**Goblin Siege**

Technical Design Document

Shahil Saha

Table of Contents

[**Project Introduction** 2](#_Toc160122469)

[Project Introduction 2](#_Toc160122470)

[Project Goals 2](#_Toc160122471)

[Challenges and Risks 2](#_Toc160122472)

[**Platforms** 2](#_Toc160122473)

[System Requirements 2](#_Toc160122474)

[Engine Summary 3](#_Toc160122475)

[Core game pillars: 3](#_Toc160122476)

[**Systems and Diagrams** 4](#_Toc160122477)

[Core Gameplay Loop 4](#_Toc160122478)

[Core Mechanics 5](#_Toc160122479)

[Player Controls 5](#_Toc160122480)

[Building Placement 5](#_Toc160122481)

[Gameplay Mechanics 6](#_Toc160122482)

[Currency Management 6](#_Toc160122483)

[Building Upgrades 7](#_Toc160122484)

[Enemy AI 7](#_Toc160122485)

[Waves 10](#_Toc160122486)

[Tower Operation 10](#_Toc160122487)

[Art Workflows 13](#_Toc160122488)

[Materials and Textures 13](#_Toc160122489)

[Lighting 13](#_Toc160122490)

[Collision 13](#_Toc160122491)

[Wireframes 14](#_Toc160122492)

[Fonts 17](#_Toc160122493)

[Audio 17](#_Toc160122494)

[Assets 17](#_Toc160122495)

[**Optimisation and Profiling** 17](#_Toc160122496)

[Profiling Systems 17](#_Toc160122497)

[**Coding Standards** 19](#_Toc160122498)

[Programming Standards 19](#_Toc160122499)

[**Production Overview** 22](#_Toc160122500)

[Moscow 22](#_Toc160122501)

[Timeline 22](#_Toc160122502)

# **Project Introduction**

## Project Introduction

"Goblin Siege" is a single-player tower defence game built for PC. It features a top-down view and is developed using the Unreal Engine. The game draws inspiration from several well-known tower Défense games such as Bloon Tower Defence, Plants vs Zombies, and Dungeon Defenders.

## Project Goals

The main goal of this game is to protect coins from the goblins. You must use turrets to stop them and keep the gold safe. The strategy in the game involves buying different turrets and upgrading them to defeat the goblins.

## Challenges and Risks

This is a very new genre of game I have tried to make this time and this is a thrilling experience in itself. There are a lot of different mechanics that will be implemented in this project and the major risk is over-scoping. It can be a challenging task to complete all the mechanics before the deadlines, but it is not impossible to achieve them. This is due to having some prior knowledge and experience with the engine.

# **Platforms**

## System Requirements

Minimum Requirements

* Platform: PC
* CPU: i5
* Ram: 8gb
* Hard disk space: 20gb
* Graphics Card: GTX 1080 or similar
* OS: Windows 10

Recommended Requirements

* Platform: PC
* CPU: i9
* Ram: 16gb
* Hard disk space: 20gb
* Graphics Card: RTX 3080
* OS: Windows 11

## Engine Summary

Engine: Unreal Engine 5.2.1

Plugins:

* Rider Plugin

## Core game pillars:

The game draws inspiration from titles such as Bloon Tower Defence and Dungeon Defenders, promising a comparable gaming experience. One of its foundational elements lies in its Strategic Depth, where players must tactically position towers on the map to thwart goblins' attempts to reach the gold.

Another key aspect is the Progressive Challenge, a ubiquitous feature in strategy games, particularly those with wave mechanics. As players advance through waves, defending against enemies becomes increasingly difficult, necessitating continuous tower upgrades and construction to repel them from reaching their objective.

However, to maintain player engagement and create a compelling experience, there must be a barrier. This leads to the final pillar of the game: Resource Management. Without this element, players could place an unlimited number of buildings on the map, resulting in a boring gameplay experience.

Hence, resource management is crucial for adding depth and enjoyment to the game.A diagram of a structure

Description automatically generated

# **Systems and Diagrams**

## Core Gameplay Loop

The core gameplay loop in this game revolves around real-time strategic decisions, such as purchasing and upgrading various turrets to effectively counter different enemy types and prevent them from looting the gold. A key aspect of the game is the strategic management of resources to acquire and enhance the more powerful turrets.

A diagram of a game

Description automatically generated

## Core Mechanics

Player Controls

|  |  |
| --- | --- |
| Camera Controls |  |
| Move | The movements on the camera are done with WASD keys on the keyboard. |
| Look | The look is enabled by pressing the RMB and disables when the button is released. For this project, only the Yaw has been enabled. This is because after observing a lot of top down games like Bloon tower defence or most of the top down games I have come across have their pitch and roll fixed. This is basically to prevent the players from being lost and having difficulties getting back to the normal rotation. |
| Zoom | The player can use the mouse wheel to zoom in and out. |
| Interact | Players may choose to upgrade the towers that are already placed and settled on the map. In order to do so, the player needs to click on the Tower to select it. |
| Pause | The player can also pause the game whenever they want using the P key on the keyboard. |

Building Placement

As previously discussed, strategic depth forms a fundamental aspect of our game design. It's pivotal for players to strategically position buildings across the map to effectively stop goblin advances. From the player's perspective, decisions revolve around where to position a tower and what type of tower best suits each location based on its unique attributes.

However, specific conditions govern the placement of the towers:

1. Towers cannot obstruct the enemy's designated path. Any overlap with the enemy route will prevent tower placement, ensuring players must strategize effectively.

2. Buildings cannot be stacked atop one another. The game actively checks for overlaps with existing buildings, disallowing placement if detected.

The viability of a placement is indicated by a decal material attached to all buildings. Green denotes permissible placement, while red signifies prohibited areas. This visual cue empowers players to make informed decisions regarding their defensive layout, enhancing the strategic depth of the gameplay experience.

If the Building button is disabled, it's either due to insufficient balance or the need for more funds to upgrade the selected building. The button will be disabled when the player's balance is low, indicated by a red material overlay. This visual cue serves as a clear indicator that additional funds are required to proceed with the desired action.

## Gameplay Mechanics

Currency Management

In "Goblin Siege," acquiring and strategically deploying towers is essential to safeguarding the treasures from enemy looting. Players begin with a set amount of in-game currency, enabling them to purchase towers within their budget. However, as the game progresses, they must confront waves of enemies to accumulate additional funds for acquiring new towers or upgrading existing ones.

The underlying logic for currency acquisition and management is straightforward:

1. Initialization: At the onset of the game, the player controller establishes a connection with the EnemyManager by binding a delegate to the OnEnemyKilled() method within the script.
2. Enemy Elimination: Whenever an enemy is defeated, the EnemyManager triggers the delegate, which in turn invokes the designated method within the Input Controller script.
3. Balance Update: As both the Player HUD and player character functionalities are overseen by the Player Controller, it assumes responsibility for updating the currency balance. Consequently, the Player Controller ensures that the balance is reflected accurately on both the HUD interface and the player character.

A diagram of a computer game

Description automatically generated

This systematic approach ensures that players are continually incentivized to engage with the game's challenges, as successful enemy eliminations directly contribute to their financial resources, facilitating further strategic decision-making and progression within the game.

Building Upgrades

Upgradeable buildings are a cornerstone of our gameplay design, offering players unique features and firing styles for each structure. Players have the option to upgrade buildings on the map by interacting with them. When the player interacts with the building, the player HUD on the right side of the screen is replaced by the Upgrade UI, featuring the Upgrade button.

The Upgrade button may be disabled for two primary reasons:

The selected building has already reached its maximum upgrade level. In this case, further upgrades are unavailable, and the button is disabled to reflect this status.

If the maximum upgrade level has not been reached, the button will be disabled due to insufficient funds in the player's balance. This signals that additional funds are required to proceed with the upgrade.

A white paper with black squares and lines

Description automatically generated

Enemy AI

The enemy AI mechanic you've described for your game, where enemies aim to surround the treasure before looting it, indeed adds a layer of complexity and strategy to the gameplay. Integrating the Environment Query System (EQS) for this purpose not only enhances the AI's decision-making capabilities but also introduces dynamic challenges for players. For optimization purposes, object pooling has been used in order to spawn a desired number of controllers at the very starting of the game.

Additionally, two containers (an array and a queue) are being utilised to manage the allocation and deallocation of the controllers. This is because as a future development process, I plan to optimize the game further by reducing the number of controllers spawn at the starting of the game. And if the enemy count exceeds the number of controllers, than the enemies would spawn once the controllers are free or when an enemy dies. In order to avoid searching through the allocated array over and over again for every enemy, a queue has been implemented to skip that step and simply assigning a controller if there is any available in the queue.

Let's break down the mechanic and its implementation steps for a clearer understanding:

Implementation Overview

Enemy Spawn and Initial Behavior:

Objective: Upon spawning, the primary objective of the enemies (goblins) is to identify the target (treasure).

Implementation: This can be achieved by setting an initial state or behavior in the Behaviour Tree that triggers a search for the treasure.

Movement Towards Target:

Objective: After identifying the treasure as their target, enemies move towards it.

Implementation: Implement pathfinding, using Unreal Engine's navigation system, to guide the enemies toward the treasure. This involves calculating a path to the target and moving the enemy along this path, adjusting as necessary if obstacles are encountered or the target moves.

Determining Proximity to Target:

Objective: Enemies continue their advance until reaching a predefined distance from the treasure.

Implementation: Utilize distance checks between the enemy and the target to determine when the enemy is close enough to execute the next step. This can be a simple vector distance calculation that triggers the next behavior once the enemy is within a certain radius of the target.

Running EQS for Positioning:

Objective: To find an optimal position around the treasure for looting, the EQS is executed.

Implementation: EQS queries are used to analyze the environment around the treasure and identify viable positions for the enemies to occupy. This step ensures that enemies don't cluster too closely together and instead surround the treasure effectively. The EQS can evaluate factors like distance from the treasure, distance from other enemies, and cover availability.

Moving to the Selected Position:

Objective: Enemies move to their assigned positions around the treasure.

Implementation: Similar to moving towards the treasure, use the navigation system to move each enemy to its designated EQS-determined position. This might involve recalculating paths to navigate around other enemies and obstacles.

Looting Process:

Objective: Once in position, enemies initiate the looting process.

Implementation: This could be represented by a timer or progress bar, during which the enemy is vulnerable to attacks from the player. Successful completion of the looting process might trigger game mechanics like a decrease in the player's gold reserves.

A diagram of a target

Description automatically generated

This AI mechanic not only enriches the strategic depth of your game but also creates a dynamic and challenging environment that requires players to continuously adapt their defense strategies.

Waves

Wave Progression is a major part of the game since this involves game progression. There is no way of moving forward without this game. And the player needs to clear all the enemies that spawns in each wave. A wave start is triggered by clicking the start button on the player HUD. Once a wave has started, there is no way to end it in anyway other than killing all the enemies.

The wave system is entirely managed by the Wave Manager.

When a new wave starts, the enemy manager is called with the wave count passed into the method which randomly decides the number of enemies to spawn in that wave. Than it moves on to calling the desired number of controllers to spawn the enemies onto the map.

A diagram of a flowchart

Description automatically generated

Tower Operation

After being positioned on the map, towers have a straightforward task. Each tower possesses its own distinct abilities and attacking method. With a designated range, they affect all enemies within this radius. Upon placement, they remain inactive until an enemy enters their range. Once this condition is met, they seek out the nearest target and commence firing upon validation. Occasionally, players may opt to upgrade a tower mid-attack. In such instances, the towers will reset to incorporate the updated stats before resuming their assault.

A diagram of a flowchart

Description automatically generated

Game Loops

A diagram of earning credits

Description automatically generated

A diagram of a diagram

Description automatically generated

A diagram of a building loop

Description automatically generated

A diagram of a wave progression loop

Description automatically generated

UML Class Diagrams

<GoblinSiegeUMLClassDiagrams.png>

## Art Workflows

Materials and Textures

This project uses a decent amount of assets from the epic store as well as some third-party stores/websites. And most of the materials and the textures used in the project comes along with the assets.

Apart from this, some of the basic materials are made by me which includes the decal materials.

Lighting

For the lighting, I have used a Post Processing volume which uses a material which can fade out the exterior parts of the map so that the players cannot see the point where the enemy spawns. There is no such need of this effect, however it gives a more realistic gameplay experience.

Collision

Regarding collisions, given the top-down nature of the game and its reliance on mouse clicks for interactions, a trace method coupled with object channels is employed. This setup filters out objects that shouldn't be traced when clicked, ensuring that only interactable elements (such as buildings) are included in the trace. This approach allows for precise and targeted interactions with the intended objects within the game environment.

Wireframes

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Fonts

Casadia Code: [link](https://fonts.google.com/specimen/Source+Code+Pro)

CopyDuck: [link](https://www.dafont.com/copyduck.font)

Audio

File format: .wav

Bits: 16/24 bit

Assets

This project will use assets from the unreal asset store. Multiple different asset packs would be used to make different assets and use them in the project.

The first asset pack is the [Goblin pack](https://www.unrealengine.com/marketplace/en-US/product/fantasy-goblin/questions) which will be used for the Enemies in the game. This asset includes different a goblin character with different textures and materials and can be applied to get different variations of goblins.

The second one is the [Turret pack](https://www.unrealengine.com/marketplace/en-US/product/automaticweaponsystem) which comes with all the different types of turrets along with the firing mechanism and projectiles that are a requirement for this project.

The map will be taken from the [here](https://www.unrealengine.com/marketplace/en-US/product/wild-west-texas-environment-pack), which comes with a demo map that is good enough for a game like this. However, the materials and the props have been used to create a custom landscape to suit the style.

As for the icons, they have been taken from [here](https://www.gamedevmarket.net/asset/flat-icons-2).

# **Optimisation and Profiling**

## Profiling Systems

Throughout the development process, planning application will be utilized to meticulously monitor the progress. This tool will also serve as a repository for tracking and documenting any encountered bugs.

To ensure that the mechanics are working properly and ensure the quality of the prototype they will be created in isolation in a separate level. Once these features meet the expectations and give the desired output, this will be integrated into the main game. Following the integration of these mechanics, to ensure quality and a bug free experience, rigorous testing will be done, involving not only the newly added mechanics but also all previously implemented systems to identify any potential issues arising from the integration process like proper spawning, interactions and movements. Should any problems surface, I will promptly record them in the planner for subsequent resolution.

A screenshot of a computer

Description automatically generated

Given that this is a single-player game, the game will be tested based on the cpu performances, memory leaks and FPS. There are several methods for testing and debugging that can be beneficial for this project.

The first one is Log Prints. Utilizing log printing proves highly valuable for verifying if the system successfully passes all checks and doesn't overlook any critical elements. This simplifies the process of identifying bugs within the game.

However, log printing may not always suffice. In such instances, the use of breakpoints becomes necessary. Breakpoints enable us to step into each state, assess the values, and ensure they yield valid results. Furthermore, breakpoints assist in confirming whether the code is being executed by the system.

As part of the performance tests, several commands and tools are provided by the engine to test them out. This includes things like enabling FPS and Stats features the FPS.

A screenshot of a video game

Description automatically generated

Visual Logger is another powerful tool which is capable of creating and recording visual representations of the gameplay state. TShis is very helpful when there are cases where there are encountered bugs in the game, and they are sometimes difficult to reproduce.

A screenshot of a computer

Description automatically generated

# **Coding Standards**

## Programming Standards

This kind of games include multiple and very complex game mechanics. Therefore, it is important to make sure that one should adhere to the coding standards diligently. There are several scripts and classes involved in making such a complex project and it is very easy for someone to create or add codes in the scripts where they are not supposed and just because it is easy to do it that way. Maintaining a consistent approach is the key to having a clean and a readable code. Failing to do so can lead to tangling up the scripts real soon and that too without even being aware about it.

Unreal Engine has several classes and packages which has their own speciality and are specifically designed to do certain kind of a job. And it is a good idea to make use those in order to ensure a smooth and optimized gameplay.

Another method of organising the code base is to put comments which should not exceed more than one line and explains what the specific piece of code is doing. However, at the same time, it is also important to make sure, we don’t overdo it. For instance, writing comments after every two lines or writing paragraphs.

When making the logic in the blueprints, it is a good idea to make use of node re-routers to be able to make it more readable.

The next thing is how to balance out the logic between blueprints and C++. As a developer, one should make sure that their logics and programs are optimized to take up as much less space as possible. When writing the code for the complex mechanics, the blueprints are likely to contain a lot of nodes that are being used to get the mechanic or the system working. And this can be problematic when it comes to optimizations. Therefore, it is a great idea to implement them in C++. A lot of times, it is possible that the same logic is written in just a few lines of code.

A screenshot of a computer screen

Description automatically generated

Using naming conventions are very useful. It is very helpful when identifying the type of property or file it is and can save a lot of time when trying to identify where a specific member is a global member and what type is it.

|  |  |
| --- | --- |
| **Naming Conventions** | |
| **Blueprints** | |
| Actors and Normal Blueprints | BP\_ |
| Data Assets | DA\_ |
| Materials | M\_ |
| Interfaces | BI\_ |
| Material Instances | MI\_ |
| Input Actions | IA\_ |
| Texture | T\_ |
| Widgets | WBP\_/W\_ |
|  |  |
| **C++** | |
| Properties | m |
| Boolean | b |
| Base Classes | Base |
| Data Asset Classes | DA\_ |
| Widgets | U |
| Enums | E |
| Structs | F |
| Delegates | (Suffix) Signature |
| Interfaces | I |
| Actor | A |

A screenshot of a computer program

Description automatically generated

# **Production Overview**

## Moscow

|  |  |  |  |
| --- | --- | --- | --- |
| Must | Should | Could | Won’t |
| A playable map | Different sounds for different situations | Different types of Enemies | Multiple Maps |
| Building Placement | Tutorial Phase | More visually appealing UI | Story line |
| Building Upgrade | Settings | Skill Tree |  |
| Enemy AI |  | Complex AI |  |
| Interactable Player HUD |  | Boss Enemies |  |
| Attacking Towers |  | Different Colors on the buildings when upgraded |  |
| Currency |  |  |  |

## Timeline

[Gantt Chart Link](https://staffsuniversity-my.sharepoint.com/personal/s015695k_student_staffs_ac_uk/Documents/PROTOTYPING/TDP_GanttChart.xlsx?web=1)