МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ

ОСНОВЫ РАЗРАБОТКИ ИНТЕРАКТИВНЫХ ТРЕХМЕРНЫХ ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ UNITY

Лабораторный практикум

ГУАП
Санкт-Петербург
2019

УДК
ББК
Б

Рецензенты:
доктор экономических наук, доцент *А. С. Будагов*;
доктор экономических наук, профессор *Е. В. Корчагина*

Утверждено
редакционно-издательским советом университета
в качестве лабораторного практикума

**Авторы: А. В. Никитин, Н. Н. Решетникова,
М. Е. Ведерникова, И. А.Гибин, А. А. Деменко,
Е. А. Ичетовкин, А. С. Логинов, Е. М. Лунин,
Д. С. Потехин, А. Н. Сергеев, С. И. Собашников**

Б Основы разработки интерактивных трехмерных приложений на платформе Unity: лабораторный практикум / А. В. Никитин, Н. Н. Решетникова, М. Е. Ведерникова и др. – СПб.: ГУАП, 2019. – 163 с.

В лабораторном практикуме приведен цикл лабораторных работ по изучению мультиплатформенной среды разработки двух- и трехмерных приложений и игр Unity версии 2017 и выше, а также созданию приложений дополненной реальности с помощью библиотеки Vuforia, работающими под операционными системами windows и android с использованием устройств Oculus Rift, Gear VR, Microsoft HoloLens, Leap Motion.

Лабораторный практикум предназначен для студентов направления 09.04.01 «Информатика и вычислительная техника», изучающих дисциплины магистерской программы «Системы мультимедиа и компьютерная графика».

УДК
ББК

Введение

Unity – это инструмент для создания двух- и трёхмерных интерактивных приложений и игр. Созданные с помощью Unity приложения работают под наиболее распространенными операционными системами – Microsoft Windows, macOS, Linux, iOS, Android и др. на различных устройствах – персональные компьютеры, мобильные и носимые устройства, игровые консоли. Имеется возможность создавать приложения для запуска в браузерах с помощью специального подключаемого модуля Unity (Unity Web Player), а также с помощью реализации технологии WebGL.

В лабораторном практикуме приведены обновленные версии лабораторных работ № 1–10 из [3] в связи с появлением новых долгосрочных версий Unity, а также ряд новых по созданию игровой анимации на основе смешивания двух или более похожих анимаций; android и windows приложений дополненной реальности с помощью библиотеки Vuforia, в т.ч. с использованием для работы устройства Microsoft HoloLens; windows и android приложений соответственно для устройств Oculus Rift и Gear VR; приложения для Leap Motion для бесконтактного человеко-компьютерного взаимодействия.

Лабораторные работы подготовили магистранты М. Е. Ведерникова (1), И. А. Гибин (17), А. А. Деменко (2), Е. А. Ичетовкин (3,5,12), А. С. Логинов (8,10), Е. М. Лунин (6,7), Д.С. Потехин (9,11,13), А. Н. Сергеев (4), С. И. Собашников (14,15,16).

Авторское редактирование выполнила А. А. Деменко.

Общая тематическая и содержательная редакция выполнены доцентами, канд. техн. наук А. В. Никитиным и Н. Н.Решетниковой.

Лабораторная работа № 1
УСТАНОВКА ПАКЕТА И ЗНАКОМСТВО С ИНТЕРФЕЙСОМ ПРОГРАММЫ UNITY 3D

Цель работы: изучение алгоритма подготовки программной среды, приобретение базовых знаний об интерфейсе программы *Unity 3D*.

Порядок выполнения работы

1. Загрузить программу *Unity 3D* с официального сайта.
2. Выполнить инсталляцию и отладку программы.
3. Изучить интерфейс программы.
4. Создать проект.

Методические указания

На официальном сайте *Unity3D* http://unity3d.com (рис. 1.1) можно бесплатно скачать установщик программы *Unity3D* (рис. 1.2): http://unity3d.com/ru/get-unity/update .

Встроенным установщиком производится загрузка файлов на жёсткий диск компьютера. По завершению установки, приложение попросит зарегистрироваться и выбрать тип лицензии (рис. 1.3). В учебных целях достаточно персональной лицензии, которая предоставляется бесплатно. Для выполнения и отладки работы необходимы программные продукты и библиотеки *Android Studio* и *JDK*.

Для работы с *java* приложениями, запуска скомпилированных и исполняемых файлов необходима библиотека *Java SE Development Kit* 9. Скачиваем бесплатно установщик с сайта http://www.oracle.com/technetwork/java/javase/downloads/2133151 и устанавливаем (рис. 1.4).

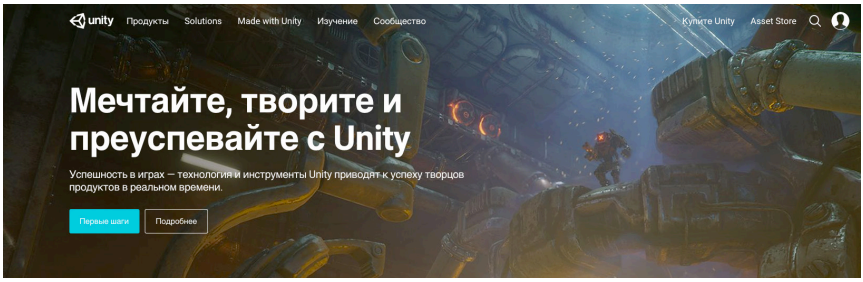


Рис. 1.1. Главная страница сайта unity3d.com

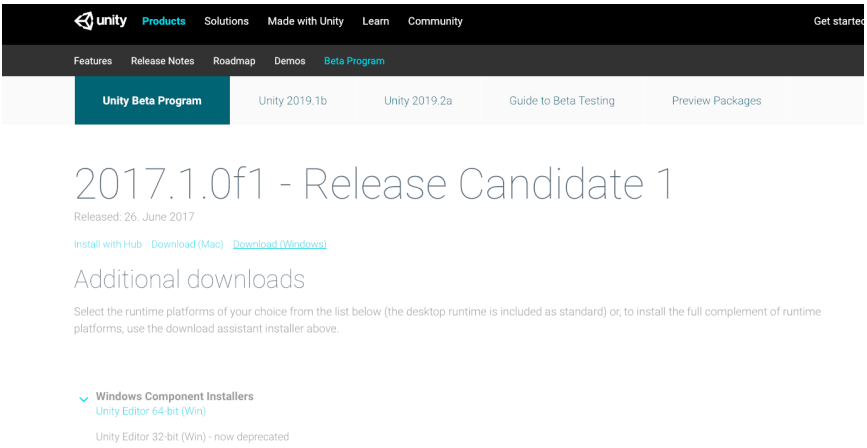


Рис. 1.2. Страница с загрузкой установщика

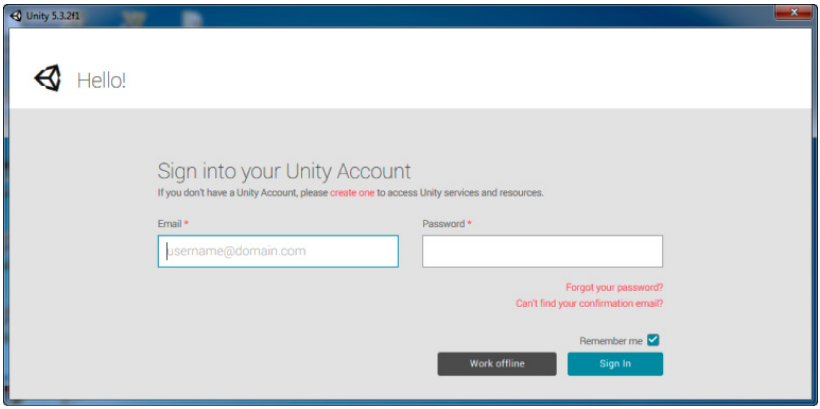


Рис. 1.3. Окно с вводом логина и пароля Unity

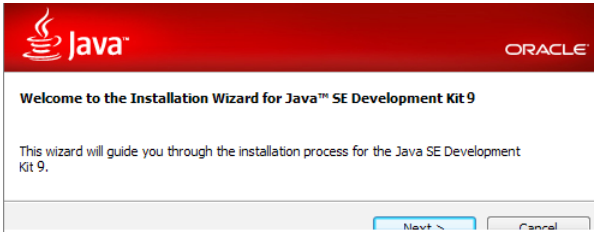


Рис. 1.4. Установка Java SE Development Kit

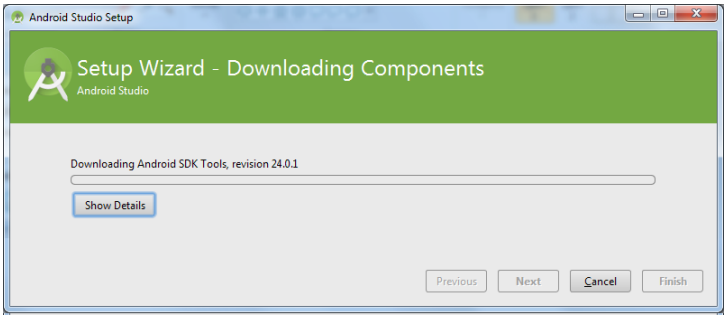


Рис. 1.5. Установка Android Studio

При создании приложений для пакета *Samsung S6* необходима среда разработки *Android Studio* 162.406. Скачиваем бесплатно с сайта https://developer.android.com/studio/index.html и устанавливаем (рис. 1.5).

Следующим этапом необходимо правильно подключить *SDK* и *JDK*, для чего задаем корректные настройки *Unity* на вкладке *External Tools* (рис. 1.6).

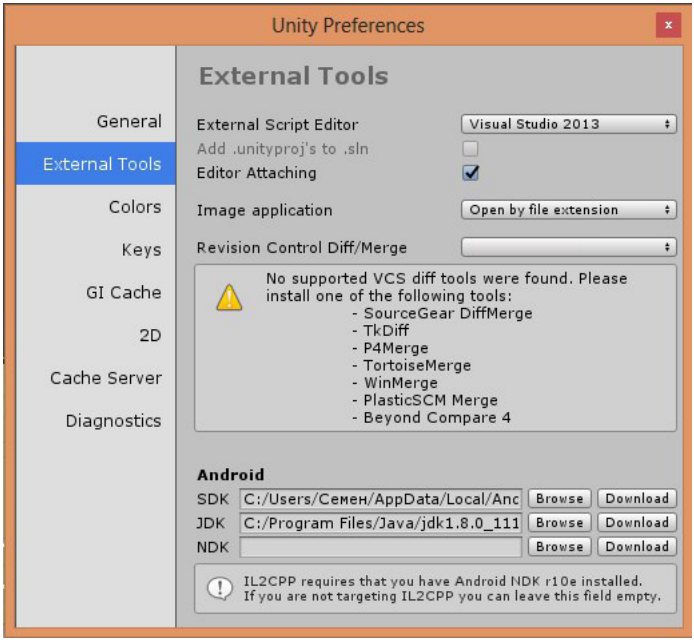


Рис. 1.6. Вкладка External Tools настроек Unity

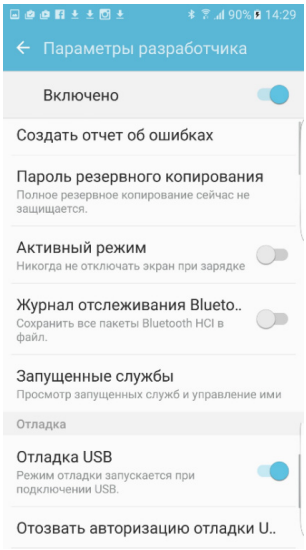


Рис. 1.7. Параметры разработчика

Для завершения подготовки программной среды, загрузки и запуска приложения на мобильном устройстве необходимо включить настройки разработчика (рис. 1.7).

Создание проекта

При запуске *Unity 3D*, отображается стартовое окно программы. В случае, если проектов нет в установленном дистрибутиве, то можно либо загрузить произвольный проект (из сети или магазина *Asset Store*), а затем открыть его через кнопку «*Open*» (указав путь к нему) или создать новый проект с помощью кнопки «*New*», которая расположена в правой верхней части окна (рис. 1.8).

В стартовом окне отображаются названия всех найденных проектов, которые есть на компьютере, а также показывается путь к файлам проекта и версия программы, в которой был выполнен проект. При создании проекта, необходимо указать название проекта и путь к нему (рис. 1.9).

Нажав на кнопку «*Asset packages*» (Пакеты ресурсов), можно выбрать необходимые пакеты ресурсов для нового проекта, а также ресурсы, которые были скачены из магазина *Asset Store* (рис. 1.10).

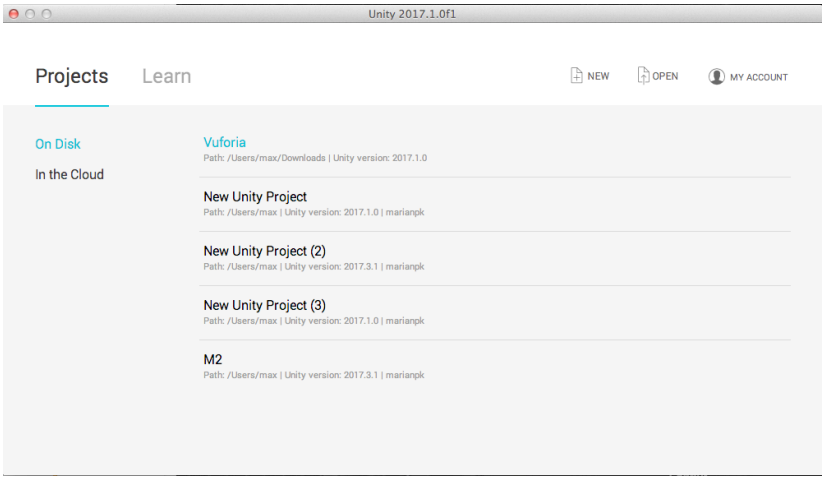


Рис. 1.8. Программа Unity3D при первом запуске, если внутри есть проекты

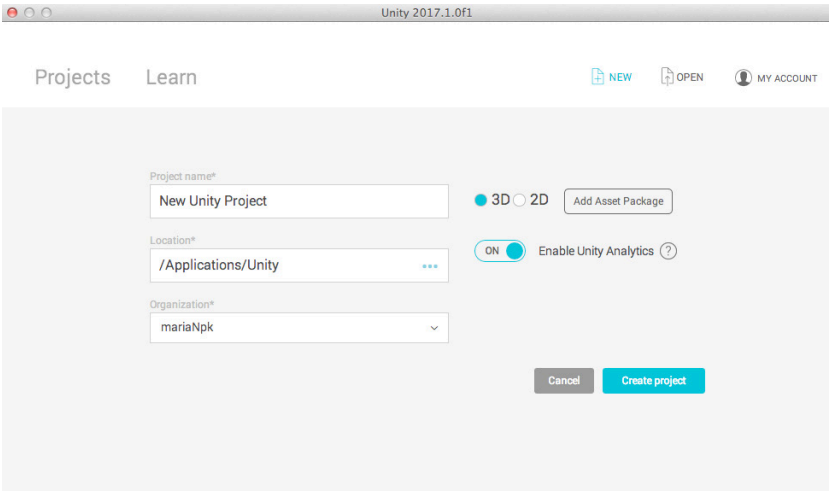


Рис. 1.9. Создание нового проекта

Краткое описание стандартных ресурсов, которые включены в *Unity*:

– *2D* – материалы для создания *2D*-проекта (анимация, префабы, скрипты и спрайты);

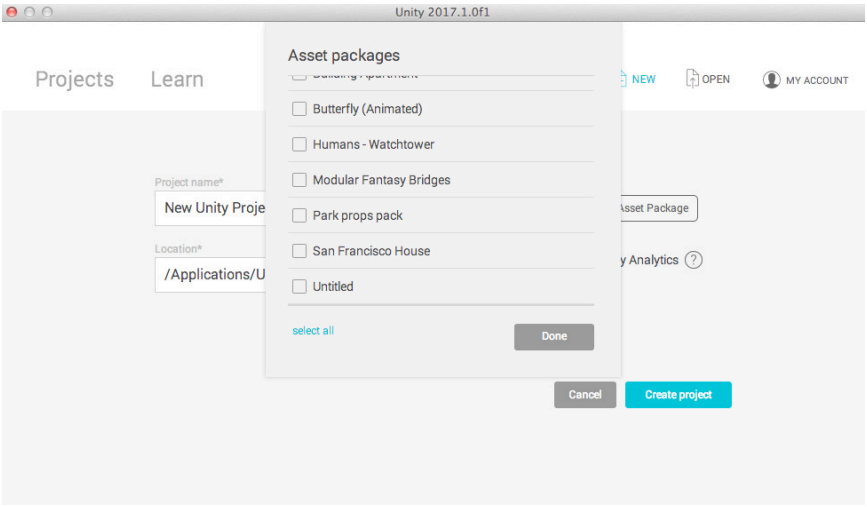


Рис. 1.10. Выбор необходимых ресурсов

– *Cameras* (Камеры) – префабы и скрипты дополнительных камер;

– *Characters* (Персонажи) – персонаж с видом от первого лица, персонаж с видом от третьего лица и персонаж-шар с видом от третьего лица;

– *CrossPlatformInput* (Кроссплатформенный ввод) – префабы, скрипты и спрайты для портирования на разные платформы;

– *Effects* (Эффекты) – эффекты со светом, фильтры для камер и другое;

– *Environment* (Окружающая среда) – деревья, вода, текстуры для ландшафта (*Terrain*);

– *ParticleSystems* (Система частиц) – создание пыли, огня, дыма, пара.

– *Prototyping* (Прототипы) —различные 3D-объекты для проектов;

– *Utility* (Утилиты) – различные скрипты связанные с контроллерами, камерами и др.;

– *Vehicles* (Транспортные средства) – модели машины и самолёта, а также скрипты, материалы, звуки и анимация, связанная с ними;

Другие пакеты в списке – это дополнительные файлы, установленные пользователем.

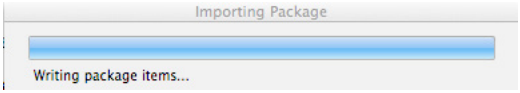


Рис. 1.11. Индикация загрузки пакетов ресурсов

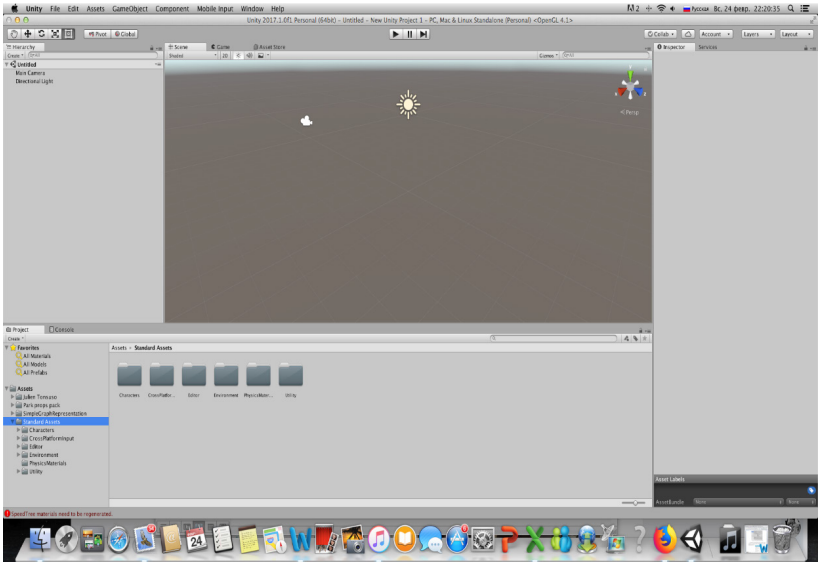


Рис. 1.12. Основное окно программы

Затем, в зависимости от количества выбранных пакетов ресурсов, необходимо подождать завершения загрузки (рис. 1.11) основного окна программы (рис. 1.12).

Примечание: количество подключаемых ресурсов влияет на размер файлов проекта, а также на скорость загрузки основного окна программы (происходит распаковка и компиляция данных).

Знакомство с интерфейсом программы

В верхней части окна приложения располагается главное меню программы (рис. 1.13).

Рис. 1.13. Главное меню

Меню включает в себя пункты:

– *File* (Файл), содержит подпункты, отвечающие за создание, открытие и сохранение проектов и сцен, а также настройки для компиляции готового приложения;

– *Edit* (Редактирование), отвечает за вырезание/вставку выделенных объектов, переключение режимов *Play*/*Pause*/*Step*, настройки сцены, параметры эмуляции и др.;

– *Assets* (*ресурсы*\активы), включает инструменты для импорта и экспорта наборов, а также позволяет создать новые материалы, текстуры, скрипты и др.;

– *GameObject* (Игровые объекты), позволяет создавать такие физические объекты как системы частиц, камеры, источники света, простейшие трехмерные формы и др.;

– *Component* (Компоненты), отвечает за настройку характеристик объектов: физика, сглаживание, отображение, аудиоэффекты и др.;

– *Window* (Окно), позволяет настроить внешний вид среды, расположение окон и позволяет получить доступ к каждому из них по отдельности;

– *Help* (*Помощь*), содержит ссылки на форумы и руководства по *Unity*, ответы на часто задаваемые вопросы и т. д.

Познакомимся с интерфейсом среды *Unity3D*. Основное место на экране занимает окно вида сцены, в котором отображается модель сцены со всеми *2D* и *3D* объектами (рис. 1.14).

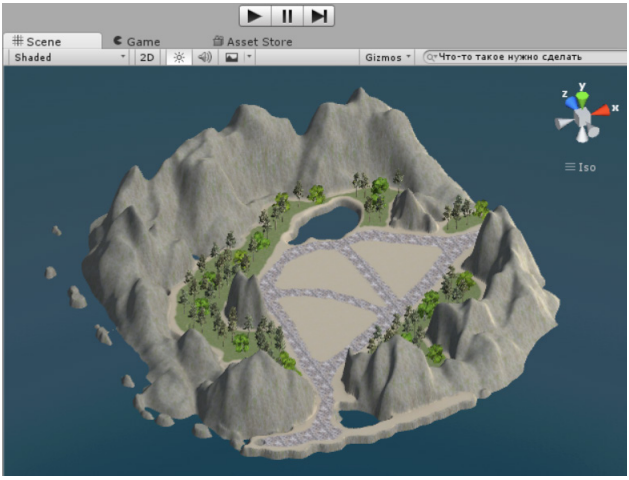


Рис. 1.14. Окно вида сцены

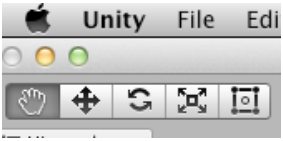


Рис. 1.15. Кнопки управления перемещением в сцене и модификации объекта

Ее можно осмотреть, нажав на значок с изображением руки в верхнем левом углу окна сцены (рис. 1.15):

После нажатия на значок с изображением руки появляется возможность осмотреть сцену: зажатая левая кнопка мыши позволяет передвинуть положение камеры; зажатая правая кнопка мыши – повернуть камеру; колесо мыши – приближение/отдаление камеры. Следующая кнопка в виде перекрещивающихся стрелок позволяет перемещать объект по сцене, кнопка с округлыми стрелками – вращать вокруг своей оси, последняя кнопка позволяет масштабировать.

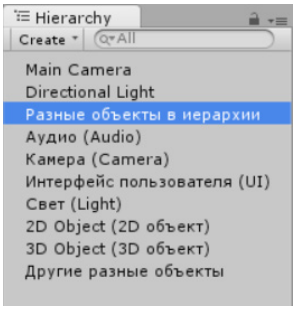


Рис. 1.16. Hierarchy (Иерархия)

На вкладке *Hierarchy* (Иерархия) перечислены все объекты, добавленные в сцену (объекты, персонажи, источники света и т. д.) (рис. 1.16).

В меню «*Project*» (Проект) перечислены все заготовки и созданные скрипты, текстуры, шрифты, сцены, входящие в текущий проект. Для поиска необходимых объектов сцены следует раскрыть соответствующий раздел проекта (рис. 1.17).

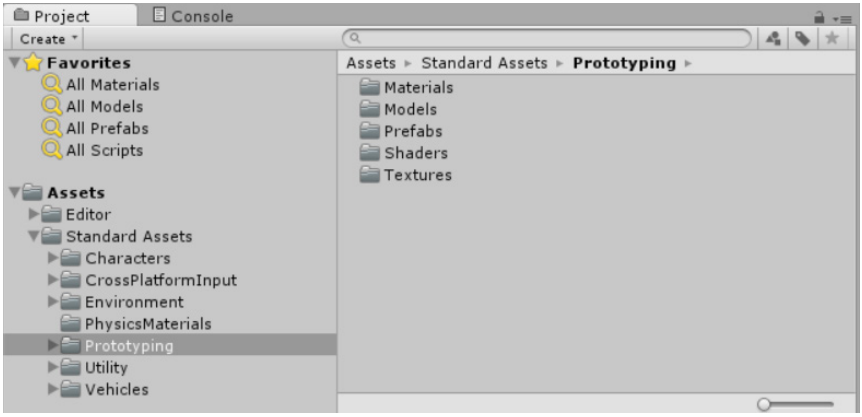


Рис. 1.17. Меню Проект

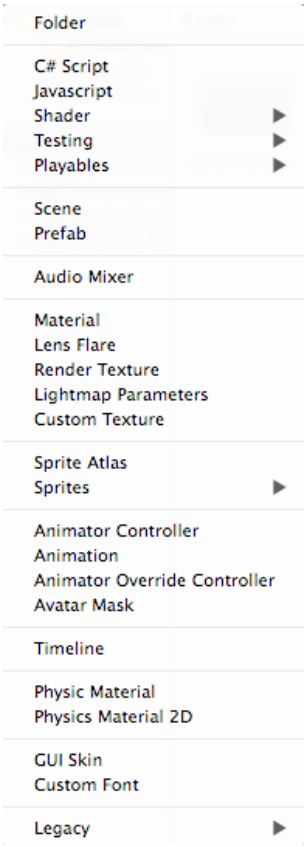


Рис. 1.18. Элементы меню Create

Рис. 1.19. Окно Inspector

Элементы меню «*Create*» (Создать) и окна «*Inspector*» (Инспектор) представлены соответственно на рис. 1.18 и 1.19.

Если из списка *Hierarchy* выбрать какой-либо объект, то в окне *Inspector* отобразится вся информация о данном объекте в сцене: положение по координатной сетке, наклоны, примененные скрипты, нанесенные текстуры, физическая модель и её настройки и др.

Контрольные вопросы

1. С какой целью для отладки программы *Unity3D* необходима среда разработки *Android Studio*?

2. С какой целью для отладки программы *Unity3D* необходима библиотека *Java SE Development Kit*?

3. Что представляют собой стандартные ресурсы: *Characters, Effects, Environment, ParticleSystems*?

4. Какую функцию выполняет *Inspector*?

ЛАБОРАТОРНАЯ РАБОТА № 2
СОЗДАНИЕ ПРОСТОЙ 3D-СЦЕНЫ

Цель работы: изучение инструментов создания ландшафта, постановки общего освещения, наложения текстур, добавления камеры и объектов.

Порядок выполнения работы

1. Создание поверхности ландшафта.
2. Создание общего освещения сцены.
3. Наложение текстур на поверхность ландшафта.
4. Размещение на ландшафте *3D* объектов.
5. Размещение персонажа на сцене.
6. Формирование отчета.

Методические указания

Для создания нового проекта в меню интерфейса откроем *File – New Project*. В открывшемся окне указываем путь к проекту («*Location*») для сохранения проекта и его название («*Project name*»). Нажимаем «*Create project*» (рис. 2.1).

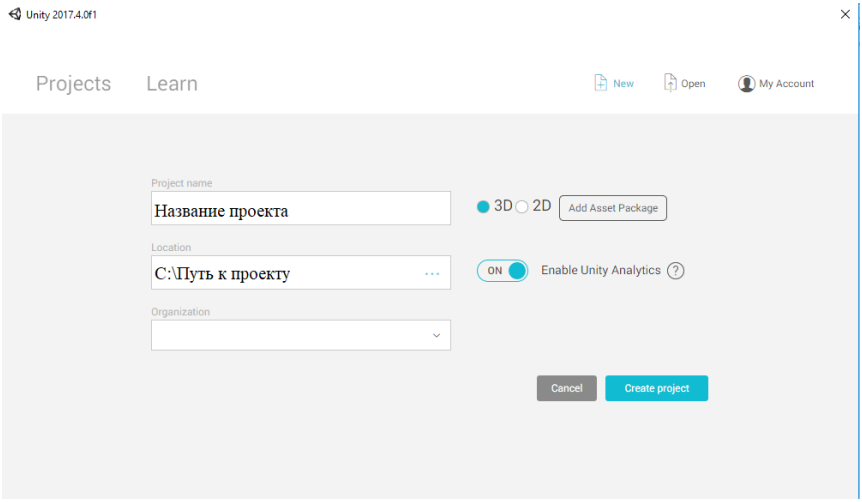


Рис. 2.1. Выбор названия проекта и пути сохранения

После создание нового проекта необходимо подгрузить пакеты ресурсов. (рис. 2.2). Для этого заходим во вкладку «*Asset*», нажимаем «*Import Package*» и выбираем 4 следующих пакета ресурсов:

1. *Characters* (Персонажи) – персонаж с видом от первого лица, персонаж с видом от третьего лица и персонаж-шар с видом от третьего лица;

2. *Environment* (Окружающая среда) – деревья, вода, текстуры для ландшафта;

3. *Prototyping* (Прототипы) – различные 3D-объекты для проектов;

4. *Vehicles* (Транспортные средства) – модели машины и самолёта, а также скрипты, материалы, звуки и анимация, связанная с ними;

После загрузки выбранных ресурсов открывается окно проекта, в котором будем создавать *3D*-сцену (рис. 2.3).

Для начала создадим стандартную поверхность, при помощи объекта «*Terrain*» (*GameObject – 3D Object – Terrain* или *Hierarchy – 3D Object – Terrain*) (рис. 2.4).

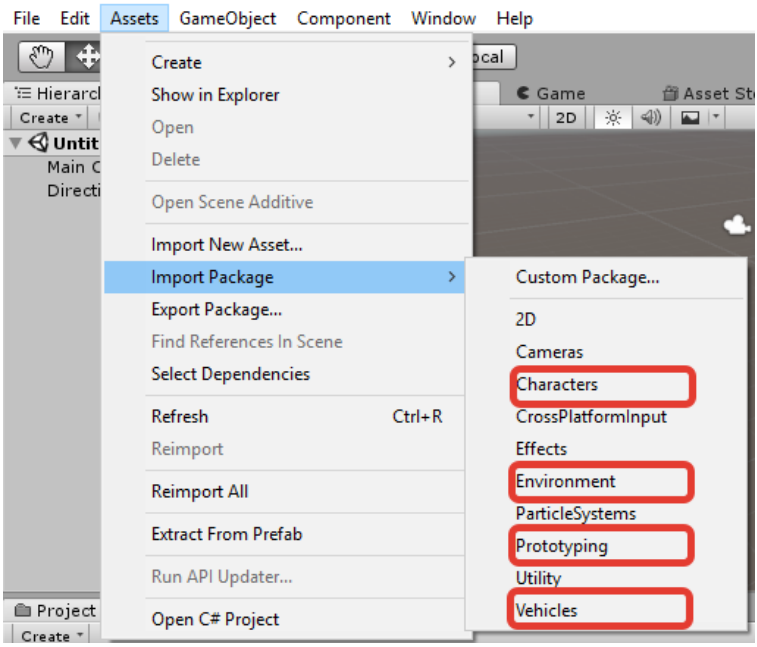


Рис. 2.2. Выбор необходимых ресурсов из списка Asset packages

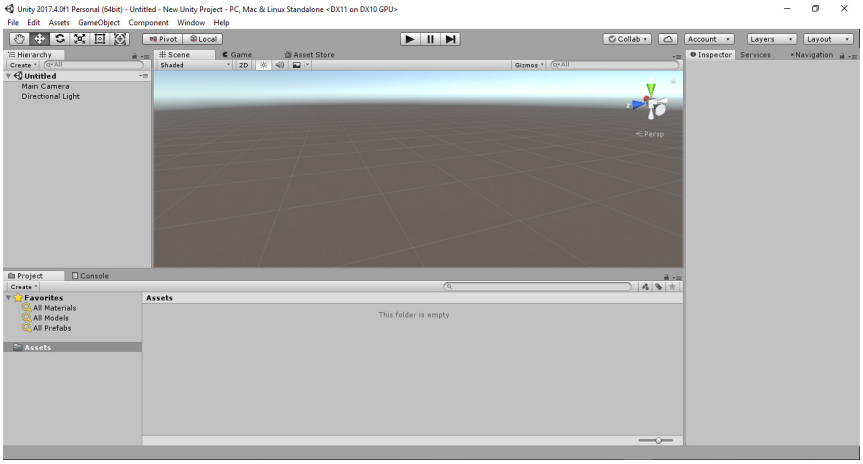


Рис. 2.3. Окно 3D-сцены

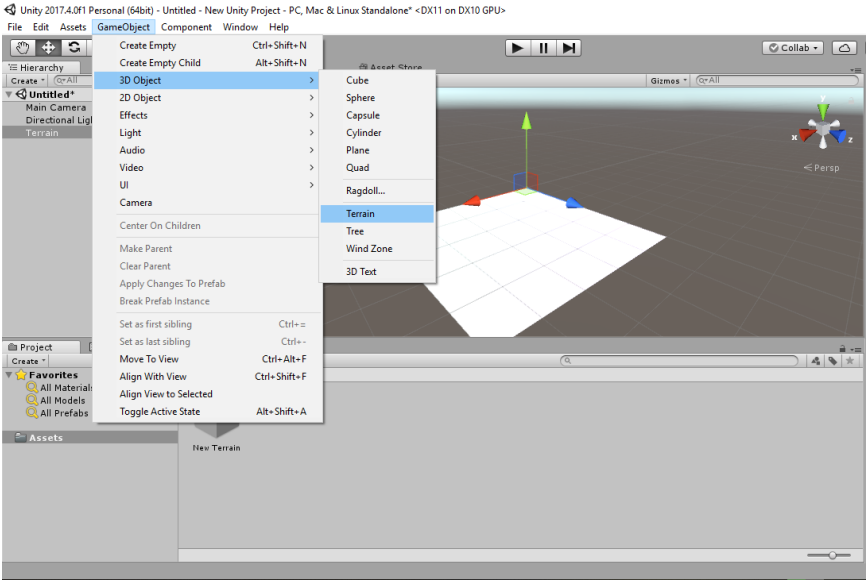


Рис. 2.4. Создание стандартного объекта «Terrain»

Создадим источник освещения, для этого открываем *GameObject – Light – Directional Light или Hierarchy – Light – Directional Light* (оба варианта показаны на рис. 2.5).

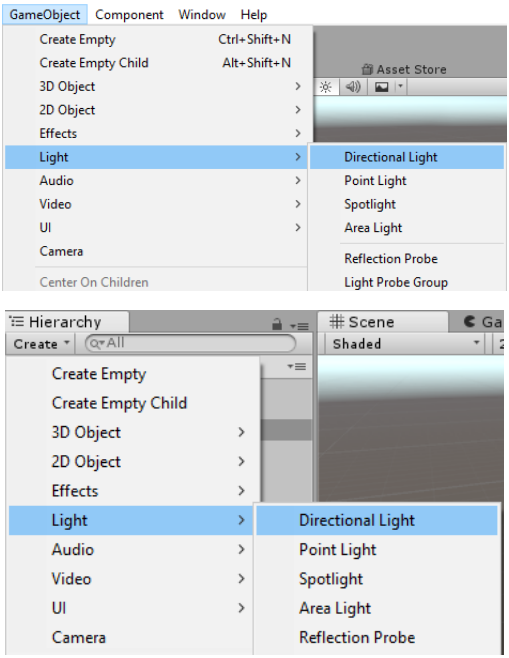
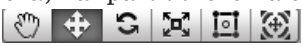


Рис. 2.5. Directional Light через меню GameObject и Hierarch

Появился источник направленного освещения. Устанавливаем и размещаем полученный источник света, направляя его на объекты *3D*-сцены с помощью инструментов  в верхней левой части окна вида сцены. Затем приступаем к настройке источника света в инспекторе (рис. 2.6).

В основных настройках можно поменять цвет исходящего света («*Color*»), задать интенсивность («*Intensity*»), выбрать тип теней («*Shadow Type*»), а также создать эффект солнечного блика при направлении камеры на источник света («*Flare*»).

Приступим к редактированию созданной поверхности «*Terrain*». Для создания неровностей в окне «*Hierarchy*» выбираем «*Terrain*». В инспекторе находим подпункт «*Terrain*» и выбираем «*Raise / Lower Terrain*» (рис. 2.7).

Теперь, щелкая левой клавишей мыши по поверхности ландшафта, создаем неровности: горы, холмы, впадины. Для того чтобы изменить размер кисти используем настройку «*Brush Size*», также можно изменить тип кисти в поле «*Brushes*». За интенсивность отвечает ползунок «*Opacity*».

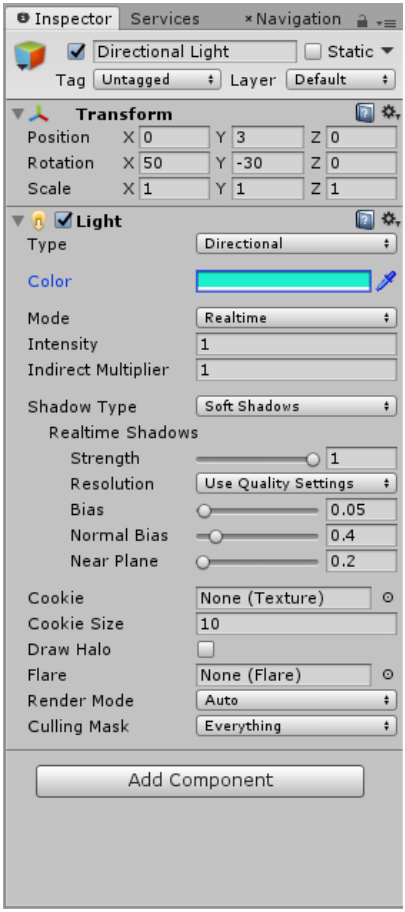


Рис. 2.6. Настройки источника освещения

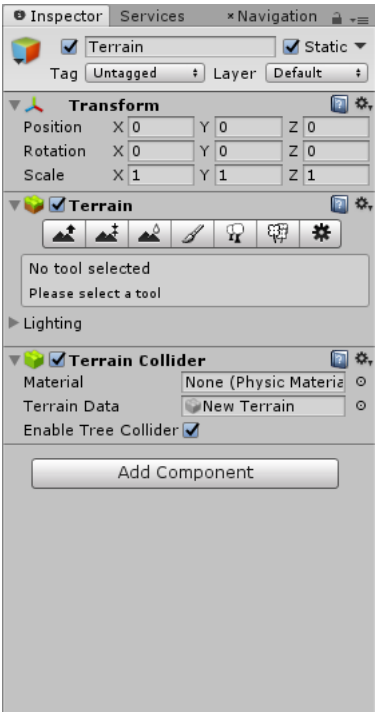


Рис. 2.7. Редактор ландшафта

Для удаления неровности необходимо зажать на клавиатуре клавишу *Shift* и с помощью нажатий левой клавиши мыши убирать ненужные неровности.

Пользуясь перечисленными выше инструментами, можно создать, например, ландшафт, представленный на рис. 2.8.

После создания рельефа наложим на ландшафт текстуру травы. В том же инспекторе находим изображение кисточки, она же вкладка «*Paint texture*» (рис. 2.9).

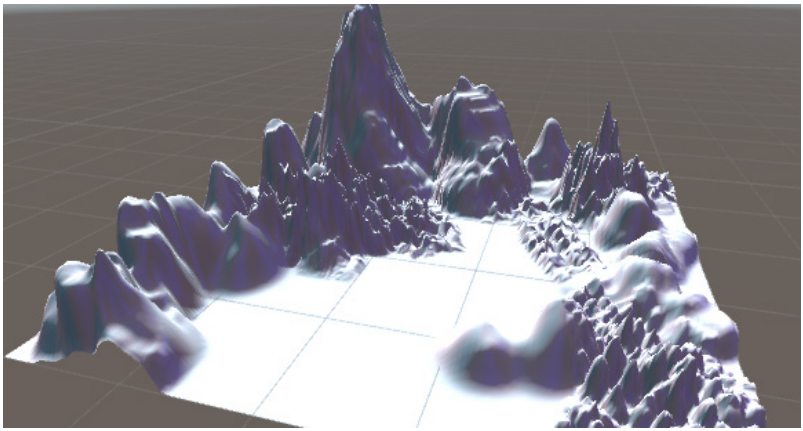


Рис. 2.8. Пример ландшафта

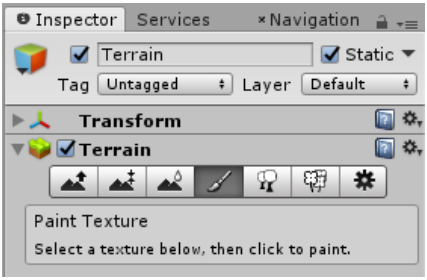


Рис. 2.9. Инструмент для наложения текстур на поверхность

Далее нажимаем кнопку «*Edit Textures*». Появляется окно добавления текстур «*Add Terrain Texture*» (рис. 2.10), где, нажав на кнопку «*Select*», можно выбрать несколько текстур.

В открывшемся окне «*Select Texrure2D*» (рис. 2.11) благодаря ранее выбранным ресурсам (*Assets*), есть некоторое количество текстур.

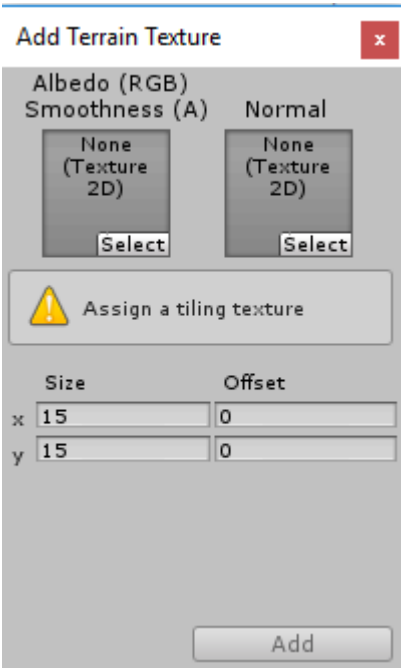


Рис. 2.10 Окно добавления текстур

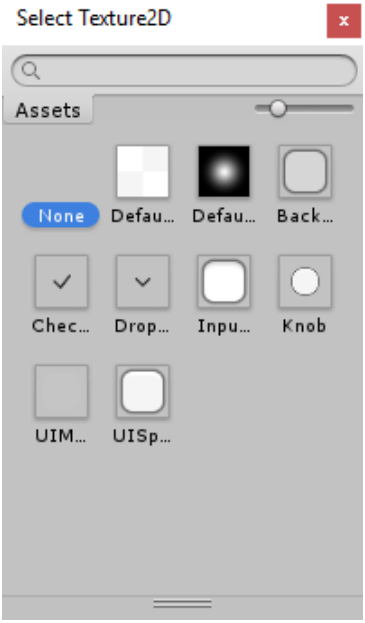


Рис. 2.11. Окно выбора текстур

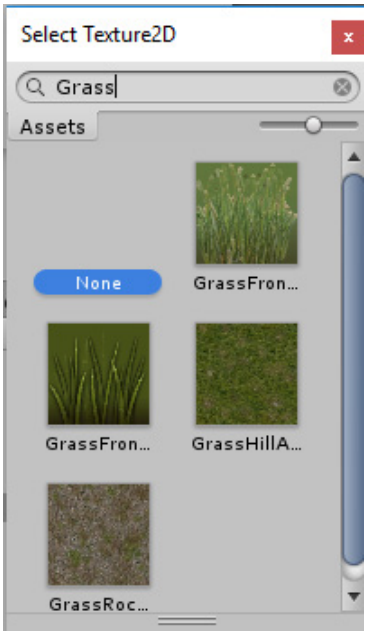


Рис. 2.12. Текстуры травы

Например, написав в поле поиска слово «*Grass*» (русск. трава), увидим несколько текстур (рис. 2.12). Выбрав подходящую, нажимаем «*Add*». Вся поверхность ландшафта заполняется текстурой травы (рис. 2.13).

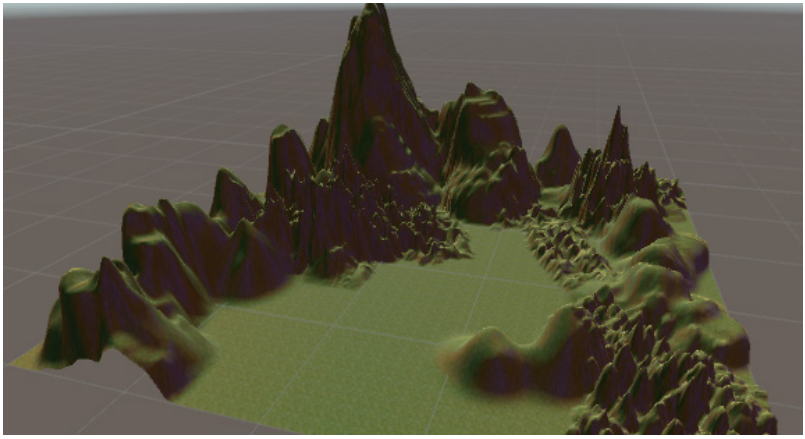


Рис. 2.13. Текстура травы на поверхности ландшафта

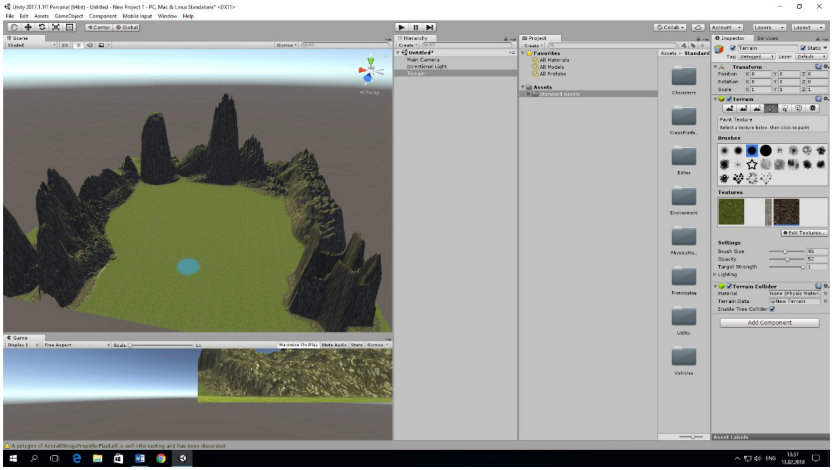


Рис. 2.14. Рельеф с различными текстурами

Нажимая на кнопку «*Edit Textures*» добавляем ещё несколько текстур и применяем их к различным частям ландшафта для создания большей реалистичности (рис. 2.14).

Для добавления деревьев, через инспектор объекта «*Terrain*» выбираем вкладу «*Place Trees*» (рис. 2.15).

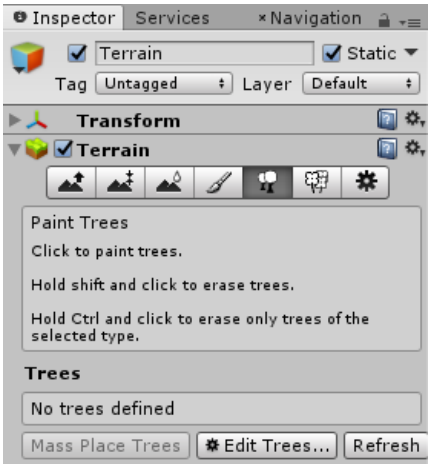


Рис. 2.15. Инструмент для добавления деревьев на поверхность

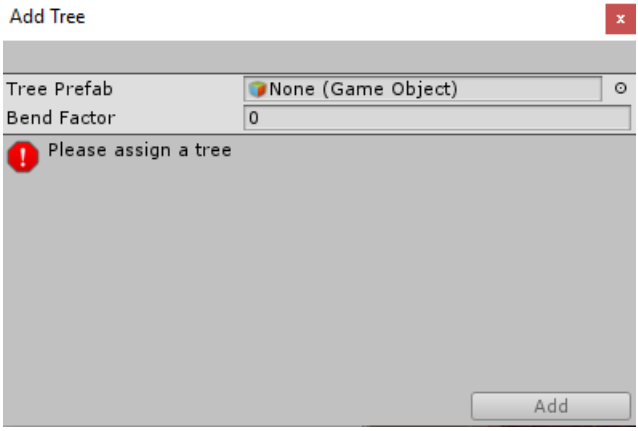


Рис. 2.16. Добавление префаба дерева

Нажимаем кнопку «*Edit Trees...*», далее «*Add trees*» и выбираем дерево. Для этого нужно нажать на кружок с точкой справа (рис. 2.16) напротив префаба дерева («*Tree Prefab*»).

Появится окно «*Select GameObject*» (рис. 2.17), в котором можно выбрать подходящую модель дерева.

Выбрав нужные модели деревьев, расставляем их на ландшафте (рис. 2.18). Стоит учитывать, что сильно деформированный ландшафт может создать эффект того, что деревья «повиснут в воздухе», если их «посадить» на изгибах и резких перепадах высот.

В настройках можно выбирать высоту («*Tree Height*») и толщину дерева («*Tree Width*»), также с помощью «*Brush Size*» можно выбрать количество добавляемых деревьев и их плотность изменением параметра «*Tree Density*» (рис. 2.19).

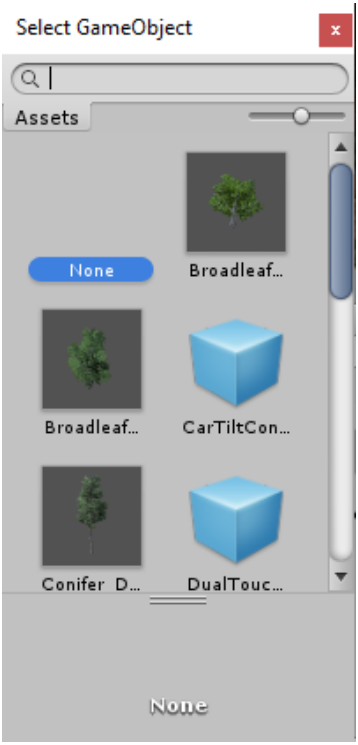


Рис. 2.17. Выбор префаба дерева

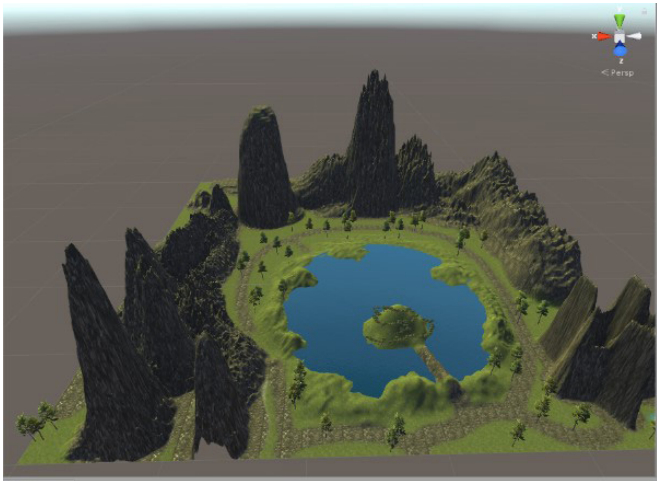


Рис. 2.18. Ландшафт с различными моделями деревьев

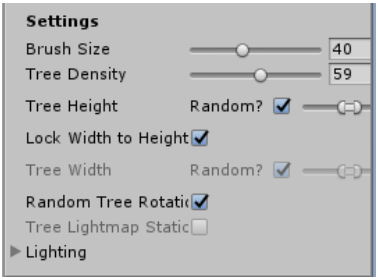


Рис. 2.19. Настройки для деревьев

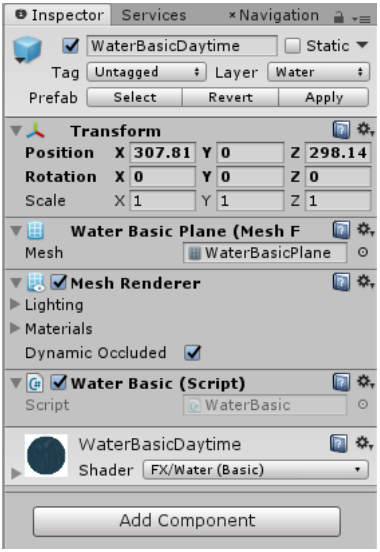


Рис. 2.20. Настройки для воды

Добавим немного воды. Находим префаб «*WaterBasicDaytime*» (*Project – Assets – Standard Assets – Environment – Water (Basic) – Prefabs*) и переносим его на сцену, затем изменяем размер объекта (*Inspector – Transform – Scale*) на необходимый для покрытия нужного участка поверхности ландшафта (рис. 2.20).

Следующая задача – это добавление в сцену действующего персонажа («*First Person Controller*»). Для этого перенесем в сцену префаб

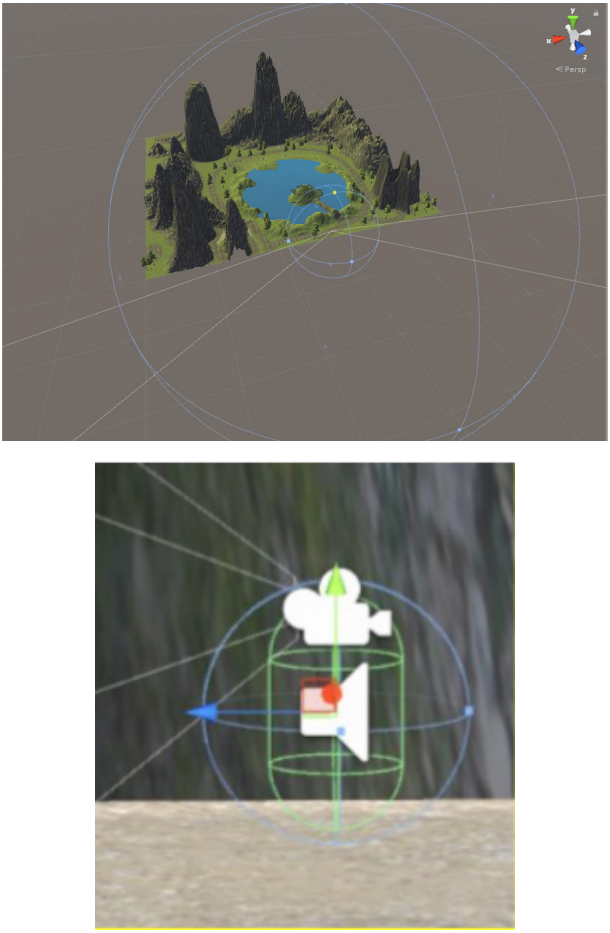


Рис. 2.21. Размещение контроллера персонажа

«*FPSController*» (*Project – Assets – Standart Assets – Characters – FirstPersonController – Prefabs*) (рис. 2.21). Учитываем расположение персонажа так, чтобы он находился на сцене, а не вне её пределов, а также, чтобы коллайдер (*Collider*) персонажа (зелёная капсула) был немного выше поверхности ландшафта.

Запускаем режим игры с помощью клавиши «*Play*», которая расположена в верхней части окна вида сцены. Теперь мы можем видеть результаты проделанной работы и перемещаться по сцене, управляя персонажем от первого лица (рис. 2.22).

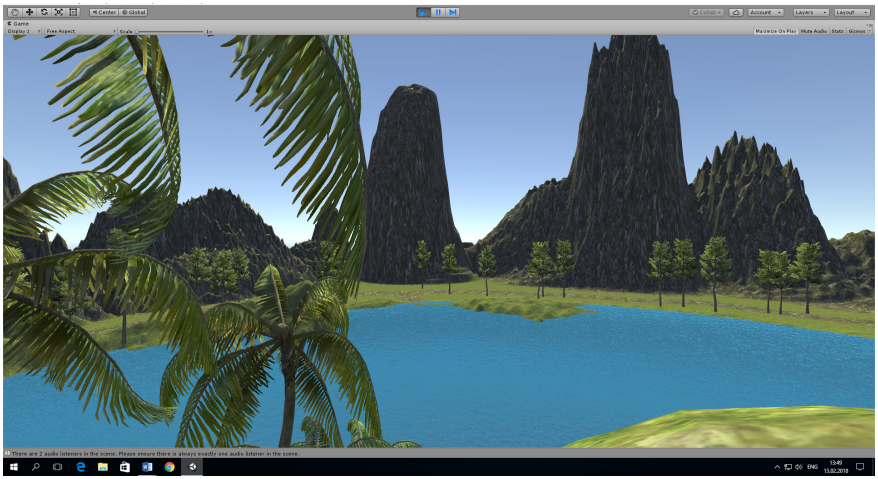


Рис. 2.22. Вид сцены в окне Game(игра) от первого лица

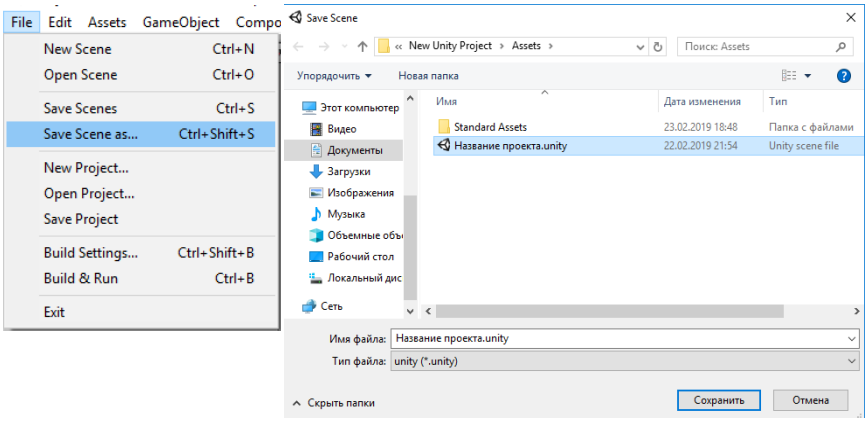


Рис. 2.23. Сохранение проекта

Наконец необходимо сохранить получившуюся сцену (*File – Save Scene As...*), затем выбираем место и название, под которым будет сохранена сцена (рис. 2.23).

Контрольные вопросы

1. С чего начинается создание ландшафта?
2. Возможно ли удалить созданную неровность с поверхности (*Terrain*)?
3. Как изменить размеры и количество деревьев?
4. Как происходит постановка общего освещения?
5. Как происходит наложение текстур?
6. Как добавляются объекты?

Лабораторная работа № 3
ИМПОРТ МОДЕЛЕЙ В ПРИЛОЖЕНИЕ, ПОСТАНОВКА ЛОКАЛЬНОГО ОСВЕЩЕНИЯ

Цель работы: ознакомление с импортом моделей в проект *Unity3D* на примере моделей *3ds Max*, а также с постановкой различных видов локального освещения.

Порядок выполнения работы

1. Импорт моделей в проект *Unity3D*.
2. Постановка различных видов локального освещения.
3. Оформление отчета.

Методические указания

Для импорта моделей следует открыть сцену, созданную в лабораторной работе № 2. Для этого нажимаем (*File -> Open Scene*) и выбираем сохраненную сцену.

После того, как загрузилась сцена, приступим к импорту моделей. *Unity* поддерживает импорт форматов *.3ds, .max, .obj, .fbx, .dae, .ma, и .mb* для *3D*-моделей. Возьмём готовую модель простого дома в формате .fbx (рис. 3.1)

Перенесём файл с моделью простого дома формата. *fbx* в папку, где находится проект (*Project*), а затем перетащим модель на сцену, если нужно изменяем размер и положение модели (рис. 3.2). Возможно, что модель на сцене отобразится без текстур, но они присутствуют в папке с импортированной моделью. Тогда файл с текстурами следует перенести на модель или можно применить какие-то свои текстуры путем перетаскивания их на нужный объект.

Чтобы персонаж не проходил сквозь объект и не проваливался в него, необходимо сделать для модели коллайдер (*Collider*). Коллайдер позволяет обрабатывать столкновение объекта с другим объектом, например, в игре персонаж не должен проходить сквозь стену, а должен столкнуться с ней.

Коллайдеры добавляются через меню *Component -> Physics* или *Add Component -> Physics* в Инспекторе (рис. 3.3).

Существует несколько видов коллайдеров:

– *Box Collider* – в форме параллелепипеда;

– *Sphere Collider* – в форме сферы;

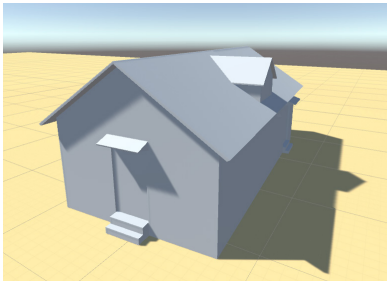


Рис. 3.1. Модель в формате. fbx

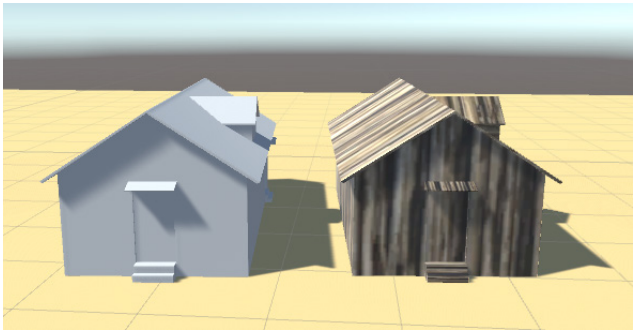


Рис. 3.2. Модель домика в сцене без текстур (слева) и с текстурами (справа)

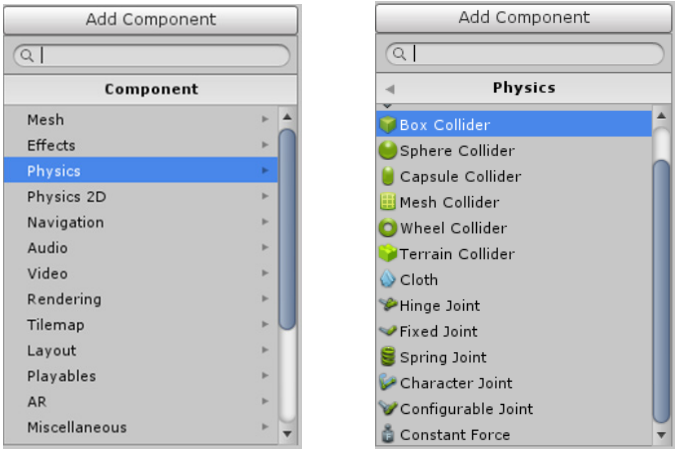


Рис. 3.3. Добавление коллайдера (слева) и выбор коллайдера(справа)

– *Capsule Collider* – в форме капсулы;

– *Mesh Collider* – автоматически создает коллайдер по форме полигональной сетки объекта;

– *Wheel Collider* – коллайдер колеса автомобиля.

Различные типы коллайдеров применяются для разных типов объектов, например, *Sphere Collider* применятся для моделей в форме шара. Для того, чтобы добавить коллайдер, который будет повторять форму объекта, в разделе *Hierarchy* выделяем нужную нам модель и применим к ней коллайдер сетки (*Mesh Collider*) *Component -> Physics -> Mesh Collider*. Если для передачи формы и модели столкновений объекта достаточно обычных коллайдеров, то рекомендуется использовать их, так как коллайдер сетки очень требователен к ресурсам системы и его применение отрицательно скажется на производительности.

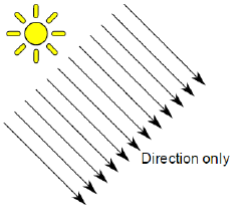
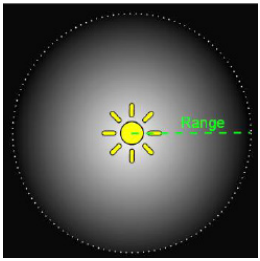
Освещение

Для реалистичности сцены создадим источники локального освещения.

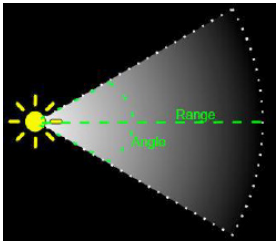
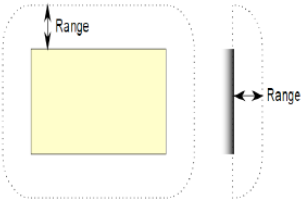
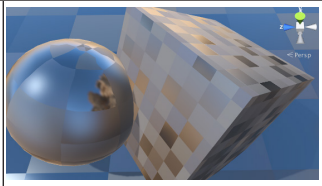
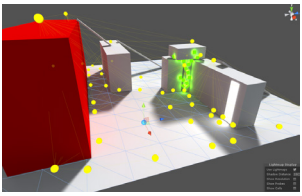
Основные типы источников приведены в табл. 3.1.

Таблица 3.1

Основные типы источников света.

Источник	Описание	Изображение
Directional Lights (Направленный свет)	Направленный свет не имеет определенной позиции источника и поэтому может быть размещен в любом месте сцены. Самый простой, имитирует солнечный свет. Представляет из себя бесконечное множество параллельных друг другу лучей	
Point Light (Точечный источник)	Источник испускает лучи света во все стороны. Точечный источник света, то есть лучи расходятся во все стороны из одной точки. Хорошим примером такого источника света будет обычная лампочка	

Продолжение табл. 3.1

Источник	Описание	Изображение
Spotlight (Прожектор)	Пучок света получается в форме конуса, расширяясь с увеличением расстояния от источника света. Примером служат фары автомобиля	
Area Light (Подсветка области)	Область света определяется прямоугольником в пространстве. Свет излучается во всех направлениях, но только с одной стороны прямоугольника. Источник света, имеющий площадь. Представьте себе прямоугольную панель, из которой исходит свет, это и будет area light. Такие источники света чаще всего используются в офисах, торговых центрах и других нежилых помещениях, где надо освещать большие пространства	
Reflection Probe (Отражение)	Источник света, который имитирует отражение внешнего источника света, подсвечивая область, подобно белому листу или фольге. При этом объект принимает и цвет источника света	
Light Probe Group (Зонды освещения)	Хоть лайтмаппинг и вносит существенный вклад в реализм сцены, он имеет один недостаток: не статичные объекты в сцене отрисовываются менее реалистично и в результате могут смотреться неестественно. Невозможно просчитать карты теней в реальном времени для движущихся объектов, но можно получить похожий эффект с помощью *light probes* (зондов освещения).	

Окончание табл. 3.1

Источник	Описание	Изображение
	Их идея в том, чтобы освещение выбиралось в стратегически важных (для освещения) точках сцены, обозначенных с помощью размещения зондов. После этого, свет в любой точке может быть аппроксимирован с помощью интерполяции между выборками, которые совершили ближайшие зонды. Интерполяция помогает избежать несоответствия освещения динамических объектов, и статичных объектов, к которым применены карты освещения, кроме того, интерполяция достаточно быстра, чтобы её можно было использовать во время игры.	

Выбираем источник света «*Spotlight*» в меню интерфейса *GameObject -> Light* (рис. 3.4) или *Hierarchy -> Create -> Light*.

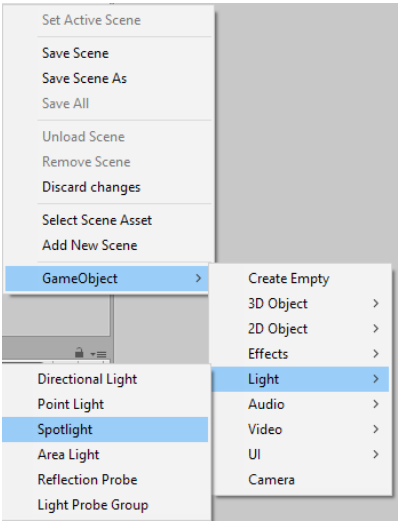


Рис. 3.4. Добавление источника света

Устанавливаем и поворачиваем полученный источник света в нужном направлении, например такой источник можно применять для создания уличного фонаря (рис. 3.5), применяя настройки света в инспекторе (рис. 3.6).

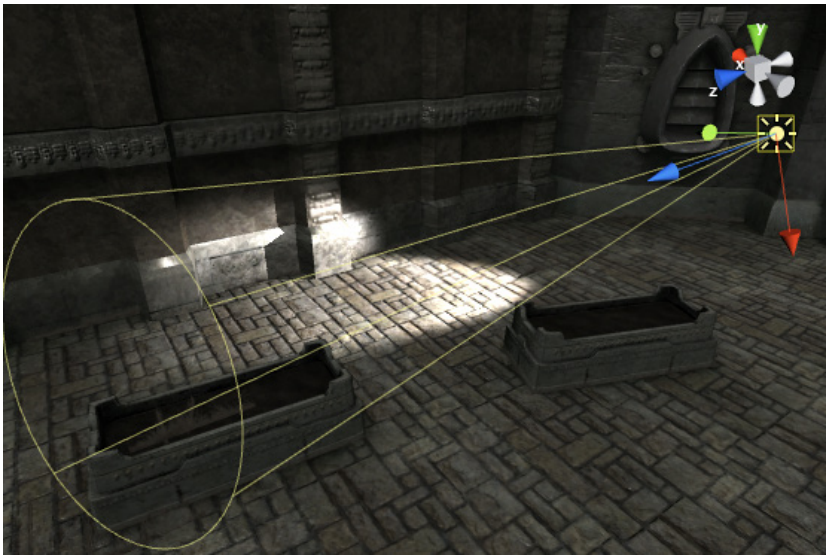


Рис. 3.5. Результат применения освещения Spotlight

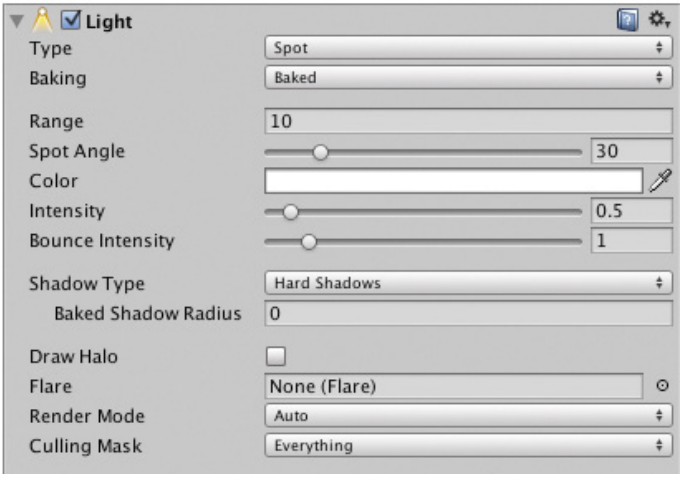


Рис. 3.6. Настройки Spotlight для создания уличного фонаря

Таким же способом создается источник освещения «*Point Light*», который удобно применять в качестве светящихся лампочек (рис. 3.7) и «*Area Light*» удобный для подсветки определённых областей на плоскостях (рис. 3.8). В основных настройках можно поменять цвет исходящего света (*Color*), задать интенсивность (*Intensity*), и размер «пятна» освещения («*Spot Angle*») для *SpotLight* и «*Range*» для *Point Light* (рис. 3.6).

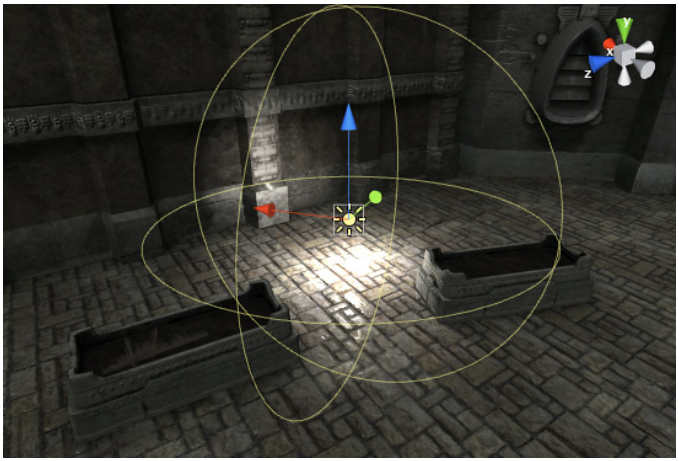


Рис. 3.7. Точечный источник света Point Light

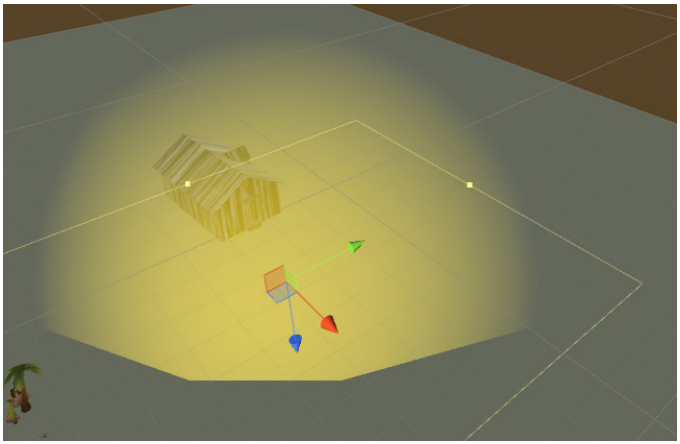


Рис. 3.8. Область с освещением Area Light

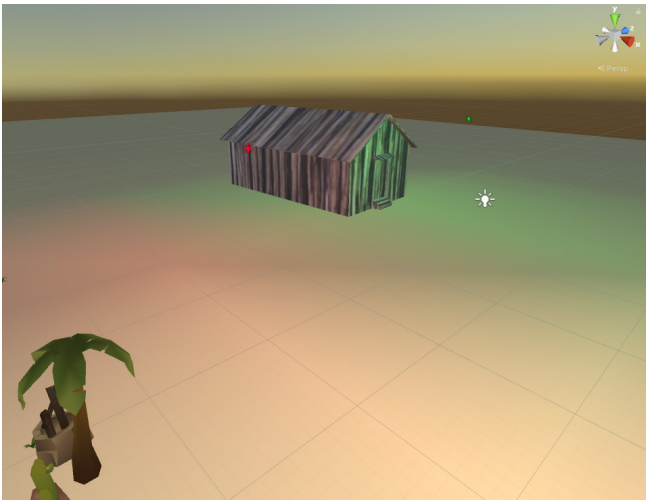


Рис. 3.9. Сцена с несколькими источниками света.

Расставив модели объектов и источники освещения, получаем готовую сцену, которая будет выглядеть, например, как на рис. 3.9.

Контрольные вопросы

1. Каким образом происходит импорт моделей в *Unity3D*?
2. Как обеспечить столкновение объектов друг с другом?
3. Какие имеются типы источников света?
4. Можно ли изменять интенсивность и цвет локального освещения?

Лабораторная работа № 4
РАЗРАБОТКА ПРОСТЕЙШЕГО ИНТЕРФЕЙСА ПРИЛОЖЕНИЯ

Цель работы: ознакомление с начальными принципами создания и использования скриптов в *Unity*, разработка собственного интерфейса приложения на основе приведенного примера.

Порядок выполнения лабораторной работы

1. Исходными данными являются данные, полученные в лабораторной работе № 3 «Импорт моделей в приложение, постановка локального освещения».
2. Изучение теоретического материала.
3. Создание скрипта.
4. Корректировка пунктов меню и проверка работы скрипта.
5. Оформление отчета.

Методические указания

Для выполнения данной лабораторной работы студенту необходимо на основе представленного ниже шаблона создать собственный интерфейс приложения.

Если студент знает иные способы создания пользовательского интерфейса, то он вправе использовать их. Но в таком случае необходимо подробно описать процесс создания интерфейса в отчёте.

Дальнейшее описание актуально для *Unity 2017.3.1f1 (64-bit)*.

Загрузить сцену, созданную в результате выполнения лабораторной работы №3. Далее необходимо создать *C#* скрипт. Для этого в папке проекта необходимо нажать правую кнопку мыши и выбрать «*C# Script*» в пункте меню «*Create*» (рис. 4.1).

Двойным щелчком по файлу открыть его для редактирования. Если при установке *Unity* была установлена *Visual Studio*, то лучше открыть файл скрипта через *sln* файл. Для этого в той же области щелкаем правой кнопки мыши и в самом низу выбираем пункт «*Open C# Project*». Откроется папка с проектом в которой необходимо открыть файл с расширением *sln*. В обозревателе решения находим созданный файл и открываем его (рис. 4.2).

В таком случае *Visual Studio* будет выдавать контекстные подсказки, что облегчит написание кода.

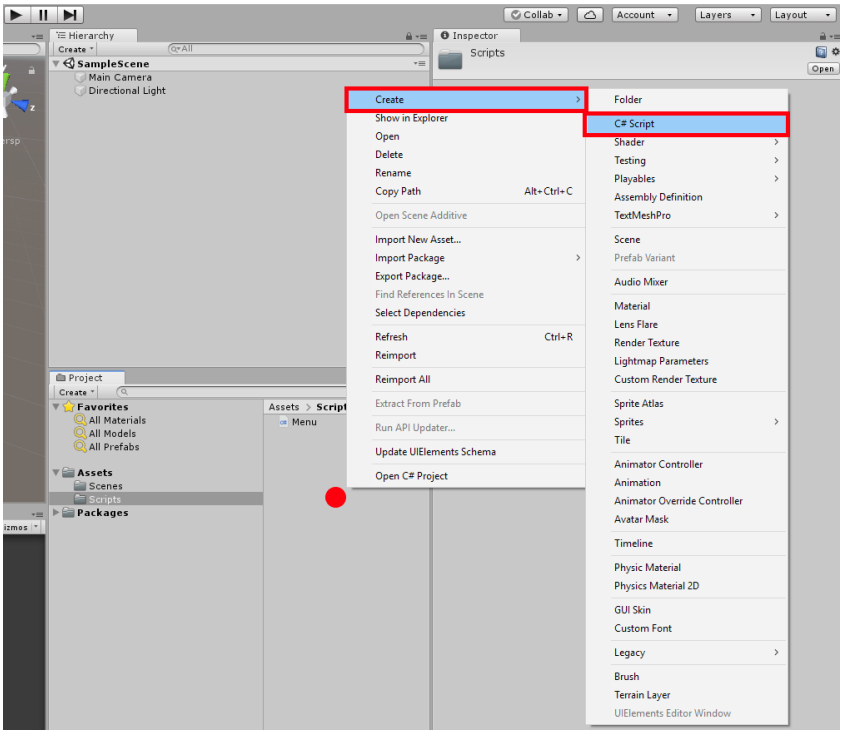


Рис. 4.1. Создание файла C# Script

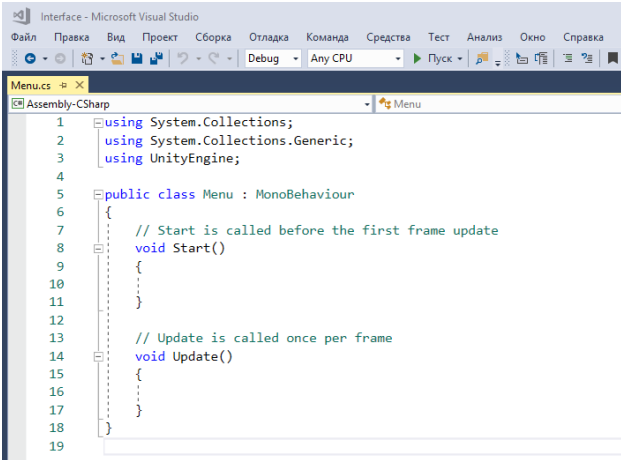


Рис. 4.2. Открытый проект в Visual Studio

Теперь заменим содержимое файла на следующий код:

```
// Стандартная библиотека движка Unity
using UnityEngine;
// Стандартная библиотека движка Unity
using System.Collections;
// Библиотека для работы со сценами
using UnityEngine.SceneManagement;

public class Menu : MonoBehaviour
{
    // Пауза в игре
    private bool _paused = false;
    // Окна меню
    private int _window = 100;

    void Update()
    {
        // При нажатии клавиши "Escape"
        if (Input.GetKeyUp(KeyCode.Escape))
        {
            //Если пауза отключена
            if (!_paused)
            {
                // Время в игре останавливается
                Time.timeScale = 0;
                // Пауза включается
                _paused = true;
                //Показывать окно меню
                _window = 0;
            }
            else
            {
                // Иначе игра продолжается
                Time.timeScale = 1;
                // Пауза выключается
                _paused = false;
                // Перестать показыватьменю
                _window = 100;
            }
        }
    }

    void OnGUI()
    {
        // Главное меню активировано при _window = 0
        if (_window == 0)
        {
            GUI.Box(new Rect(Screen.width / 2 - 100, Screen.
            height / 2 - 100, 200, 180), "Меню игры");
```

```
        if (GUI.Button(new Rect(Screen.width / 2 - 90,
        Screen.height / 2 - 80, 180, 30), "Продолжить игру"))
        {
              Time.timeScale = 1;
              _paused = false;
              _window = 100;
        }

        if (GUI.Button(new Rect(Screen.width / 2 - 90,
        Screen.height / 2 - 40, 180, 30), "Новая игра"))
        {
              Time.timeScale = 1;
              _paused = false;
              _window = 100;
              // Загрузка сцены №0
              SceneManager.LoadScene(0);
        }
        if (GUI.Button(new Rect(Screen.width / 2 - 90,
        Screen.height / 2 - 0, 180, 30), "Настройки"))
              // Переход к Настройкам, окну №1
              _window = 1;

        if (GUI.Button(new Rect(Screen.width / 2 - 90,
        Screen.height / 2 + 40, 180, 30), "Выход из игры"))
              // Выход из игры
              Application.Quit();
}

// Меню «Настройки»
if (_window == 1)
{
      GUI.Box(new Rect(Screen.width / 2 - 100, Screen.
      height / 2 - 100, 200, 180), "Настройки");

      if (GUI.Button(new Rect(Screen.width / 2 - 90,
      Screen.height / 2 - 80, 180, 30), "Видео"))
            // Открытие окна №2 (сделать своё)
            _window = 2;

      if (GUI.Button(new Rect(Screen.width / 2 - 90,
      Screen.height / 2 - 40, 180, 30), "Аудио"))
            // Открытие окна №3 (сделать своё)
            _window = 3;
      if (GUI.Button(new Rect(Screen.width / 2 - 90,
      Screen.height / 2 - 0, 180, 30), "Управление"))
            // Открытие окна №4 (сделать своё)
            _window = 4;
      if (GUI.Button(new Rect(Screen.width / 2 - 90,
```

```
                Screen.height / 2 + 40, 180, 30), "Назад") ||
                Input.GetKeyUp(KeyCode.Escape))
                    // Переход к окну №0
                    _window = 0;
            }
        }
}
```

После этого сохраняем изменения и подключаем скрипт к камере (*Main Camera*) в Инспекторе (*Inspector*) (рис. 4.3).

Запустим проект – при нажатии клавиши «*Escape*» на экране появится созданное Меню (рис. 4.4).

Выполним сборку приложения. Зайдя в настройки сборки (*File → Build Settings*), помечаем галочкой нужную сохранённую сцену (напротив неё есть номер, который понадобится, если будет много уровней/сцен в проекте), выбираем нужную операционную систему и нажимаем «*Build*» (рис. 4.5), выбираем путь и название для со-

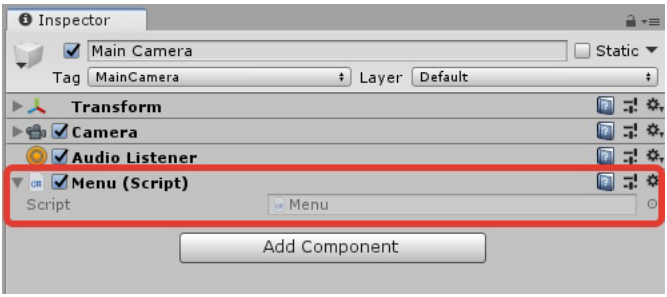


Рис. 4.3. Inspector Main Camera

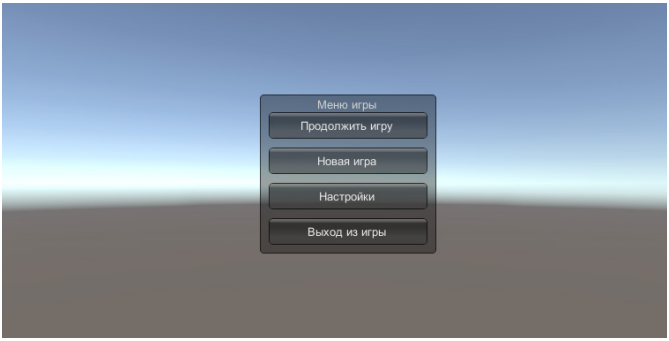


Рис. 4.4. Проверка работоспособности

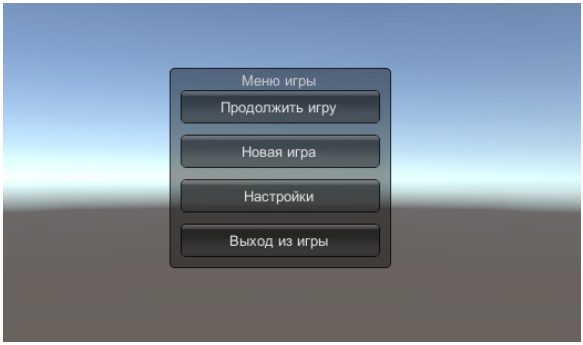


Рис. 4.6. Результат работы

хранения, а затем запускаем приложение и проверяем работоспособность меню (рис. 4.6).

Контрольные вопросы

1. Какие языки программирования можно использовать для написания скриптов в *Unity3D*?
2. С объектом какого типа нужно ассоциировать скрипт меню?
3. Как настраивается расположение на экране клавиш меню?
4. Как необходимо изменить код, чтобы меню открывалось при нажатии на клавишу «*Space*»?

Лабораторная работа № 5
ЗНАКОМСТВО С АНИМАЦИЕЙ, ФИЗИЧЕСКИМИ СВОЙСТВАМИ И СОЗДАНИЕМ ФОНОВОЙ МУЗЫКИ

Цель работы: изучение простой анимации в *Unity3D*, а также физических свойств геометрических объектов и создание фонового звукового сопровождения.

Порядок выполнения работы

1. Создание простой анимации на примере игры змейка.
2. Задание физических свойств геометрическим объектам.
3. Создание фонового звукового сопровождения.
4. Оформление отчета.

Методические указания

Создание простой анимации на примере игры змейка

Поскольку основной критерий создания данного прототипа игры это простота и надежность, в качестве основной модели для создания интерактивных объектов выберем стандартную модель «*cube*», которая представляет из себя *3D*-куб, (рис.5.1), все интерактивные объекты будут созданы на базе этой модели.

Логика игры предусматривает несколько интерактивных объектов, это:

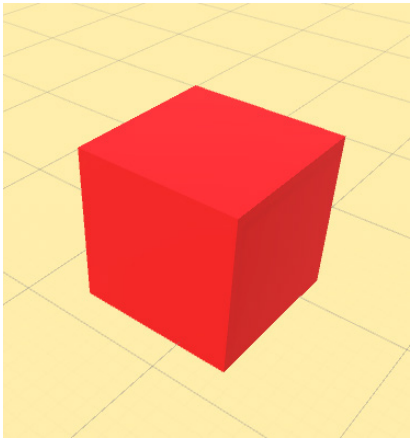


Рис. 5.1. Модель «cube»

1. Голова змейки;
2. Хвост змейки;
3. Бортик;
4. Еда.

В данной концепции, Голову змейки можно считать персонажем, поскольку она является главным действующим лицом на игровой сцене и именно головой управляет пользователь.

Бортик относится к объектам окружения, к игровой сцене, а также к интерактивным объектам, с которым возможно взаимодействовать.

Голова змейки – Snake

Как уже упоминалось ранее, *Snake* создается на базе модели «*cube*», в качестве текстуры будет использована стандартная одноцветная текстура желтого цвета (рис.5.2).

Snake включает в себя настройки нескольких уровней:

1. Настройка анимации;
2. Настройка звукового оповещения пользователя.
3. Настройка скриптов взаимодействия.

В качестве анимации, используем простой приём «*Ping Pong*», который выглядит как увеличение и уменьшения объекта. Для этого используем инструмент «*Animation*» (*Project -> Create -> Animation*),

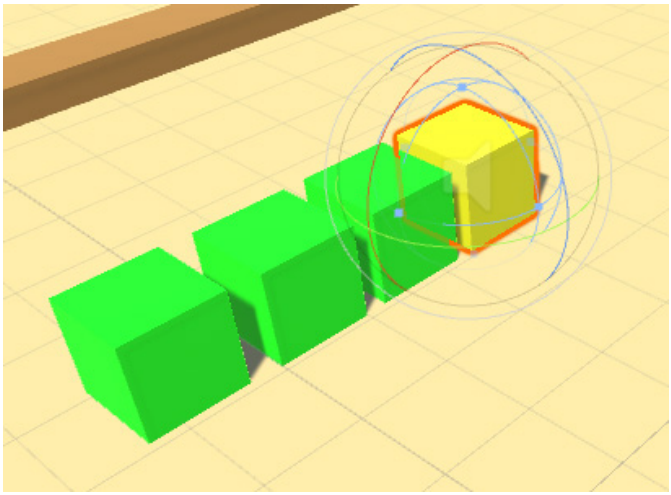


Рис. 5.2. Объект «cube –- голова змейки

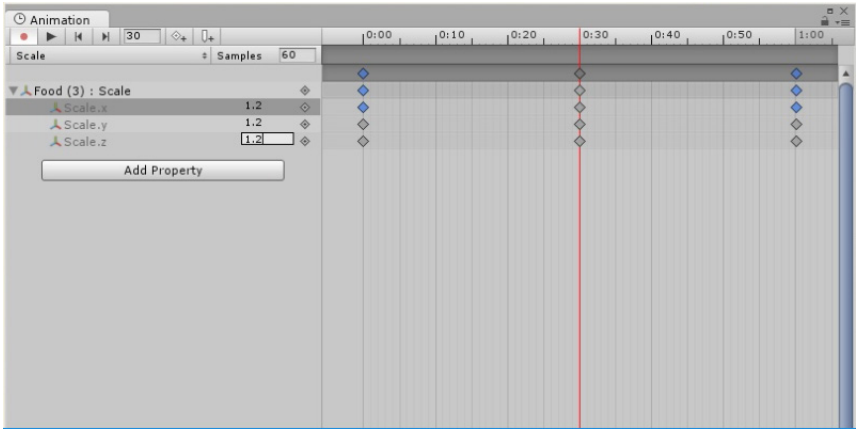


Рис. 5.3. Создание анимации «Scale»

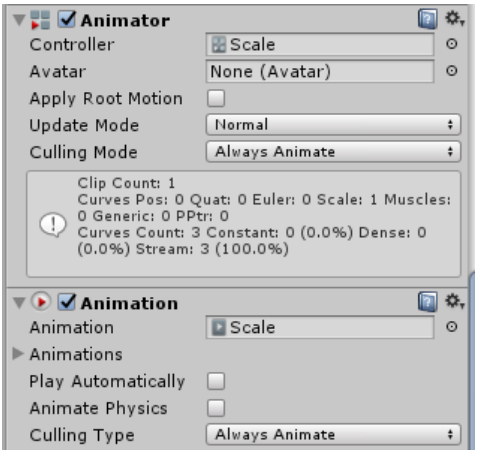


Рис. 5.4. Настройка анимации объекта «Snake»

в котором на временной шкале зададим изменения размеров объекта *Snake* (рис. 5.3). Созданную анимацию назовем: «*Scale*».

В поле инспектора объекта *Snake* добавляем параметры «*Animator*» и «*Animation*» (рис. 5.4).

Для настройки звукового оповещения необходимо произвести настройку соответствующего параметра инспектора объекта «*Snake*».

Добавим звук, который будет воспроизводиться с равномерной громкостью по всей сцене, при поедании Головой змейки еды, и при

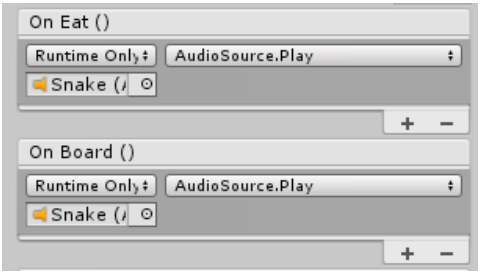


Рис. 5.5. Параметры настройки звуков, объекта «Snake»

столкновении. Для этого настроим параметры: *Component -> Audio -> Audio Source* (рис. 5.5).

Для объекта «*Snake*», осталась выполнить еще несколько настроек, основной из которых будет это подключение скрипта «*Snake Controller*», который будет описан далее.

После всех этих настроек можно сказать, что наш персонаж настроен и готов к интерактивному взаимодействию с другими игровыми объектами.

Хвост змейки – Bone

Данный объект появляется на игровой сцене, после поедания персонажем еды, и количество хвостов равно количеству съеденной еды. Данный объект практически не имеет никаких специфических настроек, и представляет собой стандартную модель типа «*cube*», с добавленным тегом «*body*» (рис. 5.6).

Для добавления тега необходимо использовать инспектор, где находим поле *Tag -> Add tag -> создаём тег «Body»*.

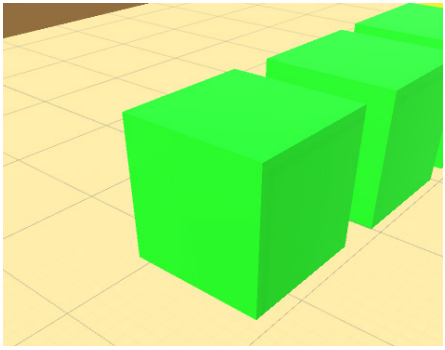


Рис. 5.6. Объект «Bone»

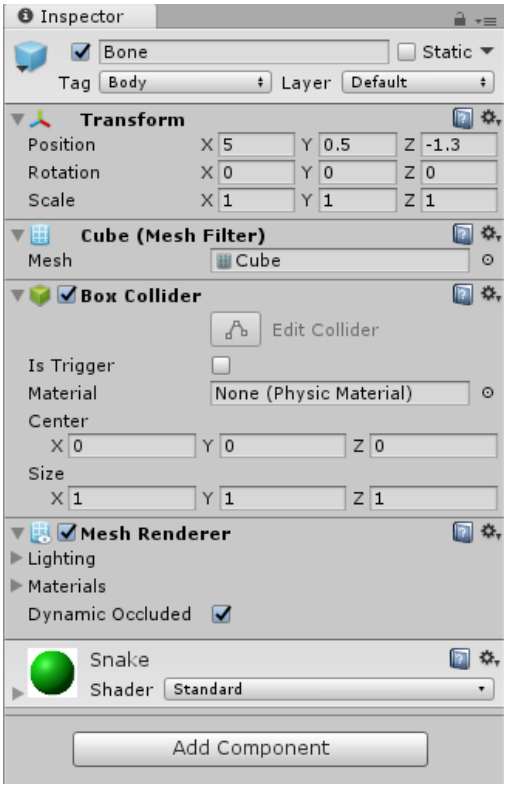


Рис. 5.7. Параметры объекта «Bone»

Текстуру объекта используем стандартную одноцветную заливку зеленного цвета (рис. 5.7).

Бортик – Board

Board представляет собой статичный объект окружающего мира, и одновременно интерактивный объект. Создается он на базе стандартной модели «*cube*».

Объект не имеет специфических параметров, кроме тега «*Board*», который добавляется по аналогии с Хвостом *Bone* (рис. 5.8).

Рис. 5.8. Объект Бортик

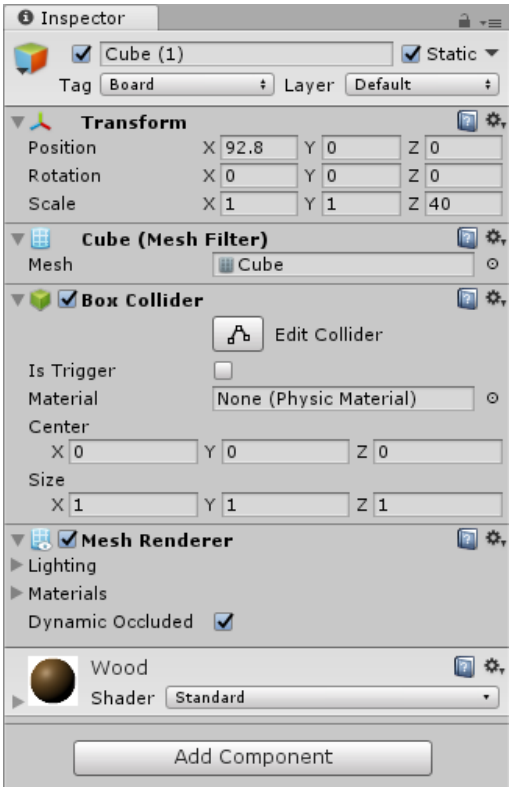


Рис. 5.9. Параметры объекта Бортик

Текстура объекта изготавливается, в виде коричневой одноцветной заливки, параметры аналогичны параметрам Хвоста (рис. 5.9)

Еда – Food

Food – интерактивный объект, созданный на базе стандартной модели «*cube*». Объект является частью игрового объекта, поскольку игроку необходимо есть еду, поэтому он становится целью игрового процесса. Текстура объекта, представляет собой красную одноцветную заливку.

Объект имеет всего две особенности: наличие тега «*Food*» и подключение анимации «*Scale*», по аналогии с объектом «*Snake*» (рис. 5.10).

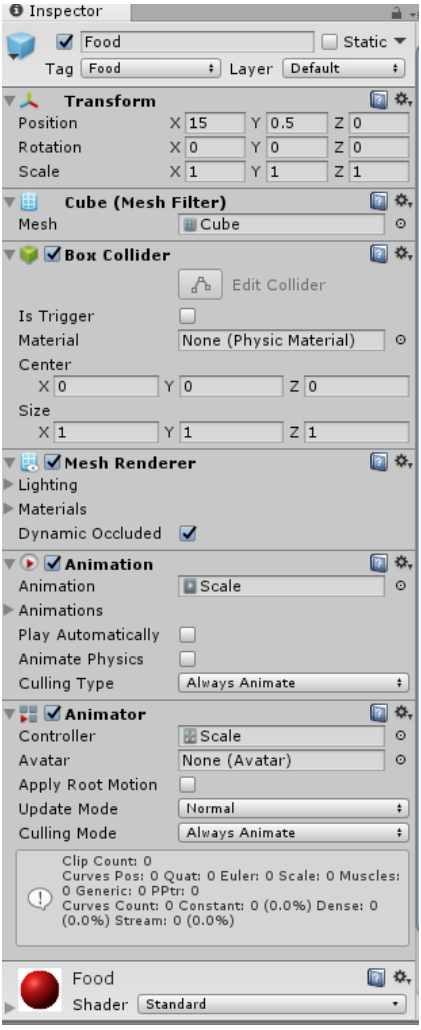


Рис. 5.10. Параметры объекта «Food»

Задание физических свойств с помощью Rigidbody

За придание объектам физических свойств отвечает вкладка *Component -> Physics*. В ней можно найти пункт *Rigidbody* (Твердое тело), с помощью него для любого объекта можно настроить действие виртуальной гравитации. Кроме того, в этой же вкладке находятся несколько других модификаторов отвечающих за настрой-

ку физических свойств. Среди них ряд модификаторов *Collider*. Коллайдер позволяет обрабатывать столкновение объекта с другим объектом, например, в игре персонаж не должен проходить сквозь стену, а должен столкнуться с ней.

Зададим свойства, например, «Голове змейки»: разместим объект в произвольном месте на некотором возвышении над поверхностью ландшафта и нажмем Play. Голова так и останется висеть в указанном месте.

Выйдем из режима проигрывания сцены, вернемся к её редактированию и применим к «Голове» модификатор *Rigidbody (Component -> Physics -> Rigidbody)*. Теперь в инспекторе можно изменить настройки данного модификатора. Для того, чтобы на объект действовала гравитация у него должна быть масса. Установим значение в поле «*Mass*», например, равное 15 (при использовании метрической системы данный показатель измеряется в килограммах). Настроим данные свойства и для других объектов сцены.

Запустив сцену на выполнение, наблюдаем за вполне реалистичным падением объектов (рис. 5.11).

В основе взаимодействия объектов лежит игровая логика, которая для данной игры может быть описана следующим образом:

«Змейка», движется прерывисто. Хвост змеи состоит из отдельных объектов. Когда Голова «Змейки» собирает еду, хвост становится длиннее на один «съеденный» объект.

С каждым перемещением (шагом) головы змеи, на её месте создается «кусочек хвоста», а последний «кусочек» удаляется. Создается

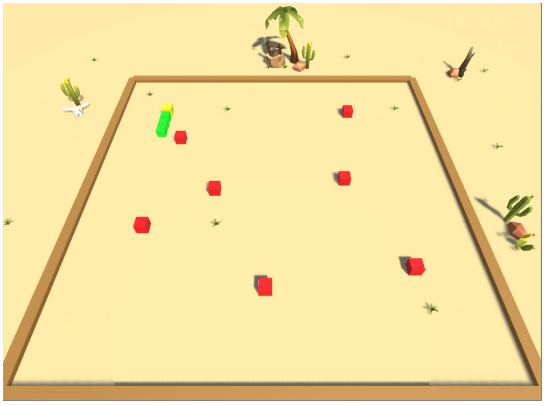


Рис. 5.11. Положение интерактивных объектов на игровой сцене

Таблица 5.1

Теги и их описание

№	Тег	Описание
1	“*Body*”	Объект – хвост
2	“*Food*”	Объект – еда
3	“*Board*”	Объект – забор

впечатление, что змея ползет. Но на самом деле перемешается только голова змеи, а объекты из которых состоит хвост просто создаются в нужном месте и остаются нужное время на игровом находятся объекты еды.

Для взаимодействия каждому объекту присвоим свой тег (табл. 5.1).

Правила взаимодействия объектов:

1. Голова змейки встречает объект с тегом «*Food*», необходимо удалить объект с тегом «*Food*» и добавить объект с тегом «*Body*» к хвосту змейки. Воспроизвести звук: *DM-CGS-11*.

2. Голова змейки встречает объект с тегом «*Board*» или «*Body*», змейка умирает, воспроизвести звук: *DM-CGS-04*.

Для написания скрипта используем среду разработки: *MonoDevelop* и язык программирования *C#*. Создадим *Script C#* (*Project -> Create -> Script C#).*

Для управления объектом используем стрелки указатели «<-» и «->».

Листинг скрипта:

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

public class SnakeController : MonoBehaviour
{
    //Хвосты
    public List<Transform> Tails;
    [Range(0, 3)]
    public float BonesDistance;
    public GameObject BonePrefab;
    //Создать шкалу выбора скорости в редакторе
    [Range(0, 4)]
    public float Speed;
```

```
    private Transform _transform;

    //События
    // Змейка ест
    public UnityEvent OnEat;
    // Змейка столкнулась с бортом
    public UnityEvent OnBoard;
    // Змейка столкнулась с телом
    public UnityEvent OnBody;

    private void Start()
    {
        _transform = GetComponent<Transform>();
    }

    //Вызывается каждый кадр
    private void Update()
    {
        // Движение
//Текущая позиция + вектор нормализованный длины * скорость
MoveSnake(_transform.position + _transform.forward * Speed);
        // Поворот
        //Реагирует на клавиши <- и -> возвращая -1 0 1
        float angel = Input.GetAxis("Horizontal") * 4;
        _transform.Rotate(0, angel, 0);
    }

    private void MoveSnake(Vector3 newPosition)
    {
        float sqrDistance = BonesDistance * BonesDistance;
        Vector3 previosPosition = _transform.position;
        //Предыдущая кость это голова
        //Transform previosBone = _transform;

        //Перебераем все кости в массиве Tails
        foreach (var bone in Tails)
        {
            //Сравниваем квадратную дистанцию
if ((bone.position - previosPosition).sqrMagnitude > sqrDistance)
            {
                var temp = bone.position;
                bone.position = previosPosition;
                previosPosition = temp;

                //Меняем позицию кости
                //bone.position = previosPosition;
            }
            else
```

```
            {
                break;
            }
        }
        _transform.position = newPosition;
    }

    private void OnCollisionEnter(Collision collision)
    {
        //Столкнулись с объектом, тег Food
        if (collision.gameObject.tag == "Food")
        {
            //Удаляем объект с которым столкнулись
            Destroy(collision.gameObject);
            //Создаём элемент хвоста
            var bone = Instantiate(BonePrefab);
            //Добавляем в массив
            Tails.Add(bone.transform);

            //Змейка съела и кто то слушает события
            if (OnEat != null)
            {
                OnEat.Invoke();
            }
        }

        //Столкнулись с объектом, тег Board
        if (collision.gameObject.tag == "Board")
        {
            //Удаляем объект с которым столкнулись
            //Destroy(collision.gameObject);
            //Создаём элемент хвоста
            //var bone = Instantiate(BonePrefab);
            //Добавляем в массив
            //Tails.Add(bone.transform);

//Змейка столкнулась с бортом и кто то слушает события
            if (OnBoard != null)
            {
                OnBoard.Invoke();
            }
        }

        //Столкнулись с объектом, тег Body
        if (collision.gameObject.tag == "Body")
        {
            //Удаляем объект с которым столкнулись
            //Destroy(collision.gameObject);
            //Создаём элемент хвоста
```

```
            //var bone = Instantiate(BonePrefab);
            //Добавляем в массив
            //Tails.Add(bone.transform);

//Змейка столкнулась с телом и кто то слушает события
            if (OnBody != null)
            {
                OnBody.Invoke();
            }
        }
    }
}
```

Создание фоновой музыки

Для полного погружения пользователя в игровой мир, необходимо создать музыкальное сопровождение игрового процесса.

В качестве звукового сопровождения выберем саундтрек: *Music Dance006(Mach058)*.

Для организации фонового звука зададим источник звука. Им может быть любой объект с примененным компонентом «*Audio Source*» (*Component -> Audio -> Audio Source*). В инспекторе настраивается композиция, которая будет звучать (в нашем случае *Dance006 (Mach058)*). Предварительно музыкальный файл необходимо поместить в папку с проектом (рис. 5.12).

Для того, чтобы сделать выбранный объект слушателем, необходимо добавить компонент «*Audio Listener*» (*Component -> Audio -> Audio Listener*). *First Person Controller* по умолчанию является слушателем и применять эти действия к нему не надо. Добавим саундтрек на главную камеру (рис. 5.12).

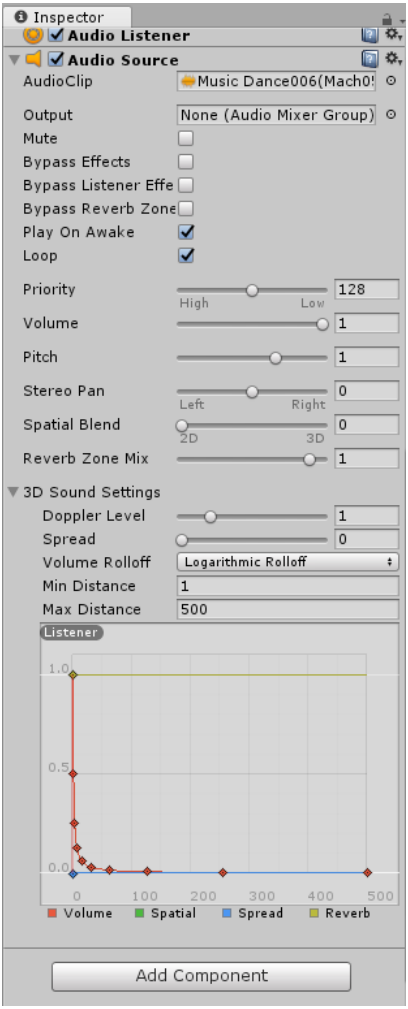


Рис. 5.12. Параметры воспроизведения музыки на главной игровой камере

Контрольные вопросы

1. Расскажите процессе создания пошаговой анимации в *Unity3D*?

2. С помощью каких настроек можно включить проигрывание анимации с самого начала сцены?

3. Какой модификатор отвечает за применение гравитации к объекту?

4. Какой модификатор отвечает за твердость и невозможность пройти сквозь

объект?

5. Расскажите о процессе создания фонового звукового сопровождения?

Лабораторная работа № 6
ОЗНАКОМЛЕНИЕ С ФУНКЦИОНАЛЬНЫМИ ВОЗМОЖНОСТЯМИ УПРАВЛЕНИЯ ХАРАКТЕРИСТИКАМИ КАМЕРЫ

Цель работы: изучение функциональных возможностей управления характеристиками камеры, предоставленными пакетом *Unity3D*, а также исследование возможностей их применения на тестовой модели.

Порядок выполнения работы

1. Ознакомиться с содержанием раздела «Теоретические сведения».
2. Изучить функциональные возможности управления характеристиками камеры, предоставленные пакетом *Unity3D*.
3. Создать простейшую тестовую сцену в *Unity3D*.
4. Исследовать описанные свойства камеры на тестовой модели.
5. Сформулировать выводы.
6. Оформить отчет.

Рекомендации по содержанию тестовой сцены:

1. Стандартный *Terrain*.
2. Глобальное освещение (*Directional light*).
3. *First Person Controller*.
4. Стандартные примитивы или простые модели.
5. Более двух камер.

Методические указания

Теоретические сведения – координатное пространство камеры, внутреннее пространство пирамиды, поле зрения, буфер глубины, виды проецирования, рассмотрены в [2].

Виртуальная камера в Unity 3D

Камеры используются для отображения пользователю игрового мира. Поэтому в каждой сцене присутствует минимум одна камера, их количество не ограничено. Несколько камер может использоваться для разделения экрана для двух игроков, создания спецэффектов. Манипулируя камерами, можно придать игре или приложению уникальный вид, значительно повысить реалистичность проектов. Изображения с камеры могут визуализироваться в любой последовательности в любом месте экрана или только в указанных

частях экрана. Таким образом, камера является достаточно гибким объектом [2].

Пакет *Unity3D* предоставляет разработчику обширные возможности управления характеристиками камеры. Рассмотрим некоторые свойства камеры в табл. 6.1.

Таблица 6.1

Свойства камеры в пакете Unity3D

Свойство	Функция
Clear Flags	Определяет, какие части экрана будут очищены. Это удобно при использовании нескольких камер для отрисовки разных элементов игры.
Background	Цвет, применяемый для фона после отрисовки всех элементов, в случае отсутствия скайбокса.
Culling Mask	Включение или исключение слоёв объектов на рендер этой камерой. Назначение слоёв объектам производится через *Inspector*.
Projection	Переключает способность камеры симулировать перспективу.
Perspective	Камера будет рисовать объекты в перспективе.
Orthographic	Камера будет рендерить объекты единообразно, без ощущения перспективы. Примечание: *Deferred* рендеринг не поддерживается в данном режиме. *Forward* рендеринг всегда используется.
Size	Размер зоны видимости камеры для ортографического режима.
Field of view	Для режима перспективы – ширина угла обзора камеры, измеряется в градусах по локальной оси *Y*.
Clipping Planes	Отсекающие плоскости – дистанция, на которой камера начинает и заканчивает рендеринг.
Near	Ближайшая точка относительно камеры, которая будет рисоваться.
Far	Дальняя точка относительно камеры, которая будет рисоваться.
Normalized View Port Rect	Четыре значения, отражающие то, в какой области экрана будет выведено изображение с данной камеры, в экранных координатах (от 0 до 1).
X	Начальная позиция области по горизонтали вида камеры, который будет рисоваться

Окончание табл. 6.1

Свойство	Функция
Y	Начальная позиция области по вертикали, где вид камеры будет рисоваться.
W (Width)	Ширина вида камеры на экране.
H (Height)	Высота вида камеры на экране.
Depth	Позиция камеры в очереди отрисовки. Камеры с большим значением будут нарисованы поверх камер с меньшим значением.
Rendering Path	Опции для определения методов рендеринга для камеры.
Use Player Settings	Камера использует метод рендеринга, установленный в *Player Settings*.
Vertex Lit	Все объекты, рисующиеся этой камерой, будут рендериться как *VertexLit*-объекты.
Forward	Все объекты будут рендериться с одним проходом на материал.
Deferred Lighting	Все объекты будут рендериться сначала без освещения, а затем будет произведён рендер освещения для всех объектов разом, в конце очереди рендеринга.
Target Texture	Ссылка на *Render Texture*, которая будет содержать результат рендеринга камеры. Назначение этой ссылки отключает способность камеры рендерить на экран.
HDR	Включение технологии *High Dynamic Range*.
Target Display	Which external device to render to. *Between 1 and 8*.

После ознакомления с основными свойствами камеры приступаем к выполнению лабораторной работы. Для достижения цели лабораторной работы необходимо создать дополнительно камеры на игровой сцене (рис. 6.1).

После добавления камеры в проект ознакомимся с основными настройками на вкладке *Inspector*. При помощи *Background* можно управлять информацией о цвете и глубине вида рендера камеры (рис. 6.2).

Culling Mask используется для выборочной визуализации объектов с использованием слоев (рис. 6.3).

Viewport Rect определяет части экрана, на которой будет отрисовано изображение с камеры (рис. 6.4, 6.5).

После завершения основных настроек запустим режим игры. В итоге получаем подобное разделение экрана (рис. 6.6).

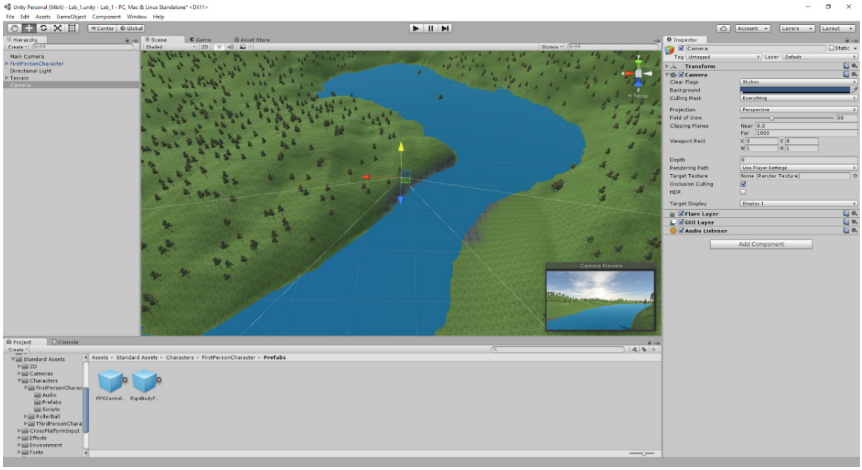


Рис. 6.1. Добавление камеры в проект

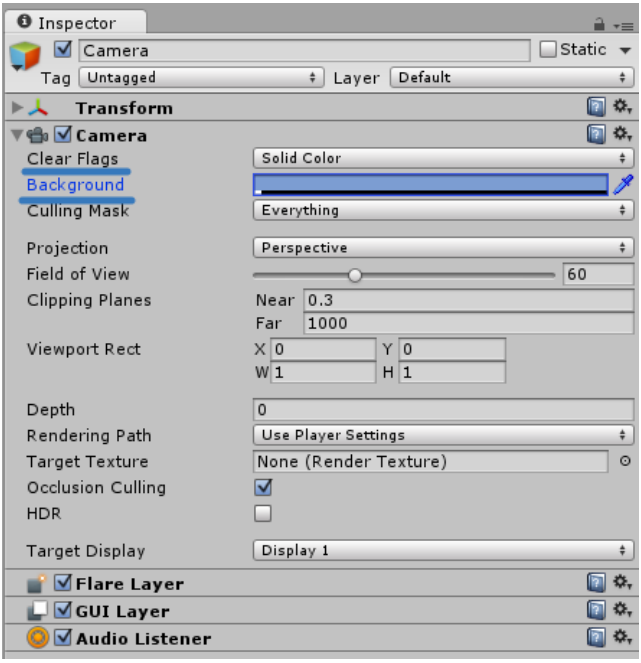


Рис. 6.2. Изменение информации о глубине и цвете рендера камеры

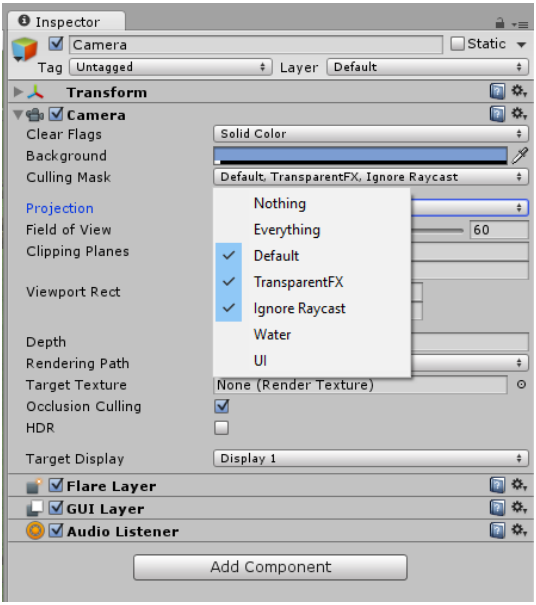


Рис. 6.3. Изменение слоев визуализации

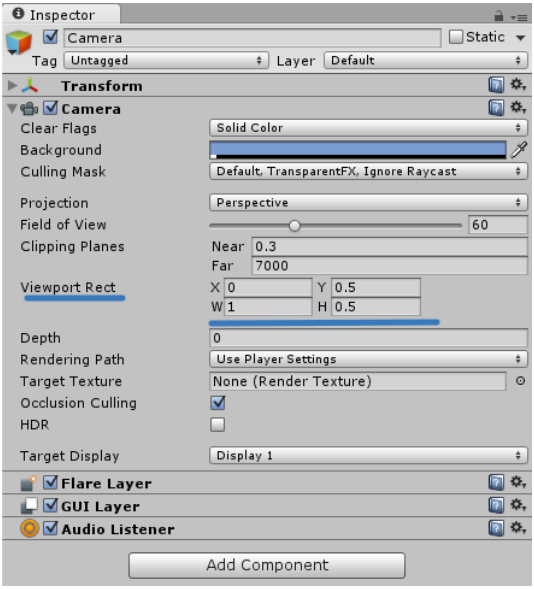


Рис. 6.4. Расположение камеры на экране

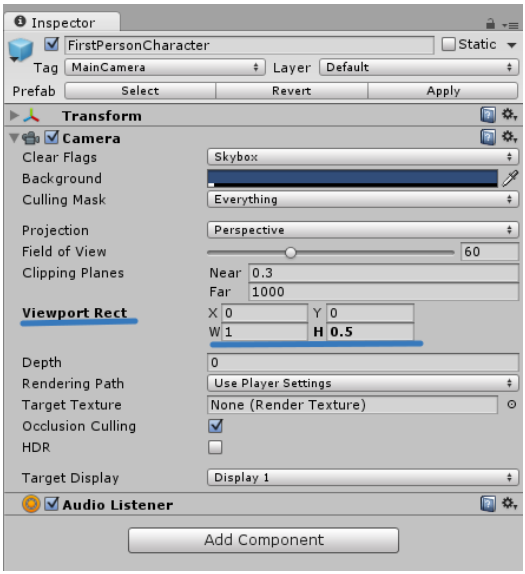


Рис. 6.5. Расположение камеры на экране

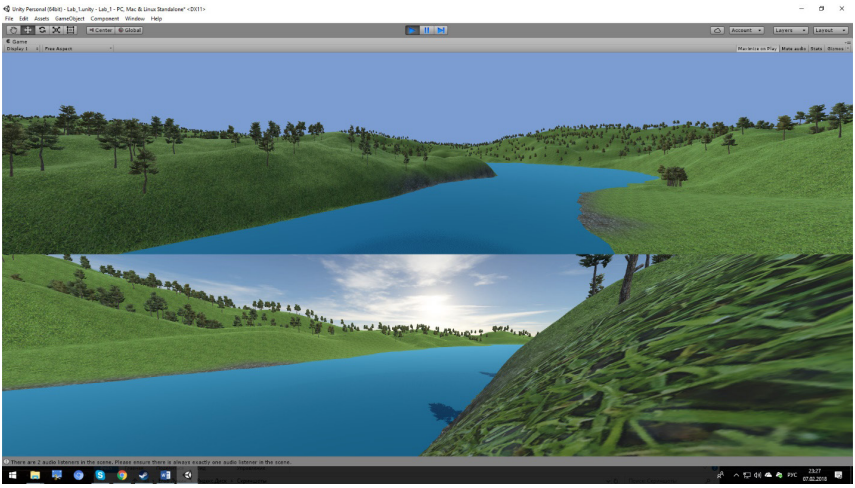


Рис. 6.6. Готовая сцена с большинством изменений

Режим *HDR (High Dynamic Range)* можно применить для получения эффекта улучшенного изображения рендера (рис. 6.7).

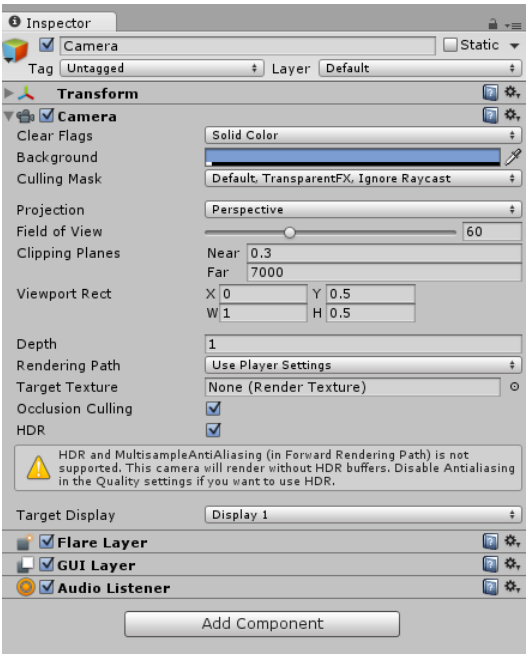


Рис. 6.7. Включение HDR режима

Также можно добавить еще 1 камеру и сделать разделение камер на большее количество (рис. 6.8).

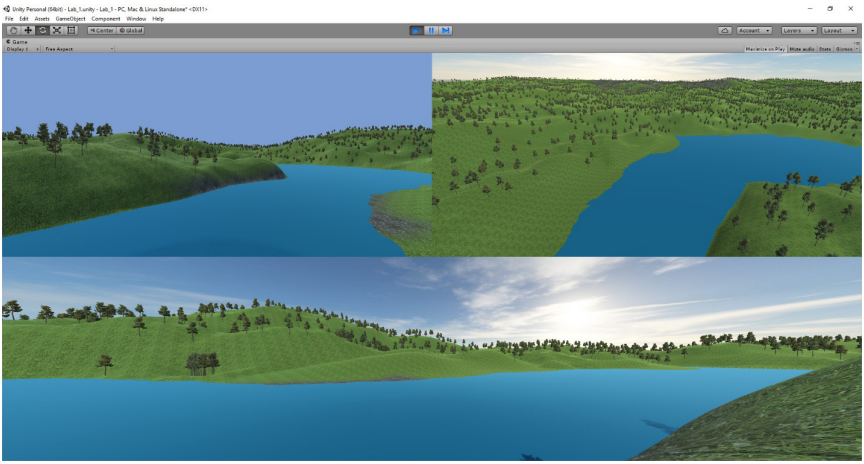


Рис. 6.8 Финальный рендер

Контрольные вопросы

1. Сколько плоскостей отсечения используется для определения видимого пространства *3D* сцены?

2. Как регулируется область видимого пространства камеры?

3. Сколько камер может быть установлено в *3D* сцене?

4. Какие реалистичные визуальные эффекты можно привязать к камере?

5. Как настроить несколько камер так, чтобы была возможность использовать любую из них как основную?

6. Назовите способы анимации камеры?

7. Возможно ли управление камерой с применением физических законов?

8. Когда применяется включение ортографического режима камеры?

Лабораторная работа № 7
ИЗУЧЕНИЕ МЕХАНИЗМА УРОВНЕЙ ДЕТАЛИЗАЦИИ

Цель работы: изучение уровней детализации (*LOD*) объектов в интерактивной трехмерной сцене, реализация статического *LOD* в *Unity3D* и анализ его влияния на параметры рендеринга.

Необходимые информационные, программные и аппаратные средства:

1. *Unity 3D*, актуально в любой версии старше 2018 г.

Порядок выполнения работы

1. Ознакомиться с содержанием раздела «Теоретические сведения».
2. Изучить способы подключения уровней детализации (*LOD*) объектов в *Unity3D*.
3. Создать простейшую тестовую сцену в *Unity3D*.
4. Подготовить заданное преподавателем количество *3D* моделей объектов с различным уровнем детализации в *формате .fbx*, используя *3D* редактор.
5. Импортировать подготовленные *3D* модели в *Unity3D* и на их основе создать дискретный *LOD*.
6. Проанализировать данные статистики для созданной *3D* сцены и сделать выводы.
7. Оформить отчет.

Рекомендации по содержанию тестовой сцены:

1. Стандартный *Terrain*, или ландшафт, разработанный в предыдущей лабораторной работе.
2. *First Person Controller*.
3. *Directional Light*.
4. Количество уровней детализации объектов (*LOD*) от 2 до 4.
5. Число полигонов для модели нулевого уровня (*LOD 0*) не должно быть менее (№ по журналу преподавателя) *100.

Методические указания

Теоретические сведения – статические и динамические *LOD*, алгоритмы дискретного, непрерывного и видозависимого *LOD*, борьба с эффектом «перепрыгивания», рассмотрены в [2].

Реализация механизма LOD в Unity3D

Unity3D имеет встроенный механизм уровней детализации, представляющий собой дискретный статический *LOD*, основанный на метрике расстояния, заданного в процентном соотношении. Для использования этого механизма в сцену добавляются специальные объекты, т. н. *LOD Group*, к которым далее присоединяются представления объекта для разных уровней детализации. Добавить *LOD Group* можно через меню *Component – Rendering*. В настройках *LOD* группы выставляются фиксированные значения % от экранной площади, при которых происходит переключение с одного уровня на другой.

Для исследований функций *LOD* создадим дополнительный объект на сцене (рис. 7.1), (*GameObject – Create Empty* или *Ctrl+Shift+N*), к которому и будет привязана *LOD Group*. В дальнейшем, чтобы избежать путаницы, объектам *LOD Group* рекомендуется давать звучные имена, например «*LOD*».

Создадим *LOD Group*. (рис. 7.2).

К группе *LOD* новые уровни добавляются через контекстное меню (*Insert Before*), вызываемое при щелчке правой кнопки по полосе, отображающей уровни детализации. Для того чтобы удалить уровень детализации используется пункт *Delete*. Добавим новые уровни *LOD* (рис. 7.3).

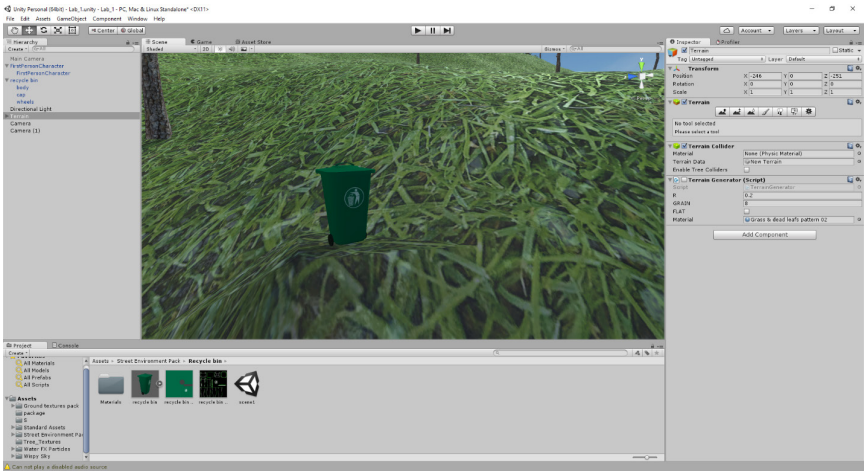


Рис. 7.1. Новый объект на сцене

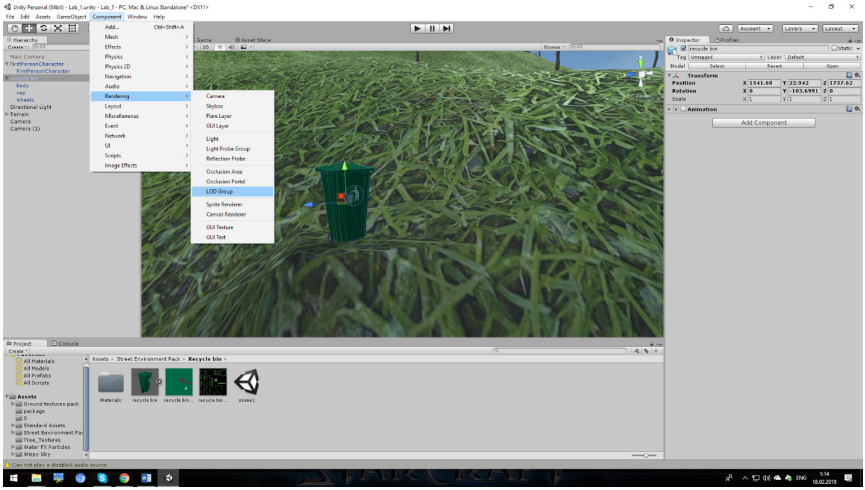


Рис. 7.2. Создание LOD Group

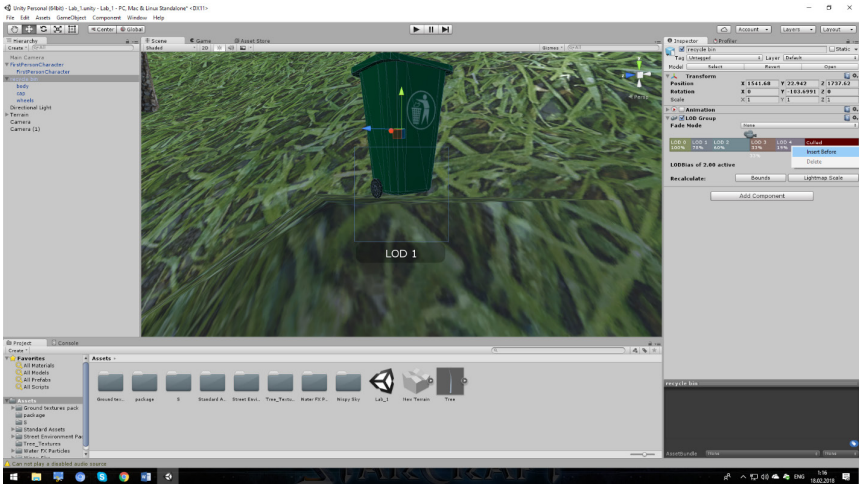
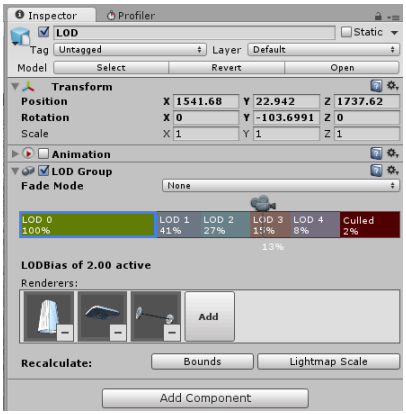


Рис. 7.3. Добавление новых уровней

Привяжем нужные объекты к уровням детализации (рис. 7.4–7.8).

После завершения работы перейдем в режим игры и проверим работоспособность добавленных *LOD* уровней (рис. 7.9).



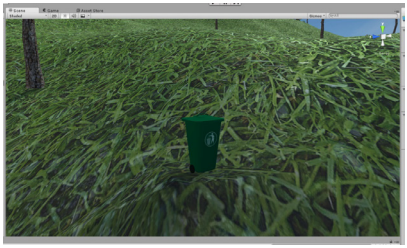
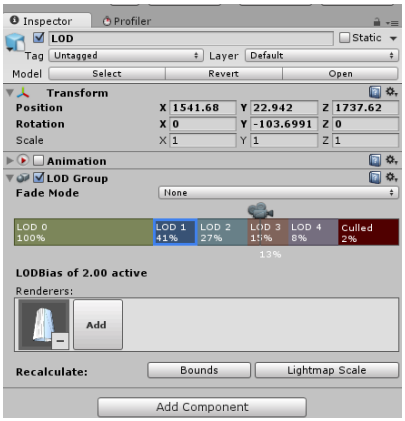


Рис. 7.4. Привязка объектов к 0 уровню



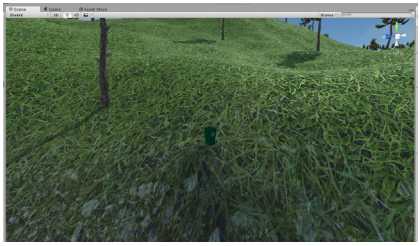
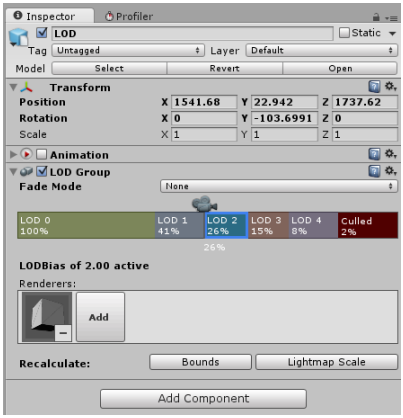


Рис. 7.5. Привязка объектов к 1 уровню



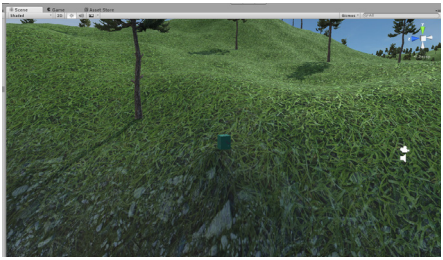
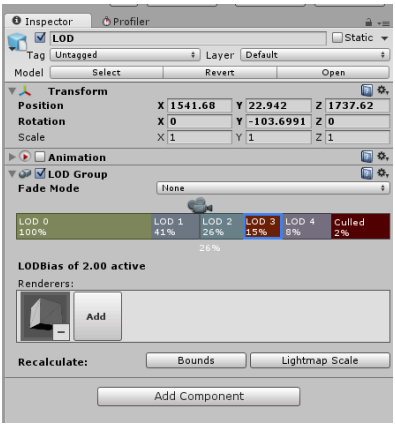
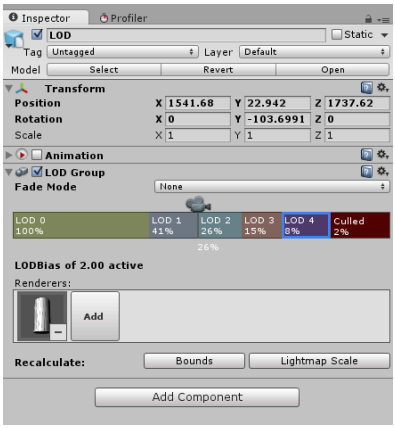


Рис. 7.6. Привязка объектов к 2 уровню





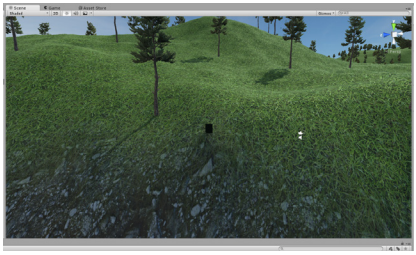


Рис. 7.7. Привязка объектов к 3 уровню

Рис. 7.8. Привязка объектов к 4 уровню

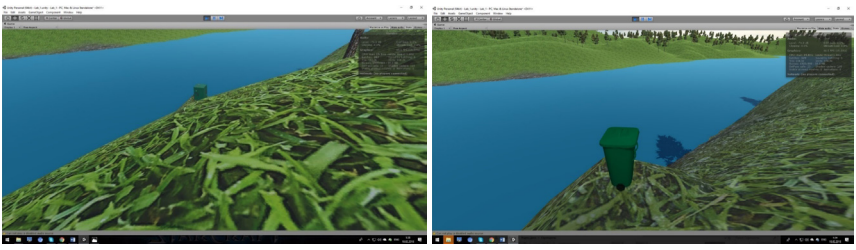


Рис. 7.9. Сравнение уровней детализации

Контрольные вопросы

1. Что такое *LOD*?
2. Перечислите известные вам методы ускорения процесса визуализации интерактивной *3D*-сцены?
3. Что происходит с объектом, если уровень *LOD* = *Culled*?

4. На какие группы делятся алгоритмы *LOD*?

5. Перечислите имеющиеся классификации *LOD* алгоритмов, если они существуют.

6. Назовите и недостатки статических *LOD*-алгоритмов.

7. Назовите преимущества и недостатки динамических *LOD*-алгоритмов

8. Назовите преимущества и недостатки видозависимого *LOD*-алгоритма.

Лабораторная работа № 8
ОЗНАКОМЛЕНИЕ С ФУНКЦИОНАЛЬНЫМИ ВОЗМОЖНОСТЯМИ УПРАВЛЕНИЯ ПИРАМИДОЙ ВИДИМОСТИ

Цель работы: изучение функциональных возможностей управления пирамидой видимости в *Unity3D*, исследование возможностей функции *Occlusion Culling* на тестовой модели.

Порядок выполнения работы

1. Ознакомиться с содержанием раздела «Теоретические сведения».
2. Изучить функциональные возможности управления пирамидой видимости, предоставленные *Unity3D*.
3. В *Unity3D* создать тестовую сцену.
4. Исследовать возможности функции *OcclusionCulling* на тестовой сцене.
5. Предоставить результаты проверки сгенерированной окклюзии:

a. Окно статистики (до и после применения *OcclusionCulling*);

b. Скриншоты экрана в режиме *Visualize* (до и после применения *OcclusionCulling*);

6. Сделать выводы о проделанной работе.
7. Оформить отчет.

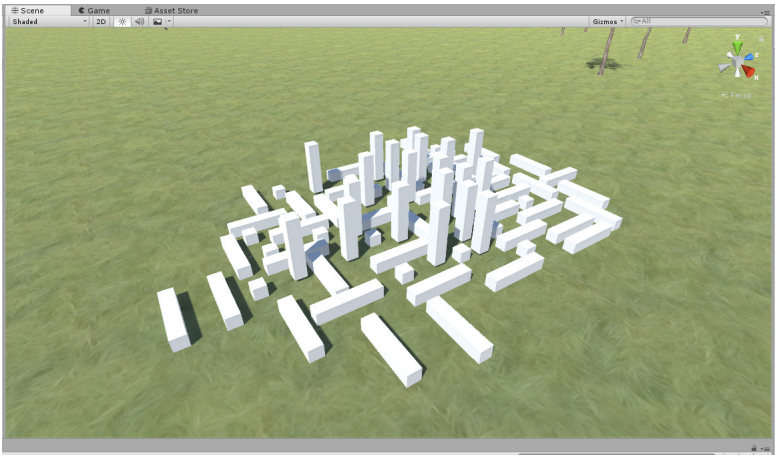


Рис. 8.1. Пример расположения объектов сцены

Рекомендации по содержанию тестовой сцены:

Тестовая сцена должна содержать:

– Стандартный *Terrain;*

– Глобальное освещение (*Directional light*);

– *First Person Controller;*

– Стандартные примитивы: куб (n*n*n) ~ 20 шт., куб (5n*n*n) ~ 20 шт., куб (n*5n*n) ~ 20 шт., куб (n*n*5n) ~ 20 шт., куб (n*n*n) ~ 10 шт.

Стандартные примитивы должны быть расставлены в случайном порядке, например, так, как показано на рис. 8.1.

Теоретические сведения – использование современных механизмов управления количеством объектов сцены, попадающих в пирамиду видимости – *OcclusionCulling* (Удаление скрытых частей) и *PVS (PotentiallyVisibleSet* – потенциально видимый набор), рассмотрены в [2].

Методические указания

Настройка OcclusionCulling

Для использования *Occlusion Culling* необходимо выполнить ряд настроек.

Первый этап – разбиение всей геометрии сцены на части разумного размера. Также важно разбить уровни на небольшие, четко определенные области, которые закрыты другими крупными объектами (здания, стены и т. п.) Видимость каждого меша будет включатся и выключатся в зависимости от данных *OcclusionCulling*. Например, если выделить каждый объект интерьера отдельным мешем, то они будут видны или невидимы в зависимости от поворота камеры. Либо если есть объект, который содержит все элементы интерьера, тогда весь набор будет либо виден, либо невиден. Во-первых, в *Inspector* следует обозначить объекты, которые нужно включить в *OcclusionCulling* как *OcclusionStatic*. Выбираем из иерархии сцены необходимые объекты и помечаем их *OccluderStatic* или *OccludeeStatic* (рис. 8.2). *OccludeeStatic* используется для прозрачных или очень маленьких объектов, которые ничего не закрывают сами, но будут закрываться другими объектами. Это помогает уменьшить вычисления. Выделяем все маленькие кубы и меняем настройку. *OccluderStatic* используется для того, чтобы применить окклюзию к объекту.

Для основных настроек *OcclusionCulling* удобнее использовать окно *OcclusionCulling (Window -> OcclusionCulling)* (рис. 8.3)

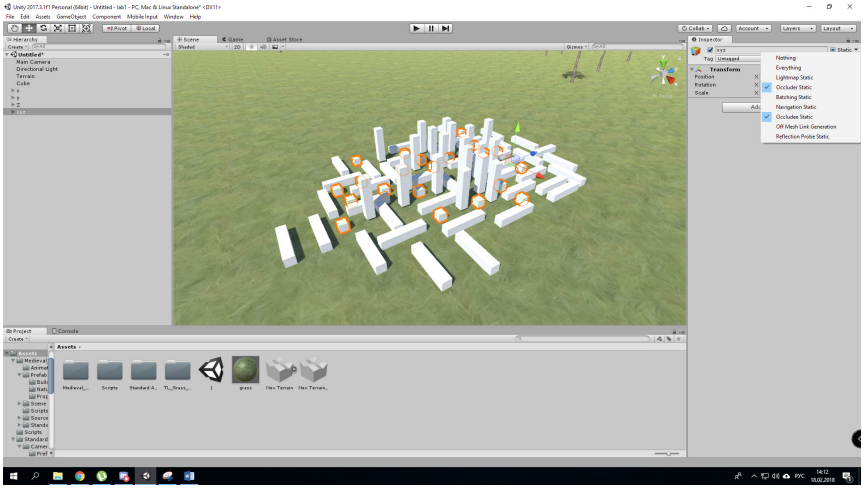


Рис. 8.2. Обозначение объекта для OcclusionCulling

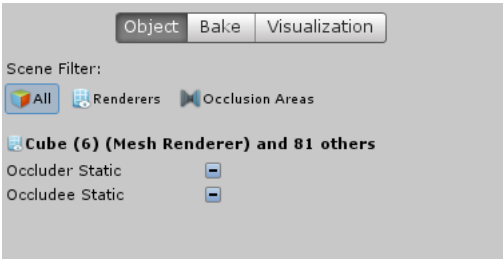


Рис. 8.3. Окно OcclusionCulling

OcclusionArea

Если не задать *OcclusionAreas*, то, во первых, по умолчанию *OcclusionCulling* будет применено ко всей сцене целиком, что приведёт к большому количеству вычислений.

Во-вторых, когда камера находится вне *OcclusionArea*, то *OcclusionCulling* не будет работать. Необходимо установить *OcclusionAreas*, чтобы определить места, где может находится камера.

В-третьих, слишком большие области *OcclusionArea* усложняют процесс просчёта. Для того, чтобы применить *OcclusionCulling* к динамичным объектам, создаются *OcclusionArea*. Размер меня-

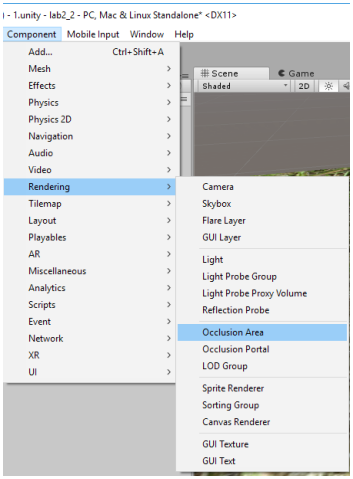


Рис. 8.4. Создание Occlusion Area

ется таким образом, чтобы в ней поместились все необходимые объекты. Для того, чтобы создать *OcclusionArea*, необходимо добавить компонент *OcclusionArea* пустому *GameObject*. *GameObject* (*Component* > *Rendering* > *Occlusion Area* из пункта меню) (рис. 8.4 и 8.5). Основные настройки *Occlusion Area* представлены в табл. 8.1.

После того, как была создана *OcclusionArea*, необходимо проверить, каким образом произошло разбиение бокса на клетки. Для

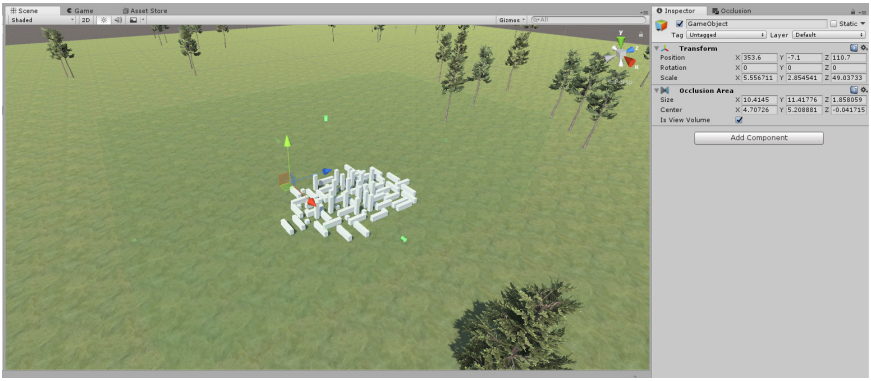


Рис. 8.5. Окно настроек оптимизированной площади и результат построение Occlusion Area

Таблица 8.1

Настройки Occlusion Area

Свойство	Функция
Size	Определяет размер зоны отсечения.
Center	Позволяет выставить центр зоны отсечения. По умолчанию его значение равно 0,0,0 и расположен он как правило в центре цветного бокса
Is View Volume	Определяет, где может находиться камера. Выставляйте этот флажок, если вам необходимо отсечь статичные объекты именно в этой зоне отсечения

Рис. 8.6. Occlusion Culling Preview Panel

этого нужно выбрать *Edit* и в панели *OcclusionCulling Preview Panel* выбрать *Visualise* (рис. 8.6).

Occlusion Culling – Bake

После окончания настройки, необходимо нажать на кнопку «*Bake*» чтобы запустить обработку данных «*Occlusion Culling*» (рис. 8.7). Основные настройки *Backing* представлены в табл. 8.2.

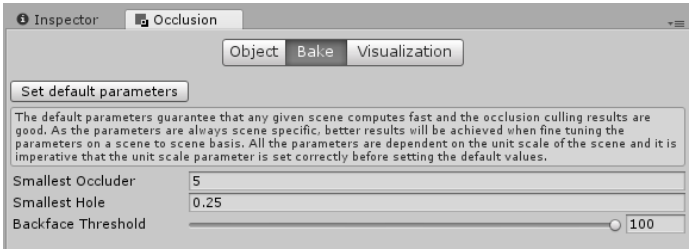


Рис. 8.7. Окно настроек Bake

Таблица 8.2

Настройки Backing

Свойства	Функция
Smallest Occluder	Размер минимального объекта, который способен скрыть другие объекты при выполнении *Occlusion Culling*. Любые объекты меньшего размера, никогда не будут вызывать *Occlusion Culling*, для объектов, которые они заслоняют. Например, если поставить значение параметра 5, то объекты, которые выше или шире, чем 5 метров будут скрывать объекты, скрытые за ними

Окончание табл. 8.2

Свойства	Функция
Smallest Hole	Это значение представляет собой наименьший разрыв, через который объекты могу просматриваться камерой. Значение представляет собой диаметр объекта, который может поместиться в отверстии. Если в сцене имеются небольшие трещины, через которые камеры должна видеть, то значение должно быть меньше
Backface Threshold	Окклюзия *Unity3D* использует оптимизацию размера данных, которая сокращает их объем, отбрасывая ненужные детали, проверяя обратные стороны объектов. Значение по умолчанию 100 не удаляет обратные части объектов из набора данных, когда значение 5 агрессивно сокращает объем данных. Идея заключается в том, что как правило, в зону видимости камеры не попадают такие поверхности, как обратная сторона ландшафта или внутренние поверхности твердотельного объекта. Со значением параметра меньше 100, *Unity* удалит эти области из набора данных, тем самым уменьшив объем хранимой информации

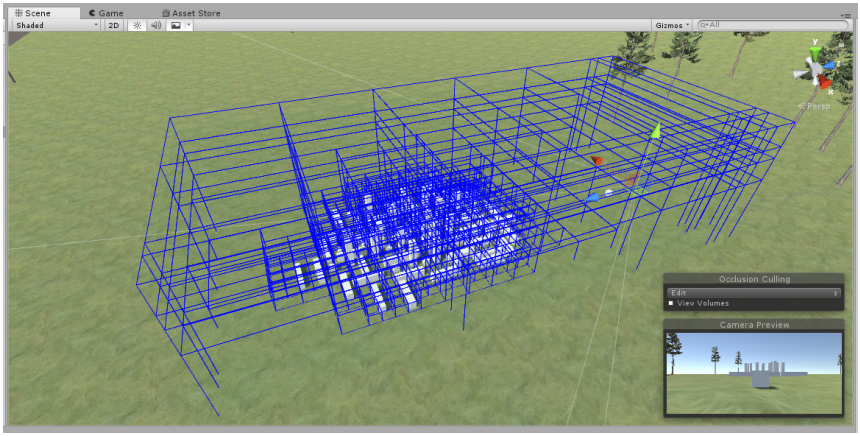


Рис. 8.8. Области Occlusion Culling

При включении визуализации *View Volumes*, с вкладкой *Bake* (рис. 8.8) можно увидеть сгенерированные области.

Проверка результатов окклюзии

Для того, чтобы проверить сгенерированную окклюзию выберите *OcclusionCulling* (в панели *Occlusion Culling Preview Panel* нажмите режим *Visualize*) и перемещайте камеру (*MainCamera*) в режиме *SceneView* (рис. 8.9, 8.10).

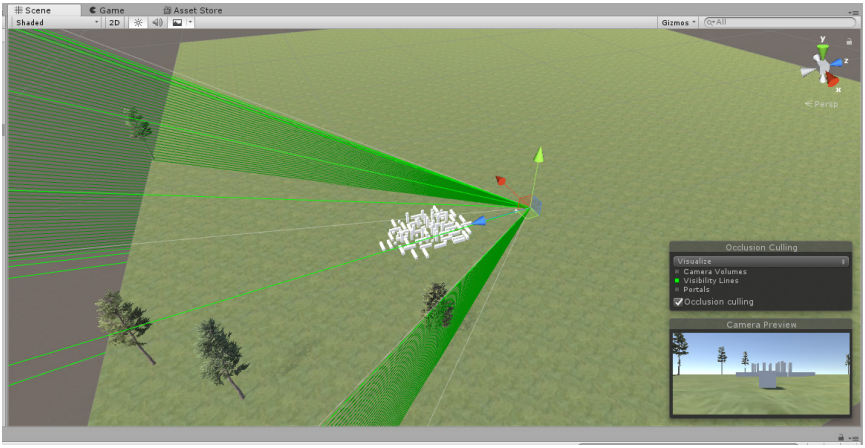


Рис. 8.9. Скриншот сцены до применения OcclusionCulling

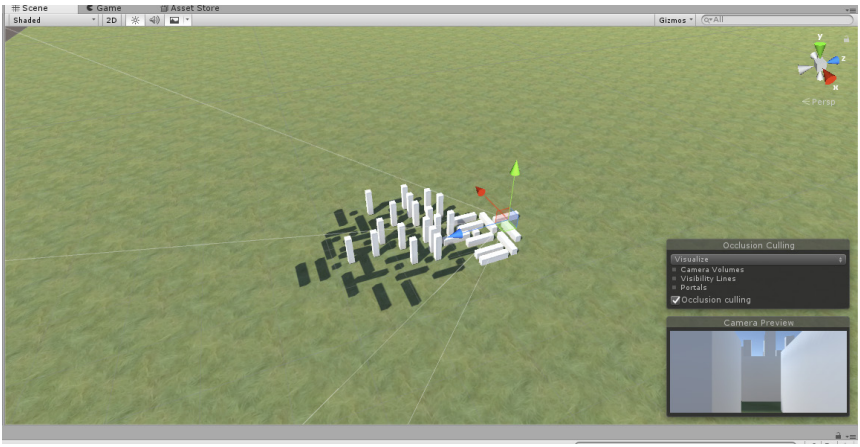


Рис. 8.10. Скриншоты сцены после применения функции OcclusionCulling

Контрольные вопросы

1. Для чего требуется применение механизмов управления количеством объектов сценой, попадающих в пирамиду видимости?

2. Что относится к основным моментам настройки функции *OcclusionCulling*?

3. Когда следует использовать *OccludeeStatic*?

4. Что такое *OcclusionArea*?

5. Какие показатели определяют качество сгенерированной окклюзии?

Лабораторная работа № 9
РАЗРАБОТКА ПРОСТОЙ МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ

Цель работы: разработка простой многопользовательской игры в *Unity3D*, создание аватара игрока и настройка системы «клиент-сервер» на основе мастер-сервера *Unity3D*.

Порядок выполнения работы

1. Ознакомиться с содержанием раздела «Теоретические сведения».
2. Создать в *Unity3D* новый проект и пустую сцену.
3. Подготовить игровой уровень.
4. Создать аватар игрока.
5. Подключить скрипт управления аватаром «*PlayerMovement*», приведенный в разделе «Теоретические сведения».
6. Добавить компонент «*Rigidbody*», подключая тем самым физику к аватару.
7. Проверить работоспособность аватара, запустив сцену в режиме игры.
8. Создать многопользовательский режим взаимодействия аватаров, используя компонент *NetworkManager*.
9. Создать интерфейс игры на основе скрипта «*Game Menu*», приведенного в разделе «Теоретические сведения».
10. Для проверки результата запустить минимум два приложения одновременно на одном компьютере (проект необходимо собрать как *File* → *Build Settings...*→ *Build*). В первом приложении – создать игру, во втором – подключиться к созданной игре и проверить корректность работы. (Изменения в игре не видны в неактивном окне).
11. Оформить отчет.

Рекомендации по содержанию тестовой сцены

Игровой уровень должен содержать:

– Плоскость (*Plane*), выступающую в роли поверхности;

– Камеру (*main Camera*);

– Источник света (например, *Directional Light*);

– Аватар игрока может быть промоделирован простыми геометрическими объектами.

Методические указания

Существует несколько способов организации многопользовательской игры, основными из которых являются построение одно-

ранговой сети и создание системы «клиент-сервер». Первый способ подразумевает распределение нагрузки по обработке данных игры между всеми устройствами сети, при втором основная нагрузка ложится на одно или несколько серверных устройств, на остальных нагрузка сводится к минимуму. В «чистых» многопользовательских играх основным является второй способ, который не только расширяет пользовательскую базу (так как в игру могут играть люди с более слабыми устройствами) но и позволяет эффективно решать многие задачи, трудновыполнимые в одноранговой сети (например, настройка и администрирование игры, технические работы или борьба с читерами). В *Unity3D* версий 5.1 и выше встроена новая усовершенствованная сетевая система, которая обладает большей гибкостью и мощностью, чем в более ранних версиях. В сетевой технологии *Unity3D* в игре есть Сервер и множество Клиентов. Когда нет выделенного сервера, один из клиентов играет роль сервера, тогда этот клиент называется «Хост» (рис. 9.1).

Хост является и сервером, и клиентом в одном процессе. Хост использует особый тип клиента, называемый *LocalClient*, а другие *RemoteClients*. *RemoteClients* подключаются к серверу через интернет соединение. Цель сетевой технологии *Unity* в том, чтобы код программы для *LocalClients* и *RemotrClients* был одним и тем же, так чтобы разработчикам приходилось думать только об одном типе клиентов. В *Unity3D* версиях 5.1 и более поздних реализован компонент *NetworkManager* для управления состоянием сети мультиплеерной игры. *NetworkManager* предоставляет широкую функциональность в одном компоненте и делает разработку, тестирование

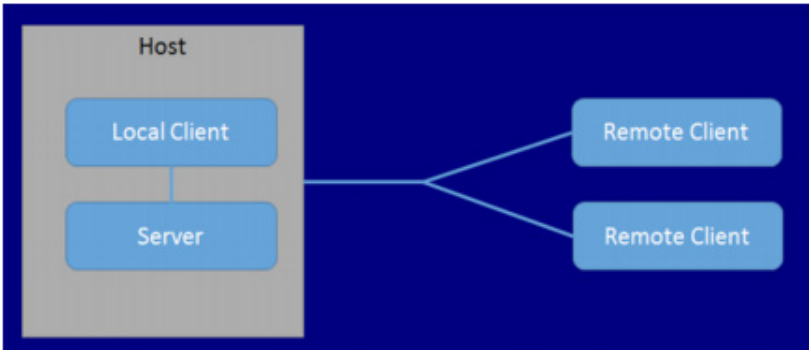


Рис. 9.1. Схема мультиплеерного приложения Unity3D

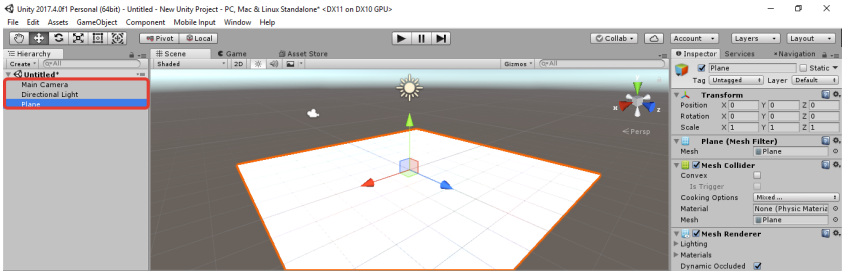


Рис. 9.2. Создание игрового уровня

и запуск мультиплеерных приложений максимально простыми. *NetworkManager* может быть использован практически без написания скриптов. Для начала работы необходимо создать в *Unity3D* новый проект и пустую сцену. Перед тем, как приступить к разработке аватара игрока необходимо подготовить игровой уровень (рис. 9.2). В рамках лабораторной работы достаточно создать плоскость (*Plane*), выступающую в роли поверхности, камеру и источник света (например, *Directional Light*).

После создания уровня можно приступить к созданию аватара игрока. Для начала создадим визуальное представление аватара, например, куб, и разместим его на сцене (рис. 9.3).

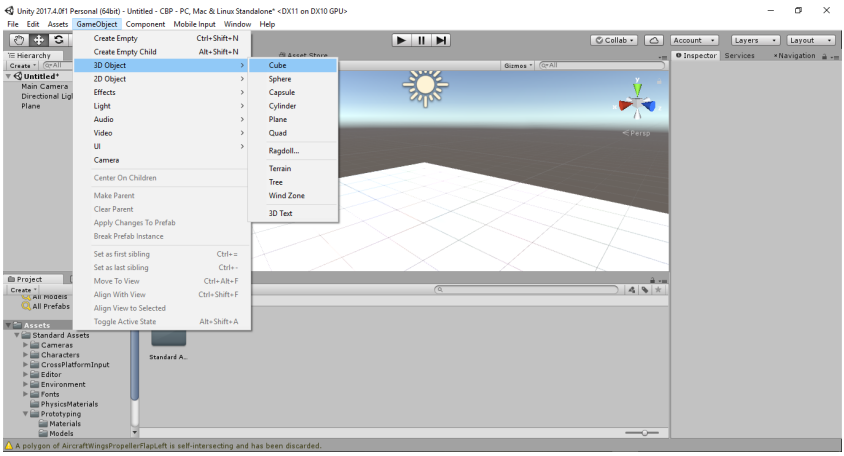


Рис. 9.3. Создание куба

Далее в ресурсах проекта необходимо создать типовой объект «*Player*» (*Assets* → *Create* → *Prefab*) и перетащить куб с панели иерархии (*Hierarchy*) на его иконку. Затем куб можно удалить со сцены и вместо него разместить типовой объект (рис. 9.4).

Теперь приступим к созданию управления аватаром. Создадим скрипт «*PlayerMovement*» (*Assests* → *Create* → *C# Script*) (рис. 9.5).

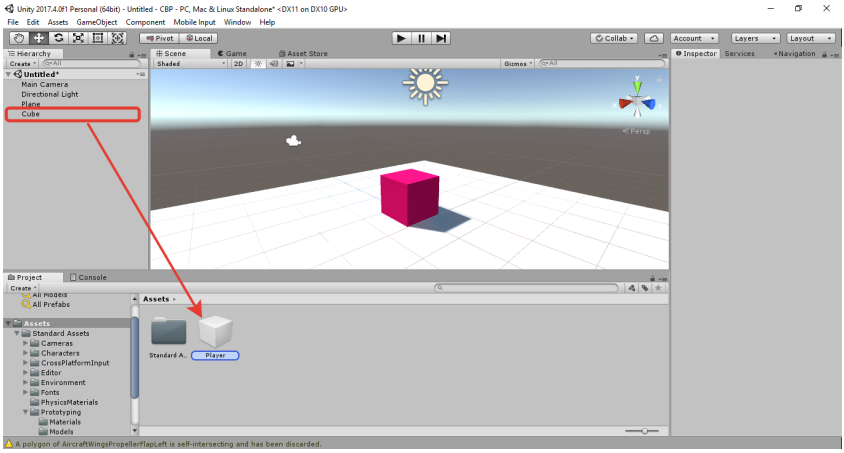


Рис. 9.4. Создание типового объекта

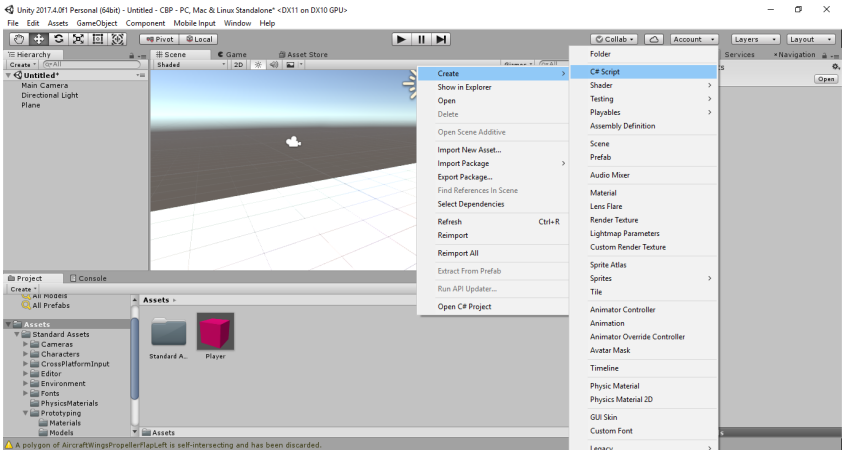


Рис. 9.5. Создание файла C# Script

Открываем скрипт и заменим шаблонный код на следующий:

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;
using UnityEngine.AI;
public class PlayerMovement : NetworkBehaviour
{
    public float speedForward = 2f;
    public float speedBackward = 1f;
    public float speedJump = 5f;
    public float speedTurn = 400f;
    private NavMeshAgent navMeshAgent;
    void Update()
    {
        InputMovement();
    }
    private void InputMovement()
    {
        //walk forward
        if (Input.GetKey(KeyCode.W))
        {
transform.Translate(0, 0, speedForward * Time.deltaTime);
        }
        //run forward
        if (Input.GetKeyDown(KeyCode.LeftShift))
        {
            speedForward *= 2f;
        }
        if (Input.GetKeyUp(KeyCode.LeftShift))
        {
            speedForward *= 0.5f;
        }
        //walk backward
        if (Input.GetKey(KeyCode.S))
        {
transform.Translate(0, 0, -speedBackward * Time.deltaTime);
        }
        //Rotate
        if (Input.GetKey(KeyCode.D))
transform.Rotate(0, speedTurn * Time.deltaTime, 0);
        if (Input.GetKey(KeyCode.A))
transform.Rotate(0, -speedTurn * Time.deltaTime, 0);
        //Jump
        if (Input.GetKey(KeyCode.Space))
        {
transform.Translate(0, speedJump * Time.deltaTime, 0);
        }
    }
}
```

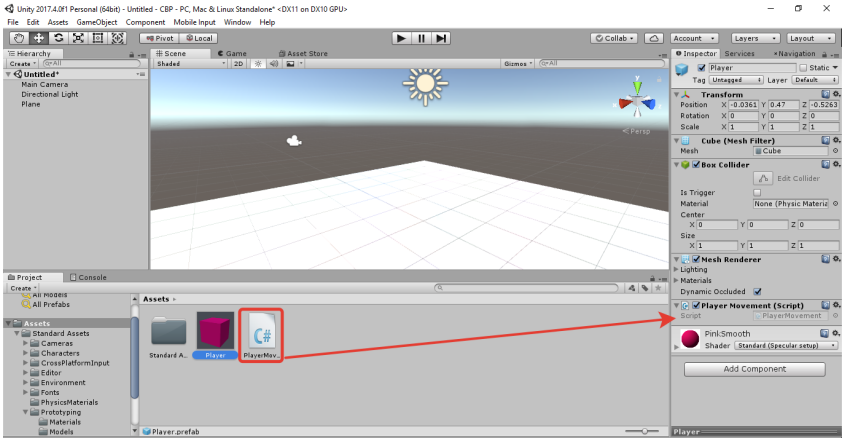


Рис. 9.6. Добавление скрипта к типовому объекту

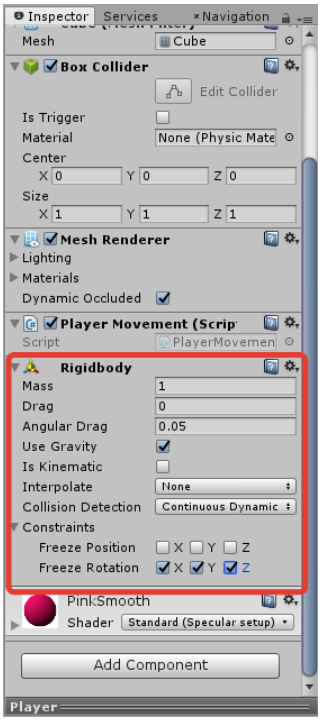


Рис. 9.7. Настройки компонента «Rigidbody»

Теперь скрипт можно добавить к типовому объекту (*Component – Scripts – Player Movement*) (рис. 9.6).

Далее добавляем к типовому объекту компонент «*Rigidbody*» (*Component → Physics → Rigidbody*), подключая тем самым физику к аватару. Настройки компонента приведены на рис. 9.7.

Проверяем работоспособность аватара, запустив сцену в режиме игры. Если аватар «слушается» управления, то можно переходить к разработке многопользовательского режима. В начале лабораторной работы аватар был помещен на сцену вручную, в многопользовательском режиме он должен появляться на сцене при подключении к игре нового пользователя и подчиняться только ему. Для создания многопользовательского режима игры необходимо создать объект, отвечающий за сетевой режим. Создадим пустой объект на сцене (*GameObject → Create Empty*). Назовем его *NetManager*. Доба-

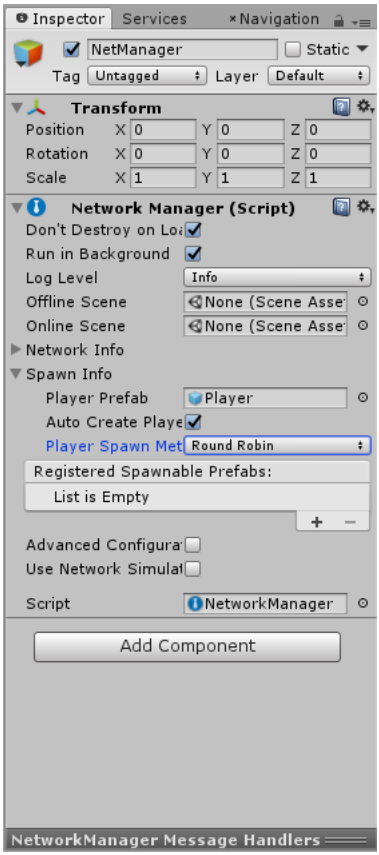


Рис. 9.8. Настройка компонента NetworkManager

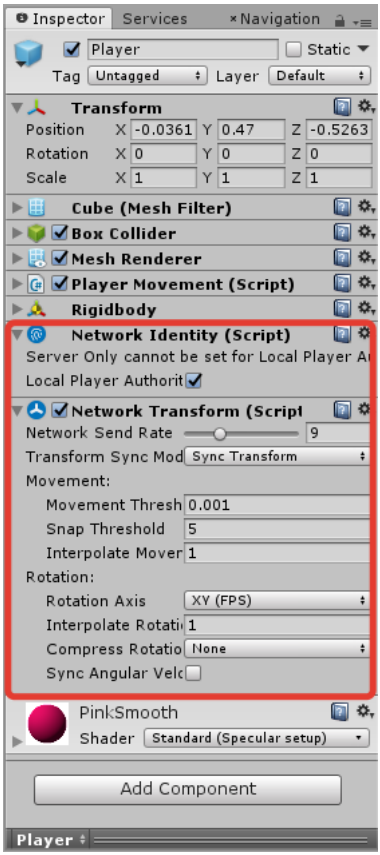


Рис. 9.9. Настройка сетевых компонентов игрока

вим к новому объекту компонент *NetworkManager* и настроим в соответствии с рисунком (рис. 9.8).

В поле *Player Prefab* необходимо добавить префаб игрока, который создается при подключении игрока к серверу. Для этого необходимо отредактировать раннее созданный префаб игрока и добавить компоненты *Network Identity* и *NetworkTransform* и настроить в соответствии с рис. 9.9.

Компонент *Network Identity* отвечает за идентификацию каждого игрока на сервере. Компонент *Network Transform* отправляет данные о перемещении объекта с клиента на сервер. Приведенный

выше скрипт «*PlayerMovement*» в многопользовательском режиме будет работать некорректно, так как отсутствует четкое указание скрипту управлять «своим» аватаром, а также синхронизация между разными аватарами.

Изменим код скрипта на следующий:

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;
public class PlayerMovement : NetworkBehaviour
{
    public float speedForward = 2f;
    public float speedBackward = 1f;
    public float speedJump = 5f;
    public float speedTurn = 400f;
    void Update()
    {
        if (!isLocalPlayer)
        {
            return;
        }
        InputMovement();
    }
    private void InputMovement()
    {
        //walk forward
        if (Input.GetKey(KeyCode.W))
        {
transform.Translate(0, 0, speedForward * Time.deltaTime);
        }
        //run forward
        if (Input.GetKeyDown(KeyCode.LeftShift))
        {
            speedForward *= 2f;
        }
        if (Input.GetKeyUp(KeyCode.LeftShift))
        {
            speedForward *= 0.5f;
        }
        //walk backward
        if (Input.GetKey(KeyCode.S))
        {
transform.Translate(0, 0, -speedBackward * Time.deltaTime);
        }
        //Rotate
        if (Input.GetKey(KeyCode.D))
transform.Rotate(0, speedTurn * Time.deltaTime, 0);
        if (Input.GetKey(KeyCode.A))
transform.Rotate(0, -speedTurn * Time.deltaTime, 0);
```

```
        //Jump
        if (Input.GetKey(KeyCode.Space))
        {
transform.Translate(0, speedJump * Time.deltaTime, 0);
        }
    }
}
```

Для создания стандартного интерфейса мультиплеерной игры необходимо добавить к объекту *NetManager* компонент *Network Manager HUD* (рис. 9.10).

Проверяем получившийся результат (рис. 9.11, 9.12). Для проверки достаточно запустить минимум два приложения одновремен-

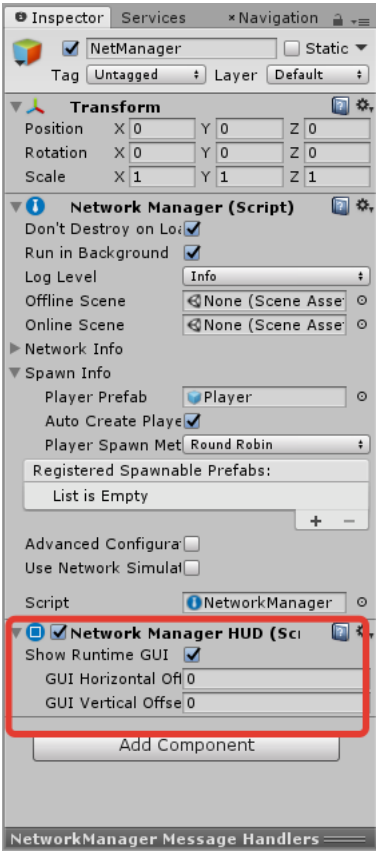
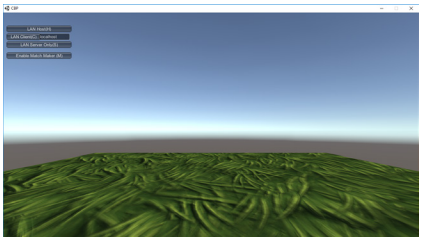


Рис. 9.10. Настройка компонента Network Manager HUD



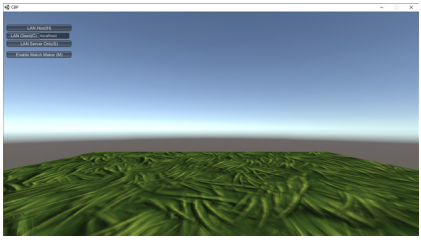
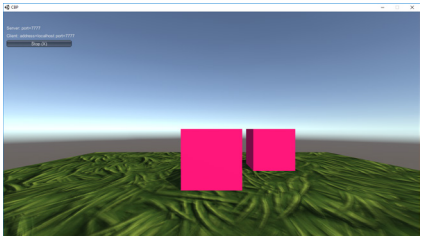


Рис. 9.11. Запущено два экземпляра приложения



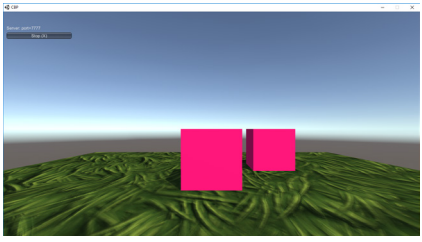


Рис. 9.12. Игроки в режиме Host и Client

но на одном компьютере, для чего проект необходимо собрать (*File → Build Settings...→ Build*). Запускаем приложения, первую игру запускаем в режиме *Host*, вторую в режиме *Client*. Проверяем работу приложения. После запуска двух игр в игре должно отобразиться два игрока. Переключение между окнами позволяет управлять игроками.

Контрольные вопросы

1. Что представляет собой аватар игрока?
2. Какие новые качества добавляет аватару компонент «*Rigidbody*»?
3. Какой скрипт обеспечивает управление аватаром в однопользовательском режиме работы?
4. Какие компоненты *Unity3D* отвечают за появление аватаров администратора и игроков в многопользовательской игре?
5. Что является основой для разработки пользовательского интерфейса?
6. Как проверить работу многопользовательской игры с помощью одного компьютера?

Лабораторная работа № 10
РАЗРАБОТКА ИНТЕЛЛЕКТУАЛЬНОГО АВАТАРА

Цель работы: разработка простого интеллектуального аватара (бота) для игры, реализованной в лабораторной работе № 9.

Порядок выполнения работы

1. Открыть проект, созданный в лабораторной работе № 9.
2. Создать визуальное представление бота, отличное от представления аватара игрока.
3. Добавить компонент «*Rigidbody*», отвечающий за физическое взаимодействие объекта с окружающим миром.
4. «Научить» бота перемещаться по игровому уровню, используя поиск пути с помощью навигационной модели.
5. «Научить» бота реагировать на перемещение игрока.
6. Откорректировать характеристики аватара игрока, опираясь на реакцию бота и сведения, приведенные в разделе «Теоретические сведения».
7. Создать скрипт «*BotAI*», реализующий систему принятия решений и добавить к боту.
8. Для проверки результата работы запустить игру. Если бот преследует близко подошедшего игрока, то лабораторная работа выполнена правильно.
9. Оформить отчет.

Методические указания

Теоретические сведения – общий состав системы игрового искусственного интеллекта, системы восприятия, навигации, принятия решений на основе правил, рассмотрены в [2].

Открываем проект, созданный в результате выполнения лабораторной работы №9. Создаем визуальное представление бота, отличное от такового у аватара игрока. Создаем типовой объект (*Prefab*), перетаскиваем модель с панели иерархии (*Hierarchy*) на иконку типового объекта, удаляем модель.

Таким образом, у нас имеется заготовка бота. Теперь к нему, как и к аватару игрока необходимо подключить компонент «Rigidbody» (*Component* → *Physics* → *Rigidbody*), отвечающий за физическое взаимодействие объекта с окружающим миром. Настройки компонента приведены на рис. 10.1.

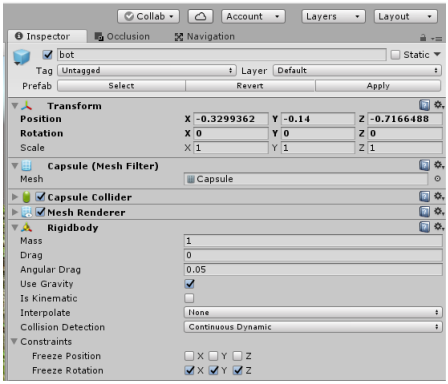


Рис. 10.1. Компонент «Rigidbody»

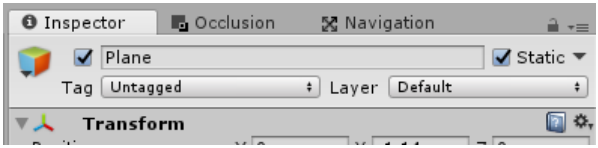


Рис. 10.2. Атрибут «Static»

Следующий шаг – «научить» бота перемещаться по уровню используя поиск пути с помощью навигационной модели. Для этого необходимо подключить компонент «*NavMeshAgent*» к боту (*Component* → *Navigation* → *NavMeshAgent*). Затем создаем саму модель на основе нашей поверхности. Для этого указываем в настройках поверхности атрибут Static (рис. 10.2).

Далее вызываем панель «*Navigaion*» (*Window* → *Navigaion*) и запускаем процесс генерации модели (кнопка *Bake*). Результат должен соответствовать рис. 10.3.

Создадим скрипт «*BotMovement*» и добавим в него следующий код:

```
using UnityEngine;
using System.Collections;

public class BotMovement : MonoBehaviour
{

    private Vector3 movementTarget;
    private UnityEngine.AI.NavMeshAgent navMeshAgent;
    private float distanceToTarget;
```

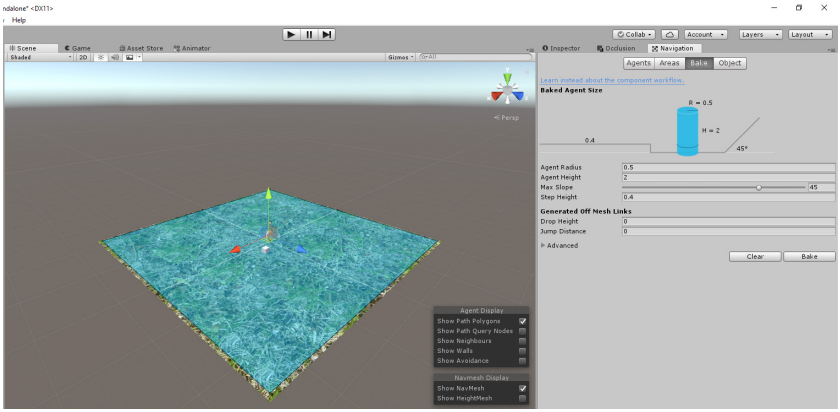


Рис. 10.3. Сгенерированная навигационная модель

```
    public float maxDistanceToTarget = 2.0f;
    private Vector3 initialPosition;

    void Start()
    {
navMeshAgent = GetComponent<UnityEngine.AI.NavMeshAgent>();
        initialPosition = transform.position;
    }

    void Update()
    {
distanceToTarget = Vector3.Distance(movementTarget, transform.
position);
        if (distanceToTarget > maxDistanceToTarget)
        {
            navMeshAgent.SetDestination(movementTarget);
        }
        else
        {
            navMeshAgent.SetDestination(initialPosition);
        }
    }
    public void SetMovementTarget(Vector3 target)
    {
        movementTarget = target;
    }
    public Vector3 getInitialPosition()
    {
        return initialPosition;
    }
}
```

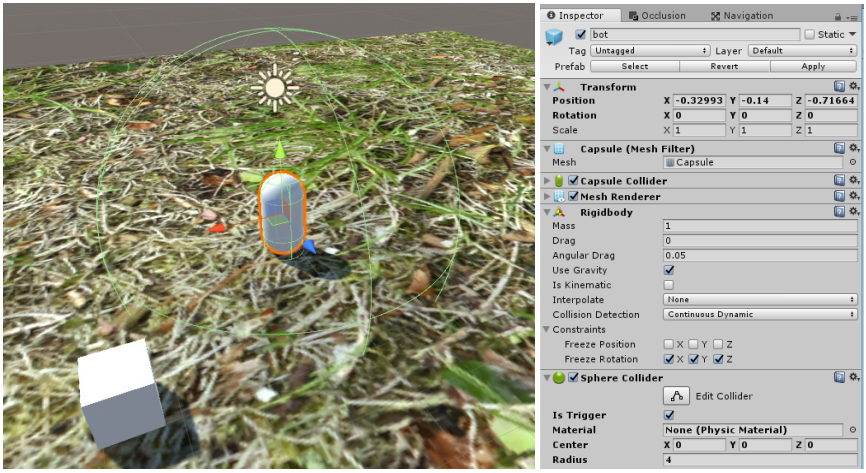


Рис. 10.4. Компонент «Sphere Collider»

Добавляем скрипт к боту. Теперь «научим» бота реагировать на перемещение игрока. Добавим к боту «область реакции» представляющую собой компонент «*Sphere Collider*» (*Component* → *Physics* → *Sphere Collider*) настроенный как триггер (рис. 10.4). Зона реакции бота на игрока настраивается параметром «*Radius*». Для корректной работы значение устанавливается ~4.

Далее создаем новый скрипт «*BotTrigger*», определяющий присутствие аватара игрока через зону реакции, и так же добавим к боту:

```
using UnityEngine;
using System.Collections;
public class BotTrigger : MonoBehaviour
{
    private BotAI AI; private Transform intruder;
    void Start()
    {
        AI = GetComponent<BotAI>();
    }
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            AI.setDisturbedState(true);
            intruder = other.transform;
        }
    }
```

```
    private void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            AI.setDisturbedState(false);
        }
    }
    public Transform getIntruder()
    {
        return intruder;
    }
}
```

Как видно из кода скрипта, бот «реагирует» только на объекты с тегом «*Player*».

И последний шаг – создание скрипта «*BotAI*», реализующего систему принятия решений:

```
using UnityEngine;
using System.Collections;
public class BotAI : MonoBehaviour
{
    private BotTrigger trigger;
    private BotMovement movement;
    private bool stateWounded;
    private bool stateDisturbed;
    void Start()
    {
        trigger = transform.GetComponent<BotTrigger>();
        movement = transform.GetComponent<BotMovement>();
        stateDisturbed = false;
    }
    void Update()
    {
        if (stateDisturbed)
        {
            Transform intruder = trigger.getIntruder();
            movement.SetMovementTarget(intruder.position);
        }
        if (!stateDisturbed)
        {
            movement.SetMovementTarget(movement.
getInitialPosition());
        }
    }
    public void setDisturbedState(bool state)
    {
        stateDisturbed = state;
    }
}
```

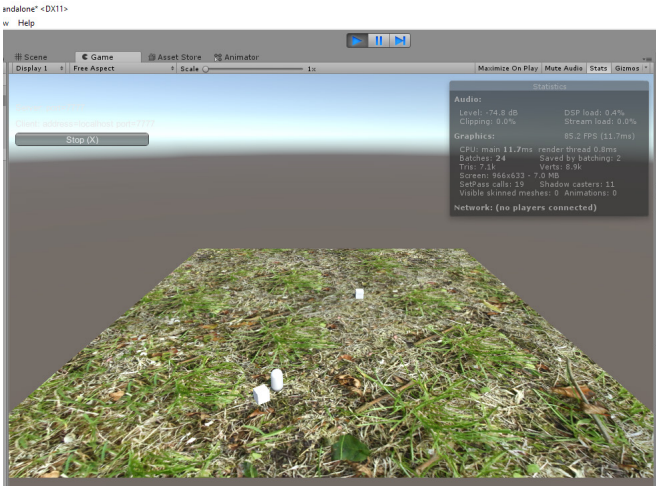


Рис. 10.5. Результат преследования

Добавляем скрипт к боту и запускаем игру (рис. 10.5). Если бот преследует близко подошедшего игрока, то лабораторная работа выполнена правильно.

Контрольные вопросы

1. Что представляет собой интеллектуальный аватар – бот?
2. Какие новые качества добавляет боту компонент «*Rigidbody*»?
3. Какой компонент *Unity3D* задает «область реакции» бота?
4. Какой скрипт обеспечивает интеллектуальное поведение бота?
5. Как проверить реакцию бота на действия аватара игрока?

ЛАБОРАТОРНАЯ РАБОТА № 11
ОЗНАКОМЛЕНИЕ С ДЕРЕВЬЯМИ СМЕШИВАНИЯ АНИМАЦИИ

Цель работы: ознакомиться с *Blend Trees* (деревья смешивания); создать простую тестовую сцену в *Unity3D*.

Порядок выполнения работы

1. Ознакомиться с содержанием раздела «Теоретические сведения».
2. Создать несколько avatarov с записанной анимацией, созданных при помощи сервиса *Mixamo*.
3. Создать в *Unity3D* новый проект и пустую сцену.
4. Импорт заготовки анимации в *Unity3D*.
5. Подготовить игровой уровень.
6. Создание меню с помощью *Canvas* для взаимодействия с приложением.
7. Подключение *C# Script «RunAnim»* к *Canvas*.
8. Создание нескольких панелей и наполнение их кнопками согласно иерархии.
9. Для проверки результата выполнить компиляцию проекта (проект необходимо собрать как *File* → *Build Settings*...→ *Build*) и запустить приложение для демонстрации результата.
10. Демонстрация результата.
11. Оформить отчет.

Рекомендации по содержанию тестовой сцены

Игровой уровень должен содержать:

– Плоскость (*Plane*), выступающую в роли поверхности;

– Камеру (*main Camera*);

– Источник света (например, *Directional Light*);

– Несколько анимационных клипов с записанной анимацией, созданных при помощи сервиса *Mixamo*;

– Панель «Canvas» для взаимодействия с персонажем при помощи кнопок и слайдера.

Методические указания

Blend Trees (Деревья смешивания)

Одной из распространённых задач в игровой анимации является смешивание двух или более похожих анимаций. Возможно, наибо-

лее известный пример, это смешивание анимаций шага и бега, в соответствии со скоростью персонажа. Другим примером является наклон персонажа влево или вправо в случае поворота во время бега.

Важно различать переходы (*Transition*) и *Blend Trees*. Хотя они оба используются для создания плавных анимаций, но применяются для разных видов ситуаций.

Transitions используется для плавного перехода от одной анимации к другой за заданное время. Переходы указаны как часть *Animation State Machine*. Переход от одного движения к абсолютно другому обычно протекает нормально, если переход происходит быстро.

Blend Trees используется для плавного смешивания нескольких анимаций путём объединения их всех в разной степени. Значение, которое каждое движение вкладывает в финальную анимацию, управляется параметром смешивания, который является одним из числовых параметров анимации связанных с *Animator Controller*. Чтобы смешанное движение выглядело осмысленно, смешиваемые движения должны быть примерно равные по длине и природе. *Blend Trees* – это особый вид анимации в *Animation State Machine*.

Примерами подобных движений могут быть анимации ходьбы и бега. Для того, чтобы смешивание работало нормально, перемещения в клипах должны происходить на том же самом месте при нормализованном времени. Например, анимации ходьбы и бега могут быть привязаны так, что в момент контакта ноги с полом, они остаются на месте при нормализованном времени (например, левая нога попадает на 0.0, а правая на 0.5). Т.к. используется нормализованное время, то длительность клипов роли не играет.

Импорт анимационных клипов

Анимационные клипы – это крошечные строительные блоки анимации в *Unity*. Они представляют собой отдельные элементы движения, такие как *RunLeft* (бег в левую сторону), *Jump* (прыжок) или *Crawl* (ползание) и могут быть объединены разными способами, образуя живые и реалистичные анимации. Анимационные клипы можно выбрать в данных, импортированных из *FBX* (рис. 11.1).

Mixamo – это сервис от *Adobe*. Этот сервис позволяет анимировать собственных персонажей или же воспользоваться уже созданной библиотекой данного сервиса. Для этого нам нужно выбрать уже заготовленные анимационные клипы и экспортировать их на локальный компьютер. Далее импортируем заготовки в любой проект *Unity* (рис. 11.2).

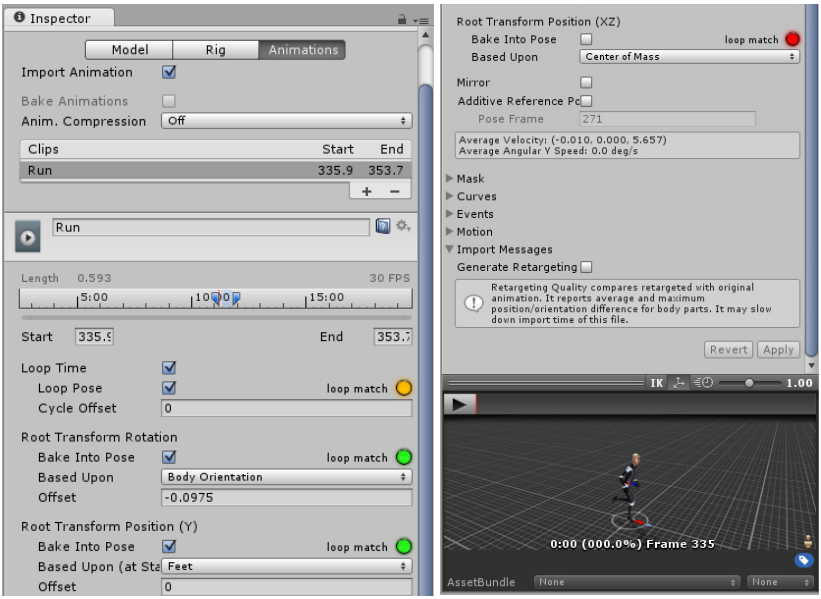


Рис. 11.1. Свойства анимационного клипа

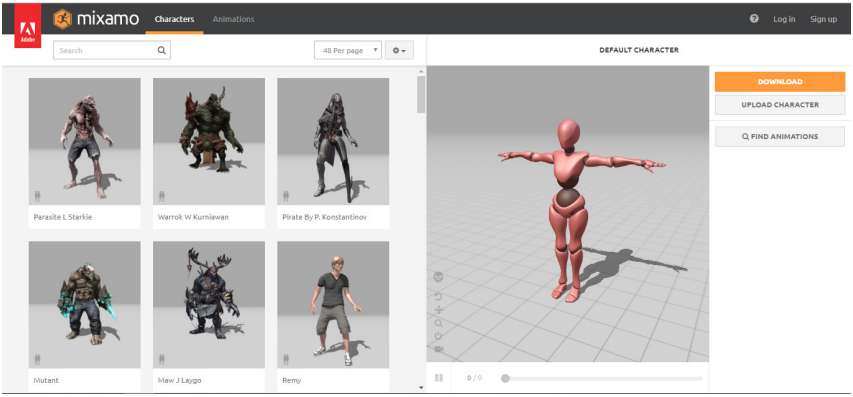


Рис. 11.2. Сервис Mixamo

Перед работой с древом смешивания необходимо ознакомиться с теоретическими сведениями данной лабораторной работы, после чего приступаем к установке определенных параметров *Animation Type* на *Humanoid* (рис. 11.3).

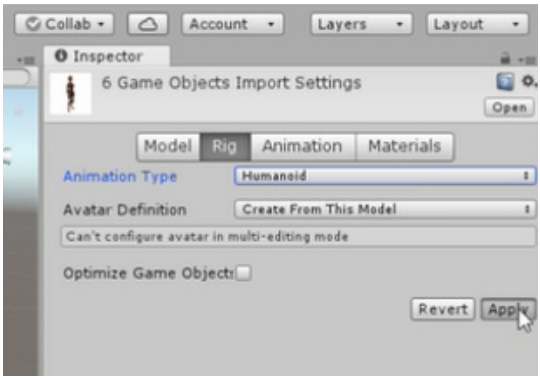


Рис. 11.3. Тип анимации Humanoid

Так же необходимо заморозить смещение клипов и зациклить анимацию. Измененные параметры, анимационного клипа:

1. *Loop time* – Зацикливание анимации;

2. *Root Transform Rotation* – корневое вращение в движение костей;

3. *Root Transform Position (Y)* – чтобы заморозить вертикальное смещение костей;

4. *Root Transform Position (XZ)* – чтобы заморозить горизонтальное смещение костей.

При применение этих параметров, в зависимости от действий, происходящих в клипе, необходимо обращать внимание на смещения анимационного клипа (*Vector3*). На рис. 11.4 приведен пример анимационного клипа «Спокойствие», следовательно, *Vector3* должен иметь координаты (0,0,0).

Теперь приступаем к созданию сцены в *Unity 3D*. Создадим освещение, используя *Directional Light*, и ландшафт, при помощи добавления стандартной поверхность *Plane* (рис. 11.5).

Перемещаем любой префаб анимационного клипа на сцену (рис. 11.6). Позиционируем аватара на центральную дорожку и добавляем компоненты:

– *Animator* (заполняем переменную ранее созданным пустым *Animator Controller*);

– *Capsule collider*;

– *Rigidbody* (замораживаем повороты по всем осям).

Создадим ключевые элементы управления сценой на которые будем ссылаться из *C#* скрипта. Переходим к настройке *Animation*

Рис. 11.4. Параметры анимации

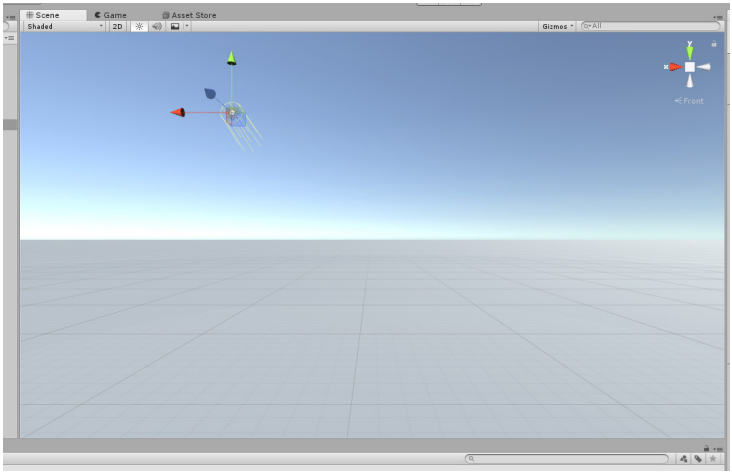


Рис. 11.5. Поверхность со светом

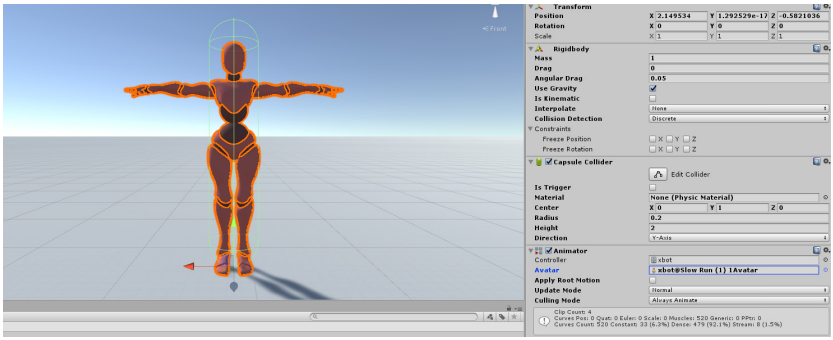


Рис. 11.6. Персонаж на сцене

Таблица 11.1

Логический порядок анимационных клипов

Анимационный клип	Параметр «run»	Параметр «strafe»	Параметр Mirror
Спокойствие	0	0	false
Шаг	0,5	0	false
Бег	1	0	false
Шаг вправо	–	1	false
Шаг влево	–	–1	true

Примечание: для движения (шага) в бок использован один и тот же анимационный клип (в одном из случаев установлен параметр зеркального отражения).

Controller. Создаем новое *blend tree* и новые параметры управления анимации типа *float*. Называем параметры «*run*» и «*strafe*». Далее добавляем слоты под анимацию в древе смешивания и подставляем клипы в логическом порядке по осям. Порядок представлен в табл. 11.1. За ось *X* отвечает параметр «*run*», а за ось *Y* параметр «*strafe*». На рис. 11.7 приведен пример настройки древа смешивания.

После создание сцены необходимо разработать «Меню управления», с помощью которого можно будет взаимодействовать с приложением.

В качестве инструмента для разработки усложненного интерфейса был использован «*Canvas*». Компонент *Canvas* представляет собой абстрактное пространство, в котором производится настройка и отрисовка *UI*. Все *UI*-элементы являются потомками игровых объектов, к которым присоединен *Canvas*.

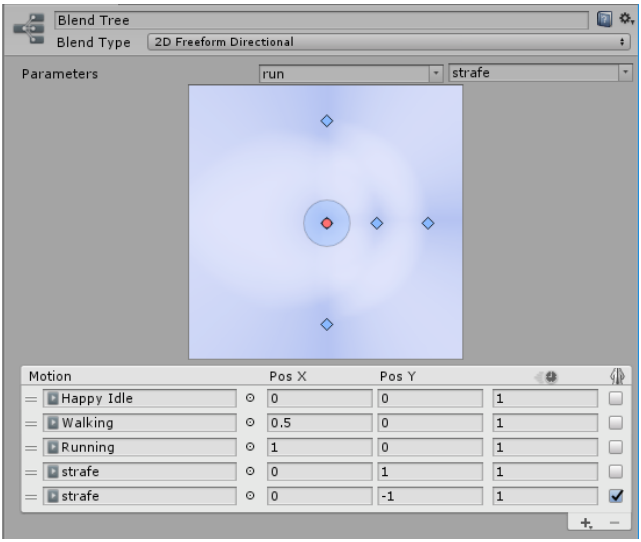


Рис. 11.7. Настроенное Blend Tree

К *Canvas* добавим новый компонент *C# Script «Anim»* и заполним его переменные. Содержимое скрипта представлено ниже:

```
using UnityEngine;
using UnityEngine.UI;

public class Anim : MonoBehaviour
{
    [Range(0, 1)]
    public float run;
    [Range(-1, 1)]
    public float strafe;
    public Slider slider;
    public Animator animator;
    [Range(0, 5)]
    public float dampTime;
    private void Start()
    {
        dampTime = 0.15f;
    }
    void Update()
    {
        run = slider.value;
animator.SetFloat("run", run, dampTime, Time.deltaTime);
animator.SetFloat("strafe", strafe, dampTime, Time.deltaTime);
    }
```

```
    public void ButtonReset()
    {
        run = 0;
        strafe = 0;
Transform avatar = animator.GetComponent<Transform>();
        avatar.position = Vector3.zero;
    }
    public void ButtonLeft()
    {
        if (strafe < 1 && !run.Equals(0))
            strafe += 0.5f;
    }

    public void ButtonRigth()
    {
        if (strafe > -1 && !run.Equals(0))
            strafe -= 0.5f;
    }
}
```

Создадим панель и наполним ее кнопками согласно иерархии, которая представлена на рис. 11.8.

На примере показано соотношение панелей согласно экрану (рис. 11.9).

В настройках кнопки выбираем соответствующие функции (рис. 11.10).

Запускаем приложение и наблюдаем результат (рис. 11.11).

Теперь, когда сцена и интерфейс созданы, можно перейти к компиляции проекта.

Для компиляции проекта необходимо: *«File»* → *«Buil Settings...»*. После чего откроется меню настроек (рис. 11.12).

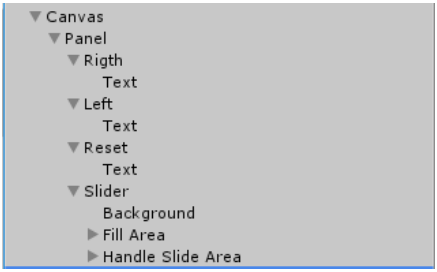


Рис. 11.8. Иерархия интерфейса

Рис. 11.9. Расположение панелей согласно экрану

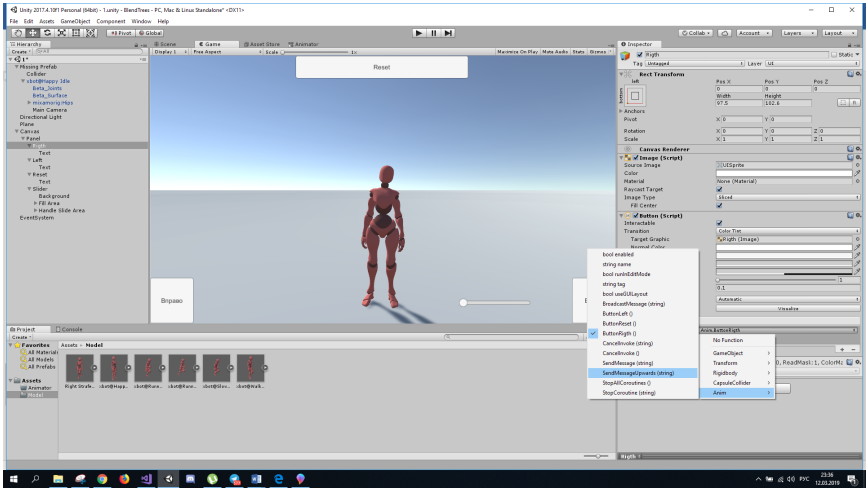


Рис. 11.10. Добавление функции для кнопки

В данном разделе необходимо выбрать платформу (*Windows*) и архитектуру (x86_64). Далее нажимаем *«Build»* и ожидаем окончания компиляции. После чего запускаем скомпилированный проект (рис. 11.13).

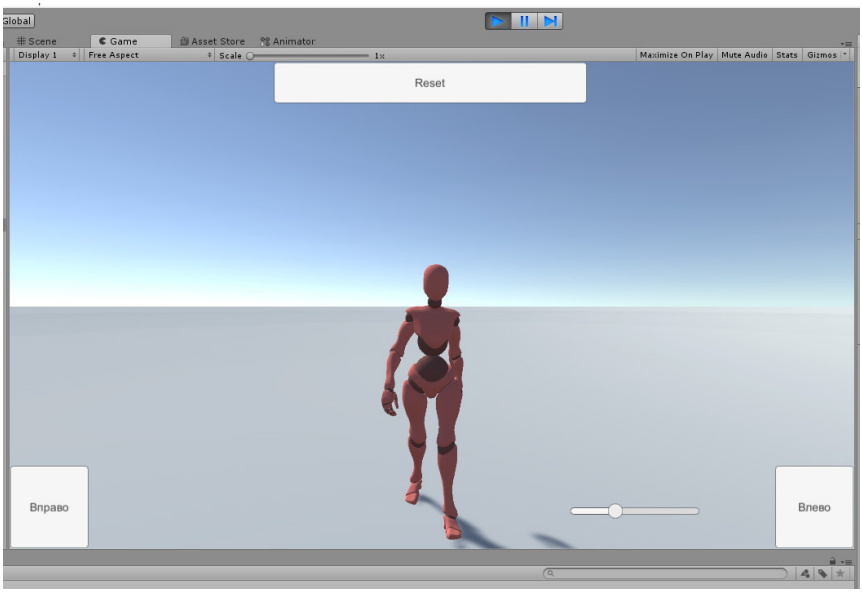


Рис. 11.11. Режим «Play»

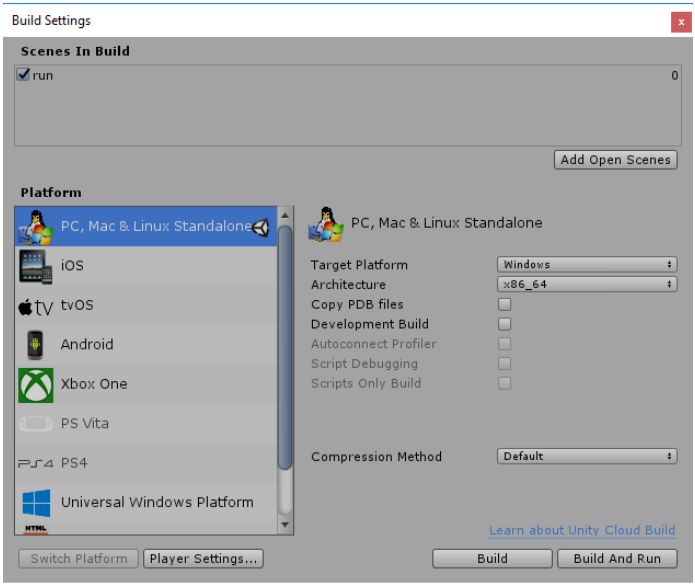


Рис. 11.12. Окно настроек компиляции проекта

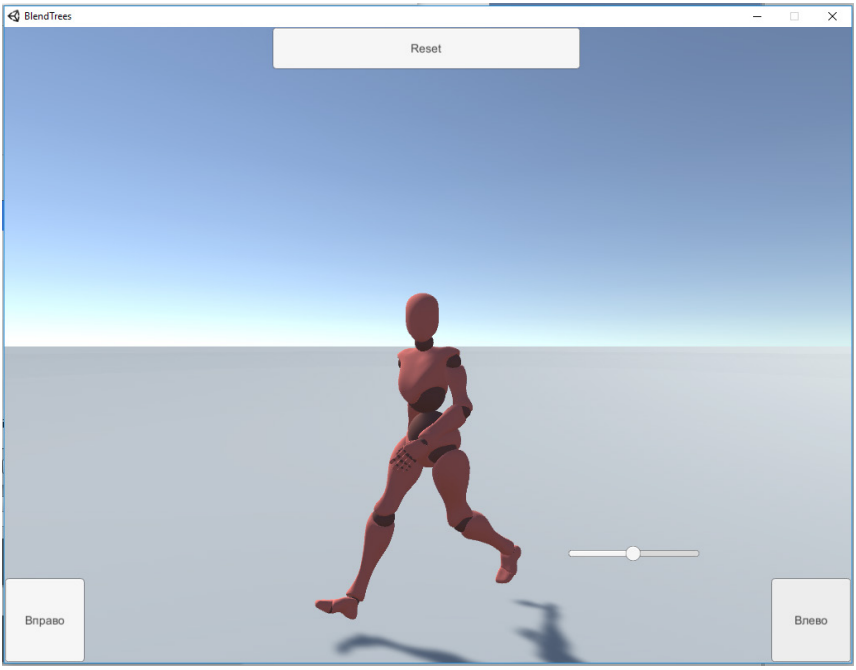


Рис. 11.13. Результат работы

Контрольные вопросы

1) Что понимается под термином *Blend Trees?*

2) Чем *Blend Trees* отличается от *Transition?*

3) Что такое «анимационные клипы» и для чего они используются?

4) Для чего нужен сервер *Mixamo?*

Лабораторная работа № 12
СОЗДАНИЕ WINDOWS ПРИЛОЖЕНИЯ ДОПОЛНЕННОЙ РЕАЛЬНОСТИ С ПОМОЩЬЮ БИБЛИОТЕКИ VUFORIA

Цель работы: разработка приложения дополненной реальности с использованием инструментов *Unity3D* и библиотеки *Vuforia Engine* для *Unity3D*.

Порядок выполнения работы

1. Установка библиотеки *Vuforia Engine*.
2. Создание и распознавание мишени (маркера).
3. Создание приложения дополненной реальности (ДР).
4. Проверка приложения ДР.
5. Оформление отчета.

Методические указания

Для выполнения работы необходим доступ в интернет и наличие *web*-камеры.

Vuforia Engine – это программная платформа для создания приложений дополненной реальности. Разработчики могут легко добавить расширенную функциональность компьютерного зрения в любое приложение, позволяя ему распознавать изображения и объекты, а также взаимодействовать с пространствами в реальном мире.

Платформа *Vuforia Engine* поддерживает разработку приложений *AR* для устройств *Android, iOS* и *UWP*. Начиная с 2017 версии, *Vuforia Engine* интегрирована в *Unity3D*.

Установка Vuforia

Загрузите и запустите *Unity Download Assistant* 2017.2 или более позднюю версию с веб-сайта *Unity*: https://unity3d.com/ .

Следуя шагам установщика «*Download And Install Unity*» в диалоге выбора компонентов выберите дополнение *Vuforia Augmented Reality Support*. Затем продолжайте установку (рис. 12.1).

Для начала работы с библиотекой необходимо создать новый *3D* проект в *Unity* (рис. 12.2).

Vuforia Engine в Unity3D

Vuforia Engine будет отображаться в меню игровых объектов *Unity*, а также в настройках сборки и настройках игрока. Компоненты *Vuforia* находятся на вкладке *GameObject* → *Vuforia* (рис. 12.3).

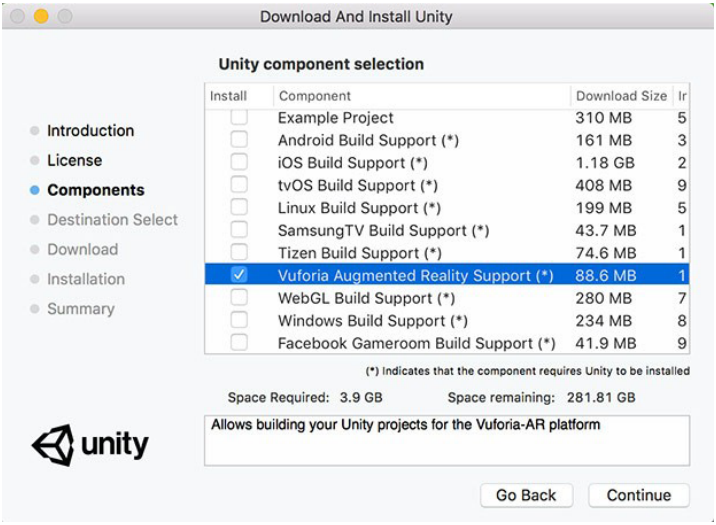


Рис. 12.1. Установка Vuforia Engine

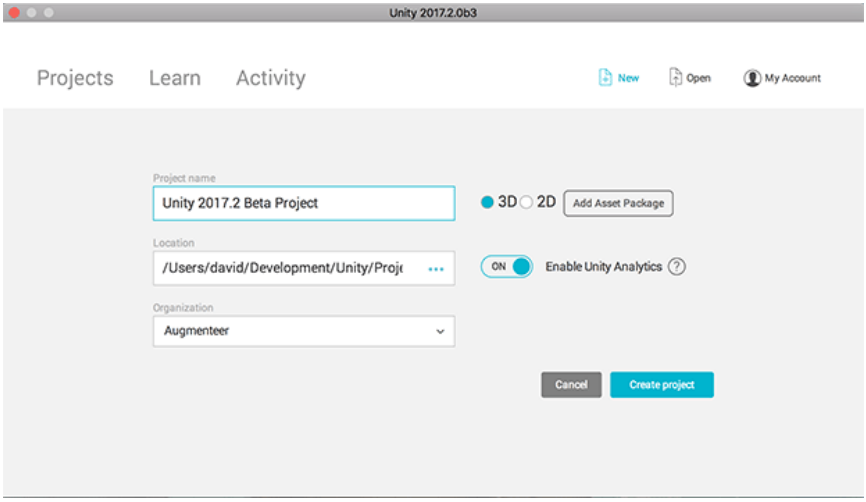


Рис. 12.2. Создание нового проекта

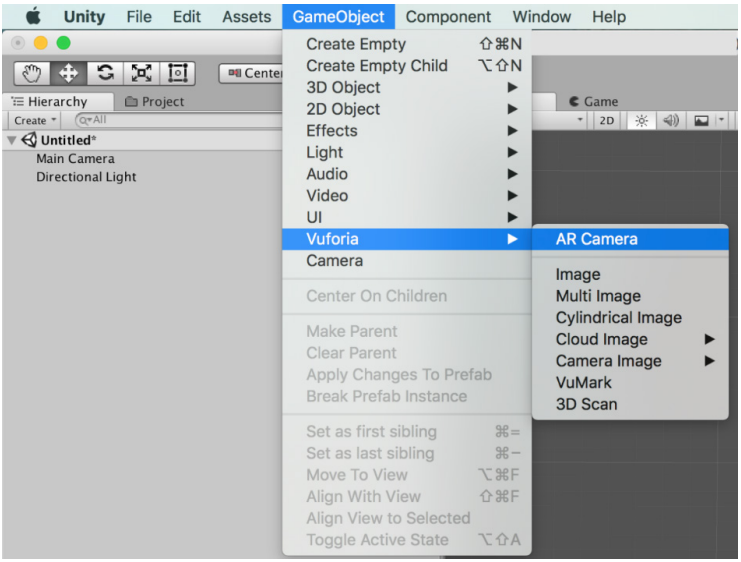


Рис. 12.3. Библиотека Vuforia в Unity 2017.2 и выше

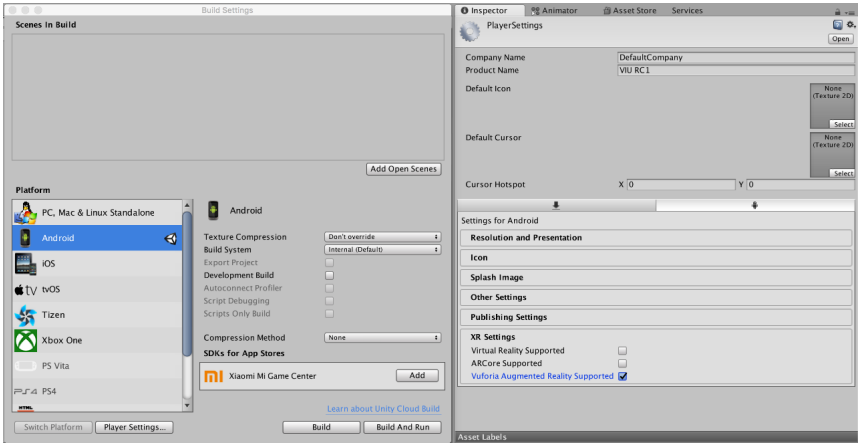


Рис. 12.4. Активация библиотеки Vuforia Engine

Активация Vuforia Engine

Для сборки приложений с библиотекой *Vuforia Engine*, необходима активация. Так же возможно использование *Vuforia Engine* только в режиме Play. Чтобы активировать *Vuforia Engine* перейди-

те в настройки инспектора, в разделе «Настройки *XR*» установите флажок «Дополненная реальность *Vuforia» (Player Settings → XR Settings → Vuforia Augmented Reality)* **(рис. 12.4).**

Начало работы с Vuforia Engine в Unity3D

После активации *Vuforia Engine* в *Unity* можно добавить в проект функции *Vuforia Engine* из меню *Unity → GameObject*.

Основные функции:

– *Model Targets:* позволяют распознавать объекты по форме с использованием уже существующих *3D*-моделей. Для размещения *AR* на самых разных предметах, таких как промышленное оборудование, транспортные средства, игрушки и бытовая техника.

– *Image Targets:* распознавание плоских изображений, таким как печатные носители и упаковка продукта.

– *Object Targets:* создаются путем сканирования объекта. Они являются хорошим вариантом для игр и других продуктов с богатыми деталями поверхности и постоянной формой.

– *Cylinder Targets:* создаются с использованием нескольких целевых изображений и могут быть упорядочены в правильные геометрические фигуры (например, прямоугольники) или в любое произвольное расположение плоских поверхностей.

– *Multi-Targets:* для распознавания изображения, нанесенного на объекты, имеющие приблизительно цилиндрическую форму (например, бутылки с напитками, кофейные чашки, банки с газировкой).

– *VuMarks:* это настраиваемые маркеры, которые могут кодировать различные форматы данных. Они поддерживают как уникальную идентификацию, так и отслеживание для приложений *AR*.

– *External Camera:* доступ к видеоданным с камеры, находящейся вне камеры телефона или планшета, при создании *AR*.

– *Ground Plane:* позволяет размещать контент на горизонтальных поверхностях в среде, такой как столы и полы.

Для начала необходимо добавить *ARCamera* (специальный тип камеры, который поддерживает приложения дополненной реальности, как для портативных устройств, так и для цифровых очков):

Добавьте *ARCamera*. Удалите основную камеру (рис. 12.5) Камера находится в *GameObject → Vuforia → ARCamera*.

1. Создание мишени.

Для этого на сайте https://developer.vuforia.com/ необходимо создать файл с мишенями для *ImageTarget*. Изображения для мишеней можно взять своё или скачать из интернета.

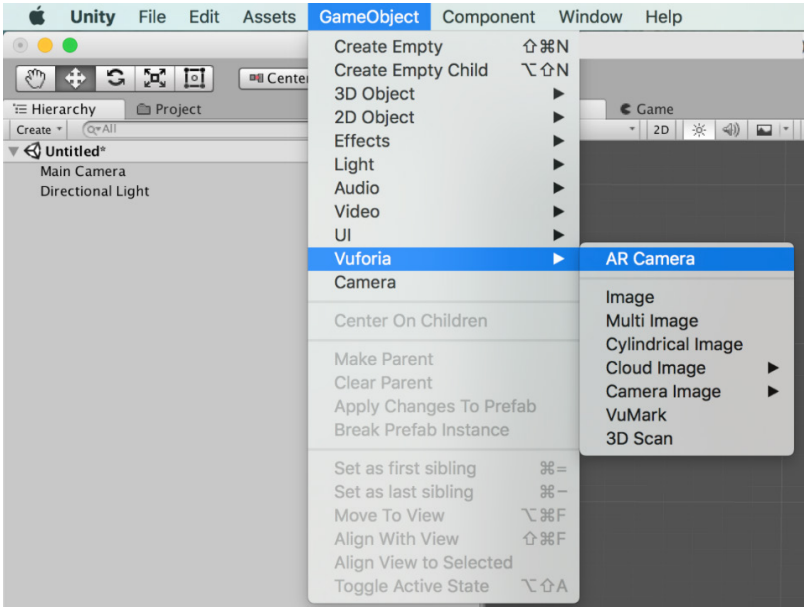


Рис. 12.5. Добавление ARCamera

Для нормальной работы *Image Target*, нужно чтобы на сайте при добавлении в мишень изображения выдавало не менее 2 звёздочек.

Хорошими изображениями для мишеней являются те, в которых много контрастных деталей. Именно на этих деталях и строится опорная матрица для последующего распознания мишеней.

Порядок создания файла с мишенями:

1. Зайти на сайт – https://developer.vuforia.com/
2. Создать свой аккаунт или получить его у преподавателя.
3. Перейти в *Target Manager*.
4. С помощью кнопки «*Create Database*» создать базу для мишеней (*Target*).
5. Войти в созданную базу.
6. Для создания новой мишени нажать на кнопку «*Add Target*».
7. Выбрать тип мишеней (например, *Image Target*)
8. Задать параметры мишени и нажать кнопку «*Add*»
9. Для загрузки мишени нужно её выделить (для выделения нужно поставить галочку у нужной мишени или у нескольких мишеней).

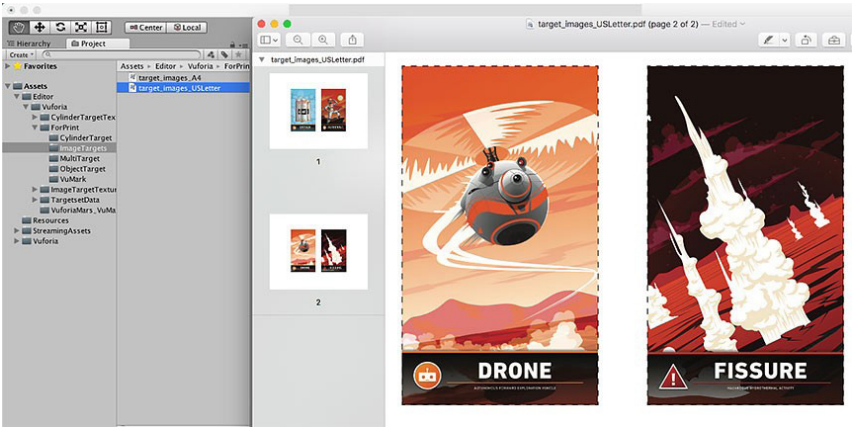


Рис. 12.6. Стандартные цели библиотеки Vuforia Engine

10. Нажать кнопку «*Download Selected Targets*», выбрать *Unity Editor*, ввести название создаваемого файла и нажать кнопку «*Create*».

Файл с мишенями подключается через подключение пакетов.

Рассмотрим способ работы, с уже готовыми мишенями из библиотеки *Vuforia Engine*, которые находиться в / *Editor* / *Vuforia* / *ForPrint* (рис. 12.6), откуда их можно распечатать, или взять уже готовые у преподавателя.

2. Добавление цели на сцену.

Чтобы добавить цели *Vuforia* к своей сцене, необходимо выбрать соответствующие игровые объекты в меню *GameObject* → *Vuforia*. Целевой игровой объект будет добавлен в иерархию вашей сцены.

Каждый целевой объект может быть настроен в Инспекторе. Для настройки целевого объекта необходимо в инспекторе выбрать базу данных и имя цели с помощью который будет происходить распознавание. (рис. 12.7).

3. Добавление модели к цели.

Для добавления модели достаточно, добавить ее как дочерний объект цели в иерархию сцены. Содержание родительского объекта с целевым объектом автоматически устанавливает необходимые параметры рендеринга и физики (см. *DefaultTrackableEventHandler.cs*) (рис. 12.8).

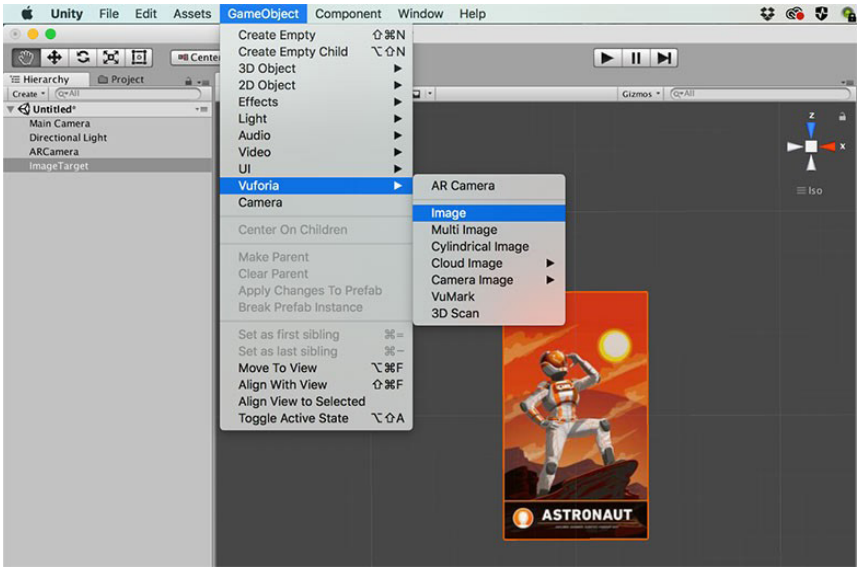


Рис. 12.7. Добавление целина сцену

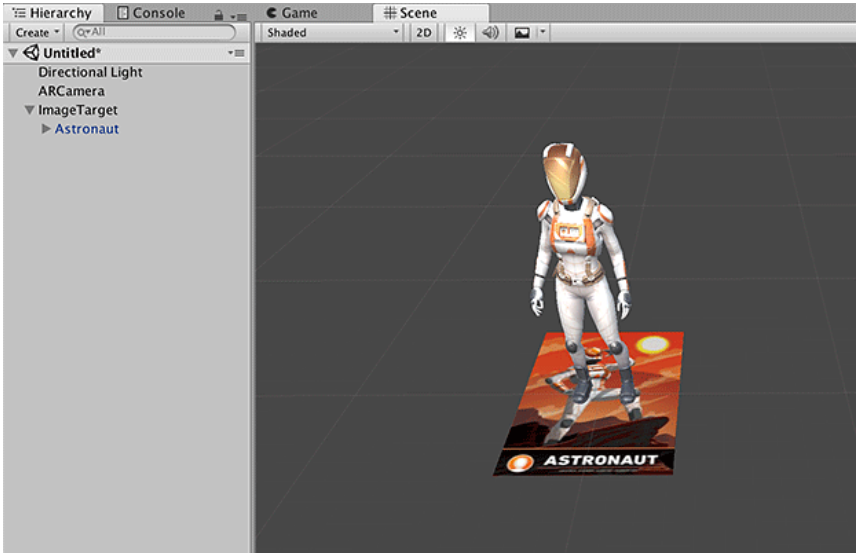


Рис. 12.8. Добавление модели к цели

4. Проверка работоспособности

Vuforia Engine предоставляет симулятор в режиме «*Play*», который можно активировать, нажав на соответствующею кнопку. Эту функцию можно использовать для оценки и быстрого создания прототипов сцены без необходимости установки на устройстве. Режим воспроизведения настраивается в разделе веб-камеры конфигурации *Vuforia*.

Сборка и запуск приложения

Приложения *Unity* на базе библиотеки *Vuforia* создаются и работают так же, как и другие приложения *Unity* для *Android*, *iOS* и *UWP*.

Контрольные вопросы

1. Как подключается библиотеки *Vuforia*?
2. Как создается и распознается мишень (маркер)?
3. Основные этапы создания приложения дополненной реальности.
4. Как проверяется приложение дополненной реальности?

Лабораторная работа № 13
СОЗДАНИЕ WINDOWS ПРИЛОЖЕНИЯ ДОПОЛНЕННОЙ РЕАЛЬНОСТИ С ПОМОЩЬЮ БИБЛИОТЕКИ VUFORIA ДЛЯ MICROSOFT HOLOLENS

Цель работы: разработка приложения дополненной реальности с использованием инструментов *Unity3D* и библиотек *Qualcomm Vuforia* и *MRT* для *Unity3D*.

Порядок выполнения работы

1. Установка библиотеки *Qualcomm Vuforia*.
2. Создание и распознавание маркера.
3. Создание приложения дополненной реальности для *Vuforia*.
4. Тестирование приложения.
5. Установка библиотеки *MRT*.
6. Настройка *Unity* для работы с *Hololens*.
7. Создание приложения дополненной реальности для *Hololens*.
8. Тестирование приложения.
9. Оформление отчета;

Методические указания

Qualcomm Viforia

Vuforia – это платформа дополненной реальности и инструментарий разработчика программного обеспечения дополненной реальности (*Software Development Kit – SDK*) для мобильных устройств, разработанные компании *Qualcomm*. *Vuforia* использует технологии компьютерного зрения, а также отслеживания плоских изображений и простых объёмных реальных объектов (к примеру, кубических) в реальном времени. С версии 2.5 *Vuforia* распознаёт текст, а с 2.6 – имеет возможность распознавать цилиндрические маркеры.

Возможность регистрации изображений позволяет разработчикам располагать и ориентировать виртуальные объекты, такие, как *3D*-модели и медиаконтент, в связке с реальными образами при просмотре через камеры мобильных устройств. Виртуальный объект ориентируется на реальном образе так, чтобы точка зрения наблюдателя относилась к ним одинаковым образом для достижения главного эффекта – ощущения, что виртуальный объект является частью реального мира.

Vuforia поддерживает различные *2D*- и *3D*-типы мишеней, включая безмаркерные *Image Target*, трёхмерные мишени *Multi-Target*, а также реперные маркеры, выделяющие в сцене объекты для их распознавания. Дополнительные функции включают обнаружение преград с использованием так называемых «Виртуальных кнопок» («*Virtual Buttons*»), детектирование целей и возможность программно создавать и реконфигурировать цели в рамках самомодифицирующегося кода.

Vuforia предоставляет интерфейсы программирования приложений на языках *C++*, *Java*, *Objective-C*, и *.Net* через интеграцию с игровым движком *Unity*. Таким образом *SDK* поддерживает разработку нативных *AR*-приложений для *iOS* и *Android*, в то же время предполагая разработку в *Unity*, результаты которой могут быть легко перенесены на обе платформы.

Microsoft HoloLens

Microsoft HoloLens– это очки дополнительной реальности, которые умеют создавать «голографические» изображения приложений, игр, звонков из Skype и т. д. с помощью технологии дополненной реальности. *HoloLens* это что-то среднее между очками *Google Glass* и шлемом виртуальной реальности *Ocultus Rift*. *HoloLens* могут превратить обычную стену в большой киноэкран или проекцию рабочего стола, комнату превратить в песочницу *Minecraft*, облепить холодильник виртуальными записками и т. д.

Для начала выполнения работы необходимо выполнить регистрацию на сайте: https://developer.vuforia.com/, далее перейти во вкладку «*Develop*» и создать уникальный ключ (рис. 13.1, 13.2).

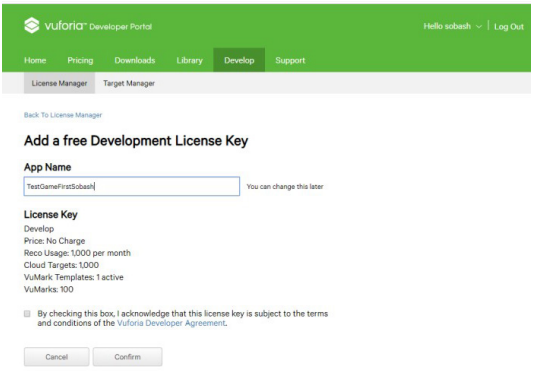


Рис. 13.1. Создание уникального ключа

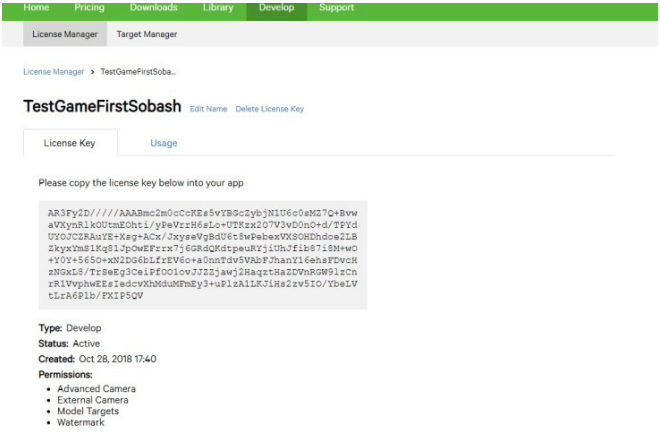


Рис. 13.2. Созданный ключ

Теперь добавляем изображение, которое будет использоваться как маркер, для этого необходимо создать базу и загрузить исходное изображение (рис. 13.3, 13.4).

Примечание:

Стоит отметить, что размер изображения не должен превышать более 2 Мб, так как оно не будет загружено. Так же необходимо выбирать изображение у которых будет как можно больше точек (маркеров). Количество точек на изображение можно посмотреть после его загрузки. Но и при максимальном рейтинге, могут возникнуть проблемы (рис. 13.5).

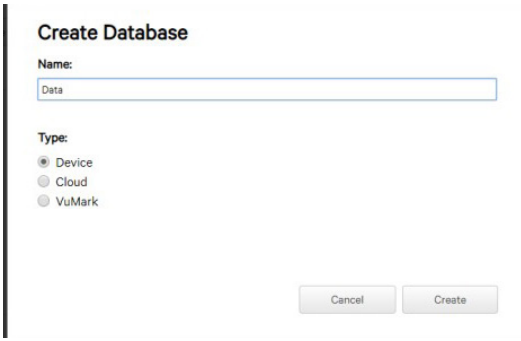


Рис. 13.3. Создание базы данных

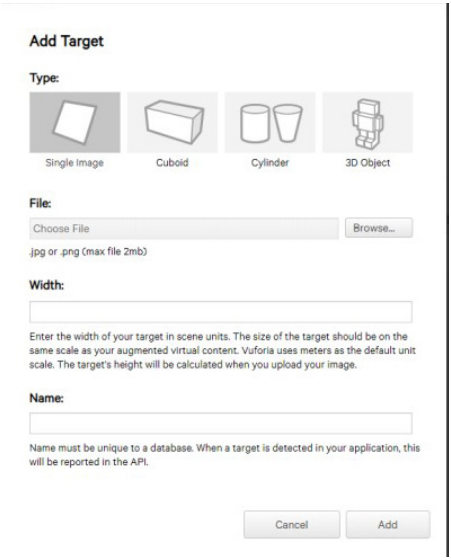


Рис. 13.4. Добавление изображения в базу данных

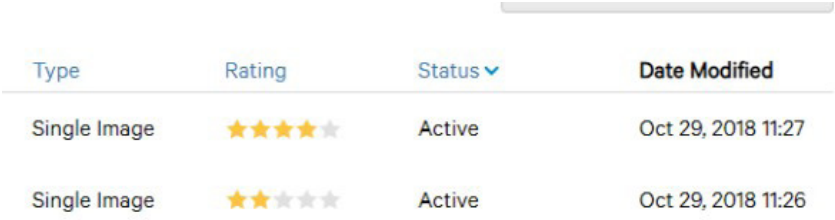


Рис. 13.5. Файлы, добавленные в базу данных и их рейтинг

После создания ключа и базы данных, подготовим модели для импорта в *Unity*, которые будут позиционироваться с помощью маркера *Vuforia*.

Далее создаём новый проект в *Unity* и подключаем библиотеку *Vuforia* (встроенную в *Unity)*, произвольную модель, базу данных (созданную на сайте *Vuforia)* (рис. 13.6).

Переходим на сайт: https://github.com/Microsoft/MixedRealityToolkit-Unity, где необходимо загрузить файлы с репозитория (рис. 13.7).

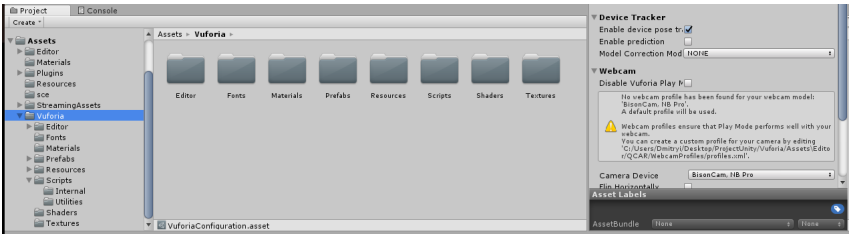


Рис. 13.6. Библиотека Vuforia

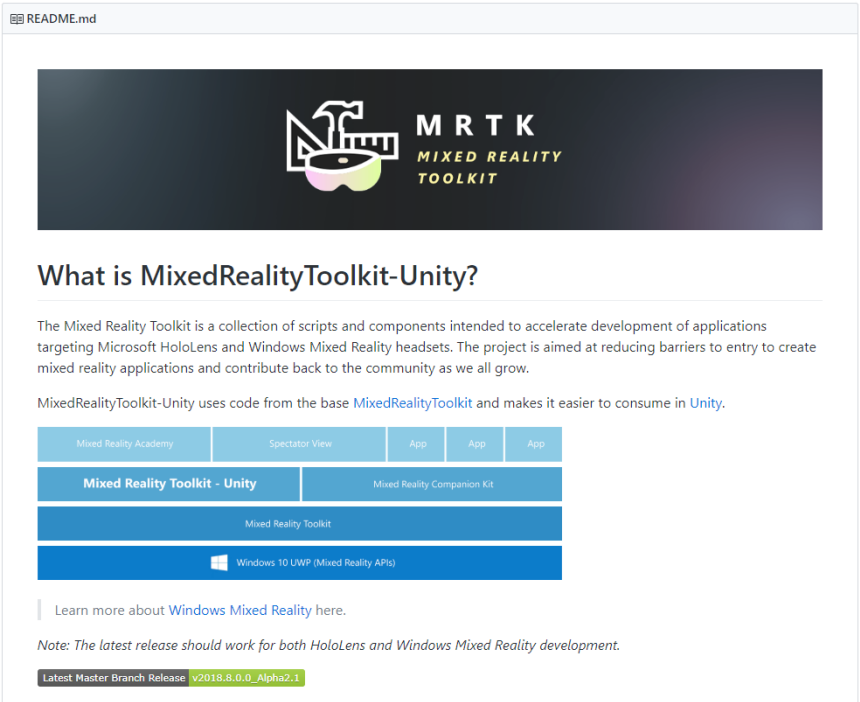


Рис. 13.7. Библиотеке MixedRealityToolkit

Далее добавляем скаченную библиотеку в проект *Unity* («*Assets*» → «*Import Package*») (рис. 13.8).

После загрузки библиотеки настраиваем проект («*MixedRealityTollkit*» → «*ProjectSetting*») (рис. 13.9).

Настраиваем сцену проекта («*MixedRealityTollkit*» → «*SceneSetting*») (рис. 13.10).

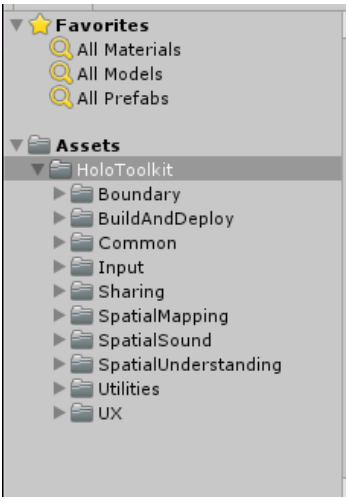


Рис. 13.8. Подгруженная библиотека

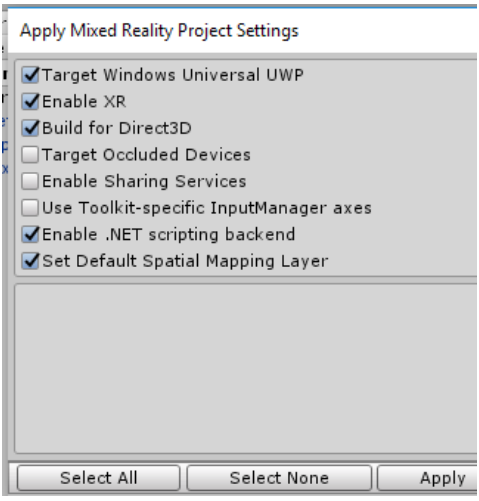


Рис. 13.9. Настройки проекта под устройства Hololens

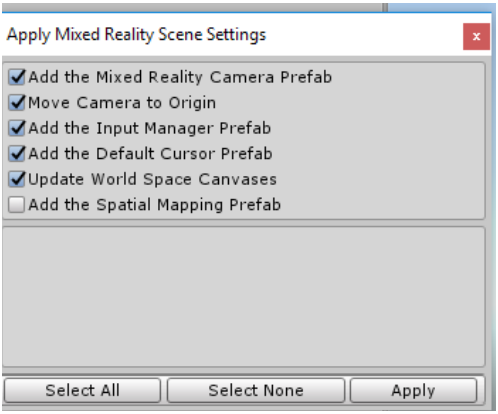


Рис. 13.10. Настройки сцены проекта

После подключения всех библиотек, можно начать работу с настройкой сцены. Для начала возьмем *prefab AR Camera* из подключенной библиотеки *Vuforia* и добавим на сцену (при правильной настройке проекта – это будет 2 камера на сцене). После откроем в *Inspectore* конфигурацию камеры (*Open Vuforia configuration*) (рис. 13.11).

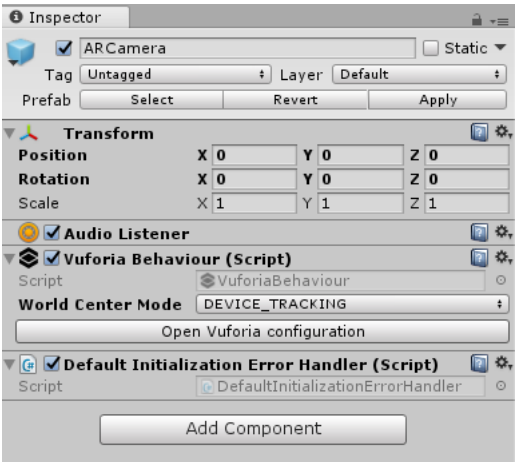


Рис. 13.11. AR камера в инспекторе

В открывшимся окне необходимо вставить раннее созданный лицензионный ключ, в поле «*App License Key*». Далее подключить и активировать созданную и загруженную базу данных (пункт *Datasets)*, а также поменять значение параметра *Divece Config* на *Hololens* (рис. 13.12).

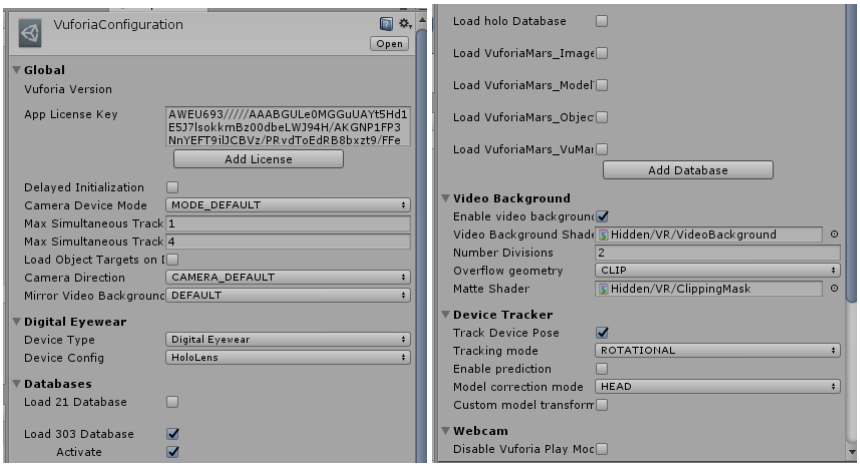


Рис. 13.12. Настройка AR камеры

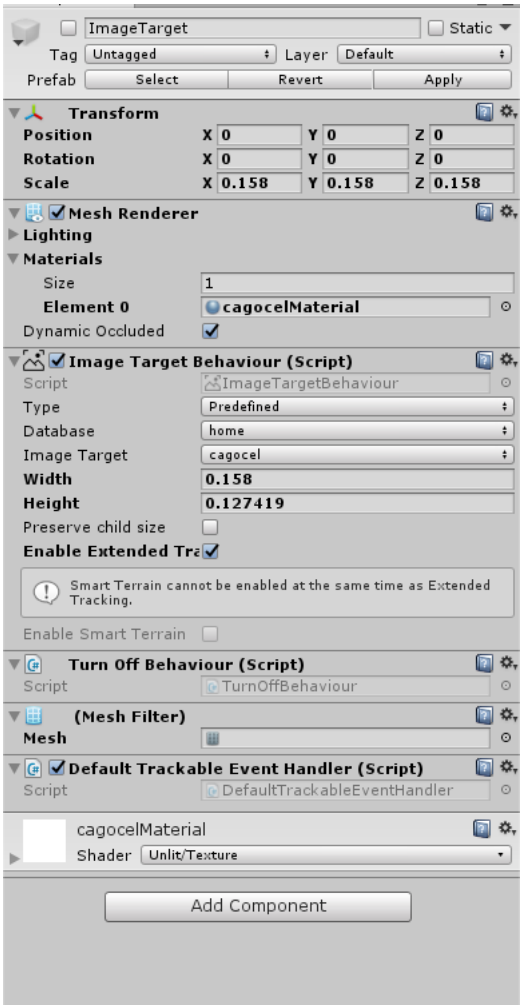


Рис. 13.13. Настройка ImageTarget

После необходимо выбрать *Prefab ImageTarget*, разместить его на сцене и в нем выбрать базу данных (рис. 13.13).

В целом библиотека *Vuforia*, подключена и настроена, теперь осталось разместить на нашем объекте *ImageTarget*.

Примечание: правильная иерархия сцены для корректной работы *Vuforia* показана на рис. 13.14.

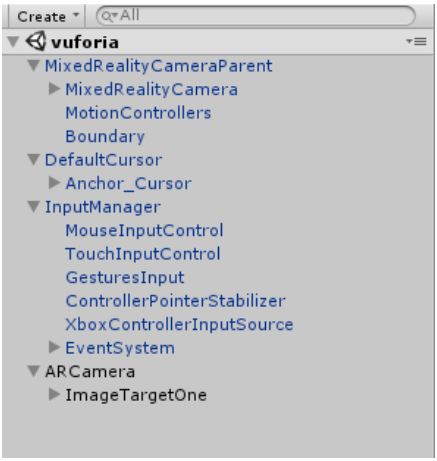


Рис. 13.14. Иерархия сцены

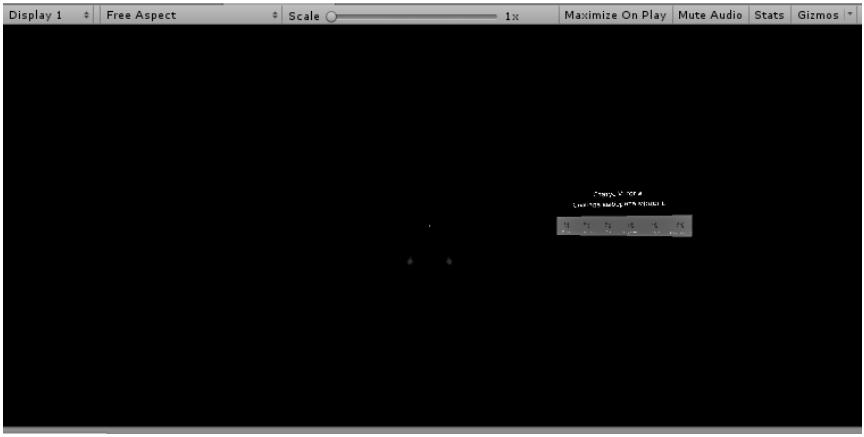


Рис. 13.15. Компиляция приложения

Для проверки работоспособности проекта необходимо развернуть приложение на *Hololens*.

Примечание: при нажатии на кнопку «*Play*» будет отображаться черный экран (рис. 13.15). Такой способ не позволяет проверить работоспособность без девайса.

Для компиляции приложения воспользуйтесь «*Build Window*» предоставленной библиотекой *MRKT* нажав на кнопку «*Builder Unity Project*» (рис. 13.16).

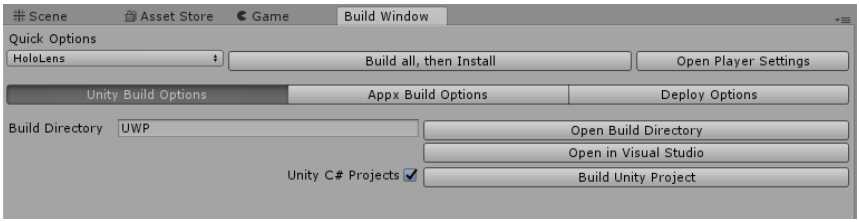


Рис. 13.16. Компиляция приложения

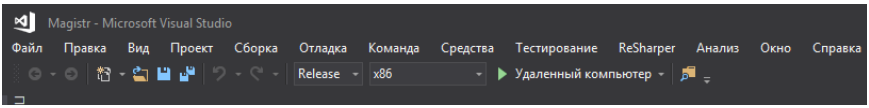


Рис. 13.17. Компиляция приложения VS

Примечание. Данное приложение возможно развернуть на *Hololens* только с помощью *Visual Studio*: https://library.vuforia.com/articles/Training/Developing-Vuforia-Apps-for-HoloLens

При компиляции проекта необходимо установить разрядность x86 и способ компиляции «Удаленный компьютер» (рис. 13.17). Обратите внимание, что версия *EAP* поддерживает только 32-битные сборки.

Проверка работоспособности

Запустим итоговое приложение и убедимся в работоспособности созданного приложения в эмуляторе (рис. 13.18).

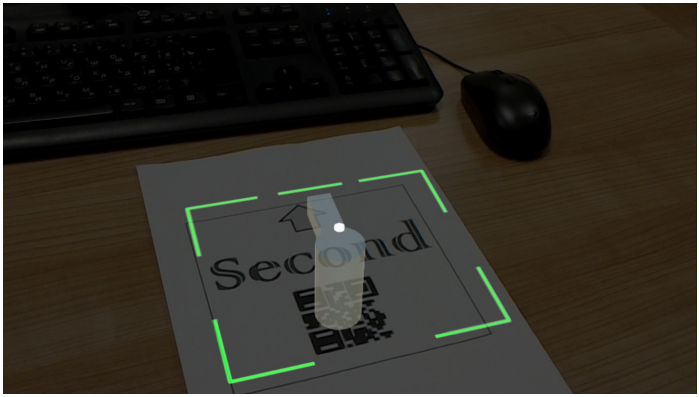


Рис. 13.18. Демонастрация результата

Контрольные вопросы

1. Как подключается библиотеки *Qualcomm Vuforia*?
2. Как создается и распознается маркер?
3. Основные этапы создания приложения дополненной реальности.
4. Как проверяется приложение дополненной реальности?
5. Какие скрипты необходимо подключить, чтобы объект следовал за камерой?

ЛАБОРАТОРНАЯ РАБОТА № 14
СОЗДАНИЕ ANDROID ПРИЛОЖЕНИЯ ДОПОЛНЕННОЙ РЕАЛЬНОСТИ С ПОМОЩЬЮ БИБЛИОТЕКИ VUFORIA

Цель работы: разработка приложения дополненной реальности с использованием инструментов *Unity3D* и библиотеки *Vuforia* для *Unity3D* для мобильного устройства на платформе *Android*.

Порядок выполнения работы

1. Подготовка проекта *Unity3D* (работа с библиотекой *Vuforia* и с персонажем).
2. Внедрение библиотеки *Qualcomm Vuforia*.
3. Создание и распознавание мишени (маркера).
4. Реализация приложения дополненной реальности.
5. Компиляция приложения в формате **apk* и его проверка.
6. Оформление отчета.

Методические указания

Vuforia – это платформа для создания *AR* – приложений для телефонов и планшетов на операционных системах *iOS* и *Android*.

Прежде чем приступить к созданию проекта и сцены в *Unity3D*, необходимо предварительно подготовить компоненты библиотеки *Vuforia*.

Для начала необходимо пройти авторизацию на сайте [https://developer.vuforia.com/] (рис. 14.1).

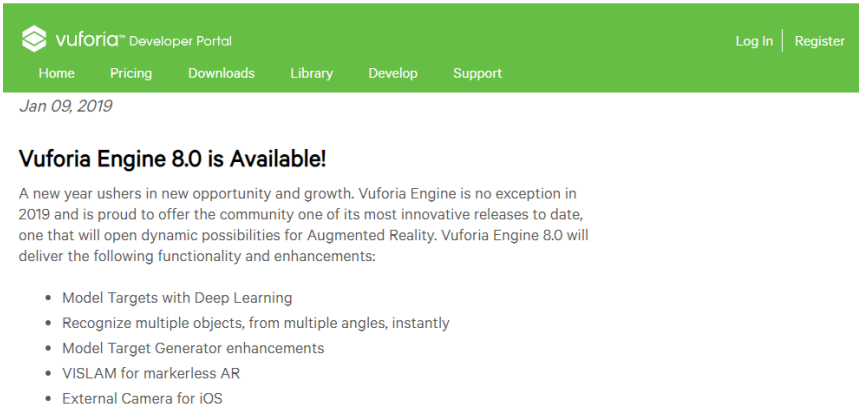


Рис. 14.1 Главная страница Vuforia

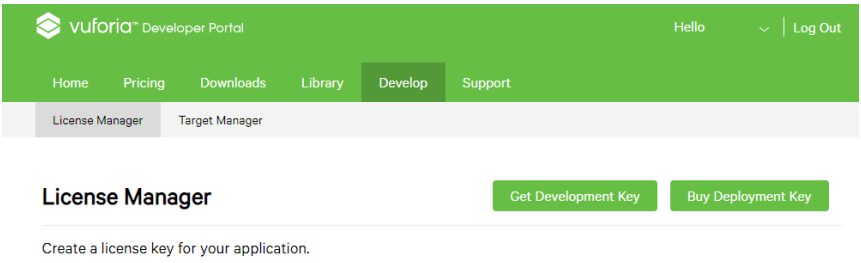


Рис. 14.2 Создание уникального ключа разработчика

Далее переходим во вкладку *Develop* → *License* Manager и выбираем пункт *Get Developer Key* (рис. 14.2). Создаём уникальный ключ разработчика для будущего проекта (рис. 14.3).

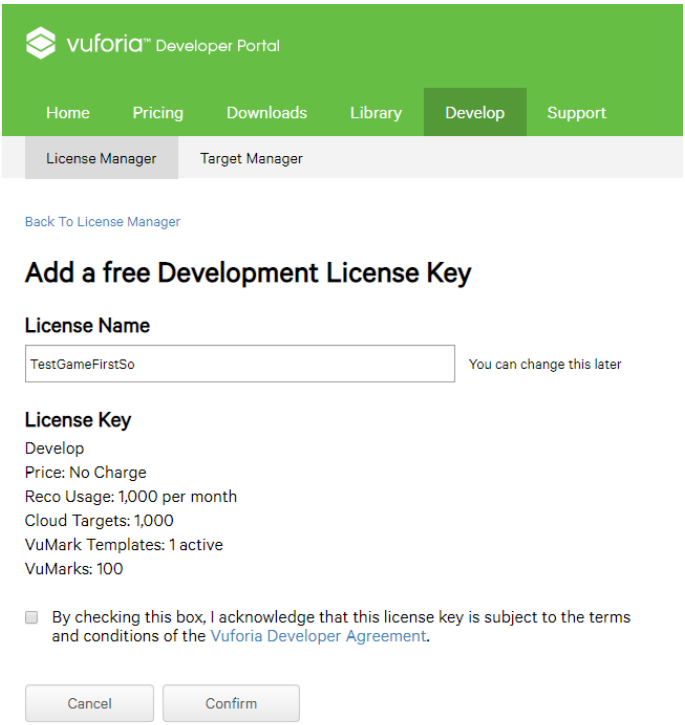


Рис. 14.3 Создание уникального ключа разработчика

Для просмотра ключа необходимо выбрать его во вкладке «License Manager» (рис. 14.4, 14.5), также в этом разделе можно изменить имя данного ключа или удалить его.

Добавлем изображение, с которым будем работать. Для этого создаём базу и загружаем исходное изображение. Для создания базы данных переходим во вкладку *Develop* → *Target Manager* и выбираем пункт *Add Database* (рис. 14.6).

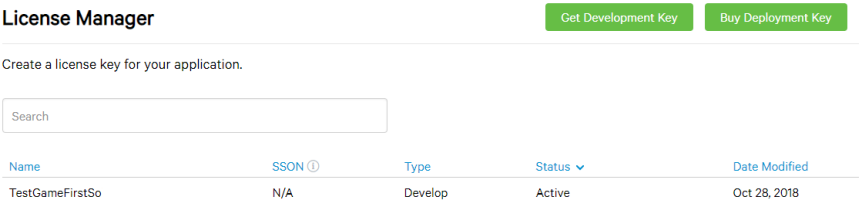


Рис. 14.4 Созданный лицензионный ключ

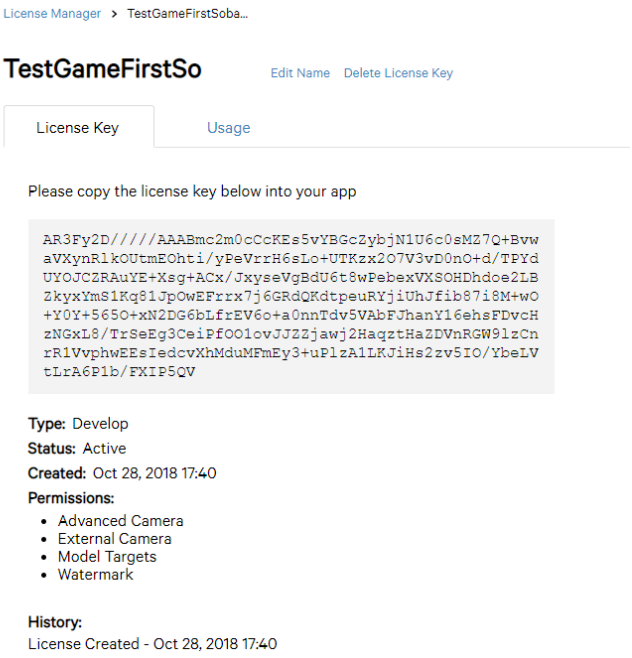


Рис. 14.5 Созданный лицензионный ключ

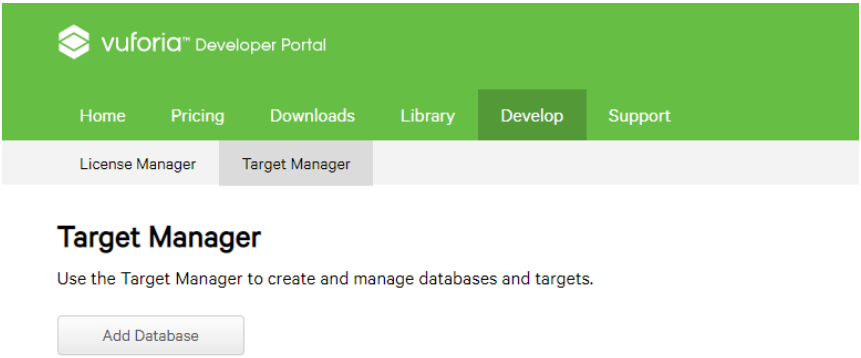


Рис. 14.6 Создание базы данных

В поле *Name* вводим название базы данных (например, *Data*). Тип базы оставим стандартный *Device* (рис. 14.7), это значит, что данные будут храниться на устройстве в конечном приложении, но можно выбрать *Cloud* тогда данные будут храниться в облаке *Vuforia* и устройство будет обращаться к нему. Далее нажимаем *Create*.

После создания базы данных она также будет отображаться в учетной записи. Теперь добавим в созданную базу изображение, для этого необходимо выбрать созданную базу данных «*Data*» (рис. 14.8) и нажать *Add Database*. Для добавления изображения нажимаем кнопку *Add Target*.

В открывшемся окне (рис. 14.10) выбирается тип изображения (в данном случае Single Image), указывается путь к исходному фалу

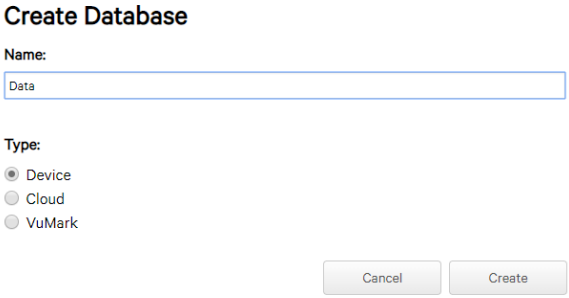


Рис. 14.7 Создание базы данных

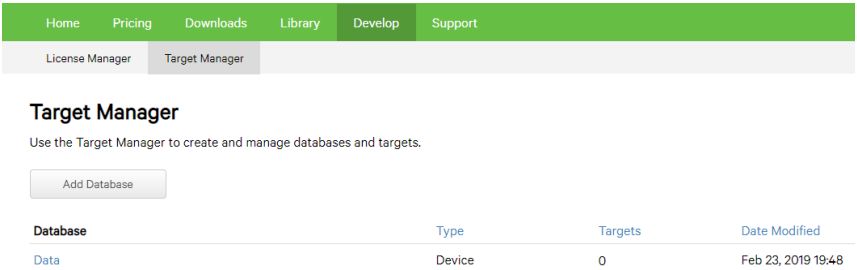


Рис. 14.8 Созданная база

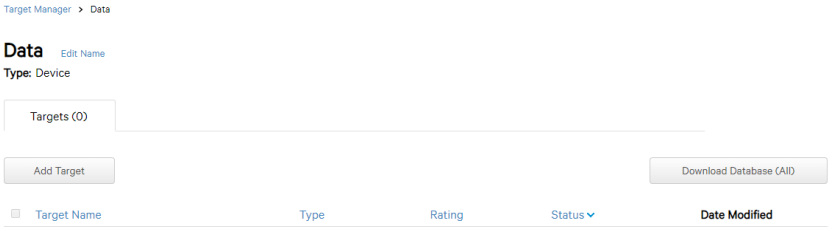


Рис. 14.9 Загрузка изображения в базу

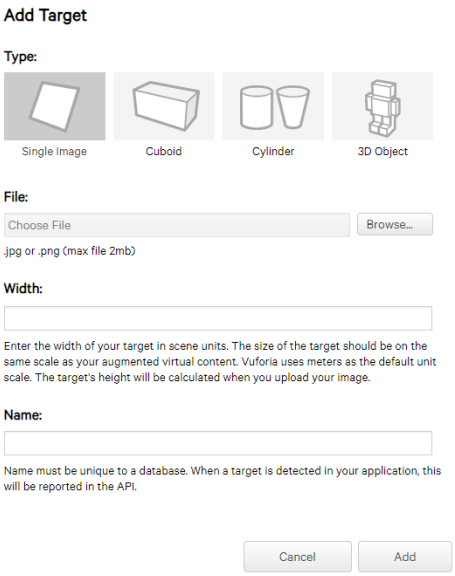


Рис. 14.10 Добавление изображения в базу данных

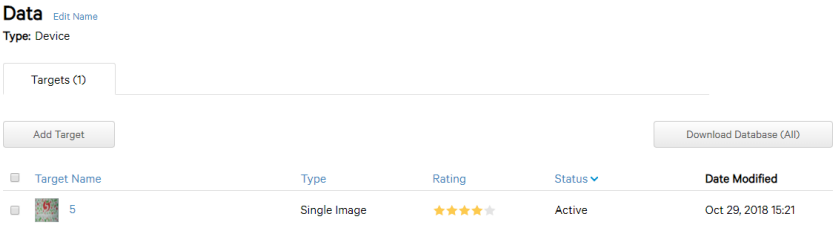


Рис. 14.11 Загруженное изображение в базу Data и ее рейтинг маркеров

(Browse), задаётся размер изображения (Width) и имя изображения (Name).

Примечание. Стоит отметить, что размер изображения не должен превышать более 2 Мб, так как оно не будет загружено. Чем больше рейтинг (рис. 14.11), тем лучше, но и при максимальном рейтинге, могут возникнуть проблемы, например, некоторые *файлы показывают максимальный рейтинг, но при тестировании и проверке маркер может быть не распознан*.

Для создания анимационного персонажа воспользуемся сервисом *Mixamo* [https://www.mixamo.com/] (рис. 14.12). Для работы требуется создать учетную запись. Там можно выбрать любого понравившегося персонажа. На данном сервисе во вкладке *Characters*, выбираем персонажа и скачиваем его в формате **fbx* (пункт *Download)*.

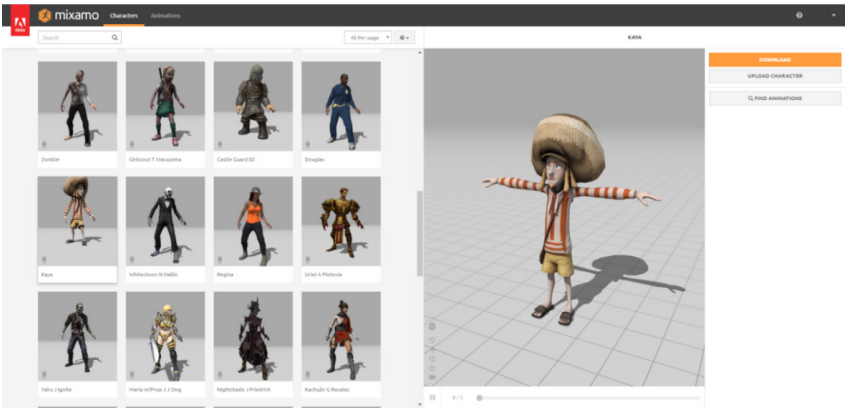


Рис. 14.12 Выбор персонажа на сайте Mixamo

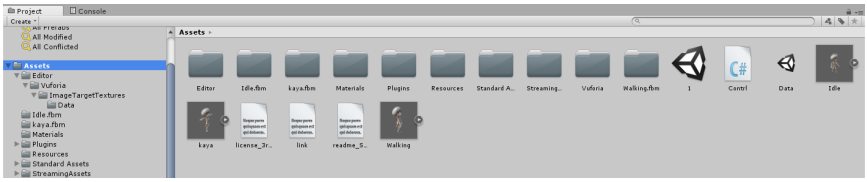


Рис. 14.13 Импорт данных и материалов в проект

Далее создём новый проект в *Unity*, подключаем библиотеку *Vuforia*, добавляем персонажа, и базу данных, созданную на сайте *Vuforia* (рис. 14.13).

Примечание. В ходе выполнения и тестирования работы использовалась версия Unity 5.6.0f3 (64-bit), и более ранняя библиотека Vuforia 6.2.10 скачанная на сайте Vuforia. При использовании Unity версии 2017 и выше, не рекомендуется ставить данный Asset Vuforia из Unity Store, т.к. при установке Unity и реализации проекта будет конфликт из-за разных версий данной библиотеки.

После подключения библиотеки Vuforia в проект, импортирования базы данных и персонажа, начинаем работу с проектом. Удалим стандартную камеру проекта и разместим на сцене *Prefab AR Camera* из подключенной библиотеки *Vuforia*. Откроем в *Inspectore* конфигурацию *AR* камеры *Open Vuforia configuration* (рис. 14.14).

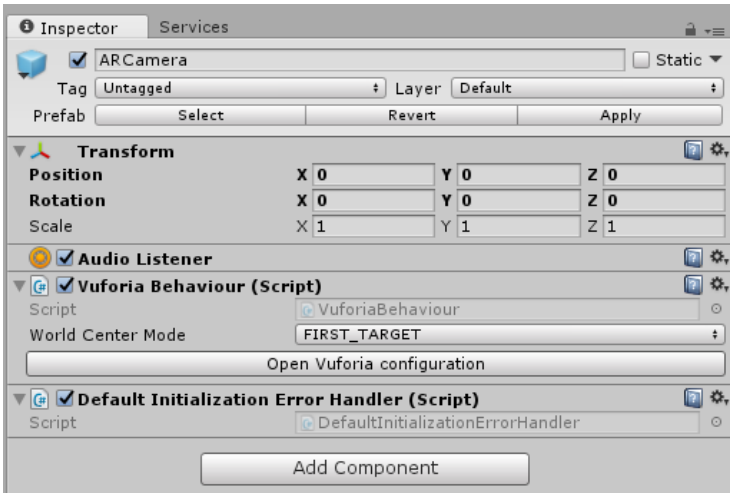


Рис. 14.14 Настройка AR камеры, часть 1

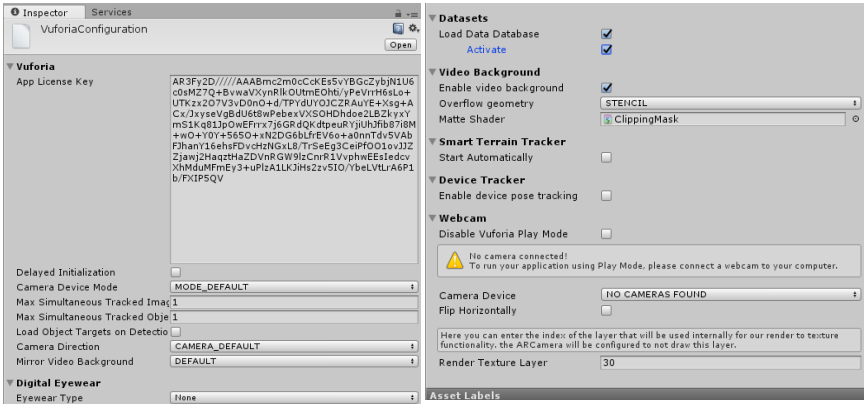


Рис. 14.15 Настройка AR камеры, часть 2

В открывшимся окне (рис. 14.15) необходимо вставить в поле *App License Key* раннее созданный лицензионный ключ на сайте *Vuforia*, а также подключить и активировать созданную и подключенную базу данных, пункт *Datasets*.

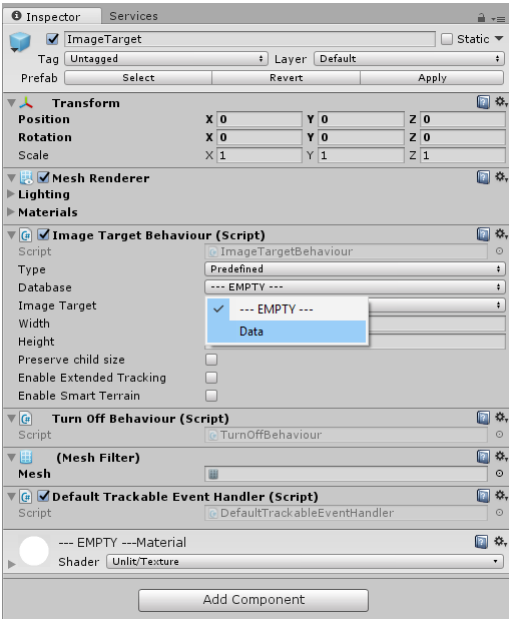


Рис. 14.16 Подключение созданной базы данных

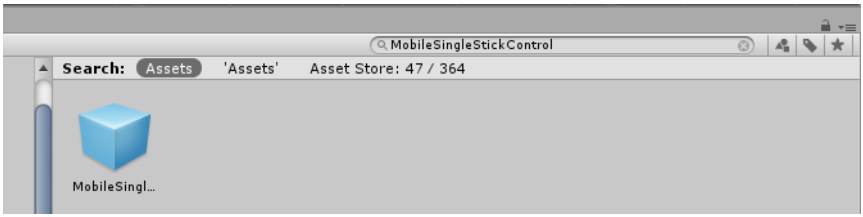


Рис. 14.17 Prefab Unity MobileSingleStickControl

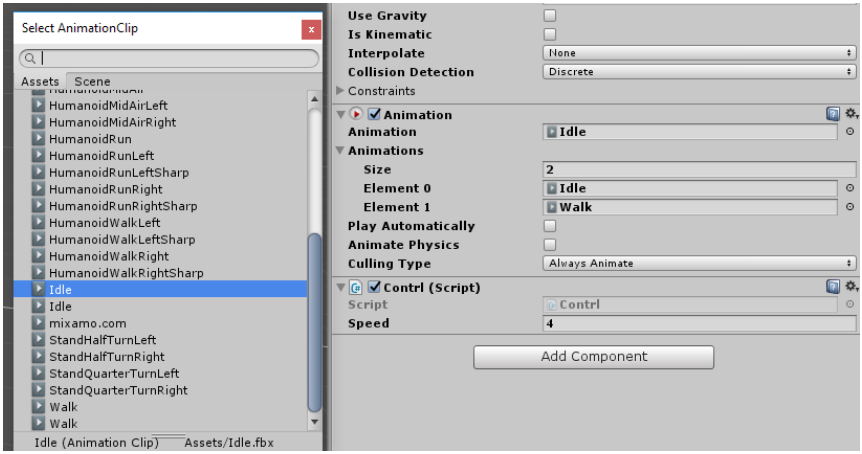


Рис. 14.18 Работа с компонентом Animator,
добавление стандартных анимаций перемещения

Далее разместим *Prefab ImageTarget* на сцене. В настройках префаба выберем созданную базу данных в скрипте *Image Target Behaviour* (рис. 14.16).

Библиотека *Vuforia* подключена и настроена. Далее размещаем на объекте *ImageTarget* персонажа, добавляем кнопки управления *Prefab Unity MobileSingleStickControl* (рис. 14.17) и анимацию, путем добавления к персонажу компонента Animation (рис. 14.18). Так же необходимо написать скрипт *Contrl* для перемещения и работы персонажа.

Проделав данные операции, на выходе имеем: проект *Unity3D* с библиотекой *Vuforia*; настроенную базу данных, с применением

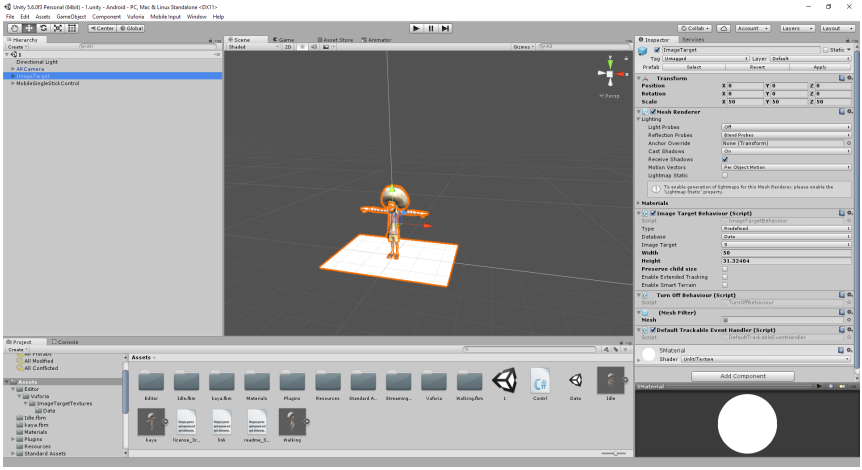


Рис. 14.19 Промежуточный проект в Unity3D

лицензионного ключа; импортированного с сервиса *Mixamo* персонажа (рис. 14.19).

Скрипт Contrl.

```
using UnityEngine;
using UnityStandardAssets.CrossPlatformInput;
public class Contrl : MonoBehaviour
{
    private Animation anim;
    private Rigidbody rb;
    [SerializeField]
    private float speed = 4f;
    // Use this for initialization
    void Start()
    {
        rb = GetComponent<Rigidbody>();
        anim = GetComponent<Animation>();
    }
    // Update is called once per frame
    void Update()
    {
float x = CrossPlatformInputManager.GetAxis("Horizontal");
float y = CrossPlatformInputManager.GetAxis("Vertical");
        Vector3 movement = new Vector3(x, 0, y);
        rb.velocity = movement * speed;
        if (x != 0 && y != 0)
transform.eulerAngles = new Vector3(transform.eulerAngles.x, Mathf.
Atan2(x, y) * Mathf.Rad2Deg, transform.eulerAngles.z);
```

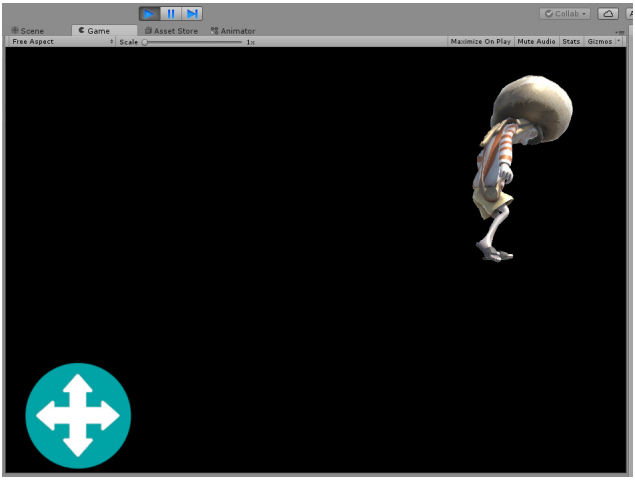


Рис. 14.20 Проверка работоспособности проекта в Unity3D

```
        if (x != 0 && y != 0)
            anim.Play("Walk");
        else
            anim.Play(«Idle»);
    }
}
```

Запускаем режим *Play* в *Unity* и проверяем работоспособность проекта. Если к персональному компьютеру подключена *web* – камера, то в библиотеке *Vuforia* есть возможность подключить ее к проекту, тем самым можно напрямую проверить работоспособность проекта (рис. 14.20).

При запуске проекта персонаж передвигается, следовательно, скрипт работает, все настроено и подключено правильно. Далее скомпилируем приложение.

Примечание. Данное приложение будет скомпилировано под платформу Android. Поэтому предварительно необходимо настроить и установить, *java.jdk* и *sdk* компоненты на ПК и подключить их к Unity3D (рис. 14.21).

Приложение будет скомпилировано в формате **.apk*. После успешной компиляции, файл необходимо скачать на смартфон с ОС Android и установить.

Запускаем итоговое приложение на устройстве с ОС *Android* и наблюдаем результат (рис. 14.22).

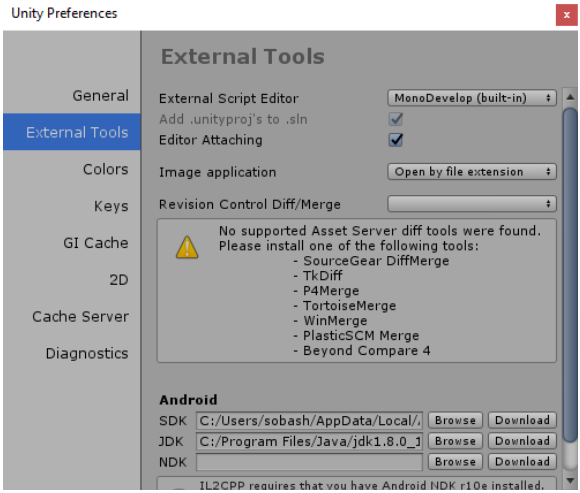


Рис. 14.21 Вкладка Unity Preferences

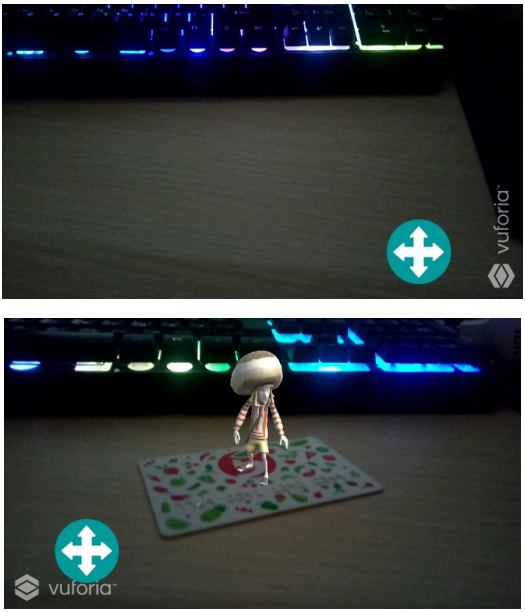


Рис. 14.22 Проверка работоспособности приложения

Контрольные вопросы

1. Что такое *Vuforia*?

2. Какие файлы могут быть использованы для инициализации в базе данных *Vuforia*?

3. Какие есть особенности при создании приложения под платформу *Android*, дополненной реальностью и без?

ЛАБОРАТОРНАЯ РАБОТА № 15
СОЗДАНИЕ ANDROID ПРИЛОЖЕНИЯ ДЛЯ GEARVR

Цель работы: разработка приложения под платформу *Samsung GearVR*, реализованного с привязкой к смартфону *Samsung Galaxy* в *Unity 3D*.

Порядок выполнения работы

1. Подготовка смартфона *Samsung Galaxy S6*.
2. Подготовка проекта *Unity3D*.
3. Внедрение библиотеки *OculusUtilites*.
4. Работа с *Android SDK*.
5. Реализация приложения.
6. Компиляция приложения в формате **apk* и его проверка.
7. Оформление отчета.

Методические указания

Для начала необходимо зайти на сайт [https://developer.oculus.com/downloads/] (рис. 15.1), перейти в раздел *Unity* и выбрать пункт (рис. 15.2) *Oculus Utilities for Unity*. При составлении методических указаний была использована версия 1.23.0.

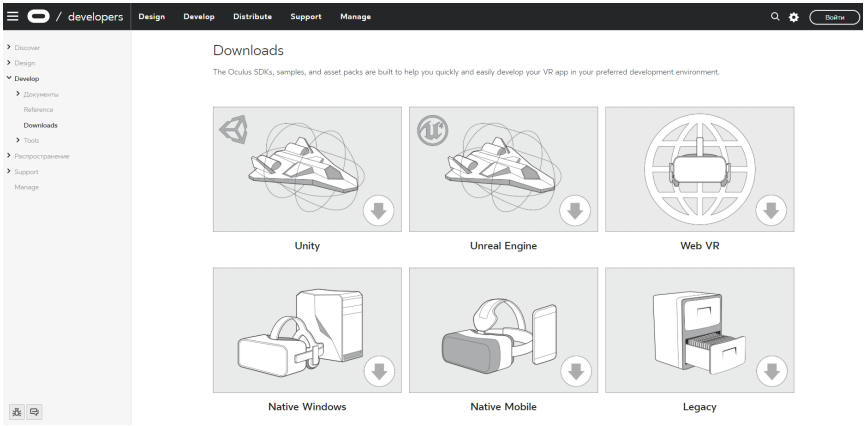


Рис. 15.1 Сайт разработчиков для Oculus

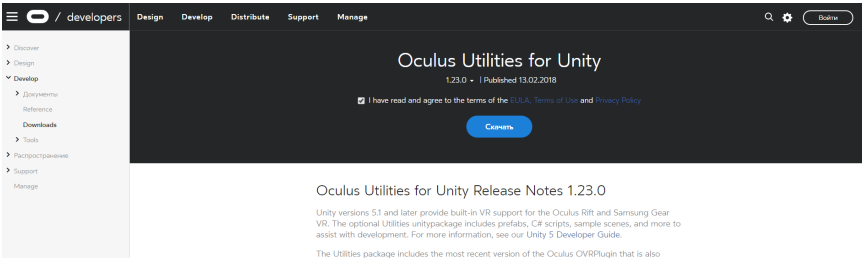


Рис. 15.2 Библиотека Oculus Utilities for Unity

Далее с помощью входящего в состав *Android SDK* инструмента *Android Debug Bridge (ADB)* определяется *Device ID* мобильного устройства.

Для этого откроем командную строку, перейдем в каталог, куда были установлены компоненты *SDK* и запустим процесс *platform-tools*.

Стоит отметить, что, при данном этапе, телефон должен быть подключен к компьютеру через *USB* и в нем должна быть включена функция «Отладка по *USB*» (рис. 15.3), а также разрешена установка приложений из неизвестных источников. Для включения режима отладки, необходимо на телефоне открыть «Настройки» → «Для разработчиков» и включить функцию «Отладка *USB*».

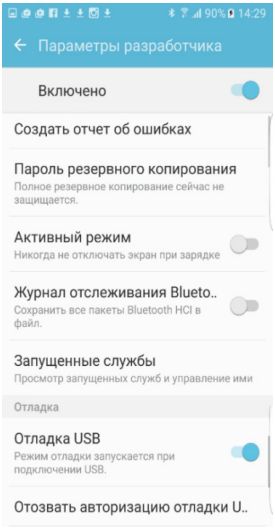


Рис. 15.3 Включение режима разработчика

Для того чтобы запустить *platform-tools* необходимо:

Открыть «Пуск» → «Все приложения» → «Служебные» → «Командная строка», при этом выбрав запуск от имени администратора. Далее перейти в директорию, где установлен *Android SDK* (например, *C:/Users/ИмяПользователя/AppData/Local/Android/Sdk/platform-tools*), и после активировать файл «*adb devices*» (рис. 15.4).

В результат будет выдан *Device ID* устройства: *06157df63cb20320*. Далее необходимо зайти на сайт [https://developer.oculus.com/unity/], выполнить авторизцию и в меню *Develop* → *Tools* → *OSIG*

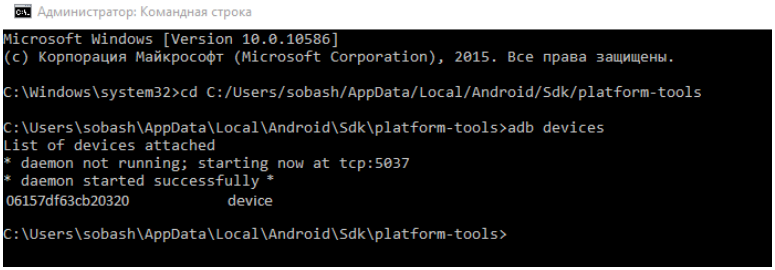


Рис. 15.4 Список ID подключенных android устройств

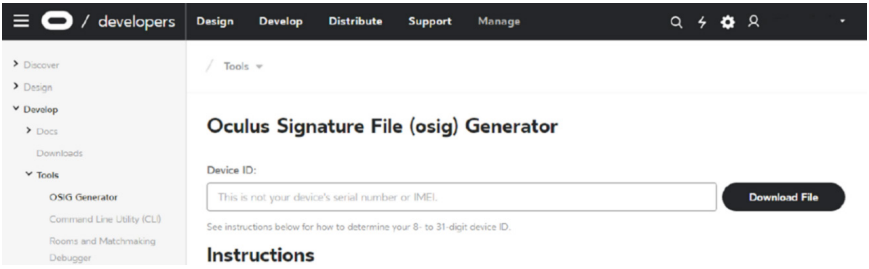


Рис. 15.5 Генерирование ключа по ID на сайте Oculus

Generator ввести полученный *Device* ID для генерации ключа «подвязки» к нашему устройству (рис. 15.5).

Созданный файл необходимо скачать и подгрузить в созданный проект *Unity3D* в папку *Assets* → *Android* (рис. 15.6).

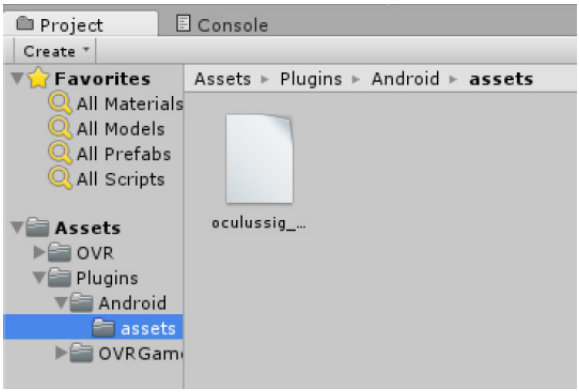


Рис. 15.6 Добавление в проект сгенерированного файла

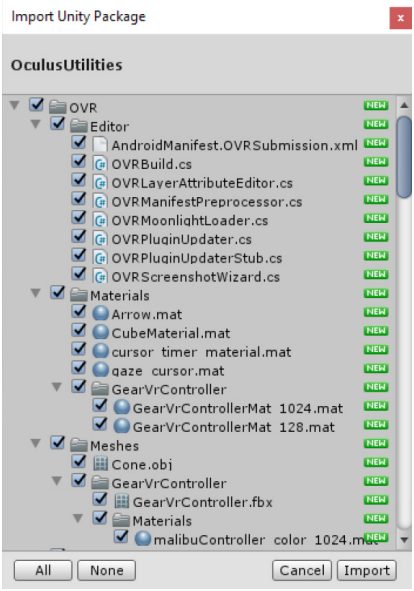


Рис. 15.7 Импорт библиотеки OculusUtilites в проект Unity

Далее импортируем в проект предварительно скачанную библиотеку *OculusUtilites* (рис. 15.7).

Теперь можно приступить к выполнению проекта. Сначала создадим стандартный объект Plane (рис. 15.8).

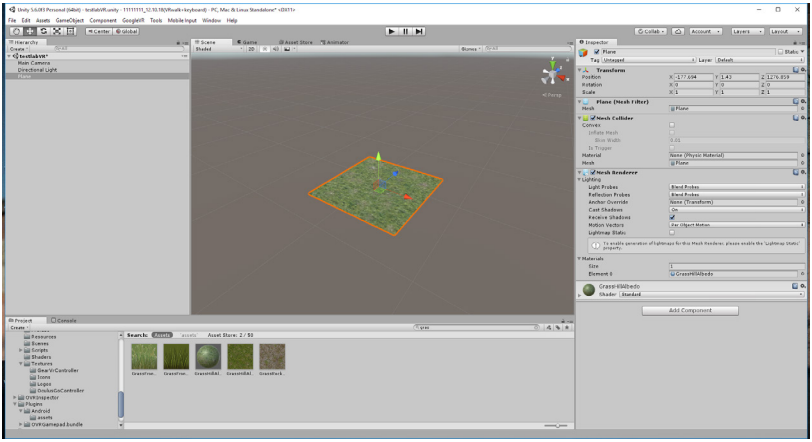


Рис. 15.8 Создание Plane

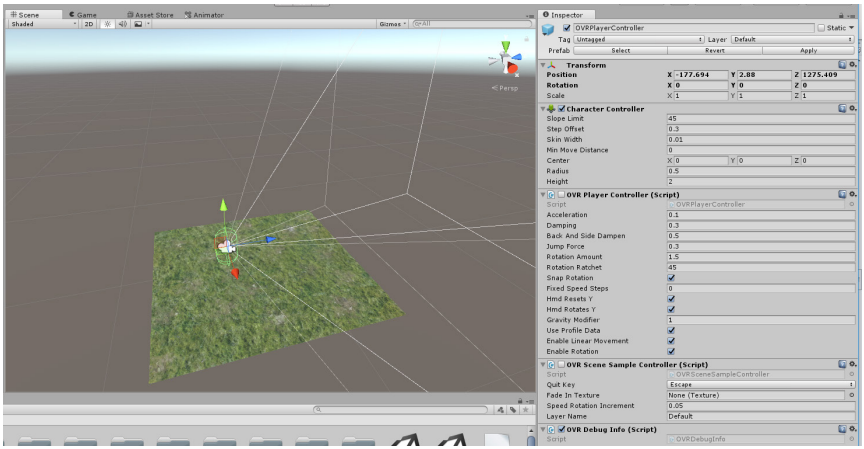


Рис. 15.9 Работа с OVRPlayerController

Далее необходимо добавить из библиотеки *Oculus Prefab OVRPlayerController*. Компоненты *OVR Player Controller* и *OVR Scene Sample Controller* можно отключить, так как в данной работе они не используются (рис. 15.9).

Добавим к контроллеру скрипт, выполняющий перемещение по сцене с помощью наклона головы. К скрипту необходимо привязать камеру (*OVRCameraRig)* и отцентровать ее расположение. Если наклон головы ниже нормали, то контроллер, персонаж перемещается

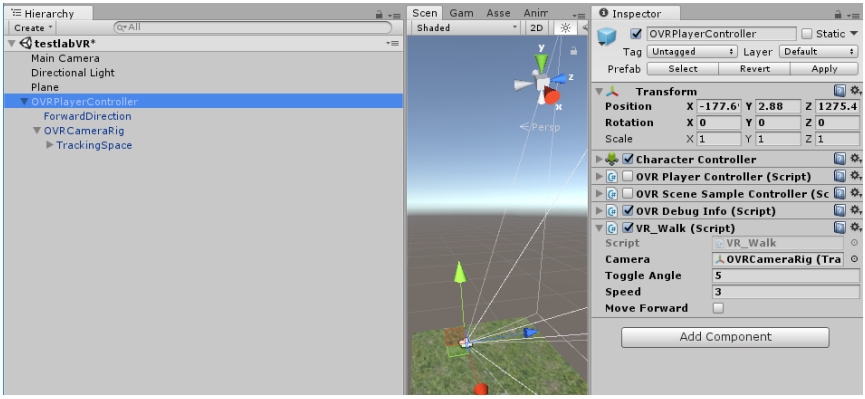


Рис. 15.10 Работа с OVRPlayerController

по сцене, если же выше нормали, то контроллер стоит, и мы можем просто осматривать сцену.

Скрипт VR_Walk

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class VR_Walk : MonoBehaviour
{
    public Transform Camera;
    public float toggleAngle = 5.0f;
    public float speed = 3.0f;
    public bool moveForward;
    private CharacterController cc;
    // Use this for initialization
    void Start()
    {
        cc = GetComponent<CharacterController>();
    }
    // Update is called once per frame
    void Update()
    {
if (Camera.eulerAngles.x >= toggleAngle && Camera.eulerAngles.x <
90.0f)
        {
            moveForward = true;
        }
        else
        {
            moveForward = false;
        }
        if (moveForward)
        {
Vector3 forward = Camera.TransformDirection(Vector3.forward);
            cc.SimpleMove(forward * speed);
        }
    }
}
```

Также стоит отметить, что в *OVRCameraRig* компонента *OVRPlayerController*, необходимо добавить камеру, при помощи которой будет происходить просмотр (рис. 15.11).

После проделанных действий можно запустить и проверить сцену, нажав «*Play*». Сцена должна отображаться, но перемещение и обзор не будут доступны (рис. 15.12), так как наголовный шлем не подключен.

Для отладки подключим к проекту библиотеку *GoogleVR* с сайта: [https://developers.google.com/vr/develop/unity/get-started-android].

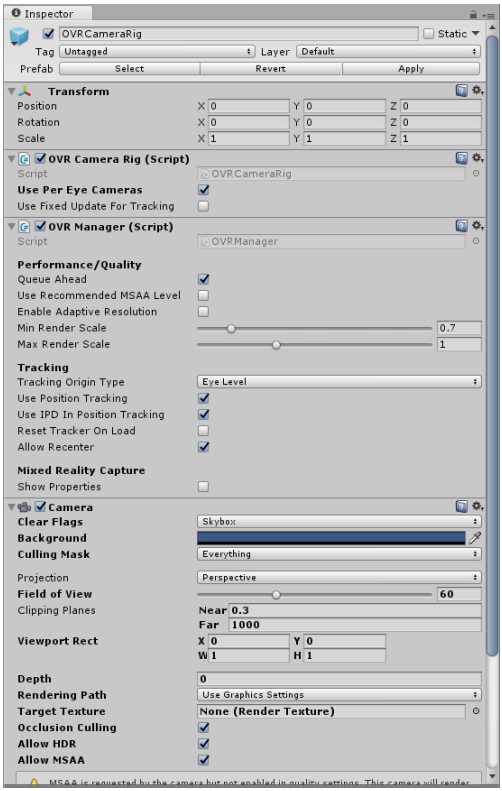


Рис. 15.11 Добавление компонента Camera в OVRCameraRig

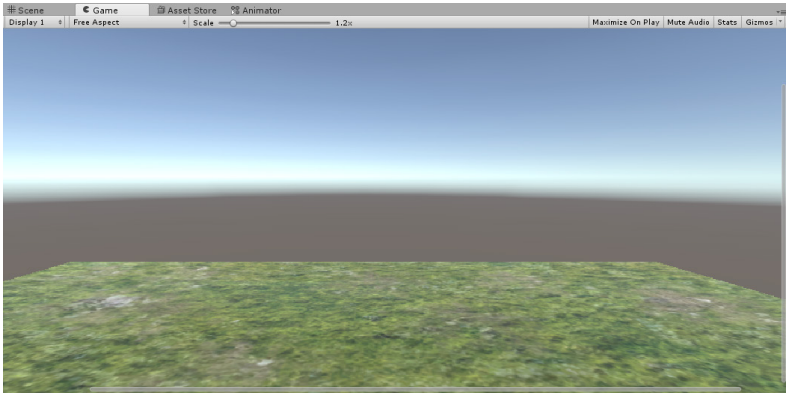


Рис. 15.12 Отображение сцены в Unity

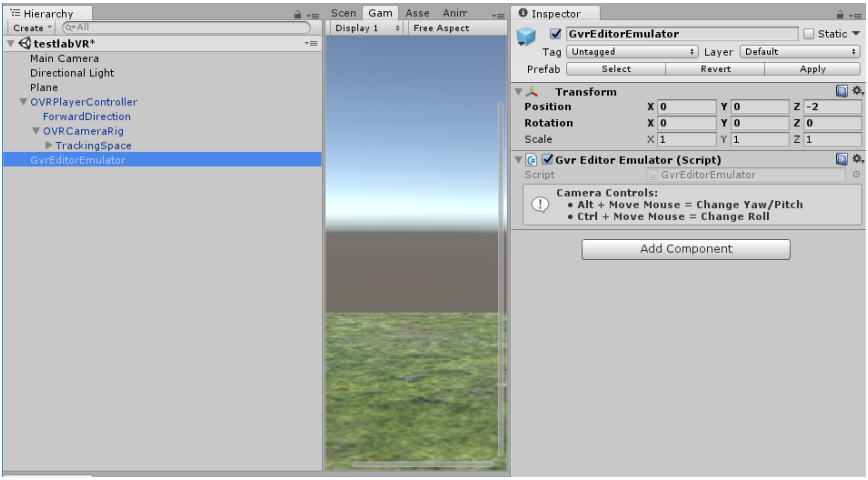


Рис. 15.13 Подключенный эмулятор VR и готовая итоговая сцена в Unity3D

После перетащим на сцену *Prefab GvrEditorEmulator*, чтобы была возможность добавить эмулятор *VR*, который позволяет взаимодействовать со сценой без шлема виртуальной реальности (рис. 15.13).

Способ управления описан в скрипте и отображается во вкладке *Inspector* в *Unity3D*, а именно:

Alt + Движение мышкой = наклон вверх либо же вниз,

Ctrl + Движение мышкой = вращение влево либо же вправо.

После проверки, приступим к компиляции приложения для телефона (в данном случае для *Samsung Galaxy S6)*.

В меню *Build Settings* необходимо выбрать платформу *Android* и *Texture Compression (GLES 3.0)*, после нажать кнопку *Switch Platform* и ждать смены платформы (рис. 15.14), далее перейдем в расширенные настройки проекта (*Player Settings*).

В инспекторе необходимо указать, что приложение будет собрано под платформу Oculus (устанавливаем галочку в «*Virtual Reality SDK`s*» и выбираем Oculus). А также, выбрать поддержку «*API Android*» версии не ниже 19, иначе проект не будет работать корректно (рис. 15.15).

Теперь нажимаем кнопку *Build* (рис. 15.16), дождемся компиляции и получения итогового файла в формате **.apk*, которое будет запущенно на устройстве (*Samsung Galaxy)* (рис. 15.17).

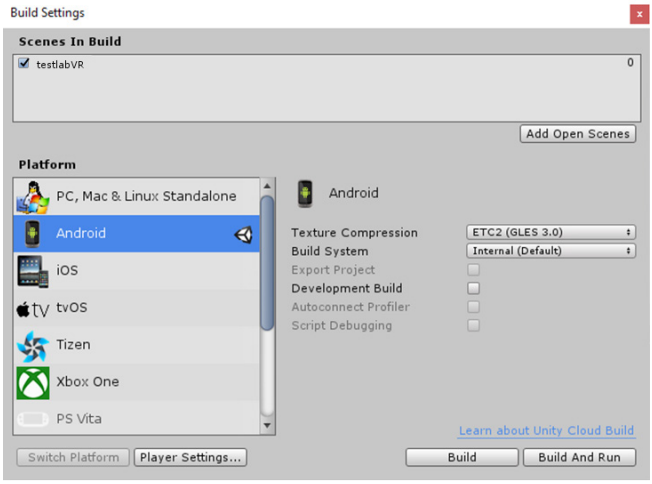


Рис. 15.14 Сборка приложения

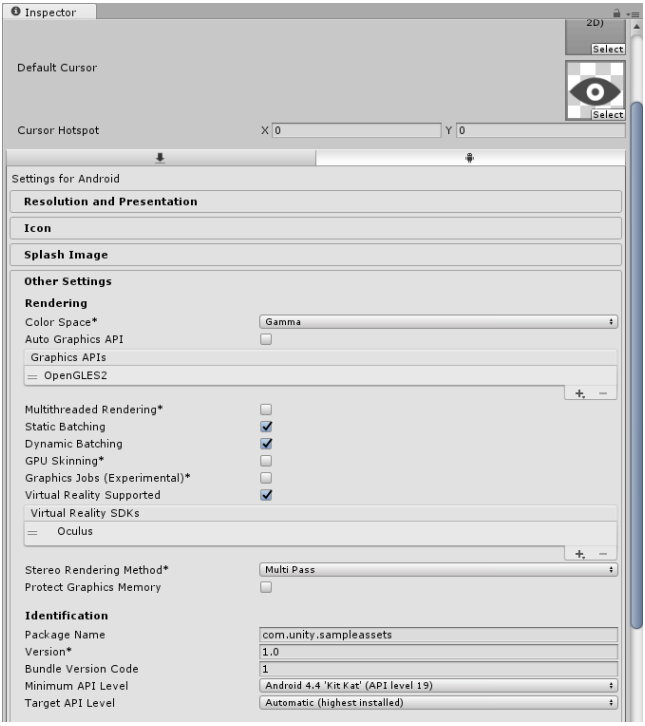


Рис. 15.15 Сборка приложения

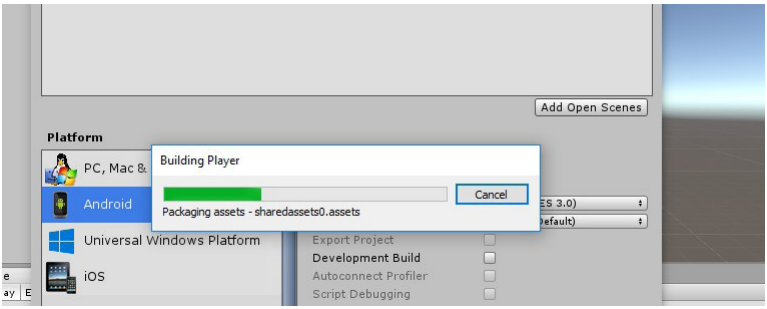


*Рис. 15.16 Компиляция *apk приложения*

Рис. 15.17 Запущенное приложение

После запуска приложения, необходимо поместить устройство в *GearVR* и проверить результат работы.

Контрольные вопросы

1. Каков принцип подключения библиотеки *Oculus* в *Unity3D*?
2. Что такое *OSIG* – файл?
3. Как происходит настройка проекта *Unity3D* для экспорта под *Oculus*?

Лабораторная работа № 16
СОЗДАНИЕ ПРИЛОЖЕНИЯ ДЛЯ ПЛАТФОРМЫ OCULUS RIFT

Цель работы*:* разработка приложения под платформу *Oculus Rift* совместно с персональным компьютером (ПК) в *Unity 3D*.

Порядок выполнения работы

1. Подготовка проекта *Unity3D*.
2. Внедрение библиотеки *OculusUtilites*.
3. Реализация приложения.
4. Компиляция приложения в формате **.exe* и его проверка совместно с *Oculus Rift*.
5. Оформление отчета.

Методические указания

Перед началом выполнения лабораторной работы необходимо перейти на сайт [https://developer.oculus.com/downloads/] (рис. 16.1), в разделе *Unity* выбрать пункт (рис. 16.2) *Oculus Utilities for Unity* и скачать версию 1.23.0.

После загрузки библиотеки приступаем к созданию проекта в *Unity* (рис. 16.3).

Далее импортируем в проект предварительно скачанную библиотеку *OculusUtilites*. (рис. 16.4).

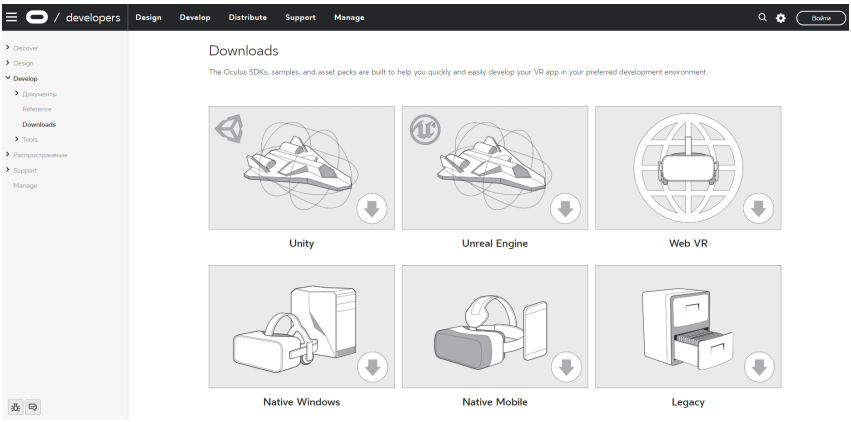


Рис. 16.1. Сайт разработчиков для Oculus

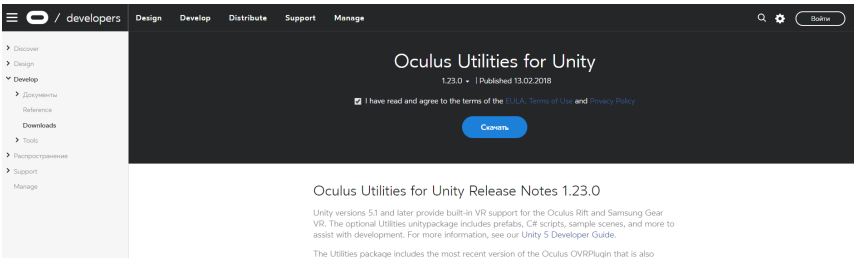


Рис. 16.2 Библиотека Oculus Utilities for Unity

Рис. 16.3. Пустой проект Unity

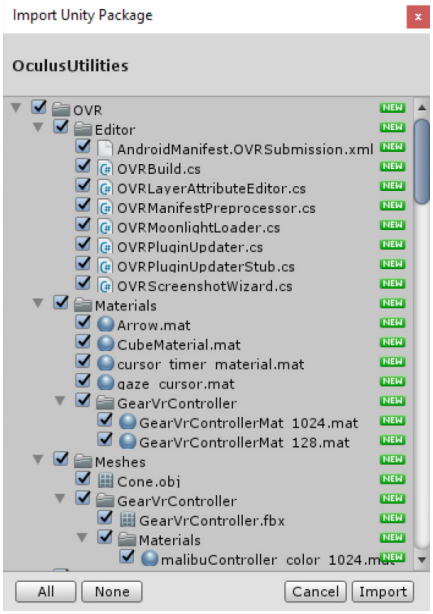


Рис. 16.4. Импорт библиотеки OculusUtilites в проект Unity

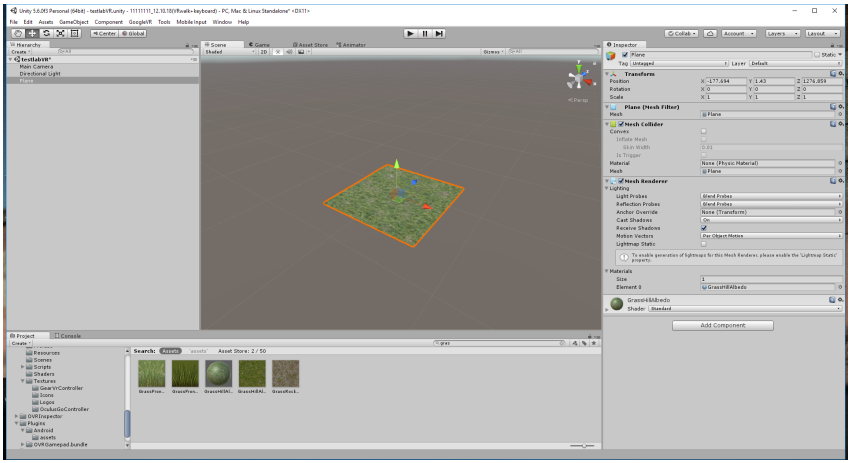


Рис. 16.5. Создание Plane

После импорта *OculusUtilites*, создадим стандартный объект Plane (рис. 16.5).

Добавим на сцену *Oculus Prefab OVRPlayerController* из импортированной библиотеки (рис. 16.6). Перемещение контроллера по сцене уже добавлено в данный префаб, возможно управление как с помощью клавиатуры, так и с помощью геймпада.

Также в *OVRCameraRig* компонента *OVRPlayerController*, необходимо добавить камеру, при помощи которой будет происходить просмотр (рис. 16.7).

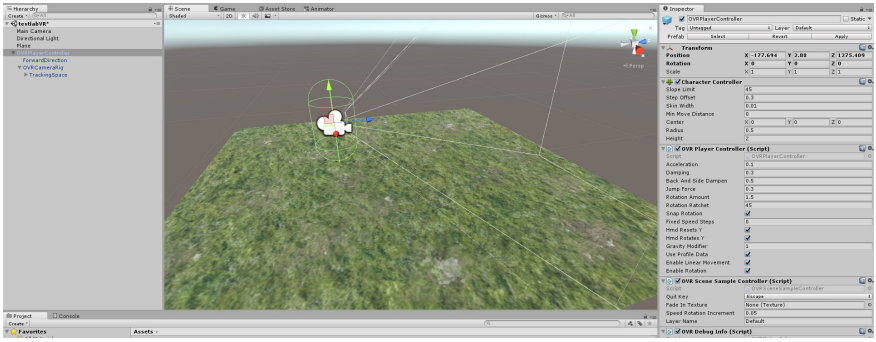


Рис. 16.6. Работа с OVRPlayerController

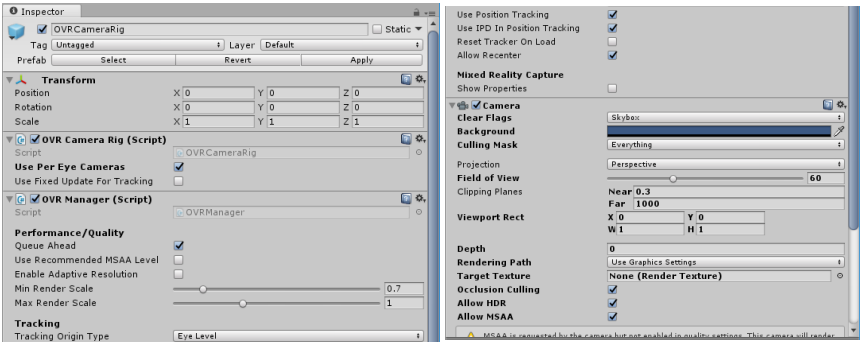


Рис. 16.7. Добавление компонента Camera в OVRCameraRig

Для проверки сцены перейдем в режим игры, нажав кнопку Play (рис. 16.8, 16.9).

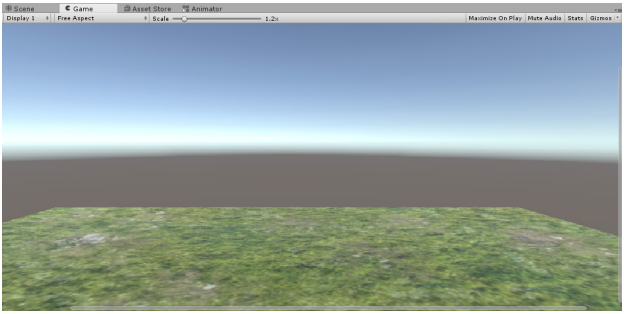


Рис. 16.8. Отображение сцены в Unity

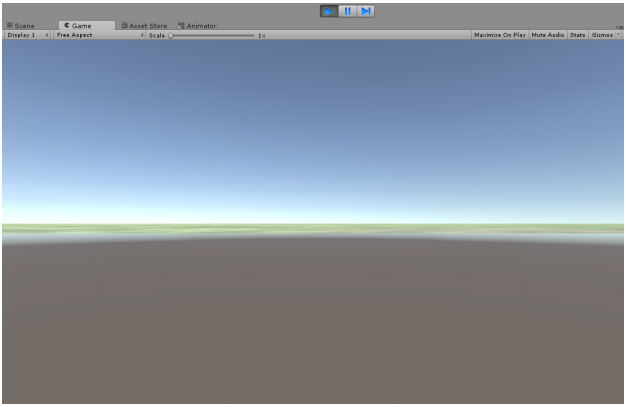


Рис. 16.9. Проигрывание сцены

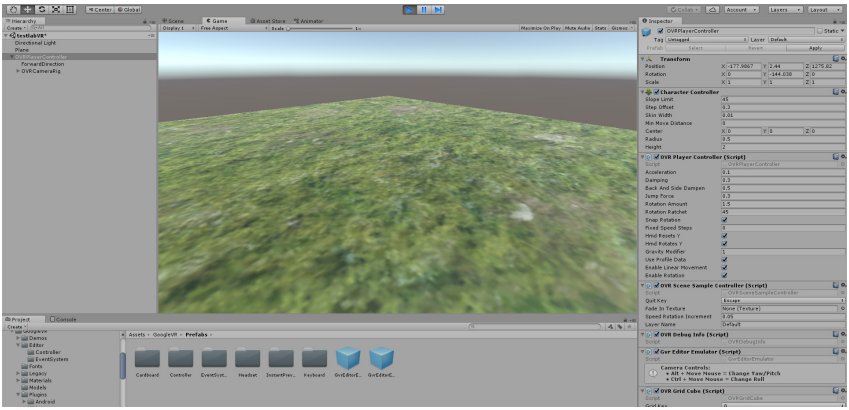


Рис. 16.10. Подключение эмулятора VR и тест приложения

В связи с тем, что шлем не был подключен, то отображение перемещения работает некорректно. Для устранения этого недостатка необходимо подключить библиотеку *GoogleVR* с сайта [https://developers.google.com/vr/develop/unity/get-started-android].

Следующим шагом перетащим на сцену *PrefabGvrEditorEmulator*, для возможности добавления эмулятора *VR* (рис. 16.10). Способ управления описан в скрипте и отображается во вкладке *Inspector* в *Unity3D*, а именно:

Alt + Движение мышкой = наклон вверх либо же вниз,

Ctrl + Движение мышкой = вращение влево либо же вправо.

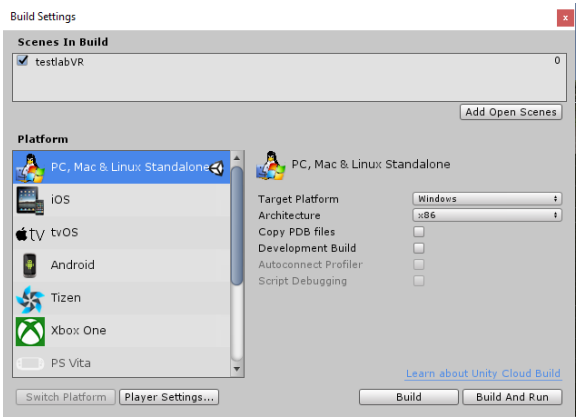


Рис. 16.11. Сборка приложения

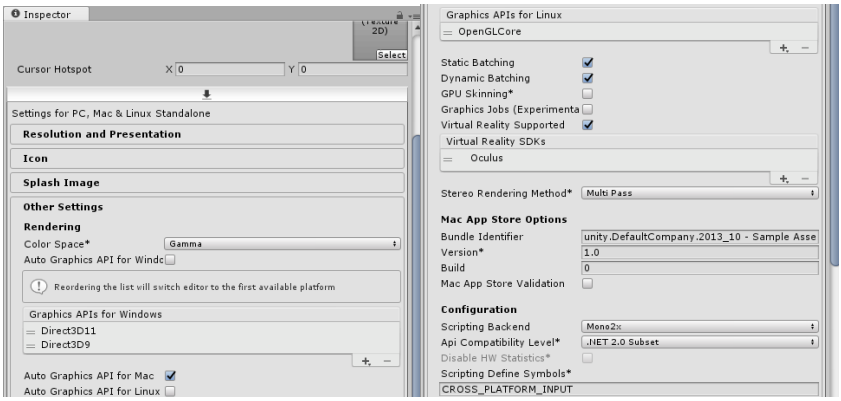


Рис. 16.12. Настройка приложения для корректной сборки

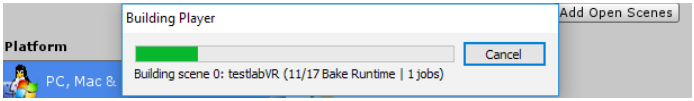


*Рис. 16.13. Компиляция *.exe приложения*

После проверки приступим к компиляции приложения. Для этого необходимо скорректировать настройки сборки проекта, откроем *Build Settings*.

При сборке приложения необходимо выбрать платформу *PC*, после нажать кнопку *Switch Platform* и ждать смены платформы, далее перейдем в расширенные настройки проекта, нажмем *Player Settings*.

В инспекторе необходимо указать, что приложение собирается под платформу Oculus, для этого необходимо поставить галочку в *Virtual Reality SDK`s* и выбрать *Oculus* (рис. 16.12).

После выполненных настроек нажимаем кнопку «*Build*» и дожидаемся окончания компиляции проекта и вывода итогового файла в разрешении **.exe*. (рис. 16.13).

Перед запуском приложения необходимо настроить на персональном компьютере *Oculus Rift*. Далее запускаем приложение и наблюдаем за работоспособностью созданной сцены в *Unity3D* под платформу *Oculus Rift*.

Контрольные вопросы

1. Принцип подключения библиотеки *Oculus* в *Unity3D*?
2. В чем отличие между *Gear VR* и *Oculus Rift*?

Лабораторная работа № 17
СОЗДАНИЕ ПРИЛОЖЕНИЯ ДЛЯ LEAP MOTION

Цель работы: создание тестового приложения для *Leap Motion* на основе *Unity3D* и *SDK LeapMotion* для изучения бесконтактного человеко-компьютерного взаимодействия.

Порядок выполнения работы

1. Ознакомиться с содержанием раздела «Методические указания».
2. Установить пакет *Leap_Motion_Setup*.
3. Протестировать корректность работы устройства.
4. Предоставить результаты проверки на одном из приложений из галереи *LeapMotion*.
5. Установка библиотеки *SDK LeapMotion* для *Unity*.
6. Разработать приложение.
7. Проверить работу приложения в *Unity*.
8. Оформить отчет.

Методические указания

Leap Motion – это сенсорное устройство, реализующее интерфейс взаимодействия с компьютером в бесконтактном режиме.

Принцип действия устройства построен на двух монохроматических камерах и трех инфракрасных светодиодах. Светодиоды выполняют подсветку рук пользователя, а камеры «захват». Полученная информация передается программным алгоритмам, которые позволяют распознавать не только ладони пользователя в целом, но и каждый из пальцев.

Установка необходимых пакетов

Для загрузи программного обеспечения *Leap Motion* переходим на сайт (https://www.leapmotion.com/setup/desktop/) и выполняем загрузку файла *Leap_Motion_Setup_XXX* (рис. 17.1). Далее подключаем устройство *Leap Motion* к порту *USB*.

После загрузки файла произведём установку скаченного ПО (рис. 17.2).

Корректность установки можно проверить по наличию значка *Leap Motion* в области уведомлений. (для *Windows*). Нажимаем на данный значок и переходим на вкладку отладки (рис. 17.3).

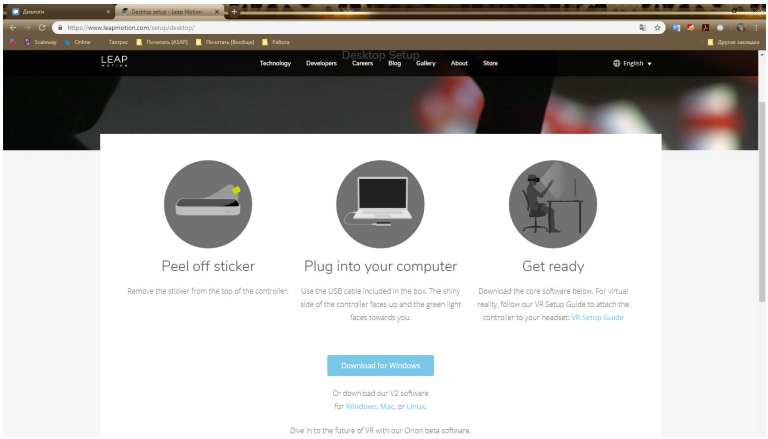


Рис. 17.1. Сайт Leap Motion

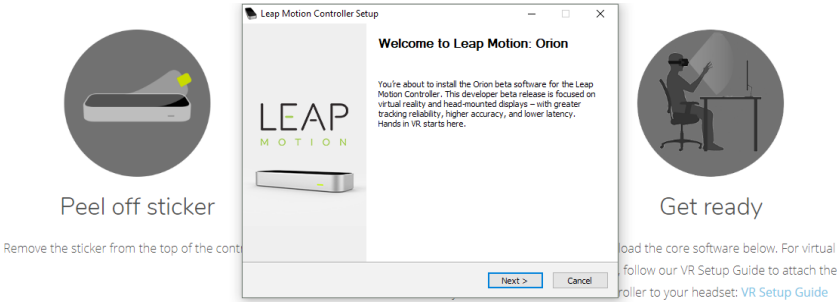


Рис. 17.2. Установка LeapMotion

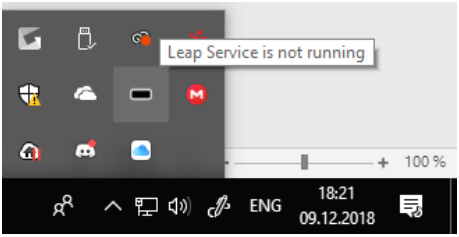


Рис. 17.3. Сервис LeapMotion в области уведомлений

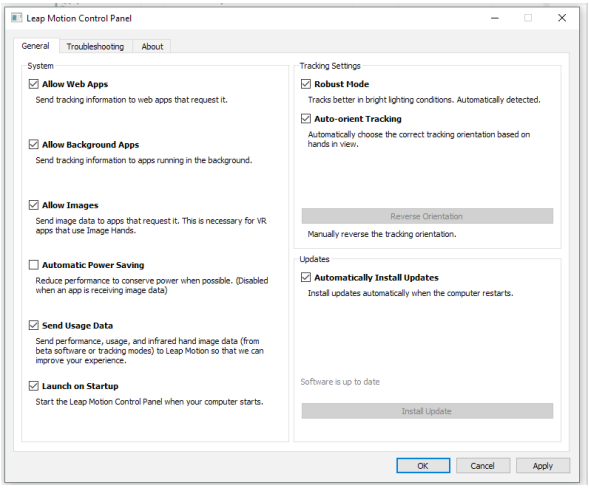


Рис. 17.4. Контрольная панель Leap Motion

На вкладке «*General*» представлены основные настройки (рис. 17.4).

Во вкладке «*Troubleshooting*» можно проверить статус устройства (рис. 17.5), а также откалибровать его, воспользовавшись режимом визуализации (рис. 17.6).

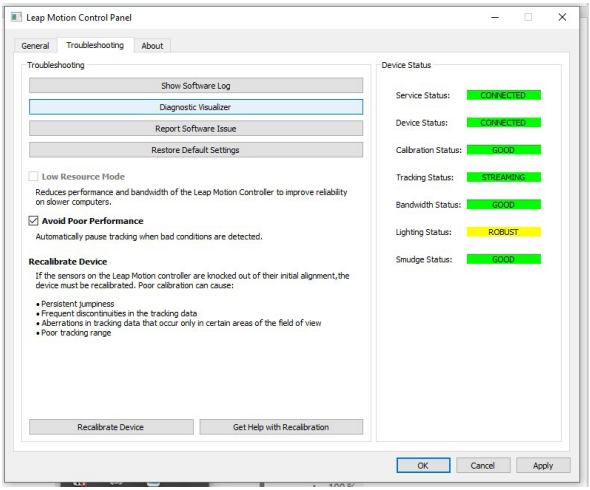


Рис. 17.5. Вкладка диагностики «Troubleshooting»

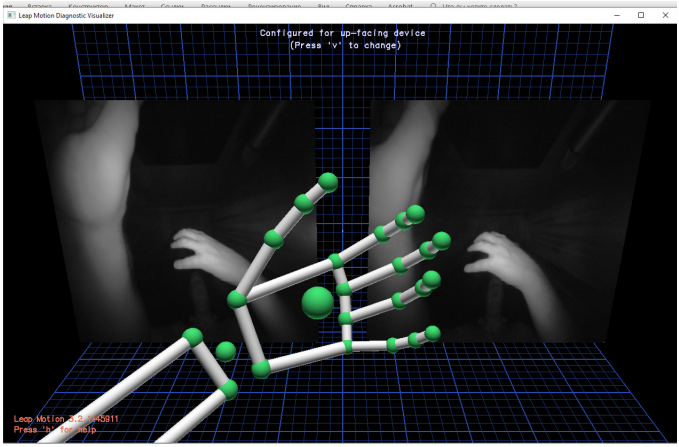


Рис. 17.6. Режим визуализации

Галерея приложений LeapMotion

Галерея приложений находится на сайте: https://gallery.leapmotion.com/category/desktop/

Большинство приложений для *Leap Motion* написано на *Unity*, поэтому проблем с запуском возникнуть не должно (рис. 17.7).

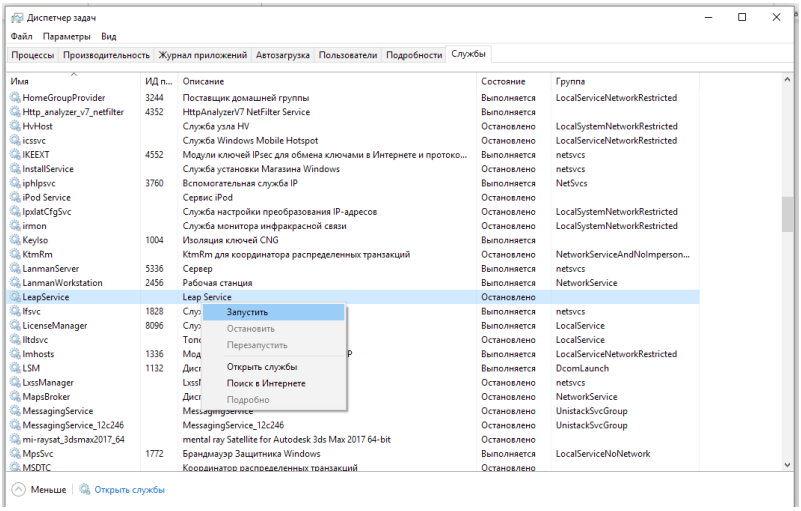


Рис. 17.7. Сервис Leap Motion не запущен

Неисправность, когда приложение «не видит» устройство, решается запуском сервиса *Leap Motion*.

Для запуска сервиса необходимо открыть Диспетчер Задач (для *Windows*) от имени администратора, перейти во вкладку «Сервисы» и найти сервис *Leap Motion*. Если сервис остановлен его необходимо запустить.

Создания приложения для LeapMotion на Unity3D

Использование игрового движка *Unity* подразумевает установку *SDK* для *LeapMotion*, который может быть загружен с официального сайта для разработчиков. (https://developer.leapmotion.com/unity/) (рис. 17.8).

Создадим новый проект *Unity*, добавив в него скачанный ассет для *SDK* (рис. 17.9).

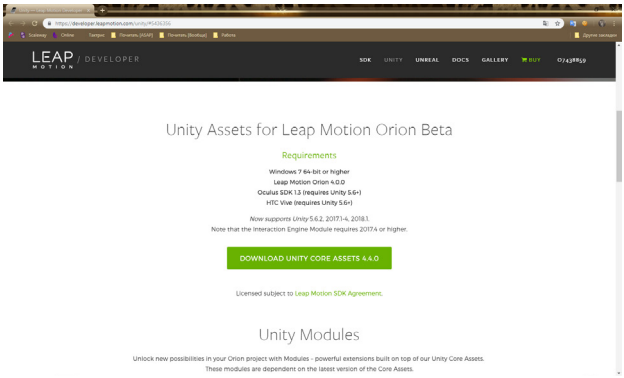


Рис. 17.8. Официальный сайт загрузчика SDK

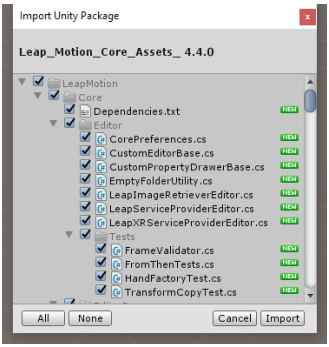


Рис. 17.9. Добавления ассетов для работы с LeapMotion

Рис. 17.10. Настройка объекта, с которым возможно взаимодействие

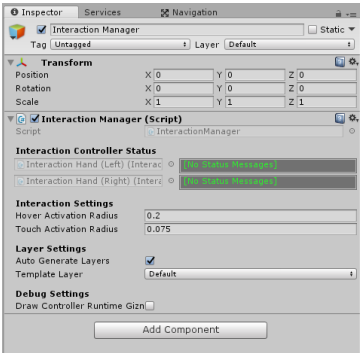


Рис. 17.11. Настройка Interaction Manager

После подключения ассетов создадим тестовую сцену, добавив несколько пребафов. Необходимо добавить на сцену контролер *Interaction Manager*, который отвечает за взаимодействие (рис. 17.11).

Далее добавим скрипт *Interaction Hand* для управления руками, а также скрипт *InteractionBehaviour* на все префабы с которыми должно быть взаимодействие (рис. 17.10).

После этого запустим приложение и переместим объекты используя *LeapMotion* (рис. 17.12, 17.13).

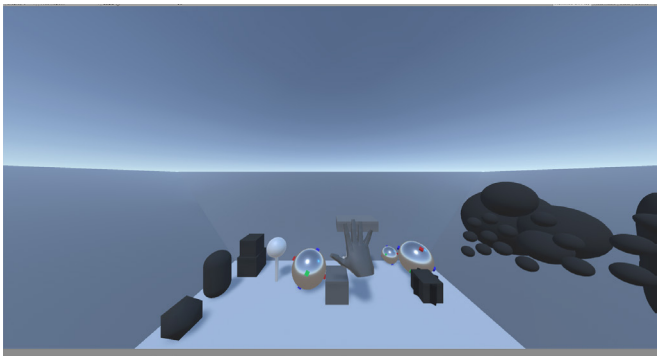


Рис. 17.12. Демонстрация результата (часть 1)

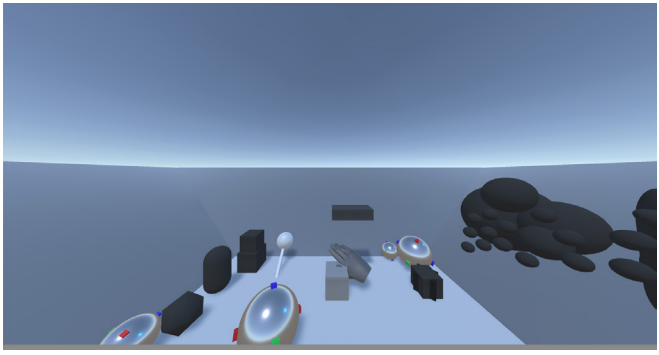


Рис. 17.13. Демонстрация результата (часть 2)

Контрольные вопросы

1. Что такое *LeapMotion* и на каких платформах доступно?
2. Каков принцип работы устройства *LeapMotion*?
3. Какие компоненты необходимы для создания приложения на *Unity?*
4. Как проверить работу приложения?

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

1. Титульный лист.
2. Цель работы.
3. Индивидуальное задание.
4. Описание основных этапов выполнения работы с необходимыми комментариями (сценарии, фрагменты настроек программ, скриншоты экрана, программный код и др.).
5. Результаты работы (таблицы значений, скриншоты экрана).
6. Выводы по результатам работы.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Основы работы в Unity3D: метод. указ. к выполнению лабораторных работ // А. Н. Жирнов, А. В. Коновалов, А. А. Преображенский. Под ред. А. В. Никитина. СПб.: ГУАП, 2014. 64 с.

2. Основы работы в Unity3D: метод. указ. к выполнению лабораторных работ // А. В. Никитин, Н. Н. Решетникова, Е. А. Полушкина, О. В. Косенко, А. А. Твердов. Ч. 2. СПб.: ГУАП, 2015. 72 с.

3. Основы разработки двух- и трехмерных приложений и игр: лабораторный практикум // В. В. Виноградов, А. В. Никитин, Н. Н. Решетникова, Н. Д. Ульянов. СПб.: ГУАП, 2016 120 с.

4. Официальный сайт Unity [Электронный ресурс]. https://unity3d.com/ru/unity (дата обращения 18.03.2019).

5. Руководство Unity [Электронный ресурс]. http://docs.unity3d.com/ru/current/Manual/index.html (дата обращения 18.03.2019).

Содержание

Учебное издание

Никитин Александр Васильевич,
Решетникова Нина Николаевна,
Ведерникова Мария Евгеньевна и др.

ОСНОВЫ РАЗРАБОТКИ ИНТЕРАКТИВНЫХ ТРЕХМЕРНЫХ ПРИЛОЖЕНИЙ НА ПЛАТФОРМЕ UNITY

Лабораторный практикум

Публикуется в авторской редакции
Компьютерная верстка *А. Н. Колешко*

Сдано в набор 11.05.18. Подписано к печати 05.09.18. Формат 60 × 84 1/16.
Усл. печ. л. 29,0. Тираж 50 экз. Заказ №

Редакционно-издательский центр ГУАП
190000, Санкт-Петербург, Б. Морская ул., 67