

Вариативная самостоятельная работа

Анализ инструментария виртуальной и дополненной реальности

1. Torch — AR development tool

Torch — это мобильное приложение AR, которое позволяет создавать интерактивные 3D-прототипы.

Его среда разработки позволяет:

- импортировать ресурсы,
- создавать сложные взаимодействия,
- организовывать несколько сцен.

Работа с 2D / 3D-файлами, инструментами или рабочими процессами не требует специальных навыков работы с 3D или программированием, что также отлично подходит для не дизайнеров.

Единственное оборудование, которое нужно для создания AR, — это мобильное устройство.

Torch предлагает доступ к функциям AR и мобильной связи:

- обнаружение изображений,
- обмен социальными сетями,
- GPS,
- голосовой ввод или оплата,
- услуги определения местоположения,
- сторонние интеграции.

Чтобы иметь доступ к активам в Torch, необходимо иметь к ним доступ с мобильного устройства, для этого нужно импортировать ресурс в проект, разместить его на сцене и расположить по своему вкусу.

2. Vectary

Приложение AR бесполезно без надлежащего контента, например, персонажей, окружение или объекты.

Vectary — это онлайн-инструмент 3D-дизайна, где можно создавать AR-контент всего несколькими щелчками мыши и экспортировать его в виде файла USDZ. Он работает во всех современных браузерах, таких как Chrome, Safari и Firefox, и имеет простой в использовании интерфейс перетаскивания. Это делает создание 3D-моделей простым и очень быстрым.

Можно создавать 3D-модели с нуля, используя бесплатный, полный набор 3D-инструментов, или выбирать из тысяч 3D-ресурсов из библиотеки Vectary.

Библиотека содержит ресурсы из

- Google Poly,
- Sketchfab,
- Thingiverse,
- MyMiniFactory.

Можно легко редактировать эти ресурсы любым удобным для способом, используя инструменты быстрого и интеллектуального моделирования Vectary.

Vectary в основном делает всю работу, без необходимости в каких-либо навыках кодирования, что делает его не только отличным генератором AR-контента, но и чрезвычайно простым в использовании конвертером.

3. Unreal Unreal Engine

Unreal Unreal Engine — игровой движок, разрабатываемый и поддерживаемый компанией Epic Games. Первой игрой на этом движке был шутер от первого лица Unreal, выпущенный в 1998 году. Хотя движок первоначально был предназначен для разработки шутеров от первого лица, его последующие версии успешно применялись в играх самых различных жанров, в том числе стелс-играх, файтингах и массовых многопользовательских ролевых онлайн-играх.

В прошлом движок распространялся на условиях оплаты ежемесячной подписки; с 2015 года Unreal Engine бесплатен, но разработчики использующих его игр обязаны перечислять 5% роялти от продаж.

Unreal Engine поддерживает разработку проектов дополненной и виртуальной реальности, но, как и Unity, он более сложен в освоении. Хотя для создания простых прототипов он не требует навыков программирования.

4. Spark AR Studio

Spark AR Studio — это платформа дополненной реальности для Mac и Windows, которая позволяет легко создавать эффекты AR для мобильных камер.

Spark AR Studio предназначен для любого человека с любым уровнем навыков. Сами эффекты могут выгружаться в Facebook или Instagram, ими можно делиться с друзьями, у себя на странице и тд. Но если получить доступ к бета-тестированию, то можно выложить созданный эффект на всеобщее использование в Instagram.

5. Amazon Sumerian

С помощью Amazon Sumerian можно легко создавать и встраивать трехмерные сцены в существующие веб-страницы.

Редактор Sumerian предоставляет готовые шаблоны сцен и интуитивно понятные инструменты перетаскивания, которые позволяют создателям контента, дизайнерам и разработчикам создавать интерактивные сцены. Разработчики, имеющие опыт работы с HTML, CSS и JavaScript, также могут писать собственные сценарии для поддержки более сложных взаимодействий.

Веб-приложения доступны через простой URL-адрес браузера и могут работать на популярном VR оборудовании. Это отличное и быстрое решение для создания 3D или VR прототипа прямо в вебе. Но Sumerian не подойдет для разработки мобильных приложений и работы с AR.

6. Unity

Unity — это игровой движок, позволяющий создавать игры под большинство популярных платформ.

С помощью данного движка разрабатываются игры, запускающиеся на

- персональных компьютерах:
 - Windows,
 - MacOS,
 - Linux,
- на смартфонах и планшетах:
 - iOS,
 - Android,
 - Windows Phone,
- на игровых консолях:
 - PS,
 - Xbox,
 - Wii.

Unity используется для разработки более 60% всего AR/VR-контента.

Сайт Unity:

“Опыт ведущих разработчиков Unity, лучших в своей отрасли, в сочетании со специальными инструментами для AR- и VR-разработчиков и партнерство с многими держателями VR/AR-платформ — это почти гарантия того, что на каждую вашу идею у нас найдутся ресурсы, специально созданные для ее воплощения в лучшем виде.”

Unity - это действительно очень функциональный, гибкий, но при этом простой, 3D-движок.

Работа с Unity

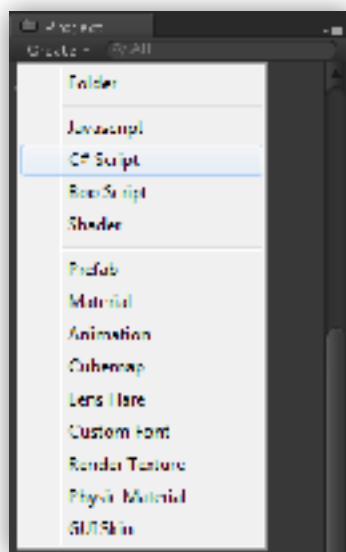
Раздел 1. Введение.

Что такое Unity3d?

Unity3d является современным кросс-платформенным движком для создания игр и приложений, разработанный Unity Technologies. С помощью данного движка можно разрабатывать не только приложения для компьютеров, но и для мобильных устройств (например, на базе Android), игровых приставок и других девайсов.

Немного о характеристиках движка. Во-первых, в среду разработки Unity интегрирован игровой движок, т.е. можно протестировать игру не выходя из редактора. Во-вторых, Unity поддерживает импорт огромного количества различных форматов, что позволяет разработчику игры конструировать сами модели в более удобном приложении, а Unity использовать по прямому назначению - разработки продукта. В-третьих, написание сценариев (скриптов) осуществляется на наиболее популярных языках программирования - C# и JavaScript.

Таким образом, Unity3d является актуальной платформой, с помощью которой можно создавать свои собственные приложения и экспортировать их на различные устройства, будь то мобильный телефон или приставка Nintendo Wii. Для того чтобы создать свою игру, вам, как минимум, нужно владеть одним из доступных (на Unity) языков программирования: C#, JavaScript или Boo.



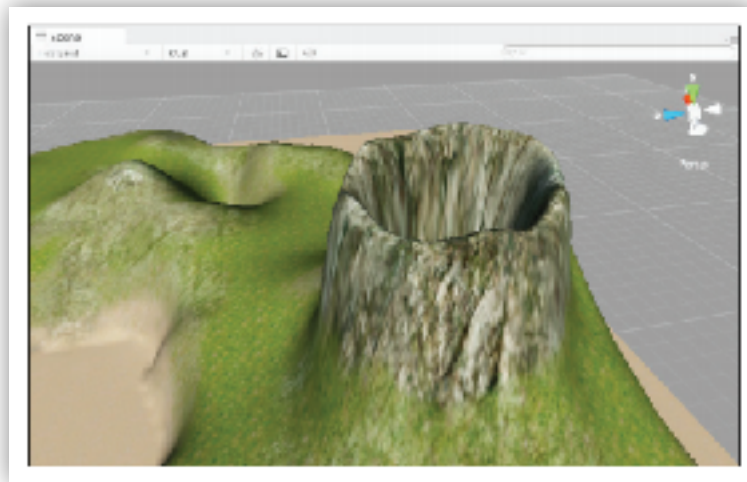
Раздел 2. Начало работы

Вспомогательная литература

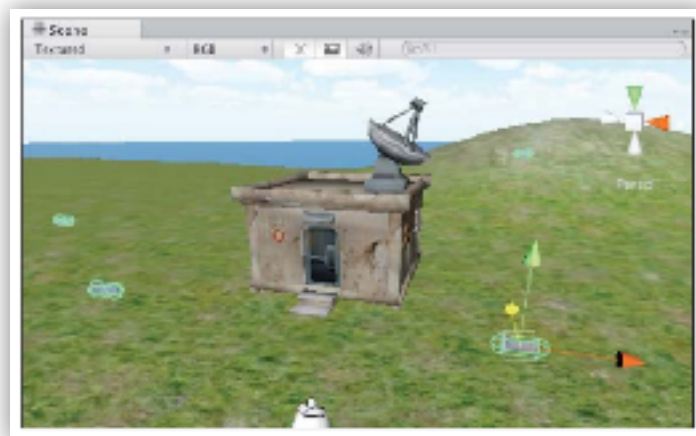
Литература, которая поможет нам изучить Unity3d, - это Unity 3.x Game Development Essentials. Книгу в свободном доступе можно найти в гугле.

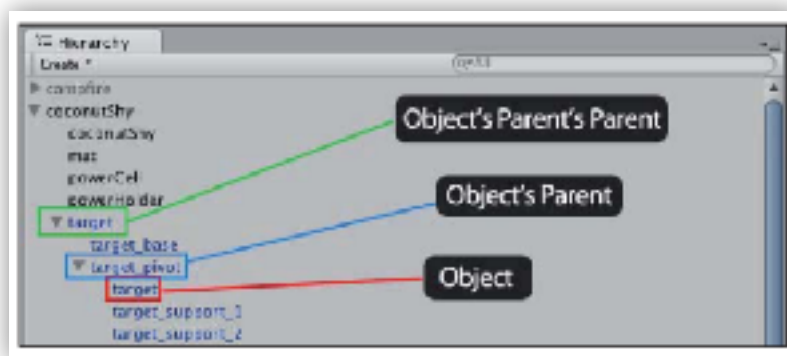
Пару слов об игре и содержании книги. Игрок оказывается на необитаемом острове, ему необходимо спастись, для этого он выполняет различные задания и, будем надеяться, спасается. Игру можно пройти за пару минут, однако создание игры занимает далеко не пару минут, даже не пару часов, пожалуй.

Книга содержит очень подробное руководство по созданию игры "с нуля". В ней описано многое, что, скорее всего, понадобится для создания игры.



Книга написана на английском языке; тем, кто не знает английский, будет не сложно интуитивно догадаться о чем идет речь, ибо написана книга без особых премудростей.





NB

° Коллайдеры

Что это такое - коллайдер? Коллайдер - это область пространства, при взаимодействии с которой выполняются те или иные скрипты, действия. Чтобы лучше понять что это такое, представьте, что вы подходите к автоматически открывающимся дверям универсама. За пару метров от дверей, срабатывает датчик и двери распахиваются перед вами, но если вы пройдете за три метра, то ничего не произойдет. Как раз та зона, в которой действует датчик движения, отвечающий за открывание дверей, и есть коллайдер. Вы входите в зону - двери открываются, выходите из зоны - и датчик уже никак не будет реагировать на ваше присутствие. Точно так же и в игре.

Так вот, о коллайдерах. Предположим, мы импортировали в наш проект автомобиль и хотим сгенерировать для него коллайдер. Можно сгенерировать автоматически коллайдер для всего автомобиля, но тогда для каждой детали автомобиля - фары, уплотнителя, зеркала, крышки - будет сгенерирован свой коллайдер.



Это абсолютно не оптимально. Безусловно, нам не нужно генерировать столь большое количество коллайдеров, достаточно ограничиться одним!



Обратите внимание, что это не тонкость. Стараться сэкономить память нужно везде. У вас есть какое-то место в игре, куда игрок не сможет попасть? Не генерируйте никакие коллайдеры для этого места вообще. У вас есть дверь, которую вы импортировали в проект из 3Ds max'a? Небоось, она очень хорошо прорисована (сам сталкивался с тем, что у двери даже шурупы были прорисованы отдельным объектом), значит, создайте один box-коллайдер для всей двери - вы сэкономите много памяти! Поверьте, что вам эта сэкономленная память еще очень и очень пригодится.

° Оптимальная работа со сценами

Поговорим об оптимизации сцен, о самих же сценах будет речь вестись далее. Представьте следующую ситуацию: у нас есть сцена "home" - когда игрок находится в здании и сцена "street" - когда игрок выходит из здания на улицу. Естественно, из второй сцены мы можем видеть дом, т.е. часть первой сцены, и наоборот (если, конечно, у здания есть окна). В таком случае оптимально следующее построение сцен. В первой сцене все, что касается улицы (т.е. того места, куда мы не можем попасть без перехода на другую сцену) надо максимально упростить. Иными словами, нам нужно оставить только внешний вид вида из окна, а всю начинку - распотрошить. Т.е. все коллайдеры удалить, разрешение и т.п. свести к минимуму. Действительно, зачем процессору напрягаться с обработкой той местности, которую мы не можем посетить? Аналогичным образом поступаем со второй сценой ("street"). В данном случае мы можем вообще все удалить из дома и оставить только его «коробку».

Вы хотите создать minimap? Пожалуйста, только избавьтесь от всех коллайдеров и снизьте разрешение, в таком случае ваша игра будет оптимальна.

Именно таким образом работа со сценами становится оптимальной. Еще раз отмечу, что это не тонкости, это серьезные вещи, которые требуют внимания создателя.

Сцены

Готовая игра - это набор сцен, соединенных между собой (точно так же, как и жизнь - это набор дней). Об оптимальном проектировании игры мы сейчас и поговорим.

Прежде чем создать свой проект - подумайте, что он должен в себя включать (какие сцены). Составьте список сцен (на листочке), обдумайте, что каждая сцена будет в себе содержать. Теперь постарайтесь каждую сцену разбить на подсцены, чем больше их будет, тем легче будет вашему ЦП. Конечно, не стоит перебарщивать с количеством сцен. Предположим, что ваш игрок находится в здании и из него никогда не выходит, в здании есть, например, пару этажей, крыша, а на каждом этаже имеется три комнаты. В данном случае, по моему мнению, было бы оптимально "разбить" здание на две сцены - два этажа и крыша. Дробление же каждого этажа на сцены с комнатами - не оптимально. Это просто не нужно, если каждая из комнат слабо загружена. Иными словами, надо грамотно расходовать память процессора, но и забывать о получении удовольствия от игры тоже не стоит.

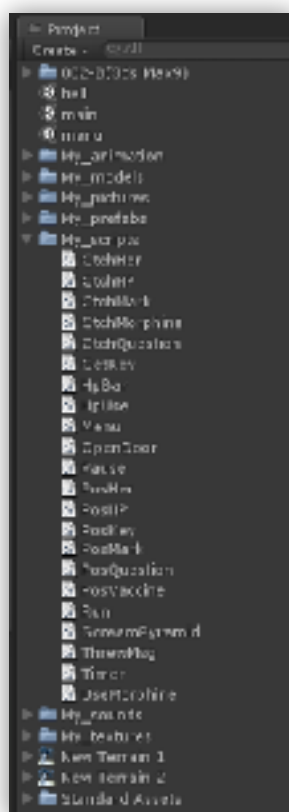
Отладка

Помните, что хороший создатель должен учитывать то, что его игра, либо приложение, будет использоваться на различных ЭВМ, а это означает, что свой контент необходимо отладить так, чтобы он мог использоваться на различных устройствах. Самый очевидный и важный момент отладки - оптимизация приложения под различные разрешения экранов. Это важный момент и создатель игры обязательно должен обратить на него свое внимание, в противном случае, результатом своей работы сможет быть доволен только он один.

Структурирование

При конструировании игры важно помнить о структурировании данных - немаловажном аспекте. Имеется ввиду, что все должно быть на своем месте. Создаете скрипты? - создайте папку для скриптов, там и храните их, с большой вероятностью, у вас будет не один скрипт, разбрасывать их по всему проекту не есть хорошо, сами потом будете мучиться. По своему опыту могу сказать, что вам точно понадобится создать папки со скриптами, звуковыми файлами, с GUI текстурами, материалами, анимациями, быть может, префабами и собственными моделями (импортированные из 3Ds max'a объекты лучше хранить именно в последней папке, дабы не разбрасывать все модели по

проекту в хаотичном порядке). Когда вы создаете тот или иной скрипт, помните о том, что название переменных, классов и т.п. должны говорить все сами за себя. Если вы хотите создать скрипт для инвентаря (поднять предмет, использовать предмет), то лучше все скрипты однообразных действий (например, поднятия) начинать с ключевого слова, например: `Catch<имя предмета>`; `Use<имя предмета>`, в таком случае вам будет легче ориентироваться в проекте, да и выглядит так намного симпатичнее.



Раздел 3. Практические советы

Помните золотое правило: хорошая программа не та, которая написана тят-ляп и отлажена до предела, хорошая программа та, которая пишется сразу правильно и требует минимальных "затрат" на отладку. Если вы будете пользоваться этим правилом, то у вас все будет намного лучше чем у тех, кто этим правилом не пользуется.

Написание скриптов

Увы, этот подраздел будет совсем короткий. Просто старайтесь сначала представить то, что вы хотите реализовать, затем попробуйте описать словами алгоритм, а далее - преобразовать слова в работающий код. Используя книгу, о которой речь шла выше, вам навряд ли придется сильно потеть над разработкой кодов и других вещей, повторюсь, что в книге описано практически все, что только может понадобиться, а все, что не описано - легко додумать самому.

Видеоуроки

Рекомендую следующие youtube каналы для ознакомления:

1. www.youtube.com/user/4GameFree
2. www.youtube.com/user/FlightDreamStudio
3. www.youtube.com/user/3DBuzz

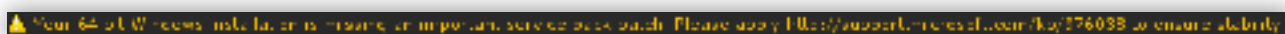
К счастью, программный продукт Unity обновляется постоянно, но, увы, видеоуроки сами это делать не могут. Получается так, что огромное количество видеороликов, актуальных год назад, теряют свою актуальность и становятся, попросту, бесполезными.

1 - здесь вы всегда найдете актуальные видеоуроки, понятное объяснение, детальный разбор скриптов и всех действий. Лучше этого канала пока что ничего не нашел. Если вы не имеете вообще никакого представления о создании игры, то вам однозначно нужно заглянуть на данный канал. 2 - серьезная команда, видеоуроков только, жаль, меньше, чем у 1. 3 - есть полезные вещи, жаль только, что видеоуроки стали почти неактуальными.

Устранение ошибок

Unity3d - такой движок, что если у вас имеются ошибки в написании скриптов, то игра ваша не запустится. Если же нет явных ошибок, то игра запускается, если что-то идет не так, то консоль вам об этом обязательно сообщит.

Это предупреждения, они просто говорят вам о возможных недочетах, недоработках:



⚠ Your 64-bit Windows installation is missing an important security patch. Please apply <https://support.microsoft.com/kb/276088> to ensure stability.

Это ошибки, если такие имеются, то игра не запустится:



❗ Assets/My_scripts/Ctchiller.cs(21,26): error CS1525: Unexpected symbol '<interia >'

Раздел 4. Завершение работы

Выводы

Unity3d - очень гибкий движок, предоставляющий большую свободу действий пользователю. Чтобы упростить жизнь разработчику и потребителю, можно предпринять некоторые шаги для оптимизации своего проекта.

Теперь мы, с заложенной базой знаний, можем спокойно приступить к созданию своего ультра-популярного приложения!



Публикация игры

Свою готовую игру можно опубликовать на одном из игровых сервисов, например, на www.kongregate.com/

*Оптимизация

Если вы читаете данный подраздел, то либо вам просто интересно, что здесь написано, либо вы не прочитали NB! из раздела 2. Будем надеяться, что вы относитесь к первой категории. В данном подразделе еще раз хочется отметить то, что нужно сразу писать игру оптимальной, а не писать ее хоть как-то, и только затем отлаживать - это гиблый путь и поступать так не стоит. По поводу оптимизации - еще раз обратитесь ко второму разделу.

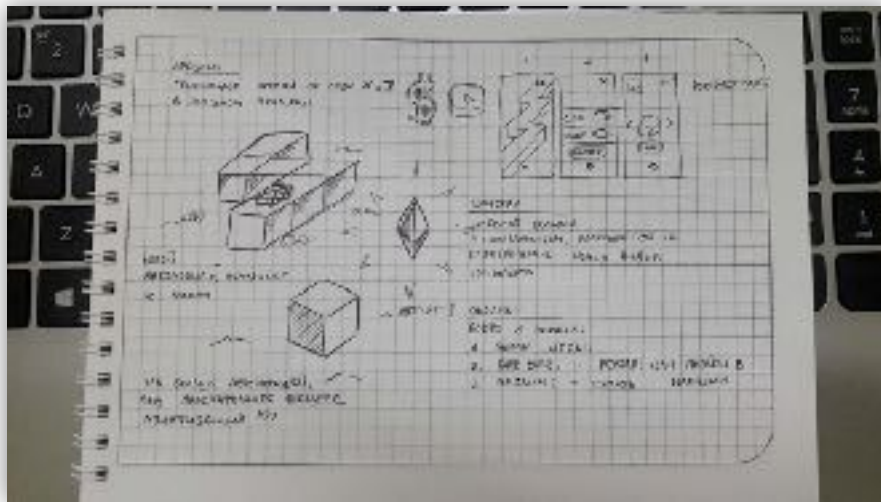
Мобильная 3D игра на Unity3D

Этап-1: создание наброска

Набросок – это видение продукта. Это техническое задание будущей игры.

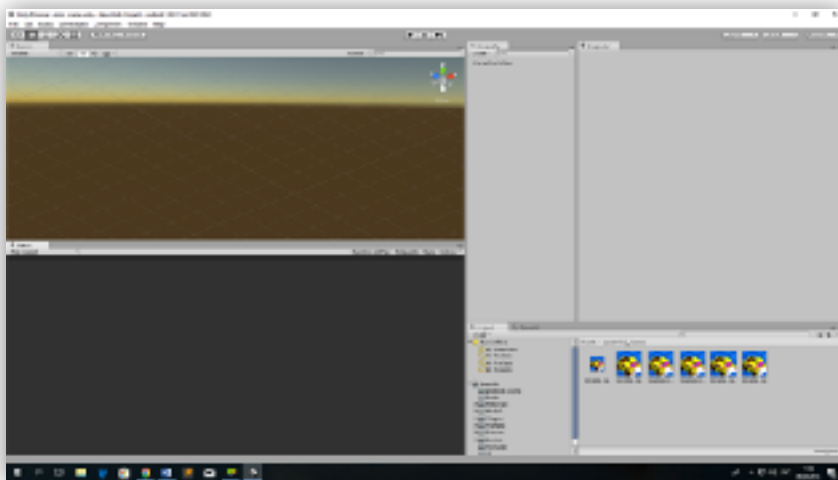
Из наброска видно, что игра предназначена для мобильных платформ, и запускается она будет в портретном режиме. Геймплей также бесхитростен: задача игрока заключается в преодолении опасного маршрута на предоставленном игрой автомобиле, попутно собирая кристаллы. За каждый

собранный кристалл и удачно пройденный поворот, игрок получает вознаграждение в виде очков бонуса. Касание по экрану заставляет изменять направление движения автомобиля по осям X и Z.



Этап-2: создание прототипа

Имея под рукой подробный план действий, можно смело приступать к созданию «мокапа» или прототипа будущей игры. По сути, данный этап – начало работы с Unity, и начинать его следует с настройки окружения.

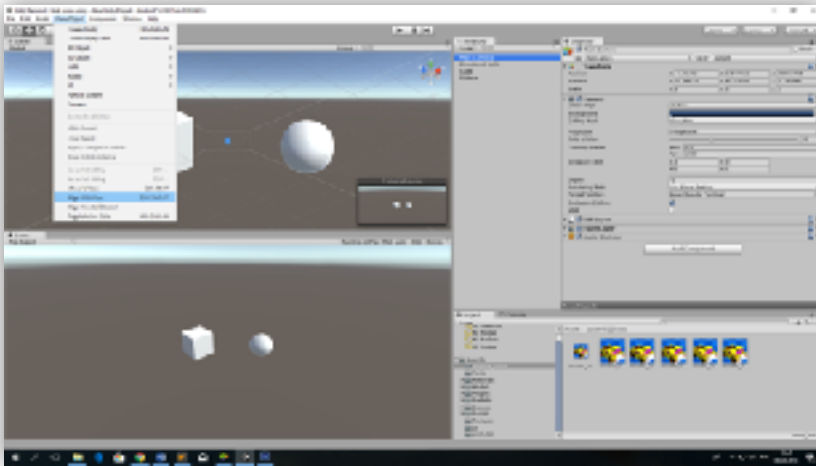


В левой части экрана расположились редактор Scene и Game. Последний отображает то, как именно игра выглядит на устройствах. В правой части: панели Hierarchy и Inspector, а чуть ниже расположены панели Project и Console.

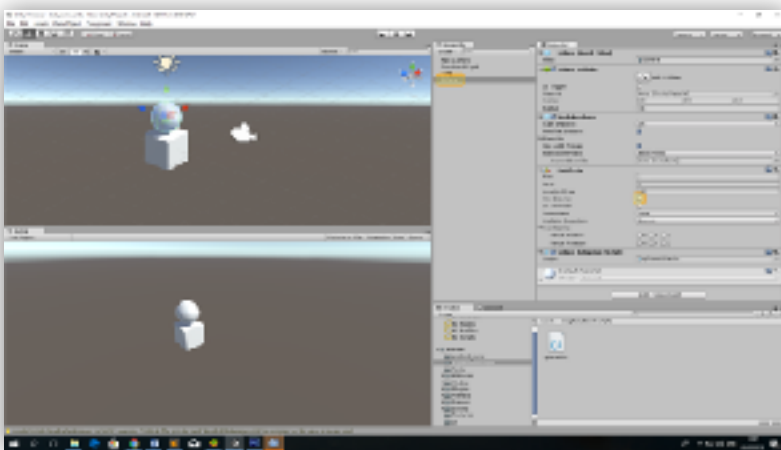
Этап-2.1: под капотом

Прототипирование всегда начинается с болванок, то есть в качестве актеров используются примитивные элементы, вроде кубов и сфер. Такой подход заметно упрощает процесс разработки, позволяя абстрагироваться от всего, что не связано с механикой игры. На первом шаге формируется базовое понимание

облика будущей игры, а так как игра создается в изометрическом стиле, первое что необходимо проделать, это настроить камеру. Это одна из ключевых особенностей Unity. Дело в том, что можно долго экспериментировать с параметрами настройки камеры, подбирая нужные значения... Но проще просто выставить понравившийся ракурс с помощью панели View, а затем активировать GameObject -> Align With View, после чего камера тотчас же примет необходимые значения. Вот такой вот shortcut от создателей Unity.



Итак, сцена готова, но как придать персонажу движение? Для начала, произведем некоторые манипуляции с объектом Sphere, добавив в него такие компоненты как Rigidbody и только что созданный скрипт sphereBehavior. Не забыть отключить галочку Use Gravity, так как на данном этапе он не понадобится.



Компонент Rigidbody позволяет объекту ощутить на себе все прелести физического мира, таких как масса, гравитация, сила тяжести, ускорение и.т.д. А теперь, чтобы заставить тело двигаться в нужном направлении, всего лишь нужно слегка изменить параметр velocity, но делать это нужно при помощи кода. Чтобы заставить сферу двигаться по оси X, нужно внести изменения в скрипт sphereBehavior:

```

using UnityEngine;
using System.Collections;

public class sphereBehavior : MonoBehaviour {

    private Rigidbody rb; // Объявление новой переменной Rigidbody
    private float speed = 5f; // Скорость движения объекта

    void Start() {
        rb = GetComponent<Rigidbody> (); // Получение доступа к Rigidbody
    }
    void Update() {
        rb.velocity = new Vector3 (speed, 0f,0f);
    }
}

```

В Unity, тела описывают своё положение и направление, посредством специальных векторов, хранящих значения по осям x, y и z. Изменяя эти значения, можно необходимого направления или положения конкретного тела. Строка `rb.velocity = new Vector3(speed, 0f,0f)` задает новое направление телу по оси X, тем самым придавая сфере нужное направление.

Если все сделано, как описано, то сфера отправится в бесконечное путешествие по оси X, со скоростью `speed`.

Теперь давайте заставим сферу изменять свое направление, при каждом клике левой клавиши мыши так, как это реализовано в игре ZIGZAG. Для этого нужно вернуться к коду `sphereBehavior` и изменить его следующим образом:

```

using UnityEngine;
using System.Collections;

public class sphereBehavior : MonoBehaviour {

    private Rigidbody rb; // Объявление новой переменной Rigidbody
    private bool isMovingRight = true; // переменная, отражающая условное
направление объекта
    private float speed = 5f; // Скорость движения объекта

    void Start() {
        rb = GetComponent<Rigidbody> (); // Получение доступа к Rigidbody
    }

    void changeDirection() {
        if (isMovingRight) {
            isMovingRight = false;
        } else {
            isMovingRight = true;
        }
    }

    void Update() {

        if(Input.GetMouseButtonDown(0)) {
            changeDirection();
        }

        if (isMovingRight) {
            rb.velocity = new Vector3 (speed, 0f, 0f);
        }
    }
}

```

```

    } else {
        rb.velocity = new Vector3 (0f, 0f, speed);
    }
}

```

Пусть когда сфера движется по оси X, то это движение называется движением «вправо», а по оси Z — «влево». Таким образом можно легко описать направление нашего тела специальной булевой переменной `isMovingRight`.

```

if(Input.GetMouseButtonDown(0)) {
    changeDirection();
}

```

Этот кусочек кода отслеживает нажатие левой клавиши мыши, и если данная клавиша все же была нажата, запускает функцию `changeDirection()`, с простой логикой: если на момент нажатия левой клавиши мыши, переменная `isMovingRight` имела значение `true`, то теперь она стала `false` и наоборот. Напомню, что булевая переменная позволяет нам ответить на один простой вопрос: истинно ли утверждение о том, что тело движется по оси X, или нет? Иными словами, нажатие на левую клавишу мыши постоянно изменяет значение `isMovingRight`, то на `true`(тело движется вправо), то на `false`(тело движется влево).

Альтернативно, функцию `changeDirection()` можно записать в одну строку:

```

void changeDirection() {
    isMovingRight = !isMovingRight;
}

```

И последнее, что необходимо сделать, это переписать метод направления движения с учетом переменной `isMovingRight`:

```

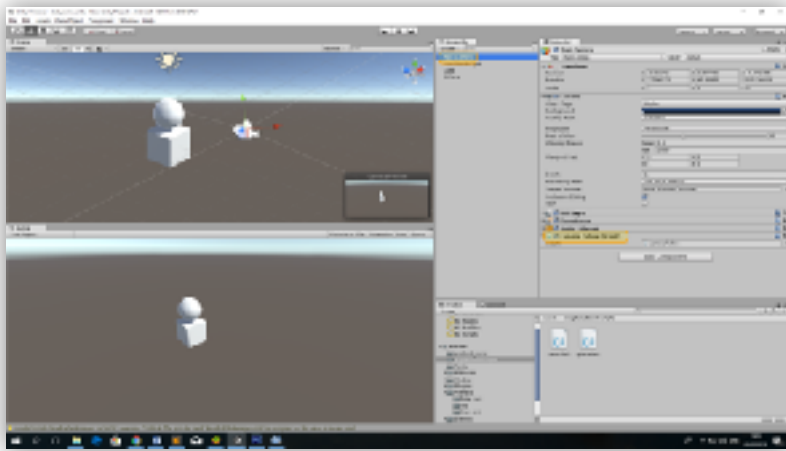
if (isMovingRight) {
    rb.velocity = new Vector3 (speed, 0f, 0f);
} else {
    rb.velocity = new Vector3 (0f, 0f, speed);
}

```

Если `isMovingRight` имеет значение `true` (если сфера действительно движется вправо), тогда значение `velocity` принимает новый вектор направления `rb.velocity = new Vector3 (speed, 0f, 0f)`; Если `isMovingRight` имеет значение `false`, значит тело более не движется вправо, а значит пришло время изменить вектор направления на `rb.velocity = new Vector3 (0f, 0f, speed)`;

Запустить игру, проделать несколько кликов мыши, и если все сделано в точности, как здесь, то сфера начнет описывать зигзаги.

Нужно доработать игру так, чтобы можно было двигаться вместе со сферой и не упускать её из виду. Для этого нужно создать скрипт cameraFollow и прикрепить его к объекту Main Camera:



А вот код скрипта cameraFollow:

```
using UnityEngine;
using System.Collections;

public class cameraFollow : MonoBehaviour {

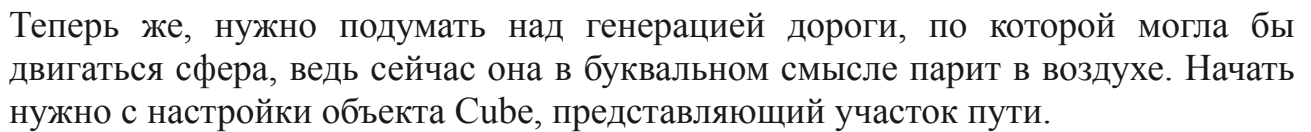
    public GameObject player;
    public Vector3 offset;

    void Start () {
        offset = transform.position - player.transform.position;
    }

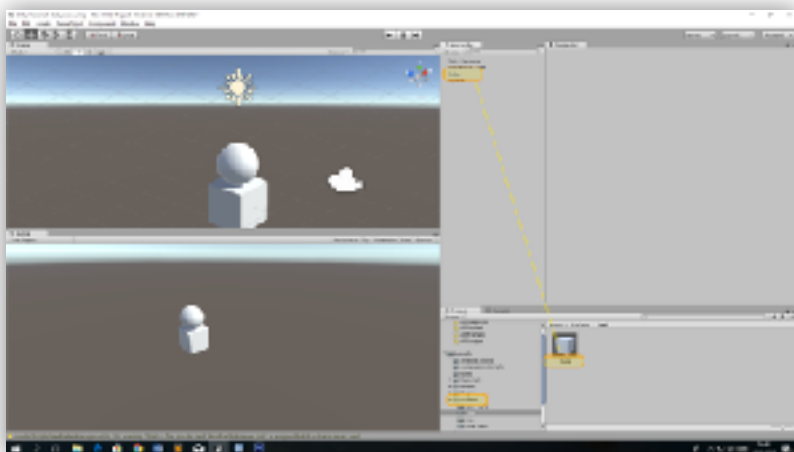
    void Update () {
        transform.position = player.transform.position + offset;
    }

}
```

Как же осуществить слежение за объектом? Для начала нужно рассчитать разницу смещения между объектами Camera и Sphere. Для этого достаточно вычесть от позиции камеры, координаты сферы, а полученную разницу сохранить в переменной offset. Но прежде, необходимо получить доступ к координатам сферы. Для этого необходима переменная player, представляющая собой простой GameObject. Так как сфера находится в постоянном движении, нужно синхронизировать координаты камеры с координатами сферы, приплюсовав полученное ранее смещение. Осталось только указать в поле player объект слежения. Нужно просто перетащить объект Sphere в поле Player, скрипта cameraFollow, как это показано на картинке (Main Camera при этом должна оставаться выделенной):

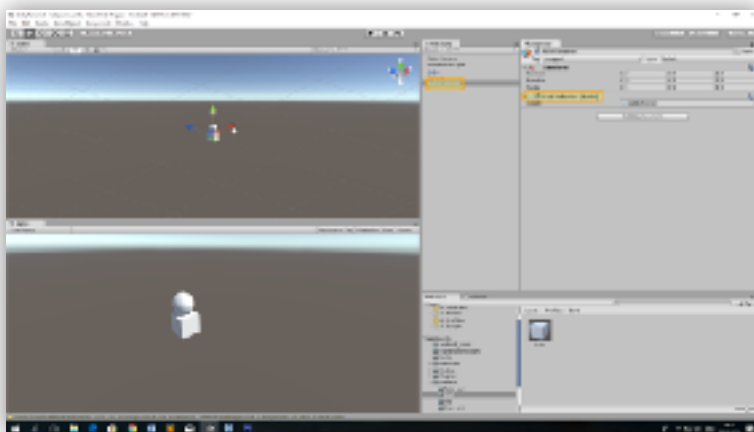
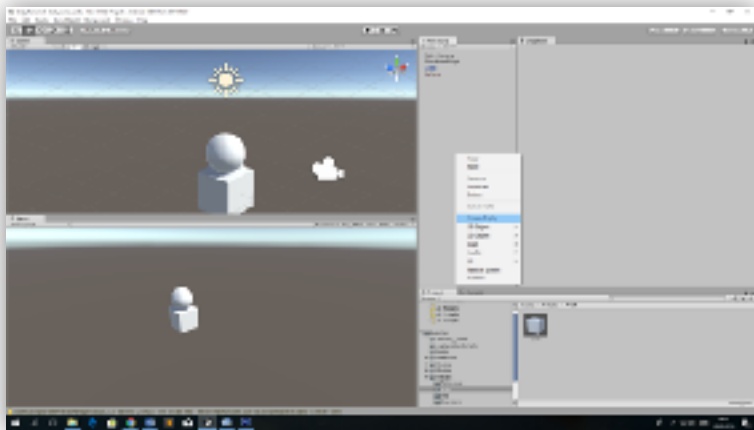


Следующим нужно создать в корне проекта специальную папку с названием Prefabs, и перетащить в нее созданный Cube, прямо из инспектора. Если после этого, имя объекта Cube стало синего цвета, значит все сделано правильно.



Префабы – это особый тип объектов, позволяющий хранить GameObject, а также все его значения и свойства в одном месте. Префабы позволяют создавать бесконечное множество объекта, а любое его изменение немедленно отражаются на всех его копиях. Иными словами, теперь можно вызывать участок пути Cube, прямо из папки Prefabs, столько раз, сколько необходимо.

Теперь нужно создать пустой `GameObject`, (щелчок правой кнопки мыши по `Hierarchy`) переименовать его в `RoadContainer` и прикрепить к нему только что созданный скрипт `roadBehavior`:



А вот и сам код roadBehavior:

```
using UnityEngine;
using System.Collections;

public class roadBehavior : MonoBehaviour {

    public GameObject road; // Префаб участка пути
    private Vector3 lastpos = new Vector3 (0f,0f,0f); // Координаты
    // установленного префаба

    void Start() {

        for(int i=0; i<10; i++) {
            GameObject _platform = Instantiate (road) as GameObject;
            _platform.transform.position = lastpos + new Vector3 (1f,0f,0f);
            lastpos = _platform.transform.position;
        }
    }
}
```

```

    }
}
}

```

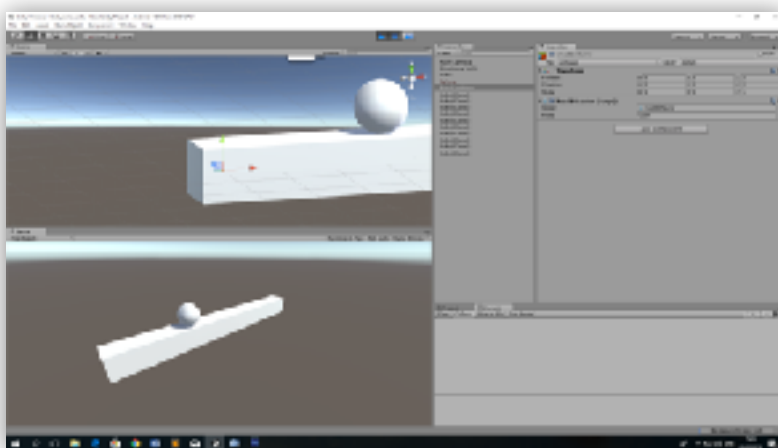
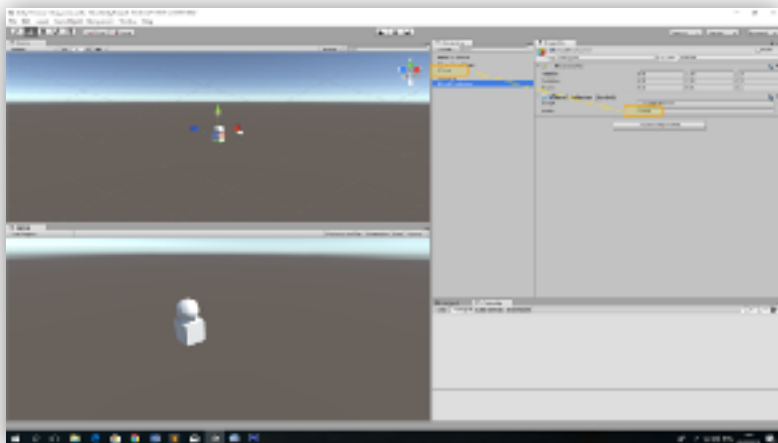
Что же тут на самом деле происходит? Есть переменная, которая позже, вручную будет привязана к префабу Cube, и есть объект Vector3, хранящий координаты последнего установленного префаба (сейчас значения равны нулю).

```

for(int i=0; i<10; i++) {
    GameObject _platform = Instantiate (road) as GameObject;
    _platform.transform.position = lastpos + new Vector3 (1f,0f,0f);
    lastpos = _platform.transform.position;
}

```

Этот участок кода выполняет следующее: до тех пор, пока $i < 10$, берется префаб, устанавливается его позиция с учетом последней позиции lastpos + позиция с учетом смещения по X, сохраняется последняя позиция. То есть в результате, получается 10 префабов Cube, установленных в точности друг за другом. Перед проверкой, нельзя забыть назначить переменной road объект Cube из папки Prefabs:



Дальше нужно продолжить установку блоков в произвольном порядке. Для этого понадобится генератор псевдослучайных чисел random. Нужно подправить скрипт roadBehavior с учетом нововведений:

```
using UnityEngine;
using System.Collections;

public class roadBehavior : MonoBehaviour {

    public GameObject road; // Префаб участка пути
    private Vector3 lastpos = new Vector3 (0f,0f,0f); // Координаты
    установленного префаба

    void Start() {

        for(int i=0; i<10; i++) {
            GameObject _platform = Instantiate (road) as GameObject;
            _platform.transform.position = lastpos + new Vector3 (1f,0f,0f);
            lastpos = _platform.transform.position;
        }

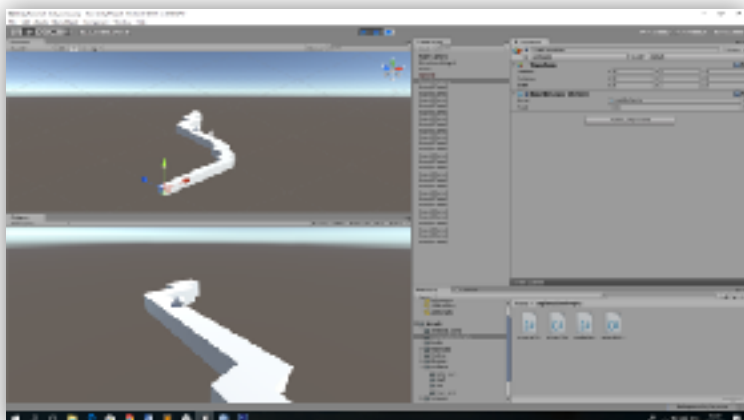
        InvokeRepeating ("SpawnPlatform", 1f, 0.2f);

    }

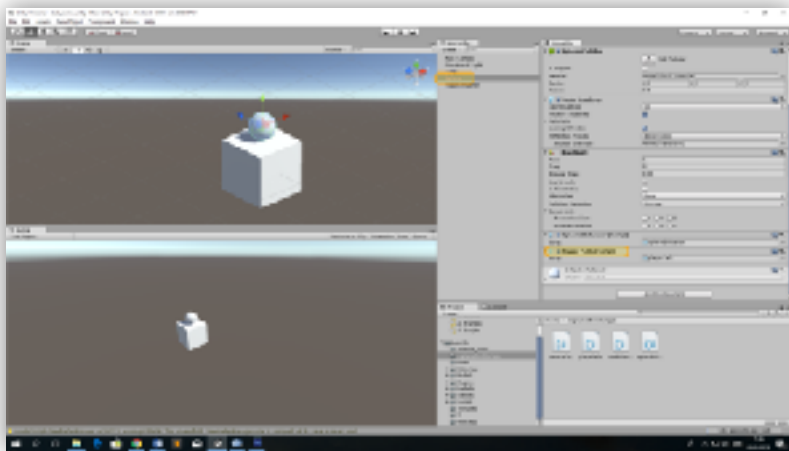
    void SpawnPlatform() {

        int random = Random.Range (0, 2);
        if (random == 0) { // Установить префаб по оси X
            GameObject _platform = Instantiate (road) as GameObject;
            _platform.transform.position = lastpos + new Vector3 (1f,0f,0f);
            lastpos = _platform.transform.position;
        } else { // Установить префаб по оси Z
            GameObject _platform = Instantiate (road) as GameObject;
            _platform.transform.position = lastpos + new Vector3 (0f,0f,1f);
            lastpos = _platform.transform.position;
        }
    }
}
```

Строчка InvokeRepeating («SpawnPlatform», 1f, 0.2f) предназначена для активации функции SpawnPlatform() спустя 1 секунду после начала игры, и повторного её вызова каждые 0.2 секунды. Каждые 0.2 секунды, система загадывает случайное число между цифрами от 0 до 1. Если система загадала 0 – устанавливается новый префаб по оси X, а если 1 – то по оси Z.



И наконец, можно заставить сферу падать каждый раз, когда она сходит с дистанции. Для этого нужно создать новый скрипт `playerFalls` и прикрепить его к объекту `Sphere`:



А вот и сам код скрипта `playerFalls`:

```
using UnityEngine;
using System.Collections;

public class playerFalls : MonoBehaviour {

    private Rigidbody rb;

    void Start() {

        rb = GetComponent<Rigidbody> ();

    }

    void Update() {

        RaycastHit hit;
        if(Physics.Raycast (transform.position, Vector3.down, out hit, 5f) &&
hit.transform.gameObject.tag == "Ground") {
            rb.useGravity = false;
        } else {
            rb.useGravity = true;
        }
    }
}
```

`Raycast` — специальный луч на подобии лазера, который излучается по направлению к сцене. В случае, если луч отражается от объекта, он возвращает информацию об объекте, с которым столкнулся. И это очень круто, потому что именно так, посредством такого луча, направленного из центра сферы вниз, будет проверяться нахождение, на платформе `Cube` или нет. И как только вы покидаете регионы дорожного полотна, то автоматически активируется параметр `Gravity` сферы после чего сфера, под воздействием гравитации, рухнет вниз.

Этап-3: визуальный дизайн

Когда все работы по игровой механике закончены, в пору переходить к визуальной части проекта. Все таки геймплей – это хорошо, а приятный геймплей – еще лучше. И несмотря на то, что в самом начале графика была обозначена, как далеко не самое главное, хочется все же добавить красок в создаваемую игру. Например:



Все ассеты были смоделированы в Blender'е – бесплатном 3D редакторе. В нём же были созданы необходимые текстуры, впоследствии доведенные до приемлемого вида в Photoshop'е. Приятным моментом оказалось то, что Unity легко импортирует 3D модели из Blender'а, без лишней головной боли, делая процесс создания приятным и безболезненным.

Этап-4: полировка

Доводка проекта – те еще грабли, ведь всегда найдется место тому, что можно улучшить, или переделать. Зачастую, случается так, что именно на этапе полировки и доводки, процесс разработки значительно теряет во времени, а то и вовсе заходит в тупик. Лучше дополнять игру уже после релиза, путем выкатки обновлений, чем затягивать разработку на неопределенный срок. В противном случае, есть риск погубить проект.

Последним, остается создать презентационные документы: краткое описание, видео, а также иконку игры. Этому этапу следует уделить как можно больше внимания, ведь именно по иконке, пользователи начинают судить проект.