ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА ОБЩЕЙ МАТЕМАТИКИ

РАЗРАБОТКА МОБИЛЬНОГО ПРИЛОЖЕНИЯ СРЕДСТВАМИ ВЫСОКОУРОВНЕВОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ И СИСТЕМ ГРАФИЧЕСКОГО МОДЕЛИРОВАНИЯ

Выпускная квалификационная работа
обучающегося по направлению подготовки
01.03.02 Прикладная математика и информатика
очной формы обучения, группы 83001305
Зенкевича Дмитрия Александровича

Научный руководитель
д.т.н., профессор
Аверин Г.В.

БЕЛГОРОД 2017

Оглавление

ВВЕДЕНИЕ

Мобильные устройства постепенно захватывают мир. Наличие ноутбука, нетбука или смартфона у рядового человека на сегодняшний день является обычным делом. Поэтому нет ничего удивительного в том, что все больше и больше программного обеспечения требуется именно для таких устройств. В последнее время многие люди хотят видеть полноценный персональный компьютер у себя в телефоне. К полноценному функционалу можно отнести такие задачи как сетевые программы, 3D - игры, GIS - сервисы и т.д. Но все же на данный момент производительность карманных устройств сильно уступает персональным компьютерам, поэтому необходимо применять особые алгоритмы для разработки приложений, а также методы программирования.

Одним из самых распространенных карманных устройств являются устройства на базе операционных систем Android и IOS. Они служат для прослушивания музыки, телефонных звонков, отправки смс и электронной почты, а также многого другого. Многие люди предпочитают игры на мобильных устройствах, поэтому данный рынок ПО развивается очень быстро. Конкуренция порождает более сложные и красочные игры. Новым этапом в разработке игр для карманных устройств стали полноценные 3D - игры, в которых используются шейдеры материалов, тени и большое количество 3D объектов.

В данной дипломной работе было разработано мобильное приложение средствами высокоуровневого языка программирования и 3D графики.

В процессе разработки пришлось столкнуться сразу с несколькими проблемами ограниченной производительности iPhone и Android:

1. Скоростью работы графического ядра устройства;
2. Объемом оперативной памяти для одного запущенного приложения;
3. Скоростью чтения с диска устройства;

4. Скоростью работы процессора.

Для разработки 3D части был взят движок Unity 5. Данный движок позволяет программировать 3D сцены для IOS, Android, Linux, Windows и т.д. Языком программирования для этой среды был выбран C# как основной скриптовый язык для 3D сцены.

Первая часть дипломной работы посвящена исследованию возможностей мобильных операционных систем, игровых движков и объектно-ориентированному программированию. Вторая часть дипломной работы посвящена разработке приложения “SpaceAttack”, в которой рассматриваются аспекты работы с Unity и высокоуровневым языком программирования C#.

Цель работы – создание игры, удовлетворяющей требованиям начальной спецификации, исследование технологий создания 3D - игр, а также приемов программирования для смартфонов.

В соответствии с поставленной целью задачами выпускной квалификационной работы являются:

- изучить необходимые для работы программные продукты: 3dMax, Unity3D, Visual Studio и т.д.;
- разработать основное содержание и сценарий игры, создать базовые сцены и модели игровых объектов;
- разработать модули и скрипты, определяющие модели поведения объектов;
- скомпоновать и отладить мобильную игру “Space Attack” на игровом движке Unity3D.

Объект исследования – 3D игра, архитектура системы мобильных устройств.

Предмет исследования – процесс создания 3D игр и технологий их разработки.

Область применения – предполагается размещение разработанного программного продукта в интернет магазинах для мобильных операционных систем.

Результаты работы – В результате проделанной работы, была создана 3D - игра “Space Attack”, которая полностью удовлетворяет начальную спецификацию. Также было проведено исследование возможностей игрового движка Unity при работе с 3D графикой. Был приобретён опыт в разработке приложений для мобильных телефонов.

Структура и объём работы – выпускная квалификационная работа содержит текста на 56 страниц, 29 рисунков, список литературы на 21 наименование, а также приложение на 7 страниц.

1 АНАЛИТИЧЕСКИЙ ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Подавляющее большинство игр написано в жанре «аркада» и «головоломка». Изредка встречаются представители других жанров. Основным ограничением для «разнообразия жанров» является устройство ввода. Например, для удобной игры в 3D-шутер желательна возможность одновременного наведения на цель и стрельбы, то есть использование двух клавиш одновременно, что затруднено на маленьких клавиатурах мобильного телефона, а стратегии в реальном времени изначально ориентировались на управление джойстиком или компьютерной мышью.

Космический симулятор - жанр компьютерных игр, воспроизводящих с различной степенью достоверности управление космическим кораблём. Действие таких игр происходит, как правило, в космосе, путем имитации соответствующих сцен.

Жанр игровых космических симуляторов делится на несколько под жанров: симуляторы полёта в космосе, в которых акцент ставится на реалистичное управление космическим кораблём; боевые, главным элементом которых являются космические сражения; а также торговые, где в центре внимания находится, прежде всего, экономический элемент игрового мира (торговля и пиратство). Также космические симуляторы можно классифицировать по степени реализма и свободы действий игрока.

Наиболее распространённые образцы жанра ведут своё происхождение ещё с игровых автоматов. В этих играх симулируются, прежде всего, красочные космические битвы, получившие популярность благодаря таким фантастическим произведениям как «Звёздные войны», «Звёздный путь», «Вавилон-5» и многим другим.

1.1 Обзор 3D - игры «Space Attack»

Данная игра представляет собой симулятор аркадного космического сражения и предназначенная для широкого круга пользователей. Данная игра отличается простыми правилами и не требует от пользователя особой усидчивости, затрат времени на обучение или каких-либо особых навыков. Она дешева в разработке и при дистрибуции.

В игре «Space Attack» игрок управляет космическим кораблем, передвигая его по X и Z осям, а также отстреливая астероиды и вражеские корабли, надвигающихся сверху экрана. Целью игры является уничтожение волн врагов, которые двигаются горизонтально, а также вертикально, по направлению к низу экрана. Игрок имеет бесконечное количество патронов. Попадая во вражеский объект, игрок уничтожает его, за что получает очки. При уничтожении всех инопланетян появляется новая, ещё более сильная волна. Количество новых волн инопланетян неограниченно, что делает игру бесконечной.

Вражеские корабли пытаются уничтожить корабль, стреляя по нему. При попадании в корабль количество брони уменьшается на одну единицу. Если количество брони становится равным нулю, то игра заканчивается, а корабль взрывается.

Игра написана на движке Unity 5. Данный движок предоставляет быстрое и удобное программирование 3D сцен. Всё взаимодействие заключается в отработке скриптов, которые прикреплены к игровым объектам. У каждого объекта может быть несколько прикрепленных скриптов. За один кадр–фрейм программа выполняет метод Update() в каждом активном скрипте, а также перерисовывает сцену в зависимости от изменений.

В игре существуют 8 основных объектов:

- Player – главный космический корабль
- Main Camera – основная и единственная камера, через которую пользователь смотрит на сцену
- Lighting – отвечает за освещение сцены
- Background – основной фон игры
- Boundary – границы существования игровых объектов
- GameController – контроллер, отвечающий за добавление объектов на сцену
- DisplayText – текст, отображающий очки игрока
- StarField – звездное небо

В игре существуют следующие скрипты (основные):

- DestroyByBoundary
- DestroyByContact
- GameController
- Mover
- PlayerController
- RandomRotator

1.2 Обзор сред разработки

Unity — это инструмент для разработки двух и трёхмерных приложений, и игр, работающий под операционными системами Windows, Linux и OS X. Созданные с помощью Unity приложения работают под операционными системами Windows, OS X, Windows Phone, Android, Apple iOS, Linux, а также на игровых приставках Wii, PlayStation 3, PlayStation 4,

Xbox 360, Xbox One и MotionParallax3D дисплеях (устройства для воспроизведения виртуальных голограмм), например, Nettlebox. Есть возможность создавать приложения для запуска в браузерах с помощью специального подключаемого модуля Unity (Unity Web Player), а также с помощью реализации технологии WebGL. Ранее была экспериментальная поддержка реализации проектов в рамках модуля Adobe Flash Player, но позже команда разработчиков Unity приняла сложное решение по отказу от этого. На рисунке 1.1 представлено окно редактора Unity.

Рис 1.1 - Окно редактора Unity

Данный движок предоставляет быстрый и гибкий способ портирования своего приложения практически на любую популярную операционную систему. Важной особенностью данного движка, из-за которого он так популярен, является возможность запуска своего приложения на любом Интернет браузере. На него необходимо просто установить плагин – Unity Web Player, а также возможность разработки продуктов для iPhone, iPod touch, iPad, Android.

1.2.1 Технические характеристики

Перечень технических характеристик:

1. игровой движок полностью интегрирован в среду разработки, что позволяет прямо в редакторе тестировать игру;
2. работа с ресурсами возможна через простой drag&drop. Интерфейс редактора настраиваем;
3. поддержка DirectX и OpenGL;
4. реализована система наследования объектов;
5. поддержка импорта из очень большого количества форматов;
6. встроенный генератор ландшафтов;
7. встроенная поддержка сети;
8. есть решение для коллективной разработки - Asset Server;
9. возможность использовать свой способ контроля версий. Tortoise SVN или Source Gear к примеру.

1.2.2 Достоинства и недостатки

Перечень достоинств и недостатков:

- широкие возможности импорта;
- кроссплатформенность;
- гибкая ценовая категория и невысокие цены;
- в версии 2.6 используется устаревшая версия PhysX, без симуляции жидкостей, мягких тел и тканей, начиная с 3 версии, этот недостаток решен;
- бесшовная организация рабочего процесса - Билд для iPhone - это полный XCode проект, который сразу же можно использовать;

- высокопроизводительные скрипты - во время создания билда JavaScript и C# компилируются в нативный ассемблерный код ARM. Это дает средний прирост производительности в 20-40 раз по сравнению с интерпретируемыми языками;
- высокопроизводительная отрисовка сетки (Mesh Rendering) - Unity использует упакованные форматы для мешей, которые позволяют выжать максимум из возможной производительности iPhone;
- удаление перекрываемых поверхностей (Occlusion culling);
- эмуляция шейдеров - имитирование графических возможностей iPhone прямо внутри Редактора Unity для быстрого создания WYSIWYG-прототипов. Unity полностью поддерживает возможности iPhone в мультитекстурировании;
- алгоритм сжатия текстур PRVTC.

1.3 Обзор среды программирования

Microsoft Visual Studio — линейка продуктов компании Майкрософт, включающих интегрированную среду разработки программного обеспечения и ряд других инструментальных средств (рис. 1.2).

Рис 1.2. Логотип Visual Studio

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 как 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). В данной работе Visual Studio используется как лучшая альтернатива MonoDevelop, которую использует Unity3D “из коробки”.

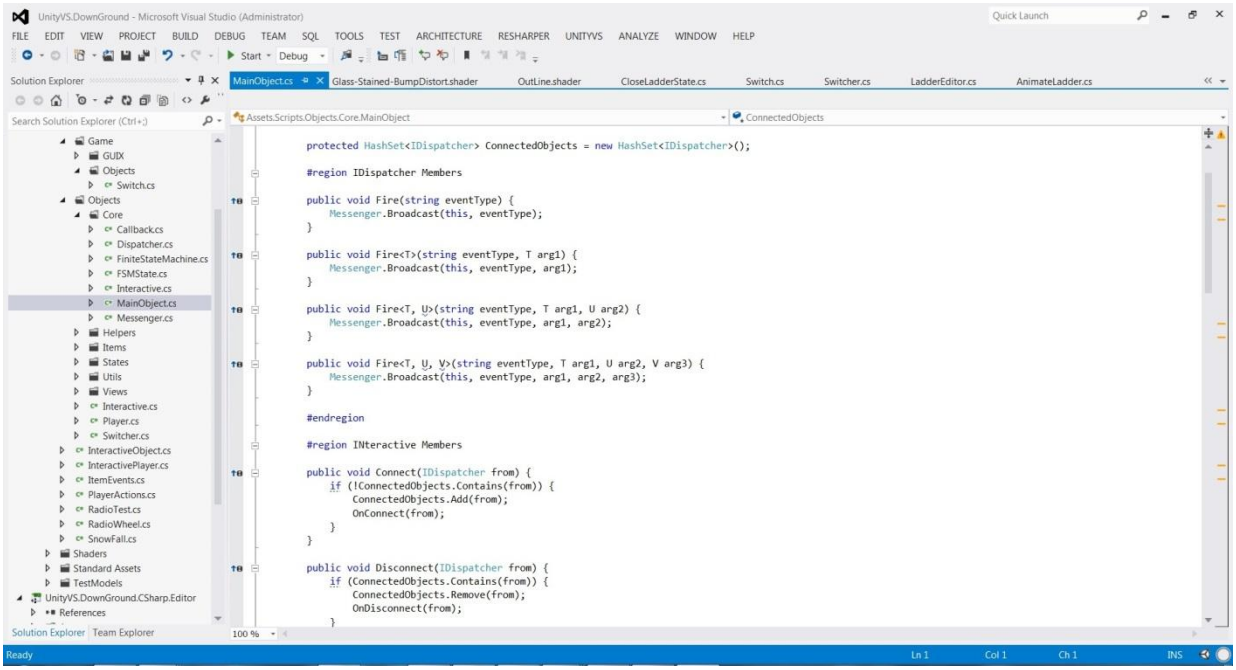

Рисунок 1.3. Графический интерфейс Visual Studio

1.4 Язык программирования C#

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

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, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML.

Переняв многое от своих аналогов — языков C++, Java, Delphi, Модула и Smalltalk — 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 множественное наследование интерфейсов).

Данный язык был выбран в качестве основного, так как обладает нужными качествами для реализации, имеет встроенную поддержку обобщений, делегатов и событий, что облегчит реализацию.

2 ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД И ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Следует сосредоточить внимание на значимые части определения:

1) объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы;

2) каждый объект является экземпляром определенного класса;

3) классы образуют иерархии. Программа считается объектно-ориентированной, только в случае исполнения всех условий. В частности, программирование, не использующее наследование, называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных.

2.1 Основные понятия объектно-ориентированного программирования

Абстракция данных — значит акцентирование важной информации и исключение с рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных, подразумевая комплект важных данных объекта, открытый остальной программе.

Инкапсуляция — свойство системы, позволяющее соединить данные и методы, действующие с ними, в классе. Одни языки отождествляют инкапсуляцию с сокрытием, однако прочие отличают данные определения.

Наследование — свойство системы, дозволяющее изложить новый класс на базе ранее имеющегося с отчасти либо целиком заимствующейся функциональностью. Класс, с которого выполняется наследование, именуется базисным, родительским либо суперклассом. Новый класс — потомком, наследником, дочерним либо производным классом.

Полиморфизм подтипов (в ООП называемый просто «полиморфизмом») — качество концепции, позволяющее применять объекты с одним и тем же интерфейсом в отсутствии данных о типе и внутренней структуре объекта. Иной тип полиморфизма — параметрический — в ООП именуют обобщённым программированием.

Класс — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).

В частности, в классах обширно применяются особые конструкции с 1-го либо нередкого 2-ух спаренных методов, отвечающих за простые процедуры с конкретным полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю. Эти блоки называются «свойствами» и почти совпадают по конкретному имени со своим полем (например, имя поля может начинаться со строчной, а имя свойства — с заглавной буквы). Иным проявлением интерфейсной натуры класса считается в таком случае, то что при копировании надлежащей переменной посредством присваивания, копируется только лишь интерфейс, однако никак не сами сведения, в таком случае имеется класс — ссылочный тип данных. Переменная-объект, принадлежащая к установленному классом типу, именуется экземпляром данного класса. При этом в некоторых исполняющих системах класс также может

представляться некоторым объектом при выполнении программы посредством динамической идентификации типа данных. Как правило классы хотят сделать таким способом, для того чтобы гарантировать соответствующие натуре предмета и решаемой проблеме целостность данных объекта, а кроме того практичный и обычный интерфейс. В свою очередь, единство предметной области объектов и их интерфейсов, а кроме того практичность их проектирования, гарантируется наследованием.

Объект — сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

2.3 Создание и использование скриптов

Поведение игровых объектов контролируется с помощью компонентов (Components), которые присоединяются к ним. Несмотря на то, что встроенные компоненты Unity могут быть очень разносторонними, вскоре вы обнаружите, что вам нужно выйти за пределы их возможностей, чтобы реализовать ваши собственные особенности геймплея. Unity позволяет вам создавать свои компоненты, используя скрипты. Они позволяют активировать игровые события, изменять параметры компонентов, и отвечать на ввод пользователя каким вам угодно способом.

Unity изначально поддерживает три языка программирования:

C# (произносится как Си-шарп), стандартный в отрасли язык подобный Java или C++.

UnityScript, язык, разработанный специально для использования в Unity по образцу JavaScript.

В отличии от других ассетов, скрипты обычно создаются непосредственно в Unity. Вы можете создать скрипт используя меню Create

в левом верхнем углу панели Project или выбрав Assets > Create > C# Script (или JavaScript/Boo скрипт) в главном меню.

Новый скрипт будет создан в папке, которую вы выбрали в панели Project. Имя нового скрипта будет выделено, предлагая вам ввести новое имя.

Лучше ввести новое имя скрипта сразу после создания чем изменять его потом. Имя, которое вы введете будет использовано, чтобы создать начальный текст в скрипте, как описано ниже.

2.3.1 Структура файла скрипта

После двойного щелчка на скрипте в Unity, он будет открыт в текстовом редакторе. По умолчанию Unity будет использовать MonoDevelop, но вы можете выбрать любой редактор из панели External Tools в настройках Unity.

Содержимое файла будет выглядеть примерно так:

```
using UnityEngine;
using System.Collections;
public class MainPlayer : MonoBehaviour {
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
    }
}
```

Скрипт взаимодействует с внутренними механизмами Unity за счет создания класса, наследованного от встроенного класса, называемого MonoBehaviour. Вы можете думать о классе как о своего рода плане для создания нового типа компонента, который может быть

прикреплен к игровому объекту. Каждый раз, когда вы присоединяете скриптовый компонент к игровому объекту, создается новый экземпляр объекта, определенный планом. Имя класса берется из имени, которое вы указали при создании файла. Имя класса и имя файла должны быть одинаковыми, для того, чтобы скриптовый компонент мог быть присоединен к игровому объекту.

Основные вещи, достойные внимания, это две функции, определенные внутри класса. Функция Update - это место для размещения кода, который будет обрабатывать обновление кадра для игрового объекта. Это может быть движение, срабатывание действий и ответная реакция на ввод пользователя, в основном всё, что должно быть обработано с течением времени во игровом процессе. Чтобы позволить функции Update выполнять свою работу, часто бывает полезно инициализировать переменные, считать свойства и осуществить связь с другими игровыми объектами до того, как будут совершены какие-либо действия. Функция Start будет вызвана Unity до начала игрового процесса (т.е. до первого вызова функции Update), и это идеальное место для выполнения инициализации.

Инициализация объекта выполняется не в функции-конструкторе. Это потому, что создание объектов обрабатывается редактором и происходит не в начале игрового процесса. Если произойдет попытка определить конструктор для скриптового компонента, он будет мешать нормальной работе Unity и может вызвать серьезные проблемы с проектом.

A UnityScript script works a bit differently to C# script:

```
#pragma strict
function Start () {
}
function Update () {
}
```

Здесь функции Start и Update имеют такое же значение, но класс не объявлен явно. Предполагается, что скрипт сам по себе определяет класс. Он будет неявно производным от MonoBehaviour и получит своё имя от имени файла скрипта.

2.3.2 Управление игровым объектом

Как было сказано ранее, скрипт определяет только план компонента и, таким образом, никакой его код не будет активирован до тех пор, пока экземпляр скрипта не будет присоединен к игровому объекту. Вы можете прикрепить скрипт перетаскиванием ассета скрипта на игровой объект в панели Hierarchy или через окно Inspector выбранного игрового объекта. Имеется также подменю Scripts в меню Component, которое содержит все скрипты, доступные в проекте, включая те, которые вы создали сами. Экземпляр скрипта выглядит так же, как и другие компоненты в окне Inspector.

2.3.3 Функции событий

Скрипт в Unity не похож на традиционную идею программы, где код работает постоянно в цикле, пока не завершит свою задачу. Вместо этого, Unity периодически передаёт управление скрипту при вызове определённых объявленных в нём функций. Как только функция завершает исполнение, управление возвращается обратно к Unity. Эти функции известны как функции событий, т.к. их активирует Unity в ответ на события, которые происходят в процессе игры. Unity использует схему именования, чтобы определить, какую функцию вызвать для определённого события. Далее указаны одни из самых важных и часто встречающихся событий.

Игра - это что-то вроде анимации, в которой кадры генерируются на ходу. Ключевой концепт в программировании игр заключается в

изменении позиции, состояния и поведения объектов в игре прямо перед отрисовкой кадра. Такой код в Unity обычно размещают в функции Update. Update вызывается перед отрисовкой кадра и перед расчётом анимаций.

```
void Update() {
   float distance = speed * Time.deltaTime * Input.GetAxis("Horizontal");
   transform.Translate(Vector3.right * distance);
}
```

Физический движок также обновляется фиксированными по времени шагами, аналогично тому как работает прорисовка кадра. Отдельная функция события FixedUpdate вызывается прямо перед каждым обновлением физических данных. Т.к. обновление физики и кадра происходит не с одинаковой частотой, то вы получите более точные результаты от кода физики, если поместите его в функцию FixedUpdate, а не в Update.

```
void FixedUpdate() {
   Vector3 force = transform.forward * driveForce * Input.GetAxis("Vertical");
   rigidbody.AddForce(force);
}
```

Также иногда полезно иметь возможность внести дополнительные изменения в момент, когда у всех объектов в сцене отработали функции Update и FixedUpdate и рассчитались все анимации. В качестве примера, камера должна оставаться привязанной к целевому объекту; подстройка ориентации камеры должна производиться после того, как целевой объект сместился. Другим примером является ситуация, когда код скрипта должен переопределить эффект анимации (допустим, заставить голову персонажа

повернуться к целевому объекту в сцене). В ситуациях такого рода можно использовать функцию LateUpdate.

```
void LateUpdate() {
   Camera.main.transform.LookAt(target.transform);
}
```

Зачастую полезно иметь возможность вызвать код инициализации перед любыми обновлениями, происходящими во время игры. Функция Start вызывается до обновления первого кадра или физики объекта. Функция Awake вызывается для каждого объекта в сцене в момент загрузки сцены. Стоит учитывать, что хоть для разных объектов функции Start и Awake и вызываются в разном порядке, все Awake будут выполнены до вызова первого Start. Это значит, что код в функции Start может использовать всё, что было сделано в фазе Awake.

В Unity есть система для отрисовки элементов управления GUI поверх всего происходящего в сцене и реагирования на клики по этим элементам. Этот код обрабатывается несколько иначе, нежели обычное обновление кадра, так что он должен быть помещён в функцию OnGUI, которая будет периодически вызываться.

```
void OnGUI() {
   GUI.Label(labelRect, "Game Over");
}
```

Также можно определять события мыши, которые срабатывают у GameObject’a, находящегося в сцене. Это можно использовать для наведения орудий или для отображения информации о персонаже под указателем курсора мыши. Существует набор функций событий OnMouseXXX (например, OnMouseOver, OnMouseDown), который позволяет скрипту реагировать на действия с мышью пользователя.

Например, если кнопка мыши нажата в то время, когда курсор мыши находится над определённым объектом, то, если в скрипте этого объекта присутствует функция OnMouseDown, она будет вызвана.

Физический движок сообщит о столкновениях с объектом с помощью вызова функций событий в скрипте этого объекта. Функции OnCollisionEnter, OnCollisionStay и OnCollisionExit будут вызваны по началу, продолжению и завершению контакта. Соответствующие функции OnTriggerEnter, OnTriggerStay и OnTriggerExit будут вызваны, когда коллайдер объекта настроен как Trigger (т.е. этот коллайдер просто определяет, что его что-то пересекает и не реагирует физически). Эти функции могут быть вызваны несколько раз подряд, если обнаружен более чем один контакт во время обновления физики, поэтому в функцию передаётся параметр, предоставляющий дополнительную информацию о столкновении (координаты, “личность” входящего объекта и т.д.).

```
void OnCollisionEnter(otherObj: Collision) {
   if (otherObj.tag == "Arrow") {
      ApplyDamage(10);
   }
}
```

3 РАЗРАБОТКА ПРИЛОЖЕНИЯ «SPACE ATTACK»

Приложение реализовано как связь различных предметов с 3D сценой и иными объектами, а также отработкой скриптов, закрепленных к данным объектам. Поэтому приложение построено как взаимодействие игровых объектов, которые инкапсулируют каждый свою логику.

3.1 Термины и основные принципы

В среде разработке есть целый ряд понятий и связанных с ними технологий, которые условно можно отнести к технологиям по работе с текстурами. Все они прямо или косвенно направлены на отображении текстур. К ним можно отнести:

Меш – это координатная сетка, состоящая из треугольников. Она задается массивом из треугольников, треугольник в свою очередь задается смещением каждой вершины по материалу.

В результате при необходимости возможно приобрести нелинейное представление текстуры. Меш бывает, как многоугольный, так и складывающийся с 2-ух треугольников. В случае данного приложения был применен стандартный меш.

Материал – это структура, которая содержит разнообразные параметры по отображению текстуры. Такие как способ отображения, уровень фильтрации, а также смещение и масштаб на текстуре. Материал и

меш в Unity3D существуют раздельно, так как меш может отвечать за физику объекта. Материал может быть, как общим, т.е. принадлежать сразу нескольким объектам, так и частным.

Текстура – это массив цветов каждого пикселя. В Unity3D есть класс, который представляет текстуру – это Texture2D. Он содержит набор полей и методов для работы с текстурой непосредственно, такие как «изменить пиксель» или «применить компрессию».

Шейдер - это подпрограмма прорисовки текстуры. Обычно она пишется на собственном языке. Её концепция написания такова, что мы задаем последовательность прорисовки и на каждом этапе применяем какие-нибудь фильтры или алгоритмы. Шейдер может быть рассчитан на несколько текстур. Все современные 3D приложения используют шейдеры для создания реалистичных поверхностей и много другого. В случае с iPhone шейдеры пишутся на специальном эмулирующем языке. В Unity3D Шейдер как подпрограмма не может существовать без материала.

В Unity3D работа с текстурами специально упрощена и сделана удобной. Мы можем складывать необходимые изображения в папке Resourses, а потом, когда программа запущена, по мере необходимости подгружать их оттуда. Это делается одним методом с указанием относительного пути до файла. При загрузке изображения мы создаем экземпляр объекта Texture2D и присваиваем ему результат загрузки. И можем свободно присоединять текстуры к материалу, для отображения на сцене.

3.2 Настройка проекта

Для реализации поставленной задачи был создан новый проект в Unity3D. Импортированы дополнительные пакеты для разработки приложения и сохранена сцена.

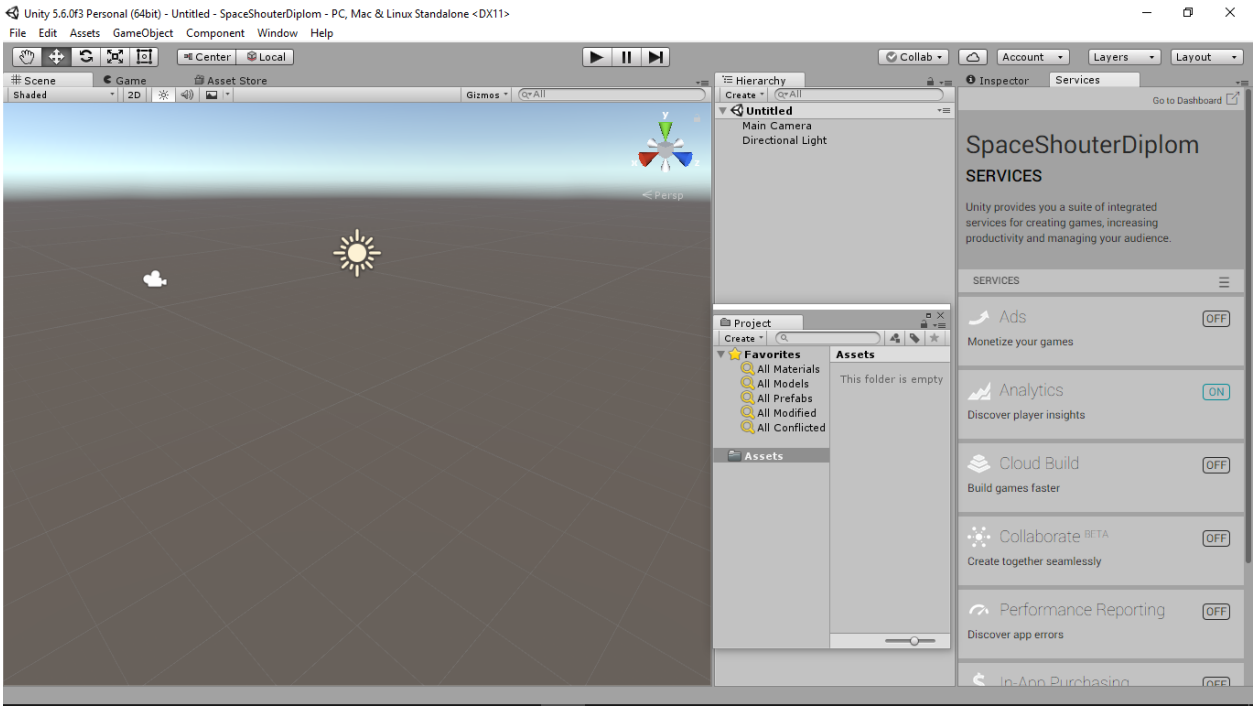

Рис 3.1. Окно проекта

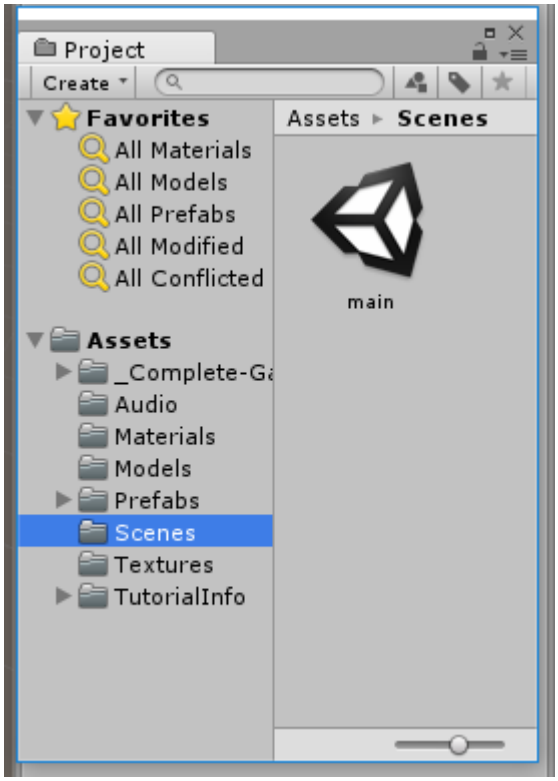

Рис 3.2. Окно Project

По умолчанию работа в конструкторе происходит со сборкой PC, Mac & Linux, все пакеты и скрипты будут оптимизированы под данные платформы. Unity может создавать игры под множество платформ, но одновременно можно создавать только для одной выбранной платформы.

Для того чтобы настроить сборку, открыл меню Files->Build Settings.

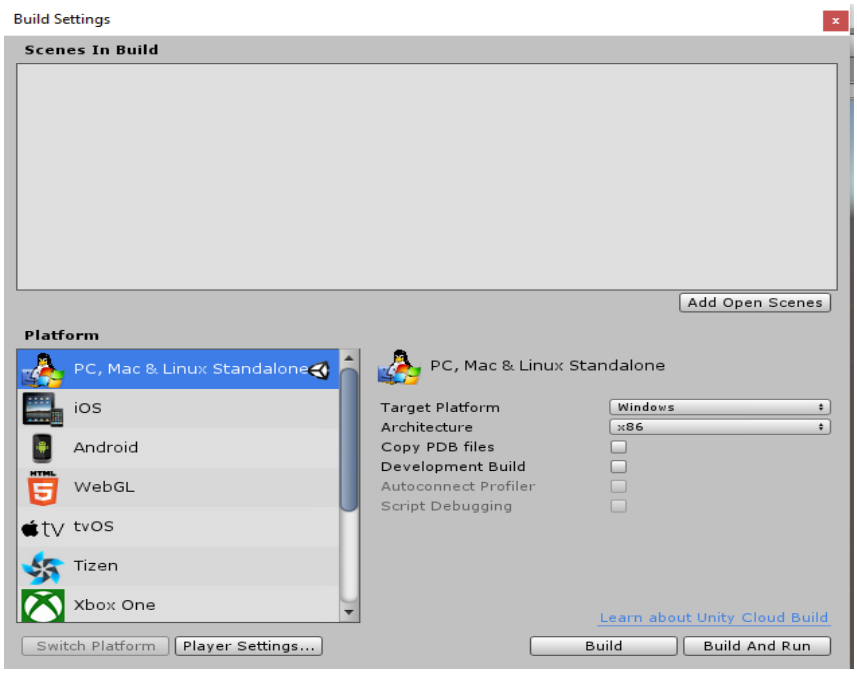

Рис 2.3. Окно настроек сборки

После чего были настроены детали нашей сборки и выбрана подходящая платформа – ios.

После выбора сборки были настроены детали сцены в разделе Player Settings в текущем окне. Необходимо добиться портретной ориентации. Указана ширина – 600, а длина – 900.

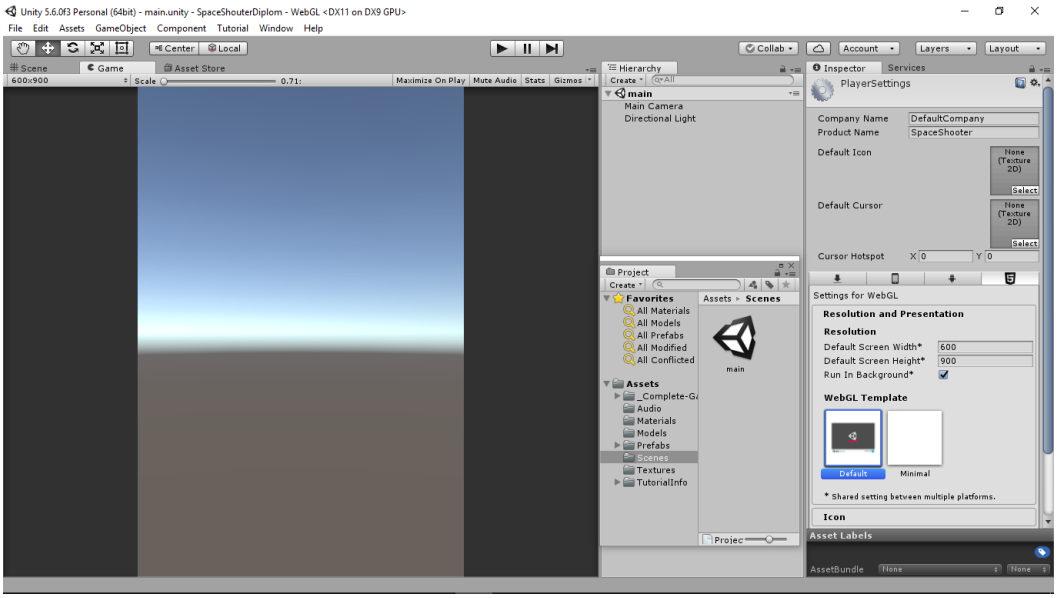

Рис 3.4. Готовая сцена

3.3 Добавление и настройка игрока

После настройки проекта была добавлена модель игрока. В приложении роль игрока выполняет космический корабль. Модель корабля разработана с помощью специальных систем для создания и редактирования трёхмерной графики и анимации. Для данного проекта использовалась программа Autodesk 3ds Max, т.к. она содержит самые современные средства для художников и специалистов в области мультимедиа. Работает в операционных системах Windows и Windows NT. 3Ds Max располагает обширными средствами для создания разнообразных по форме и сложности трёхмерных компьютерных моделей, реальных или фантастических объектов окружающего мира, с использованием разнообразных техник и механизмов.

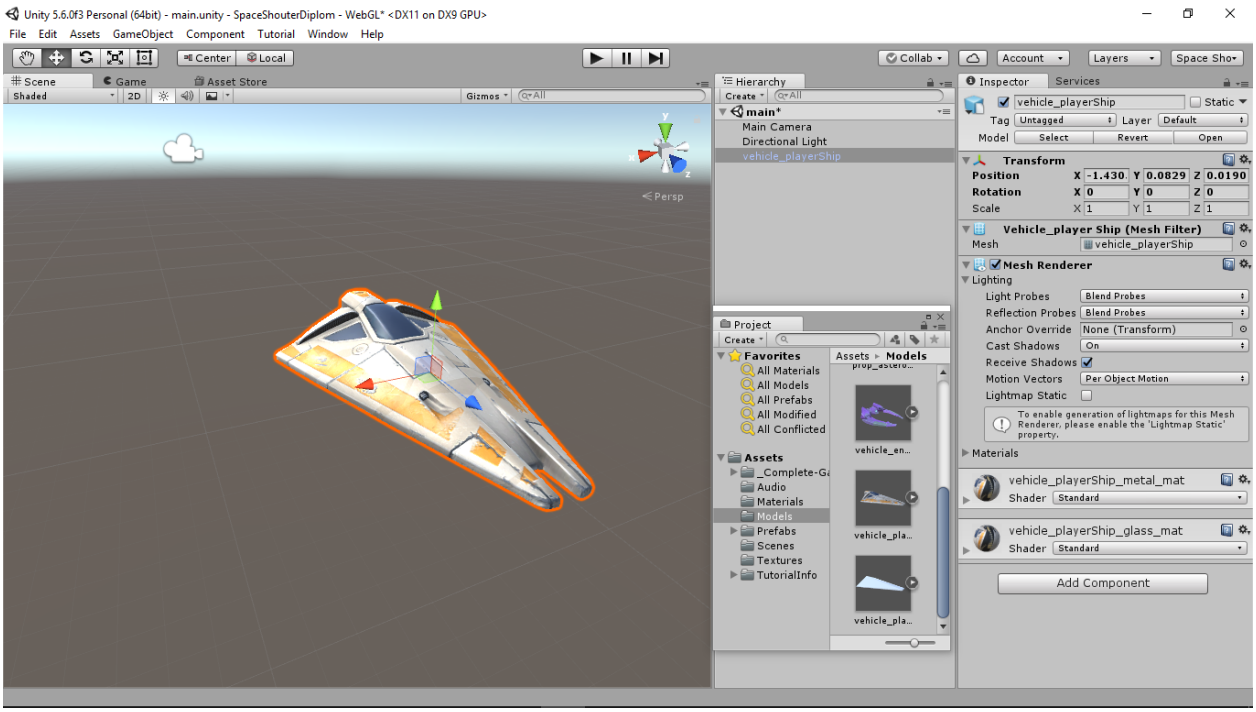

Рис 3.5. Модель корабля спереди

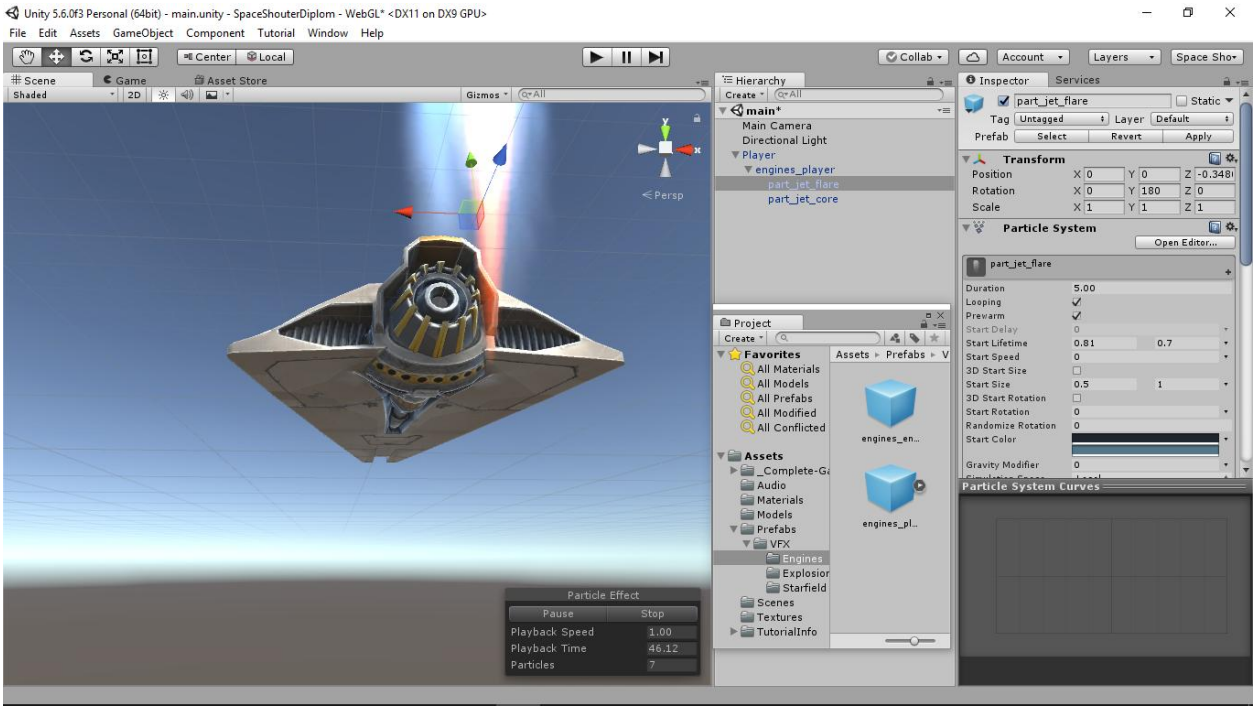

Рис 3.6. Модель двигателя корабля

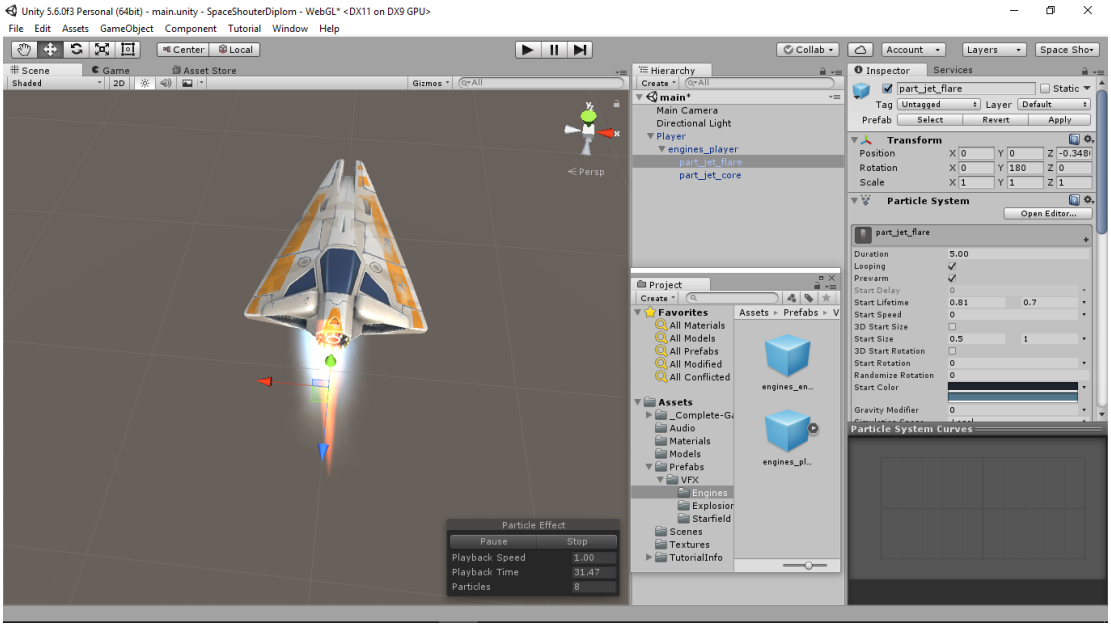

Рис 3.7. Модель корабля сверху

Добавлена модель корабля на сцену проекта и расположена в центре мировых координат. Сцена имеет три плоскости с координатами в виде осей x, y, z начало которых ноль по всем трем осям. Иконка шестеренки в разделе Transform и вкладка Reset позволят сбросить значения координат объекта в нулевые значения.

Для настроек гравитации, столкновения и перемещения для корабля установлены физические свойства с помощью компонента Rigidbody. Так

как корабль находится в космосе, флажок Use Gravity в свойствах объекта снят.

Чтобы правильно рассчитать столкновение объектов необходимо знать занимаемый в пространстве объем. Для этого к объекту добавлен компонент Mesh Collider и создана сетка обтекания корабля.

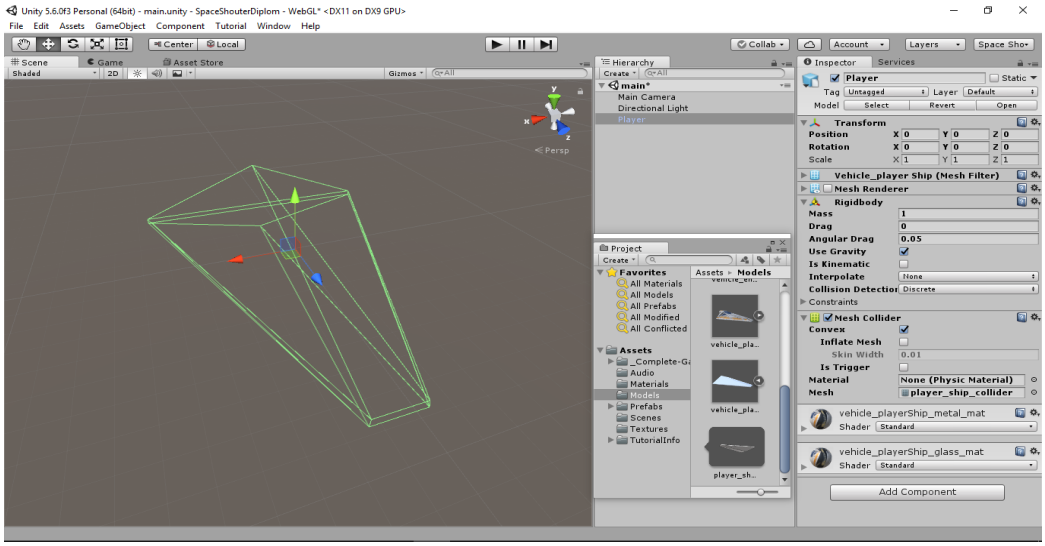

Рис 3.8. Сетка коллайдера корабля

3.4 Настройка камеры и освещения

Расположение камеры в игре находится сверху вниз, а обзор игрового поля происходит из фиксированного положения. Камера и свет создаются по умолчанию со сценой поэтому камера была перемещена в центр игровой области, повернута по оси x на 90 градусов, поднята по оси y на 10 пунктов и выбран ортографический режим проецирования вместо перспективного. Так же изменен цвет фона на черный с помощью свойств Background и Clearaalags.

После настройки фона отрегулирована камера для положения корабля снизу и масштаба (свойства Position Z и Size).

Далее было настроено световое оснащение сцены Direction Light, сравнимое с освещением солнца. В игре потребовалось три таких источника света, который освещают корабль со всех сторон создавая

хорошую игровую атмосферу. Источник света Direction Light зависит от направления лучей, а не от позиции на осях. Увеличена интенсивность света и направление лучей.

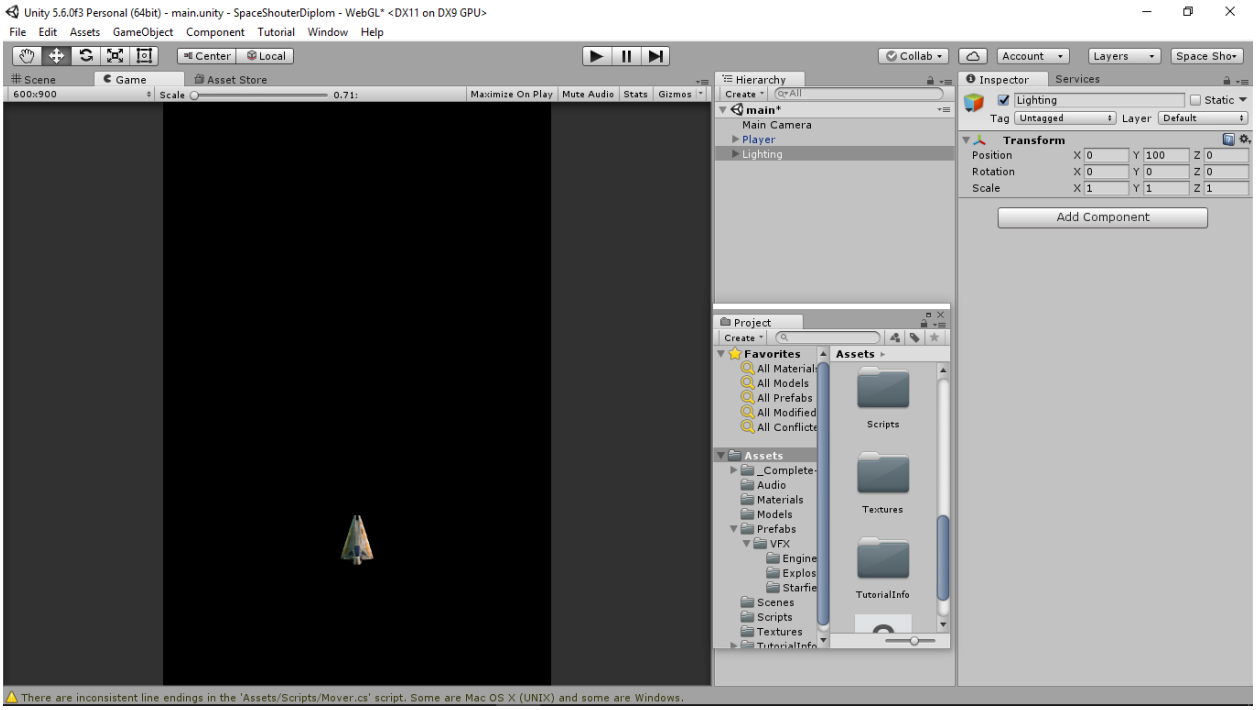

Рис 3.9. Итоговая сцена игры

3.5 Настройка фона в игре

Фоновое изображение содержит добавленный на сцену объект Quad по оси X с текстурой, продемонстрированной на рисунке ниже. Размер текстуры, как и размер игровой области 600x900 пикселей. Далее был изменен шейдер материала на Unlit-> Texture для естественного и без бликового изображения.

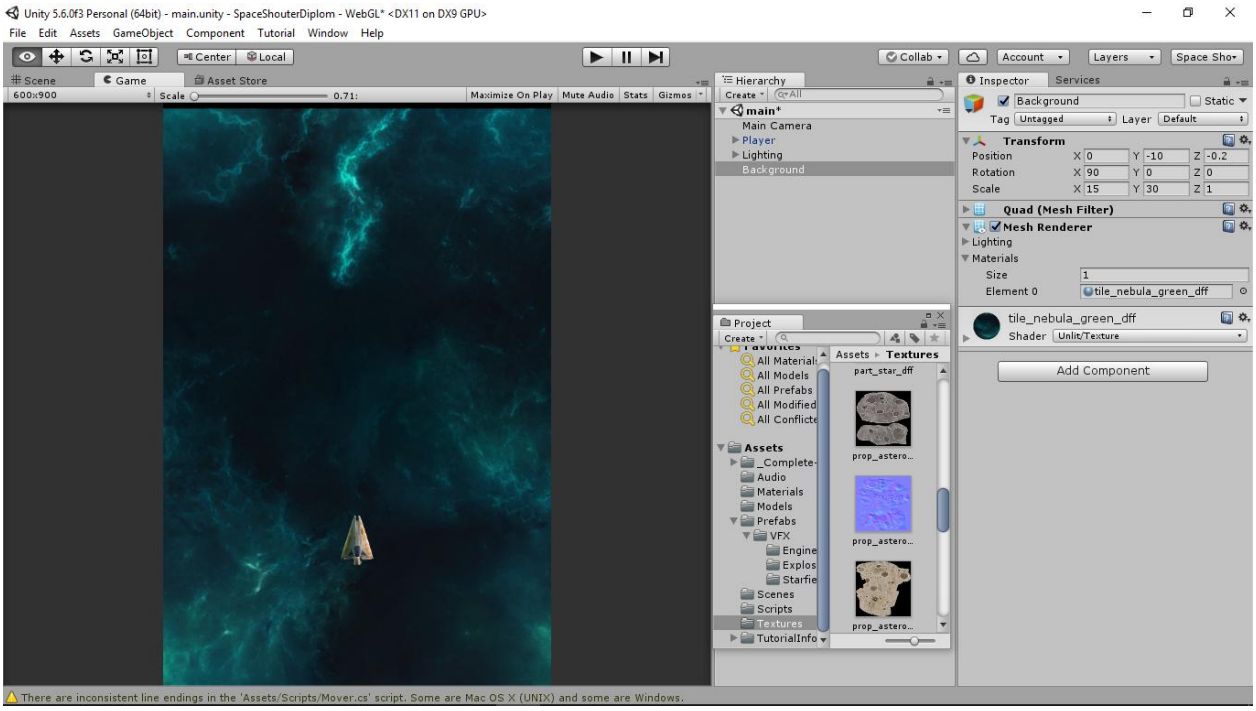

Рис 3.10. Космический фон игры

3.5.1 Прокрутка фона

В настоящий момент фон игры статический. Для придания реализма игровой сцене, фон был зациклен функцией Math.Repeat(). Она изменяет значение координат фона от 0 до 30 смещая его по оси Z и возвращая обратно. Публичная переменная scrollSpeed отвечает за скорость фона, tileSize за высоту фона и приватная переменная currentObject содержащая ссылку на объект. Код скрипта представлен ниже:

```
public class ScrollBackground : MonoBehaviour {
    public float scrollSpeed;
    public float tileSize;
    private Transform currentObject;

    void Start()
    {
        currentObject = GetComponent<Transform>();

    }
    void Update()
    {
        currentObject.position = new Vector3(currentObject.position.x,
currentObject.position.y,
            Mathf.Repeat(Time.time*scrollSpeed, tileSize));
    }
   }
```

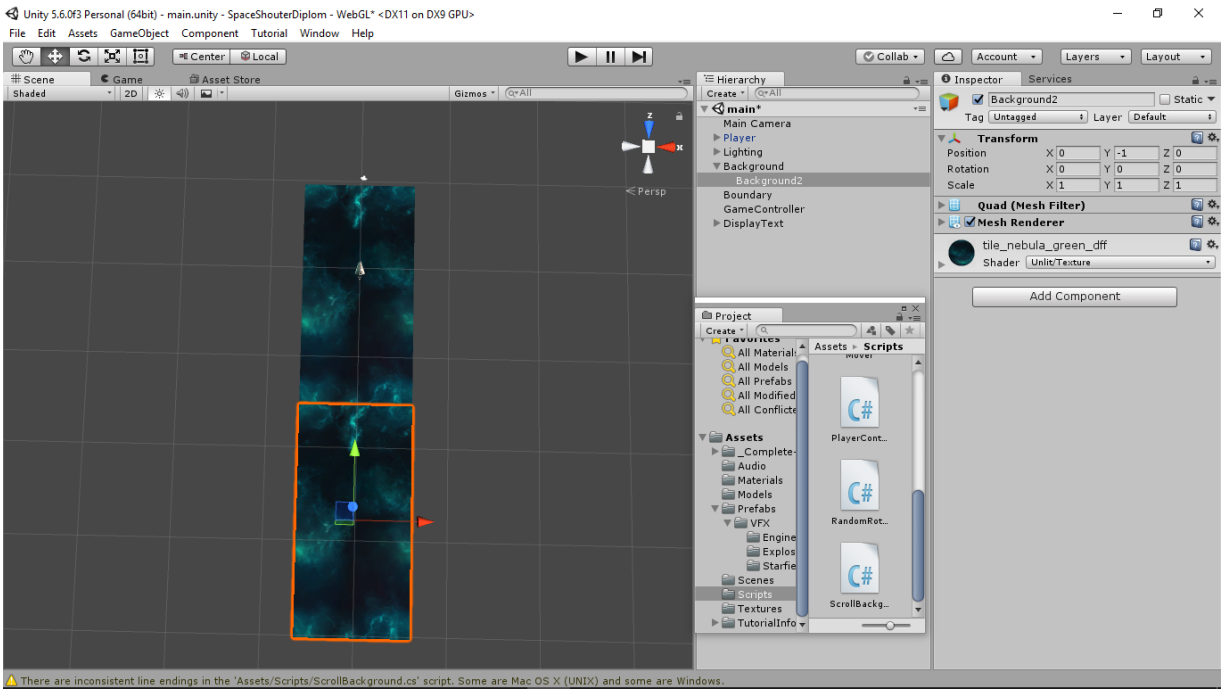

Рис 3.11. Итоговая сцена игры

Для того чтобы фон проходил полный цикл без прерываний создан его дубликат и помещён сзади.

2.5.2 Создание звездного неба

Для создания эффекта звездного неба создан пустой объект Starfield. В нем создан объект Particle System. Этот объект используется в Unity для создания спецэффектов. Он имитирует жидкие субстанции наподобие разных жидкостей, облаков и чего-нибудь, связанного с огнём путём генерации и анимации в сцене большого количества небольших 2D изображений. После редактирования свойств объекта была получена анимация звездного неба.

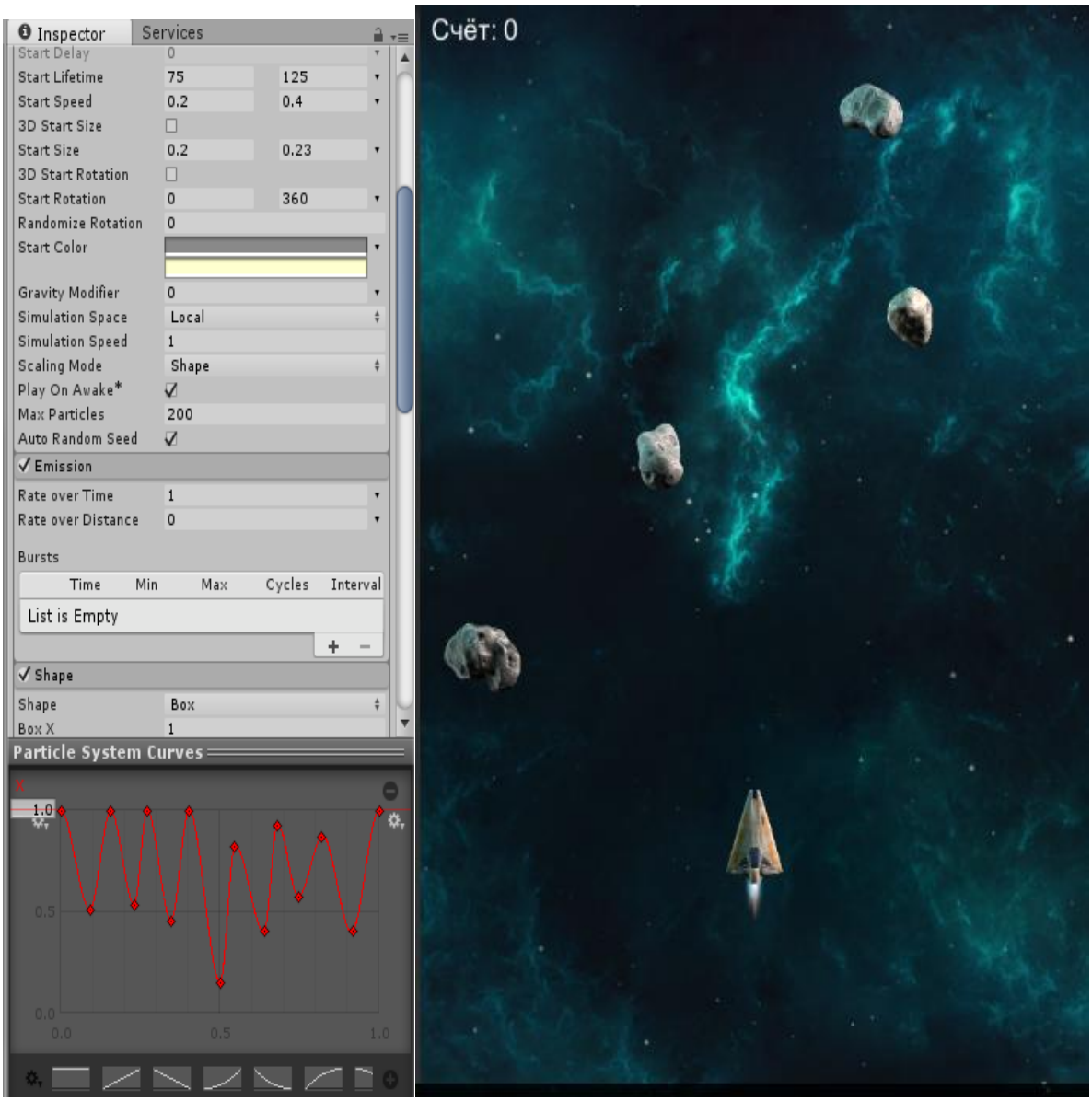

Рис 3.12. Настройка эффекта звёздного неба

3.6 Создание движения объекта

Для перемещения объектов используются скрипты на языке программирования C#. Когда отслеживается игровая логика и взаимодействия, анимации, позиции камеры и т.д. есть несколько разных событий, которые можно использовать. По общему шаблону, большая часть задач выполняется внутри функции Update, но есть также ещё другие функции, которые можно использовать.

FixedUpdate: Зачастую случается, что FixedUpdate вызывается чаще чем Update. FU может быть вызван несколько раз за кадр, если FPS низок и

функция может быть и вовсе не вызвана между кадрами, если FPS высок. Все физические вычисления и обновления происходят сразу после FixedUpdate. При применении расчётов передвижения внутри FixedUpdate, не нужно умножать значения на Time.deltaTime. Потому что FixedUpdate вызывается в соответствии с надёжным таймером, независящим от частоты кадров.

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

LateUpdate: LateUpdate вызывается раз в кадр, после завершения Update. Любые вычисления произведённые в Update будут уже выполнены на момент начала LateUpdate. Часто LateUpdate используют для преследующей камеры от третьего лица.

Движение корабля осуществляеся с помощью функции FixedUpdate. Unity вызывает функцию FixedUpdate каждый раз перед тем как выполнить расчеты физики.

Код перемещения корабля:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[System.Serializable]
public class Boundary
{
    public float xMin, xMax, zMin, zMax;
}

public class PlayerController : MonoBehaviour {

    public float speed = 10;
    public Boundary boundary;
    public float tilt;//для угла наклона

    private void FixedUpdate()
    {

        //метод GetAxis определяет к какой из осей обратился пользователь
        //другими словами какие кнопки нажаты(вправо, влево, вверх, вниз)
        //метод возвращает значение от -1 до 1
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");

        GetComponent<Rigidbody>().rotation = Quaternion.Euler(
            0f,
            0f,
            GetComponent<Rigidbody>().velocity.x * -tilt
            );

        //Rigidbody>().velocity принимает значение от структуры Vector3 и двигает
корабль
```

```
        GetComponent<Rigidbody>().velocity = new Vector3(moveHorizontal, 0f,
moveVertical)*speed;
        GetComponent<Rigidbody>().position = new Vector3(
            Mathf.Clamp(GetComponent<Rigidbody>().position.x, boundary.xMin,
boundary.xMax),
            0f,
            Mathf.Clamp(GetComponent<Rigidbody>().position.z, boundary.zMin,
boundary.zMax)
            );
    }
}
```

Корабль движется, но очень медленно, т.к. функция GetAxis() возвращает значение между -1 и 1, значит движение корабля никогда не будет больше 1 единицы в секунду. Чтобы увеличить скорость по заданной оси, значение полученное от GetAxis() нужно умножить на число не равное 0 или умножить всю структуру Vector3. Для этого использовалась публичная переменная speed, которую можно изменять в инспекторе.

Для того, чтобы корабль не выходил за пределы поля нужно ограничить пределы игрового пространства с помощью функции Mathf.Clamp(). Данная функция ограничивает любое число в рамках между минимальным и максимальным значением. Создан отдельный класс Boundary для хранения максимальных и минимальных значений по осям X и Z.

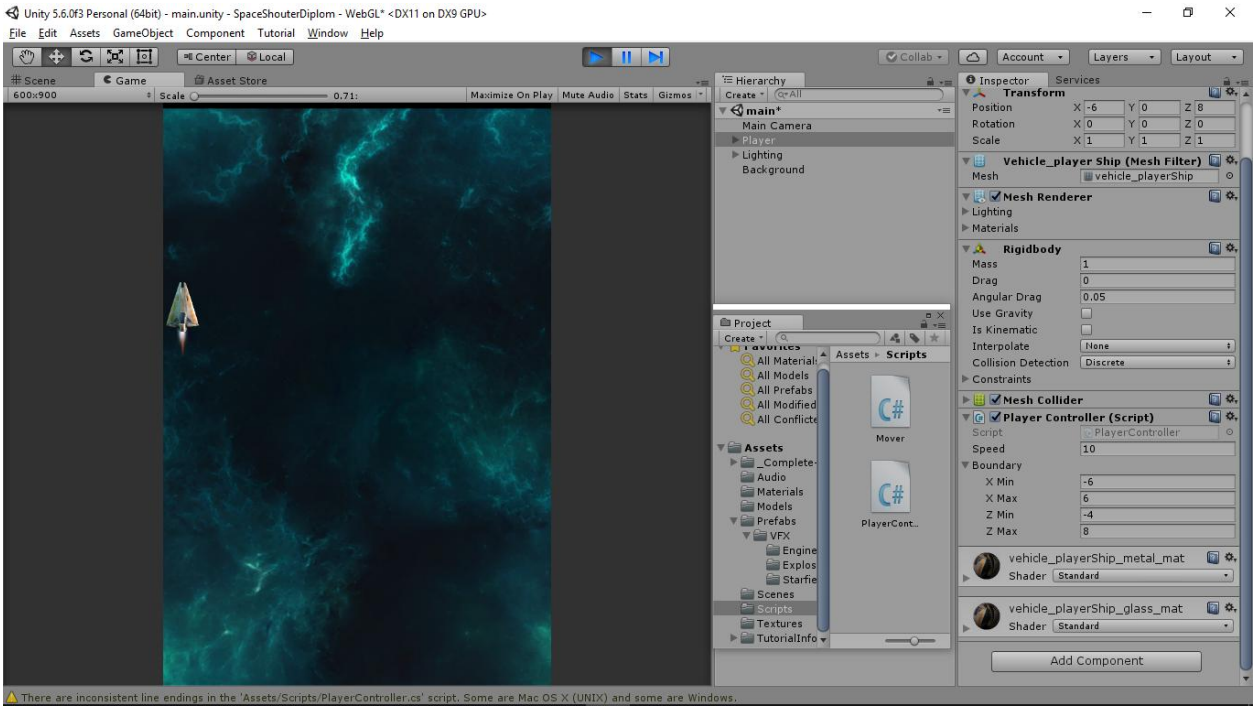

Рис 3.12. Перемещение корабля по сцене

Далее был добавлен наклон при перемещении корабля вдоль оси X с помощью свойства Rotation компонента RigidBody.

3.7 Создание выстрела и летящей пули

Снаряд должен быть похож на лазерный луч. Для создания эффекта летящего луча был добавлен объект Quad, повернутый на 90% по оси X и объект Material на который наложена текстура лазерного луча. Затем был сменен шейдер на Mobile-> Particles-> Additive, который рассматривает изображение как альфа канал.

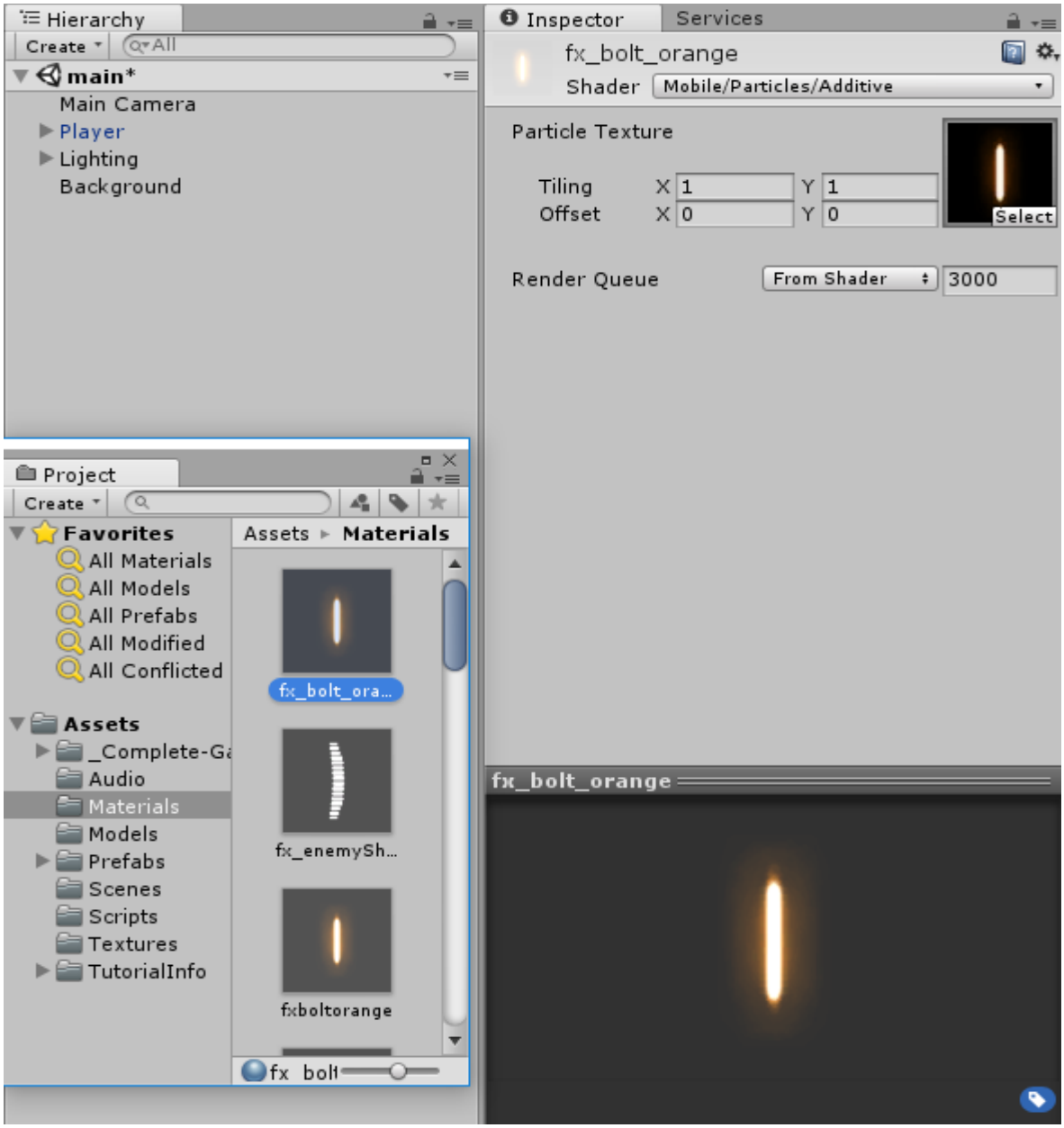

Рис 3.13. Окно создания выстрела

Далее для управления снарядом был добавлен компонент Rigid Body с отключенной гравитацией, Capsule Collider и скрипт Mover для автоматического перемещения лазерного луча вперед по сцене.

Код скрипта Mover:

```
public class Mover : MonoBehaviour {
    public float speed;
       public void Start()
    {
        GetComponent<Rigidbody>().velocity =
GetComponent<Rigidbody>().transform.forward * speed;
    }
       }
```

Корабль должен стрелять, для этого была создана анимация пули. По задумке пуля должна вылетать из передней части корабля, когда игрок нажимает кнопку, отвечающую за стрельбу. Идея в том, чтобы клонировать объект пули при нажатии кнопки. Для этого мы используем функцию Update() и функцию Instantiate() которая устанавливает позицию и угол поворота в сцене клонированного объекта. Ссылка на объект пули находится в переменной shot, а координаты передаются от переменной ShotSpaw компонента transform, которая в свою очередь ссылается на одноименный объект.

Фрагмент кода отвечающий за выстрел:

```
public GameObject shot;
    public Transform shotSpaw;
    public float fireRate = 0.5f;//отвечает, как часто будут вылетать пули
    public float nextFire = 0.0f;//регулирует разрешение на стрельбу


    public void Update()
    {
        if(Input.GetButton("Fire1") && Time.time > nextFire)
        {
            nextFire = Time.time + fireRate;
            Instantiate(shot, shotSpaw.position, shotSpaw.rotation);
        }

          }
```

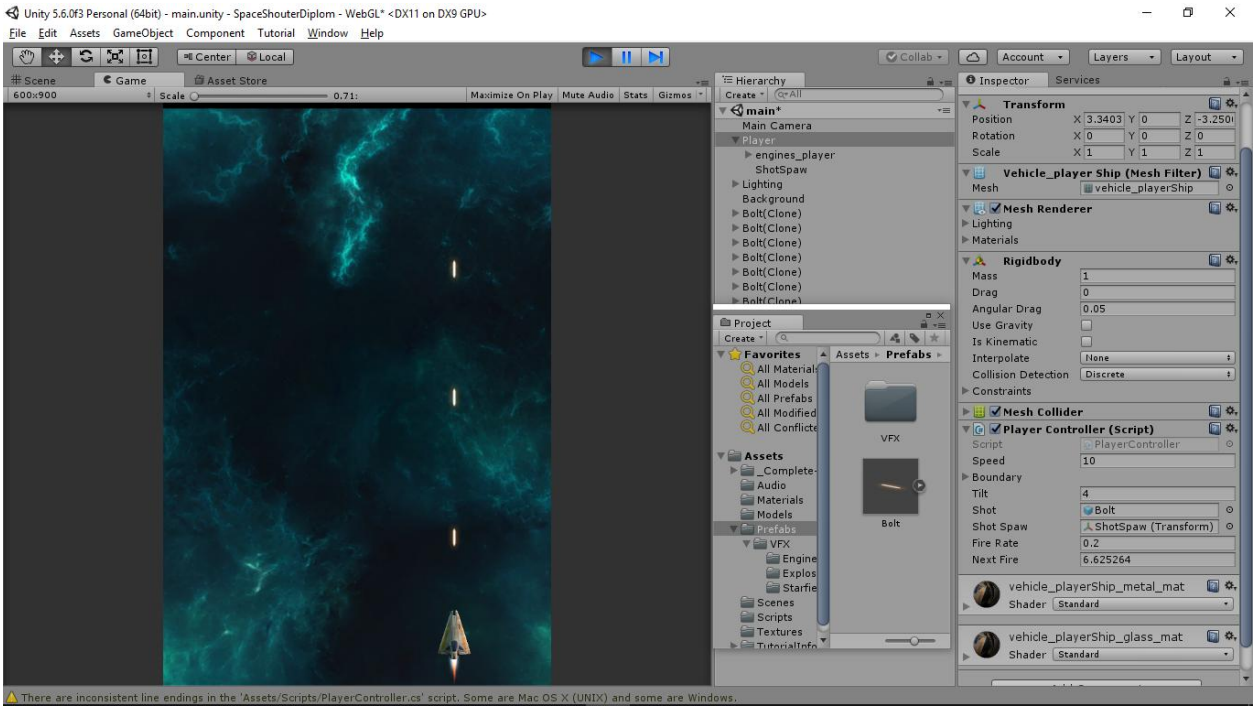

Рис 3.14. Стреляющий корабль

Пули, летящие по оси Z к бесконечности, накапливаются и занимают ресурсы компьютера. Для решения данной проблемы созданы границы игрового поля в виде куба. Любой объект попадающий за пределы границы куба удаляется. Скрипт DestroyByBoundary выполняющий данные задачи представлен ниже:

```
public class DestroyByBoundary : MonoBehaviour {

    private void OnTriggerExit(Collider other)
    {
        Destroy(other.gameObject);
    }
    }
```

3.8 Создание, вращение и уничтожение астероидов

Для создания вращающегося астероида на сцену был добавлен новый пустой объект Asteroid. К нему добавлена созданная в 3DMax модель prop_asteroid_01 и компоненты RigidBody и CapsuleCollaider. Поскольку игра ведется в плоскости X и Z флажок Use Gravity снят, чтобы астероид не проваливался в пустоту. Для придания эффекта вращения создан скрипт:

```
public float tumble;//задает скорость вращения астероида
    private Rigidbody rb;
       // Код внутри данной функции выполняется один раз при запуске сцены
       void Start () {
        rb = GetComponent<Rigidbody>();
        rb.angularVelocity = new Vector3(1, 1, 1)*tumble;
        rb.angularVelocity = Random.insideUnitSphere * tumble;
                  }
```

Чтобы астероид не останавливался, угловое сопротивление Angular Drag отключено.

Далее создан скрипт удаления астероида при взаимодействии с пулей корабля:

```
private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Bolt")
        {
            Destroy(other.gameObject);
            Destroy(gameObject);
        }
           }
```

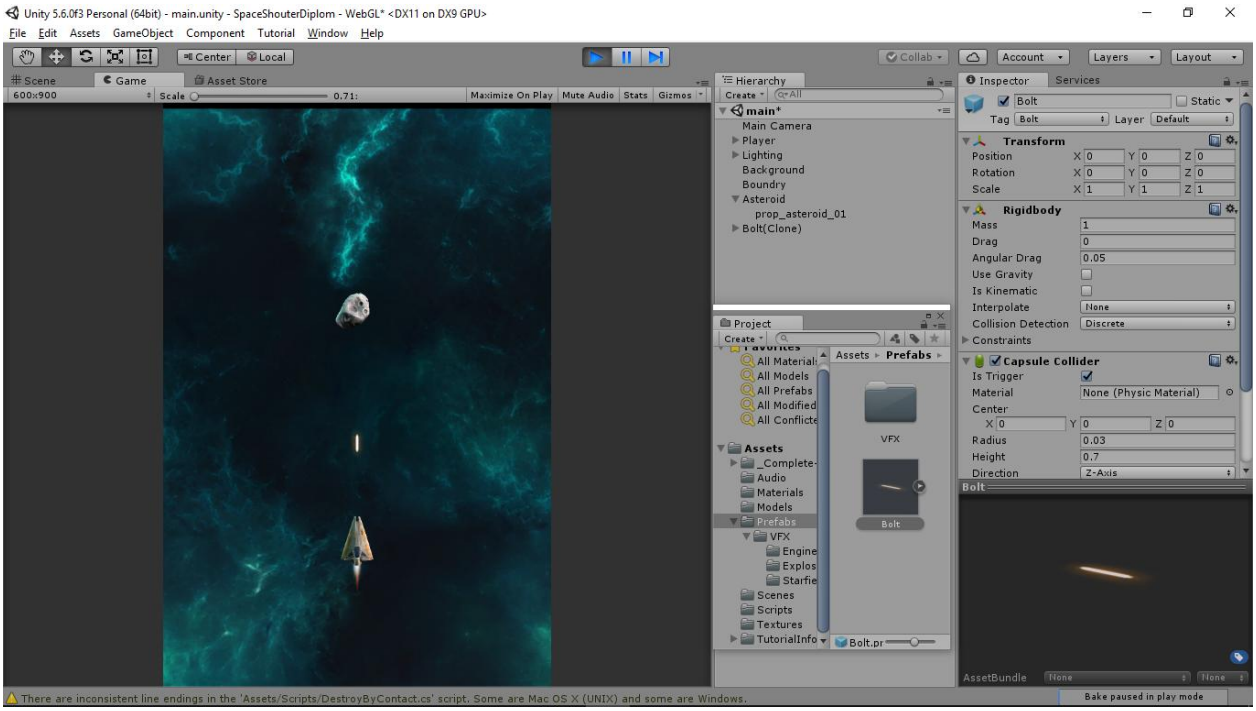

Рис 3.15. Вражеский астероид

В скрипте DestroyByContact создана переменная Explosion, которая будет ссылаться на визуальный объект взрыва. Для клонирования объектов применена статическая функция Instantiate, принимающая три параметра: существующий объект, позицию и поворот. Статические функции могут вызываться без создания экземпляра класса. Анимация взрыва была

создана в среде 3dMax и передана в переменную Explosion. Теперь, астероид взрывается при попадании пули в сетку его коллайдера.

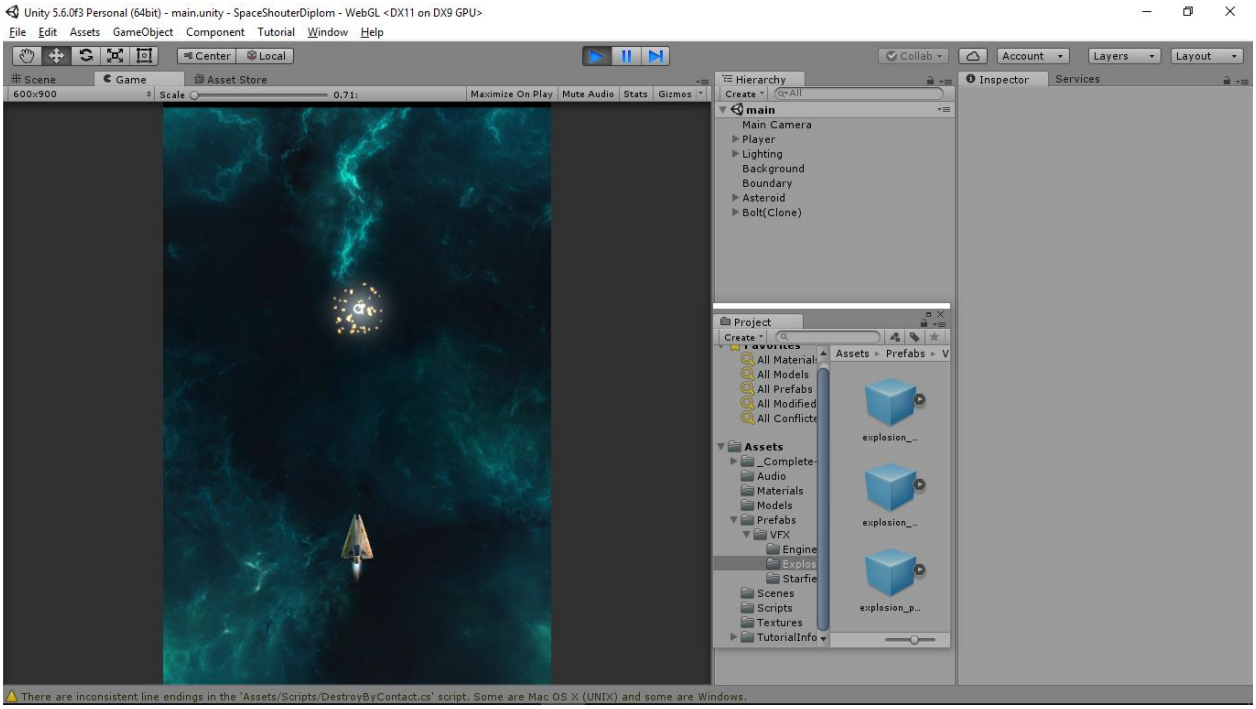

Рис 3.16. Взрыв астероида

Затем кораблю был задан тег Player и в этом же скрипте добавлена переменная explosionPlayer, хранящая эффект взрыва Корабля и условие, что при входящем объекте с тегом Player произойдет взрыв и удаление обоих объектов.

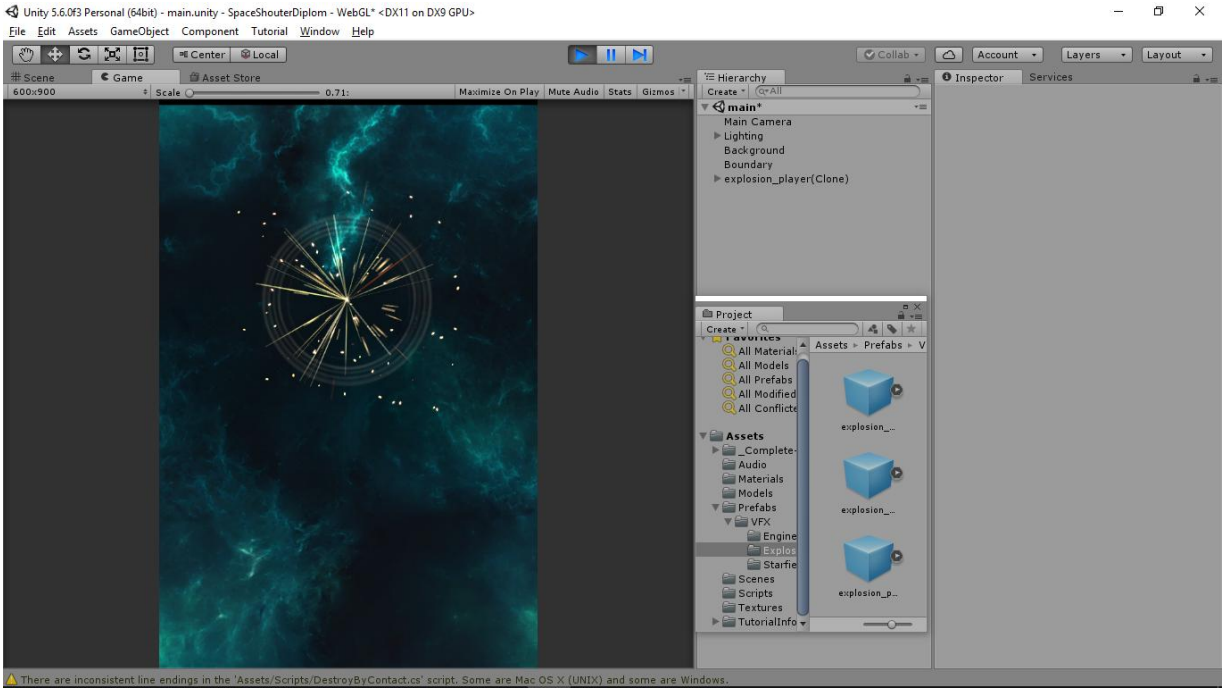

Рис 3.17. Взрыв корабля

Также добавлено движение астероида в сторону корабля с помощью скрипта, используемого для перемещения пули, с отрицательной скоростью.

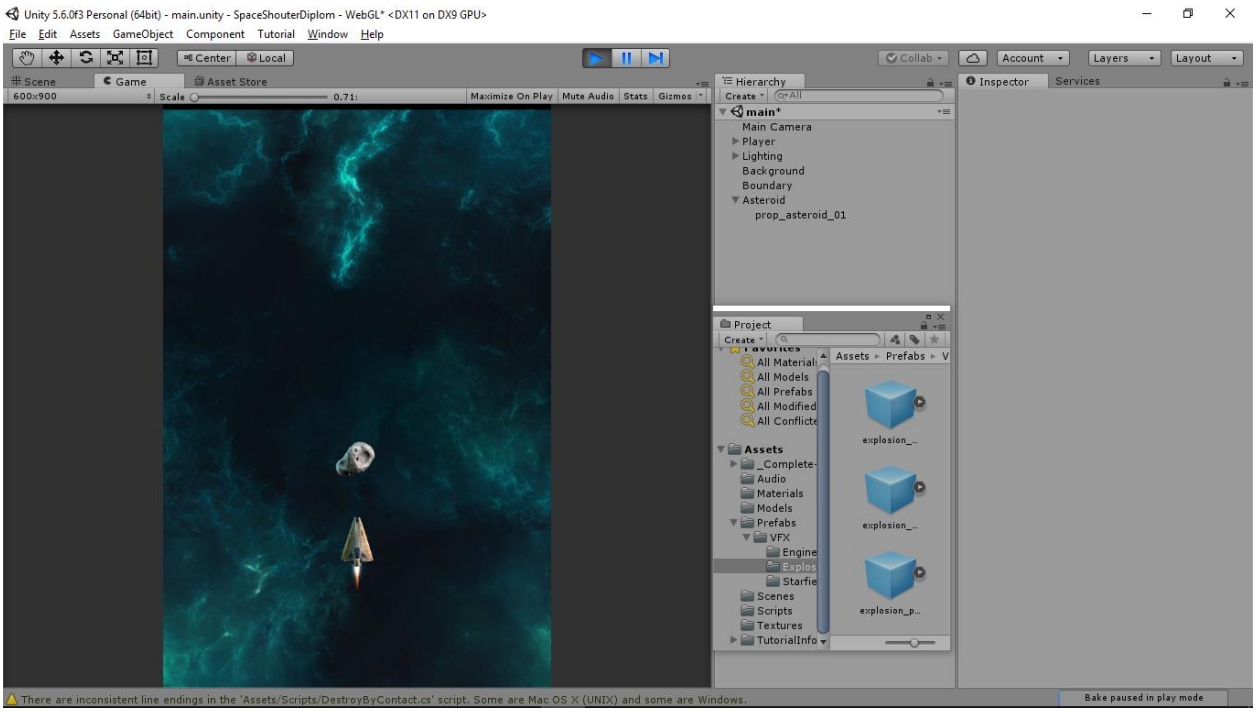

Рис 3.18. Перемещение астероида

Код работы скрипта:

```
public class DestroyByContact : MonoBehaviour
{
    public GameObject explosion;
    public GameObject explosionPlayer;
    private GameObject cloneExplosion;
    private void OnTriggerEnter(Collider other)
    {

        if(other.tag == "Player")
        {
            cloneExplosion = Instantiate(explosionPlayer,
GetComponent<Rigidbody>().position, GetComponent<Rigidbody>().rotation) as GameObject;

            Destroy(other.gameObject);
            Destroy(gameObject);
            Destroy(cloneExplosion, 1f);
        }
        if (other.tag == "Bolt")
        {
            cloneExplosion = Instantiate(explosion, GetComponent<Rigidbody>().position,
GetComponent<Rigidbody>().rotation) as GameObject;

            Destroy(other.gameObject);
            Destroy(gameObject);
            Destroy(cloneExplosion, 1f);
        }
    }
       }
```

3.9 Создание игрового контроллера, генерирующего клоны объектов.

Для автоматизации добавления различных объектов на игровое поле создан пустой объект, тег и скрипт GameController. Игровой контроллер выполняет несколько задач, однако основная задача — это генерация опасностей для корабля. За это отвечает функция SpawnWaves. Так как функция работает на протяжении всей игры она вызывается в самом начале игры с помощью функции Start для начала игрового процесса. На данном этапе в переменную Hazard хранящую ссылку на создаваемый объект поместим созданный ранее астероид. Для того, чтобы астероиды возникали на игровом поле в случайных позициях и со случайной задержкой воспользовались классом Random.

Для создания нескольких астероидов используется цикл и счетчик hazardCount определяющий необходимое количество астероидов. Также создана переменная spawnWait и StartWait для паузы в создании объектов и начале игры. С помощью цикла While созданы потоки волн астероидов, сменяющиеся друг за другом, пока не закончится игра или не будет уничтожен корабль.

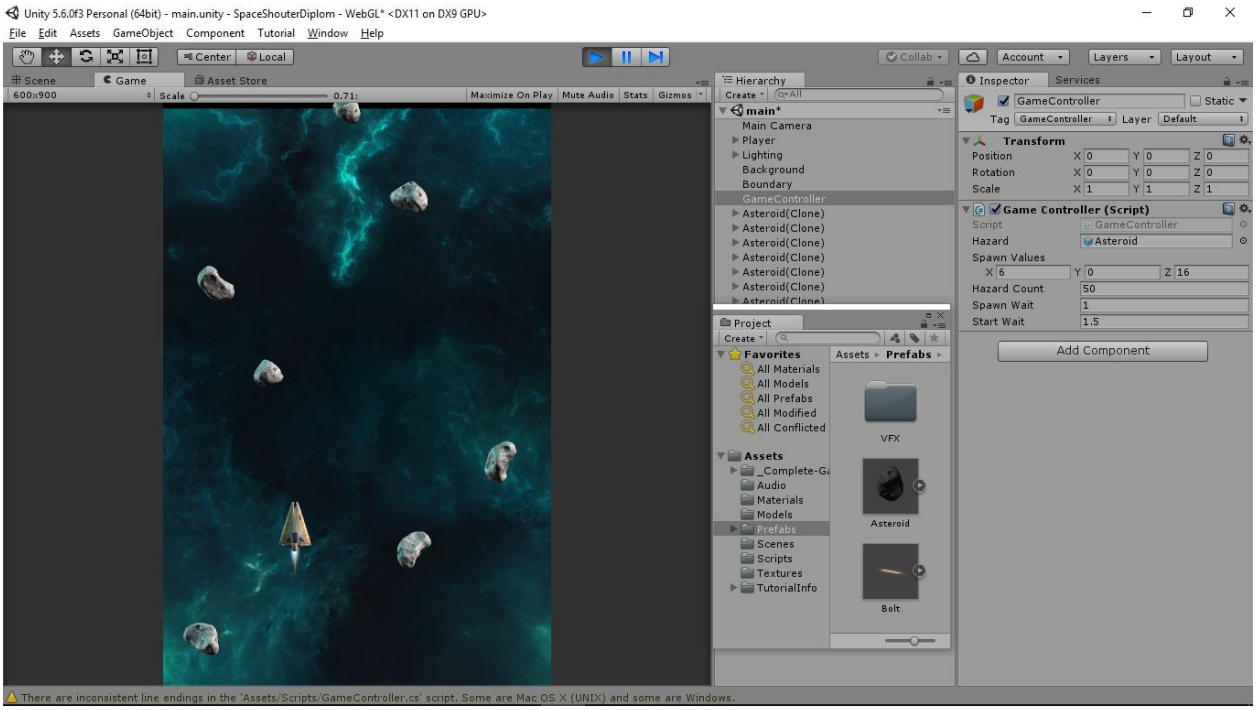

Рис 3.19. Процесс добавления астероидов на сцену

Код контролера:

```
public class GameController : MonoBehaviour {
    public GameObject hazard;
    public Vector3 spawnValues;
    public int hazardCount;
    public float spawnWait;
    public float startWait;
    public float waveWait;
    private void Start()
    {
        StartCoroutine( SpawnWaves());
    }
    IEnumerator SpawnWaves()
    {
        yield return new WaitForSeconds(startWait);
        while (true)
        {
            for (int i = 0; i < hazardCount; i++)
            {
                Vector3 spawnPosition = new Vector3(Random.Range(-spawnValues.x,
spawnValues.x), spawnValues.y, spawnValues.z);
                Quaternion spawnRotation = Quaternion.identity;
                Instantiate(hazard, spawnPosition, spawnRotation);

                yield return new WaitForSeconds(Random.Range(0.5f, spawnWait));
            }
            yield return new WaitForSeconds(waveWait);
        }
    }
}
```

3.10 Звук

Ни одна современная игра не обходится без музыкального сопровождения и всевозможных звуковых эффектов. Так и при создании данного приложения, была необходимость использовать звук.

3.10.1 Принцип и особенности работы со звуком в Unity3D

Работа со звуком в Unity3D также относительно проста, хотя возможности весьма велики. Можно накладывать фильтры, закреплять звуковые эффекты за объектами, в результате чего громкость и стереоэффект регулируется автоматически. Для того чтобы закрепить за объектом звук, необходимо создать и присоединить к объекту экземпляр

класса AudioSource, а потом, обращаясь к методам, добавлять к проигрыванию необходимые ресурсы. Например, код для звука выстрела корабля: GetComponent<AudioSource>().Play();

3.10.2 Работа со звуком.

В игре существуют несколько видов звуков. Первый – это мгновенные звуки, которые звучат, когда происходит выстрел, взрыв и т.д. Второй тип — это фоновая музыка – звук, который необходимо проигрывать пока приложение запущенно, но так как время работы приложения может быть велико, нецелесообразно делать файл большого размера, гораздо проще повторно запускать фоновый звук. Для добавления звукового сопровождения к объекту достаточно переместить аудио файл на объект. В это случае Unity сам создаст компонент Audio Source и в свойстве Audio Clip создаст ссылку на аудио файл.

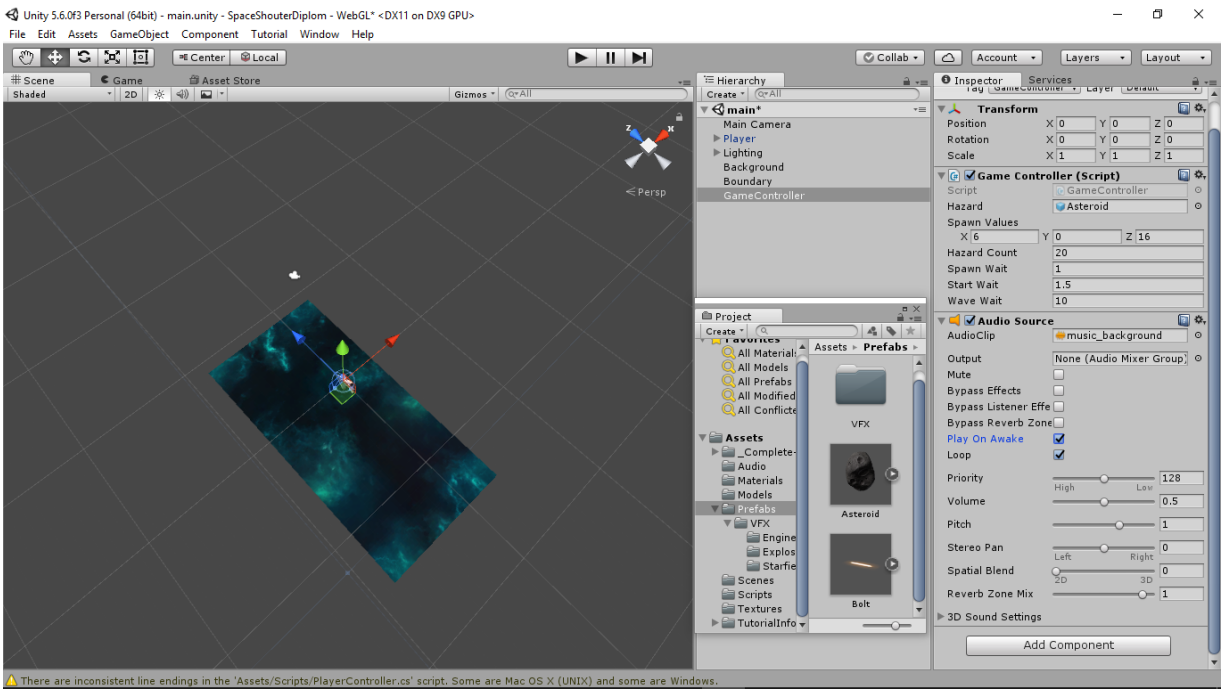

Рис 3.20. Окно настройки звукового компонента

3.11 Подсчет набранных очков

Очки в компьютерных и/или ролевых играх — вознаграждение, выражаемое в числовой форме, получаемое игроком за успешное выполнение тех или иных действий, часто — уничтожение врагов. Накопление очков опыта при достижении некоторого порогового значения ведёт к повышению уровня. Идея подобного вознаграждения тесно связана с идеями протестантской этики, господствующей в США.

В игре “Space Attack” игрок получает 10 очков за каждый сбитый астероид и 30 очков за каждый сбитый корабль. Для подсчета очков был создан объект ScoreText и подключен компонент GUI Text. С помощью данного компонента производится настройка расположения текста на сцене, его шрифт, размер и т.д. После настройки текстового индикатора был создан скрипт для подсчета очков, представленный ниже.

```
  public GUIText scoreText;//ссылка на текстовый объект отражающий счет
  private int score;//подсчет набранных очков
private void Start()
    {

        score = 0;
        UpdateScore();
           }
void UpdateScore()
    {
        scoreText.text = "Счёт: " + score;
    }
public void AddScore(int newScoreValue){
        score += newScoreValue;
        UpdateScore();
        }
```

Данные передаются в переменную при взрыве астероида. Для этого был также редактирован скрипт DestroyByContact.

```
public int scoreValue;
private GameController gameController;//ссылка на класс
  private void Start()
    {
        GameObject GameControllerObject = GameObject.FindWithTag("GameController");
        if(GameControllerObject != null)
```

```
        {
            gameController = GameControllerObject.GetComponent<GameController>();
        }
        if (GameControllerObject == null)
        {
            Debug.Log("Скрипт 'Game Controller' не найден");
        }
    }
gameController.AddScore(scoreValue);
```

Счетчик обновляется в двух местах. В самом начале игры и когда астероид разрушается. Когда игра начинается необходимо обнулить значение счетчика. Для этого значение переменной score установлено в 0.

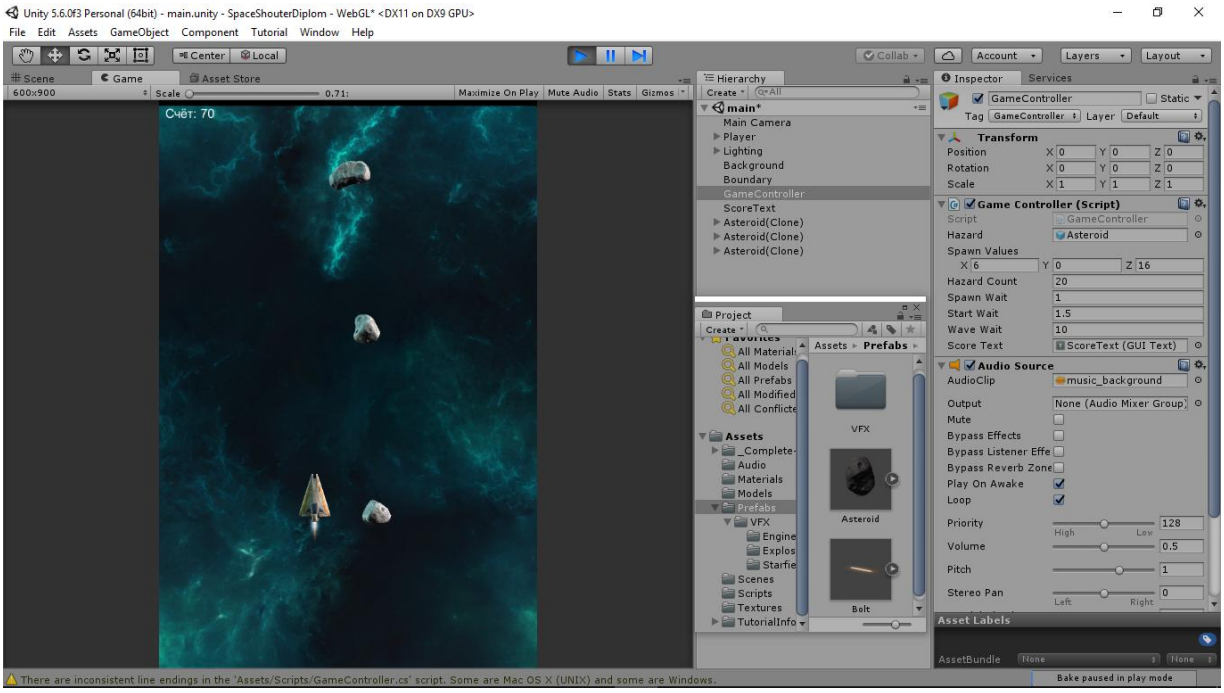

Рис 3.21. Отображение набранных очков

3.12 Перезагрузка игры

Для того, чтобы игра заканчивалась при уничтожении корабля не продолжаясь бесконечно, а также для возможности перезагрузки игры добавим два текстовых объекта на сцену GameOver Text и Restart text. Также редактируем скрипт Game Controller, чтобы управлять текстовыми объектами через ссылки. Код представлен ниже:

```
public GUIText restartText;
    public GUIText gameOverText;

    private bool gameover;
    private bool restart;

    public void Update()
    {
        if (restart)
        {
            if (Input.GetKeyDown(KeyCode.R))
            {
                SceneManager.LoadScene("main", LoadSceneMode.Single);
//будет загружена только main сцена, а остальные будут закрыты
            }
        }
    }   private void Start()
    {
        gameOver = false;
        restart = false;
        restarText.text = "";//присваеваем пустые поля этим переменным чтобы они не
отображались
        gameOverText.text = "";
           }
if (gameover) {
                restarText.text = "Нажмите R для перезагрузки";
                restart = true;
                break;
                   }
public void gameOver()
    {
        gameOverText.text = "Игра окончена!";
        gameover = true;
           }
```

Сигналом окончания игры служит уничтожение корабля. Это событие отслеживает скрипт астероида Destroy By Contact. Он должен вызывать ункцию GameOver(), поэтому добавлен код:

```
gameController.GameOver();
```

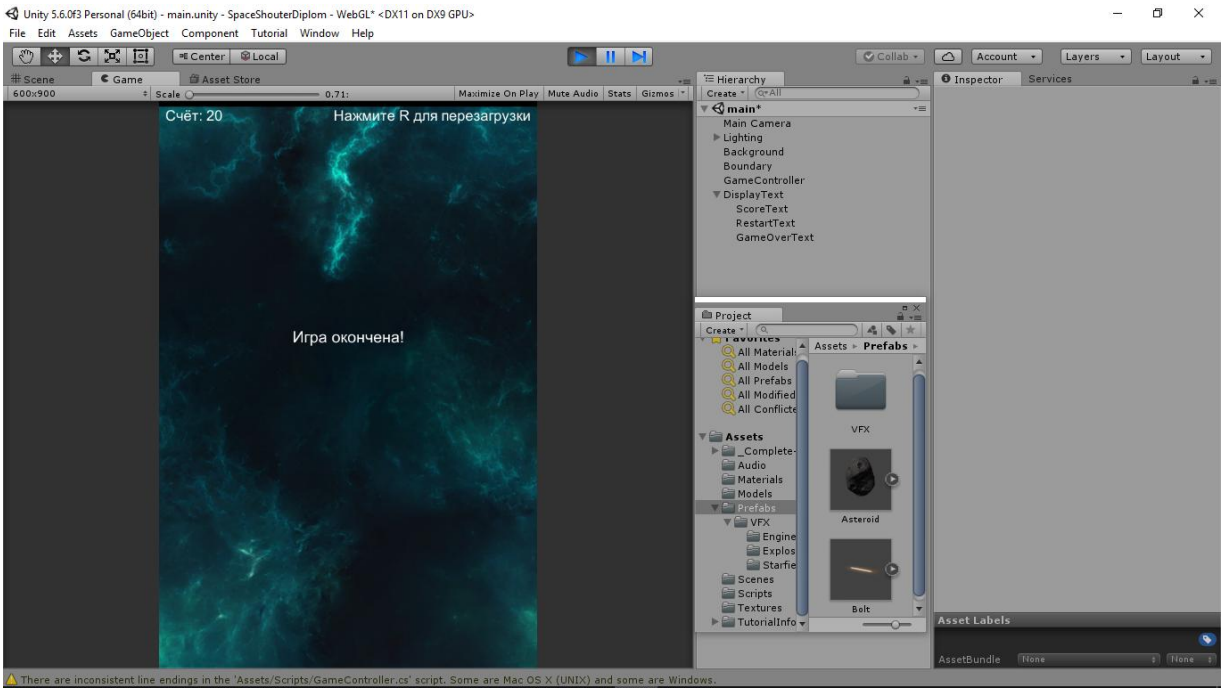

Рис 3.22. Окно окончания игры

Теперь при уничтожении корабля в центре экрана появляется Надпись игра окончена, а для перезагрузки игры необходимо нажать клавишу R.

3.13 Создание вражеских кораблей

Для создания вражеского корабля на сцену добавлен пустой объект Enemy Ship в который вставлена модель и анимация двигателя корабля, созданная в 3DMax. Чтобы была возможность управлять кораблем добавлен компонент Rigid Body. Для возможности взаимодействия с кораблем добавлен компонент Sphere Collider и установлена галочка Is Trigger.

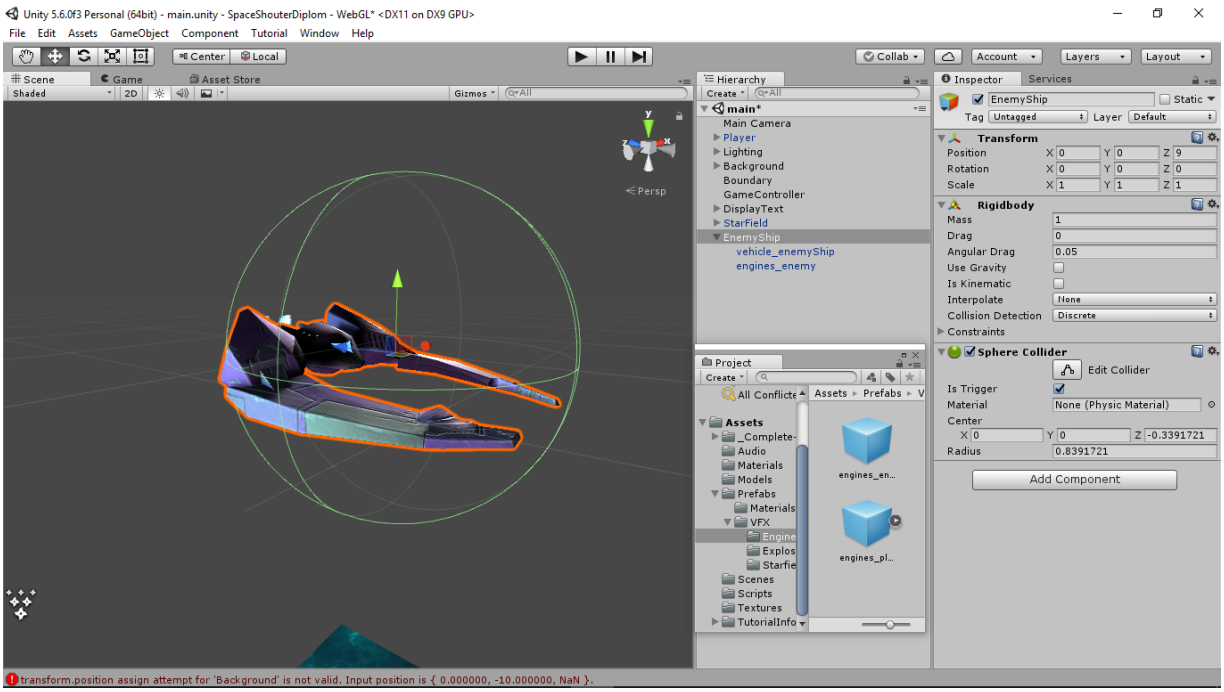

Рис 3.23. Модель вражеского корабля

Так как условия взаимодействия с окружающими объектами у вражеского корабля такие же как у главного корабля скрипт взрыва при контакте с пулей или кораблем остается тот же. Чтобы научить корабль стрелять и летать дополнительно создан скрипт Weapon Controller. Код скрипта представлен ниже:

```
public class WeaponController : MonoBehaviour {

    public GameObject shot;//ссылка на префаб пули
    public Transform shotSpaw;//ссылка на объект в координатах которого создается пуля
    public float fireRate;// частота с которой будут вылетать пули
    public float delay;//задержка функции Repeating
    private AudioSource audioS;

    private void Start()
    {
        audioS = GetComponent<AudioSource>();//получаем ссылку на аудио фаил текущего
объекта
        InvokeRepeating("Fire", delay, fireRate);//функция вызывающая в 1 параметре
метод, через Delay секнуд, с частотой указанной в 3 параметре
    }

    private void Fire()
    {
        Instantiate(shot, shotSpaw.position, shotSpaw.rotation);
        audioS.Play();
    }

    }
```

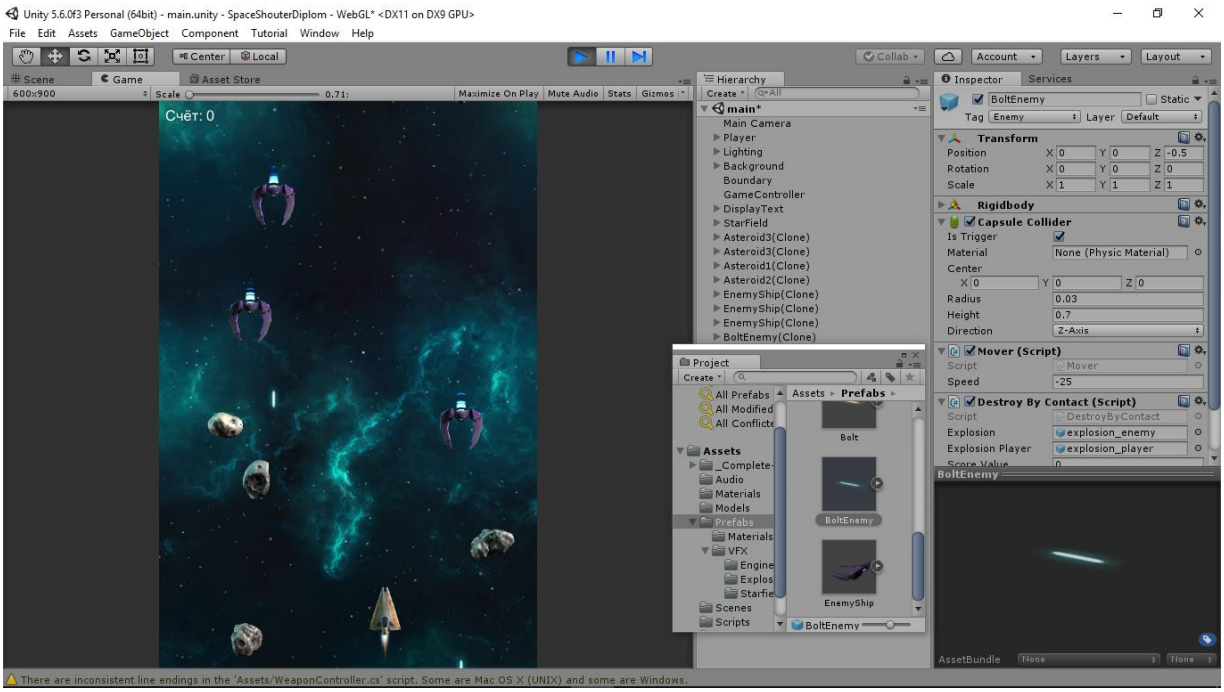

Рис 3.24. Атакующие вражеские корабли

Для движения корабля создан скрипт EnemyMoneuver. Принцип работы заключается в том, что корабль со случайной периодичностью смещается влево или вправо, при этом продолжая двигаться вперёд. Также обязательным условием является невозможность выхода корабля за пределы игровой плоскости. Для того, чтобы программа не зависла в бесконечном цикле, цикл помещён в функцию. Применяемая функция позволяет выходить из неё в основной поток программы, не завершая свою работу. Запускаться данная функция будет один раз при появлении корабля на сцене.

3.14 Компиляция проекта и запуск игры

Игра полностью совместима с мобильными операционными системами. После разработки были построены три проекта – для операционной системы Windows x86-x64, для Android и IOS. При запуске игры появляется диалоговое окно с основными настройками, разрешением экрана и конфигурацией управления, которую можно менять. В процессе

тестирования игры ошибок не замечено. Продукт готов для запуска на рынке мобильных приложений.

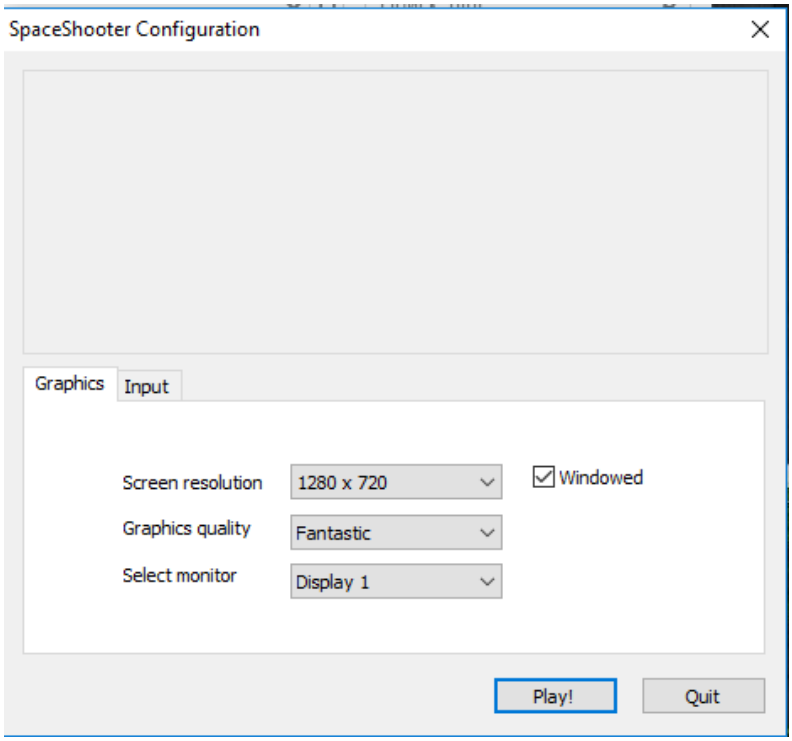

Рис 2.25. Окно настройки игры

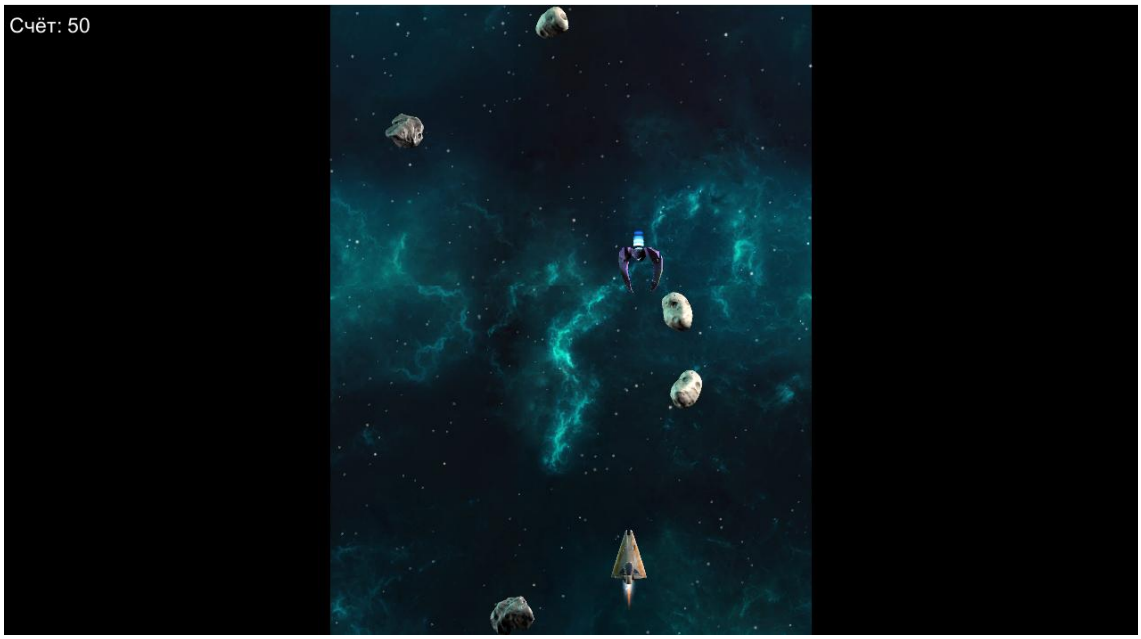

Рис 3.26. Игровой процесс

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы решена научно-техническая задача создания игры, удовлетворяющей требованиям начальной спецификации, исследование технологий создания 3D - игр, а также приемов программирования для смартфонов.

Выводы и результаты работы сводятся к следующему.

1. Спроектирована и реализована 3D - игра “Space Attack”. Получены большие знания в области разработки приложений: работа с графикой, памятью, оптимизацией производительности, искусственным интеллектом.

2. Изучена среда разработки Unity3D: возможности редактора движка, слабые и сильные стороны этого движка. Произведен обзор графических возможностей системы и пользовательского интерфейса.

3. Получен опыт разработки 3D сцен: изучены основные понятия 3D моделирования, изучен язык написания шейдеров, изучена и понята работа с текстурами как в 3D, так и в 2D.

4. Приобретен опыт программирования на языке C# и работы с использованием объектно-ориентированного программирования. Более детально изучена библиотека .NET. Изучена и проанализирована библиотека Unity3D, составлены собственные классы, которые её дополняют.

.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Официальная документация по Unity3D [Электронный ресурс] - URL: http://unity3d.com/support/documentation (дата обращения: 06.04.2010).

2. Официальное сообщество разработчиков на Unity3D [Электронный ресурс] -URL: http://unity3d.com/support/community (дата обращения: 26.04.2010).

3. Русское сообщество разработчиков на Unity3D [Электронный ресурс] - URL: http://forum.unity3d.com (дата обращения: 06.04.2010).

4. Типовые примеры и решения при разработке приложений на Unity3D [Электронный ресурс] - URL: http://blogs.unity3d.ru (дата обращения: 01.05.2010).

5. Справочная информация по устройству фирмы Apple - Apple iPhone [Электронный ресурс] - URL: http://www.apple.com/support/iphone (дата обращения: 11.05.2010).

6. Справочная информация по устройству фирмы Apple - Apple iPod touch [Электронный ресурс] - URL: http://www.apple.com/support/ipod (дата обращения: 26.05.2010).

7. Официальная документация по Unity iPhone [Электронный ресурс] URL: http://unity3d.com/unity/features/iphone-publishing.html (дата обращения: 12.04.2010).

8. Троелсен Э. C# и платформа .NET. - СПб.:Питер, 2004. - 796 с.- (Библеотека программиста).

9. Орлов С.А. Технология разработки программного обеспечения - Питер,2003.-464с.-(Учебник для ВУЗов).

10. Энди Кармайкл, Дэн Хейвуд. Быстрая и качественная разработка программного обеспечения.-Вильямс,2003.-400 с.

11. Орловский. С. Нас ждет ренессанс стратегий [Электронный ресурс]. Режим доступа:http://kanobu.ru/articles/nas-zhdet-renessans-strategij-300471/

12. Скорик. М. Gamification мобильных игр [Электронный ресурс]. Режим доступа:http://habrahabr.ru/post/167595/

13. Fierz M. Strategy Game Programming [Электронный ресурс]. Режим доступа:http://www.fierz.ch/strategy1.htm

14. Saltzman. M. General Game Design: Strategy Games. – 2-nd ed., Brady Games, 2000

15. Community of independent game players and creators [Электронный ресурс]. Режим доступа:"http://www.tigsource.com/

16. Micic A., Arnarsson D., and Jonsson V. Developing Game AI for the Real-Time Strategy Game StarCraft. Technical report, Reykjavik University, 2011.

17. Ильченко В. Изучение методов построения игрового искусственного интеллекта и разработка информационной технологии для ее реализации в стратегиях реального времени [Электронный ресурс]. Режим доступа:http://masters.donntu.org/2013/fknt/ilchenko/

18. Буга К. Изучение методов построения игрового искусственного интеллекта и разработка информационной технологии для ее реализации в пошаговых стратегиях [Электронный ресурс]. Режим доступа:"http://masters.donntu.org/2013/fknt/buga/

19. Илькун В. Инструментальные средства для разработки компьютерных игр жанра экшн от первого и третьего лица [Электронный ресурс]. Режим доступа:"http://masters.donntu.org/2013/fknt/ilkun/

20. Валиуллин С. Этапы разработки игры глазами гейм-дизайнера [Электронный ресурс]. Режим доступа:"http://www.gamedev.ru/gamedesign/articles/development_planning

21. Зыков И. Компьютерные игры: этапы разработки [Электронный ресурс]. Режим доступа:http://www.megabyte-web.ru/likbez/igryi-etapyi-razrabotki.html

ПРИЛОЖЕНИЕ. ТЕКСТЫ ПРОГРАММ

Листинг скрипта DestroyByBoundary.css

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

public class DestroyByBoundary : MonoBehaviour {

    private void OnTriggerExit(Collider other)
    {
        Destroy(other.gameObject);
    }
       }
```

Листинг скрипта DestroyByContact

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

public class DestroyByContact : MonoBehaviour
{
    public GameObject explosion;
    public GameObject explosionPlayer;
    private GameObject cloneExplosion;
    public int scoreValue;
    private GameController gameController;//ссылка на класс

    private void Start()
    {
        GameObject GameControllerObject = GameObject.FindWithTag("GameController");
        if(GameControllerObject != null)
        {
            gameController = GameControllerObject.GetComponent<GameController>();
        }
        if (GameControllerObject == null)
        {
            Debug.Log("Скрипт 'Game Controller' не найден");
        }
    }

    private void OnTriggerEnter(Collider other)


    {

        if(other.tag == "Player")
        {
```

```
            cloneExplosion = Instantiate(explosionPlayer,
GetComponent<Rigidbody>().position, GetComponent<Rigidbody>().rotation) as GameObject;

            gameController.GameOver();

            Destroy(other.gameObject);
            Destroy(gameObject);
            Destroy(cloneExplosion, 1f);
        }
        if (other.tag == "Bolt")
        {
            cloneExplosion = Instantiate(explosion, GetComponent<Rigidbody>().position,
GetComponent<Rigidbody>().rotation) as GameObject;

            Destroy(other.gameObject);
            Destroy(gameObject);
            Destroy(cloneExplosion, 1f);

            gameController.AddScore(scoreValue);
        }
    }
       }
```

Листинг скрипта GameController

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class GameController : MonoBehaviour {
    public GameObject [] hazards;
    public Vector3 spawnValues;
    public int hazardCount;
    public float spawnWait;
    public float startWait;
    public float waveWait;
    public GUIText scoreText;//ссылка на текстовый объект отражающий счет
    private int score;//подсчет набранных очков
    public GUIText restartText;
    public GUIText gameOverText;
    private bool gameover;
    private bool restart;
    public void Update()
    {
        if (restart)
        {
            if (Input.GetKeyDown(KeyCode.R))
            {
                SceneManager.LoadScene("main", LoadSceneMode.Single);//будет загружена
только main сцена а остальные будут закрыты
            }
        }  }
    private void Start()
    {
        gameover = false;
        restart = false;
        restartText.text = "";
        gameOverText.text = "";
```

```
        StartCoroutine( SpawnWaves());
        score = 0;
        UpdateScore();
    }
    IEnumerator SpawnWaves()
    {
        yield return new WaitForSeconds(startWait);
        while (true)
        {
            for (int i = 0; i < hazardCount; i++)
            {
                Vector3 spawnPosition = new Vector3(Random.Range(-spawnValues.x,
spawnValues.x), spawnValues.y, spawnValues.z);
                Quaternion spawnRotation = Quaternion.identity;

                GameObject hazard = hazards[Random.Range(0,hazards.Length)];

                Instantiate(hazard, spawnPosition, spawnRotation);

                yield return new WaitForSeconds(Random.Range(0.5f, spawnWait));
            }
            yield return new WaitForSeconds(waveWait);

            if (gameover) {
                restartText.text = "Нажмите R для перезагрузки";
                restart = true;
                break;
            }

        }
    }
    void UpdateScore()
    {
        scoreText.text = "Счёт: " + score;
    }
    public void AddScore(int newScoreValue){
        score += newScoreValue;
        UpdateScore();
        }

    public void GameOver()
        {
            gameOverText.text = "Игра окончена!";
            gameover = true;
        }
}
```

Листинг скрипта Mover

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

public class Mover : MonoBehaviour {
    public float speed;
       public void Start()
    {
```

```
        GetComponent<Rigidbody>().velocity = GetComponent<Rigidbody>().transform.forward
* speed;
    }
}
```

Листинг скрипта PlayerController

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[System.Serializable]
public class Boundary
{
    public float xMin=-6, xMax=6, zMin=-4, zMax=8;
}

public class PlayerController : MonoBehaviour {

    public float speed = 10;
    public Boundary boundary;
    public float tilt;//для угла наклона

    public GameObject shot;
    public Transform shotSpaw;
    public float fireRate = 0.5f;//отвечает как часто будут вылетать пули
    public float nextFire = 0.0f;//регулирует разрешение на стрельбу


    public void Update()
    {
        if(Input.GetButton("Fire1") && Time.time > nextFire)
        {
            nextFire = Time.time + fireRate;
            Instantiate(shot, shotSpaw.position, shotSpaw.rotation);
            GetComponent<AudioSource>().Play();
        }
    }
    private void FixedUpdate()
    {
        //метод GetAxis определяет к какой из осей обратился пользователь
        //другими словами какие кнопки нажаты(вправо, влево, вверх, вниз)
        //метод возвращает значение от -1 до 1
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");

        GetComponent<Rigidbody>().rotation = Quaternion.Euler(
            0f,
            0f,
            GetComponent<Rigidbody>().velocity.x * -tilt
            );
        //Rigidbody>().velocity принимает значение от структуры Vector3 и двигает корабль
        GetComponent<Rigidbody>().velocity = new Vector3(moveHorizontal, 0f,
moveVertical)*speed;
        GetComponent<Rigidbody>().position = new Vector3(
            Mathf.Clamp(GetComponent<Rigidbody>().position.x, boundary.xMin,
boundary.xMax),
            0f,
```

```
            Mathf.Clamp(GetComponent<Rigidbody>().position.z, boundary.zMin,
boundary.zMax)
            );
    }
       }
```

Листинг скрипта RandomRotator

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

public class RandomRotator : MonoBehaviour {

    public float tumble;//задает скорость вращения астероида
    private Rigidbody rb;

       void Start () {// Код внутри данной функции выполняется один раз при запуске сцены
        rb = GetComponent<Rigidbody>();
        rb.angularVelocity = new Vector3(1, 1, 1)*tumble;
        rb.angularVelocity = Random.insideUnitSphere * tumble;
            }
       }
```

Листинг скрипта ScrollBackground

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

public class ScrollBackground : MonoBehaviour {

    public float scrollSpeed;
    public float tileSize;
    private Transform currentObject;

    void Start()
    {
        currentObject = GetComponent<Transform>();

    }

    void Update()
    {
        currentObject.position = new Vector3(currentObject.position.x,
currentObject.position.y,
            Mathf.Repeat(Time.time*scrollSpeed, tileSize));
    }
             }
```

Листинг скрипта WeaponController

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

public class WeaponController : MonoBehaviour {

    public GameObject shot;//ссылка на префаб пули
    public Transform shotSpaw;//ссылка на объект в координатах которого создается пуля
    public float fireRate;// частота с которой будут вылетать пули
    public float delay;//задержка функции Repeating
    private AudioSource audioS;

    private void Start()
    {
        audioS = GetComponent<AudioSource>();//получаем ссылку на аудио фаил текущего
объекта
        InvokeRepeating("Fire", delay, fireRate);//функция вызывающая в 1 параметре
метод, через Delay секнуд, с частотой указанной в 3 параметре
    }

    private void Fire()
    {
        Instantiate(shot, shotSpaw.position, shotSpaw.rotation);
        audioS.Play();
    }

}
```

Листинг скрипта EnemyMoneuver

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

public class EnemyMoneuver : MonoBehaviour {

    public float dodge;
    public float smoothing;
    public float tilt;
    public Vector2 startWait;
    public Vector2 maneuverTime;
    public Vector2 maneuverWait;
    public Boundary boundary;

    private float currentSpeed;
    private float targetManeuver;
    private Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
        currentSpeed = rb.velocity.z;
        StartCoroutine(Evade());
```

```
    }

    IEnumerator Evade()
    {
        yield return new WaitForSeconds(Random.Range(startWait.x, startWait.y));

        while (true)
        {
            targetManeuver = Random.Range(1, dodge) * -Mathf.Sign(transform.position.x);
            yield return new WaitForSeconds(Random.Range(maneuverTime.x,
maneuverTime.y));
            targetManeuver = 0;
            yield return new WaitForSeconds(Random.Range(maneuverWait.x,
maneuverWait.y));
        }
    }

    void FixedUpdate()
    {
        float newManeuver = Mathf.MoveTowards(rb.velocity.x, targetManeuver,
Time.deltaTime * smoothing);
        rb.velocity = new Vector3(newManeuver, 0.0f, currentSpeed);
        rb.position = new Vector3
        (
            Mathf.Clamp(rb.position.x, boundary.xMin, boundary.xMax),
            0.0f,
            Mathf.Clamp(rb.position.z, boundary.zMin, boundary.zMax)
        );

        rb.rotation = Quaternion.Euler(0.0f, 0.0f, rb.velocity.x * -tilt);
    }
   }
```

Выпускная квалификационная работа выполнена мной совершенно самостоятельно. Все использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

«___» _____ _____ г.

_____ _____

(подпись) *(Ф.И.О.)*