Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий

Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой
_____О.В. Непомнящий
«___»_____2023 г.

БАКАЛАВРСКАЯ РАБОТА

090301.62 Информатика и вычислительная техника

Мобильная 2D-Игра

Руководитель _____ _____ доцент, канд. физ.-мат. наук К.В. Коршун
подпись *дата* *должность, ученая степень*

Выпускник _____ _____ Р.А. Злобин
подпись *дата*

Нормоконтролёр _____ _____ доцент, канд. физ.-мат. наук К.В. Коршун
подпись *дата* *должность, ученая степень*

Красноярск 2023

Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий

Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой
_____О.В. Непомнящий
«___»_____2022 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
в форме бакалаврской работы

Красноярск 2022

Студенту Злобину Ростиславу Андреевичу
фамилия, имя, отчество

Группа ЗКИ18-07Б Направление (специальность) 09.03.01.62
номер код

Информатика и вычислительная техника
наименование

Тема выпускной квалификационной работы: Мобильная 2D-Игра

Утверждена приказом по университету № 7667/С от 17.05.2023 г.

Руководитель ВКР: К.В. Коршун, канд. физ.-мат. наук, доцент каф. ВТ
инициалы, фамилия, учёная степень, должность, место работы

ИКИТ СФУ

Исходные данные для ВКР:

1) среда разработки Unity;

2) операционная система Windows 10;

3) язык программирования C#.

Перечень разделов ВКР:

1) анализ предметной области;

2) разработка мобильной 2D-Игры;

3) тестирование и отладка игрового процесса;

Перечень графического материала: Презентация в программе Microsoft PowerPoint

Руководитель ВКР _____ Коршун К.В
подпись инициалы, фамилия

Задание принял к исполнению _____ Злобин Р.А
подпись инициалы, фамилия

« » _____ 2022 г.
дата

РЕФЕРАТ

Выпускная квалификационная работа по теме «Мобильная 2D-Игра» содержит 52 страниц текстового документа, 32 иллюстрации, 1 таблицу и 10 использованных источников.

UNITY, C#, ДВИЖОК, КЛАССЫ, АНИМАЦИЯ, ИГРОВОЙ ИНТЕРФЕЙС, МАШИНА СОСТОЯНИЙ, ИГРОВАЯ МЕХАННИКА.

Объектом исследования данной области является игровой движок Unity и язык программирования C#.

Целью данной работы является создание полноценной игры с интерфейсом, анимациями, механикой противника и игрока, также сборка проекта и тестирование готового продукта.

В процессе работы был произведён анализ теоретических сведений по взаимодействию с игровым движком, а так же базовый кодинг языка программирования C#.

Результатом работы является мобильная игра с подробным описание структуры, анимации, механик и алгоритмов работы.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы обусловлена тем что, мобильные игры на данный момент являются наиболее популярными и прибыльными во всей игровой индустрии.

Цель работы — разработка мобильной 2D-Игры.

Задачи работы:

1. Анализ предметной области.
2. Разработка игровых ресурсов.
3. Разработка программного кода.
4. Разработка интерфейса игры.
5. Отладка и тестирование.

В результате выполнения выпускной квалификационной работы получена однопользовательская игра с интерфейсом, графикой и механикой персонажа и противников.

1 Анализ предметной области

1.1 Обзор игрового движка

Существует два основных вида компьютерных игр: 2D и 3D. 2D игры, как правило, имеют более простую графику и геймплей, и часто являются более доступными для начинающих игроков. Они могут быть представлены в различных жанрах, таких как платформеры, файтинги, головоломки и т.д.

3D игры, с другой стороны, используют более продвинутую графику и часто более сложный геймплей. Они могут быть представлены в различных жанрах, таких как шутеры от первого лица, ролевые игры, гонки и т.д. Они также могут быть более реалистичными и иметь более открытый мир для исследования.

Мобильные 2D игры популярны благодаря нескольким причинам [2]:

– часто более доступны и просты в использовании, чем более сложные 3D игры;

– многие 2D игры могут быть быстро созданы с помощью инструментов, таких как движки для игр, что делает их более доступными для малых команд разработчиков;

– 2D игры могут обладать уникальным и привлекательным художественным стилем, что делает их более запоминающимися и привлекательными для игроков;

– многие 2D игры могут быть более аркадными и быстрыми, что может быть более привлекательным для игроков, которые хотят быстро получить удовольствие от игры.

Большая часть игр разрабатывается на движке Unity. Unity – это кроссплатформенный игровой движок, который используется для разработки мобильных игр [4]. Он позволяет создавать игры для различных платформ, включая iOS, Android и Windows Phone.

Одной из особенностей Unity является его простота в использовании. Интерфейс разработки довольно интуитивно понятный, что позволяет быстро создавать игровые объекты и элементы. Кроме того, движок имеет обширную документацию и сообщество разработчиков, которые готовы помочь и поделиться своим опытом.

Одним из преимуществ Unity является его графический движок. Он позволяет создавать высококачественную графику и спецэффекты, что делает игры, созданные на этой платформе, очень привлекательными для пользователей.

Кроме того, Unity имеет обширный набор инструментов для создания анимаций и физики, что позволяет создавать интересные и реалистичные игровые механики. Наконец, Unity поддерживает различные языки программирования, включая C# и JavaScript. Это позволяет разработчикам использовать тот язык, который им наиболее удобен. В целом, Unity – это мощный и удобный инструмент для разработки мобильных игр, который позволяет создавать высококачественные игры для различных платформ [1].

На Unity написаны сотни игр и приложений, платформа используется такими крупными разработчиками, как Blizzard и Epic Games. По данным американских игровых порталов, в 2021 году почти 50% всех платных игр, вышедших в Steam, были сделаны на Unity, в области мобильных игр эта цифра превысила 50%. А самыми популярными из этих игр неизменно являются те, которые используют в своей графике 2D- и 3D-ассеты.

Ниже рассмотрены одни из самых популярных 2D игр, созданных на Unity**.**

В данной работе я использовал такие средства, как Visual Studies для написания самого кода на C#, так как он хорошо связан с игровым движком Unity.

Игровой движок Unity [9] для создания локаций, меню, персонажа и противников, игрового интерфейса, анимаций и управления персонажа, а также отладки игры. Язык программирования использовался C# [1].

1.2 Обзор компьютерных игр на Unity

В данном разделе будут перечислены имеющиеся на рынке игры, а также их механики и способы взаимодействия с игроком, интерфейс, сюжет, и история разработки.

1.2.1 Hollow Knight

«Hollow Knight» – это увлекательная игра, которая была создана небольшой австралийской студией Team Cherry. Суть игры заключается в том, что игроку предстоит управлять насекомым–рыцарем, который путешествует по подземному миру Холлоунест, населенному различными существами и переполненному опасностями. В игре есть элементы платформера, головоломок и битв с боссами [7].

Сюжет игры разворачивается в фэнтезийном мире, где насекомые являются главными персонажами. Игроку предстоит исследовать мир, сражаться с монстрами, собирать ресурсы и развивать своего персонажа. В игре много секретов и тайн, которые нужно раскрыть, чтобы продвигаться дальше по сюжету.

Игра разрабатывалась более трех лет, начиная с 2014 года, и была выпущена в 2017 году на платформах Windows, macOS, Linux и Nintendo Switch. Разработчики использовали движок Unity для создания игры, что позволило им создать уникальный и привлекательный визуальный стиль. В игре прекрасно сочетаются темные и светлые цвета, что создает особую атмосферу.

Одним из главных преимуществ «Hollow Knight» является ее высокая вариативность развития сюжета. Игроки могут исследовать мир в своем темпе и выбирать разные пути, что создает множество вариантов прохождения игры. Кроме того, в игре есть различные концовки, которые зависят от действий игрока и выбора пути. Скриншот из уровня игры «Hollow Knight» представлен на рисунке 1.

Игра получила высокие оценки от критиков за ее уникальный художественный стиль, геймплей и музыку. Она также была успешной коммерчески и получила несколько дополнений и расширений.

Рисунок 1 – Скриншот из уровня игры «Hollow Knight»

1.2.2 Cuphead

«Cuphead» – это 2D платформер, который был создан студией «StudioMDHR» и выпущен в 2017 году. Игроки управляют персонажем по имени Капхэд и его братом Магманом, которые должны сражаться с боссами в различных уровнях, чтобы вернуть свои души, которые они продали Дьяволу [7].

Одна из уникальных особенностей игры – ее визуальный стиль, который был вдохновлен американскими анимационными короткометражными фильмами 1930-х годов. Игра была создана с использованием традиционных методов анимации и рисования вручную, чтобы создать уникальный визуальный стиль игры.

Разработка игры «Cuphead» заняла более 6 лет, и студия столкнулась с множеством трудностей на пути к ее созданию. Помимо финансовых проблем они также столкнулись с проблемами, связанными с производством анимации, так как были использованы традиционные методы анимации вместо компьютерной графики.

Однако все это усилие и труды окупились, поскольку игра стала популярной среди игроков и критиков благодаря своей уникальной графике и сложному геймплею. Игра «Cuphead» была номинирована на множество премий и получила множество наград, включая награду The Game Awards за «Лучший арт-дизайн» и «Лучшую музыку/звуковое оформление».

Игра имеет более 20 уровней, каждый из которых представляет собой уникальный вызов для игроков. В «Cuphead» есть множество боссов, которые являются главными противниками игроков на каждом уровне. Каждый босс имеет уникальный дизайн и неповторимые атаки, что делает игру более интересной и разнообразной.

Кроме того, в игре есть возможность играть вдвоём. Это делает игру еще более веселой, особенно для тех, кто любит играть со своими друзьями. На рисунке 2 представлен скриншот из игры «Cuphead».

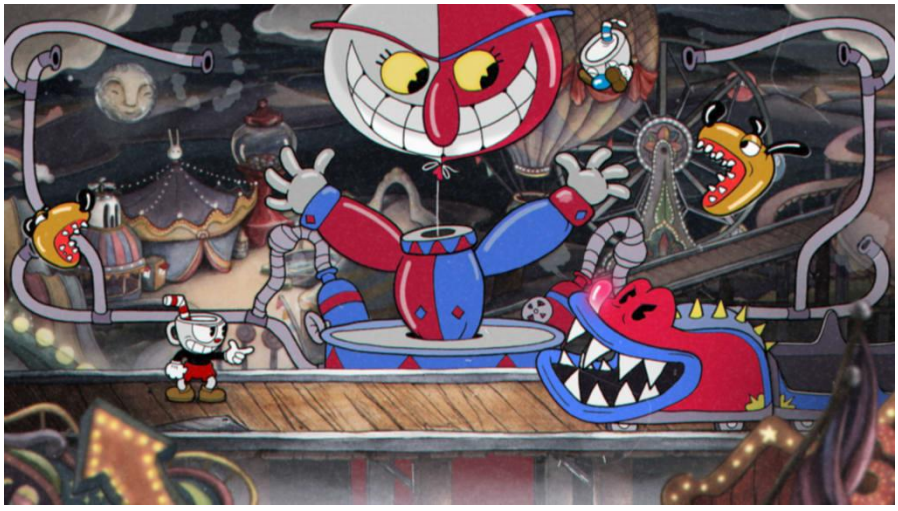


Рисунок 2 – Скриншот из игры Cuphead

1.2.3 Ori and the Blind Forest

«Ori and the Blind Forest» – это 2D игра с захватывающим сюжетом и интуитивно понятным геймплеем. Игроки влюбляются в эту игру с первых минут игрового процесса благодаря уникальной атмосфере и стилю, который ей присущ [7].

Игра рассказывает историю маленького существа по имени Ори, которое живет в лесу, где оно сталкивается с различными опасностями и проблемами. Ори должен найти способ спасти свой лес и своих друзей от зла, которое начинает распространяться по местности.

В процессе игры игроку приходится исследовать различные уровни, бороться с врагами и решать головоломки. Игровой процесс очень хорошо сбалансирован, и он предоставляет игроку множество интересных вызовов, которые не дадут ему заскучать.

Одним из главных преимуществ игры является ее красивая графика. Игра использует яркие и насыщенные цвета, что делает ее визуально привлекательной и запоминающейся. Кроме того, музыкальное сопровождение игры также заслуживает внимания, так как оно помогает создать уникальную атмосферу в игре.

«Ori and the Blind Forest» была создана студией Moon Studios с использованием движка Unity, который позволяет разработчикам создавать красивые и уникальные игры в 2D формате. Игра была выпущена в 2015 году на Xbox One и PC и быстро стала одной из самых популярных игр в своем жанре.

На рисунке 3 показан скриншот из игры «Ori and the Blind Forest».

Рисунок 3 – Скриншоты из игры «Ori and the Blind Forest»

1.3 Общее описание разрабатываемой игры

Сюжет игры заключается в том, что главный персонаж защищает свой дом от набегов противников.

Правила игры:

– у персонажей должны быть характеристики такие как: здоровье, наносимый урон, скорость бега и атаки у противников. Эти характеристики должны изменяться в игровом движке, а не в коде;

– у каждого персонажа должна быть определённая анимация будь то бег, атака;

– персонажи должны получать урон друг от друга и умирать;

– должны быть несколько волн противников;

– противники должны выбирать цель и бежать к ней;

– должно быть оружие которое можно будет добавить и использовать.

Целью у каждого персонажа должны быть свои, у игрока это победить все волны противников, а у врага дойти до игрока и убить его.

1.4 Выводы

В аналитической части были рассмотрены три популярные 2D игры: «Hollow Knight», «Cuphead», «Ori and the Blind Forest». Каждая из игр имеет свой уникальный дизайн, логику игры, интересные уровни, анимацию и музыкальное сопровождение.

В дальнейшем на основе вышеперечисленных игр будет создаваться собственная 2D-Игра, которая будет заимствовать механики стрельбы, анимаций состояний противника и игрока, взаимодействия игрока с интерфейсом. Программный код будет написан на языке программирования C#. а также использоваться встроенные средства разработки игрового движка Unity.

2 Разработка мобильной 2D-Игры

2.1 Правила и принципы игры

Для разработки игры нужны определенные правила и законы, по которым она будет работать.

– игрок – для правильного функционирования персонажа нужно чтобы у него было определённое количество здоровья, оружие из которого он будет стрелять во врагов, урон который он может нанести своим оружием, также игрок в случае проигрыша должен пропадать с поля и появляться большая надпись «ПРОИГРАЛ»;

– противник – правила для противника более обширные, чем для игрока, так как на нём будет завязана большая часть геймплея. Противник, как и игрок должен иметь определённое количество здоровья, он должен выбирать цель в нашем случае это игрок, противник должен двигаться к своей цели с определённой скоростью и анимацией и когда противник достигает определённого расстояния он должен начать атаковать с заданным уроном, а если противник уничтожен он должен исчезать и за него начисляются монеты;

– шкалы здоровья и волн – шкала волн должна показывать то, сколько противников появилось и движется к игроку во время определённой волны. Шкала здоровья должна показывать, сколько здоровья у игрока в настоящий момент;

– появление противников – первая волна противников должна появляться автоматически, а последующие посредством нажатия кнопки «NextWave», а также волны должны быть нарастающие, а именно больше противников и меньше времени на появление;

– меню – должно быть паузой и способом выхода из игры;

– магазин – в магазине должен быть ассортимент оружия со своим названием ценой и картинкой при покупке монеты должны тратиться и показывать купленный предмет затемнением цены, а крестик в правом верхнем углу дол-

жен закрывать магазин, также как и меню, магазин должен ставить игру на паузу.

2.2 Разработка объектов игры

2.2.1 Разработка карты и декораций

Для создания карты использовались 2 заранее заготовленных спрайта фона, горы и лес (Рисунки 4, 5).

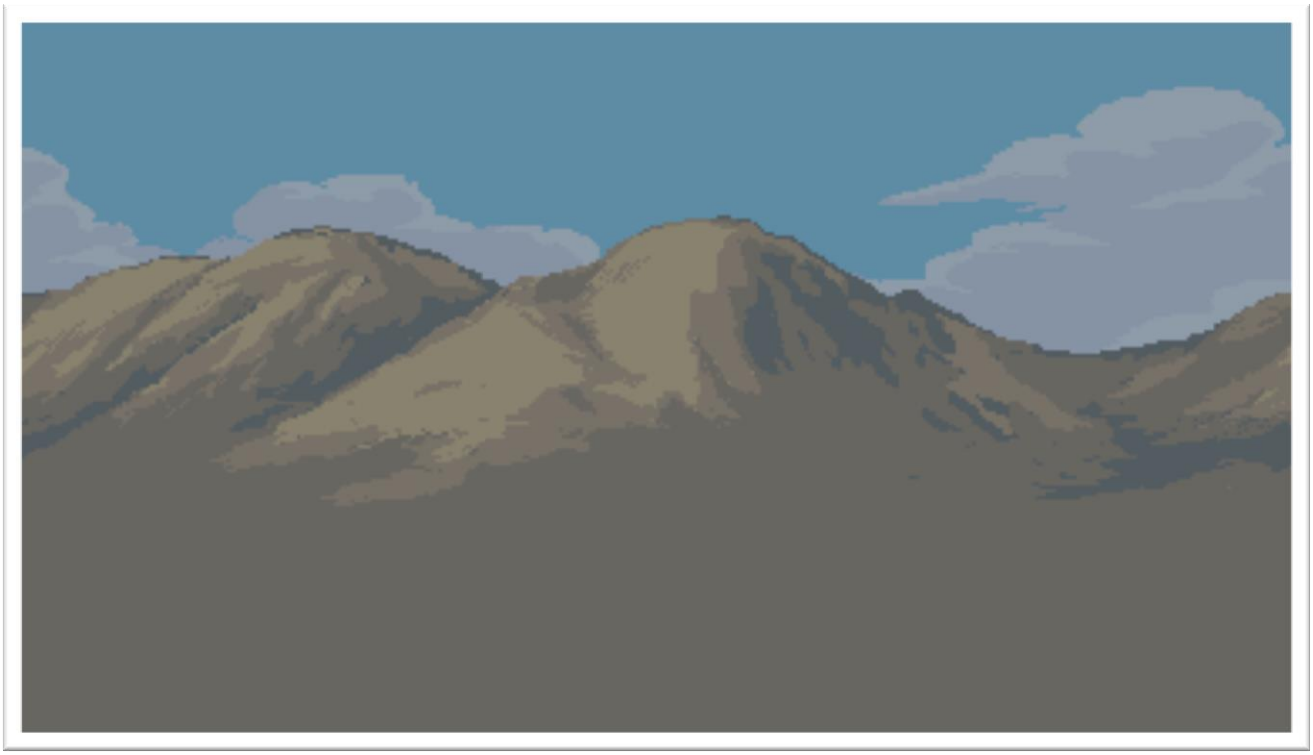


Рисунок 4 – Фон горы

Для того чтобы горы и лес смотрелись органично необходимо настроить слои которые будут отображать их, в нашем случае это -10 для гор и -9 для леса. Так же желательно убрать у обоих изображений все фильтры, чтобы они были более пиксельные, что соответствует стилю игры.

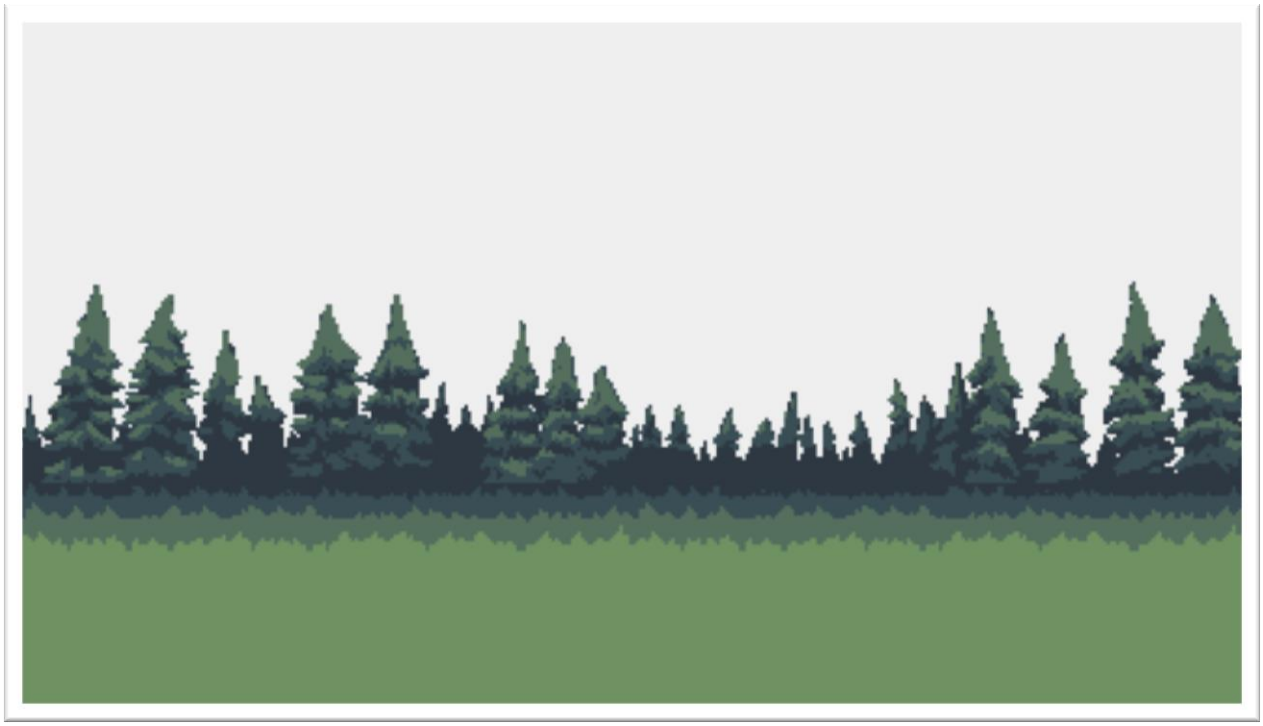


Рисунок 5 – Фон лес

При настройке фона с помощью картинок нужно учитывать некоторые настройки:

– Pixel Per Unit (Пиксель на единицу) – эта настройка отвечает сколько пикселей будет на 1 Unit, в нашем случае нужно 20 Unit, тогда картинка будет в хорошем формате и не была размыта;

– Filter mode – эта настройка отвечает за сглаживание кадра, в нашем случае т.к у нас 2D игра мы убираем эти фильтры и ставим пиксельную картинку;

– Order in Layer – данная настройка отвечает за слой кадра на сцене, так как фон должен быть позади то и его число должно быть меньше всех остальных (Рисунок 6).

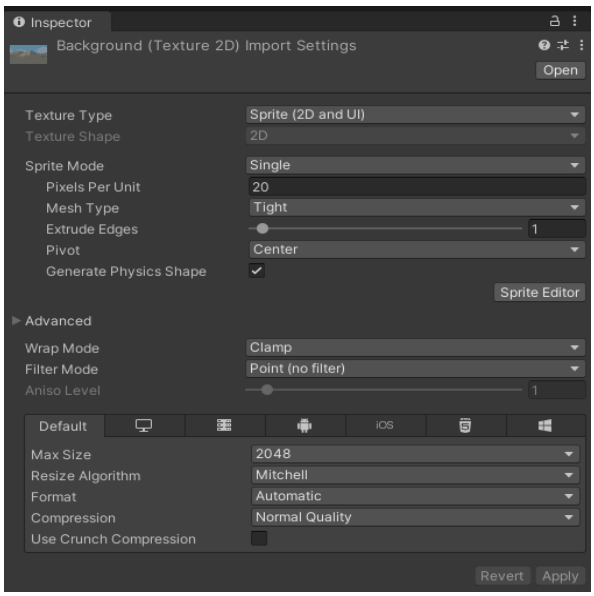


Рисунок 6 – Окно настройки основы картинки

В Unity есть такой инструмент как Tilemap, он позволяет создавать объекты сцены и задавать им определённые физические свойства, его удобство заключается в том, что модель ландшафта можно использовать как кисть на холсте, для этого в TileMap предусмотрена сетка размером в 1 Unit, где и будут закрашиваться элементы ландшафта.

Для того чтобы работать с Tilemap нам необходим такой инструмент как Tilepalette (Рисунок 7), он отвечает за сами элементы ландшафта и используется как кисти для хоста.

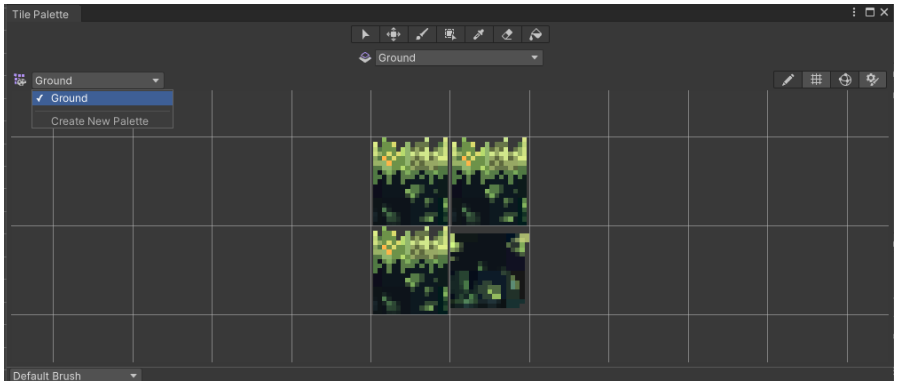


Рисунок 7 – Окно Tilepalette

В данной работе будет использовано всего 3 вида декораций это Дом, Деревья, и указатель (Рисунок 8).

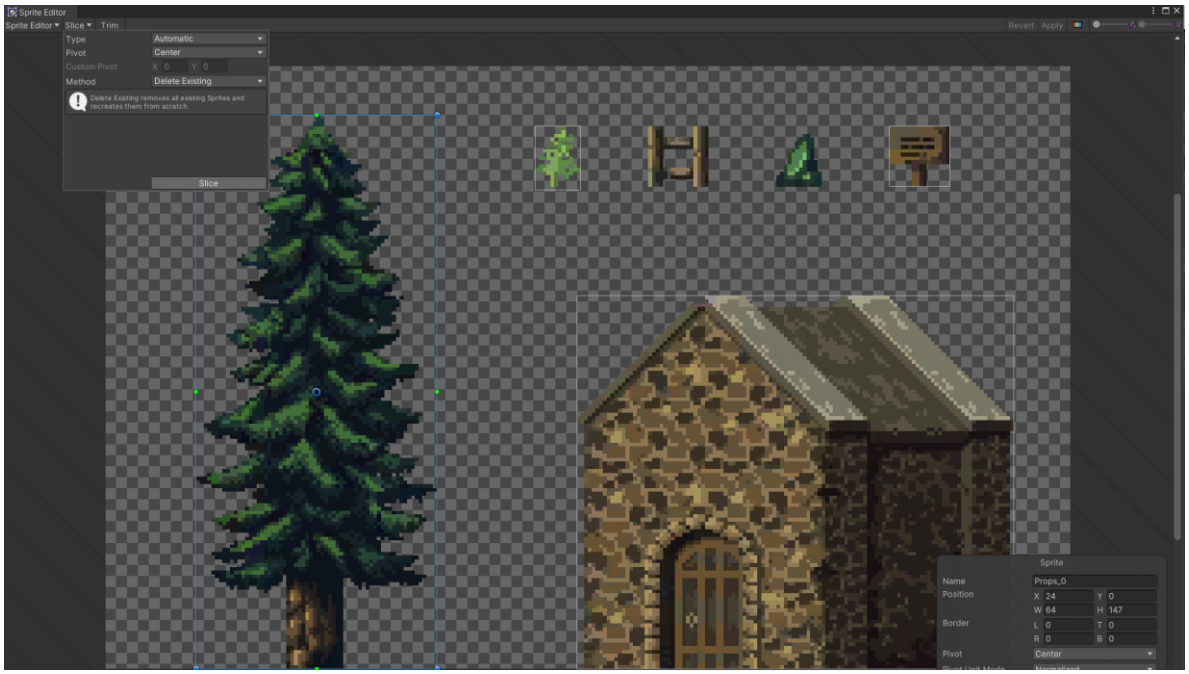


Рисунок 8 – Окно Sprite Editor

После нарезки спрайтов они появятся во всплывающем окне в ресурсах игры.

2.2.2 Разработка анимации

Создание анимации занимает обширный пласт работы в играх и делается несколькими способами в зависимости от исходных данных, которые есть. На примере данной игры можно рассмотреть несколько вариантов создания анимации.

Первый пример это покадровая анимация. Для того чтобы создать анимацию нам необходимо зайти в специальный инструмент по созданию анимации Animation (Рисунок 12).

Для того чтобы сделать анимацию необходимо выбрать необходимый нам объект и сохранить его в нужную директорию, далее к этому объекту можно

будет реализовать множество различных анимаций. Для примера реализуем по кадровую анимацию стрельбы.

Чтобы реализовать по кадровую анимацию нам необходимо спрайт персонажа с разными позициями на каждый момент времени (Рисунок 9).

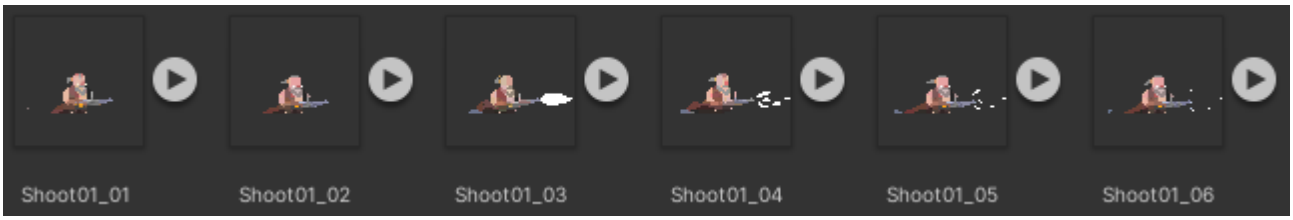


Рисунок 9 – Кадры анимации стрельбы в каждый момент времени

Далее мы выбираем все эти модели и переносим их в окно Animation и программа сама ставит анимацию в нужный момент времени, потом в настройках нужно поставить время для выполнения всей анимации, чтобы это сделать нужно двойным нажатием в директиве открыть создавшуюся анимацию, после чего откроется настройка всех анимаций, связанных с объектом (Рисунок 10).

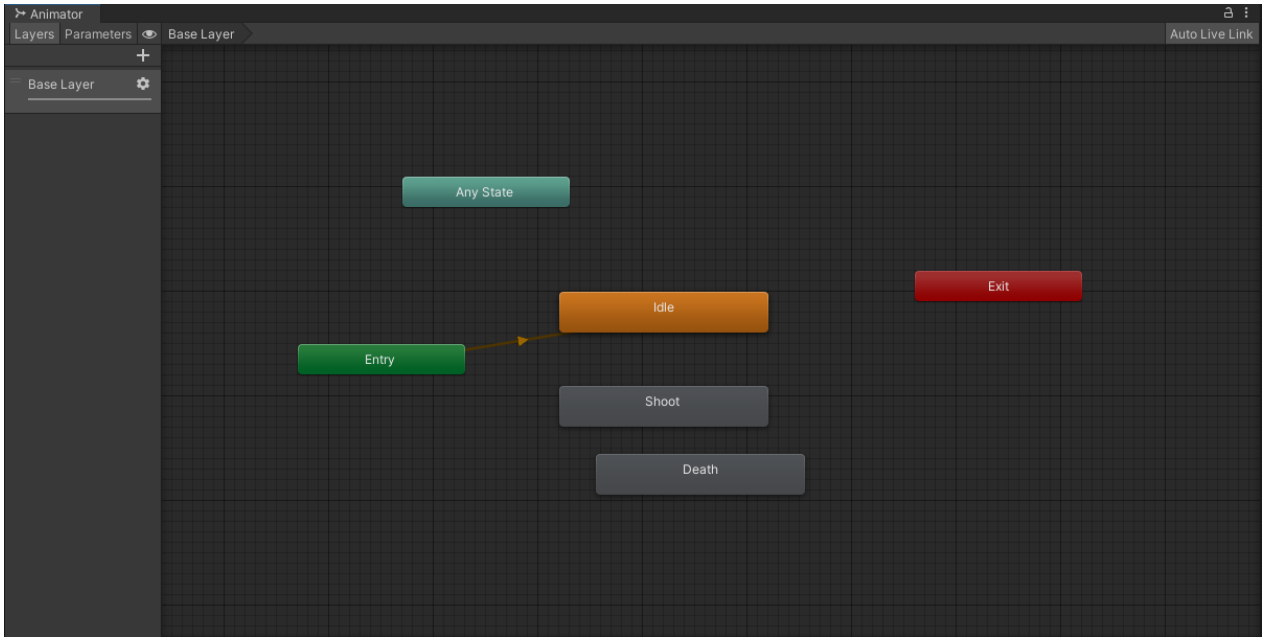


Рисунок 10 – Окно настройки анимации

После этого нам необходимо выбрать нужную нам анимацию и установить необходимое время проигрывания анимации (Рисунок 11).

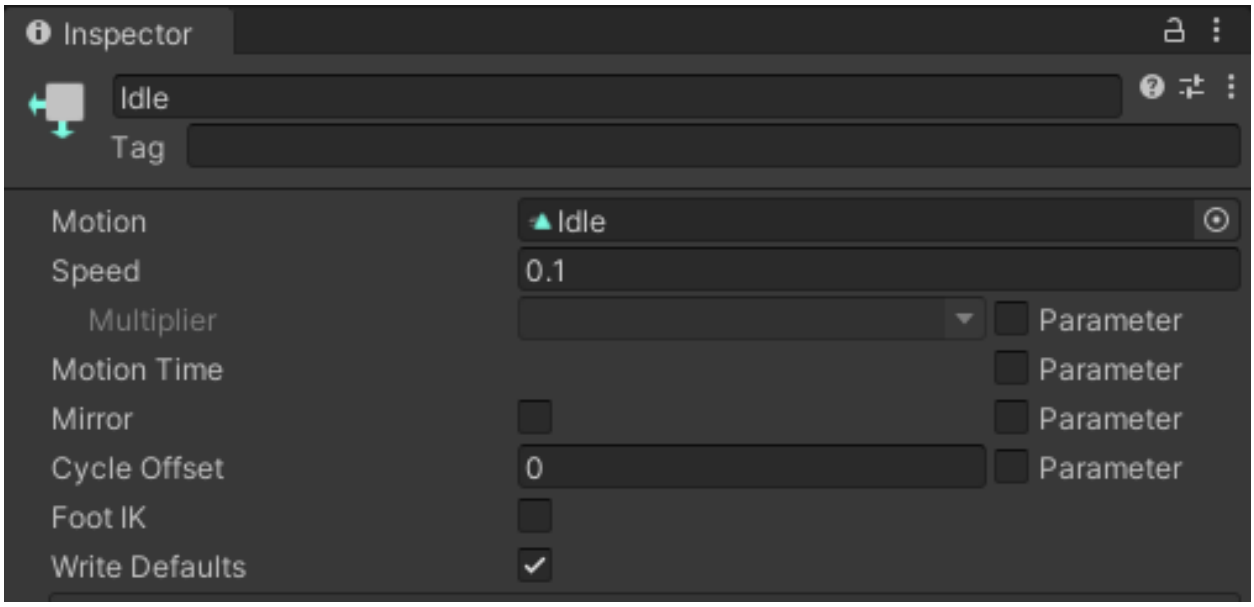


Рисунок 11 – Установка параметров для анимации

Вторым примером будет по-костная анимация, она реализована на примере дерева, изначально дерево было статичным, а нам необходимо, чтобы оно было в движении.

Для того чтобы сделать костную анимации нужно зайти в Sprite Editor картинки которую хотим анимировать, далее в меню выбрать раздел Skinning Editor и выбрать нужную нам модель, и нанести необходимые нам кости которые буду гнуть наше дерево пример (Рисунок 12).

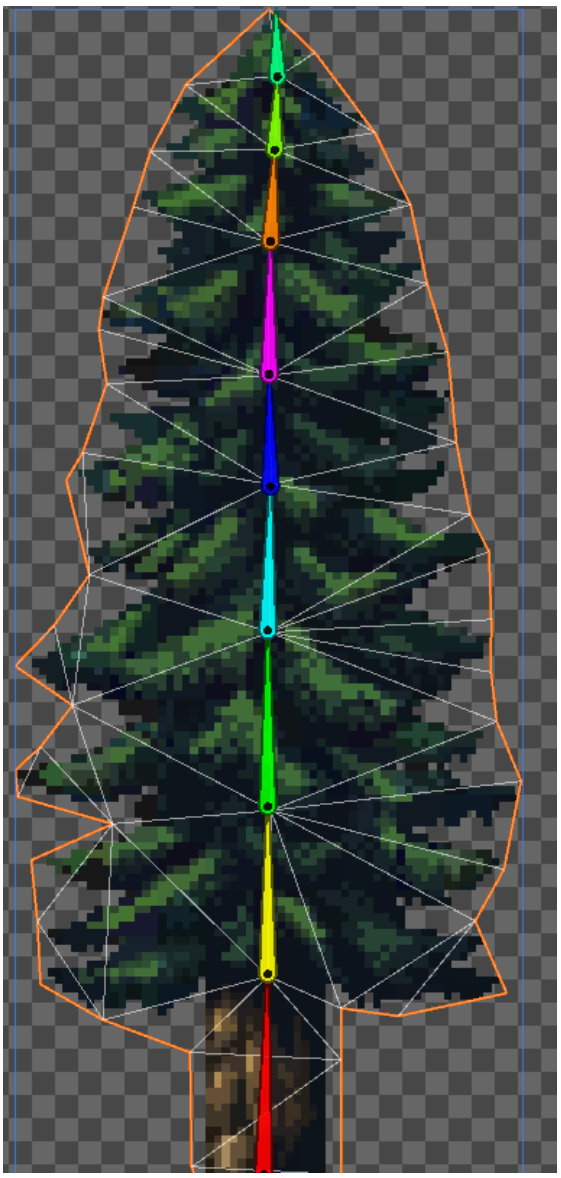


Рисунок 12 – Дерево с наложенными костями

Дальше остаётся только сделать по кадровую анимацию аналогично с вышеуказанным примером.

2.3 Разработка программного кода

2.3.1 Диаграмма наследования классов

Диаграмма наследования классов включает следующие функции [8] (Приложение А):

Встроенным классом в Unity является MonoBehaviour у него есть встроенные функции, которые позволяют настраивать проект и управлять прорисовкой кадров так как нужно разработчику.Start: Функция Start вызывается до обновления первого кадра (first frame) только если скрипт включен;

– OnEnable: (вызывается только если объект активен) – эта функция вызывается сразу после включения объекта. Это происходит при создании образца MonoBehaviour, например, при загрузке уровня или был вызван GameObject с компонентом скрипта;

– OnDisable: эта функция вызывается, когда объект отключается или становится неактивным;

– Update: вызывается раз за кадр. Это главная функция для обновлений кадров;

– Awake: эта функция всегда вызывается до любых функций Start и также после того, как префаб был вызван в сцену (если GameObject неактивен на момент старта, Awake не будет вызван, пока GameObject не будет активирован, или функция в каком-нибудь прикреплённом скрипте не вызовет Awake);

– Reset: (сброс) вызывается для инициализации свойств скрипта, когда он только присоединяется к объекту и тогда, когда используется команда Reset;

– OnDestroy: эта функция вызывается после всех обновлений кадра в последнем кадре объекта, пока он ещё существует (объект может быть уничтожен при помощи Object.Destroy или при закрытии сцены).

Во всех этих функциях представлены основные инструменты для настройки кода для игры.

2.3.2 Создание игровых персонажей

2.3.2.1 Разработка игрока

Первым делом нужно создать объект для самого персонажа, для этого нужно перенести исходную модель персонажа на сцену и настроить его размер и слой расположения.

Можно представить игрока в виде конечного автомата (Рисунок 13)

Рисунок 13 – Конечный автомат персонажа

– Shooting: состояние стрельбы игрока при нажатии на экран телефона;

– Calmness: состояние покоя для игрока;

– Getting coins: начисление монет на баланс;

– Death: состояние проигрыша игрока.

После того как визуальная часть была сделана и разработана схема работы персонажа нужно написать функционал персонажа в виде кода представленный в приложении Б.

В данном коде указан весь функционал персонажа, а именно – уровень здоровья персонажа который уменьшается благодаря методу ApplyDamage, текущее оружие, атаку персонажа и текущий баланс.

2.3.2.2 Разработка противника

Для настройки поведения персонажа в данной работе была написана машина состояний персонажа, который выполняет весь функционал противника (Приложение Б).

В машине состояний (Приложение Б) указаны все состояния которые может принимать противник (Рисунок 14) NoMove – это начальное состояние врага которое он имеет при старте игры, далее противник выбирает свою цель в нашем случае это персонаж игрока и переходит в состояние Move, т.к противников несколько первый из них может победить противника и те кто не дошёл до него могу остановиться т.к цель исчезла, в случае если противник приблизился к игроку машина переходит в состояние Attack до тех порт пока или у врага или у игрока не будет ниже 0 здоровья, в таком случае машина перейдёт в 1 из двух состояний Winner или Death.

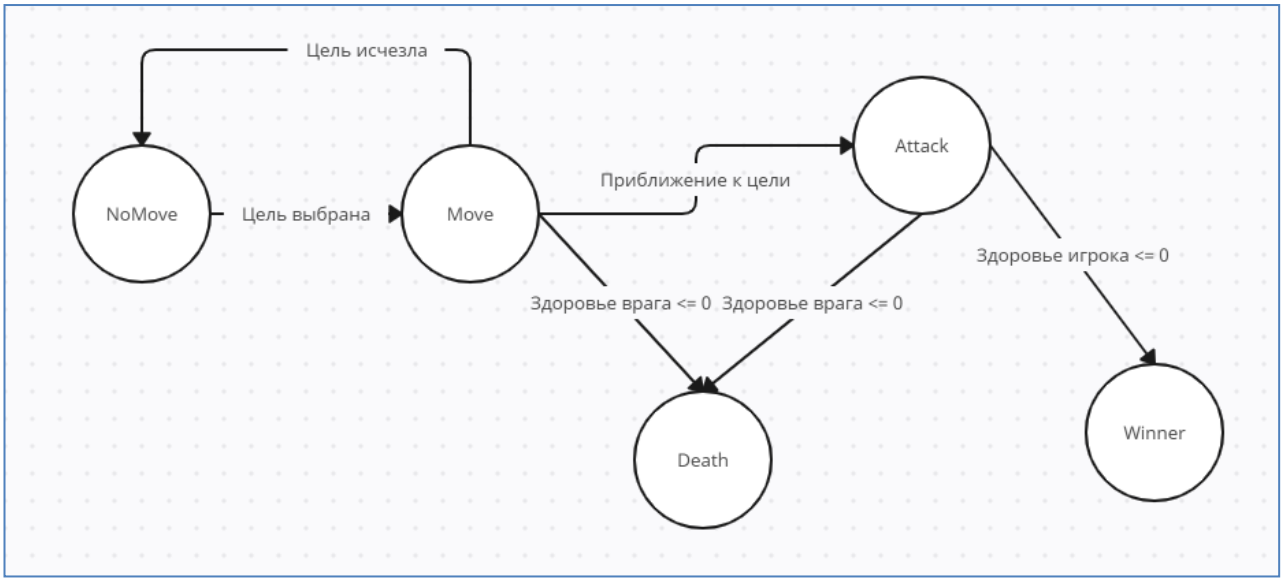


Рисунок 14 – Конечный автомат машины состояний противника

– NoMove: состояние покоя противника, когда нет цели или она не выбрана;

– Move: состояние пути к цели, когда она появилась;

– Attack: состояние атаки, когда противник приблизился к цели на расстояние, отведённое в настройках;

– Death: состояние смерти, когда у противника здоровья меньше одной единицы;

– Winner: состояние победы, когда у цели меньше одной единицы здоровья.

Для того чтобы выбрать точку появления противников нам необходимо создать новый пустой объект и вставить в него данный код, в дальнейшем на сцене разместить его туда, откуда нам нужно чтобы шли враги.

2.3.3 Разработка оружия

Данный код будет реализованы как арсенал оружия, который можно расширять до бесконечности, но для примера я приведу 1 оружие пистолет.

В коде ниже указаны основные данные об оружии, а именно название, стоимость, иконка оружия и есть ли у нас в наличии это оружие (Рисунок 15).

```
7       [SerializeField] private string _label;
8       [SerializeField] private int _price;
9       [SerializeField] private Sprite _icon;
10      [SerializeField] private bool _isBuyed = false;
11
12      [SerializeField] protected Bullet Bullet;
```

Рисунок 15 – Описание оружия

В коде ниже мы просчитываем полёт пули до объекта и удаляем его, когда он достигает цели (Рисунок 16).

```
8   public class Bullet : MonoBehaviour
9   {
10      [SerializeField] private int _damage;
11      [SerializeField] private float _speed;
12
            Сообщение Unity | Ссылок: 0
13      private void Update()
14      {
15          transform.Translate(Vector2.left * _speed * Time.deltaTime, Space.World);
16      }
17
            Сообщение Unity | Ссылок: 0
18      private void OnTriggerEnter2D(Collider2D collision)
19      {
20          if (collision.gameObject.TryGetComponent(out Enemy enemy))
21          {
22              enemy.TakeDamage(_damage);
23
24              Destroy(gameObject);
25          }
26      }
27  }
```

Рисунок 16 – Присчитывание траектории пули и её уничтожение при столкновении.

У нашей пули есть своя модель и для того чтобы не выносить на сцену пулю, а вызывать её при команде игрока используются так называемые Prefab это что-то вроде шаблона с помощью которого можно изменить все дочерние объекты так и изменить 1 объект не трогая остальные. Чтобы создать Prefab, нужно перенести из иерархии объект в директорию, и он создастся автоматически.

2.4 Создание интерфейса

В данной работе был создан объект, который отвечает за игровой интерфейс, кнопки, шкалы и различные меню.

Для того чтобы создать этот интерфейс нам необходим новый объект Canvas, в котором мы и будем делать весь интерфейс игры.

В Canvas входит множество элементов интерфейса такие как: кнопки, кнопки с текстом, текст, панели на которых можно расположить объекты взаимодействия с игроком, расширяемое меню, различные индикаторы, иконки и заполняемые панели в зависимости от задач которые требуются.

2.4.1 Шкала здоровья и прогресса волн

Шкала прогресса и здоровья схожи по настройке в Unity, но имеют разный код. В нашем случае будут использоваться несколько объектов, такие как ScrollBar этот объект изменяется в зависимости от величины какой либо переменной будь-то здоровье или количество противников, нужен объект Image, который отвечает за визуальное представление интерфейса, чтобы нам понимать что это, здоровье или прогресс волн. Чтобы все объекты были, как единое целое нам необходимо делать их наследуемыми от одного объекта в нашем случае это Image ProgressBar (Рисунок 17).

Рисунок 17 – Иерархия объектов ProgressBar

По итогу всех манипуляций у нас должен получиться рабочая шкала прогресса здоровья (Рисунок 18) и прогресса (Рисунок 19).

Рисунок 18 – Шкала здоровья

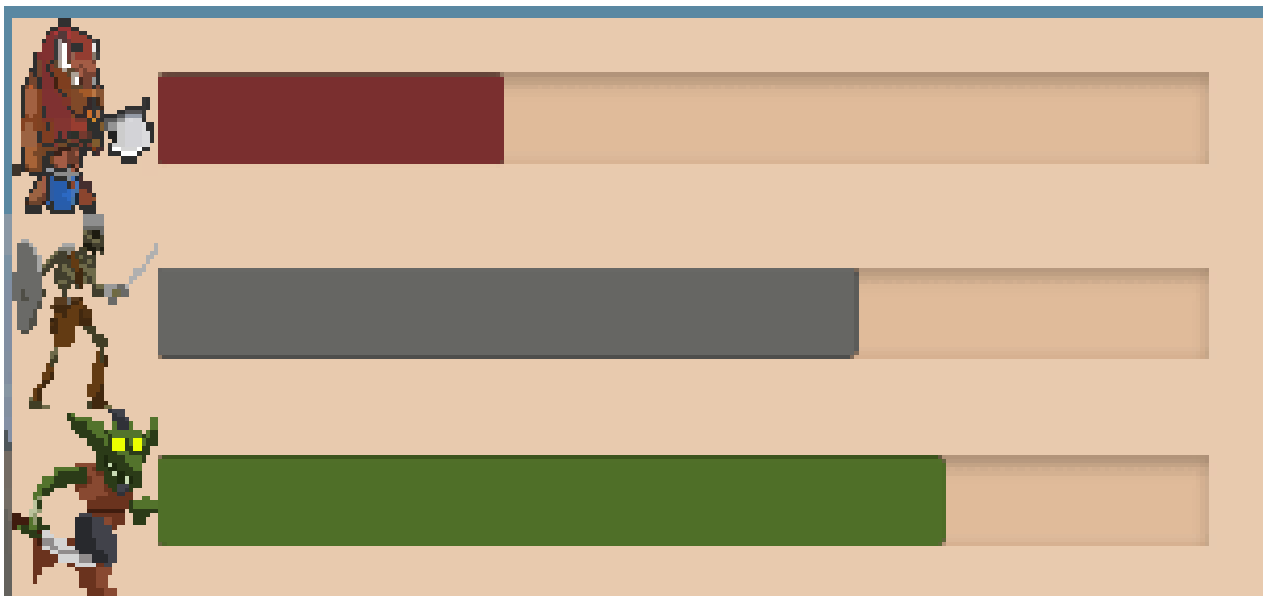


Рисунок 19 – Шкала прогресса волн

Данный фрагмент кода отвечает за уровень здоровья, отображаемый на шкале (Рисунок 20).

```
5   public class HealthBar : Bar
6   {
7       [SerializeField] private Player _player;
8
        Сообщение Unity | Ссылок: 0
9       private void OnEnable()
10      {
11          _player.HealthChanged += OnValueChanged;
12          Slider.value = 1;
13      }
14
        Сообщение Unity | Ссылок: 0
15      private void OnDisable()
16      {
17          _player.HealthChanged -= OnValueChanged;
18      }
19  }
```

Рисунок 20 – Отображение здоровья

Здесь указано значение шкалы прогресса волны (Рисунок 21).

```
5   public class ProgressBar : Bar
6   {
7       [SerializeField] private Spawner _spawner;
8
        Сообщение Unity | Ссылок: 0
9       private void OnEnable()
10      {
11          _spawner.EnemyCountChanged += OnValueChanged;
12          Slider.value = 0;
13      }
14
        Сообщение Unity | Ссылок: 0
15      private void OnDisable()
16      {
17          _spawner.EnemyCountChanged -= OnValueChanged;
18      }
19  }
```

Рисунок 21 – Отображение прогресса противников

Разница между здоровьем и прогрессом только в том, что у прогресса берётся количество противников, основываясь на длине массива который создаёт объект Spawner, а здоровье в свою очередь на значении здоровья игрока.

2.4.2 Меню и магазин

Для создания меню нам необходим материнский класс, от которого все будут наследоваться в нашем случае это обычный Image в который мы, и будем вставлять необходимые нам кнопки (Рисунок 22).

Рисунок 22 – Кнопки продолжения, новая игра и выхода.

Функции, которые указаны в классе меню также можно применять и к другим окнам интерфейса, чтобы останавливать игру и выполнять различные манипуляции. В данном коде реализованы функции открытия меню посредством нажатия кнопки Menu в интерфейсе (Рисунок 23).

```
6   public class Menu : MonoBehaviour
7   {
        Ссылок: 0
8       public void NewGame()
9       {
10          SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
11          Time.timeScale = 1;
12      }
        Ссылок: 0
13      public void OpenPanel(GameObject panel)
14      {
15          panel.SetActive(true);
16          Time.timeScale = 0;
17      }
18
        Ссылок: 0
19      public void ClosePanel(GameObject panel)
20      {
21          panel.SetActive(false);
22          Time.timeScale = 1;
23      }
24
        Ссылок: 0
25      public void Exit()
26      {
27          Application.Quit();
28      }
29  }
```

Рисунок 23 – Функции основного меню

А закрытия меню и продолжение игры осуществляется за счёт нажатия на кнопку Continue (Рисунок 24), для того чтобы нам выйти из игры и закрыть приложение необходимо нажать на кнопку Exit (Рисунок 24).

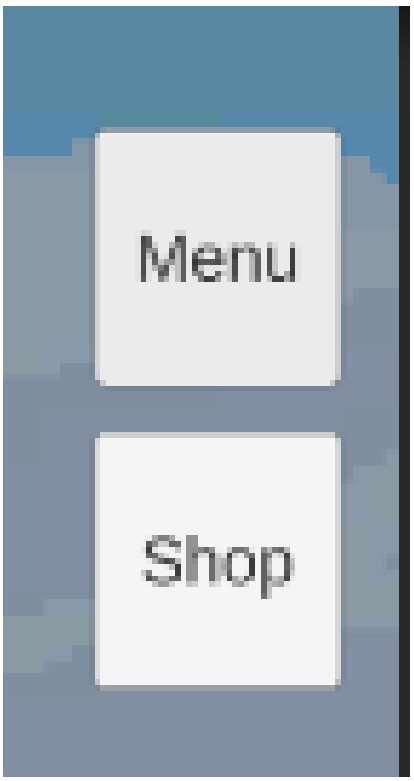


Рисунок 24 – Кнопки меню и магазина в интерфейсе игры

Магазин в свою очередь осуществляется немного сложнее, чем меню, так как в нем реализовано больше элементов, таких как Иконка закрытия магазина, монеты начисляемы за убийства противников, а также сам ассортимент магазина.

Монеты схожи по принципу со шкалами, так как и там и там идёт заполнение только в этом случае меняется число и только (Рисунок 25). Данный код отвечает за начисление и трату денег (Рисунок 26).

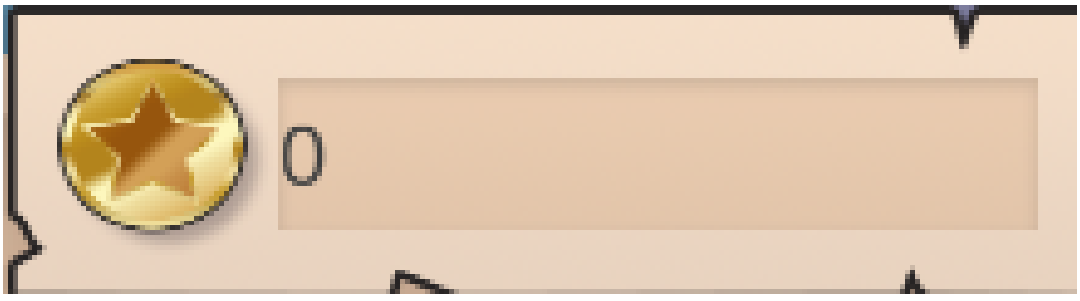


Рисунок 25 – Баланс на счету для покупки в магазине

```
6   public class MoneyBalance : MonoBehaviour
7   {
8       [SerializeField] private TMP_Text _money;
9       [SerializeField] private Player _player;
10
        Сообщение Unity | Ссылок: 0
11      private void OnEnable()
12      {
13          _money.text = _player.Money.ToString();
14          _player.MoneyChanged += OnMoneyChanged;
15      }
16
        Сообщение Unity | Ссылок: 0
17      private void OnDisable()
18      {
19          _player.MoneyChanged -= OnMoneyChanged;
20      }
21
        Ссылок: 2
22      private void OnMoneyChanged(int money)
23      {
24          _money.text = money.ToString();
25      }
26  }
```

Рисунок 26 – Начисление монет за победу над противником

Далее нам необходимо меню, в котором будет сам ассортимент, и чтобы этот ассортимент мог пополняться, нам необходим определённый объект ScrollView, он позволяет делать так, чтобы в случае пополнения магазина можно было листать и покупать необходимое оружие (Рисунок 27).

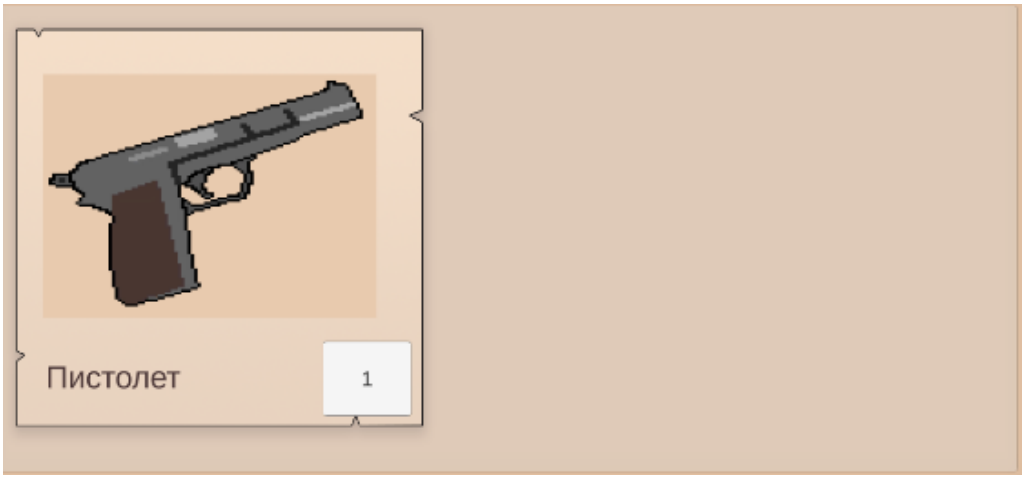


Рисунок 27 – Меню ассортимента в магазине

Для того чтобы нам не пришлось руками добавлять новое оружие в случае пополнения арсенала необходимо оружие с его ценами, названием и иконкой добавить в Prefab, это позволит пополнять магазин без лишьних манипуляций с кодом игры.

Для того чтобы сделать ассортимент нам необходимо внутри Content (Рисунок 28) создать объект который будет продаваться, соответственно для него мы создаём иконку, название, а также кнопку покупки SellButton.

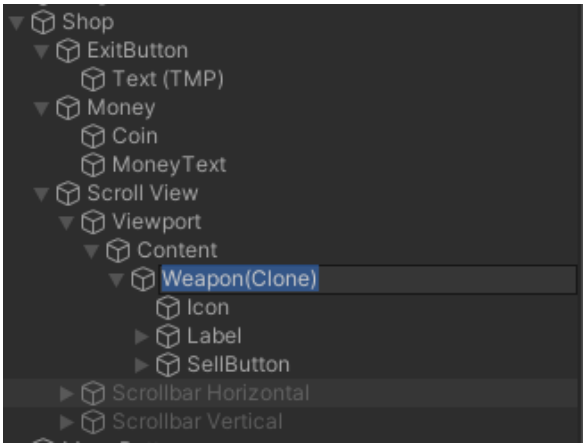


Рисунок 28 – Иерархия магазина

В данном коде указаны все данные по оружию в магазине, а именно: его цена, куплено оно или нет, иконка, а также функция для его покупки (Рисунок 29).

```
8   public class WeaponView : MonoBehaviour
9   {
10      [SerializeField] private TMP_Text _label;
11      [SerializeField] private TMP_Text _price;
12      [SerializeField] private Image _icon;
13      [SerializeField] private Button _sellButton;
14
15      private Weapon _weapon;
16
17      public event UnityAction<Weapon, WeaponView> SellButtonClick;
18
            Сообщение Unity | Ссылок: 0
19      private void OnEnable()
20      {
21          _sellButton.onClick.AddListener(OnButtonClick);
22          _sellButton.onClick.AddListener(TryLockItem);
23      }
24
            Сообщение Unity | Ссылок: 0
25      private void OnDisable()
26      {
27          _sellButton.onClick.RemoveListener(OnButtonClick);
28          _sellButton.onClick.RemoveListener(TryLockItem);
29      }
30
            Ссылок: 2
31      private void TryLockItem()
32      {
33          if (_weapon.IsBuyed)
34              _sellButton.interactable = false;
35      }
36
            Ссылок: 1
37      public void Render(Weapon weapon)
38      {
39          _weapon = weapon;
40
41          _label.text = weapon.Label;
42          _price.text = weapon.Price.ToString();
43          _icon.sprite = weapon.Icon;
44      }
45
            Ссылок: 2
46      private void OnButtonClick()
47      {
48          SellButtonClick?.Invoke(_weapon, this);
49      }
50  }
```

Рисунок 29 – Данные по оружию в магазине

Так же нам нужна кнопка выхода из магазина, которая является обычным объектом Button без текста (Рисунок 30).

Рисунок 30 – Кнопка выхода из магазина

По итогу у нас получается полноценное меню магазина, которое можно пополнять различным оружием (Рисунок 31).

Рисунок 31 – Меню магазина

2.5 Выводы

В данной главе мы разобрали и сделали большую часть аспектов разработки мобильных игр, таких как анимация, настройка сцены, программирования, с помощью которого мы взаимодействуем со всеми объектами в игре.

Также мы использовались с основными механиками мобильных игр, например: машина состояний, которая отслеживает то, как должен себя вести персонаж в определённый промежуток времени, создания игрового персонажа и его умения, появление противников и количество которое нужно нам для полноценной игры.

Также мы разобрались с основными инструментами движка Unity, таких как: Аниматор, нарезка спрайтов, настройка фона, управления объектами и создание шаблонов для дальнейшего использования.

В следующей главе мы разберем, как нам настроить и собрать проект, а также протестируем готовую игру на мобильном устройстве.

3 Тестирование и отладка игрового процесса

3.1 Сборка

Сборка в Unity делается за счёт встроенных функций движка и не требует кода или иных манипуляций.

Для того чтобы начать сборку нам необходимо выбрать в меню File функцию Build Settings, в окне нам необходимо выбрать платформу под которую мы будем собирать проект (Рисунок 32). В нашем случае Android.

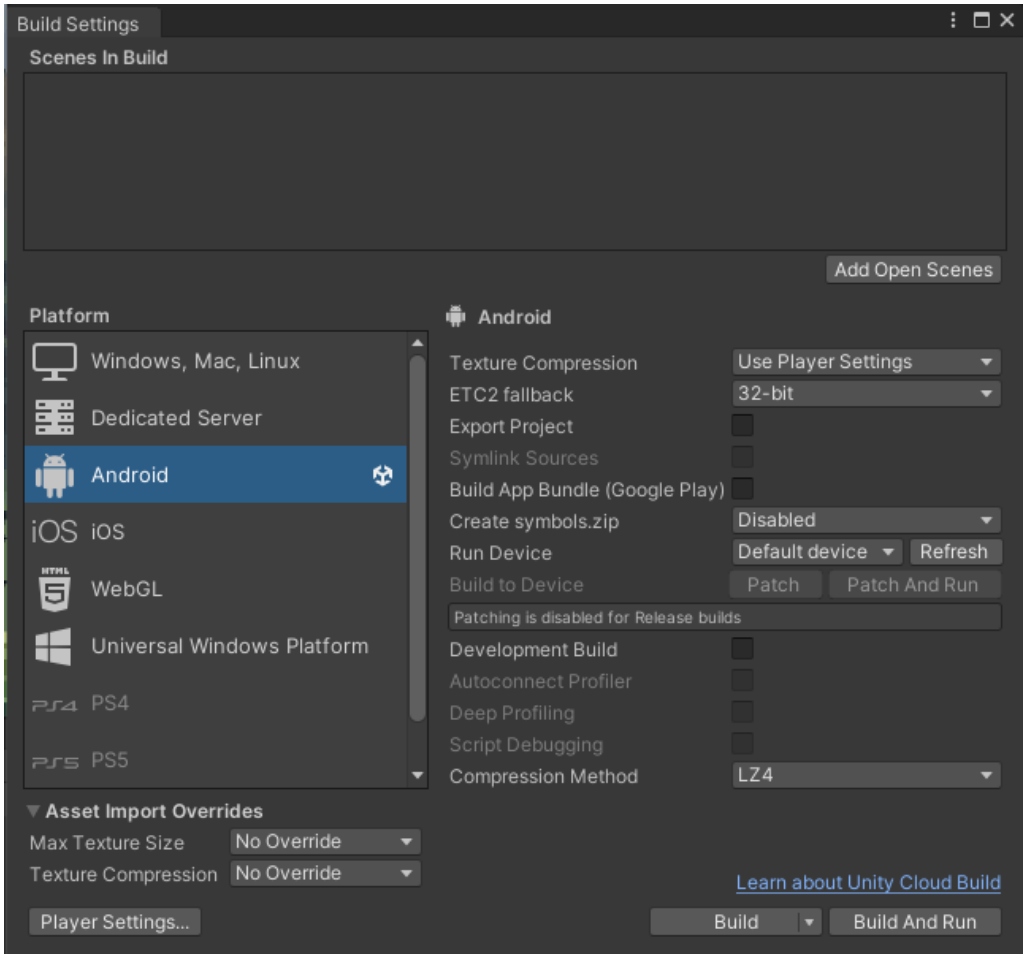


Рисунок 32 – Окно сборки проекта

Далее нам необходимо перейти в раздел «Player Settings» (Рисунок 33) и в нём задать необходимые настройки, а именно название компании, название

продукта, версия сборки и иконка, которая будет отображаться на экране телефона.

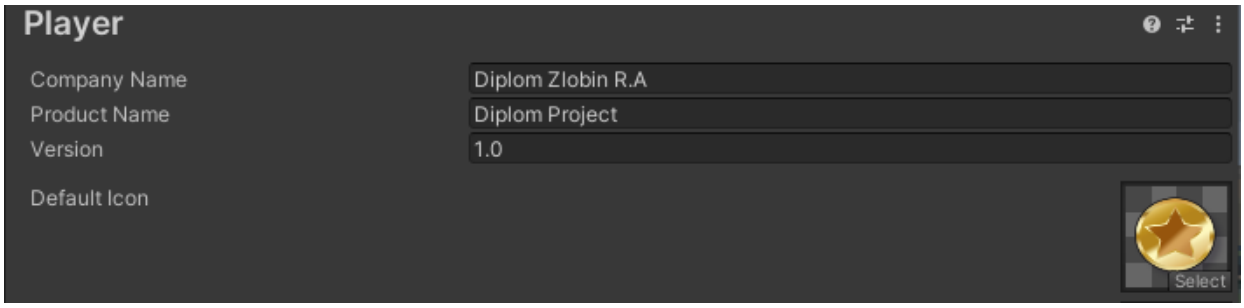


Рисунок 33 – Окно «Player Settings»

Дальше нам необходимо настроить то, как игра будет представлена на экране телефона, за это отвечает меню Resolution and Presentation (Рисунок 34).

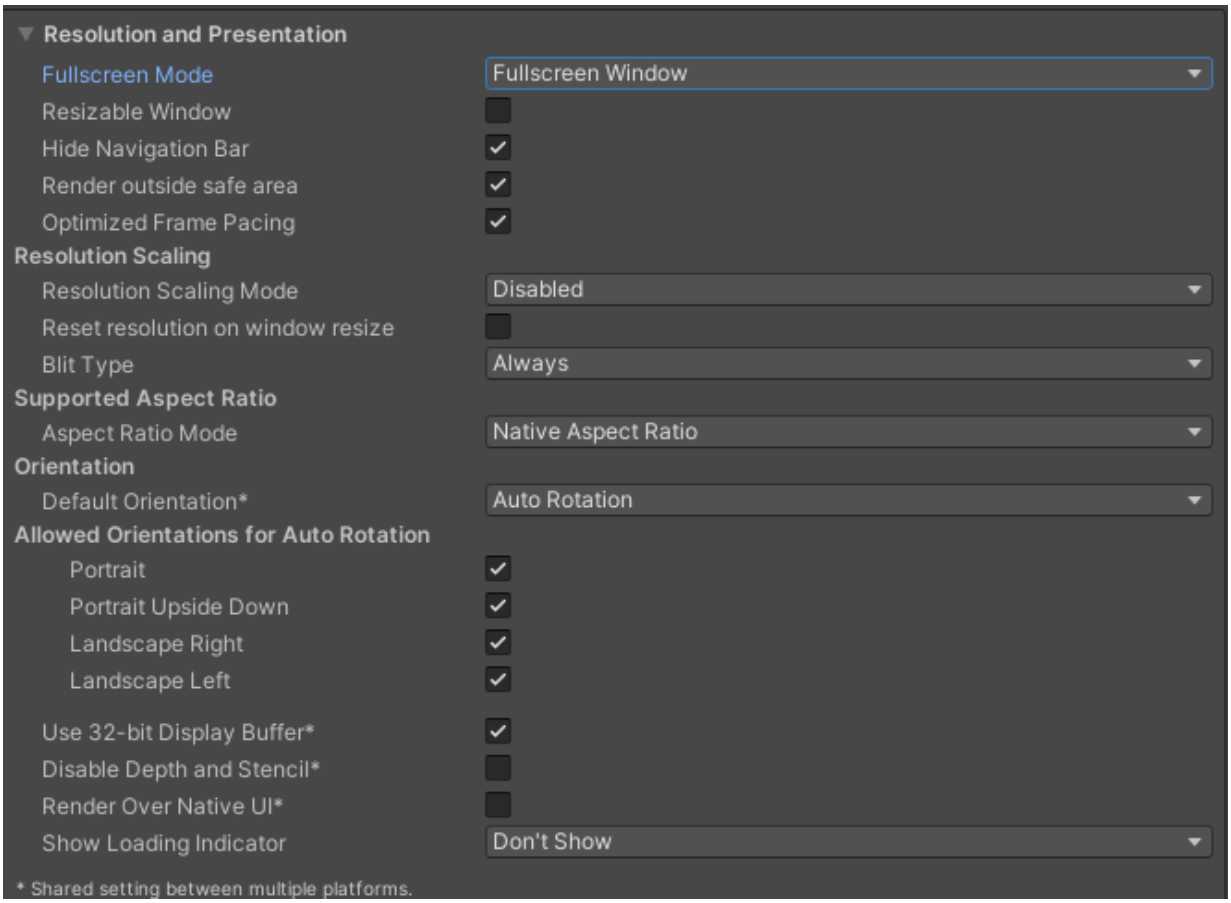


Рисунок 34 – Окно Resolution and Presentation

В этих настройках нам необходимо выбрать Fullscreen Mode, так как наша игра будет во весь экран.

Resizable Window эта настройка отвечает за то будет ли игра происходить в зоне «Чёлки» на современных смартфонах, она нам не нужна поэтому отключаем эту настройку.

Далее идёт настройка Default Orientation – эта настройка отвечает за то, будет ли наша игра менять своё положение при повороте экрана телефона, так как у нас игра будет в альбомном режиме то нужно ставить настройку Landscape Right.

Настройка Splash Image (Рисунок 35) – данная настройка отвечает за то, как нас будет встречать игра, т.к. у нас бесплатная версия Unity мы не можем убрать стандартную заставку, но можем добавить свой логотип в самом начале.

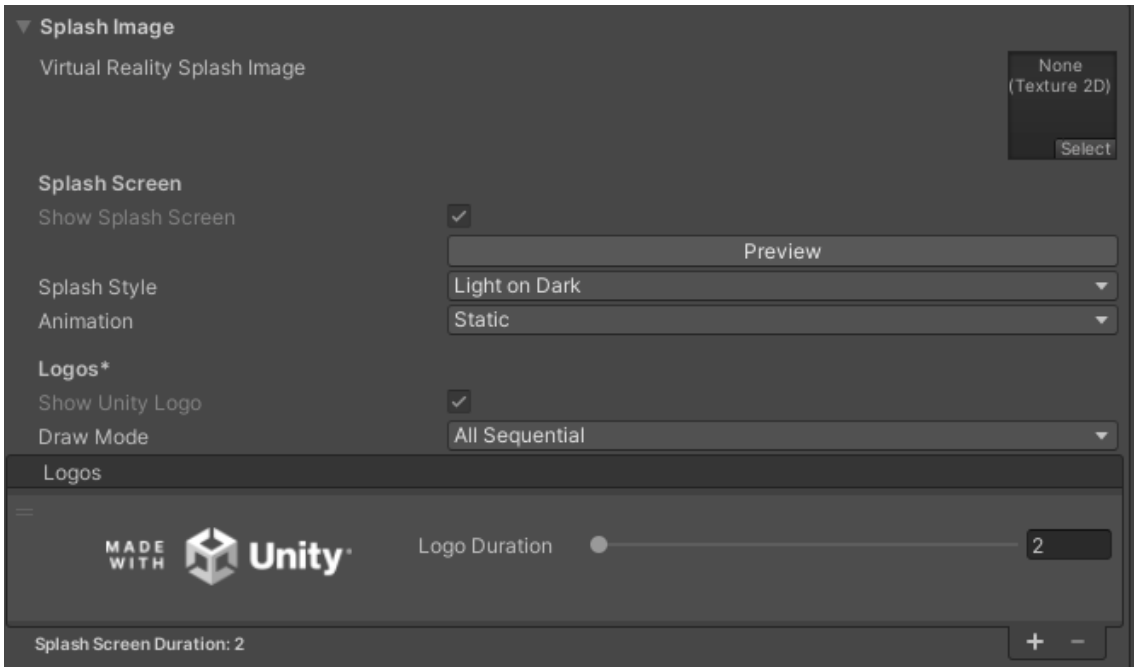


Рисунок 35 – Окно Splash Image

В нашей игре после всех этих процедур можно закрывать настройки сборки для игры.

Далее нам остаётся только нажать кнопку Build и выбрать папку, куда мы хотим сохранить игру и после этого игра соберётся полностью.

3.2 Тестирование

Тестирование игры начинается с запуска игры на смартфоне, далее я проверил стрельбу по противникам и нанесение им урона, далее я проверил получение урона игроком, а также его проигрыш далее я зашел в магазин и проверил то начисляются ли монетки, а также тратятся ли они на покупку оружия, потом я проверил то вызывается ли следующая волна противников и сколько их с какой скоростью они появляются, последним я проверил меню и то дают ли оно паузу и выходи из игры. Соблюдение правил игры представлено в таблице 1.

Таблица 1 – Соблюдение правил игры

Законы игры	**Выполнение закона**
Игрок	Наличие здоровья (Да), Наличие Денег (Да), Пополнение баланса (Да), Наличие Оружия (Да), Потеря здоровья при получении урона (Да), Исчезновение с карты при проигрыше (Да), Появление надписи(Да)
Противник	Наличие здоровья (Да), Выбор цели (Да), Движение к цели с анимацией движения (Да), Атака на определённом расстоянии заданным уроном (Да), Исчезновение при смерти (Да),
Шкала волн и здоровья	Количество врагов в определённой волне (Да), Здоровье в реальном времени(Да)
Появление противников	Первая волна автоматически (Да), Следующая волна сильнее, чем предыдущая (Да), Волна вызывается по нажатию кнопки (Да)
Меню	Меню ставит игру на паузу (Да), Можно выйти из игры (Да)
Магазин	У оружия есть Название (Да) Иконка (Да) Цена (Да), Монеты тратятся при покупке Оружия (Да), Цена затемняется при покупке (Да), Можно выйти из магазина нажав на крестик (Да), Пауза (Да)

3.3 Результат

В результате сборки и тестирования не было выявлено, каких либо ошибок, игра исправна и готова к использованию (Таблица 1).

ЗАКЛЮЧЕНИЕ

В выпускной квалификационной работе были поставлены и выполнены следующие задачи:

– произведён теоретический анализ предметной области, в ходе которой был выбран самый оптимальный из игровых движков и языков программирования, а также графической и звуковой составляющей;

– сформулированы методы разработки и отладки мобильной игры. Игра должна иметь в себе графический интерфейс для взаимодействия игры и игрока, механику игры завязанной на уничтожении множества противников, а также улучшения персонажа с помощью встроенных средств;

– в качестве программных средств был выбран игровой движок Unity, IDE Visual Studies и Blender для создания моделей персонажей и противников. В качестве языка программирования был выбран C#. Платформой для мобильной игры выбран Android.

В данной выпускной квалификационной работе была разработана мобильная игра под платформу Android с использованием C# и Unity. В ней будет использована собственная графика, а также графика с бесплатного ресурса.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Бонд, Д. Unity и C#. Геймдев от идеи до реализации / Д. Бонд ; пер. с англ. Киселёв А.Н. – Санкт-Петербург: Питер, 2019. – 928 с.: ил.

2. Гейг, М. Разработка игр на Unity 2018 за 24 часа / М. Гейг ; пер. с англ. Райтмана М. А. – Москва: Эксмо, 2020. – 464 с.: ил.

3. Ламмерс, К. Шейдеры и эффекты в Unity / пер. с англ. Е. А. Шапочкина, под редакцией Симонова В. В. – Москва : ДМК Пресс, 2014. – 274 с.

4. Меннинг, Д. Unity для разработчика. Мобильные мультиплатформенные игры / Д. Меннинг, П. Бадфилд-Эддисон ; пер. с англ. А. Киселева. – Санкт-Петербург : Питер, 2018. – 308 с.

5. Торн, А. Основы анимации Unity / А. Торн ; пер. с англ. Р. Н. Рагимова. – Москва: ДМК Пресс, 2016. – 176 с.: ил.

6. Хокинг, Д. Unity в действии. Мультиплатформенная разработка на C# / Д. Хокинг ; пер. с англ. И. Рузмайкиной. – Санкт-Петербург : Питер, 2016. – 336 с.

7. Топ лучших 2D и 3D игр на движке Unity : сайт. – URL: http://itmentor.by/articles/top-luchshih-2d-i-3d-igr-na-dvizhke-unity (дата обращения: 23.04.2023).

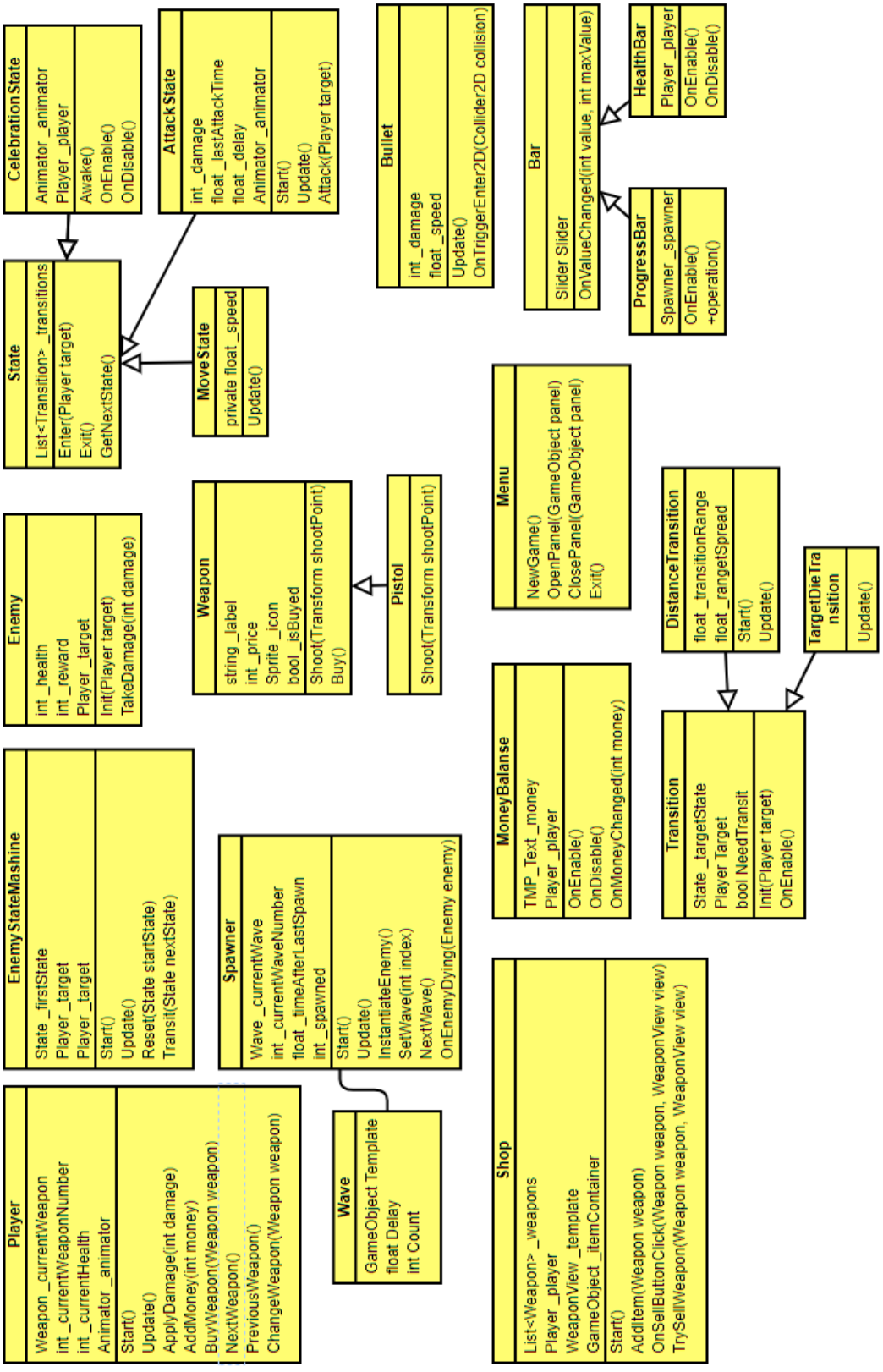
8. Unity – руководство Unity : сайт. – URL: https://docs.unity3d.com/ru/530/Manual/UnityManual.html (дата обращения: 18.03.2023).

9. Unity, платформа разработки в реальном времени : сайт. – URL: https://unity.com/ru (дата обращения: 15.01.2023).

10. UnityHub – руководство Unity на русском : сайт. – URL: https://unityhub.ru/manual/index (дата обращения: 09.04.2023).

ПРИЛОЖЕНИЕ А

Диаграмма классов



ПРИЛОЖЕНИЕ Б

Программный код игровых персонажей

Игрок:

```
1. using System;
2. using System.Collections.Generic;
3. using System.Threading;
4. using UnityEngine;
5. using UnityEngine.Events;
6.
7. [RequireComponent(typeof(Animator))]
8. public class Player : MonoBehaviour
9. {
10.     [SerializeField] private int _health;
11.     [SerializeField] private List<Weapon> _weapons;
12.     [SerializeField] private Transform _shootPoint;
13.     [SerializeField] private GameObject LoseText;
14.
15.     private Weapon _currentWeapon;
16.     private int _currentWeaponNumber = 0;
17.     private int _currentHealth;
18.     private Animator _animator;
19.
20.     public int Money { get; private set; }
21.
22.     public event UnityAction<int, int>
  HealthChanged;
23.     public event UnityAction<int> MoneyChanged;
24.
25.     private void Start()
26.     {
27.     ChangeWeapon(_weapons[_currentWeaponNumber]);
28.     _currentHealth = _health;
29.     _animator = GetComponent<Animator>();
30.     }
31.
32.     private void Update()
33.     {
34.     if (Input.GetMouseButtonDown(0))
35.     {
36.     _animator.Play("PlayerShot");
37.     _currentWeapon.Shoot(_shootPoint);
38.     }
39.     else if (Input.GetMouseButtonUp(0))
40.     {
41.     _animator.Play("PlayerIdle");
42.     }
43.     }
44.
45.     public void ApplyDamage(int damage)
```

```
46.     {
47.     _currentHealth -= damage;
48.     HealthChanged?.Invoke(_currentHealth, _health);
49.
50.     if (_currentHealth <= 0)
51.     {
52.     Destroy(gameObject);
53.     LoseText.SetActive(true);
54.     Time.timeScale = 0;
55.     }
56.     }
57.
58.     public void AddMoney(int money)
59.     {
60.     Money += money;
61.     MoneyChanged?.Invoke(Money);
62.     }
63.
64.     public void BuyWeapon(Weapon weapon)
65.     {
66.     Money -= weapon.Price;
67.     MoneyChanged?.Invoke(Money);
68.     _weapons.Add(weapon);
69.     }
70.
71.     public void NextWeapon()
72.     {
73.     if (_currentWeaponNumber == _weapons.Count - 1)
74.     _currentWeaponNumber = 0;
75.     else
76.     _currentWeaponNumber++;
77.
78.     ChangeWeapon(_weapons[_currentWeaponNumber]);
79.     }
80.
81.     public void PreviousWeapon()
82.     {
83.     if (_currentWeaponNumber == 0)
84.     _currentWeaponNumber = _weapons.Count - 1;
85.     else
86.     _currentWeaponNumber--;
87.
88.     ChangeWeapon(_weapons[_currentWeaponNumber]);
89.     }
90.
91.     private void ChangeWeapon(Weapon weapon)
92.     {
93.     _currentWeapon = weapon;
94.     }
95.     }
```

Противник:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

[RequireComponent(typeof(Animator))]
public class Enemy : MonoBehaviour
{
[SerializeField] private int _health;
    [SerializeField] private int _reward;

    private Player _target;
    private Animator _animator;

    public int Reward => _reward;
    public Player Target => _target;

    public event UnityAction<Enemy> Dying;

    public void Init(Player target)
    {
    _target = target;
    }

    public void TakeDamage(int damage)
    {
    _health -= damage;

    if (_health <= 0)
    {
    Dying?.Invoke(this);
    Destroy(gameObject);
    }
    }

    }
```

Машина состояний:

```
using UnityEngine;

[RequireComponent(typeof(Enemy))]
public class NewBehaviourScript : MonoBehaviour
{
[SerializeField] private State _firstState;

private Player _target;
private State _currentState;

    public State Current => _currentState;

```

```
13.     private void Start()
14.     {
15.     _target = GetComponent<Enemy>().Target;
16.     Reset(_firstState);
17.     }
18.
19.     private void Update()
20.     {
21.     if (_currentState == null)
22.     return;
23.
24.     var nextState = _currentState.GetNextState();
25.     if (nextState != null)
26.     Transit(nextState);
27.     }
28.
29.     private void Reset(State startState)
30.     {
31.     _currentState = startState;
32.
33.     if (_currentState != null)
34.     _currentState.Enter(_target);
35.     }
36.
37.     private void Transit(State nextState)
38.     {
39.     if (_currentState != null)
40.     _currentState.Exit();
41.
42.     _currentState = nextState;
43.
44.     if (_currentState != null)
45.     _currentState.Enter(_target);
46.     }
47.     }
```

Данный код отвечает за состояние нашего персонажа для того чтобы противник знал куда идти и кого атаковать:

```
1. using UnityEngine;
2.
3. public class DistanceTransition : Transition
4. {
5. [SerializeField] private float _transitionRange;
6. [SerializeField] private float _rangeSpread;
7.
8. private void Start()
9. {
10.     _transitionRange += Random.Range(-_rangeSpread,
   _rangeSpread);
11.     }
```

```
12.     private void Update()
13.     {
14.     if (Vector2.Distance(transform.position, Tar-
  get.transform.position) < _transitionRange)
15.     {
16.     NeedTransit = true;
17.     }
18.     }
19.     }
```

Проверка расстояния от противника до персонажа:

```
1. using UnityEngine;
2.
3. [RequireComponent(typeof(Animator))]
4.
5. public class AttackState : State
6. {
7. [SerializeField] private int _damage;
8. [SerializeField] private float _delay;
9.
10.     private float _lastAttackTime;
11.     private Animator _animator;
12.
13.     private void Start()
14.     {
15.     _animator = GetComponent<Animator>();
16.     }
17.
18.     private void Update()
19.     {
20.     if (_lastAttackTime <= 0)
21.     {
22.     Attack(Target);
23.     _lastAttackTime = _delay;
24.     }
25.     _lastAttackTime -= Time.deltaTime;
26.     }
27.
28.     private void Attack(Player target)
29.     {
30.     _animator.Play("Attack");
31.     target.ApplyDamage(_damage);
32.     }
33.     }
```

Атака персонажа, когда он достигает своей цели:

```
1. using UnityEngine;
2.
3. [RequireComponent(typeof(Animator))]
4.
5. public class CelebrationState : State
```

```
6. {
7. private Animator _animator;
8.
9. private void Awake()
10.     {
11.     _animator = GetComponent<Animator>();
12.     }
13.
14.     private void OnEnable()
15.     {
16.     _animator.Play("Celebration");
17.     }
18.
19.     private void OnDisable()
20.     {
21.     _animator.StopPlayback();
22.     }
23.     }
```

Состояние победы противника над нашим игроком:

```
1. using UnityEngine;
2.
3. public class MoveState : State
4. {
5. [SerializeField] private float _speed;
6.
7. private void Update()
8. {
9. transform.position = Vec-
   tor2.MoveTowards(transform.position, Tar-
   get.transform.position, _speed * Time.deltaTime);
10.     }
11.     }
```

Состояние пути до нашего игрока:

```
1. using System.Collections;
2. using System.Collections.Generic;
3. using UnityEngine;
4.
5. public abstract class State : MonoBehaviour
6. {
7. [SerializeField] private List<Transition>
   _transitions;
8.
9. protected Player Target { get; private set; }
10.
11.     public void Enter(Player target)
12.     {
13.     if (enabled == false)
14.     {
15.     Target = target;
```

```
16.     enabled = true;
17.     foreach (var transition in _transitions)
18.     {
19.     transition.enabled = true;
20.     transition.Init(Target);
21.     }
22.     }
23.     }
24.
25.     public void Exit()
26.     {
27.     if (enabled == true)
28.     {
29.     foreach (var transition in _transitions)
30.     transition.enabled = false;
31.
32.     enabled = false;
33.     }
34.     }
35.
36.     public State GetNextState()
37.     {
38.     foreach (var transition in _transitions)
39.     {
40.     if (transition.NeedTransit)
41.     return transition.TargetState;
42.     }
43.
44.     return null;
45.     }
46.     }
```

Выход и вход в различные состояния:

```
1. using UnityEngine;
2.
3. public abstract class Transition : MonoBehaviour
4. {
5. [SerializeField] private State _targetState;
6.
7. protected Player Target { get; private set; }
8.
9. public State TargetState => _targetState;
10.
11.     public bool NeedTransit { get; protected set; }
12.
13.     public void Init(Player target)
14.     {
15.     Target = target;
16.     }
17.
18.     private void OnEnable()
19.     {
```

```
20.     NeedTransit = false;
21.     }
22.     }
```

Код места появления противников

```
1. using System.Collections;
2. using System.Collections.Generic;
3. using UnityEngine;
4. using UnityEngine.Events;
5.
6. public class Spawner : MonoBehaviour
7. {
8. [SerializeField] private List<Wave> _waves;
9. [SerializeField] private Transform _spawnPoint;
10.     [SerializeField] private Player _player;
11.
12.     private Wave _currentWave;
13.     private int _currentWaveNumber = 0;
14.     private float _timeAfterLastSpawn;
15.     private int _spawned;
16.
17.     public event UnityAction AllEnemySpawned;
18.     public event UnityAction<int, int> EnemyCount-
  Changed;
19.
20.     private void Start()
21.     {
22.     SetWave(_currentWaveNumber);
23.     }
24.
25.     private void Update()
26.     {
27.     if (_currentWave == null)
28.     return;
29.
30.     _timeAfterLastSpawn += Time.deltaTime;
31.
32.     if (_timeAfterLastSpawn >= _currentWave.Delay)
33.     {
34.     InstantiateEnemy();
35.     _spawned++;
36.     _timeAfterLastSpawn = 0;
37.     EnemyCountChanged?.Invoke(_spawned,
  _currentWave.Count);
38.     }
39.
40.     if (_currentWave.Count <= _spawned)
41.     {
42.     if (_waves.Count > _currentWaveNumber + 1)
43.     AllEnemySpawned?.Invoke();
44.
45.     _currentWave = null;
```

```
46.     }
47.     }
48.
49.     private void InstantiateEnemy()
50.     {
51.     Enemy enemy = Instanti-
  ate(_currentWave.Template, _spawnPoint.position,
  _spawnPoint.rotation,
  _spawnPoint).GetComponent<Enemy>();
52.     enemy.Init(_player);
53.     enemy.Dying += OnEnemyDying;
54.     }
55.
56.     private void SetWave(int index)
57.     {
58.     _currentWave = _waves[index];
59.     EnemyCountChanged?.Invoke(0, 1);
60.     }
61.
62.     public void NextWave()
63.     {
64.     SetWave(++_currentWaveNumber);
65.     _spawned = 0;
66.     }
67.
68.     private void OnEnemyDying(Enemy enemy)
69.     {
70.     enemy.Dying -= OnEnemyDying;
71.
72.     _player.AddMoney(enemy.Reward);
73.     }
74.     }
75.
76.     [System.Serializable]
77.     public class Wave
78.     {
79.     public GameObject Template;
80.     public float Delay;
81.     public int Count;
82.     }
```

Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий

Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой
_____ О.В. Непомнящий
«20» 06 2023 г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 – Информатика и вычислительная техника

Мобильная 2D-Игра

Руководитель	_____ *подпись*	20.06.23 *дата*	доцент, канд. физ.-мат. наук *должность, ученая степень*	К.В. Коршун
Выпускник	_____ *подпись*	20.06.23 *дата*		Р.А. Злобин
Нормоконтролёр	_____ *подпись*	20.06.23 *дата*	доцент, канд. физ.-мат. наук *должность, ученая степень*	К.В. Коршун

Красноярск 2023