Geekbrains

Разработка игрового мобильного приложения на платформе Android с помощью среды разработки Unity.

IT-специалист:
Программист мобильных
устройств
Рассолодина А. В.

Самара
2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Игровая индустрия в настоящее время успешно развивается. Рынок игр огромен. В разработку игр вкладываются большие деньги. И далеко не факт, что она окупится. Однако не всегда было так.

Разработка игр времён зарождения игровой индустрии была намного меньше по стоимости, чем сейчас. Например, игра, написанная одним программистом, могла разойтись сотней тысяч копий и окупиться сполна.

С развитием компьютерных технологий, размер групп разработчиков увеличивался, так как пользователь хочет новых особенностей в играх, которых он раньше не видел.

Существует огромное множество игр разных жанров и с разным сюжетом. И у каждого жанра есть свои плюсы и минусы. Пользователь выбирает жанр исключительно из своих предпочтений.

Также недавно начали развиваться мобильные игры, которые сейчас занимают одну нишу с компьютерными играми.

Цель данного документа - подробно разобрать разработку приложения.

Задачи:

1) Провести аналитический обзор целевой аудитории;
2) Провести обзор требований, аналогичных приложений и средств разработки.
3) Провести тестирования.

В рамках данного документа под приложением подразумевается разрабатываемая игра.

1. АНАЛИТИЧЕСКИЙ ОБЗОР

.1 Анализ игровых жанров для мобильных устройств

Существует много жанров игр на данный момент. Однако не все подходят для мобильных устройств. Мобильное устройство ограниченно в управлении по сравнению с персональным компьютером. Жанры, где надо быстро совершить несколько действий сразу не подходят на мобильные устройства. Например, FPS (First Person Shooter) не подходит для мобильных устройств, так как в таком жанре надо одновременно перемещаться, менять угол обзора и стрелять. Но существую жанры, которые очень подходят для мобильных устройств.

Ниже приведены некоторые из этих жанров:

. Runner. В этом жанре персонаж бесконечно бежит в каком-то направлении, игроку следует преодолевать препятствия, обычно просто перепрыгивая их.

. Пошаговые стратегии. Игры, где время на принятие решений неограниченно или очень длинное. Игры этого жанра могут быть с различной механикой.

. Гоночные симуляторы. Управление осуществляется с помощью встроенного в мобильное устройство гироскопа.

. Tower Defense. В играх этого жанра игроку даётся возможность строить башни для защиты, которые стреляют по проходящим рядом врагам, идущим по определённому маршруту

Таким образом, жанры для мобильных устройств должны удовлетворять следующим требованиям:

Простота управления.

Малое число одновременно совершаемых действий

Малая продолжительность

1.2 Анализ целевой аудитории

Одно из важнейших действий в разработке игры является анализ целевой аудитории. Иногда разработчики хотят угодить всем пользователям, добавляя в игру излишний функционал, который может никогда не использоваться. Чтобы избежать этого, следует провести анализ целевой аудитории, чтобы понять, какой жанр игр сейчас популярен и почему.

Игроков можно разбить на несколько категорий:

1) Казуальные игроки - люди, играющие в несложные игры и проводящие в них мало времени, с целью развлечения или просто быстрого проведения времени. Большинство казуальных игроков предпочитают играть на смартфонах. Основными жанрами для казуальных игроков являются те жанры, которые можно объединить в группу “тайм-киллеров” - игр без особого смысла и сюжета, но с увлекательным гемплеем.

2 Хардкорные игроки - люди, которые обычно избегают простых игр, и проводящие за играми большое кол-во времени. Основными жанрами, в которые играют хардкор-геймеры, являются RPG, Action, FPS. Также же у хардкорных игроков пользуются популярностью MMORPG, которых можно провести не один месяц непрерывной игры.

Исходя из этого, был выбран жанр, разрабатываемой игры в рамках выпускной квалификационной работы, - Tower Defense. Такой жанр подходит больше для казуальных игроков.

1.3 Анализ требований к приложению

Мобильные приложения менее требовательны к аппаратному обеспечению, нежели чем приложения на ПК.

Большинство современный мобильных игры имеют следующие системные требования:

1) Объём оперативной памяти не меньше 1 Гб.
2) Частота процессора не меньше 1 ГГц.
3) Операционная система Android должна быть не ниже версии 4.0

Исходя из этого для игры, разрабатываемой в ходе выпускной квалификационной работы, были определены следующие требования:

. Требования к составу и параметрам мобильного устройства:

На мобильное устройство должна быть установлена операционная система Android версии 4.0 или выше.

Объём оперативной памяти должен быть не менее 512мб

Процессор должен иметь частоту не менее 1Ггц.

Мобильное устройство должно иметь сенсорный экран с разрешением не менее 640x480

. Требования к программной документации:

Документ для разработчика (с кодом, комментариями и техническим описанием).

Техническое задание для разработчиков.

Руководство пользователя.

.4 Анализ аналогичных приложений

На данный момент существует множество игр в жанре Tower Defense. Самые известные из них указаны в таблице 1.3. Они различаются игровой механикой, стилем, управлением. Целью этого обзора является выявление наиболее предпочтительных особенностей игр этого жанра.

Таблица 1.3 Самые известные игры в жанре

Характеристика	Название игры		
	Kingdom Rush	Plants vs. Zombies	GemCraft
Вид графики	2D	2D	2D
Разработчик	Ironhide Game Studio	PopCap Games	Game in a Bottle <https://en.wikipedia.org/wiki/Halfbrick_Studios>
Платформы	Windows, iOS, Android	Windows, iOS, Android, Xbox 360,Nintendo DS	Всё, что поддерживает Adobe Flash
Управление	Сенсорный экран, клавиатура + мышь	Сенсорный экран, мышь	Сенсорный экран, мышь

Из приведённой выше таблицы, можно сделать вывод, что игры этого жанра имеют преимущественно 2D-графику. Так же наиболее популярные платформы являются Windows, iOS, Android.Управление осуществляется с помощью мыши или сенсорного экрана, в зависимости от устройства, на котором запущена игра.

1.5 Анализ технологий для разработки мобильных приложений

Существует множество технологий разработки мобильных приложений, которые отличаются дизайном, функциями и языком разработки.

. Среда разработки Eclipse.

Свободная среда разработки мультиплатформенных приложений, разработанная компанией Eclipse Foundation. Основным

преимуществом является то, что любой разработчик может расширить Eclipse своими модулями. Eclipse написана на Java, поэтому является мультиплатформенным продуктом. Исключением является библеотека SWT, которая разрабатывается отдельно для каждой из операционных систем. Так же Eclipse поддерживает большинство языков программирования, благодаря встраиваемым модулям.

Для реализации игр с помощью Eclipse, придётся подключать дополнительные графические библиотеки, которые замедляют работу игры и усложнит написание кода.

. Android Studio

Интегрированная среда разработки приложений для платформы Android. Одна из особенностей является расширенный редактор макетов: WYSIWYG <https://ru.wikipedia.org/wiki/WYSIWYG>, способность работать с UI <https://ru.wikipedia.org/wiki/Интерфейс_пользователя> компонентами при помощи Drag-and-Drop <https://ru.wikipedia.org/wiki/Drag-and-drop>, функция предпросмотра макета на нескольких конфигурациях экрана.

3. Game Maker: Studio

Популярный движок для разработки приложений для множества платформ. Используется специально разработанный язык программирования. Возможна интеграция системы контроля версий.

Основным недостатком является крайне неудобная работа с 3D графикой.

. Unity

Среда для разработки приложений, поддерживающих множество платформ. Также поддерживается разработка веб-приложений, была специально разработана для разработки игр. Многие элементы, которые в остальных IDE надо было реализовывать в качестве отдельных классов или участков кода, здесь поддерживаются на уровне движка,

например, коллизии игровых объектов.

Также имеется пространство, разбитое на координаты, в котором реализуется игра или приложение.

Разрабатываемое приложение состоит из отдельных скриптов, отвечающих за определённые функции. Скрипт представляет собой отдельный класс со своими полями и методами.

Из выше представленных преимуществ и недостатков, можно сделать вывод, что Unity наиболее подходит для разработки игр, так как в ней представлен широкий функционал для разработки игр. В отличии от GameMaker, Unity хорошо работает с 3D графикой.

2. ПРОЕКТИРОВАНИЕ

Для успешной разработки любого приложения, сначала надо его спроектировать, то есть разработать концепцию, интерфейс и алгоритмы реализации функциональных требований.

2.1 Разработка концепции игрового приложения

Первым шагом проектирования является разработка концепции игры.

2.1.1 Разработка сеттинга игры

Сеттинг - среда или стиль, в которой происходит действие игры. Например, в игре “Монополия” может быть сеттинг Нью-Йорка или сеттинг СССР, но поменяется только стиль игры, а не её механика. Это может быть, как выдуманный мир, так и существующий.

Был разработан следующий сеттинг игры:

На поселение Микроорганизмов нападают Паразиты. Паразиты хотят уничтожить Микроорганизмы. Микроорганизмы встали на защиту своего поселения.

Микроорганизмы будут выполнять роль башен стреляющим по противникам, представленных Паразитами.

Микроорганизмы будут 3 видов:

1) Воин. Будет иметь среднюю дальность стрельбы и среднюю скорострельность (рис. 2.1.1(а)).

2) Стрелок. Будет атаковать с большого расстояния, но иметь низкую скорострельность (рис. 2.1.1(б)).

3) Слепой. Будет иметь маленькую дальнобойность, но высокую скорострельность (рис. 2.1.1(в)).

а) б) в)

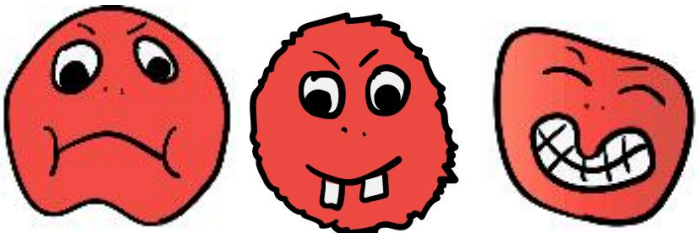

Рисунок 2.1.1 - Спрайты Микроорганизмов: а) Вони б) Стрелок в) Слепой

У Микроорганизмов будет анимация стрельбы, которая будет состоять из двух спрайтов.

Спрайт-графический объект в компьютерной графике, который свободно перемещается по экрану. В большинстве случаях спрайты используют в 2D-графике, так как при изменении угла обзора спрайт искажается.

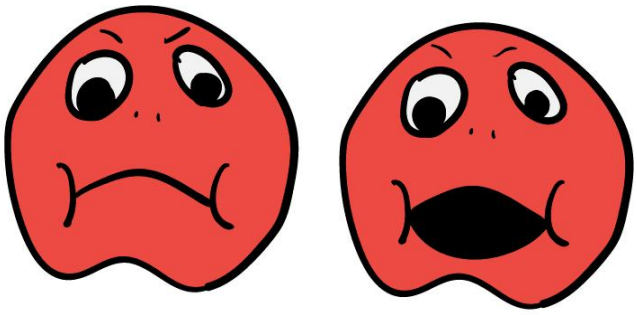

Рисунок 2.1.2 - Спрайты анимации стрельбы

Спрайты, представленные на рисунке 2.1.2 будут сменять друг друга во время стрельбы.

Паразиты будут 2 видов:

1) Воин. Имеет средний запас здоровья и среднюю скорость

передвижения. (рис. 2.1.3(а))

2) Командир. Имеет большой запас здоровья и низкую скорость передвижения (рис. 2.1.3(б))

а) б)

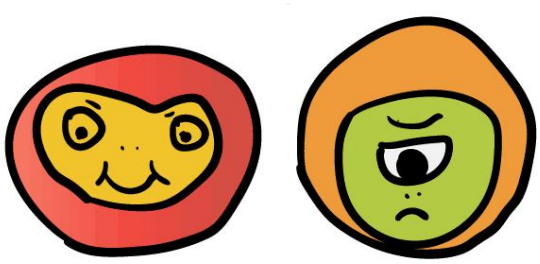

Рисунок 2.1.3 - Спрайты Паразитов: а) Воин б) Командир

2.1.2 Разработка основной механики

Есть множество вариантов жанра Tower Defense. Они различаются логикой постройки и улучшения башен, движения противников. Например, в игре Kingdom Rush, башни строятся в определённых местах карты, что ограничивает число стратегий постройки этих зданий, но с другой стороны загоняет игрока в жёсткие условия, так как, места строительства могут располагаться в неоптимальных местах.

Для приложения была выбрана механика постройки башен, описанная выше. Для того чтобы игра была с одной стороны была простой для освоения, а с другой стороны была трудной для прохождения.

Был выбран тип маршрута движения противников. В большинстве похожих игр маршруты - это лабиринты, по которым идут противники к конечной точке. В некоторых играх игрок сам строит маршрут используя для этого башни. Был выбран первый вариант, чтобы упростить игру игроку.

Был выбран обычный вариант улучшения. Для большинства игра такого жанра характерна обычная система улучшения: за определённую

сумму денег улучшаются характеристики. В других играх реализован механизм слияния башен, в результате которого из двух башен образуется одна.

2.2 Разработка интерфейса игры

Интерфейс игры играет важную роль в понимании пользователем игры. Поэтому он должен быть с одной стороны минималистическим, а с другой стороны содержать ценную для пользователя информацию.

Игра будет разбита на несколько сцен:

1) Главное меню.
2) Уровни.

Для каждого уровня будет отдельная сцена, но интерфейс будет общим для всех уровней.

Для этих сцен надо разработать интерфейс, который будет включать в себя необходимый функционал.

2.2.1 Главное меню

Главное меню - сцена, которую пользователь видит, как только заходит в игру. Здесь располагаются 2 кнопки (рисунок 2.2.1):

1) Кнопка перехода на меню выбора уровней. После её нажатия игрок переходит в меню выбора уровней, где будет выбор из уровней до которых дошёл игрок ранее.
2) Кнопка выхода из игры. При нажатии закрывает приложение.

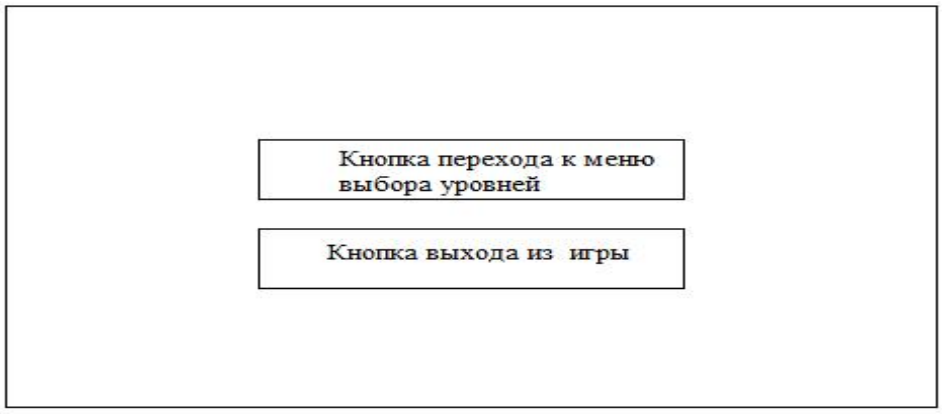

Рисунок.2.2.1 - Главное меню

В меню выбора уровней будут располагаться кнопки перехода на уровень и кнопка возврата в главное меню (рисунок 2.2.2):

Рисунок 2.2.2 - Меню выбора уровней.

2.2.2 Уровни

Интерфейс будет включать в себя текущую информацию о уровне, запасе денег и жизней игрока, кнопки постройки башен. Также будет кнопка паузы, чтобы приостанавливать игру. (рисунок 2.2.3): В меню паузы будет 2 кнопки: возобновить игру и выход в главное меню(рисунок 2.2.4).

Рисунок 2.2.3 - Интерфейс уровней

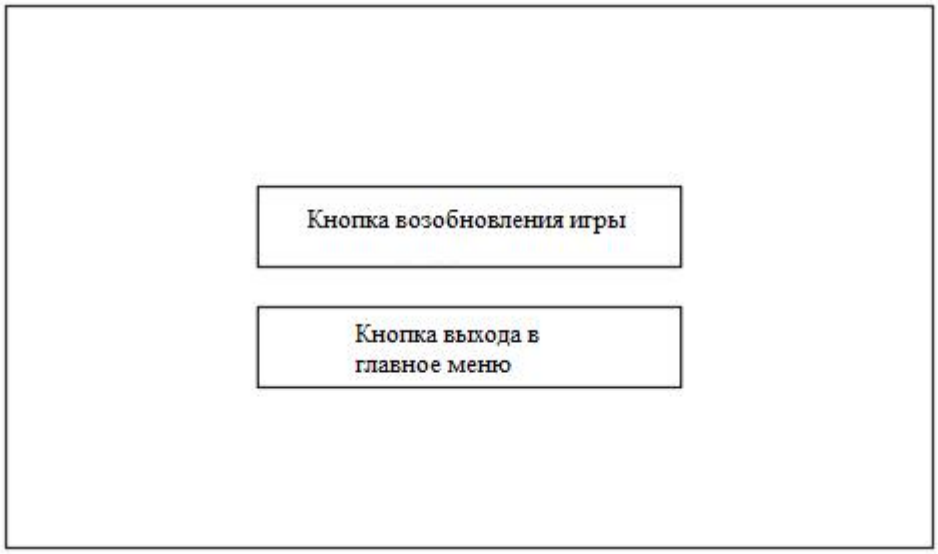

Рисунок 2.2.4 - Меню паузы

2.3 Разработка алгоритмов

Правильное планирование алгоритмов позволит избежать многих ошибок в реализации. В соответствии с разработанной механикой игры надо разработать алгоритмы, реализующие эту механику.

2.3.1 Разработка постройки башен

Башни будут строится в определённых местах карты. Для этого надо разработать места для постройки. При нажатии на них, если выбрана какая-либо башня для постройки и достаточно денег, будет построена выбранная башня. Если башня не выбрана или недостаточно денег, башня не будет построена. Блок-схема алгоритма представлена на рисунке 2.3.1.

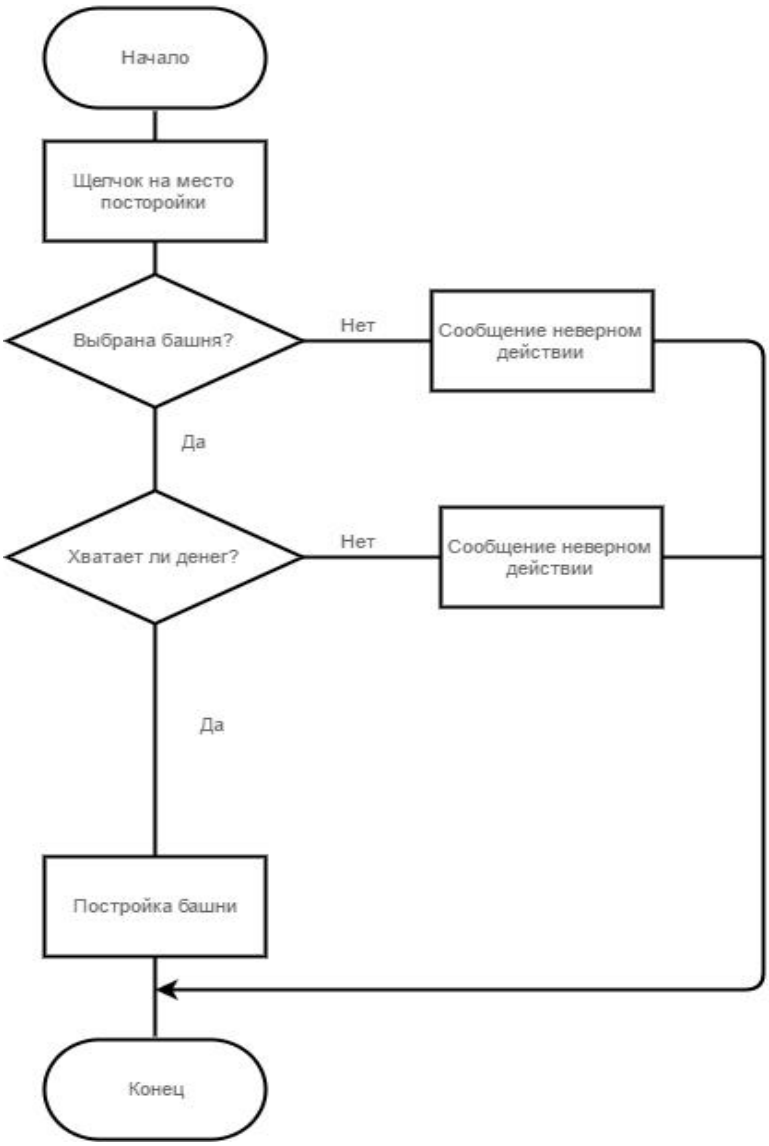

Рисунок 2.3.1 - Блок-схема алгоритма постройки башен.

2.3.2 Разработка противников и их передвижение

Противники будут двигаться по определённому маршруту. Маршрут в игре будет представлен в виде извивающегося пути, а в коде в виде списока маршрутных точек. Противники создаются в первой точке маршрута и идут к следующей, до тех пор, пока не достигнут конца или не будут уничтожены. При достижении конечной точки у игрока отнимается 1 жизнь. Блок-схема алгоритма представлена на

рисунке 2.3.2.

Рисунок 2.3.2 - Блок-схема алгоритма передвижения противников

2.3.3 Разработка стрельбы и снарядов

У каждой башни будет определённый радиус поражения, в котором она может поражать цели. При появлении в нём противника, он записывается в список целей. Если противник уходит за радиус, он удаляется из списка. Башня последовательно стреляет в цели, при уничтожении цели она переключается на следующую в списке цель.

Блок-схема алгоритма представлена на рисунке 2.3.3.

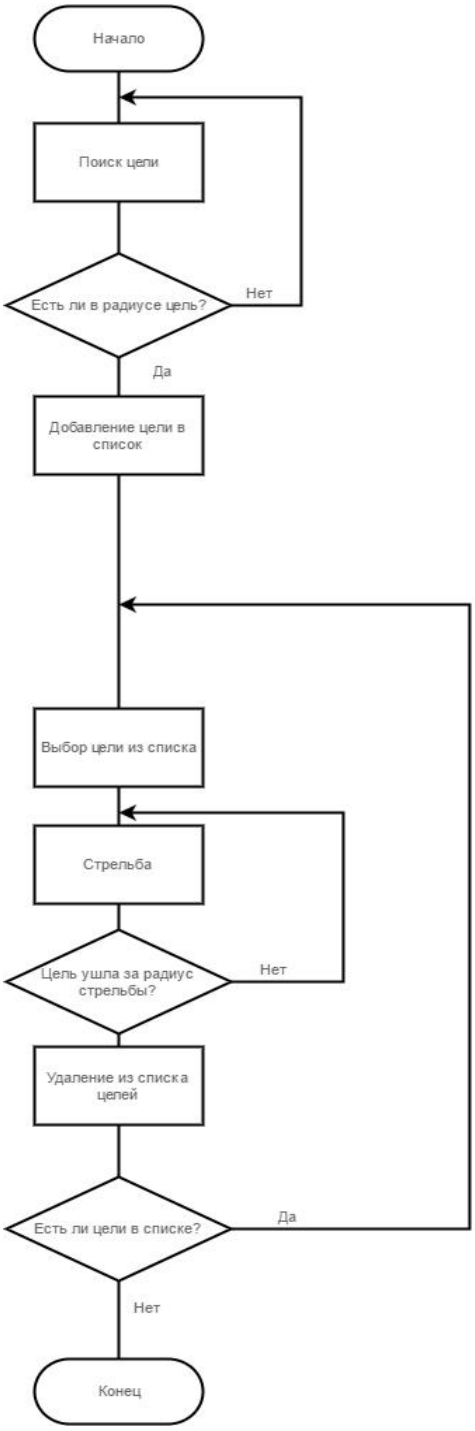

Рисунок 2.3.3 - Блок-схема алгоритма стрельбы

При выстреле создаётся снаряд, который целенаправленно летит в цель. При попадании в цель он уничтожается и у противника

отнимаются очки здоровья. Если здоровье упадёт до 0, противник погибает.

3. РЕАЛИЗАЦИЯ

В выбранной среде разработки поддерживаются 2 языка программирования: JavaScript и C#. Был выбран язык C#, так как он предоставляет больше возможностей для ООП.

3.1 Реализация алгоритмов на языке C#

В Unity код представляется в виде скриптов. Скрипты в Unity-классы со своими полями и методами. Большинство скриптов активно используют функцию Update, которая вызывается при каждой смене кадров на экране. Благодаря этому функции можно реализовать множество алгоритмов, работающих в реальном времени.

3.1.1 Описание переменных скриптов

Описание переменных приведены в виде таблице 3.1

Таблица 3.1 Описание переменных скрипта ShootEnemies

Название переменной	Тип переменной	Назначение
enemiesInRange	List<GameObject>	Список целей башни
lastShotTime	Float	Время последнего выстрела, используется для определения времени следующего выстрела
monsterData	MonsterData	Информация о башне, представленная в виде отдельного класса
Target	GameObject	Цель для стрельбы
minimalEnemyDistance	Float	Используется для нахождения минимального расстояние до последней точки маршрута.
distanceToGoal	Float	Используется для хранения расстояния до конечной точки маршрута текущего противника, сравнивается с minimalEnemyDistance
Direction	Vector3	Используется определения направления стрельбы, чтобы поворачивать башню в сторону цели
bulletPrefab	GameObject	Используется для хранения прототипа снаряда.
startPosition	Vector3	Начальная позиция снаряда
targetPosition	Vector3	Позиция цели

newBullet	GameObject	Создаваемый снаряд
Animator	Animator	Используется для хранения анимаций и воспроизведения их
audioSource	AudioSource	Используется для хранения и воспроизведения звука выстрела

.1.2 Реализация выбора цели и стрельбы

Для каждого объекта можно создать компонент Collider. На Рисунке 3.1.1 показан компонент CircleCollider, который задаёт круглую коллизионную область определённого радиуса.

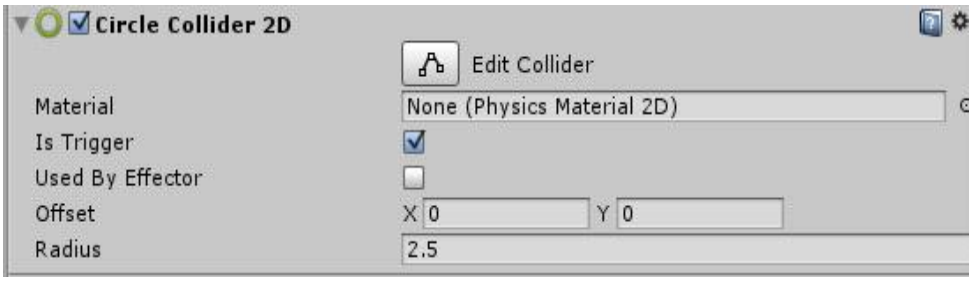

Рисунок 3.1.1 - Компонент CircleCollider

Если в эту область заходит объект с компонентом Collider, у этих объектов вызывается функция void OnTriggerEnter2D с передаваемой ссылкой на объект, который зашёл в область. Эта функция используется для заполнения списка целей. Сначала у объекта проверяется тэг (ключевое слово для разделения объектов на типы) «Enemy» (Рисунок 3.1.2), чтобы другие башни в радиусе на попадали в список.

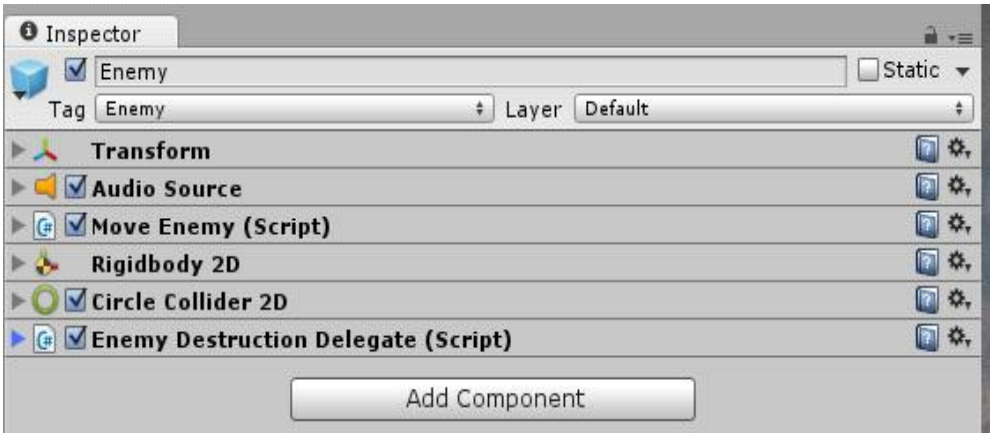

Рисунок 3.1.2 - Установка тэга «Enemy» для противника

Когда объекта областями коллизий расходятся, вызывается функция void OnTriggerExit2D с ссылкой на объект, который вышел из области коллизий. Эта функция применяется для удаления врагов из списка целей. Когда враг выходит из радиуса, он удаляется из списка целей.

Выбор цели производится в функции Update().Цель выбирается из списка целей EnemyinRange. Если он пуст, то переменной target присваивается значение null. Если нет, то переменной target присваивается ссылка на противника, который находится ближе других к последней точке маршрута.

Если переменная target не равна null, то вызывается функция Shoot.

Функция Shoot создаёт объект bullet и присваивает ей следующие параметры:

1) 1)target. Будет присвоена текущая цель.

2) startPosition. Будет присвоена позиция башни, обозначающая начальное положение снаряда.

3) targetPosition. Будет присвоена позиция цели.

После создание объекта bullet, он начинает двигаться к цели. Между башней и целью проводится линия. Движение по линии осуществляется с помощью функции Lerp класса Vector3. Конечная точка полёта меняется при движении цели

Функция Lerp возвращает значение типа Vector3.

У функции Lerp 3 входных параметра:

1) Начальная точка

2) Конечная точка

3) Параметр, показывающий положение между начальной и конечной точками. Параметр принимает значение от 0 до 1. В данном случае этот параметр принимает значение timeInterval * speed / distance, где timeInterval - время, прошедшее с момента выстрела, speed - скорость, distance - расстояние до цели. Когда позиции снаряда и цели будут равны, у цели ищется наследственный компонент класса healthbar(Рисунок 3.1.3) . В нём находится переменная currenthealth, отвечающая за текущие очки здоровья цели. Из неё вычитается значения равное количеству урона снаряда. Если у цели осталось меньше 0 очков здоровья, она уничтожается, игроку начисляется награда. У всех башен, которые имели в списке целей уничтоженного врага, вызывается функция OnEnemyDestroy с ссылкой на уничтоженный объект, которая убирает эту цель из списка.

Рисунок 3.1.3 - Объект класса Enemy

После попадания в цель снаряд уничтожается.

3.1.3 Настройка работоспособности игры

Для того чтобы скрипт выполнял заложенную в него функцию, его надо привязать к какому-либо объекту. Для работоспособности башни были написаны 2 скрипта:

1) ShootEnemies.cs. Отвечает за стрельбу по противникам.

2) MonsterData.cs. Хранит в себе информацию о башне и её улучшениях.

Привязка скриптов производится путём добавления компонента Script для объекта, как показано на рисунке 3.1.4

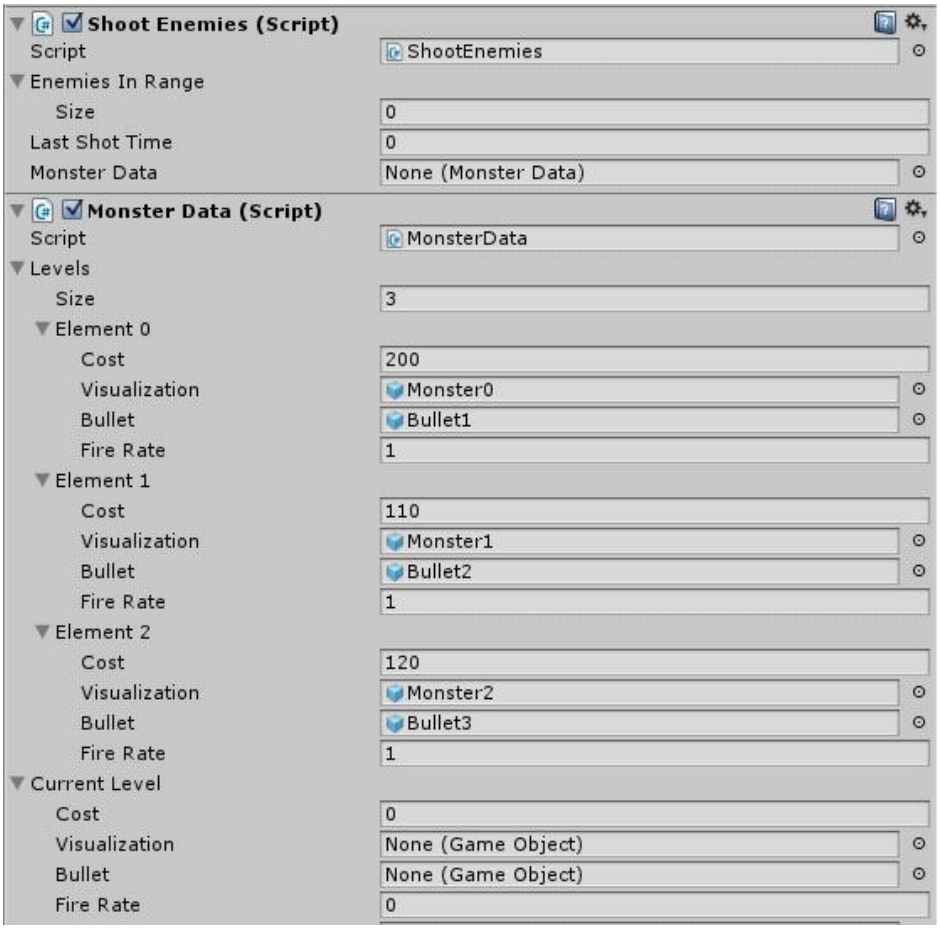

Рисунок 3.1.4 - Компоненты Script объекта Monster

Как показано на рисунке 3.1.4 для скриптов надо задать начальные значения. Для скрипта MonsterData надо задать размер массива с уровнями башни. У каждого уровня есть стоимость Cost, визуализация башни Vizualization, тип снаряда Bullet, скорость стрельбы Fire Rate.

Визуализация башни Vizualization представлена в виде объекта с параметрами показанном на рисунке 3.1.5

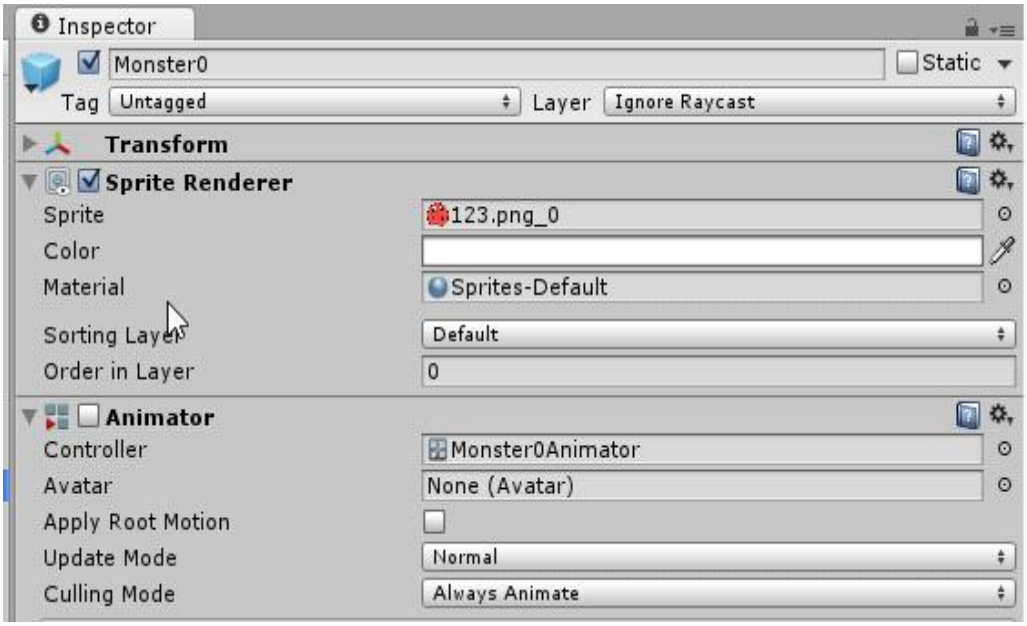

Рисунок 3.1.5 - Объект Monster

Компонент Sprite Render отвечает за обрисовку спрайта башни на экране. В параметрах компонента Sprite Render можно задать сам спрайт, прозрачность, материал.

Компонент Animator отвечает за воспроизведение анимации. В параметре Controller надо указывать файл типа Animator Controller. Этот файл отвечает за последовательность действий при анимации как показанно на рисунке 3.1.6

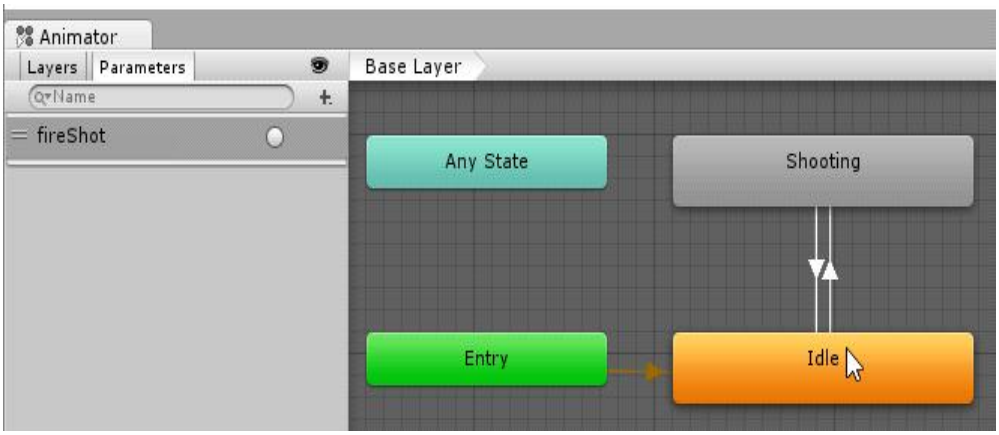

Рисунок 3.1.6 - Интерфейс Animator Controller

После создания башни, она переходит в состояние Idle, в которой он просто ничего не делает. Если башня стреляет, она переходит в состояние Shooting и выполняется анимация, привязанная к состоянию Shooting (Рисунок 3.1.7)

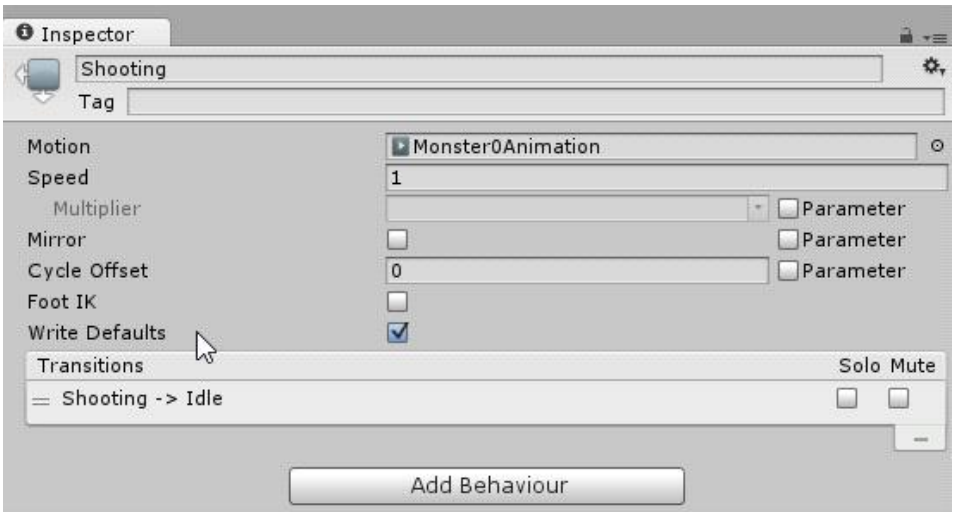

Рисунок 3.1.7 - Интерфейс состояния Shooting

Интерфейс создания анимации показан на рисунке 3.1.8. Здесь можно задавать действия, совершаемые над объектом. Анимация зацикленная, то есть после последнего кадра идёт первый.

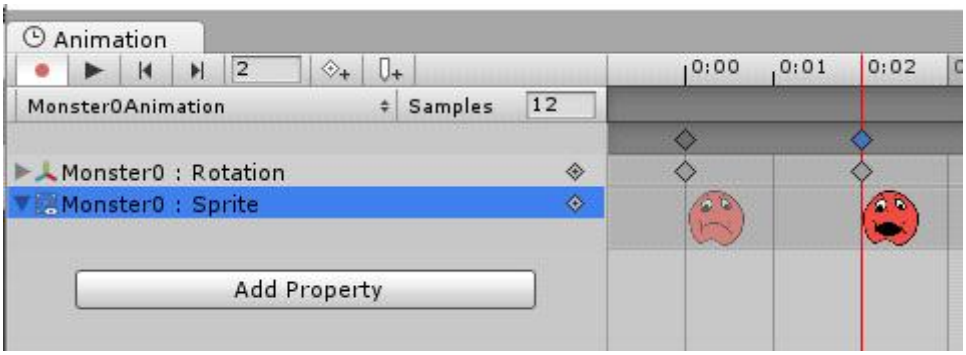

Рисунок 3.1.8 - Интерфейс создания анимации

Тип снаряда Bullet представляется в виде объекта с привязанным к нему скриптом Bullet Behavior. Параметры объекта показаны на рисунке 3.1.9

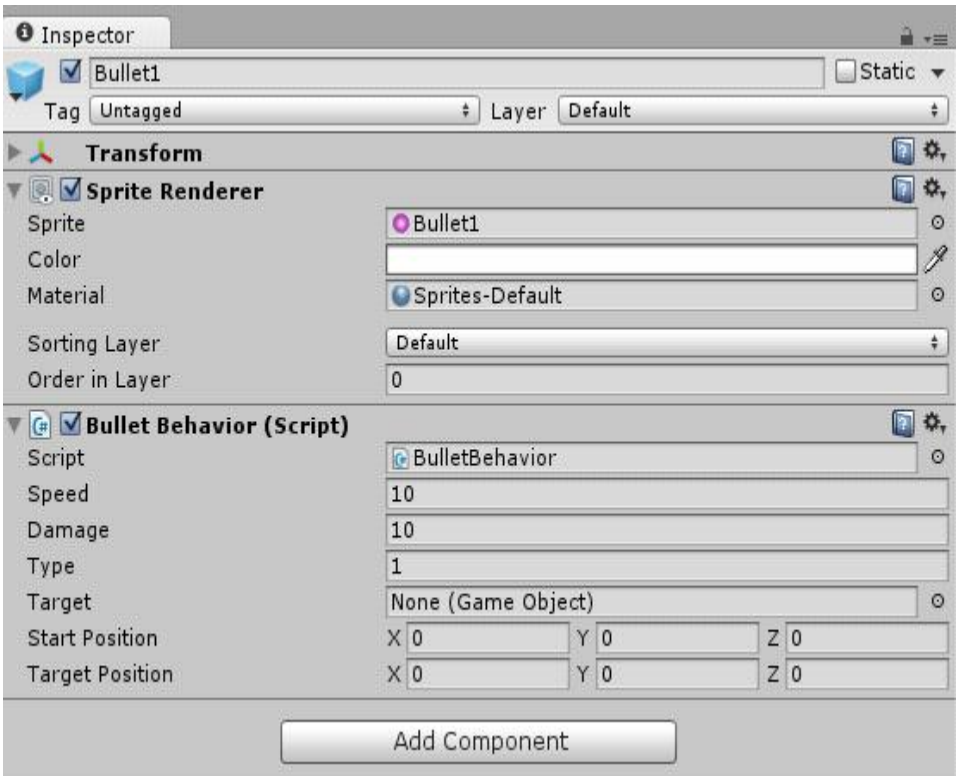

Рисунок - 3.1.9 Объект Bullet

В скрипте Bullet Behavior есть следующие параметры:

) Speed. Скорость полёта снаряда.

) Damage. Урон, наносимый противнику при попадании.

) Target. Цель, в которую летит снаряд.

) Start Position. Начальное положение снаряда.

) Target Position. Позиция текущей цели.

Были настроены места постройки башен. На рисунке 3.1.10 показаны параметра объекта Openspot, отвечающего за места постройки

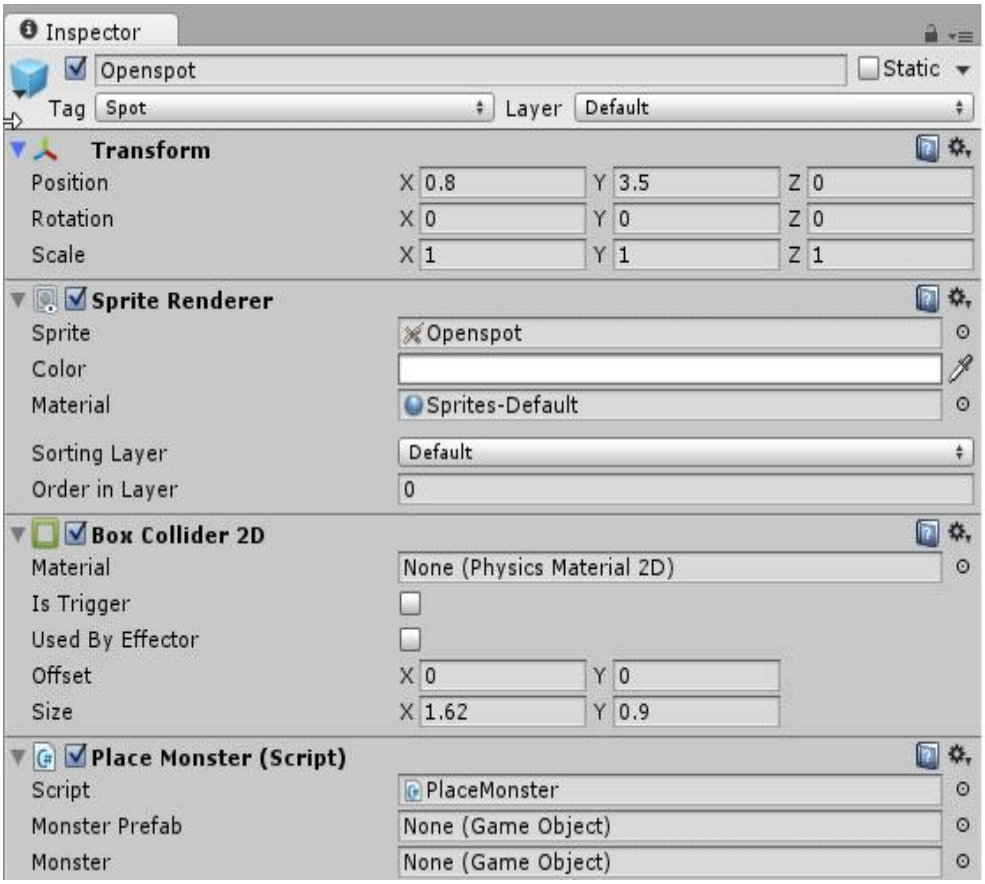

Рисунок 3.1.10 - Объект Openspot

У привязанного скрипта Place Monster есть 2 параметра:

1) Monster Prefab. Ссылка на выбранную игроком башню.

2) Monster. Ссылка на текущую построенную башню.

Маршрут движения противников представлен в виде вспомогательных объектов Waypoint, у которых только один компонент Transform, отвечающий за расположение.

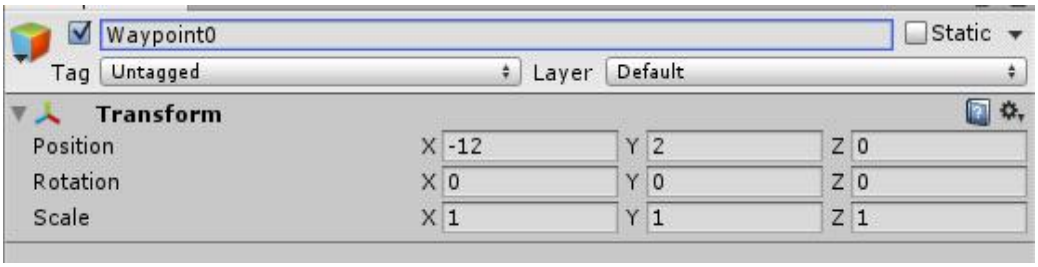

Рисунок 3.1.11 - Объект Waypoint

Объект Road отвечает за хранение маршрута и за создание противников. Параметры объекта Road показаны на рисунке 3.1.12

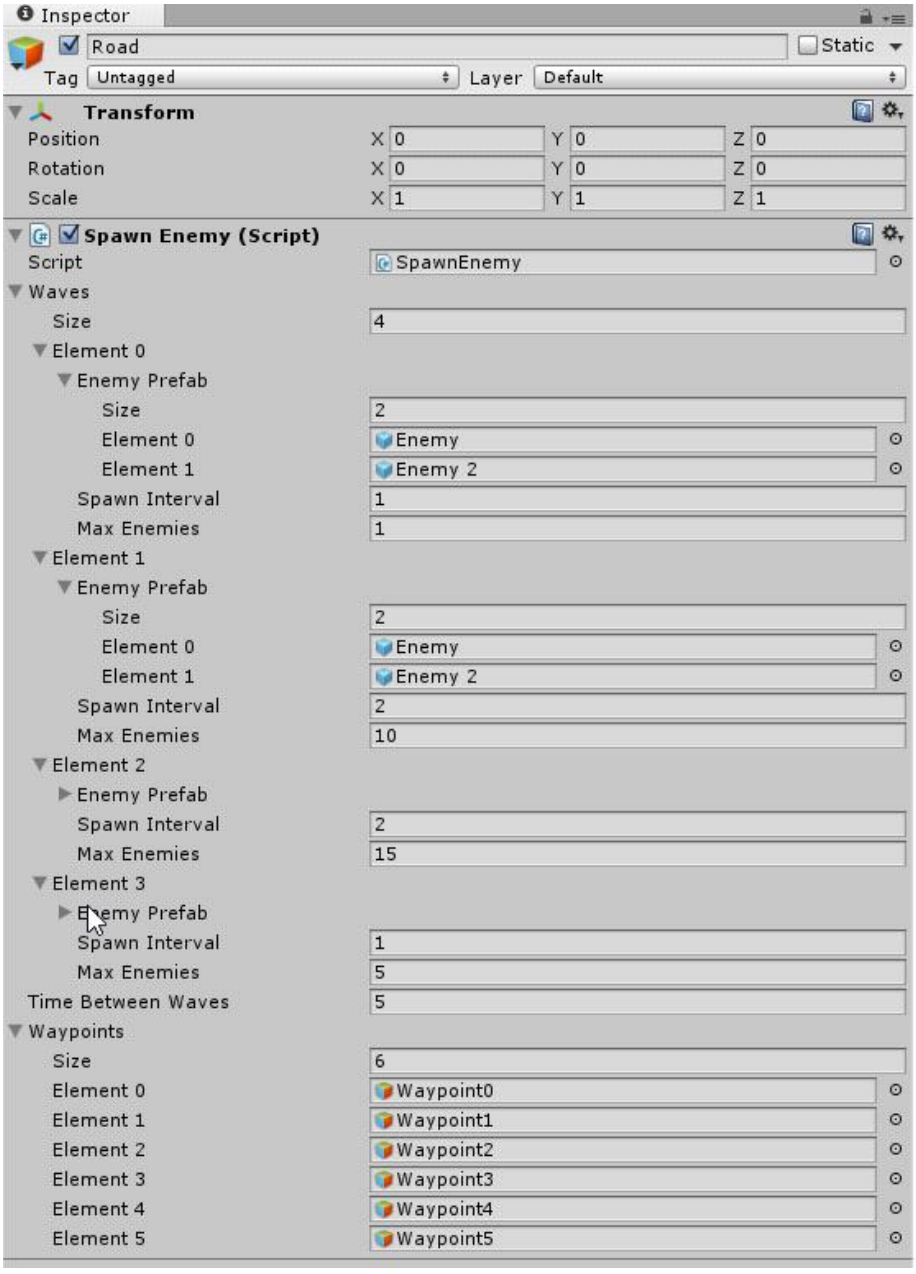

Рисунок 3.1.12 - Объект Road

Привязанный скрипт SpawnEnemy имеет следующие параметры:

1) Waves. Массив типа Wave, где хранится информацию о

волнах противников. В классе Wave указаны ссылки на противников, интервал их появления и максимальное число.

2) Time Between Waves. Время между волнами.

3) WayPoints. Массив маршрутных точек.

.2 Реализация интерфейса игры

Интерфейс игры в Unity реализуется с помощью компонента Canvas. Далее на него выносятся элементы, которые необходимы для интерфейса. Canvas располагается вне игрового поля в редакторе, но при запуске игры Canvas масштабируется под разрешение и расположится поверх игрового поля

3.2.1 Реализация интерфейсы главного меню

Для главного меню создана отдельная сцена. Она включает себя стартовое меню и меню выбора уровней. Они объединены в компоненты типа Panel, чтобы было легче переключатся между ними.

В стартовом меню всего 2 кнопки.

1) Play- кнопка перехода на меню выбора уровней. К ней привязан скрипт Maintochoose.cs. При нажатии на кнопку, из скрипта вызывается функция void toggle(), которая скрывает стартовое меню и делает видимым меню выбора уровней (Рисунок 3.2.1).

2) Exit - кнопка выхода из игры. К ней привязан скрипт Exit.cs. При нажатии на кнопку, из скрипта вызывается функция void QuitGame(), которая закрывает приложение.

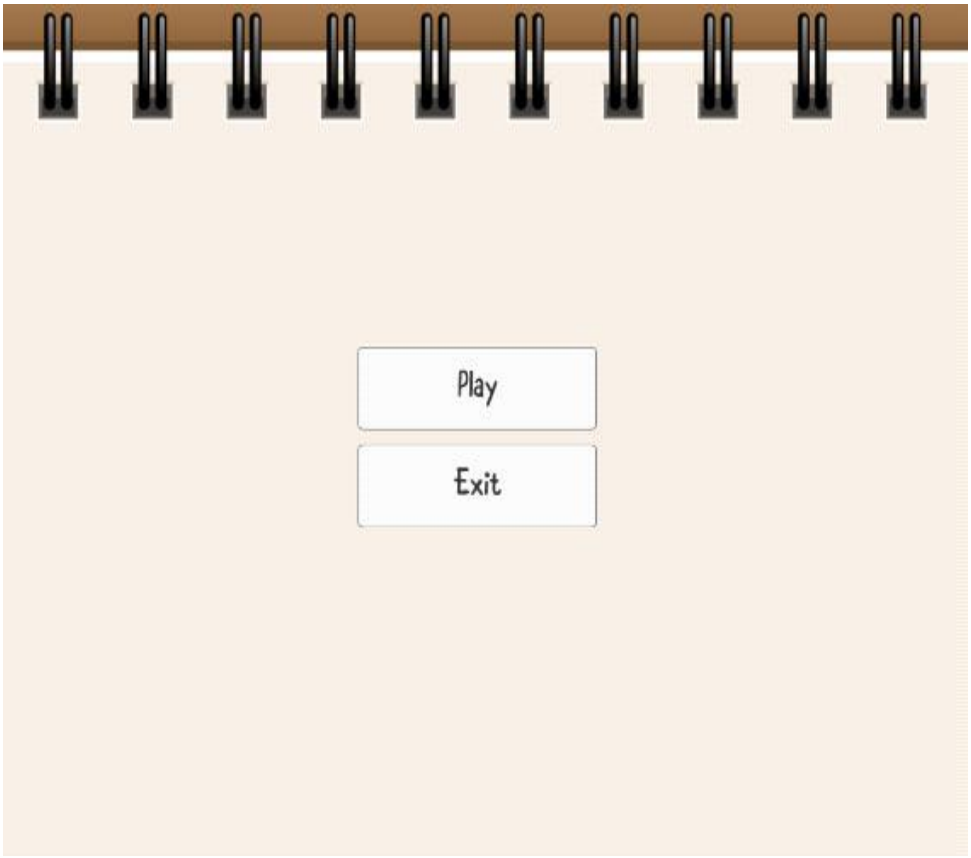

Рисунок 3.2.1 - Интерфейс стартового меню

В меню выбора уровней реализованы кнопки перехода на уровни и кнопка возврата в стартовое меню(Рисунок 3.2.2).

Все кнопки перехода работают одинаково. К ним привязаны скрипты, имеющих функцию void cl1ick(), которая срабатывают при нажатии на кнопку. Функция void cl1ick() загружает сцену с уровнем, соответствующий кнопке.

К кнопке назад привязан скрипт ChooseToMain.cs, имеющий функцию void toggle(), которая скрывает меню выбора уровней и делает видимым стартовое меню

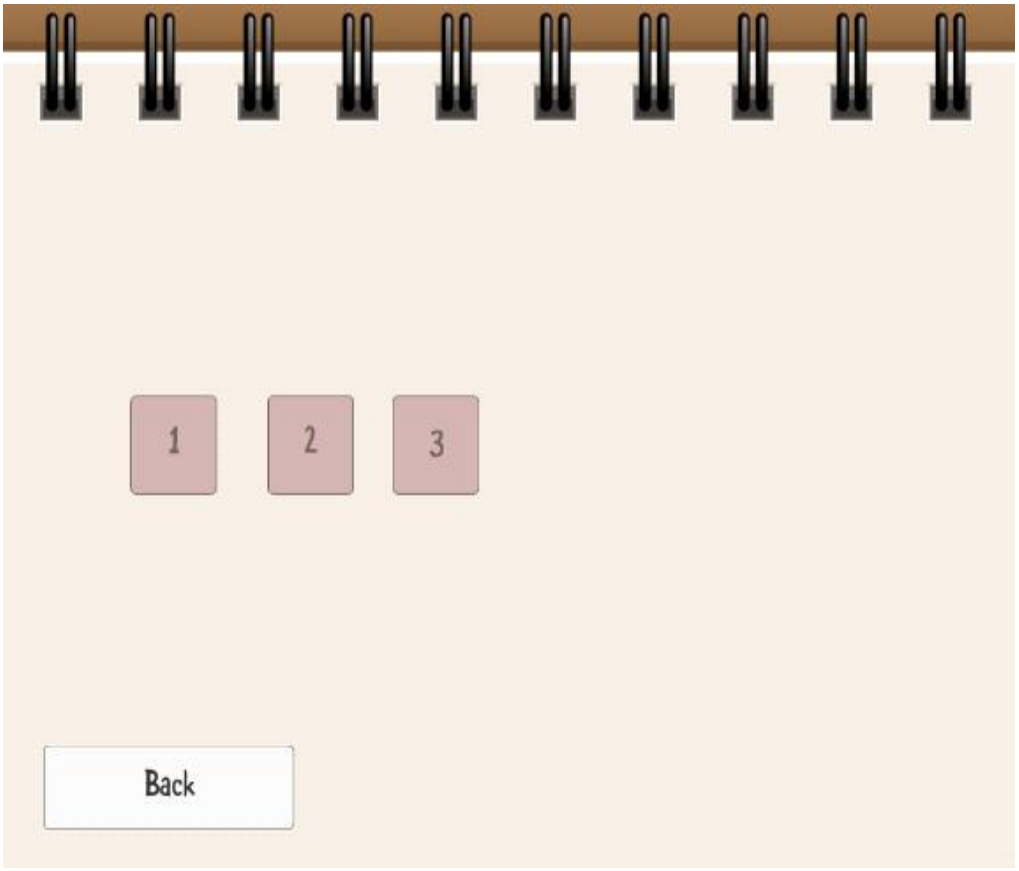

Рисунок 3.2.2 - Интерфейс выбора уровней

3.2.2 Интерфейс уровней

Для отображения информации, сверху расположены 3 компонента Text. В GameManager хранится сама информация и ссылки на компоненты Text. Каждый кадр эта информация обновляется.

К кнопкам выбора башен привязан скрипт tower_type_1.cs, хранящий в себе ссылку на тип башни, который закреплён за кнопкой. При нажатии на кнопку, скрипт передаёт тип башни GameManager.C помощью этого есть возможность переключения между типами башен.

Так же в левом нижнем углу есть кнопка паузы. К ней привязан скрипт Pause.cs. Реализованная в нём функция void togglePause() останавливает время в игре и показывает меню паузы(Рисунок 3.2.3).

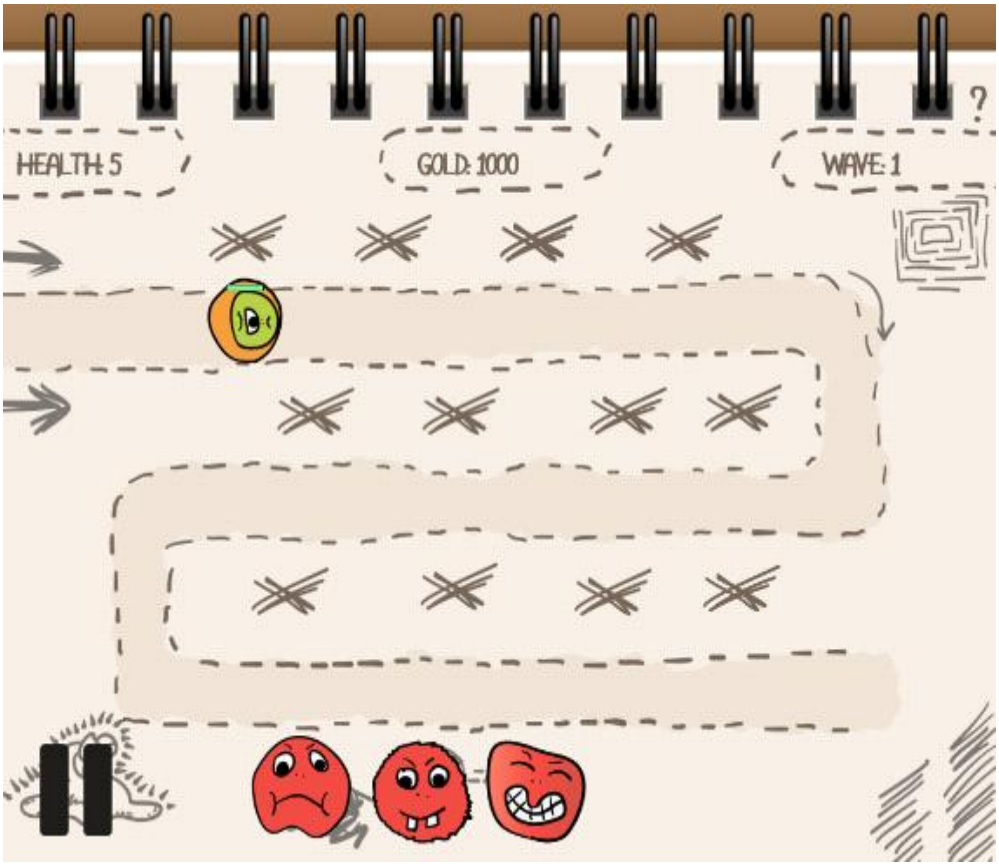

Рисунок 3.2.3 - Интерфейс уровня

Кнопка Resume использует тот же скрипт, что и кнопка паузы. Функция void togglePause() возобновляет время и скрывает меню паузы.

Функция void togglePause() в зависимости от Time.timeScale, обозначающим масштаб времени, выполняет разные функции. Если время не остановлено, то она останавливает время и показывает меню паузы, иначе возобновляет игру и скрывают меню паузы (Рисунок 3.2.4).

(Time.timeScale != 0f)

{.SetActive(true);.timeScale = 0f;

}

{.SetActive(false);.timeScale = 1f;

}

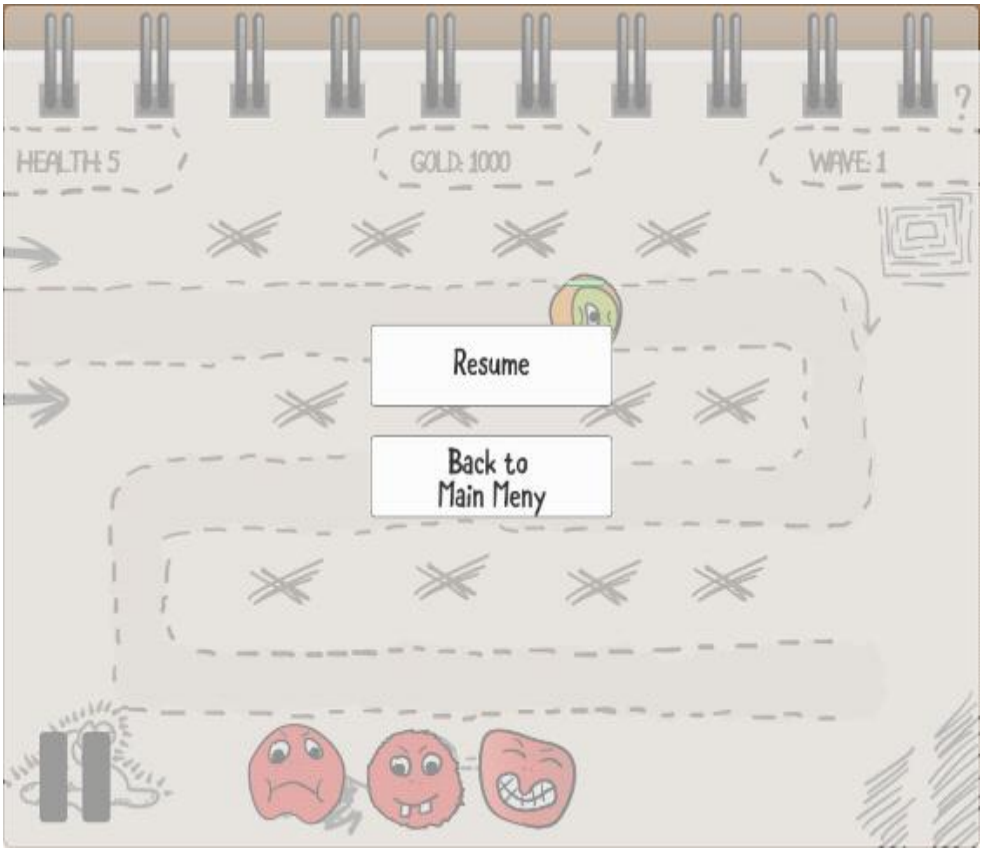

Рисунок 3.2.4 - Интерфейс меню паузы

4. ТЕСТИРОВАНИЕ

.1 Методы тестирования

Существует множество методов тестирования ПО

4.1.1 Модульное тестирование

Модульное тестирование - тестирование отдельных модулей разрабатываемой программы отдельно от других модулей, в целях изоляции от воздействия других модулей.

Игра, разработанная в Unity, состоит из множества скриптов. Их всех надо проверить на работоспособность. Для этого лучше всего подойдёт модульное тестирование.

4.1.2 Функциональное тестирование

Функциональное тестирование программы на соответствие функциональным требованиям.

Чтобы проверить функции, которым должна соответствовать игра, необходимо применить функциональное тестирование для тестирования игры целиком. Эти два метода позволят протестировать игру с внешней и с внутренней стороны игры.

4.2 Результаты тестирования

.2.1 Результаты модульного тестирования

Каждый модуль (скрипт) тестировался отдельно. Модульное тестирование проводилось во время реализации игры для более

быстрого исправления появляющихся ошибок. Скрипты в финальной их версии работоспособны и выполняют возложенную на них функцию.

4.2.2 Результаты функционального тестирования

Каждая функция тестировалась согласно функциональным требованиям приложения.

В качестве тестировщиков выбраны независимые лица из числа одногруппников и знакомых. Был выдан список функций, требующие проверки.

Результаты функционального тестирования приведено в таблице 4.1

Таблица 4.1 - Результаты функционального тестирования

Функция	Результат
Постройка башни	При нажатии на место постройки башня строится, если выбрана башня
Выбор типа башни	Кнопки выбора типов башен работают корректно, выбирается нужный тип башни
Генерация противников	Противники генерируются в начальной точке маршрута.
Движение противников	Противники движутся плавно, но разворот происходит резко
Стрельба башен	Все башни стреляют в нужные цели, башни разворачиваются в сторону цели. Снаряды попадая в цель отнимают у них очки здоровья.
Уничтожение противника	Если здоровье противника опускается ниже 0, то он уничтожается и счётчик золота увеличивается.
Достижение противниками конечной точки маршрута	При достижении конечной точки противник уничтожается и у игрока отнимается одна жизнь
Пауза	Кнопка паузы выполняет свою функцию, корректно останавливает игру и показывает меню паузы
Выбор уровней	Кнопки выбора уровней загружают нужные уровни.

ЗАКЛЮЧЕНИЕ

В результате разработки была создана игра «Parasite Attack» на платформу Android с помощью среды разработки Unity. В процессе работы были проведены: аналитический обзор целевой аудитории, требований к приложению, были разработаны и реализованы алгоритмы. Проведено тестирование и было выявлено, что разработанная игра соответствует разработанным требований.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

3. Руководство Unity [Электронный ресурс]: Unity-Руководство / Руководство по Unity- Режим доступа: http://docs.unity3d.com/ru/current/Manual/

. Что такое платформа Eclipse и как ее использовать? [Электронный ресурс]: IBM DeveloperWorks- Режим доступа: https://www.ibm.com/developerworks/ru/library/os-eclipse/

. Тарас Нудиев / Стоит ли переходить на Android studio? [Электронный ресурс]: Awesomedevelop- Режим доступа: http://awesomedevelop.blogspot.ru/2014/12/android-studio.html

. Unity VS GameMaker: Studio [Электронный ресурс]: Stopgame - Режим доступа: http://stopgame.ru/blogs/topic/58024

. Рукаводство по программированию на C# [Электронный ресурс]: Microsoft Developer Network - Режим доступа: https://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx

ПРИЛОЖЕНИЕ

Скрипты проекта

) BulletBehavior.csUnityEngine;System.Collections;class BulletBehavior : MonoBehaviour {float speed = 10;int damage;int type;GameObject target;Vector3 startPosition;Vector3 targetPosition;float distance;float startTime;GameManagerBehavior gameManager;

// Use this for initializationStart () {= Time.time;= Vector3.Distance (startPosition, targetPosition);gm = GameObject.Find("GameManager");= gm.GetComponent<GameManagerBehavior>();

}

// Update is called once per frameUpdate () {=target.transform.position;timeInterval = Time.time - startTime;.transform.position = Vector3.Lerp(startPosition, targetPosition, timeInterval * speed / distance);(gameObject.transform.position.Equals(targetPosition)) {(target != null) {

Transform healthBarTransform = target.transform.FindChild("HealthBar");

HealthBar healthBar = .gameObject.GetComponent<HealthBar>();.currentHealth -= Mathf.Max(damage, 0);(type==2){speedTarget= target.GetComponent<MoveEnemy>();(speedTarget.currentspeed==speedTarget.speed)

{.currentspeed*=0.5f;

}

}(healthBar.currentHealth <= 0) {(target);audioSource =

target.GetComponent<AudioSource>();.PlayClipAtPoint(audioSource.clip, transform.position);.Gold += 50;

}

}(gameObject);

}

}

}

)

GameManagerBehavior.csUnityEngine;System.Collections;UnityEngine.UI; class GameManagerBehavior : MonoBehaviour {GameObject TowerPref;Text goldLabel;int gold;Text waveLabel;GameObject[] nextWaveLabels;bool gameOver = false;int wave;Text healthLabel;GameObject[] healthIndicator;int health;int Gold {{ return gold; }{= value;.GetComponent<Text>().text = "GOLD: " + gold;

}

}

// Use this for initializationStart () {= 0;= 1000;= 5;

}

// Update is called once per frameUpdate () {

}int Health {{ return health; }{(value < health) {.main.GetComponent<CameraShake>().Shake();

}= value;.text = "HEALTH: " + health;(health <= 0 && !gameOver) {= true;gameOverText = GameObject.FindGameObjectWithTag ("GameOver");.GetComponent<Animator>().SetBool("gameOver", true);

}(int i = 0; i < healthIndicator.Length; i++) {(i < Health) {[i].SetActive(true);

} else {[i].SetActive(false);

```
        }
        }
        }
        }int Wave {{ return wave; }{= value;(!gameOver) {(int i = 0; i < nextWaveLabels.Length; i++) {[i].GetComponent<Animator>().SetTrigger("nextWave");
        }
        }.text = "WAVE: " + (wave + 1);
        }
        }
        }
        )
        MonsterData.csUnityEngine;System.Collections;System.Collections.Generic;UnityEngine.UI;
        [System.Serializable]class MonsterLevel {int cost;GameObject visualization;GameObject bullet;float fireRate;
        }class MonsterData : MonoBehaviour {List<MonsterLevel> levels;MonsterLevel currentLevel;Button SellB;Button UpgradeB;
        // Use this for initializationStart () {
        }
        // Update is called once per frame
        void Update () {
        }MonsterLevel CurrentLevel {{currentLevel;
        }{= value;currentLevelIndex = levels.IndexOf(currentLevel);levelVisualization = levels[currentLevelIndex].visualization;(int i = 0; i < levels.Count; i++) {(levelVisualization != null) {(i == currentLevelIndex)
```

{[i].visualization.SetActive(true);

} else {[i].visualization.SetActive(false);

}

}

}

}OnEnable() {= levels[0];

}MonsterLevel getNextLevel() {currentLevelIndex = levels.IndexOf (currentLevel);maxLevelIndex = levels.Count - 1;(currentLevelIndex < maxLevelIndex) {levels[currentLevelIndex+1];

} else {null;

}

}void increaseLevel() {currentLevelIndex = levels.IndexOf(currentLevel);(currentLevelIndex < levels.Count - 1) {= levels[currentLevelIndex + 1];

}

}

}UnityEngine;System.Collections;class MoveEnemy : MonoBehaviour {

[HideInInspector]GameObject[] waypoints;int currentWaypoint = 0;float lastWaypointSwitchTime;float speed = 1.0f;float currentspeed;

// Use this for initializationStart () {= speed;= Time.time;

}

// Update is called once per frame

void Update () {

// 1

Vector3 startPosition = waypoints

[currentWaypoint].transform.position;

Vector3 endPosition = waypoints [currentWaypoint + 1].transform.position;

// 2 pathLength = Vector3.Distance (startPosition, endPosition);

if (currentspeed != speed) {

}totalTimeForPath = pathLength / currentspeed;

float currentTimeOnPath = Time.time - lastWaypointSwitchTime;.transform.position = Vector3.Lerp (startPosition, endPosition, currentTimeOnPath / totalTimeForPath);

// 3 (gameObject.transform.position.Equals(endPosition)) {(currentWaypoint < waypoints.Length - 2) {

// 3.a ++;= Time.time;();

} else {

// 3.b (gameObject);

AudioSource audioSource = gameObject.GetComponent<AudioSource>();

AudioSource.PlayClipAtPoint(audioSource.clip, transform.position);gameManager =.Find("GameManager").GetComponent<GameManagerBehavior>();.Healt h -= 1;

}

}

}

) MoveEnemy.csfloat distanceToGoal() {distance = 0;+= Vector3.Distance(.transform.position, [currentWaypoint + 1].transform.position);(int i = currentWaypoint + 1; i < waypoints.Length - 1; i++) {startPosition = waypoints [i].transform.position;endPosition =

waypoints [i + 1].transform.position;+= Vector3.Distance(startPosition, endPosition);

}distance;

}void RotateIntoMoveDirection() {newStartPosition = waypoints [currentWaypoint].transform.position;newEndPosition = waypoints [currentWaypoint + 1].transform.position;newDirection = (newEndPosition - newStartPosition);x = newDirection.x;y = newDirection.y;rotationAngle = Mathf.Atan2 (y, x) * 180 / Mathf.PI+90;sprite = (GameObject).transform.FindChild("Sprite").gameObject;.transform.rotation = .AngleAxis(rotationAngle, Vector3.forward);

}

}

)

PlaceMonster.csUnityEngine;System.Collections;UnityEngine.UI;class PlaceMonster : MonoBehaviour {GameObject monsterPrefab;GameObject monster;GameManagerBehavior gameManager;Text test;

// Use this for initializationStart () {=.Find("GameManager").GetComponent<GameManagerBehavior>();

}

// Update is called once per frameUpdate () {

//monsterPrefab=gameManager.TowerPref;

}bool canPlaceMonster(){cost = monsterPrefab.GetComponent<MonsterData> ().levels[0].cost;monster == null && gameManager.Gold >= cost && monsterPrefab != null;

}OnMouseUp () {= gameManager.TowerPref;(canPlaceMonster ()) {

//if(monster==null && monsterPrefab!=null){= (GameObject)(monsterPrefab, transform.position,

Quaternion.identity);audioSource = gameObject.GetComponent<AudioSource> ();.PlayOneShot (audioSource.clip);

gameManager.Gold -= monster.GetComponent<MonsterData> ().CurrentLevel.cost;

} else (canUpgradeMonster ()) {(monster != null) {

/*test.text="asa";.SetActive(true);.transform.position=new Vector3(2,2);*/.GetComponent<MonsterData> ().increaseLevel ();audioSource = gameObject.GetComponent<AudioSource> ();.PlayOneShot (audioSource.clip);.Gold -= monster.GetComponent<MonsterData> ().CurrentLevel.cost;

}

}

}bool canUpgradeMonster() {cost = monsterPrefab.GetComponent<MonsterData> ().levels[0].cost;(monster != null) {monsterData = monster.GetComponent<MonsterData> ();nextLevel = monsterData.getNextLevel();(nextLevel != null) {gameManager.Gold >= nextLevel.cost;

}

}false;

}

}

)

ShootEnemies.csUnityEngine;System.Collections;System.Collections.Generic;class ShootEnemies : MonoBehaviour {List<GameObject> enemiesInRange;float lastShotTime;MonsterData monsterData;Start () {= new List<GameObject>();= Time.time;=

gameObject.GetComponentInChildren<MonsterData> ();

}Update () {target = null;minimalEnemyDistance = float.MaxValue;(GameObject enemy in enemiesInRange) {distanceToGoal = enemy.GetComponent<MoveEnemy>().distanceToGoal();(distanceToGoal < minimalEnemyDistance) {= enemy;= distanceToGoal;

}

}(target != null) {(Time.time - lastShotTime > monsterData.CurrentLevel.fireRate)
{(target.GetComponent<Collider2D>());= Time.time;

}direction = gameObject.transform.position - target.transform.position;.transform.rotation = Quaternion.AngleAxis(.Atan2 (direction.y, direction.x) * 180 / Mathf.PI,Vector3 (0, 0, 1));

}

}OnEnemyDestroy (GameObject enemy) {.Remove (enemy);

}OnTriggerEnter2D (Collider2D other) {(other.gameObject.tag.Equals("Enemy")) {.Add(other.gameObject);del =.gameObject.GetComponent<EnemyDestructionDelegate>();.enemyDeleg ate += OnEnemyDestroy;

}

}OnTriggerExit2D (Collider2D other) {(other.gameObject.tag.Equals("Enemy")) {.Remove(other.gameObject);del =.gameObject.GetComponent<EnemyDestructionDelegate>();.enemyDeleg ate -= OnEnemyDestroy;

}

}Shoot(Collider2D target) {bulletPrefab = monsterData.CurrentLevel.bullet; startPosition = gameObject.transform.position;targetPosition = target.transform.position;.z

= bulletPrefab.transform.position.z;.z = bulletPrefab.transform.position.z;newBullet = (GameObject)Instantiate (bulletPrefab);.transform.position = startPosition;bulletComp = newBullet.GetComponent<BulletBehavior>();.target = target.gameObject;.startPosition = startPosition;.targetPosition = targetPosition;animator = .CurrentLevel.visualization.GetComponent<Animator> ();.SetTrigger ("fireShot");audioSource = gameObject.GetComponent<AudioSource>();.PlayOneShot(audioSource.clip);

}

}

) SpawnEnemy.csUnityEngine;System;System.Collections;

[System.Serializable]class Wave {GameObject[] enemyPrefab;float spawnInterval = 2;int maxEnemies = 20;

}class SpawnEnemy : MonoBehaviour {Wave[] waves;int timeBetweenWaves = 5;GameManagerBehavior gameManager;float lastSpawnTime;int enemiesSpawned = 0;GameObject[] waypoints;

// Use this for initializationStart () {

//Instantiate(testEnemyPrefab).GetComponent<MoveEnemy>().waypoints = waypoints;= Time.time;

//gameManager =GameObject.Find("GameManager").GetComponent<GameManagerBehavior>();

}

// Update is called once per frameUpdate () {currentWave = gameManager.Wave;.Random rng=new System.Random();number=rng.Next

```
(0, waves [currentWave].enemyPrefab.Length);(currentWave < waves.Length) {timeInterval = Time.time - lastSpawnTime;spawnInterval = waves[currentWave].spawnInterval;(((enemiesSpawned == 0 && timeInterval > timeBetweenWaves) ||
        timeInterval > spawnInterval) &&
        enemiesSpawned < waves[currentWave].maxEnemies) { = Time.time;newEnemy = (GameObject)(waves[currentWave].enemyPrefab[number]);.GetComponent <MoveEnemy>().waypoints = waypoints;++;
        }(enemiesSpawned == waves[currentWave].maxEnemies &&
        GameObject.FindGameObjectWithTag("Enemy") == null) {.Wave++;.Gold = Mathf.RoundToInt(gameManager.Gold * 1.1f);= 0;= Time.time;
        }
        } else {.gameOver = true;gameOverText = GameObject.FindGameObjectWithTag ("GameWon");.GetComponent<Animator>().SetBool("gameOver", true);
        }
        }
        }
```