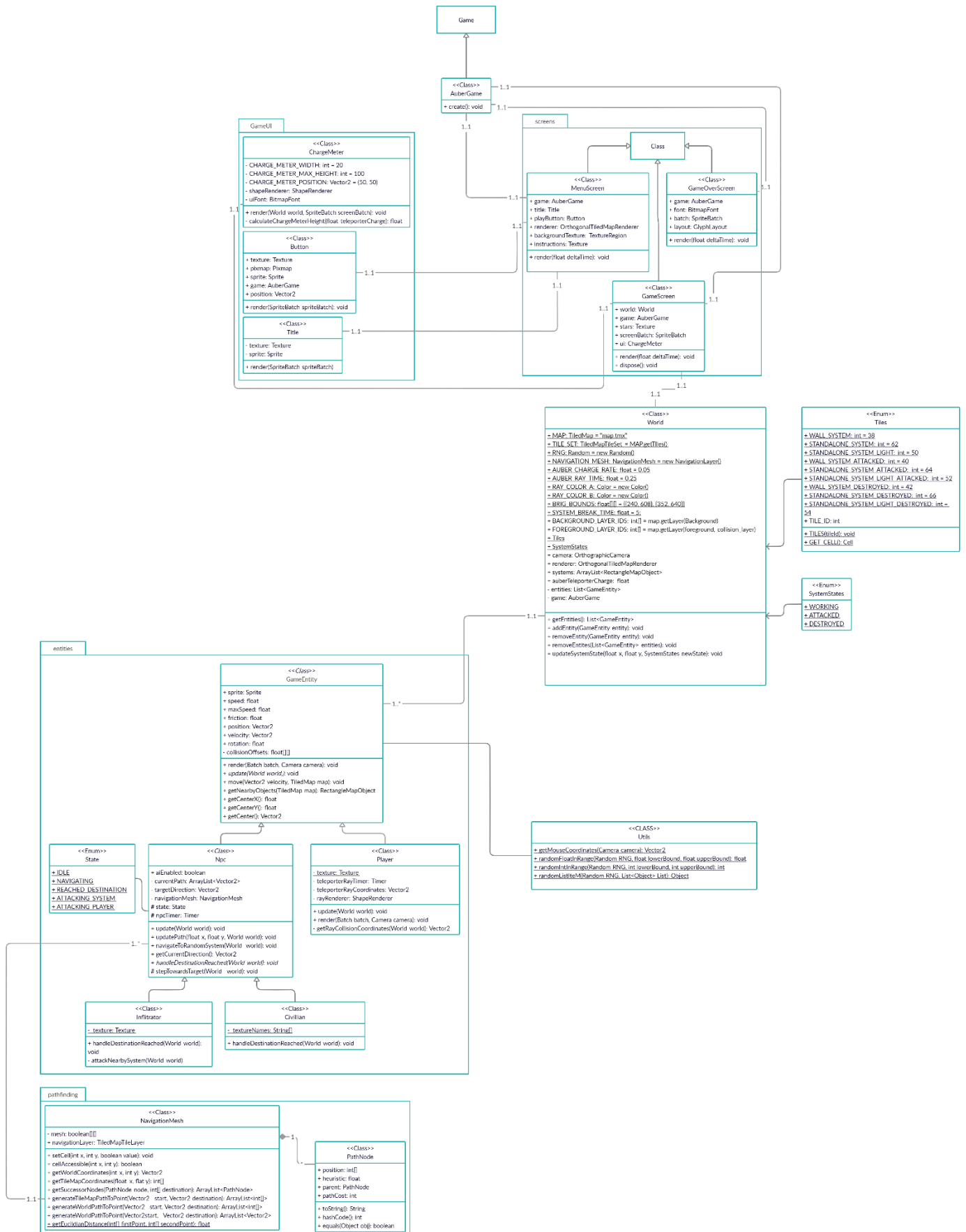
Generating our architecture plans

To generate our abstract and concrete architecture, we decided to utilise the Unified Modelling Language (UML). This is a standardised language consisting of diagrams which we will use to help ourselves when it comes to the implementation stage of our project. UML is well known, well documented and takes advantage of many best engineering practices, so we can be sure that the implementation process will be as seamless as possible.

We created class diagrams to plan what classes would be needed, the attributes of each class and the operations it can perform, and the interactions required between those classes. We chose this method because of its flexibility and readability. As forementioned, it is also the most ubiquitous design language used by software developers so should be readily understood by our team. To create these diagrams, we used the websites creately.com and lucidchart.com due to their ease of use and accessibility.

Abstract Architecture

Concrete Architecture

Game

<<Class>>
AuberGame
+ create(): void

<<Class>>
ChargeMeter
- CHARGE_METER_WIDTH: int = 20
- CHARGE_METER_MAX_HEIGHT: int = 100
- CHARGE_METER_POSITION: Vector2 = (50, 50)
- shapeRenderer: ShapeRenderer
- uiFont: BitmapFont
+ render(World world, SpriteBatch screenBatch): void
- calculateChargeMeterHeight(float teleporterCharge): float

GameUI

Class

<<Class>>
MenuScreen
+ game: AuberGame
+ title: Title
+ playButton: Button
+ renderer: OrthogonalTiledMapRenderer
+ backgroundTexture: TextureRegion
+ instructions: Texture
+ render(float deltaTime): void

<<Class>>
GameOverScreen
+ game: AuberGame
+ font: BitmapFont
+ batch: SpriteBatch
+ layout: GlyphLayout
+ render(float deltaTime): void

screens

<<Class>>
Button
+ texture: Texture
+ pixmap: Pixmap
+ sprite: Sprite
+ game: AuberGame
+ position: Vector2
+ render(SpriteBatch spriteBatch): void

<<Class>>
Title
- texture: Texture
- sprite: Sprite
+ render(SpriteBatch spriteBatch): void

<<Class>>
GameScreen
+ world: World
+ game: AuberGame
+ stars: Texture
+ screenBatch: SpriteBatch
+ ui: ChargeMeter
- render(float deltaTime): void
- dispose(): void

<<Class>>
World
+ MAP: TiledMap = "map.tmx"
+ TILE_SET: TiledMapTileSet = MAP.getTiles()
+ RNG: Random = new Random()
+ NAVIGATION_MESH: NavigationMesh = new NavigationLayer()
+ AUBER_CHARGE_RATE: float = 0.05
+ AUBER_RAY_TIME: float = 0.25
+ RAY_COLOR_A: Color = new Color()
+ RAY_COLOR_B: Color = new Color()
+ BRIG_BOUNDS: float[][] = {{240, 608}, {352, 640}}
+ SYSTEM_BREAK_TIME: float = 5;
- BACKGROUND_LAYER_IDS: int[] = map.getLayer(Background)
- FOREGROUND_LAYER_IDS: int[] = map.getLayer(foreground, collision_layer)
+ Tiles
+ SystemStates
- camera: OrthographicCamera
- renderer: OrthogonalTiledMapRenderer
+ systems: ArrayList<RectangleMapObject>
+ auberTeleporterCharge: float
- entities: List<GameEntity>
- game: AuberGame
+ getEntities(): List<GameEntity>
+ addEntity(GameEntity entity): void
+ removeEntity(GameEntity entity): void
+ removeEntities(List<GameEntity> entities): void
+ updateSystemState(float x, float y, SystemStates newState): void

<<Enum>>
Tiles
+ WALL_SYSTEM: int = 38
+ STANDALONE_SYSTEM: int = 62
+ STANDALONE_SYSTEM_LIGHT: int = 50
+ WALL_SYSTEM_ATTACKED: int = 40
+ STANDALONE_SYSTEM_ATTACKED: int = 64
+ STANDALONE_SYSTEM_LIGHT_ATTACKED: int = 52
+ WALL_SYSTEM_DESTROYED: int = 42
+ STANDALONE_SYSTEM_DESTROYED: int = 66
+ STANDALONE_SYSTEM_LIGHT_DESTROYED: int = 54
+ TILE_ID: int
+ TILES(field): void
+ GET_CELL(): Cell

<<Enum>>
SystemStates
+ WORKING
+ ATTACKED
+ DESTROYED

entities

<<Class>>
GameEntity
+ sprite: Sprite
+ speed: float
+ maxSpeed: float
+ friction: float
+ position: Vector2
+ velocity: Vector2
+ rotation: float
- collisionOffsets: float[][]
+ render(Batch batch, Camera camera): void
+ update(World world,): void
+ move(Vector2 velocity, TiledMap map): void
+ getNearbyObjects(TiledMap map): RectangleMapObject
+ getCenterX(): float
+ getCenterY(): float
+ getCenter(): Vector2

<<Enum>>
State
+ IDLE
+ NAVIGATING
+ REACHED_DESTINATION
+ ATTACKING_SYSTEM
+ ATTACKING_PLAYER

<<Class>>
Npc
+ aiEnabled: boolean
- currentPath: ArrayList<Vector2>
- targetDirection: Vector2
- navigationMesh: NavigationMesh
state: State
npcTimer: Timer
+ update(World world): void
+ updatePath(float x, float y, World world): void
+ navigateToRandomSystem(World world): void
+ getCurrentDirection(): Vector2
+ handleDestinationReached(World world): void
# stepTowardsTarget(World world): void

<<Class>>
Player
- texture: Texture
- teleporterRayTimer: Timer
- teleporterRayCoordinates: Vector2
- rayRenderer: ShapeRenderer
+ update(World world): void
+ render(Batch batch, Camera camera): void
- getRayCollisionCoordinates(World world): Vector2

<<CLASS>>
Utils
+ getMouseCoordinates(Camera camera): Vector2
+ randomFloatInRange(Random RNG, float lowerBound, float upperBound): float
+ randomIntInRange(Random RNG, int lowerBound, int upperBound): int
+ randomListItem(Random RNG, List<Object> List): Object

<<Class>>
Infiltrator
- texture: Texture
+ handleDestinationReached(World world): void
- attackNearbySystem(World world)

<<Class>>
Civilian
- textureNames: String[]
+ handleDestinationReached(World world): void

pathfinding

<<Class>>
NavigationMesh
- mesh: boolean[][]
- navigationLayer: TiledMapTileLayer
+ setCell(int x, int y, boolean value): void
- cellAccessible(int x, int y): boolean
- getWorldCoordinates(int x, int y): Vector2
- getTileMapCoordinates(float x, flat y): int[]
- getSuccessorNodes(PathNode node, int[] destination): ArrayList<PathNode>
+ generateTileMapPathToPoint(Vector2 start, Vector2 destination): ArrayList<int[]>
+ generateWorldPathToPoint(Vector2 start, Vector2 destination): ArrayList<int[]>
+ generateWorldPathToPoint(Vector2 start, Vector2 destination): ArrayList<Vector2>
+ getEuclidianDistance(int[] firstPoint, int[] secondPoint): float

<<Class>>
PathNode
+ position: int[]
+ heuristic: float
+ parent: PathNode
+ pathCost: int
+ toString(): String
+ hashCode(): int
+ equals(Object obj): boolean

The abstract architecture pictured above is intended as a prescriptive tool to guide the implementation of the project. It gives an overview of what must be implemented without giving strict rules about how it must be implemented, to allow for flexibility in development.

The classes and methods illustrated in our abstract architecture reflect the requirements we have elicited from our brief and customer interviews. Design decisions such as infiltrators' special abilities are represented in the diagram, but without the decisions about the implementations of those abilities.

The diagram contains a low level of detail for a few different reasons. It allows us to capture only the most critical information at this early design stage to ensure we have the ideas for the game's core features. This allows for later expansion and additional features after the minimum viable product has been delivered. It also allows for flexibility in developing the concrete architecture if the implementation must be changed due to limitations of the engine or evolving requirements. Our abstract architecture does not contain any language or engine specific methods so that it can be adapted to the tools and methods we decide to use for the project, for example if these were to change in development.

Extrapolating from the abstract design, the developed concrete architecture is pictured above. While the abstract architecture outlines the initial thoughts regarding the implementation of the software, it is limited in its scope – it only has 7 classes. During development of the program, we add more classes and classify them into 5 main categories, GameUI, Screen, World, Infiltrators, and Pathfinding. We can find, for example, during development that we delete the Station class and change it to Pathfinding and Screen classes. NPC and Player class are all included in the Entities class. Additionally, we add the World class which include the basic information regarding the world. These are some of the main differences between the two drafts of the software architecture.

Below will be represented many of the main changes from the development of the abstract architecture into the concrete representation, and how they correlate directly with requirements outlined in our project's requirements document for traceability.

GameUI

GameUI includes 3 classes. Its functions are responsible for features such as returning the UI elements for rendering, such as buttons and title text.

Screens

Screens includes 3 classes.
- MenuScreen is responsible for showing what is on the menu screen (the page the player is shown upon first load of the game).
- GameOverScreen is displayed when the player wins or loses the game.

Requirement **FR_MENU** is achieved here, alongside **FR_WIN_CONDITION** and **FR_LOSS_CONDITION**.

World

The World class stores information about the game world, which is used in other aspects of the software such as the GameScreen page. General information such as the navigation mesh and charge rate for the Auber's weapon is determined here.

This class is designed to fulfil **FR_MAP** and **FR_MENU**.

Entities

Entities includes many classes. It is responsible for holding information about the entities in the game.

- GameEntity contains all the information regarding the entities in a generalised way. Information such as speed, friction and position are stored here, to be obtained by classes implementing from this master class. This fulfils the **FR_HOSTILES** and **FR_SABOTAGE** requirements.
- The NPC class contains two subclasses:
    - The Infiltrator class represents the hostile infiltrators which need to be defeated by the player and attack the key systems in the station.
    - The Civilian class represents the non-hostile aliens on-board the station.

  These two classes fulfil the requirements **FR_ALIENS_COUNT, FR_ALIENS**, and **FR_HOSTILES_SPECIAL.**
- The Player class represents the entity controlled by the user.

  This class fulfils the requirements **FR_ATTACK_NOTIF, FR_HEAL, FR_ARREST**.

Pathfinding

Pathfinding includes two classes and is responsible for the movement of entities around the map.

- NavigationMesh is responsible for the map section of the game - it gives the entities a reminder about the accessible path around the map, allowing them to navigate between key locations in the most efficient way possible. This fulfils the requirements **FR_TELEPORTER** and **FR_MAP.**
- PathNode is a class designed to aid in the navigation of entities around the map using heuristics and path costs. This will fulfil the requirement **FR_PLAYER_TILES**.