МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Отделение информационных технологий и вычислительной техники

Утверждаю

Руководитель

отделения ИТВТ \_\_\_\_\_\_\_\_\_\_\_.

\_\_\_ *Рощенко О.Е.*\_\_

(подпись, фамилия, инициалы)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**ДИПЛОМНЫЙ ПРОЕКТ**

**НА ТЕМУ**

**РАЗРАБОТКА КОМПЬЮТЕРНОЙ ИГРЫ**

**В ЖАНРЕ «АРКАДА»**

Автор дипломного проекта

(подпись студента, выполнившего дипломный проект)

*Захаров И.Е.*  Группа *ИЗ-71*    
(фамилия, инициалы студента) (в которой обучался студент)

*Институт социальных технологий*

(факультет)

Специальность *09.02.03 Программирование в компьютерных системах*

(код и наименование специальности)

Руководитель дипломного проекта *И.Н. Прохорова*

(подпись, инициалы, фамилия)

Новосибирск, 2022 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Отделение информационных технологий и вычислительной техники

Утверждаю

Руководитель

отделения ИТВТ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_ *Рощенко О.Е.*\_\_\_\_\_

(подпись, фамилия, инициалы)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

**ЗАДАНИЕ НА ДИПЛОМНЫЙ ПРОЕКТ**

студент*у(ке)* *Захарову Илье Евгеньевичу* группы *ИЗ-71*

(фамилия, инициалы) (обучения)

1. Тема **Разработка компьютерной игры в жанре «Аркада»**

Утверждена приказом по НГТУ № \_\_\_\_\_\_\_\_\_ от «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

2. Дата представления проекта к защите «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

3. Цели проекта (исходные данные*) Разработать игровое приложение в жанре «Аркада». Приложение должно представлять собой двумерную космическую аркаду, в которой реализованы все игровые механики, свойственные этому жанру. Основная цель игры - продержаться наибольшее количество волн.*

4. Содержание пояснительной записки

*4.1 Введение*

*4.2 Глава 1. Анализ предметной области и теоретические основы разработки  
 4.3 Постановка задачи  
 4.4 Классификация видеоигр*

*4.5 Обоснование выбора средств программной разработки*

*4.6 Особенности производства видеоигр*

*4.7 Глава 2. Практическая реализация проекта*

*4.8 Техническое задание на разработку проекта*

*4.9 Функциональные требования*

*4.10 Дизайн проекта*

*4.11 Структурное описание проекта*

*4.12 Создание прототипа компьютерной игры*

*4.13 Функциональное описание разработки*

*4.14 Конфигурирование*

*4.15 Клиентская часть*

*4.16 Инструкция по использованию программного продукта*

*4.17 Этап тестирования*

*4.18 Заключение*

*4.19 Список использованной литературы*

*4.20 Приложение*

5. Перечень графического и (или) иллюстрационного материала

**Руководитель проекта** Прохорова И.Н.

(подпись, дата) (фамилия, инициалы)

**Задание принял к исполнению** Захаров И.Е.

(подпись студента, дата) (фамилия, инициалы студента)

# ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 6](#_Toc106059383)

[ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ 8](#_Toc106059384)

[1.1. Постановка задачи 8](#_Toc106059385)

[1.2. Классификация видеоигр 8](#_Toc106059386)

[1.3. Обоснование выбора средств программной разработки 13](#_Toc106059387)

[1.4. Особенности производства видеоигр 23](#_Toc106059388)

[ГЛАВА 2. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ ПРОЕКТА 25](#_Toc106059389)

[2.1. Техническое задание на разработку проекта 25](#_Toc106059390)

[2.1.1 Функциональные требования 25](#_Toc106059391)

[2.1.2 Дизайн проекта 26](#_Toc106059392)

[2.2. Структурное описание разработки 26](#_Toc106059393)

[2.3 Создание прототипа компьютерной игры 28](#_Toc106059394)

[2.4 Производство компьютерной игры 37](#_Toc106059395)

[2.5 Функциональное описание разработки 55](#_Toc106059396)

[2.5.1 Конфигурирование 55](#_Toc106059397)

[2.5.2 Клиентская часть 56](#_Toc106059398)

[2.5.3 Инструкция по использованию программного продукта 58](#_Toc106059399)

[2.6 Этап тестирования 58](#_Toc106059400)

[ЗАКЛЮЧЕНИЕ 60](#_Toc106059401)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 61](#_Toc106059402)

[ПРИЛОЖЕНИЕ 62](#_Toc106059403)

# ВВЕДЕНИЕ

С момента самой первой выпущенной видеоигры прошло шестьдесят лет. За это время игровая индустрия развивалась с неимоверной скоростью, что позволяло нам наблюдать и играть во многие культовые игры. За последние 15 лет игровая индустрия совершила прорыв с точки зрения разработки игр, появилось всё больше разнообразных механик, геймплейных реализаций, была улучшена визуальная составляющая. Это обусловлено тем, что все устройства, на которых проходят игровые сессии, развиваются.

Видеоигры являются неотъемлемой частью культуры. Люди играют в них дома, по пути на работу, на обеденном перерыве. Это позволяет им отвлечься от повседневной рутины. Из-за такой популярности видеоигр, начинаются обсуждение на тему пользы и вреда видеоигр. Но множество исследований подтверждают пользу видеоигр, что в свою очередь делает беспочвенными множество обвинений в пользу вреда игр.

В свою очередь популярность видеоигр является двигателем прогресса, как технологического и экономического, так и социального.

В технологическом плане идет развитие и создание новых комплектующих для всех устройств, появляются новые микрочипы видеокарт и процессоров, улучшается качество устройств из-за постоянной конкуренции на рынке технологий, что в свою очередь является прогрессом для многих разных сфер и технологий, например транспорт.

В экономическом плане создаётся множество игровых студий и компаний, которые открывают вакансии для работников, что положительно влияет на экономическую ситуацию в стране.

В социальном плане игровая индустрия является одной из частей массовой культуры, которая доступна множеству людей. Люди могут выбирать, во что играть, есть разные жанры видеоигр и классификаций, которые мы рассмотрим позже. Практически у каждой игры есть своё комъютнити (от англ. community – сообщество), в котором люди взаимодействуют, могут находить новых друзей, знакомых с похожими интересами из разных частей мира. Это в свою очередь является формой социальной глобализации, ведь множество людей из разных стран могут играть в одну игру, могут в рамках этой игры общаться и взаимодействовать.

Поэтому можно сделать вывод, что тема моей дипломной работы, а именно - Разработка компьютерной игры в жанре "Аркада" является актуальной.

В рамках данной выпускной квалификационной работы будет разработана компьютерная игра в жанре «Аркада» на платформе Unity

Цель выпускной квалификационной работы − разработать компьютерную игру в жанре «Аркада».

Задачи выпускной квалификационной работы:

1. Проанализировать предметную область и теорию разработки видеоигр.
2. Проанализировать и обосновать выбор инструментария для разработки.
3. Продумать концепцию игры.
4. Разработать требования к проектируемой видеоигре.
5. Разработать пользовательский интерфейс (UI).
6. Разработать прототип и реализовать основные механики игры.
7. Выбрать музыкальное и звуковое сопровождение игрового процесса.
8. Протестировать игровое приложение.

Результатом работы должна быть корректно функционирующая видеоигра в жанре «Аркада» на платформе Windows.

# ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ

# Постановка задачи

Требуется разработать компьютерную игру в жанре «Аркада» для десктопных устройств. Приложение должно содержать завершенные механики персонажей игрока и противников, генерирующихся предметов, цикличности игры, интуитивно понятный интерфейс.

Основные этапы разработки:

* анализ предметной области;
* выбор программных средств и технологий;
* составление технического задания на разработку;
* создание прототипа;
* разработка приложения;
* тестирование приложения.

Результатом должно стать приложение, полностью соответствующим требованиям технического задания.

# Классификация видеоигр

Компьютерные игры разделены на множество категорий и разновидностей игры могут одновременно находиться в нескольких категориях из-за их разнообразия и уникальности.

Всё многообразие компьютерных игр можно делить на группы, используя множество самых различных способов. Рассортируем все эти способы, соблюдая структуру игровой индустрии и рассмотрим их ниже.

1. Платформа:
   1. Персональный компьютер (ПК, ноутбук, нетбук).
   2. Игровая консоль или приставка (PS, Xbox, Nintendo).
   3. Мобильное устройство (телефон, планшет).
   4. Портативные компьютеры (PSP, SteamDeck, Nintendo Switch).
   5. Игровой автомат.
   6. Браузерная игра.
   7. VR устройства (шлемы виртуальной реальности).
2. Графическое изображение:
   1. Классификация по технологии графического изображения.
      1. Отсутствие графики (текстовые игры, псевдографика)**.**
      2. Двухмерная графика (векторная, растровая).
      3. Трехмерная графика.
      4. Объемное изображение (стерео очки).
      5. Дополненная реальность (мобильные устройства с видеокамерой).
      6. Виртуальная реальность (шлем виртуальной реальности).
   2. Классификация по расположению игровой камеры:
      1. Вид от 1-го лица (вид из глаз).
      2. Вид от 3-го лица (вид сзади).
      3. Двухмерный вид сбоку (2D вид сбоку).
      4. Трехмерный вид сбоку (3D вид сбоку, псевдотрехмерность).
      5. Двухмерный вид сверху (2D TopDown).
      6. Трехмерный вид сверху (3D TopDown, изометрия).
3. Содержание игры:
   1. Классификация игр по жанру:
      1. Action:
         1. 3D-шутеры, «бродилки-стрелялки».
         2. Шутеры от первого лица и шутеры от 3 лица.
         3. Кровавые шутеры.
         4. Тактические шутеры.
         5. Файтинги.
         6. Избей их всех.
         7. Слешер.
         8. Аркада.
         9. Стелс-экшен.
      2. Симуляторы:
         1. Технические симуляторы.
         2. Аркадные симуляторы.
         3. Экономические симуляции.
      3. Стратегии:
         1. Стратегии реального времени.
         2. Пошаговые стратегии.
         3. Симуляторы бога.
         4. Экономические симуляторы.
         5. Варгеймы.
         6. Глобальные стратегии.
      4. Спортивные.
      5. Приключения:
         1. Текстовые квесты.
         2. Графические квесты.
         3. Головоломки.
         4. Экшн-адвенчуры.
         5. Симуляторы свиданий.
      6. Ролевые игры:
         1. RPG-повествование.
         2. RPG - песочница.
         3. Зачистка подземелий.
      7. Головоломки, логические, пазлы.
      8. Традиционные и настольные.
      9. Текстовые. Игры в псевдографике.
   2. Классификация по сеттингу:
      1. По месту действий (тип вымышленного мира):
         1. Реальный мир (игровой мир мало чем отличается от нашего мира).
         2. Параллельные миры (реальный мир и переходы в искаженные миры).
         3. Альтернативная история (другие варианты исторических событий).
         4. Фэнтези (сказочный мир с наличием магии).
         5. Геройская мифология (сверхсущества: супергерои, мутанты, полубоги).
         6. Христианская мифология (ангелы, демоны, рай, ад).
         7. Современная мифология (нашествие зомби, нашествие инопланетян, постапокалипсис).
      2. По времени действий (историческая эпоха):
         1. Зарождение жизни (простейшие организмы, игры на клеточном уровне).
         2. Доисторические времена (эра динозавров, пещерные люди).
         3. Зарождение цивилизаций (Древний Рим, Древний Египет).
         4. Средневековье (рыцарские походы, междоусобицы, инквизиция).
         5. Эпоха колонизации (морские путешествия, новые земли).
         6. Эпоха индустриализации (XVIII – XIX века, стимпанк).
         7. Прошедшие войны (Вторая Мировая война, локальные конфликты).
         8. Наше время (привычный мир).
         9. Информационная эпоха (киберпанк, антиутопия будущего).
         10. Освоение космоса (научная фантастика, SciFi, космические путешествия).
         11. Эволюция (наличие нескольких эпох в одной игре).
   3. Классификация по цели игры:
      1. Игра на прохождение (выполнение целей, сюжет).
      2. Обучающая игра (получение новых знаний).
      3. Казуальная (повседневная) игра (наслаждение самим процессом).
      4. Игра-песочница (творческие возможности, выбор целей).
      5. Игра-соревнование (дуэль, чемпионат) (соперничество).
      6. Хардкорная (очень сложная) игра (улучшение игровых навыков).
   4. Классификация по левелдизайну:
      1. Линейная игра (один путь для прохождения).
      2. Вариативная игра (есть развилки в сюжете, несколько путей).
      3. Иммерсив (много путей и способов прохождения).
      4. Метроидвания (основной путь и заблокированные ответвления).
      5. Рогалик (бесконечное число случайных локаций).
      6. Открытый мир (песочница, нет ограничений на перемещения).
4. Издательские критерии:
   1. Классификация по бюджету разработки:
      1. Профессиональная игра высшего качества (ААА-класс, блокбастеры).
      2. Профессиональная игра (игры со средним бюджетом).
      3. Инди-игра (независимая игра, малобюджетный класс).
      4. Любительская игра (бесплатная игра с минимальным качеством).
   2. Классификация по издательскому формату:
      1. Оригинальная игра.
      2. Очередная игра в игровой серии (сиквел, приквел).
      3. Дополнение к игре.
      4. Скачиваемый контент к игре (DLC).
5. Тип распространения:
   1. Платные игры:
      1. Игра на физическом носителе (диски, картриджи).
      2. Цифровая копия игры (продажа игр через Интернет).
      3. Оплата за игровое время (игровые автоматы, некоторые MMO-игры).
   2. Бесплатные игры:
      1. Условно бесплатная игра (shareware).
      2. Бесплатная игра с микротранзакциями (free to play, free2play) (большая часть MMO-игр).
      3. Бесплатная игра (флеш-игра, браузерная игра, скачиваемая игра).
6. Количество игроков:
   1. Одиночная игра (Синглплейер, англ. Singleplayer).
   2. Совместная игра на одном устройстве (Hotseat, Splitscreen).
   3. Многопользовательская игра (Мультиплейер, Multiplayer).
   4. Массовая онлайн игра (англ. Massively multiplayer online game, MMO).

По данной классификации будет разработана научно-фантастическая двумерная компьютерная игра, в жанре «Аркада» на одного игрока.

# Обоснование выбора средств программной разработки

Язык программирования С++

C++ - язык программирования, который используется во время разработки на Unreal Engine 4. C++ (читается си-плюс-плюс) — компилируемый, статически типизированный язык программирования общего назначения.

Поддерживает такие парадигмы программирования, как процедурное программирование, объектно-ориентированное программирование, обобщённое программирование. Язык имеет богатую стандартную библиотеку, которая включает в себя распространённые контейнеры и алгоритмы, ввод-вывод, регулярные выражения, поддержку многопоточности и другие возможности. C++ сочетает свойства как высокоуровневых, так и низкоуровневых языков. В сравнении с его предшественником — языком C — наибольшее внимание уделено поддержке объектно-ориентированного и обобщённого программирования.

C++ широко используется для разработки программного обеспечения, являясь одним из самых популярных языков программирования. Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также игр. Существует множество реализаций языка C++, как бесплатных, так и коммерческих и для различных платформ. Например, на платформе x86 это GCC, Visual C++, Intel C++ Compiler, Embarcadero (Borland) C++ Builder и другие. C++ оказал огромное влияние на другие языки программирования, в первую очередь на Java и C#.

Синтаксис C++ унаследован от языка C. Изначально одним из принципов разработки было сохранение совместимости с C. Тем не менее C++ не является в строгом смысле надмножеством C. Множество программ, которые могут одинаково успешно транслироваться как компиляторами C, так и компиляторами C++, довольно велико, но не включает все возможные программы на C.

Язык программирования C#

C# — объектно-ориентированный язык программирования. Разработан в 1998—2001 годах группой инженеров компании Microsoft под руководством Андерса Хейлсберга и Скотта Вильтаумота как язык разработки приложений для платформы Microsoft .NET Framework и .NET Core. Впоследствии был стандартизирован как ECMA-334 и ISO/IEC 23270.

C# — это язык программирования, используемый во время разработки на Unity.

C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к C++ и Java. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, переменные, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML.

Переняв многое от своих предшественников — языков C++, Delphi, Модула, Smalltalk и, в особенности, Java — C#, опираясь на практику их использования, исключает некоторые модели, зарекомендовавшие себя как проблематичные при разработке программных систем, например, C# в отличие от C++ не поддерживает множественное наследование классов (между тем допускается множественная реализация интерфейсов).

C#‎ разрабатывался как язык программирования прикладного уровня для CLR и, как таковой, зависит, прежде всего, от возможностей самой CLR. Это касается, прежде всего, системы типов С#‎, которая отражает BCL. Присутствие или отсутствие тех или иных выразительных особенностей языка диктуется тем, может ли конкретная языковая особенность быть транслирована в соответствующие конструкции CLR.

Так, с развитием CLR от версии 1.1 к 2.0 значительно обогатился и сам C#. Подобного взаимодействия следует ожидать и в дальнейшем (однако, эта закономерность была нарушена с выходом C# 3.0, представляющего собой расширения языка, не опирающиеся на расширения платформы .NET). CLR предоставляет С#‎, как и всем другим .NET-ориентированным языкам, многие возможности, которых лишены «классические» языки программирования. Например, сборка мусора не реализована в самом C#‎, а производится CLR для программ, написанных на C# точно так же, как это делается для программ на VB.NET, J# и др.

Unity

Unity — межплатформенная среда разработки компьютерных игр, разработанная американской компанией Unity Technologies. Unity позволяет создавать приложения, работающие на более чем 25 различных платформах, включающих персональные компьютеры, игровые консоли, мобильные устройства, интернет-приложения и другие. Выпуск Unity состоялся в 2005 году и с того времени идёт постоянное развитие.

Основными преимуществами Unity являются наличие визуальной среды разработки, межплатформенной поддержки и модульной системы компонентов. К недостаткам относят появление сложностей при работе с многокомпонентными схемами и затруднения при подключении внешних библиотек.

На Unity написаны тысячи игр, приложений, визуализации математических моделей, которые охватывают множество платформ и жанров. При этом Unity используется как крупными разработчиками, так и независимыми студиями.

Редактор Unity имеет простой Drag&Drop интерфейс, а также установкой плагинов KALI который легко настраивать, состоящий из различных окон, благодаря чему можно производить отладку игры прямо в редакторе. Движок использует для написания скриптов C#. Ранее поддерживались также Boo (диалект Python, поддержку убрали в 5-й версии) и модификация JavaScript, известная как UnityScript (поддержка прекращена в версии 2017.1). Расчёты физики производит физический движок PhysX от NVIDIA. Графический API — DirectX (на данный момент DX 11, поддерживается DX 12).

Проект в Unity делится на сцены (уровни) — отдельные файлы, содержащие свои игровые миры со своим набором объектов, сценариев, и настроек. Сцены могут содержать в себе как, собственно, объекты (модели), так и пустые игровые объекты — объекты, которые не имеют модели («пустышки»). Объекты, в свою очередь содержат наборы компонентов, с которыми и взаимодействуют скрипты. Также у объектов есть название (в Unity допускается наличие двух и более объектов с одинаковыми названиями), может быть тег (метка) и слой, на котором он должен отображаться. Так, у любого объекта на сцене обязательно присутствует компонент Transform — он хранит в себе координаты местоположения, поворота и размеров объекта по всем трём осям. У объектов с видимой геометрией также по умолчанию присутствует компонент Mesh Renderer, делающий модель объекта видимой.

К объектам можно применять коллизии (в Unity т. н. коллайдеры — collider), которых существует несколько типов.

Также Unity поддерживает физику твёрдых тел и ткани, а также физику типа Ragdoll (тряпичная кукла). В редакторе имеется система наследования объектов. Дочерние объекты будут повторять все изменения позиции, поворота и масштаба родительского объекта. Скрипты в редакторе прикрепляются к объектам в виде отдельных компонентов.

При импорте текстуры в Unity можно сгенерировать alpha-канал, mip-уровни, normal-map, light-map, карту отражений, однако непосредственно на модель текстуру прикрепить нельзя — будет создан материал, которому будет назначен шейдер, и затем материал прикрепится к модели. Редактор Unity поддерживает написание и редактирование шейдеров. Редактор Unity имеет компонент для создания анимации, но также анимацию можно создать предварительно в 3D-редакторе и импортировать вместе с моделью, а затем разбить на файлы.

Unity 3D поддерживает систему Level Of Detail (сокр. LOD), суть которой заключается в том, что на дальнем расстоянии от игрока высокодетализированные модели заменяются на менее детализированные, и наоборот, а также систему Occlusion culling. Суть системы Occlusion culling в том, что у объектов, не попадающих в поле зрения камеры не визуализируется геометрия и коллизия, что снижает нагрузку на центральный процессор и позволяет оптимизировать проект. При компиляции проекта создается исполняемый (.exe) файл игры (для Windows), а в отдельной папке — данные игры (включая все игровые уровни и динамически подключаемые библиотеки).

Движок поддерживает множество популярных форматов. Модели, звуки, текстуры, материалы, скрипты можно запаковывать в формат .unitypackage и передавать другим разработчикам, или выкладывать в свободный доступ. Этот же формат используется во внутреннем магазине Unity Asset Store, в котором разработчики могут бесплатно и за деньги выкладывать в общий доступ различные элементы, нужные при создании игр. Чтобы использовать Unity Asset Store, необходимо иметь аккаунт разработчика Unity. Unity имеет все нужные компоненты для создания мультиплеера. UNet был удалён, начиная с версии 2018.4. Решение «из коробки» для мультиплеера отсутствует. Также можно использовать подходящий пользователю способ контроля версий. К примеру, Tortoise SVN, Git или Source Gear.

В Unity входит Unity Asset Server — инструментарий для совместной разработки на базе Unity, являющийся дополнением, добавляющим контроль версий и ряд других серверных решений.

Как правило, игровой движок предоставляет множество функциональных возможностей, позволяющих их задействовать в различных играх, в которые входят моделирование физических сред, карты нормалей, динамические тени и многое другое. В отличие от многих игровых движков, у Unity имеется два основных преимущества: наличие визуальной среды разработки и межплатформенная поддержка. Первый фактор включает не только инструментарий визуального моделирования, но и интегрированную среду, цепочку сборки, что направлено на повышение производительности разработчиков, в частности, этапов создания прототипов и тестирования. Под межплатформенной поддержкой предоставляется не только места развертывания (установка на персональном компьютере, на мобильном устройстве, консоли и т. д.), но и наличие инструментария разработки (интегрированная среда может использоваться под Windows и Mac OS).

Третьим преимуществом называется модульная система компонентов Unity, с помощью которой происходит конструирование игровых объектов, когда последние представляют собой комбинируемые пакеты функциональных элементов. В отличие от механизмов наследования, объекты в Unity создаются посредством объединения функциональных блоков, а не помещения в узлы дерева наследования. Такой подход облегчает создание прототипов, что актуально при разработке игр.

В качестве недостатков приводятся ограничение визуального редактора при работе с многокомпонентными схемами, когда в сложных сценах визуальная работа затрудняется. Вторым недостатком называется отсутствие поддержки Unity ссылок на внешние библиотеки, работу с которыми программистам приходится настраивать самостоятельно, и это также затрудняет командную работу. Ещё один недостаток связан с использованием шаблонов экземпляров (англ. prefabs). С одной стороны, эта концепция Unity предлагает гибкий подход визуального редактирования объектов, но с другой стороны, редактирование таких шаблонов является сложным. Также, WebGL-версия движка, в силу специфики своей архитектуры (трансляция кода из C# в С++ и далее в JavaScript), имеет ряд нерешённых проблем с производительностью, потреблением памяти и работоспособностью на мобильных устройствах.

Unity Technologies предлагает разные лицензионные условия начиная от бесплатных до платных, а именно:

1. Personal – бесплатная. Доход компании до 100.000$ в год.
2. Plus – 399$ в год. Доход компании до 200.000$ в год.
3. Pro – 1800$ в год. Доход компании неограничен.
4. Enterprise – 200$. Доход компании неограничен.

Проанализировав все аналоги, на которых можно качественно выполнить разработку компьютерной игры, было решено использовать игровой движок Unity.

Visual Studio

Microsoft Visual Studio — линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментов. Данные продукты позволяют разрабатывать как консольные приложения, так и игры и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms. А также веб-сайты, веб-приложения, веб-службы как в родном, так и в управляемом кодах для всех платформ, поддерживаемых Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework и Silverlight.

Visual Studio включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Остальные встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных. Visual Studio позволяет создавать и подключать сторонние дополнения (плагины) для расширения функциональности практически на каждом уровне, включая добавление поддержки систем контроля версий исходного кода (как, например, Subversion и Visual SourceSafe), добавление новых наборов инструментов (например, для редактирования и визуального проектирования кода на предметно-ориентированных языках программирования) или инструментов для прочих аспектов процесса разработки программного обеспечения (например, клиент Team Explorer для работы с Team Foundation Server).

Abode Photoshop

Adobe Photoshop — многофункциональный графический редактор, разрабатываемый и распространяемый компанией Adobe Systems. В основном работает с растровыми изображениями, однако имеет некоторые векторные инструменты. Продукт является лидером рынка в области коммерческих средств редактирования растровых изображений и наиболее известной программой разработчика.

В настоящее время Photoshop доступен на платформах macOS, Windows и iPadOS. Для Windows Phone и Android доступна упрощённая версия приложения под названием Adobe Photoshop Touch. Также существует версия Photoshop Express для Windows Phone 8 и 8.1. В 2014 году в США проходило бета-тестирование потоковой версии продукта для Chrome OS. Ранние версии редактора были портированы под SGI IRIX, но официальная поддержка прекращена начиная с третьей версии продукта. Для версий 8.0 и CS6 возможен запуск под Linux с помощью альтернативы Windows API — Wine.

Хотя изначально программа была разработана как редактор изображений для полиграфии, в данное время она широко используется и в веб-дизайне. Вместе с более ранними версиями Photoshop распространялась специальная программа для этих целей — Adobe ImageReady (для анимации файлов GIF), которая была исключена из поставки Photoshop CS3 за счёт интеграции её функций в основной графический редактор, а также включения в линейку программных продуктов Adobe Fireworks, перешедшего в собственность Adobe после приобретения компании Macromedia.

Photoshop тесно связан с другими программами для обработки медиафайлов, анимации и пр. творчества. Совместно с такими программами, как Adobe ImageReady (программа упразднена в версии CS3), Adobe Illustrator, Adobe Premiere, Adobe After Effects и Adobe Encore DVD, он может использоваться для создания профессиональных DVD, обеспечивает средства нелинейного монтажа и создания таких спецэффектов, как фоны, текстуры и т. д. для телевидения, кинематографа и Интернета. Photoshop также прижился в кругах разработчиков компьютерных игр.

Основной формат Photoshop — PSD — может быть экспортирован и импортирован всеми перечисленными выше программными продуктами. Photoshop CS поддерживает создание меню для DVD. Совместно с Adobe Encore DVD Photoshop позволяет создавать меню или кнопки DVD. Photoshop CS3 в версии Extended поддерживает также работу с трёхмерными слоями. Из-за высокой популярности Photoshop поддержка специфического для неё формата PSD была реализована во многих графических программах, таких, как Adobe Fireworks, Photo-Paint, WinImages, GIMP, SAI, PaintShop Pro и других.

Photoshop поддерживает следующие цветовые модели или способы описания цветов изображения (в аннотации самой программы — режим изображения):

* RGB;
* LAB;
* CMYK;
* в градациях серого;
* чёрно-белые;
* duotone;
* с 256-цветовой палитрой (Indexed);
* многоканальные (Multichannel).

Поддерживается обработка изображений с глубиной цвета 8 бит (256 градаций на один канал), 16 бит (используется 15 бит плюс один уровень, то есть 32769 уровней) и 32 бита (используются числа одинарной точности с плавающей запятой).

Выбранные программные средства

Учитывая всё вышеперечисленное, в качестве игрового движка был выбран Unity и язык программирования C#, а в качестве графического редактора был выбран Abode Photoshop.

Выбор Photoshop обусловлен широким выбором инструментов для работы с цветом и обладает огромной популярностью.

Unity – движок проверенный временем, имеет обширную документацию и большое сообщество разработчиков.

С#, язык используемый для написания скриптов на Unity, более удобен и имеет поддержку от Microsoft.

# Особенности производства видеоигр

При разработке приложения проектирование – один из важнейших этапов в успешной разработке приложения

Есть множество разных моделей проектирования приложения. Мы возьмем одну из них.

**Подготовка к производству или препродакшн**

Здесь начинается каждый проект. По сути, препродакшн определяет, о чем игра, зачем ее делать и что нужно для ее создания. У вас может быть отличная идея для типа игры, истории, которую вы хотите воплотить в жизнь, или вы можете захотеть создать такую, которая использует определенный тип технологий (например, VR, новый контроллер или консоль). На этапе подготовки к производству вы должны иметь ответы на такие вопросы, как:

1. О чем игра?
2. На какую аудиторию ориентирована игра?
3. Какие есть аналоги на рынке?
4. На какой платформе будет происходить разработка?
5. Какое количество ресурсов будет затрачено?

Этот этап может длиться от недели до года, в зависимости от типа проекта, использования ресурсов и финансов, и обычно занимает до 20% от общего времени производства.

Важным подготовительным этапом является создание прототипа.

Прототип видеоигры - это необработанный тест, который содержит полезный опыт, игровой процесс, механику и художественное оформление. Прототипирование происходит на стадии подготовки к производству, чтобы проверить, будет ли идея игры работать и стоит ли ее реализовывать. Многие идеи не соответствуют этой стадии.

**Производство**

Производство - это самый длинный этап конвейера. Игра занимает от 1 до 4 лет, и именно тогда игра действительно должна получить форму. История уточняется, ресурсы привлечения, правила игры, уровни и миры преследования, пишется код и многое другое.

Во время производства, для разработки игры, создаются:

* архитектура проекта;
* визуальное сопровождение;
* звуковое сопровождение;
* заполнение контентом игры.

# ГЛАВА 2. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ ПРОЕКТА

# Техническое задание на разработку проекта

При препродакшне, перед началом создания прототипа, даётся ответ на поставленные в предыдущей главе вопросы.

Космический шутер (space shooter) в жанре аркада. Игра ориентирована на людей всех возрастов, с разными интересами. У этой игры есть множество аналогов на рынке, но подобные игры пользуются спросом, учитывая специфику жанра, можно с уверенностью сказать, что космические шутеры всегда будут актуальны.

Игра будет разрабатываться на платформе Unity на языке программирования C#. Благодаря движку и его встроенному инструментарию можно сделать оптимизированную игру за короткий промежуток времени.

# 2.1.1 Функциональные требования

Из технического задания сформированы следующие функциональные требования:

1. Персонаж игрока и противника стреляет автоматически.
2. Персонаж должен иметь ограниченное количество жизней.
3. Два вида противников: босс и обычные противники.
4. Возможность восстановления здоровья.
5. Настройки игры: игра должна иметь настройки графики, уровня громкости музыки и звуков.
6. Звуковое сопровождение игры: звуки выстрелов, взрывов, фоновая музыка.
7. Игровой процесс состоит из волн с увеличивающейся сложностью.
8. В конце каждой волны должен появляться босс.
9. Управление должно быть реализовано на клавишах WASD.
10. Цель игры: продержаться как можно большее количество волн.

# 2.1.2 Дизайн проекта

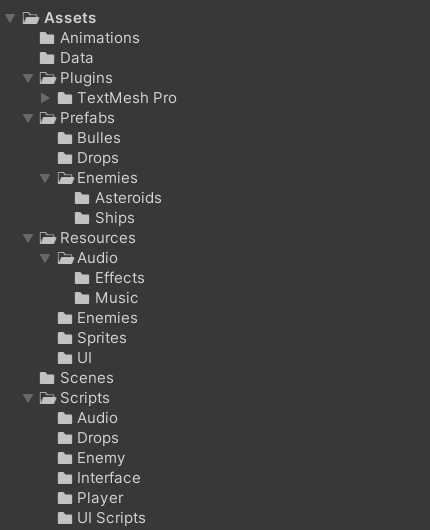
Дизайн проекта должен соответствовать следующим требованиям:

1. Игра должна быть реализована в 2D графике.
2. В игре должно быть несколько визуально отличающихся противников.
3. Игровая локация должна визуально напоминать космос.
4. Интерфейс должен быть реализован так, что бы игроку было интуитивно понятно, как им пользоваться.

# Структурное описание разработки

В проекте есть множество папок с файлами, но рабочей папкой является папка Assets.

Все файлы и объекты, которые будут созданы, хранятся в этой папке. Но для удобного взаимодействия с проектом будут созданы папки для разделения объектов по сущностям (Рисунок 1).



*Рисунок 1. Архитектура проекта.*

Папки будут называться следующим образом:

* папка Scripts – там будут храниться, написанные на C#, скрипты;
* папка Animations – для хранения анимаций;
* папка Scenes – содержит игровые и промежуточные сцены игры;
* папка Plugins – там хранятся плагины которые импортируются в движок;
* папка Data – там хранится информация о настройках игры;
* папка Resources – там хранятся все игровые ресурсы.

В папке Scripts и Resources так же есть папки, которые делят объекты на сущности.

В папке Scripts:

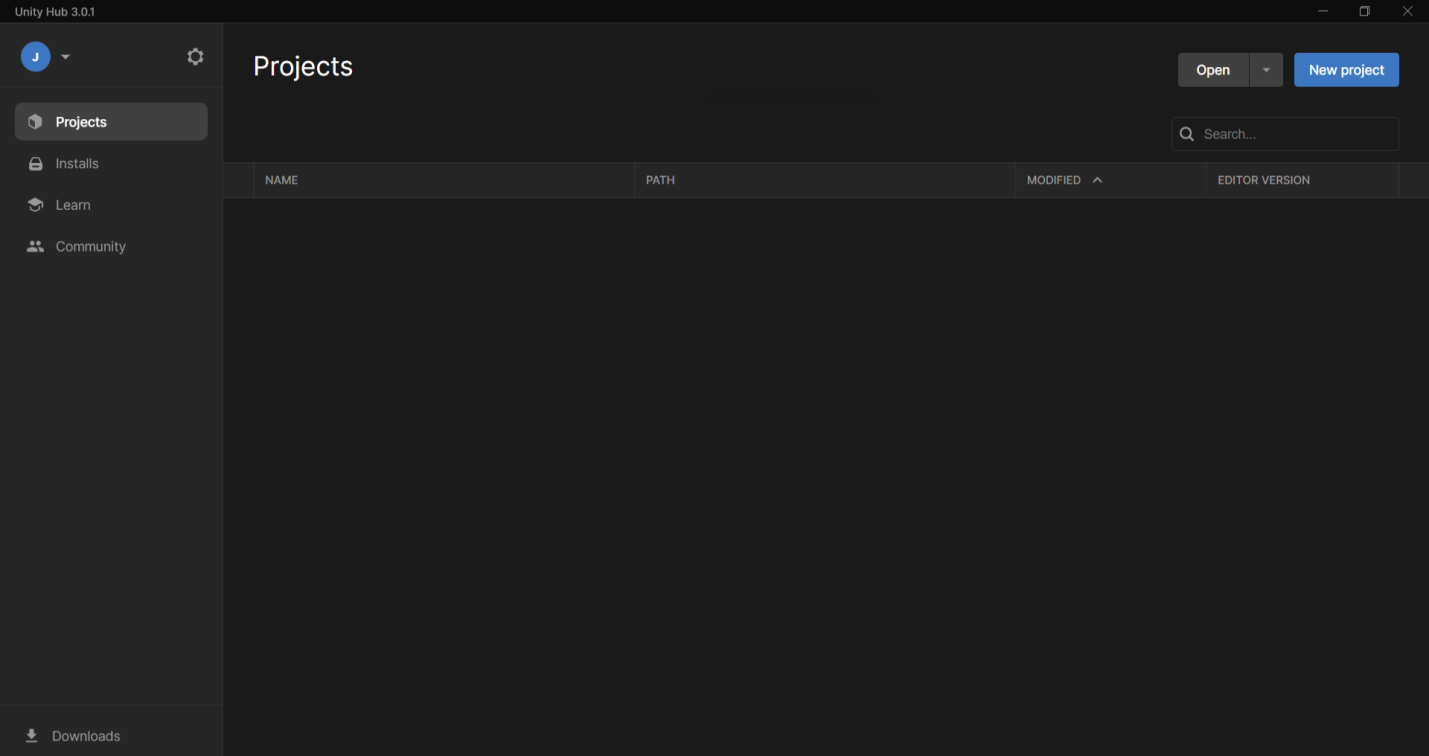
* папка Audio – всё что отвечает за звуковое и музыкальное сопровождение;
* папка Drops – где хранится логика предмета, восстанавливающего количество жизней;
* папка Enemy – в ней содержится логика, связанная с противниками;
* папка Player – содержит логику, связанную с игроком;
* папка Interface – интерфейсы;
* папка UI Scripts – всё что относится к UI.

В папке Resources:

* папка Audio – папка для хранения звуков и музыки игры;
* папка Enemies – содержит в себе спрайты противника;
* папка Sprites – хранятся все остальные спрайты;
* папка UI – хранит в себе все элементы UI.

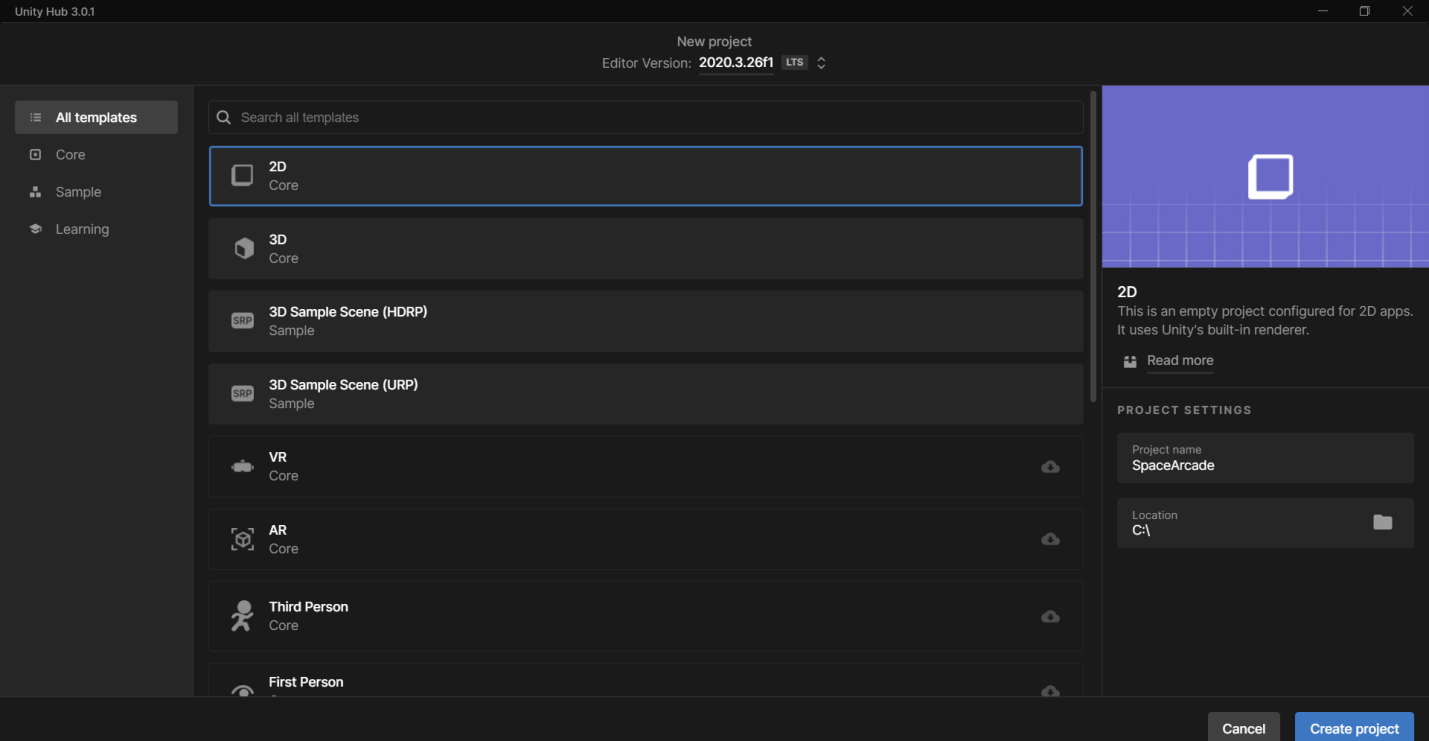
# 2.3 Создание прототипа компьютерной игры

Для того что бы начать работу над прототипом создадим проект. Запускаем Unity Hub (Рисунок 2).



*Рисунок 2. Unity Hub.*

Создаём проект в 2D c названием SpaceArcade (Рисунок 3)

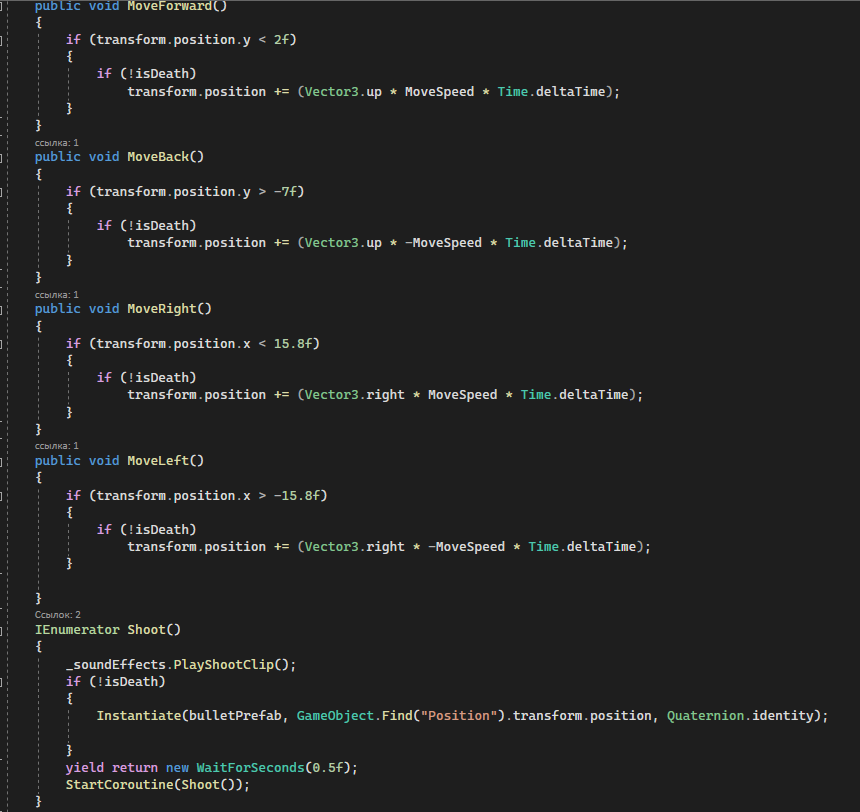


*Рисунок 3. Создание проекта в Unity.*

После того как мы создали проект переходим к разработке прототипа игры.

Для начала создадим 3 скрипта: PlayerSpaceShip, ShipBullet и KeyboardInputs.

Эти три скрипта будут отвечать за основной функционал космического корабля. Начнем с PlayerSpaceShip (Рисунок 4):



*Рисунок 4. Скрипт PlayerSpaceShip***.**

У него есть функции для передвижения, а именно MoveForvard, MoveBack, MoveRight. MoveLeft. Они, как и указано в своем названии, отвечают за передвижение вперед, назад, вправо, влево.

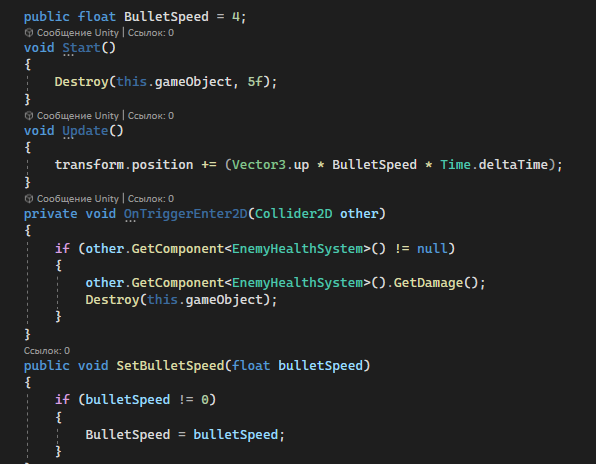
Так же есть функция Shoot, которая отвечает за стрельбу.

Далее переходим к скрипту ShipBullet. Это скрипт который будет отвечать за нашу снаряд. Напишем логику движения снаряда у неё одна строчка, а именно:

*Transform.position += Vector3.up \* BulletSpeed \* Time.deltaTime;*

Эта строчка позволяет нам передвигать пулю в одном направлении.

Далее создадим скрипт EnemyHealthSystem и создадим в нем пустую функцию GetDamage, для того что бы в будущем создать систему здоровья для всех противников и вражеских объектов, у которых есть здоровье. Вернемся к скрипту ShipBullet и дописываем его (Рисунок 5).



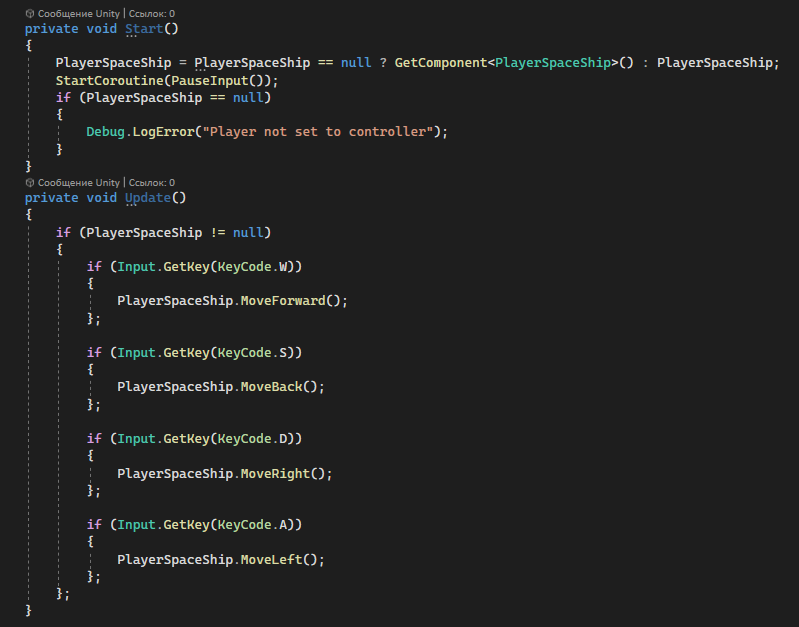
*Рисунок 5. Скрипт ShipBullet.*

Мы дописали логику снаряда игрока. Добавили функцию OntriggerEnter, в ней при попадание снаряда в другой объект происходит проверка на EnemyHealthSystem, если да – то у этого объекта вызывается функция GetDamage.

Следующая функция SetBulletSpeed – эта функция получает значение с плавающей запятой и проверяет, равна ли скорость пули 0, если нет – то он перезаписывает значение.

Написав все функции, которые отвечают за механику космического корабля, переходим к написанию скрипта KeyboardInputs.

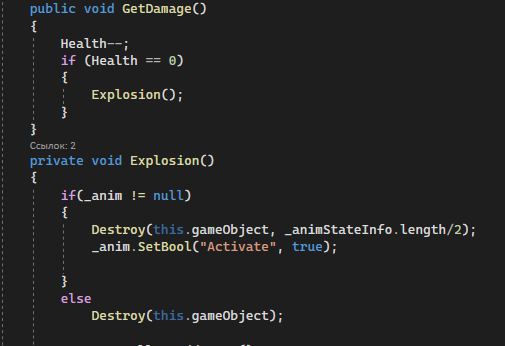
Этот скрипт отвечает за обработку ввода с клавиатуры и будет заставлять двигаться корабль (Рисунок 6)



*Рисунок 6. Скрипт KeyboardInputs.*

Здесь две функции Update и Start. В функции Start мы связываемся со скриптом PlayerSpaceShip, а в функции Update происходит проверка на нажатия клавиш WASD, и при их нажатии вызывается функция корабля для движения.

На этом персонаж написан, переходим к написанию скрипта для противника. Так как создан скрипт EnemyHealthSystem, создаем скрипты DefaultEnemy, Shipbullet. Начнем с EnemyHealthSystem. Там в созданной функции GetDamage сделаем логику уменьшения здоровья. После этого создадим функцию Explosion, которая будет отвечать за смерть объекта (Рисунок 7).



*Рисунок 7. Скрипт EnemyHealthSystem.*

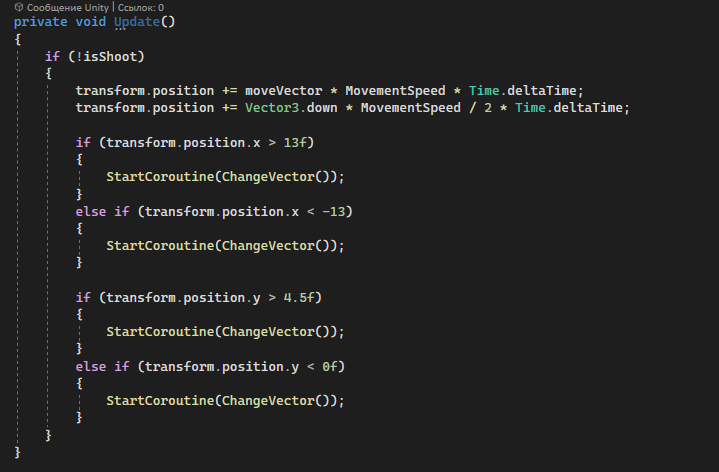
Так же свяжем между собой эти две функции, что бы при попадании снаряда снижалось здоровье, а когда оно достигало нуля происходила смерть. Так же предусмотрим будущее расширение и добавим логику для проигрывания анимации.

После этого переходим к работе над скриптом DefaultEnemy. У него создадим функции TryToShoot, Shoot , ChangeVector.

TryToShoot и Shoot – отвечают за выстрел вражеского корабля.

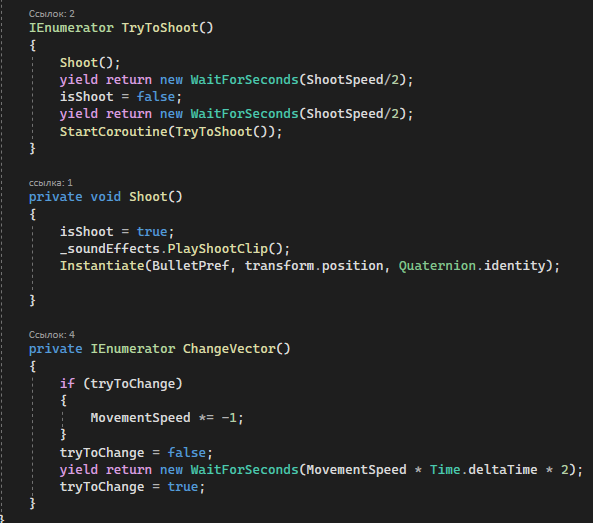
ChangeVector – для изменения направления движения.

В функции Update напишем логику для передвижения противника (Рисунок 8)



*Рисунок 8. Передвижение противника.*

Вражеский корабль передвигается по тому же принципу что и корабль игрока. Функции Shoot реализована так же, как и у корабля персонажа (Рисунок 9).

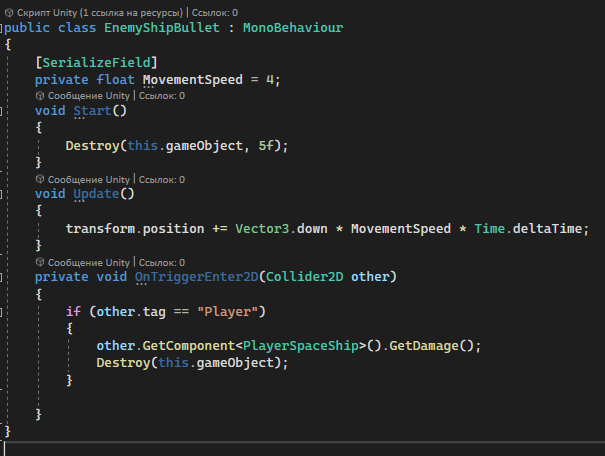


*Рисунок 9. Функции ChangeVector, TryToShoot, Shoot.*

Но функция TryToShoot вызывает функцию Shoot с задержкой что бы вражеский корабль останавливался во время стрельбы, а функция Change Vector – меняет направление, но так же после этого делает паузу, что бы не было багов во время изменения направления.

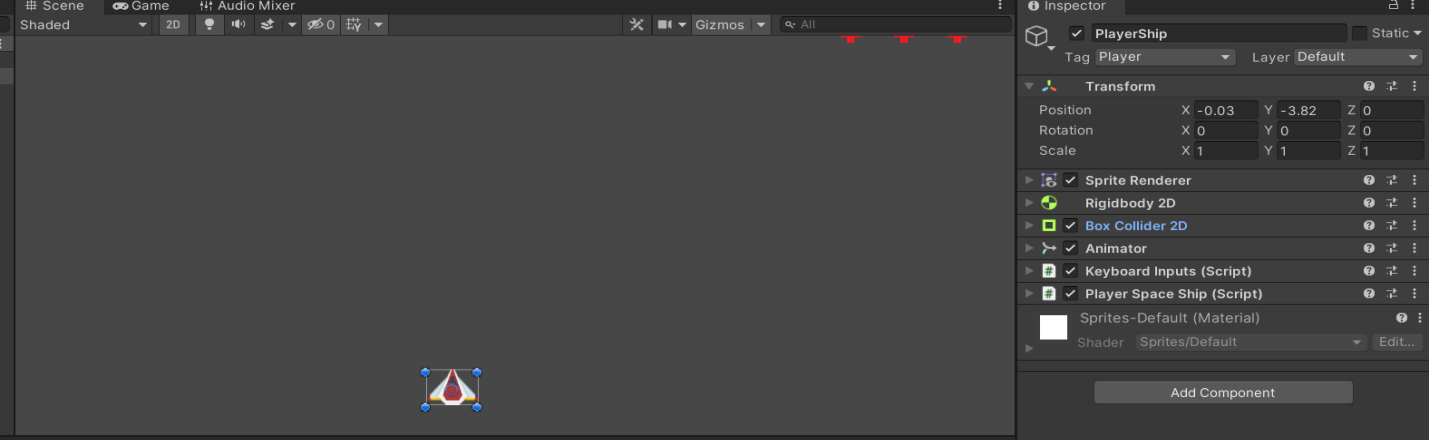
После этого переходим к написанию скрипта EnemyShipBullet.

Тут подобная реализация, как и у ShipBullet только функция GetDamge вызывается у корабля игрока. Поэтому допишем эту функцию у PlayerSpaceShip (Рисунок 10).



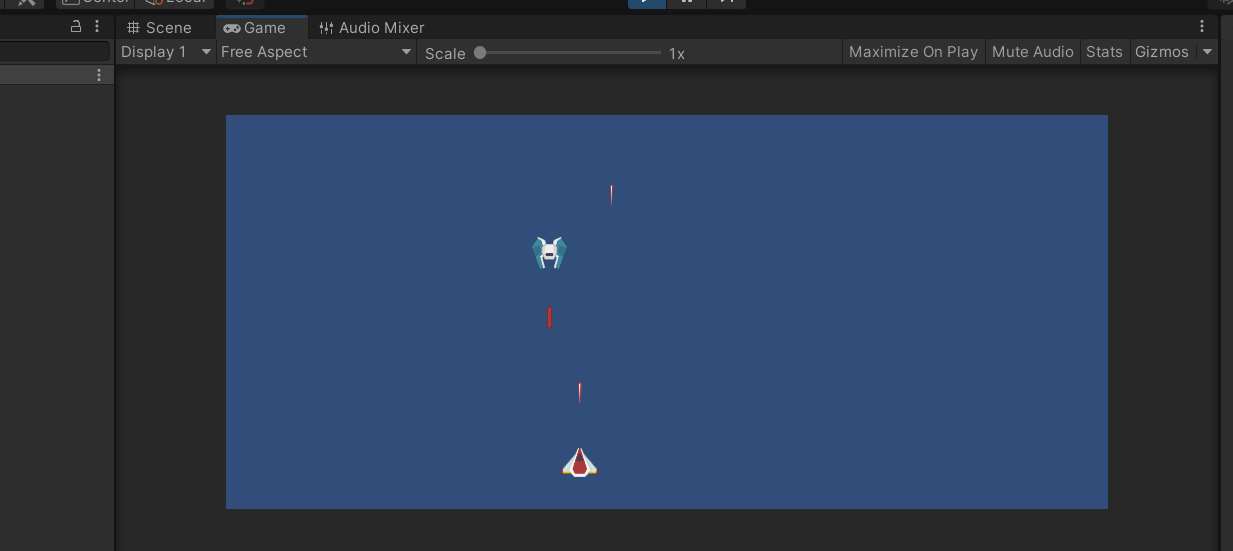
*Рисунок 10. Скрипт EnemyShipBulet.*

На этом скрипты для прототипа написаны, осталось создать объекты и добавить на них эти скрипты. Заранее подготовленную модельку корабля добавляем на сцену. В созданный объект на сцене добавляем скрипты и BoxColider, для того что бы работали функции OntriggerEnter2D (Рисунок 11).



*Рисунок 11. Объект корабля игрока.*

Повторяем данную манипуляцию для противника и делаем проверку (Рисунок 12).



*Рисунок 12. Готовый прототип игры.*

Убедившись в том, что всё корректно работает, а именно персонаж игрока и противника передвигаются, персонажи стреляют, получают урон и умирают, завершаем работу над прототипом.

# Производство компьютерной игры

На этапе производства мы дорабатываем игру, а именно:

1. Заполняем контентом.
2. Добавляем визуальную составляющую.
3. Добавляем звуковое и музыкальное сопровождение.
4. Разрабатываем UI.

Заполнение контентом у нас будет в виде добавление нового противника, а именно босса, и падающих объектов – астероидов. Это придаст игре динамичности. Кроме добавления противника сделаем возможность восполнения игрового корабля здоровьем.

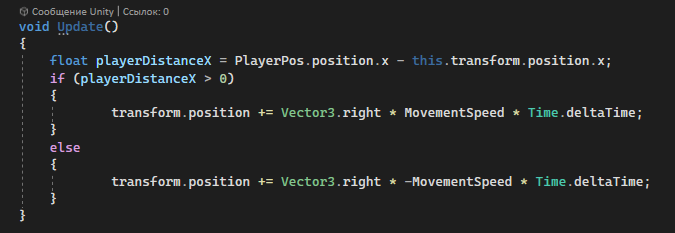
Визуальная составляющая проявляется в виде добавления фона и визуального разнообразия противников, анимацией взрыва.

Звуковое и музыкальное сопровождение в виде звуков стрельбы, взрывов, фоновой музыки.

Разработка UI подразумевает разработку меню, настроек, паузы, отображения здоровья и очков.

Начнем полноценную разработку с добавлением ещё одного типа противников. Создаём скрипт EnemyShip – этот скрипт будет отвечать за работу босса, который будет призываться в конце волны. Реализуем метод стрельбы таким же образом как и в скрипте DefaultEnemy.

После этого начнем реализовывать движение. Он будет преследовать корабль игрока. Для этого мы будем высчитывать разницу между этим кораблём и кораблём игрока, и двигать в соответствии с полученными координатами (Рисунок 13).



*Рисунок 13. Система движение в скрипте EnemyShip.*

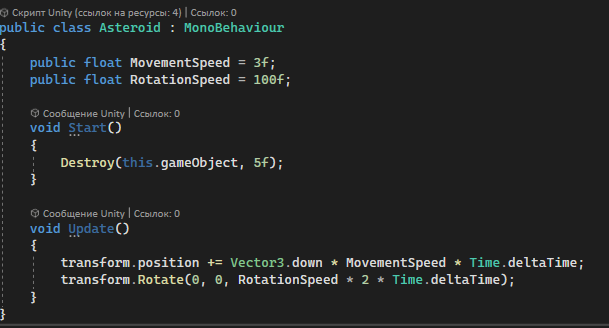
После этого можем создать этого противника на сцене и добавить ему скрипты EnemyShip и EnemyHealthSystem.

После того как мы создали два противника можем перейти к созданию астероидов.

Их мы добавляем для создания атмосферы космоса и увеличения сложности игры.

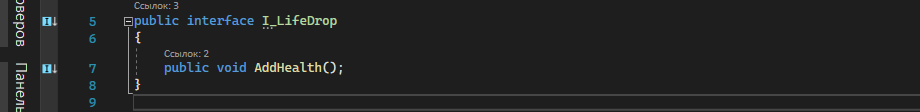
Создаем скрипт Asteroid. В нем будет логика падения вниз и вращения вокруг своей оси.

Так же в методе Start будет содержаться функция которая будет уничтожать этот объект через 5 секунд, что бы не было слишком большой нагрузки на компьютер (Рисунок 14).



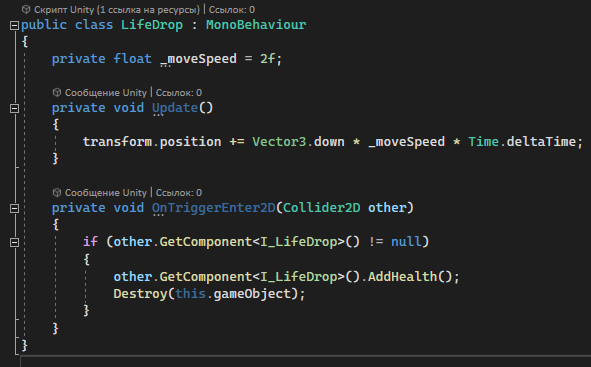
*Рисунок 14. Скрипт Astreroid.*

Для того что бы игрок рано не сдавался в игровой сессии добавим объект который будет пополнять здоровье игрока. Создадим скрипт LifeDrop. Но перед тем как начать работать над этим скриптом создадим интерфейс I\_LifeDrop, что бы расширить функционал PlayerSpaceShip. В этом интерфейсе мы создадим функцию AddHealth (Рисунок 15).



*Рисунок 15. Интерфейс I\_LifeDrop.*

После этого переходим в скрипт LifeDrop. В методе Update по таким же способом что и прежде заставляем этот объект падать вниз. И добавим функцию OnTriggerEnter2D. В нем мы делаем проверку на компонетет I\_LifeDrop. И если проверка успешна, то мы вызываем у этого объекта функцию AddHealth (Рисунок 16).

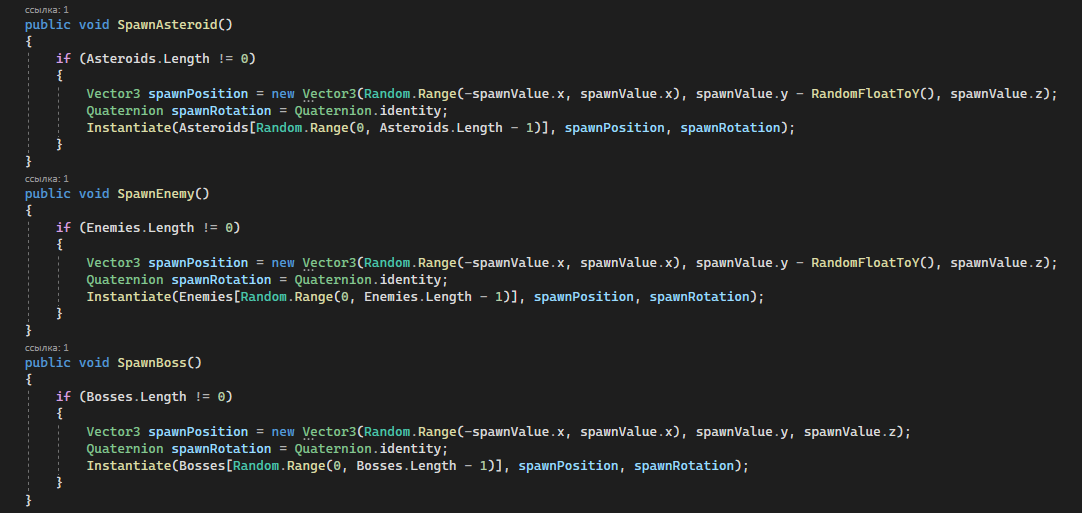


*Рисунок 16. Скрипт LifeDrop.*

После этого реализуем этот интерфейс в PlayerSpaceShip. И в этом методе просто делаем инкремент здоровья.

После этого переходим к созданию скрипта Spawner и GameController. Первый скрипт имеет набор функций по созданию объектов на сцене во время игры, второй – отвечает за их вызов, добавление очков, начало новой волны.

Начнем со скрипта Spawner, в нем создадим функции SpawnAsteroid, SpawnEnemy, SpawnBoss, SpawnHeal. Названия этих функций говорят за себя. Все они имеют одинаковую реализацию (Рисунок 17).

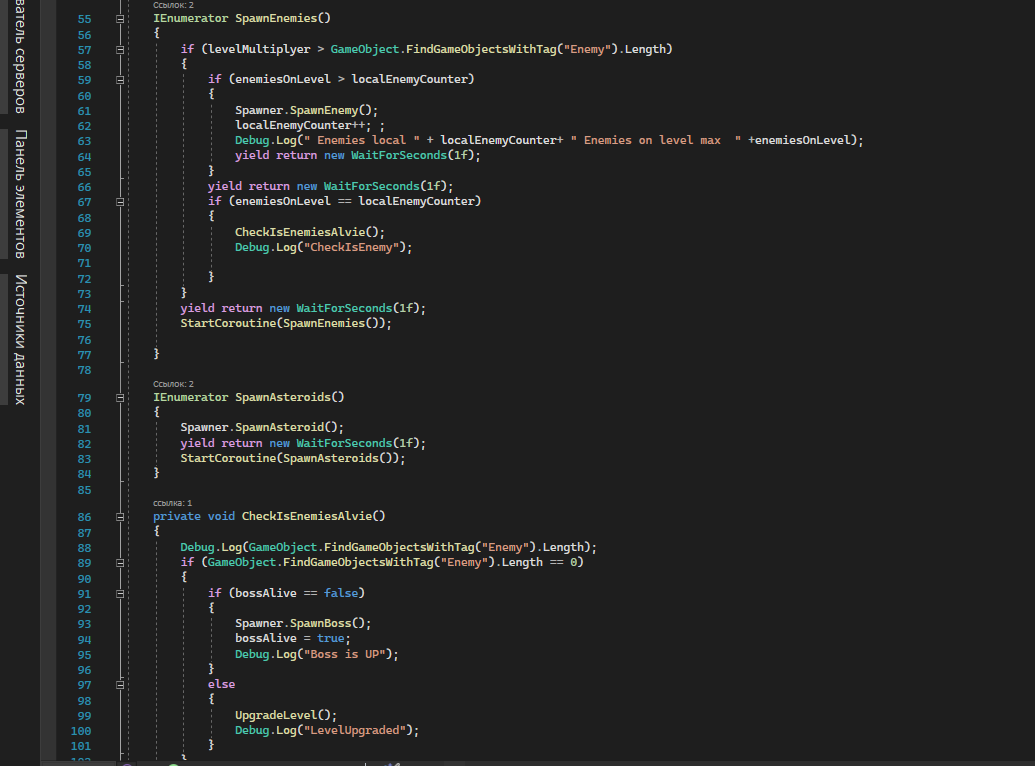


*Рисунок 17. Функции SpawnAsteroid, SpawnEnemy, SpawnBoss.*

После этого перейдем к работе над скриптом GameController. Создадим методы AddScore, UpgradeLevel, SpawnEnemies, SpawnAsteroids, CheckIsEnemyAlive, SpawnHeal, IncreaseSpawnRateHeal.

В методе AddScore увеличиваем очки. В методе UpgradeLevel увеличиваем количество противников на уровне и одновременное количество их, уменьшаем время появление аптечек. В методе SpawnAsteroids вызываем метод SpawnAsteroid скрипта Spawner каждую секунду.

Функции SpawnEnemies и CheckIsEnemiesAlive идет более сложная реализцаия. У метода SpawnEnemies идет проверка на количество уже созданных противников, и если оно не превышает заданное число, то тогда эта функция создает противника. После этого идёт счет количества противников, которые были созданы, и когда это значение равно количеству противников на волне – он создает босса, эта часть реализуется в функции CheckIsEnemiesAlive, она так же проверяет, есть ли противники на сцене, что бы создать босса (Рисунок 18).



*Рисунок 18. Основные методы GameController.*

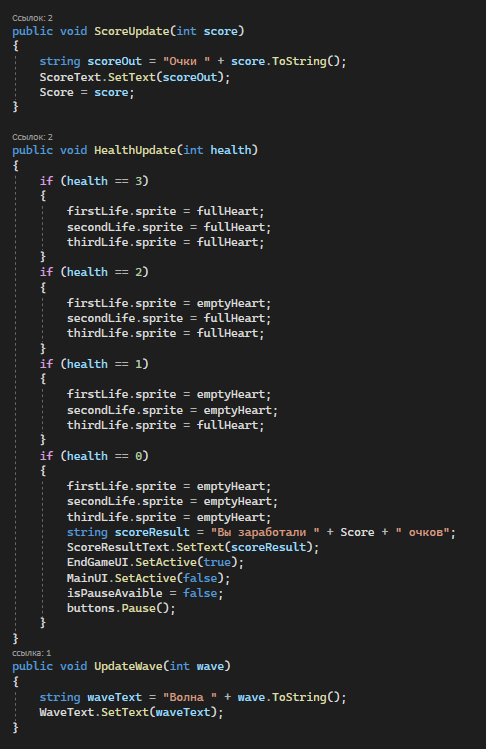
После этого перейдём к созданию интерфейса. Создадим Canvas на сцене, назовём его UI, и в этот объект добавим элементы интерфейса, такие как: количество очков, здоровье и текущая волна (Рисунок 19).



*Рисунок 19. Объект UI.*

После того как мы создали холст на сцене переходим к созданию скрипта, который будет отвечать за UI – InGameUI.

В этом скрипте создаем методы UpdateScore, UpdateHelath, UpdateWave. Эти методы будут получать значения, и с их помощью они будут менять значения на UI (Рисунок 20).



*Рисунок 20. Методы скрита InGameUI.*

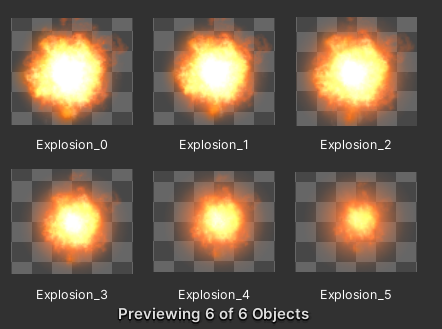
После этого мы привязываем метод UpdateHelath, к методу GetDamage и AddHealth скритпа PlayerSpaceShip.

Метод UpdateScore к методу AddScore скрипта GameController. После этого мы добавлем метод AddScore к EnemyHelathSystem.

После того как мы реализовали игровой интерфейс, вернемся к кораблям и астероидам.

У этих объектов добавляем анимацию взрыва на смерть, что бы корабли и астероиды не пропадали бесследно после смерти, а оставляли визуальный след в виде анимации.

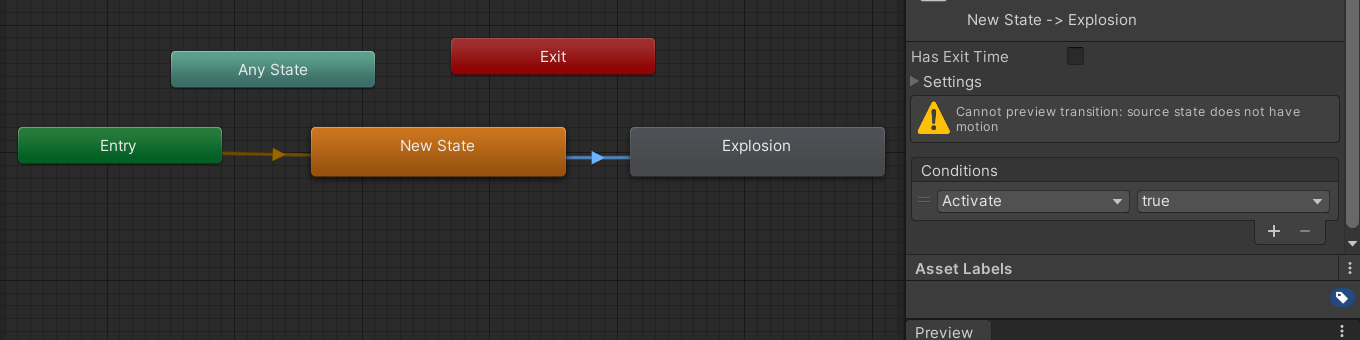
Для этого нужно создать анимацию. Заранее подготовленные спрайты в Photoshop добавляем в проект и перетаскиваем на сцену (Рисунок 21).



*Рисунок 21. Спрайты для анимации.*

После того как мы их добавили на сцену у нас создалась анимация, но для её работы нужно создать AnimationController.

В нем мы передадим функционал работы анимации. А именно то, что она должна проиграть один раз. При условии Activate равно true (Рисунок 22).



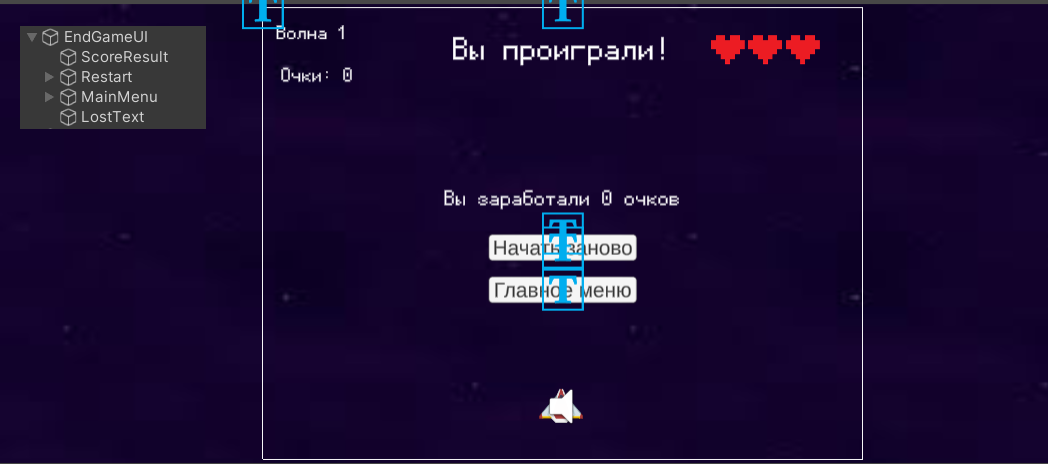
*Рисунок 22. AnimationController взрыва.*

После этого мы добавляем анимацию всем объектом со здоровьем, а именно: вражеским космическим кораблям, космическому кораблю игрока и астероидам. После этого нужно модернизировать код. Мы вызываем этот метод включения анимации в EnemyHealthSystem и PlayerSpaceShip, когда здоровье равно нулю (Рисунок 23).



*Рисунок 23. Метод для включения анимации.*

После этого переходим к доработке UI, а именно создание экрана смерти. Создаем холст, который назовём EndGameUI. В нем будет две кнопки «Начать заново» и «Главное меню». Так же в нем будет два текстовых поля одно предназначено для вывода текста с поражением, другое – отображения очков, набранных в этой игровой сессии (Рисунок 24).

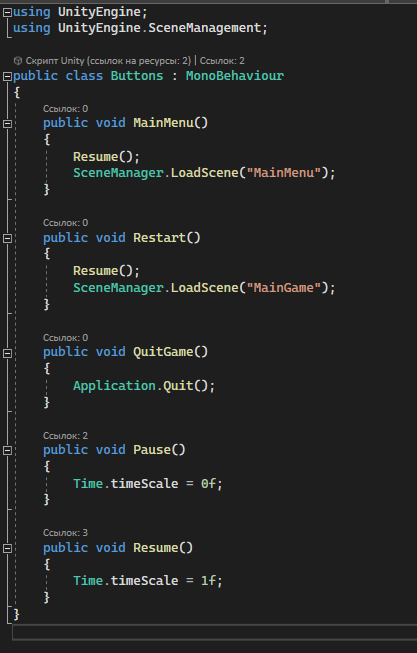


*Рисунок 24. Иерархия EndGameUI и вид на сцене.*

Этот холст отображается, когда здоровье игрока равно нулю и вызывается он в скрипте InGameUI.

Для того чтобы, когда корабль игрока уничтожался, игра останавливалась мы создадим скрипт Buttons, который будет отвечать за кнопки в интерфейсе и в дальнейшем паузу.

В этом скрипте создадим следующие методы: MainMenu, Restart, Pause, Resume, QuitGame. Так же названия методов говорят то, что они реализуют. (Рисунок 25)



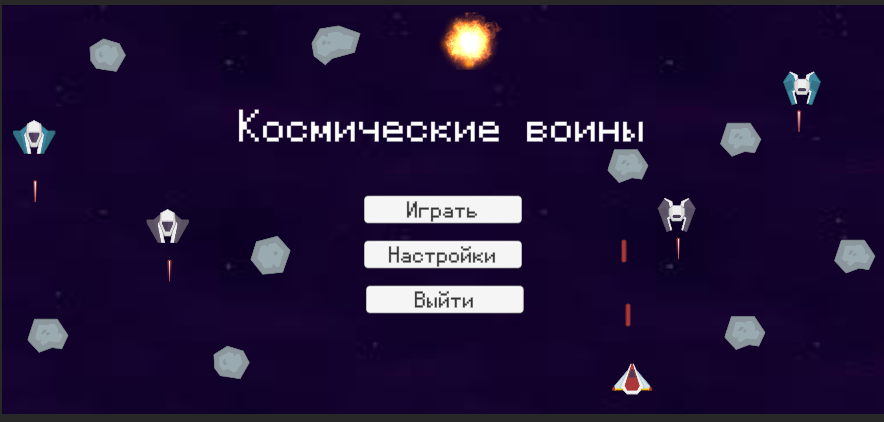
*Рисунок 25. Скрипт Buttons.*

После этого переходим к созданию главного меню. Создаем сцену с названием MainMenu, а эту сохраняем как MainGame (Рисунок 26).



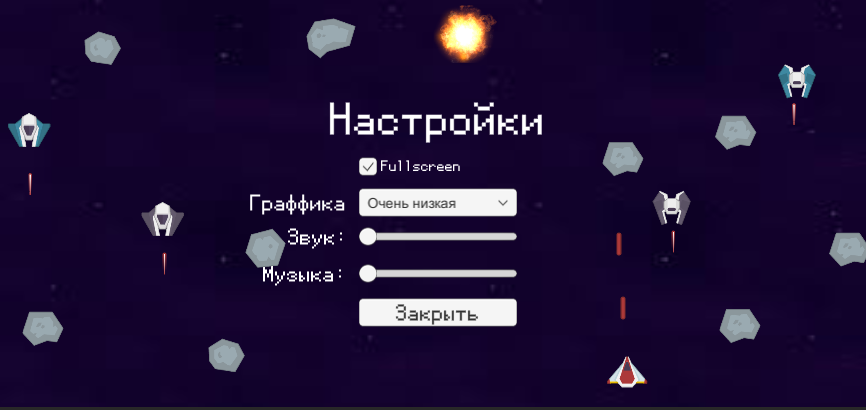
*Рисунок 26. Сцены.*

В главном меню добавляем фон, два холста – один для главного меню, другой для настроек. На холсте главного меню добавляем кнопки: начать игру, настройки и выход. Так же добавляем текст названия игры (Рисунок 27).



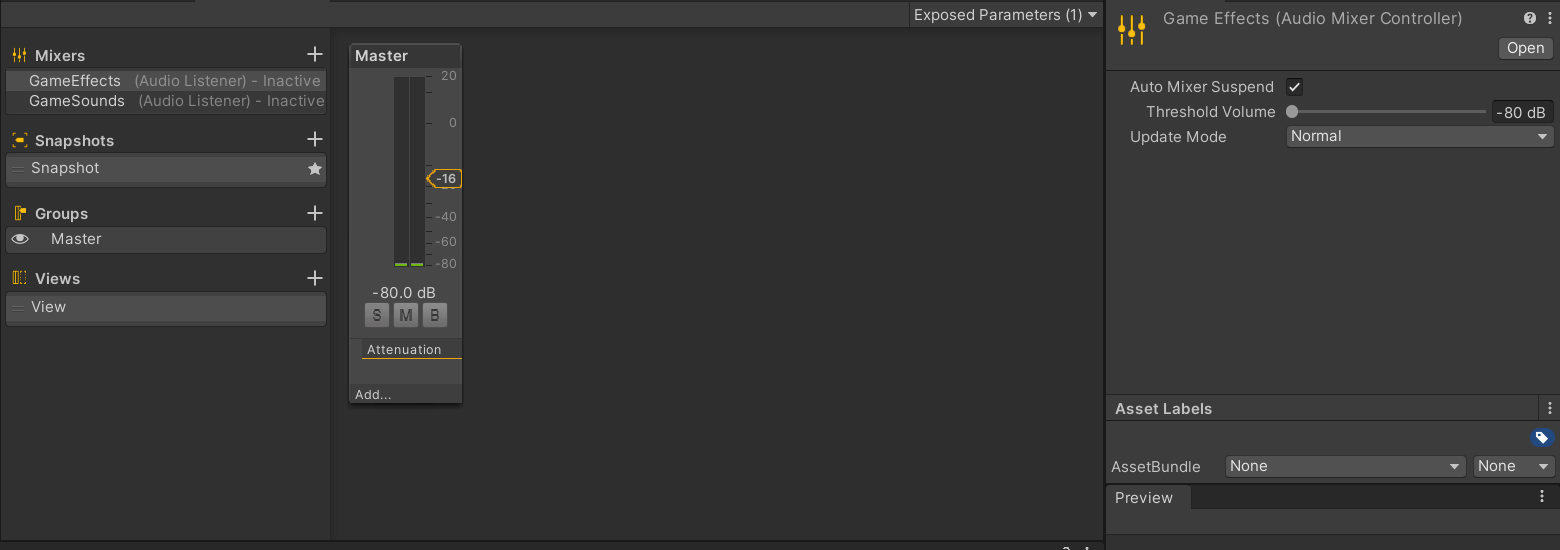
*Рисунок 27. Главное меню.*

В настройках добавляем переключатель для полноэкранного режима, выпадающий список для графики, два ползунка для звуков и музыки. И так же текстовое поле для надписи «Настройки» (Рисунок 28).



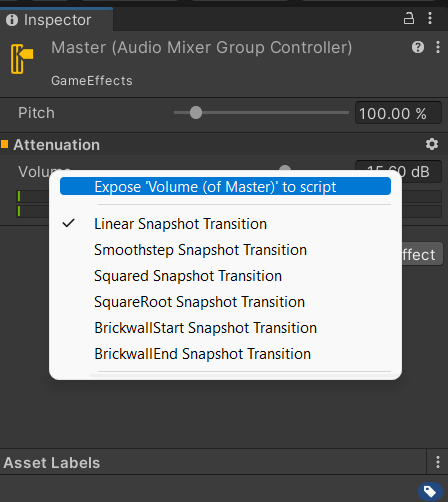
*Рисунок 28. Настройки.*

После этого создаём скрипт для настроек, так как скрипт для кнопок для главного меню уже готов. Перед тем как начать работать над этим скриптом создаём Audio Mixer Controller для звуков и эффектов (Рисунок 29).



*Рисунок 29. Меню Audio Mixer.*

Добавляем параметр для значения Volume и называем его Volume – для громкости музыки, Effects – для громкости звуков (Рисунок 30).

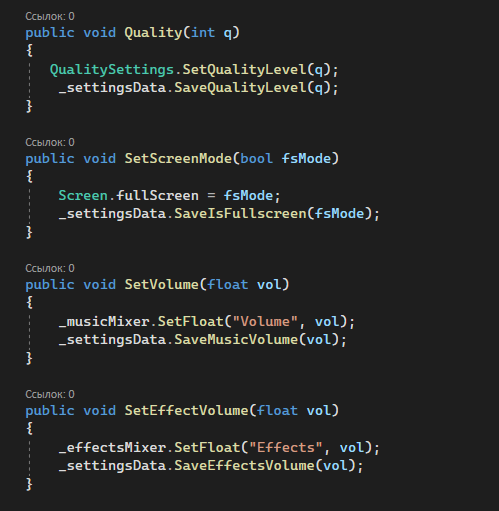


*Рисунок 30. Добавление значений для параметра Volume.*

Переходим к работе над скриптом Settings.

Создаем методы Quality, SetScreenMode, SetVolume, SetEffectsVolume.

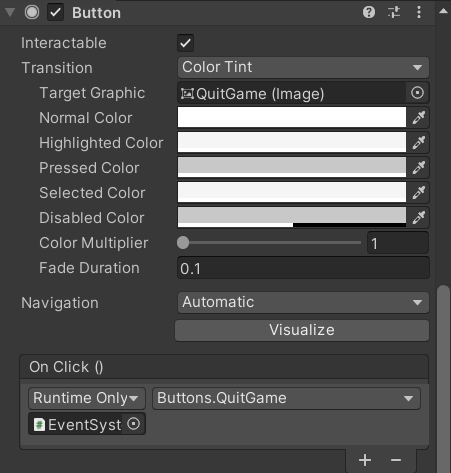
У каждого метода будет входной параметр, который будет передаваться из интерфейса (Рисунок 31).



*Рисунок 31. Скрипт Settings.*

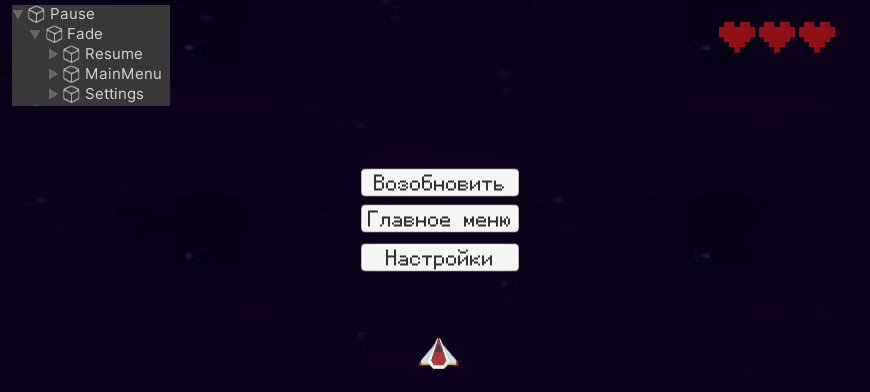
Добавляем этот скрипт на объект, затем привязываем эти методы к кнопкам, переключателям и ползункам.

Для этого переходим ко всем объектам интерфейса и назначаем там методы On Click, On Value Change (Рисунок 32).



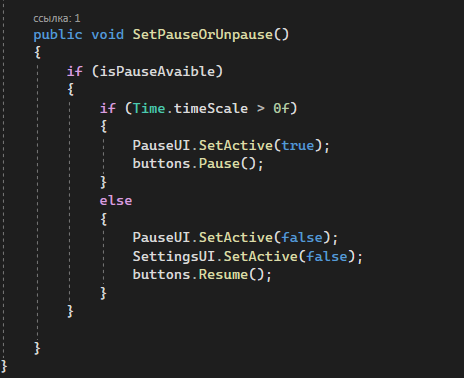
*Рисунок 32. Метод On Click у кнопки «Выход».*

После этого на сцене MainGame добавляем объект паузы. В нем создаем следующие кнопки: «Возобновить», «Главное меню», «Настройки». Так же делаем копию настроек из главного меню (Рисунок 33).



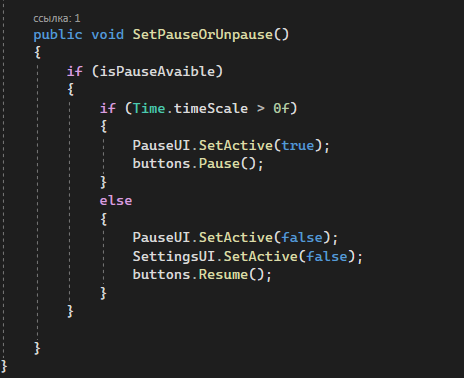
*Рисунок 33. Меню паузы.*

Переходим в InGameUI и создаем метод SetPauseOrUnpause. Для того что бы ставить паузу в игре. В этом методе проверяем Time Scale, если оно больше 0 – возобновляем игру, в другом случае – ставим на паузу. Пауза ставится у нас в скрипте Buttons (Рисунок 34).



*Рисунок 34. Метод SetPauseOrUnpause.*

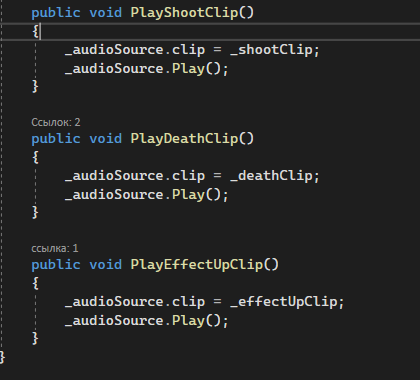
Теперь нам нужно вызывать этот метод. Этот метод будет вызываться при нажатии клавиши на клавиатуре. А скрипт который отвечает за считывание нажатия клавиш – это KeyboardInputs. Во время того как timeScale равен нулю не работает Update, то считывание этой клавиши будет в отдельной функции PauseInput. У него будет проверка на нажатие клавиши и InGameUI. (Рисунок 35)



*Рисунок 35. Метод PauseInput.*

После того как мы реализовали интерфейсную часть игры переходим к звуковому и музыкальному сопровождению. Для этого создадим два скрипта первый для фоновой музыки - BackGroundMusic, второй для звуков - SoundEffects.

В скрипте SoundEffects создадим три метода: PlayShootClip, PlayeDeathClip, PlayEffectUpClip. Эти методы имеют одинаковую реализацию (Рисунок 36).

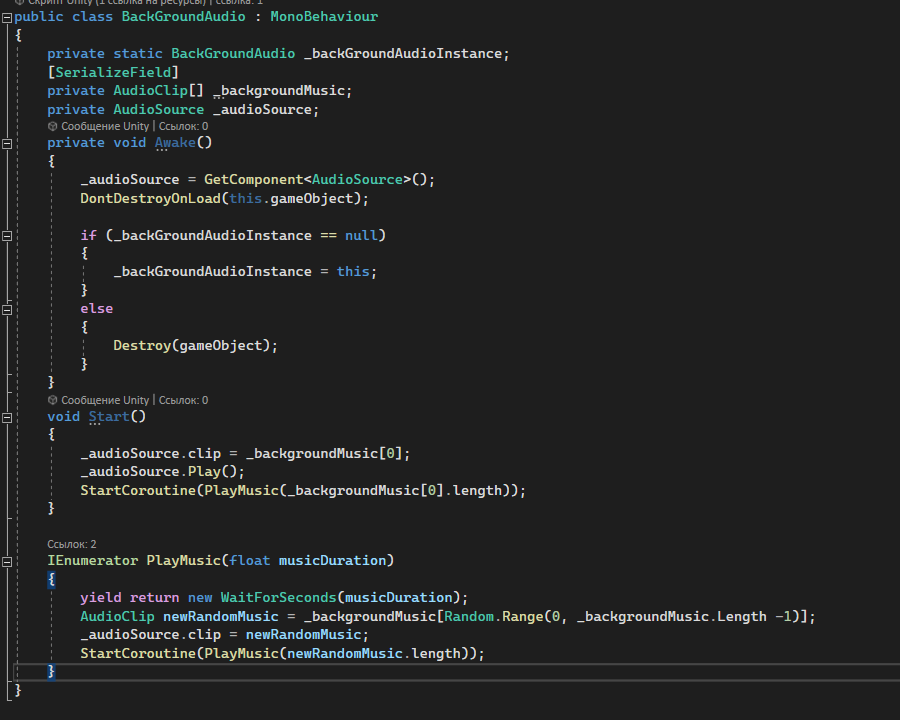


*Рисунок 36. Скрипт SoundEffects.*

В скрипте BackGroundMusic используется другая реализация. Объект с этим скриптом создаётся в главном меню и переносится на другую сцену, так что мы используем метод DontDestroyOnLoad.

Так же так как фоновая музыка меняется, по окончании одной звуковой дорожки запускается другая звуковая дорожка.

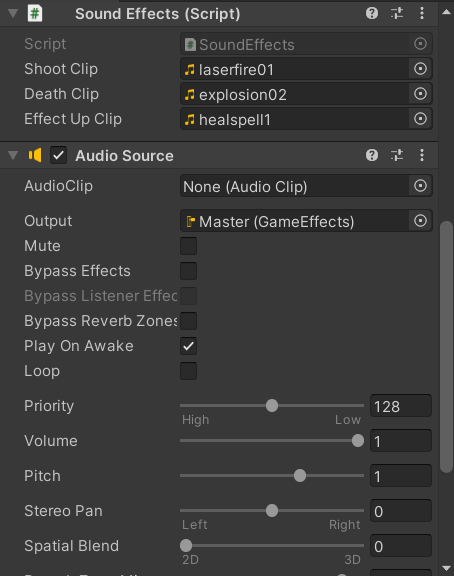
Это уже передаётся в методе PlayMusic с входным значением, являющейся длиной звуковой дорожки (Рисунок 37).



*Рисунок 37. Скрипт BackGroundMusic.*

После написания этих скриптов нужно их интегрировать. Для BackGroudMusic просто создаётся объект и добавляется в него скрипт.

В свою очередь интеграция SoundEffects требует добавить на каждый объект, который будет воспроизводить звуки, этот скрипт и Audio Source (Рисунок 38).



*Рисунок 38. Добавление в объект SoundEffects и Audio Source.*

Основная логика игры реализована. Аналогично создаются вариации противников и астероидов, согласно описанным выше приёмам. После этого игра приобретает законченный вид.

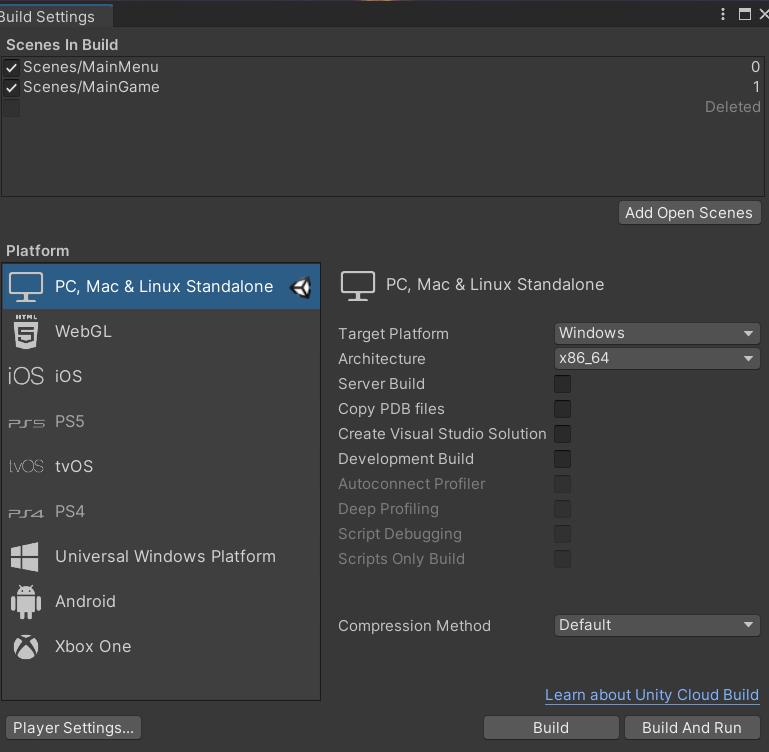
# Функциональное описание разработки

# Конфигурирование

В качестве конфигураций проекта следующие параметры в Build Settings:

1. Добавляются сцены главного меню и сцены игры в Scenes in Build.
2. Платформа выбирается PC, Mac, & Linux Standalone.
3. Target Platform – Windows.
4. Архитектура – x86\_64.
5. Compression Method – Default.

Все остальные параметры отключаются (Рисунок 39).



*Рисунок 39. Конфигурация проекта.*

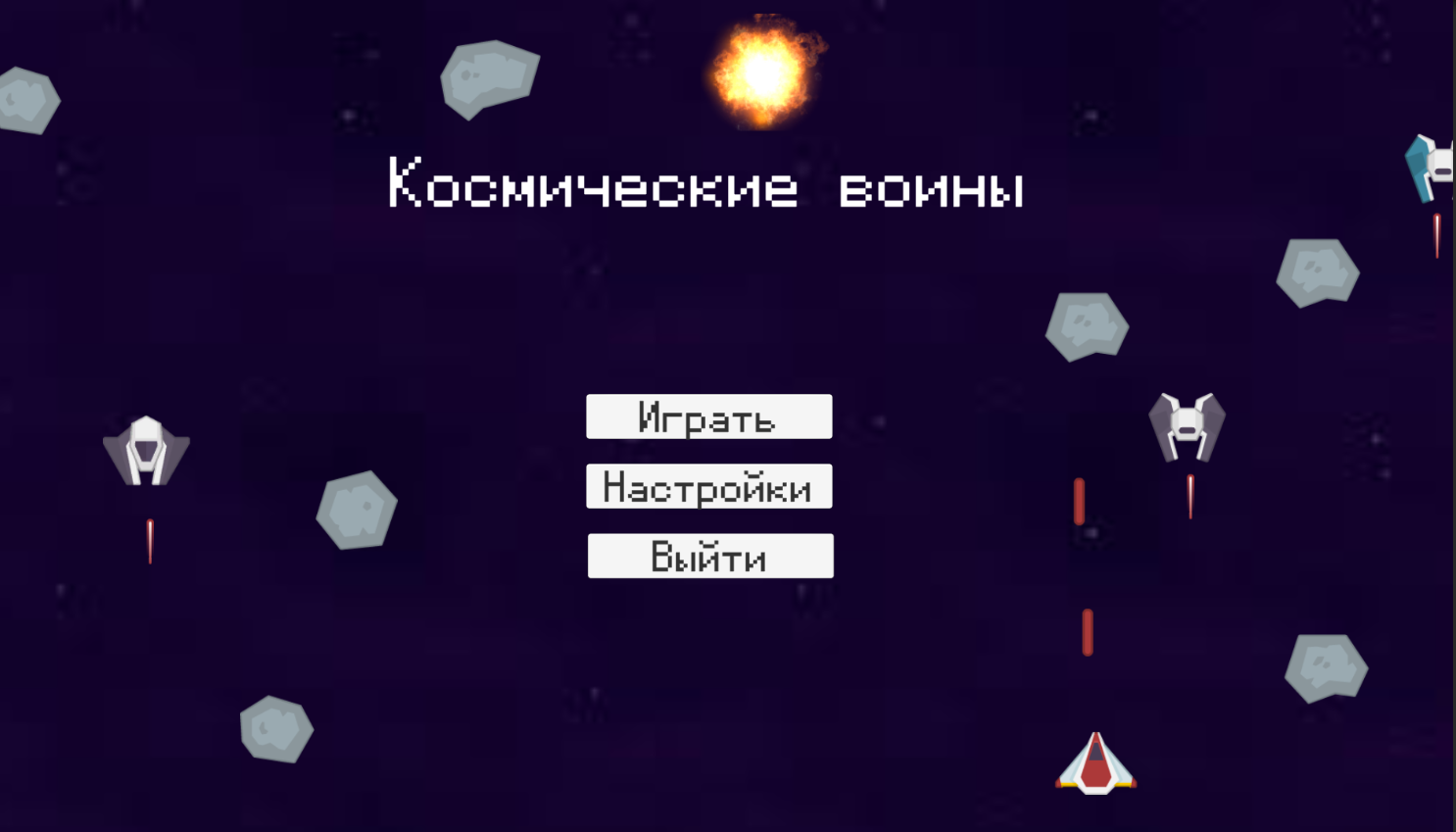
# Клиентская часть

Цель игры - продержаться большое количество волн (Рисунок 43).

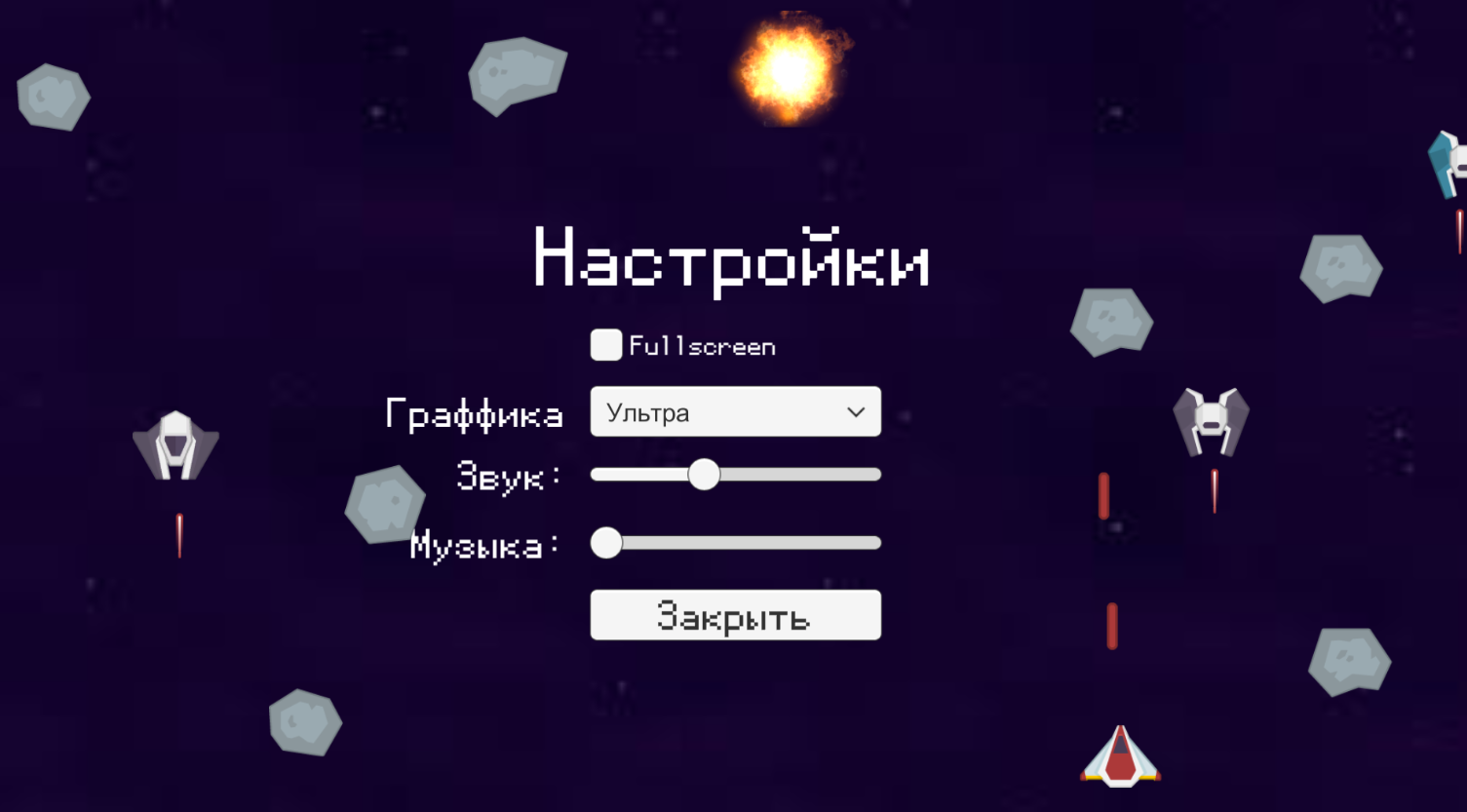
Так как игра не имеет конца, как в классической аркаде, дополнительного сюжета не требуется в игру, ведь в играх такого жанра дополнительный сюжет может испортить впечатление от игры.

Каждый способен сам придумать себе предысторию, что положительно влияет на восприятие игры.

Главное меню и меню настроек выполнено в стиле минимализм с Pixel Art шрифтом (Рисунок 40-41).



*Рисунок 40. Главное меню.*



*Рисунок 41. Меню настроек.*

****

*Рисунок 42. Игровой процесс.*

# Инструкция по использованию программного продукта

В ходе разработки был разработан интуитивно понятный интерфейс.

Удобное управление WASD, которое стало стандартным во всех играх, главное меню не насыщено большим количеством ненужной информации.

Учитывая вышеперечисленное можно с уверенностью сказать, что пользователь самообучается в процессе игры и дополнительных инструкций не требуется.

# 2.6 Этап тестирования

Есть множество видов тестирования, но в рамках ВКР решено использовать функциональное тестирования.

Функциональное тестирование — это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности ПО в определённых условиях решать задачи, нужные пользователям.

Приложение будет тестироваться с помощью тестирования черного ящика. Тестирование чёрного ящика или поведенческое тестирование — стратегия тестирования функционального поведения объекта с точки зрения внешнего мира, при котором не используется знание о внутреннем устройстве тестируемого объекта. Проверяется поэтапно вся игра, начиная с элементов интерфейса заканчивая проверками внутриигровых событий. В случае нахождения ошибок во время тестирования ищется ошибка в проекте и исправляется. В данной ситуации ошибок во время тестирования не было найдено, приложение успешно прошло этап тестирования.

# ЗАКЛЮЧЕНИЕ

В данной работе в соответствии с заданием на выпускную квалификационную работу была разработана 2D игра в жанре «Аркада» на движке Unity, а также описаны этапы проектирования и разработки. Во время анализа доступных источников было проведено исследование понятия видеоигра, во время которогобыла проведена классификация видеоигр. Для решения поставленной цели были выполнены следующие задачи:

1. Изучены и проанализированы предметная область и теория разработки видеоигр.
2. Проанализирован и обоснован выбор инструментария для разработки.
3. Продумана концепция игры.
4. Разработаны требования к проектируемой видеоигре.
5. Разработан пользовательский интерфейс (UI).
6. Разработан прототип и реализованы основные механики игры.
7. Выбраны музыкальное и звуковое сопровождение игрового процесса.
8. Протестировано приложение.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

* 1. Unity: Wikipedia Информационный портал [Электронный ресурс]. режим доступа - [https://ru.wikipedia.org/wiki/ Unity\_(игровой\_движок)](https://ru.wikipedia.org/wiki/%20Unity_(%D0%B8%D0%B3%D1%80%D0%BE%D0%B2%D0%BE%D0%B9_%D0%B4%D0%B2%D0%B8%D0%B6%D0%BE%D0%BA)) – Загл. с экрана.
  2. Видеоигра: Wikipedia Информационный портал [Электронный ресурс]. режим доступа - [https://ru.wikipedia.org/wiki/Видеоигра](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D0%B4%D0%B5%D0%BE%D0%B8%D0%B3%D1%80%D0%B0) – Загл. с экрана.
  3. Документация по Unity: Wikipedia [Электронный ресурс]. режим доступа - <https://docs.unity3d.com/> – Загл. с экрана.
  4. Adobe Illustrator: Wikipedia Информационный портал [Электронный ресурс]. режим доступа - [https://ru.wikipedia.org/wiki/Adobe\_Illustrator](https://ru.wikipedia.org/wiki/Adobe_Illustrator%20)  – Загл. с экрана.
  5. Visual Studio: Wikipedia Информационный портал [Электронный ресурс]. режим доступа - https://ru.wikipedia.org/wiki/Visual\_Studio – Загл. с экрана.
  6. Язык программирования C#: Wikipedia Информационный портал [Электронный ресурс]. режим доступа - <https://ru.wikipedia.org/wiki/C_Sharp> – Загл. с экрана.
  7. Язык программирования C++: Wikipedia Информационный портал [Электронный ресурс]. режим доступа - https://ru.wikipedia.org/wiki/C++ – Загл. с экрана
  8. Adobe Photoshop: Wikipedia Информационный портал [Электронный ресурс]. режим доступа - [https://ru.wikipedia.org/wiki/Adobe\_Photoshop](https://ru.wikipedia.org/wiki/Adobe_Photoshop%20)  – Загл. с экрана.
  9. Unreal Engine 4: Wikipedia Информационный портал [Электронный ресурс]. режим доступа - https://ru.wikipedia.org/wiki/Unreal\_Engine – Загл. с экрана.

# ПРИЛОЖЕНИЕ

**Исходные коды скриптов выпускной квалификационной работы**

**Скрипт PlayerSpaceShip.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PlayerSpaceShip : MonoBehaviour, I\_LifeDrop

{

[SerializeField]

private float MoveSpeed = 1f;

private int Health = 3;

private bool isDeath;

[SerializeField]

private InGameUI GameUI;

private Rigidbody2D \_rigidbody;

[SerializeField]

private GameObject bulletPrefab;

private Animator \_anim;

private AnimatorStateInfo \_animStateInfo;

private SoundEffects \_soundEffects;

private void Awake()

{

\_rigidbody = GetComponent<Rigidbody2D>();

\_anim = GetComponent<Animator>();

\_animStateInfo = \_anim.GetCurrentAnimatorStateInfo(0);

isDeath = false;

\_soundEffects = GetComponent<SoundEffects>();

}

private void Start()

{

StartCoroutine(Shoot());

}

public void MoveForward()

{

if (transform.position.y < 4.5f)

{

if (!isDeath)

transform.position += (Vector3.up \* MoveSpeed \* Time.deltaTime);

}

}

public void MoveBack()

{

if (transform.position.y > -4.5f)

{

if (!isDeath)

transform.position += (Vector3.up \* -MoveSpeed \* Time.deltaTime);

}

}

public void MoveRight()

{

if (transform.position.x < 10.85f)

{

if (!isDeath)

transform.position += (Vector3.right \* MoveSpeed \* Time.deltaTime);

}

}

public void MoveLeft()

{

if (transform.position.x > -10.85f)

{

if (!isDeath)

transform.position += (Vector3.right \* -MoveSpeed \* Time.deltaTime);

}

}

IEnumerator Shoot()

{

\_soundEffects.PlayShootClip();

if (!isDeath)

{

Instantiate(bulletPrefab, GameObject.Find("Position").transform.position, Quaternion.identity);

}

yield return new WaitForSeconds(0.5f);

StartCoroutine(Shoot());

}

public void GetDamage()

{

Health--;

GameUI.HealthUpdate(Health);

if (Health == 0)

{

Death();

isDeath = true;

}

}

private void Death()

{

if (\_anim != null)

{

\_anim.SetBool("Activate", true);

\_soundEffects.PlayDeathClip();

Destroy(this.gameObject, \_animStateInfo.length);

}

}

public void AddHealth()

{

if (Health < 3)

{

Health++;

GameUI.HealthUpdate(Health);

\_soundEffects.PlayEffectUpClip();

}

}

}

**Скрипт ShipBullet.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class ShipBullet : MonoBehaviour

{

public float BulletSpeed = 4;

void Start()

{

Destroy(this.gameObject, 5f);

}

void Update()

{

transform.position += (Vector3.up \* BulletSpeed \* Time.deltaTime);

}

private void OnTriggerEnter2D(Collider2D other)

{

if (other.GetComponent<EnemyHealthSystem>() != null)

{

other.GetComponent<EnemyHealthSystem>().GetDamage();

Destroy(this.gameObject);

}

}

public void SetBulletSpeed(float bulletSpeed)

{

if (bulletSpeed != 0)

{

BulletSpeed = bulletSpeed;

}

}

}

**Скрипт KeyboardInputs.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class KeyboardInputs : MonoBehaviour {

public PlayerSpaceShip PlayerSpaceShip;

[SerializeField]

private InGameUI gameUI;

private void Start()

{

PlayerSpaceShip = PlayerSpaceShip == null ? GetComponent<PlayerSpaceShip>() : PlayerSpaceShip;

StartCoroutine(PauseInput());

if (PlayerSpaceShip == null)

{

Debug.LogError("Player not set to controller");

}

}

private void Update()

{

if (PlayerSpaceShip != null)

{

if (Input.GetKey(KeyCode.W))

{

PlayerSpaceShip.MoveForward();

};

if (Input.GetKey(KeyCode.S))

{

PlayerSpaceShip.MoveBack();

};

if (Input.GetKey(KeyCode.D))

{

PlayerSpaceShip.MoveRight();

};

if (Input.GetKey(KeyCode.A))

{

PlayerSpaceShip.MoveLeft();

};

};

}

IEnumerator PauseInput()

{

if (Input.GetKeyUp(KeyCode.Escape))

{

if (gameUI != null)

gameUI.SetPauseOrUnpause();

else

Debug.LogError("GameUI not assinged");

}

yield return new WaitUntil(() => Input.GetKeyUp(KeyCode.Escape));

StartCoroutine(PauseInput());

}

}

**Скрипт EnemyHealthSystem.cs**

using UnityEngine;

public class EnemyHealthSystem : MonoBehaviour

{

[SerializeField]

private int Health = 3;

private Animator \_anim;

private AnimatorStateInfo \_animStateInfo;

private SoundEffects \_soundEffects;

private GameController GameController;

void Start()

{

\_anim = GetComponent<Animator>();

\_animStateInfo = \_anim.GetCurrentAnimatorStateInfo(0);

GameController = GameObject.Find("GameRules").GetComponent<GameController>();

\_soundEffects = GetComponent<SoundEffects>();

}

public void GetDamage()

{

Health--;

if (Health == 0)

{

Explosion();

}

}

private void Explosion()

{

if(\_anim != null)

{

Destroy(this.gameObject, \_animStateInfo.length/2);

\_anim.SetBool("Activate", true);

}

else

Destroy(this.gameObject);

GameController.AddScore();

\_soundEffects.PlayDeathClip();

}

private void OnTriggerEnter2D(Collider2D collision)

{

if (collision.tag == "Player")

{

Explosion();

collision.GetComponent<PlayerSpaceShip>().GetDamage();

}

}

}

**Скрипт EnemyShipBullet.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class EnemyShipBullet : MonoBehaviour

{

[SerializeField]

private float MovementSpeed = 4;

void Start()

{

Destroy(this.gameObject, 5f);

}

void Update()

{

transform.position += Vector3.down \* MovementSpeed \* Time.deltaTime;

}

private void OnTriggerEnter2D(Collider2D other)

{

if (other.tag == "Player")

{

other.GetComponent<PlayerSpaceShip>().GetDamage();

Destroy(this.gameObject);

}

}

}

**Скрипт DefaultEnemy.cs**

using System.Collections;

using UnityEngine;

public class DefaultEnemy : MonoBehaviour

{

public GameObject BulletPref;

public float ShootSpeed = 1.5f;

public float MovementSpeed = 4f;

private Vector3 moveVector;

private bool isShoot;

private bool tryToChange;

private SoundEffects \_soundEffects;

// Start is called before the first frame update

void Start()

{

\_soundEffects = GetComponent<SoundEffects>();

tryToChange = true;

StartCoroutine(TryToShoot());

if (transform.position.x > 0)

{

moveVector = Vector3.right \* -1;

}

else

{

moveVector = Vector3.right;

}

}

private void Update()

{

if (!isShoot)

{

transform.position += moveVector \* MovementSpeed \* Time.deltaTime;

transform.position += Vector3.down \* MovementSpeed / 2 \* Time.deltaTime;

if (transform.position.x > 10f)

{

StartCoroutine(ChangeVector());

}

else if (transform.position.x < -10)

{

StartCoroutine(ChangeVector());

}

if (transform.position.y > 4.5f)

{

StartCoroutine(ChangeVector());

}

else if (transform.position.y < 0f)

{

StartCoroutine(ChangeVector());

}

}

}

IEnumerator TryToShoot()

{

Shoot();

yield return new WaitForSeconds(ShootSpeed/2);

isShoot = false;

yield return new WaitForSeconds(ShootSpeed/2);

StartCoroutine(TryToShoot());

}

private void Shoot()

{

isShoot = true;

\_soundEffects.PlayShootClip();

Instantiate(BulletPref, transform.position, Quaternion.identity);

}

private IEnumerator ChangeVector()

{

if (tryToChange)

{

MovementSpeed \*= -1;

}

tryToChange = false;

yield return new WaitForSeconds(MovementSpeed \* Time.deltaTime \* 2);

tryToChange = true;

}

}

**Скрипт EnemyShip.cs**

using System.Collections;

using UnityEngine;

public class EnemyShip : MonoBehaviour

{

[SerializeField]

private float MovementSpeed = 4.5f;

public float ShootSpeed = .3f;

private Transform PlayerPos;

public GameObject BulletPrefab;

private Rigidbody2D \_rigidbody;

private SoundEffects \_soundEffect;

void Start()

{

\_soundEffect = GetComponent<SoundEffects>();

\_rigidbody = GetComponent<Rigidbody2D>();

PlayerPos = GameObject.Find("PlayerShip").transform;

StartCoroutine(Shoot());

}

void Update()

{

float playerDistanceX = PlayerPos.position.x - this.transform.position.x;

if (playerDistanceX > 0)

{

transform.position += Vector3.right \* MovementSpeed \* Time.deltaTime;

}

else

{

transform.position += Vector3.right \* -MovementSpeed \* Time.deltaTime;

}

}

IEnumerator Shoot()

{

Instantiate(BulletPrefab, this.transform.position, Quaternion.identity);

\_soundEffect.PlayShootClip();

yield return new WaitForSeconds(ShootSpeed);

StartCoroutine(Shoot());

}

}

**Скрипт Asteroid.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Asteroid : MonoBehaviour

{

public float MovementSpeed = 3f;

public float RotationSpeed = 100f;

void Start()

{

Destroy(this.gameObject, 5f);

}

void Update()

{

transform.position += Vector3.down \* MovementSpeed \* Time.deltaTime;

transform.Rotate(0, 0, RotationSpeed \* 2 \* Time.deltaTime);

}

}

**Скрипт LifeDrop.cs**

using UnityEngine;

public class LifeDrop : MonoBehaviour

{

private float \_moveSpeed = 2f;

private void Update()

{

transform.position += Vector3.down \* \_moveSpeed \* Time.deltaTime;

}

private void OnTriggerEnter2D(Collider2D other)

{

if (other.GetComponent<I\_LifeDrop>() != null)

{

other.GetComponent<I\_LifeDrop>().AddHealth();

Destroy(this.gameObject);

}

}

}

**Интерфейс I\_LifeDrop.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public interface I\_LifeDrop

{

public void AddHealth();

}

**Скрипт Spawner.cs**

using UnityEngine;

public class Spawner : MonoBehaviour

{

[SerializeField]

private Vector3 spawnValue;

public GameObject[] Enemies;

public GameObject[] Asteroids;

public GameObject[] Bosses;

public GameObject[] Heal;

public void SpawnAsteroid()

{

if (Asteroids.Length != 0)

{

Vector3 spawnPosition = new Vector3(Random.Range(-spawnValue.x, spawnValue.x), spawnValue.y, spawnValue.z);

Quaternion spawnRotation = Quaternion.identity;

Instantiate(Asteroids[Random.Range(0, Asteroids.Length - 1)], spawnPosition, spawnRotation);

}

}

public void SpawnEnemy()

{

if (Enemies.Length != 0)

{

Vector3 spawnPosition = new Vector3(Random.Range(-spawnValue.x, spawnValue.x), spawnValue.y, spawnValue.z);

Quaternion spawnRotation = Quaternion.identity;

Instantiate(Enemies[Random.Range(0, Enemies.Length - 1)], spawnPosition, spawnRotation);

}

}

public void SpawnBoss()

{

if (Bosses.Length != 0)

{

Vector3 spawnPosition = new Vector3(Random.Range(-spawnValue.x, spawnValue.x), spawnValue.y, spawnValue.z);

Quaternion spawnRotation = Quaternion.identity;

Instantiate(Bosses[Random.Range(0, Bosses.Length - 1)], spawnPosition, spawnRotation);

}

}

public void SpawnHeal()

{

if (Heal.Length != 0)

{

Vector3 spawnPosition = new Vector3(Random.Range(-spawnValue.x, spawnValue.x), spawnValue.y, spawnValue.z);

Quaternion spawnRotation = Quaternion.identity;

Instantiate(Heal[Random.Range(0, Heal.Length - 1)], spawnPosition, spawnRotation);

}

}

}

**Скрипт GameController.cs**

using System.Collections;

using UnityEngine;

public class GameController : MonoBehaviour

{

private Spawner Spawner;

private InGameUI GameUI;

private int Score;

private int Level = 1, levelMultiplyer = 1;

private int enemiesOnLevel = 3;

private int localEnemyCounter;

private bool bossAlive;

private float healSpawnRate = 15f;

private void Start()

{

GameUI = gameObject.GetComponent<InGameUI>();

Spawner = Spawner == null ? GetComponent<Spawner>() : Spawner;

if (Spawner == null)

{

Debug.LogError("Enemy Spawner not set to game controler");

}

bossAlive = false;

StartCoroutine(SpawnEnemies());

StartCoroutine(SpawnAsteroids());

StartCoroutine(SpawnHeal());

}

public void AddScore()

{

Score += 5;

GameUI.ScoreUpdate(Score);

}

private void UpgradeLevel()

{

Level += 1;

levelMultiplyer += 2;

enemiesOnLevel += 5;

localEnemyCounter = 0;

bossAlive = false;

Score += 20;

Debug.Log(Score);

IncreaseSpawnRateHeal();

GameUI.UpdateWave(Level);

GameUI.ScoreUpdate(Score);

}

IEnumerator SpawnEnemies()

{

if (levelMultiplyer > GameObject.FindGameObjectsWithTag("Enemy").Length)

{

if (enemiesOnLevel > localEnemyCounter)

{

Spawner.SpawnEnemy();

localEnemyCounter++; ;

Debug.Log(" Enemies local " + localEnemyCounter+ " Enemies on level max " +enemiesOnLevel);

yield return new WaitForSeconds(1f);

}

yield return new WaitForSeconds(1f);

if (enemiesOnLevel == localEnemyCounter)

{

CheckIsEnemiesAlvie();

Debug.Log("CheckIsEnemy");

}

}

yield return new WaitForSeconds(1f);

StartCoroutine(SpawnEnemies());

}

IEnumerator SpawnAsteroids()

{

Spawner.SpawnAsteroid();

yield return new WaitForSeconds(1f);

StartCoroutine(SpawnAsteroids());

}

private void CheckIsEnemiesAlvie()

{

Debug.Log(GameObject.FindGameObjectsWithTag("Enemy").Length);

if (GameObject.FindGameObjectsWithTag("Enemy").Length == 0)

{

if (bossAlive == false)

{

Spawner.SpawnBoss();

bossAlive = true;

Debug.Log("Boss is UP");

}

else

{

UpgradeLevel();

Debug.Log("LevelUpgraded");

}

}

}

IEnumerator SpawnHeal()

{

yield return new WaitForSeconds(healSpawnRate);

Spawner.SpawnHeal();

StartCoroutine(SpawnHeal());

}

private void IncreaseSpawnRateHeal()

{

if (healSpawnRate > 5)

{

healSpawnRate -= 0.5f;

}

}

}

**Скрипт Buttons.cs**

using UnityEngine;

using UnityEngine.SceneManagement;

public class Buttons : MonoBehaviour

{

public void MainMenu()

{

Resume();

SceneManager.LoadScene("MainMenu");

}

public void Restart()

{

Resume();

SceneManager.LoadScene("MainGame");

}

public void QuitGame()

{

Application.Quit();

}

public void Pause()

{

Time.timeScale = 0f;

}

public void Resume()

{

Time.timeScale = 1f;

}

}

**Скрипт InGameUI.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using TMPro;

public class InGameUI : MonoBehaviour

{

private int Score;

[SerializeField]

private GameObject EndGameUI, MainUI, PauseUI, SettingsUI;

public Sprite fullHeart, emptyHeart;

[SerializeField]

private TextMeshProUGUI ScoreText, ScoreResultText, WaveText;

[SerializeField]

private SpriteRenderer firstLife, secondLife, thirdLife;

private bool isPauseAvaible;

private Buttons buttons;

private void Awake()

{

buttons = GetComponent<Buttons>();

isPauseAvaible = true;

}

public void ScoreUpdate(int score)

{

string scoreOut = "Очки " + score.ToString();

ScoreText.SetText(scoreOut);

Score = score;

}

public void HealthUpdate(int health)

{

if (health == 3)

{

firstLife.sprite = fullHeart;

secondLife.sprite = fullHeart;

thirdLife.sprite = fullHeart;

}

if (health == 2)

{

firstLife.sprite = emptyHeart;

secondLife.sprite = fullHeart;

thirdLife.sprite = fullHeart;

}

if (health == 1)

{

firstLife.sprite = emptyHeart;

secondLife.sprite = emptyHeart;

thirdLife.sprite = fullHeart;

}

if (health == 0)

{

firstLife.sprite = emptyHeart;

secondLife.sprite = emptyHeart;

thirdLife.sprite = emptyHeart;

string scoreResult = "Вы заработали " + Score + " очков";

ScoreResultText.SetText(scoreResult);

EndGameUI.SetActive(true);

MainUI.SetActive(false);

isPauseAvaible = false;

buttons.Pause();

}

}

public void UpdateWave(int wave)

{

string waveText = "Волна " + wave.ToString();

WaveText.SetText(waveText);

}

public void SetPauseOrUnpause()

{

if (isPauseAvaible)

{

if (Time.timeScale > 0f)

{

PauseUI.SetActive(true);

buttons.Pause();

}

else

{

PauseUI.SetActive(false);

SettingsUI.SetActive(false);

buttons.Resume();

}

}

}

}

**Скрипт Settings.cs**

using UnityEngine;

using UnityEngine.Audio;

using UnityEngine.UI;

public class Settings : MonoBehaviour

{

[SerializeField]

private Toggle \_fullscreenToggle;

[SerializeField]

private Dropdown \_qualityDropdown;

[SerializeField]

private Slider \_musicSlider, \_effectSlider;

[SerializeField]

private AudioMixer \_musicMixer, \_effectsMixer;

[SerializeField]

private GameDataScrObj \_settingsData;

private void Start()

{

Init();

}

public void Quality(int q)

{

QualitySettings.SetQualityLevel(q);

\_settingsData.SaveQualityLevel(q);

}

public void SetScreenMode(bool fsMode)

{

Screen.fullScreen = fsMode;

\_settingsData.SaveIsFullscreen(fsMode);

}

public void SetVolume(float vol)

{

\_musicMixer.SetFloat("Volume", vol);

\_settingsData.SaveMusicVolume(vol);

}

public void SetEffectVolume(float vol)

{

\_effectsMixer.SetFloat("Effects", vol);

\_settingsData.SaveEffectsVolume(vol);

}

private void Init()

{

\_settingsData.Init();

\_musicMixer.SetFloat("Volume", \_settingsData.LoadMusicVol());

\_musicSlider.value = \_settingsData.LoadMusicVol();

\_effectsMixer.SetFloat("Effects", \_settingsData.LoadEffectVol());

\_effectSlider.value = \_settingsData.LoadEffectVol();

\_qualityDropdown.value = \_settingsData.LoadQuality();

QualitySettings.SetQualityLevel(\_settingsData.LoadQuality());

\_fullscreenToggle.isOn = \_settingsData.LoadIsFullscreen();

Screen.fullScreen = \_settingsData.LoadIsFullscreen();

}

}

**Скрипт BackgroundMusic.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class BackGroundAudio : MonoBehaviour

{

private static BackGroundAudio \_backGroundAudioInstance;

[SerializeField]

private AudioClip[] \_backgroundMusic;

private AudioSource \_audioSource;

private void Awake()

{

\_audioSource = GetComponent<AudioSource>();

DontDestroyOnLoad(this.gameObject);

if (\_backGroundAudioInstance == null)

{

\_backGroundAudioInstance = this;

}

else

{

Destroy(gameObject);

}

}

void Start()

{

\_audioSource.clip = \_backgroundMusic[0];

\_audioSource.Play();

StartCoroutine(PlayMusic(\_backgroundMusic[0].length));

}

IEnumerator PlayMusic(float musicDuration)

{

yield return new WaitForSeconds(musicDuration);

AudioClip newRandomMusic = \_backgroundMusic[Random.Range(0, \_backgroundMusic.Length -1)];

\_audioSource.clip = newRandomMusic;

StartCoroutine(PlayMusic(newRandomMusic.length));

}

}

**Скрипт SoundEffects.cs**

using UnityEngine;

public class SoundEffects : MonoBehaviour

{

[SerializeField]

private AudioClip \_shootClip, \_deathClip, \_effectUpClip;

private AudioSource \_audioSource;

private void Awake()

{

\_audioSource = GetComponent<AudioSource>();

}

public void PlayShootClip()

{

\_audioSource.clip = \_shootClip;

\_audioSource.Play();

}

public void PlayDeathClip()

{

\_audioSource.clip = \_deathClip;

\_audioSource.Play();

}

public void PlayEffectUpClip()

{

\_audioSource.clip = \_effectUpClip;

\_audioSource.Play();

}

}