Gold n' Warlocks

Linda Gratzer
HCC, FH Hagenberg

ABSTRACT

This paper describes the design and implementation of an interactive AR tower defense game for Android phones and PCs.

1 INTRODUCTION

"Gold n' Warlocks" is a tower defence game implemented in Unity (2019.4.21f1) using Vuforia. The player can interactively place towers anywhere to protect a treasure chest from enemy attacks. This project should demonstrate how markers can be used to create an interactive AR game for multiple devices, mainly desktop PCs and Android smartphones.

2 GAME DESIGN

Tower defence games are a popular video game genre where the player has to defend an objective, like his house in the game "lants vs. Zombies" [1], against (multiple) attacks of enemy waves. As a means of defence different kind of structures, like towers or traps, weapons or obstacles can be employed to destroy the attackers. Since the placement of the defenders and the timing when they are deployed is a crucial part of the game play, tower defence games also include strategic gameplay aspects. [2]

2.1 Gameplay

In "Gold n' Warlocks" the player has to defend a chest from a total of four enemy waves trying to steal it. He can freely place three different towers, each having a different firing speed and damage output, between the spawn of the enemies and the treasure chest. However, each tower can only attack a single enemy at a time. If an enemy reaches the chest, the player loses a life. The game is over when either the player has no more lives left or all enemy waves have been successfully cleared.

Before the game can start, the markers for the spawn and the chest have to be placed by the player, and they have to be properly tracked by the game. A random path between the start and end is generated each time the player starts a new game to offer a more diverse gaming experience.

Enemy Types There are a total of four enemy types (see Fig. 1) differing in movement speed and maximum health:

- Lich: Average number health points and speed.
- Slimes: Fastest and squishiest enemy type.
- Spiny Shell: Average movement speed with a slightly higher health point count.
- Mimic Chest: Slowest enemy type with the most health points.

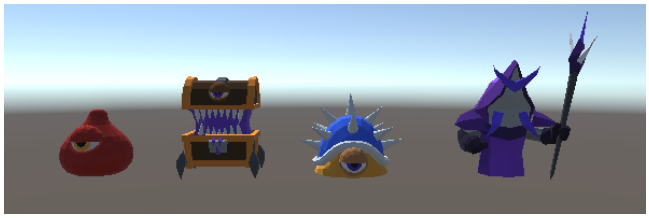

Figure 1: Enemy types from left to right: Slime, Mimic Chest, Spiky Shell, Lich.

Towers The towers (see Fig. 2) which are used to defend the treasure chest also have different properties regarding their shooting speed and damage output:

- Tower 1: Average shooting speed and regular damage.
- Tower 2: High shooting frequency but little damage.
- Tower 3: Low frequency with a high damage.

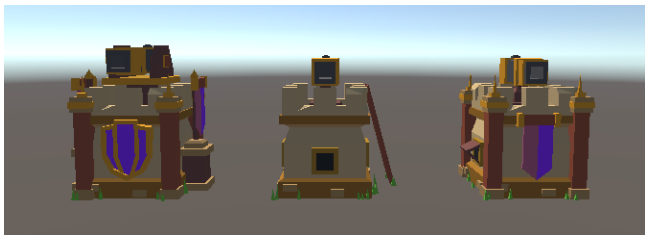

Figure 2: Tower types from left to right: Tower 1, Tower 2, Tower 3.

Stages There are a total of four stages the player has to win. In each stage, a different combination of enemy types and the maximum count per type is defined:

- Stage 1: Lich (5x), Mimic Chest (3x)
- Stage 2: Lich (5x), Spiky Shell (6x)
- Stage 3: Lich (5x), Slime (8x)
- Stage 4: Lich (5x), Mimic Chest (3x), Spiky Shell (6x), Slime (8x)

3 IMPLEMENTATION

This project was implemented using Unity version 2019.4.21f1 and Vuforia for marker tracking. Assets were imported from the Unity asset store and a complete list of used resources can be found in the README.

In this section the system architecture is layed out and supporting diagrams show how the classes and services interact with each other.

3.1 Services

The following services are used by multiple components throughout the whole application.

ServiceManager This class is the only singleton in the application and is responsible for managing references to other services used in the game. It acts as a simple DI-container and offers methods to register, unregister and get services by their type.

GameStateService The GameStateService is a service which manages the current and previous GameState as well as the current stage number. It also exposes an event which is invoked whenever the current state of the game changes.

PathGenerator The PathGenerator service contains the logic to generate a random path between a given start and end point and it triggers an event whenever the current path has changed.

ReferencablesContainer This container manages all ReferencableComponents in the application. Instead of using the static method GameObject.Find() components which are to be used in other components should use the ReferencableComponent. This way the references are added only once and can simply be accessed through the container, instead of calling the Find method over and over, which can become very inefficient.

3.2 UI

Every element displayed in the UI has a corresponding UiElementComponent. This component utilises a CanvasGroup to change the visibility of the GameObjects. The interaction between the UI elements and the main application is illustrated in figure Fig. 3.

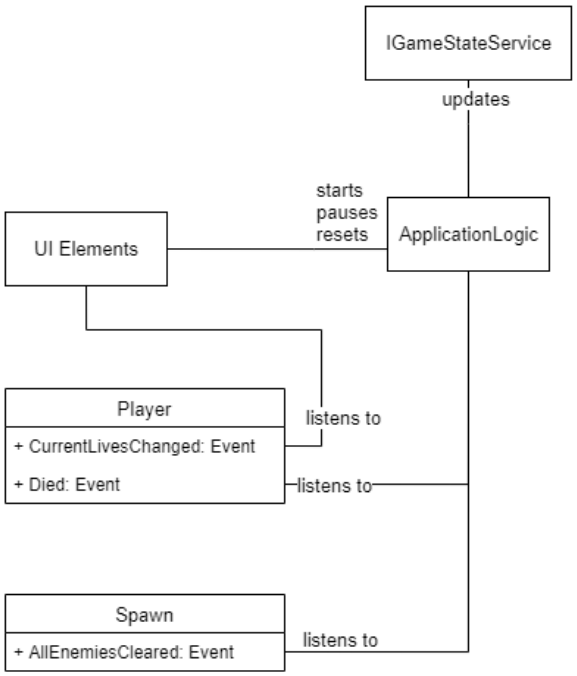

Figure 3: Interaction between the main application logic and the other services and components.

UiElement This class serves as the abstract base class for all UI element handlers. It declares a property to get all GameState when it is visible in the UI and a method to update its visibility.

UiVisibilityManager The UiVisibilityManagerComponent listens to changes in the current GameState and updates the visibility of all UiElements accordingly. It retrieves the references to these elements via the ReferencablesContainer.

3.3 Application Startup

In order to set up a simple dependency-injection (DI) system, a bootstrapping class was implemented. The bootstrapping process involves two methods: one is called before and one after the scene is loaded. Before the scene is initially loaded by the runtime, the services managed by the ServiceManager are instantiated and registered in the DI container (see Fig. 4). After the scene was loaded and after awake was called in all MonoBehaviour objects, the bootstrapping processes finishes the initialization of the necessary services (see Fig. 5).

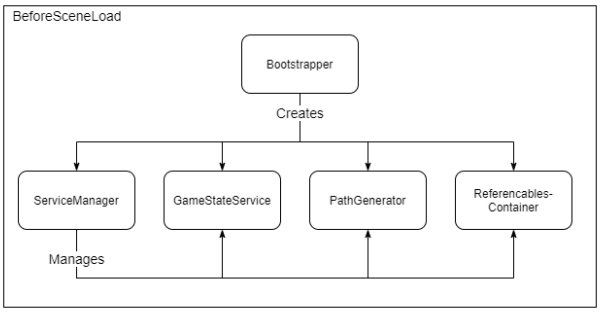

Figure 4: Bootstrapping process before the scene is loaded.

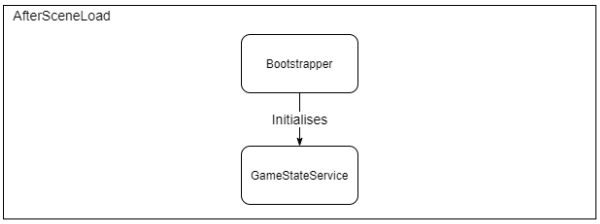

Figure 5: Bootstrapping process after the scene is loaded.

3.4 GameState Lifecycle

To avoid hard-coded dependencies there is no direct communication between the main application logic and the other services and components, like the Player, EnemySpawner or Towers. The game lifecycle is controlled via the GameState, which is only changed by the main application component. Other components and services listen to changes in the current GameState and handle these changes accordingly. Figure Fig. 6 shows the life cycle of the different GameState values. The GameState can take following states:

- **Unknown**: Initial value in the GameStateService.
- **Initialized**: Set after the service has been initialized.
- **Ready**: Everything has been initialized and the markers are being tracked correctly.
- **Missing Markers**: At least one required marker is not being tracked.
- **Running**: The game is currently running.
- **Paused**: The game is currently paused.
- **Stage Cleared**: The current stage has been cleared.

- **Game Over**: The player lost all lives before all stages were cleared.
- **Won**: The player cleared all stages.

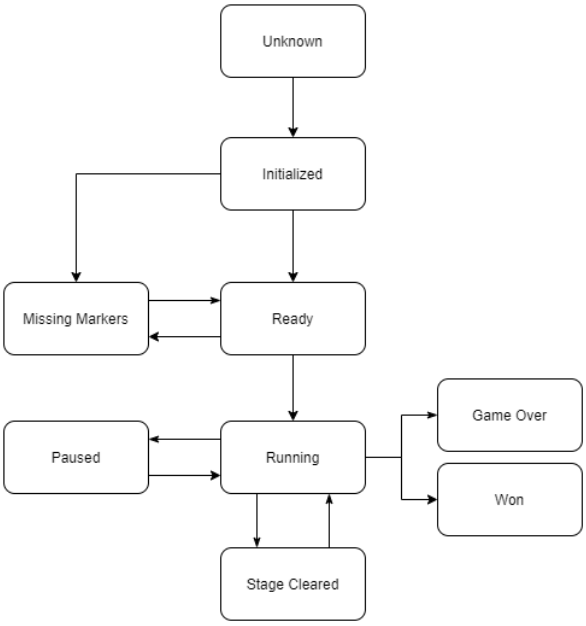

Figure 6: The life cycle of the GameState.

3.5 Path Generation

To create a more varied game experience a new path is randomly generated between each round. Before the game is started the ApplicationComponent uses the PathGenerator to generate a new path. Based on this path it instantiates waypoints and stores them.
The path generation logic is based on two algorithms:

- Random path with end target
- Generate a random path between two predefined points

The path generation can be briefly summarised as follows: While the end is not reached a new point between the previously generated point and the end is determined using linear interpolation. The values of the new point are modified using scaling factors and a random angle between 40° and 120°.

3.6 Spawn System

The EnemySpawner is responsible for spawning and managing references to all enemies of the current stage. What types of enemies and how many of each type can spawn in what stages can be configured in the inspector GUI of the component. While the game is running a new enemy is instantiated until the maximum number for this enemy type is reached. It also registers an event handler for the Killed event of the Enemy component to correctly update the references and check whether all enemies have been killed. When all enemies have been killed the corresponding event is invoked.

3.7 Defender Enemy Tracking

A Defender uses an EnemyFinder to get the next enemy as a target. The EnemyFinder requires a Collider component and it is set to be a trigger. Whenever an enemy enters it, the EnemyFinder adds the enemy to a list and removes it after it exited the trigger. While the defender's target is available it instantiates new projectiles, otherwise it gets the next enemy as target from the EnemyFinder.

4 User Guide

To play the game, the player first has to place the markers for the spawn and the end on a surface. Before the game can be started, the spawn and end markers must be tracked properly to generate a new path between those two points. To defend the treasure chest the player can place up to three different towers and move them around freely. The game is won, if the player successfully cleared all four stages. If he loses all lives, before all stages have been cleared, the game is lost. While the game is running, the player can also pause the game and resume it again. He also has the option to forfeit the current game in the pause screen.

5 Conclusion

A simple and interactive AR tower defence game was contrived and implemented using Unity and Vuforia. Unity offers an easy way to create AR appliation for multiple platforms. The concept of tower defence games, the game design and application implementation were described, including diagrams to show the interaction between the various application components.

References

[1] Wikipedia. Plants vs. zombies (video game), 2021. [Online; accessed 21-March-2021].
[2] Wikipedia. Tower defense, 2021. [Online; accessed 21-March-2021].