NATIONAL RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS

Faculty of Computer Science
Bachelor's Programme "Applied Mathematics and Informatics"

Software Project Report on the Topic:
Unity Game

Submitted by the Student:

group #БПАД212, 3rd year of study Martynov Denis Olegovich

Approved by the Project Supervisor:

Lvovich Sergey Makarov
Associate Professor
Faculty of Computer Science, HSE University

Moscow 2024

Contents

Annotation

The goal of the project was to program a video game from scratch using the Unity engine. The result is the creation of scripts for C sharp classes and their successful implementation within the unity editor resulting in a complete standalone game. The Tower Defense genre to which the created game is related is surprisingly underdeveloped in terms of innovative gameplay mechanics when it comes to more popular games. Only a handful of them allow the player to control their own character, even fewer allow them to interact with the world in ways other than placing traps and engaging in combat. This idea is being explored in this project, adding time management and economy into the mix.

Аннотация

Целью проекта было программирование видеоигры с нуля с использованием движка Unity. Результатом является создание скриптов для C Sharp классов и их успешная реализация в редакторе Unity, в результате чего получается полноценная игра. Жанр Tower Defense, к которому относится создаваемая игра, на удивление слабо развит с точки зрения инновационных игровых механик, если речь идет о более популярных играх. Лишь немногие из них позволяют игроку управлять своим персонажем, еще меньше позволяют ему взаимодействовать с миром способами помимо расстановки ловушек и участия в бою. Эта идея исследуется в данном проекте, добавляя в смесь тайм-менеджмент и экономику.

Keywords

Unity, Tower Defense, 3D, C Sharp

1 Introduction

1.1 Unity

Unity is a cross-platform game engine developed by Unity Technologies. It is considered easy to use for beginner developers, and is popular for indie game development. The engine can be used to create three-dimensional (3D) and two-dimensional (2D) games, as well as interactive simulations and other experiences. Unity offers a primary scripting API in C Sharp using Mono, for both the Unity editor in the form of plugins, and games themselves, as well as drag and drop functionality.[4]

1.2 Definitions

This subsection contains the definitions for Unity elements and game development concepts readers may be unfamiliar with.

1.2.1 Scenes

Scenes are where you work with content in Unity. They are assets that contain all or part of a game or application. Each scene contains its own environments, characters, obstacles, decorations, UI, etc. You can create any number of scenes in a project.[4]

1.2.2 Event Handler

When an event occurs, it's sent to all elements registered as listeners of that event. Each listener has it's own function dedicated to handling the event.[4]

1.2.3 Prefabs

The Prefab Asset acts as a template from which you can create new Prefab instances in the Scene. Any edits that you make to a Prefab Asset are automatically reflected in the instances of that Prefab.[4]

1.2.4 Box collider

The Box Collider is a basic cuboid-shaped collision primitive.[4]

1.2.5 HUD

Heads Up Display is the method by which information is visually relayed to the player as part of a game's user interface

1.3 Main Inspirations

In this project Unity is used in order to create a Tower Defense game.

Tower defense (TD)[8] is a subgenre of strategy games where the goal is to defend a player's territories or possessions by obstructing the enemy attackers or by stopping enemies from reaching the exits, usually achieved by placing defensive structures on or along their path of attack. This typically means building a variety of different structures that serve to automatically block, impede, attack or destroy enemies.

Existing Tower Defense games will serve as inspirations for the project. In particular:

"Orcs Must Die!"[6] In it, the player does not just control the placement of defensive structures, but also engages in defense through a playable character, which is rare for TD games. However, unlike in "Orcs Must Die!", in this project the player character will not engage in combat. Instead, it will be used to collect resource, destroy obstacles on the map and carry traps from their creation station to wherever the player wants to place them.

"Bloons Tower Defense"[5] serves as the main inspiration for trap types, trap upgrades and wave management.

"Plants vs Zombies"[7] for the grid trap placement system and resource management economy.

An inspiration for the time management and quick decision making required of the player is also taken from the Real Time Strategy genre.

1.4 Borrowed Assets

Most visual and audio assets were not created for the purpose of this project, but were borrowed from free asset websites. Visual 3D assets[2], sound effects[1], music[3].

2 System Description

This section contains a description of all implemented C Sharp classes for the project.

2.1 Game Input

A single instance of this class exists in the game. It reads player's inputs and invokes each input's corresponding event. It also contains a function for calculating normalized movement vector of the player character.

The player inputs are as follows:
WASD - movement
E - Interact
F - Alternative Interact
R - Interact Rotate
ESC - Pause

2.2 Game Manager

An instance of this class controls the state of the game. States include: Waiting, Countdown, Playing and Game Over. Player's inputs can only be read in state Playing, with the exception of pause input. The class includes logic for changing states, invoking events related to changes in state and handles pause input. When the game is paused it causes ingame time to stop and invokes the related event. It also contains the real timer and wave counter for the game.

2.3 Timer UI

This script controls the player's HUD showing current in game time and wave taking this information from the "Game Manager".

2.4 Countdown UI

This script controls the countdown UI present during the Countdown state of the game and hides it afterwards.

2.5 Pause UI

The pause UI covers the entire screen and contains multiple buttons handled by this script. Resume button unpauses the game. Music button causes "Music Manager" to change music volume. Sound button causes "Sound Manager" to change sound volume. Main Menu button changes scene to the Main Menu.

2.6 Music Manager

This class handles playing and changing volume of the ingame music.

2.7 Sound Manager

This class handles changing volume of the ingame sounds and contains the public function that is triggered by classes when a sound needs to be played.

2.8 InGame Object

The purpose of this class was to implement an ability to transfer game objects from one entity to the other, with different behaviours stemming from ownership of said object. Any such object must contain information about it's holder or parent. The class definition contains functions implementing changing the ownership of the object and spawning this object into existence. Each different object of this class must also have a corresponding "Scriptable Object" information.

2.9 InGame Object SO

Scriptable Objects, from here on referred to as SOs, are simple scripts containing a reference to the corresponding object's prefab or prefabs, name and any additional information that object possesses that is the same for any instance of that object. The InGame Object SO contains 2 prefabs, one for the holdable object and one for the corresponding Trap, explained later, serving as a link between them.

2.10 I Object Parent

This is an interface that must be inherited by any class that could potentially hold an InGame Object. Inheriting it requires the class to have definitions for functions that attach and clear the InGame Object, get information about the InGame Object and get the position where the object should be shown in relation to the parent. These functions will not be mentioned in descriptions of classes bellow inheriting I Object Parent, but they are present in the code.

2.11 Player

This is the class explaining the logic behind the player controlled character. More specifically it handles player movement and interaction by listening to "Game Input" events. The movement function handles changing player character's position while taking into account potential collisions with other objects. The interaction function works by changing the "Selected" object. Only the object in front of the player character is counted as "Selected" and any interact input triggers the interact function on the "Selected" object only. There are 3 different possible interaction functions: interact, interact alternative and rotate. "Selected" object must be of class "Base" or inherited class "Base". Player class also inherits "I Object Parent".

2.12 Selected Visual

This script is assigned to a hidden component within every interactable object. It's purpose is to activate additional visual component if the base it is attached to is currently "Selected" and hide it otherwise.

2.13 Player Animation

This class is responsible for triggering a walking animation for the player character. Originally the animation was implemented, however it was discarded after changing assets. It is currently unused but remains as a placeholder for future implementation.

2.14 Obstacle

This class does not contain any logic within and only serves as a tag for other classes functions regarding collision.

2.15 Base

Base class is named this way because it serves as a base for any interactable object. It inherits "I Object Parent" and "Obstacle", and contains empty virtual functions for all interaction actions.

2.16 I Has Progress

This is an interface that requires any inheriting class to implement the event that triggers when it's progress is changed, whatever that progress may be. It is mainly used for "Progress Bar UI" integration.

2.17 Progress Bar UI

This UI is attached to any object inheriting "I Has Progress" and it shows the progress in form of a partially filled bar above said object. Each time it listens to an event triggered by the "I Has Progress" object it changes the percentage of the filled portion of the bar.

2.18 Resource UI

The game features a usable resource similar to currency. This class serves as both a calculator of the resource and as a UI editor. It has a public function which, when triggered by other classes, recalculates the amount of resource held by the player and changes the number indicating this amount on the player's HUD.

2.19 Breakable Obstacle

It inherits "Base" and "I Has Progress". Objects of this class serve to block the player character's movements and their interaction with a "Ground Tile" below it. When player interacts with it the object begins to "break". Breaking it completely requires interacting with it multiple times and when it happens the objects frees the "Ground Tile" below it and disappears.

2.20 Resource Gathering Spot

This class works similar to the "Breakable Object" class. It also inherits "Base" and "I Has Progress" and multiple interactions with it are needed to trigger it's effect. However, when progress on it reaches a certain threshold a predetermined amount of resource is awarded to the player and the progress is reset.

2.21 Chest

It inherits "Base" class and upon a single interaction it awards the player a large amount or resource and disappears, freeing the "Ground Tile" underneath it.

2.22 Workbench

Workbenches essentially serve as vendors for the player to trade resource for "Traps". Each workbench prefab has it's own "InGame Object SO" and price assigned to it. Upon interaction with it, if the player isn't already holding an object and player has enough resource to pay, a new instance of the holdable prefab of the assigned "InGame Object SO" is spawned with player as it's parent and the price is deducted from player's stored resource amount.

2.23 Ground Tile

When player is holding something it slightly changes player character's logic for the "Selected" object choice. Instead of picking the object directly in front of it a Ground Tile in front is chosen instead. This means that player is unable to interact with things when they hold an object with the exception of Ground Tile and they are unable to interact with a Ground Tile with empty hands. When a Ground Tile is interacted with, holdable object is transferred from the player to that tile and a new instance of a "Trap" is spawned on top of the Ground Tile. The type of trap spawned depends on the holdable object in possession of the Ground Tile.

2.24 Trap

This class inherits "Base" class and redefines alternative interaction and rotate functions but not the main interact. Rotate changes the direction that the visual of the object is facing. Alternative interaction destroys the object and transfers the bellow Ground Tile's holdable object back to the player. The class also contains the function to spawn a trap which is triggered by Ground Tile.

2.25 Ballista

This is one of the final trap scripts. It inherits "Trap". Ballistas activate every few seconds, spawning an arrow and sending it in the direction ballista's visual is facing. Upon spawn the arrow is assigned it's movement speed, damage and maximum distance by the ballista. Interacting with a ballista upgrades it, provided the player has enough resource. It can be upgraded up to 4 times, each increasing it's fire rate, arrow damage, future upgrade cost and the size of it's visual component.

2.26 Tower

Tower is a second type of trap. it acts similar to ballista, sharing it's interact function. Unlike the ballista, it doesn't spawn a single, long-range arrow, but 8 short-range arrows each 45 degrees from the last.

2.27 Ballista Arrow

This class defines arrows spawned by ballistas. It contains it's spawn function and movement function. It moves according to it's assigned speed, direction and maximum distance. If on it's path it collides with an "Obstacle" or an "Enemy" it gets destroyed dealing it's assigned damage to the "Enemy".

2.28 Tower Arrow

This class defines arrows spawned by towers. It's only difference from "Ballista Arrow" is that it does not get destroyed upon collision with an obstacle. The presence of the obstacle collision is unnecessary due to short range and could lead to undesired behaviour for diagonally moving arrows.

2.29 Projectile SO

Sriptable objects of this class contain a prefab to previously described arrows and are attached to their corresponding tower to link them.

2.30 Enemy

All prefabs of the enemies in the game contain the same script. It defines enemy's movement speed, starting health and damage they deal. Class's functions define enemy spawn, enemy taking and dealing damage, it's single direction movement, it's turning in case it is standing on a "Path Turn Tile".

2.31 Enemy SO

This SO contains each enemy prefab and it's corresponding index used for their spawn.

2.32 Path Base

Only purpose of this class is to be inherited by all other Path classes.

2.33 Path Turn

Same as "Obstacle" this class only serves as a tag for enemies to trigger turn function on.

2.34 Path Spawn

This class handles "Enemy" spawn in according to the logic described within it. Depending on the current wave, which is obtained from the "Game Manager", it decides which enemies to spawn and how fast to spawn them. It contains the list of "Enemy SO's" and lists of enemies per wave presented in enemy indexes. After wave 10 wave patterns begin to repeat with increased intensity up to infinity, constantly increasing game difficulty.

2.35 Path End

This script is given to the last path tile. When an "Enemy" reaches this tile it disappears and deals damage to the player according to the "Enemy"s damage.

2.36 Life UI

This class serves as both a calculator of the player health and as it's UI editor. It has a public function which, when triggered by "Path End", recalculates the amount of health left and changes the number indicating this amount on the player's HUD. Upon reaching 0 health it triggers and event for "Game Manager" to change state to Game over.

2.37 Game Over UI

Upon reaching Game Over state this script's ui covers the screen and presents information about reached time and wave taken from the "Game Manager". Pressing the interact button in Game Over state causes the Main Menu scene to load.

2.38 Main Menu UI

The main menu UI contains multiple buttons handled by this script. Play button loads the Game scene. Quit button closes the application. Controls button brings up the controls UI.

2.39 Controls UI

This script handles showing the controls UI in the Main Menu and the button to hide it.

2.40 Loader and Loader Call Back

When a scene is changed, instead of loading the next scene directly the Loading scene is loaded first instead. These two scripts handle loading this scene and preserving the wanted next scene between loads.

2.41 Hide Element On Start

This script is attached to any object with visuals that are undesired during the game but are helpful during level development and turns them off at the start of each scene.

3 Map

The map of the Game scene is designed by using Unity's Tilemap system. Tilemaps work as grids for the designer to put objects in the center of each square of the grid. Five tilemaps constitute a level: Ground, OnGround, OnGroundVisual, Border and OnBorder. Border serves as walls for the map. OnBorder serves for decorations on top of the Border. Ground is for Ground and Path tiles. OnGround is for workbenches, obstacles and resource gathering spots. OnGroundVisual is for sctrictly decorative items on the ground. Using the Tilemap system allows for an easy creation and iteration of level design cycles.

4 Player's perspective

When user first loads the game they see the Main Menu screen from which they can access the controls, quit the application or start the game in a single click. The Background of the Main Menu screen features most of the ingame assets.

Upon starting the game, player sees the countdown in the middle of the screen and has some time to analyze the map and prepare. Once the countdown reaches zero, the player is given control of his character and he may begin playing the game.

During the game the player's available actions are clearly shown by the highlighted object they can interact with at any given moment. If interacting with the object will cause the player to lose resource a UI showing the resource and the amount to be taken will be presented above the interactable object. The enemy path is presented as a single line of color different from the normal ground. Enemies spawning on one end of the road and marching to the other clearly indicates the goal of the game.

Upon reaching a Game Over, the player is presented with his results and is prompted to press the interact key which takes them back to the Main Menu.

5 Conclusion

The result of the project is a fully functional and playable game. All implemented systems are working well without any critical bugs or errors. The scale of the game, however is rather small. More playable maps, placeable traps, resource types and enemy types can be added without limit. The visuals and sounds currently consist of free, borrowed assets. No animations are implemented. As any game, this one can be improved infinitely with infinite amount of additional mechanics and systems. A game is finished only when it's developers decide so, no matter the state of it.

References

[1] FreeSound. *Collaborative collection of free sounds.* URL: https://freesound.org/.

[2] Kenney. *Collection of free 2D and 3D assets.* URL: https://kenney.nl/.

[3] Code Monkey. *Unity tutorials and assets website.* URL: https://unitycodemonkey.com/.

[4] Unity. *User Manual.* URL: https://docs.unity3d.com/Manual/.

[5] Wikipedia. *Bloons Tower Defense.* URL: https://en.wikipedia.org/wiki/Bloons_Tower_Defense.

[6] Wikipedia. *Orcs Must Die!* URL: https://en.wikipedia.org/wiki/Orcs_Must_Die!.

[7] Wikipedia. *Plants vs. Zombies.* URL: https://en.wikipedia.org/wiki/Plants_vs._Zombies.

[8] Wikipedia. *Tower Defense.* URL: https://en.wikipedia.org/wiki/Tower_defense.