Департамент физической культуры, спорта и дополнительного образования
Тюменской области
ГАУ ДО ТО «Дворец творчества и спорта «Пионер»

Пушкарев А.Н.

ОСНОВЫ ПРОГРАММИРОВАНИЯ В СРЕДЕ UNITY

Учебно-методическое пособие для преподавателей и учащихся
учреждений дополнительного образования детей и молодежи

Тюмень 2023

Пушкарев А.Н. ОСНОВЫ ПРОГРАММИРОВАНИЯ В СРЕДЕ UNITY: Учебно-методическое пособие для преподавателей и учащихся учреждений дополнительного образования детей и молодежи. – Тюмень: ГАУ ДО ТО «Дом детского творчества и спорта «Пионер», 2023. – 165 с.

В данном учебно-методическом пособии излагаются основы создания и программирования трёхмерных сцен в среде разработки графических приложений Unity.

Учебно-методическое пособие адресовано педагогам и учащимся учреждений дополнительного образования детей и молодежи.

© ГАУ ДО ТО «Дом детского творчества и спорта «Пионер»

Оглавление

Введение

Развитие современных информационных технологий сопровождается появлением новых стандартов, предъявляющих всё более высокие требования к создаваемым цифровым продуктам. Одно из них – наличие удобного пользовательского интерфейса, позволяющего человеку легко взаимодействовать с электронно-вычислительным устройством. Применительно к сфере разработки компьютерных программ указанное требование подразумевает организацию графического представления результатов работы приложения. Например, решение определённого круга задач, таких как разработка компьютерных игр, сцен виртуальной и дополненной реальности, предполагает создание двумерных и трёхмерных объектов и настройку способов взаимодействия с ними. Облегчить работу по созданию приложений с элементами визуализации помогают специализированные компьютерные системы – среды разработки.

В настоящем учебно-методическом пособии излагаются основы создания и программирования трёхмерных сцен в среде разработки графических приложений Unity.

Цель данного пособия – помощь учащимся в освоении основ создания и программирования трёхмерных сцен в среде разработки графических приложений Unity, а также помощь педагогам учреждений дополнительного образования детей и молодежи в проведении занятий по данной теме.

В рамках указанной цели решаются следующие задачи:

– познакомиться с процессом установки Unity и регистрации учётной записи;

– научиться использовать инструменты настройки объектов трёхмерных сцен;

– получить навык программного изменения объектов;

– освоить настройку основных элементов пользовательского интерфейса.

Каждая из указанных задач подробно рассмотрена в соответствующей части учебно-методического пособия: «Подготовка к работе», «Создание трёхмерной сцены», «Программное изменение объектов», «Настройка пользовательского интерфейса».

Кроме того, структура настоящего пособия содержит введение и заключение.

Глава 1. Подготовка к работе

1.1. Установка Unity

Unity – межплатформенная среда разработки компьютерных игр и других приложений, использующих визуализацию в форматах 2D и 3D. Программа Unity позволяет создавать приложения для более чем 25 различных платформ – персональных компьютеров, игровых консолей, мобильных устройств, интернет-приложений и многих другие. Первая версия Unity вышла в 2005 году, и в последние годы данный программный продукт стремительно совершенствуется, становясь всё проще в освоении и предоставляя всё больше новых возможностей для разработчиков визуальных приложений.

Система Unity является бесплатным приложением для всех пользователей, которые используют её в некоммерческих целях (не ради финансовой выгоды, а с целью своего развития, творчества, совершенствования навыков программирования и работы с графикой и видео). Для остальных пользователей – индивидуальных предпринимателей и организаций, получающих финансовую прибыль от своей деятельности, – требуется покупка лицензии на использование системы Unity в коммерческих целях.

Поскольку мы с вами будем использовать Unity в образовательных целях, ничто не мешает нам установить данное приложение и бесплатно пользоваться им.

Первым делом нам потребуется скачать дистрибутив (установщик) Unity. Рекомендую делать это с официального сайта http://unity3d.com , поскольку на официальных сайтах выставляются самые последние версии программных продуктов (с исправленными ошибками и новыми возможностями).

Чтобы не тратить время на поиск раздела с установщиками, перейдите по ссылке:

https://unity3d.com/ru/get-unity/download/archive

Вы попадёте в раздел «Архив загрузок Unity» официального сайт.

Внизу вы увидите список релизов (выпусков) прошлых версий Unity, сгруппированных по версиям:

- Unity 2022.x
- Unity 2021.x
- Unity 2020.x
- Unity 2019.x
- Unity 2018.x
- Unity 2017.x
- Unity 5.x

Версия Unity 2018.3.13f – не самая последняя, однако для изучения основ это не столь принципиально.

Следует отметить, что, начиная с 2017 года (версии из группы «Unity 2017.x»), на указанном официальном сайте выставляются версии Unity, предназначенные для установки на 64-разрядные операционные системы Windows. Скорее всего, именно такая операционная система установлена и на вашем компьютере. Если же у вас установлена более ранняя 32-разрядная версия операционной системы Windows, то найти подходящий для неё установщик можно в группе «Unity 5.x» (не самая свежая версия, но является стабильной и вполне подходящей для работы).

Ещё одна причина, по которой может потребоваться установить более раннюю версию Unity – маломощный компьютер. Новые версии программных продуктов, как правило, требуют от компьютера больше ресурсов. Особенно это касается приложений для работы с графикой и видео, к которым относится и система Unity. Поэтому если ваш компьютер медленно работает с другими графическими приложениями (например, Adobe Photoshop, Corel DRAW, Sony Vegas, Pinnacle Studio, Autodesk 3ds Max и др.), возможно, имеет смысл сначала попробовать версии Unity из групп «Unity 2018.x» или «Unity 2017.x».

Итак, перейдём к скачиванию и установке Unity. Первым делом выберите версию Unity и в соответствующей ей строке нажмите на указывающую вниз стрелку одного из списков, начинающихся со слова «Downloads» («Загрузки»). Если вас на компьютере установлена операционная система Windows, раскройте список «Downloads (Win)», для MacOS загляните в список «Downloads (Mac)», а для Linux – в список «Downloads (Linux)». Далее обратите внимание на самые первые пункты раскрытого списка:

- «Unity Installer» («Установщик Unity»);
- «Unity Editor 64-bit» («64-разрядный редактор Unity»).

Все последующие инструкции и описания я буду приводить на примере операционной системы Windows. Пункт «Unity Installer» позволяет скачать веб-установщик – файл размером менее 1 мегабайта (1 Мб), который при запуске начнёт скачивать с Интернета и устанавливать на вашем компьютере программу Unity.

Пункт «64-разрядный редактор Unity» позволяет скачать напрямую сам дистрибутив Unity, размер которого составляет примерно 1 гигабайт (1 Гб).

Таким образом, веб-установщик занимает примерно в тысячу раз меньше места, чем скачиваемый через него классический установочный файл Unity. Каким вариантом воспользоваться – решать вам. Я предпочитаю не пользоваться веб-установщиком, а скачать исходный дистрибутив, поскольку в этом случае отсутствует риск, что во время установки сбой Интернет-связи приведёт к её некорректному завершению.

После того, как вы скачаете установочный exe-файл с сайта Unity, запустите его. Откроется окно установки Unity. Я не буду подробно описывать

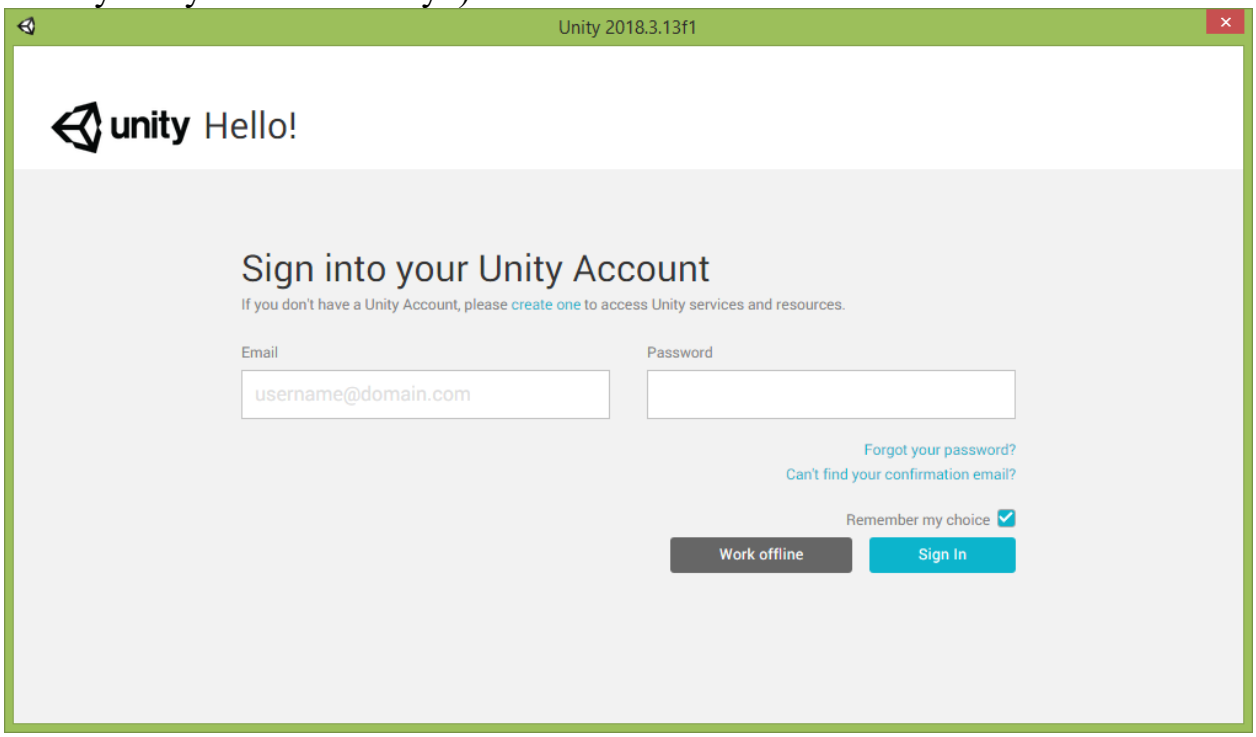
– они вполне стандартные и их состав может различаться в зависимости от той версии Unity, которую вы устанавливаете. Достаточно просто последовательно нажимать кнопки «Установить» и «Далее». Но всё же рекомендую попутно обращать внимание на то, что написано на каждом шаге процесса установки, чтобы знать, с какими настройками вы устанавливаете приложение (например, в какую папку оно будет установлено).

1.2. Регистрация учётной записи

После того, как процесс установки завершится, закройте последнее окно установки и запустите программу Unity, используя созданный на рабочем столе ярлык «Unity» или найдя её в списке программ в меню «Пуск».

Замечание. Иногда бывает, что при первом запуске появляется серое окно с сообщением на английском языке о том, что на компьютере нет файла лицензии Unity. В этом случае нажмите кнопку «Force Quit» («Принудительный выход») и попробуйте повторно запустить Unity – должно сработать.

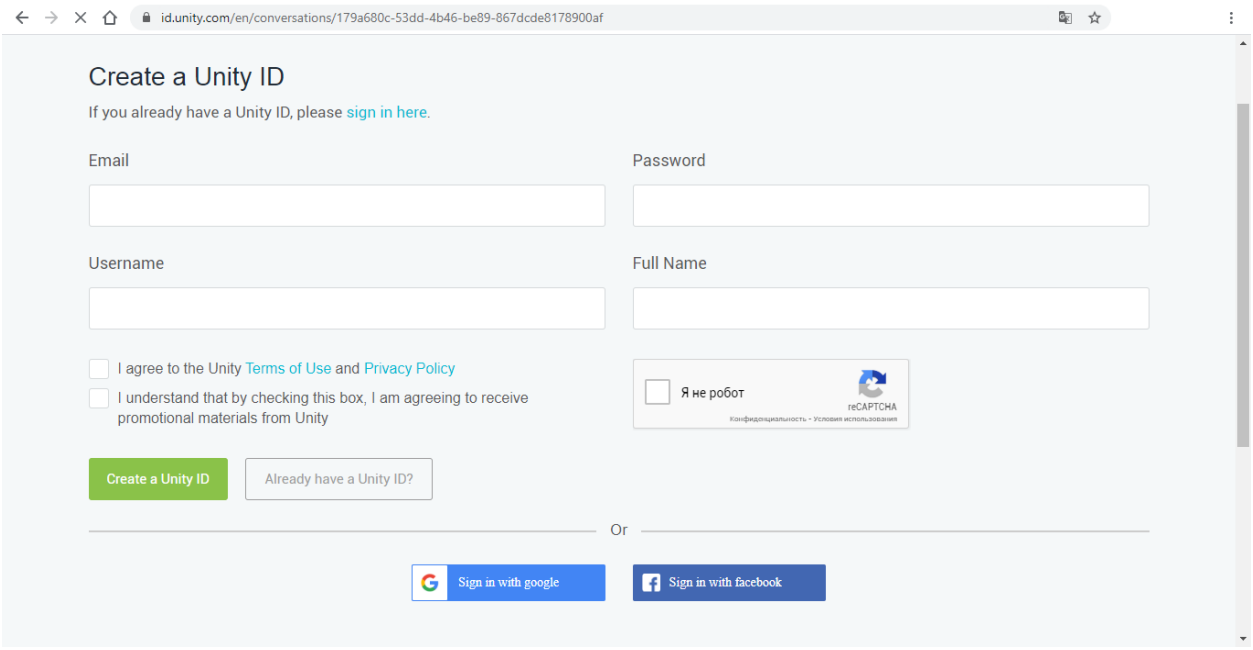
На экране появится окно «Sign into your Unity Account» («Войдите в свою учётную запись Unity»).



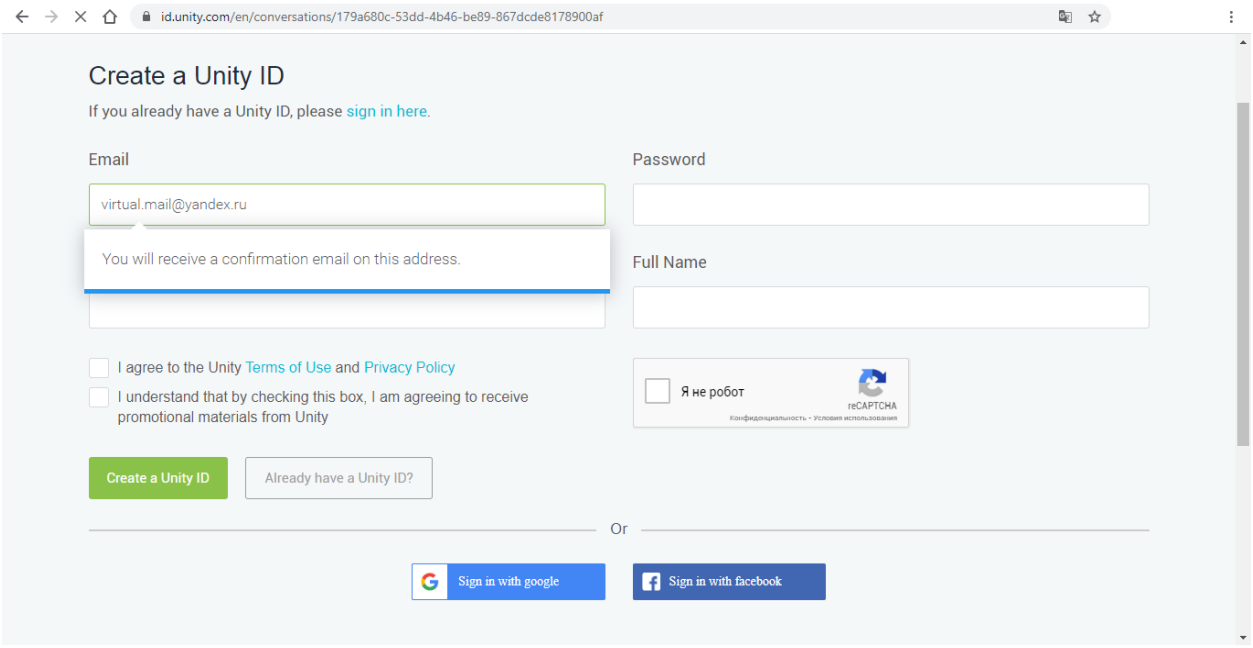
Если ранее вы уже были зарегистрированы на сайте Unity, то введите указанные при регистрации адрес электронной почты и пароль доступа к учётной записи Unity, а затем нажмите кнопку «Sign in» («Войти»).

Если же вы ещё не регистрировались, то первым делом следует сделать именно это, поскольку без учётной записи не получится создать лицензионный файл для работы в приложении. Для начала регистрации нажмите выделенную цветом надпись «create one» («создать новую») или откройте браузер и перейдите по адресу: https://id.unity.com/account/new .

Откроется страница сайта Unity с формой «Create a Unity ID» («Создание учётной записи Unity»).



В поле «Email» («Электронная почта») следует указать адрес вашей электронной почты. Всплывающая надпись «**You will receive a confirmation email on this address**» поясняет, что «На этот адрес Вы получите письмо для подтверждения регистрации».

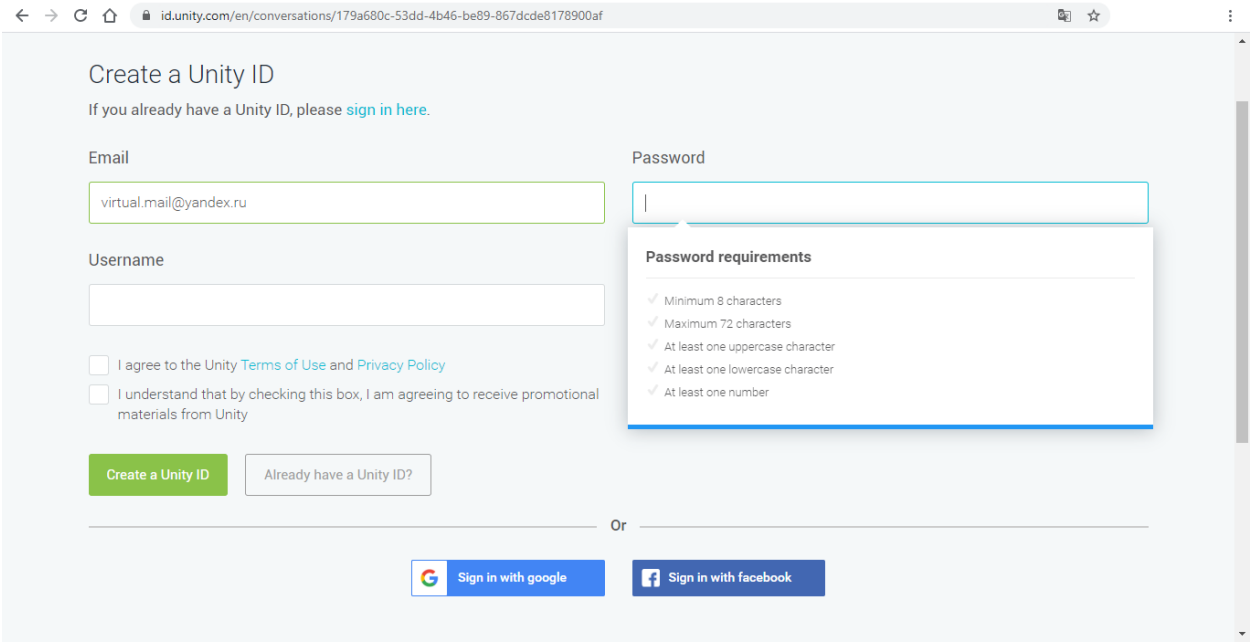


Если указанный адрес электронной почты введён без ошибок и ранее не использовался для регистрации на сайте Unity, то у поля ввода появится зелёная рамка, в противном случае рамка станет красной и всплывающая надпись отобразит красный текст с пояснением об ошибке.

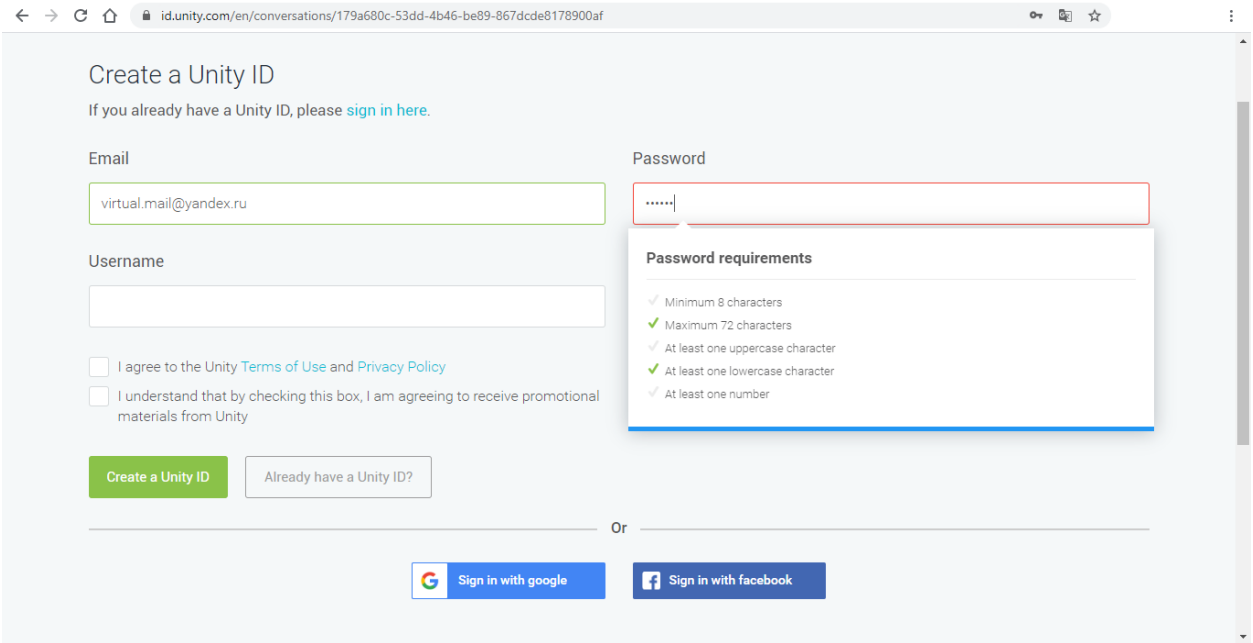
Важное замечание. Если у вас ещё нет электронной почты, завести её не составит особого труда – это можно сделать быстро и бесплатно на сайтах таких крупных поисковых порталов, как Google, Yandex, Rambler, Mail.ru и многих других.

Далее следует придумать пароль для своей будущей учётной записи и ввести его в поле «Password» («Пароль»). Всплывающая надпись «Password requirements» («Требования к паролю») подскажет вам, каким должен быть ваш пароль:

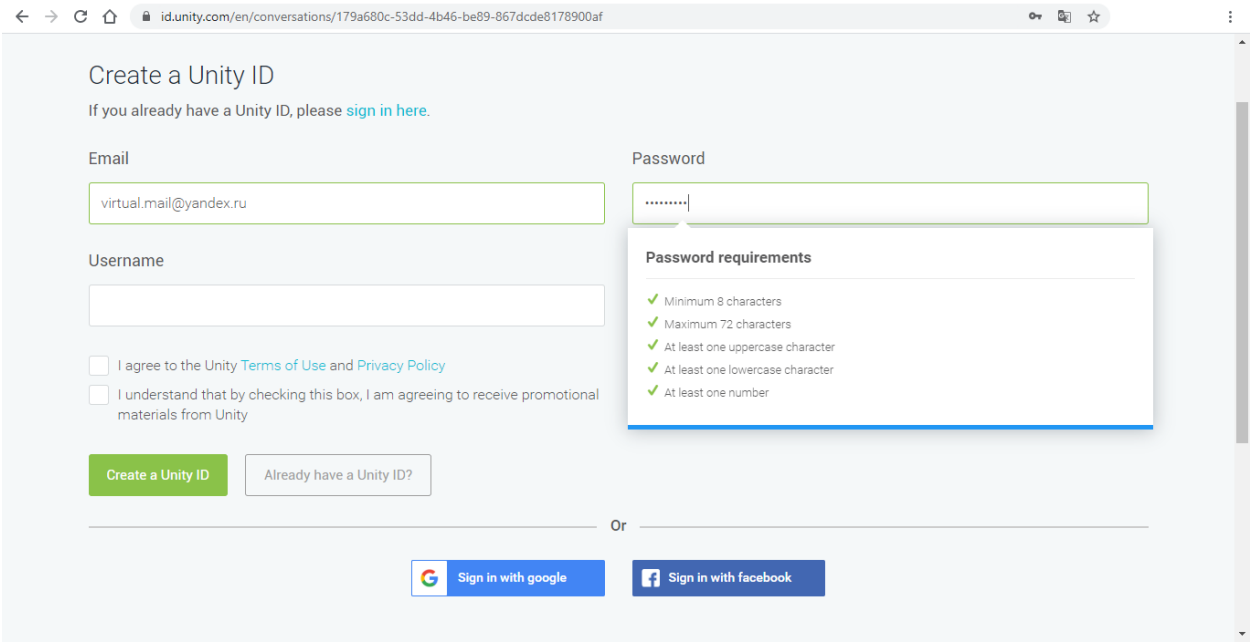
- Minimum 8 characters («Минимальная длина – 8 символов»);
- Maximum 72 characters («Максимальная длина – 72 символа»);
- At least one uppercase character («По крайней мере одна заглавная буква»);
- At least one lowercase character («По крайней мере одна строчная буква »);
- At least one number («По крайней мере одна цифра»).



При наборе пароля выполненные условия будут помечаться галочками.

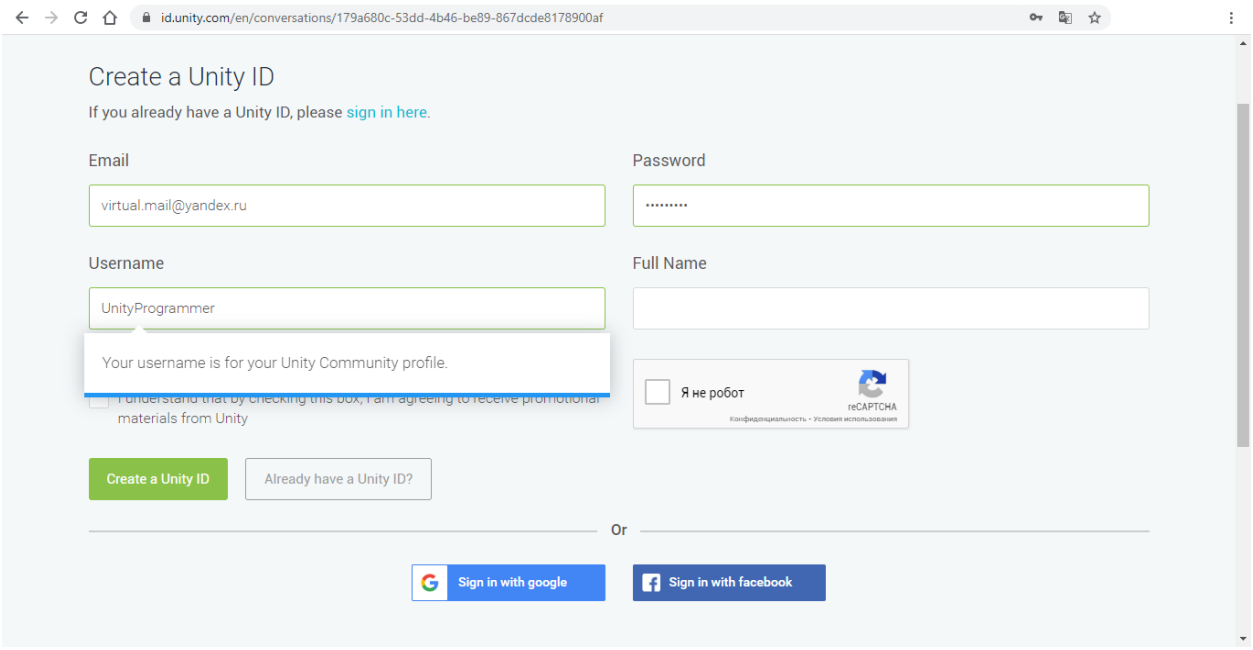


Как только будет набран пароль, соответствующий всем указанным требованиям, галочки будут проставлены напротив всех пунктов, а у поля ввода появится зелёная рамка.



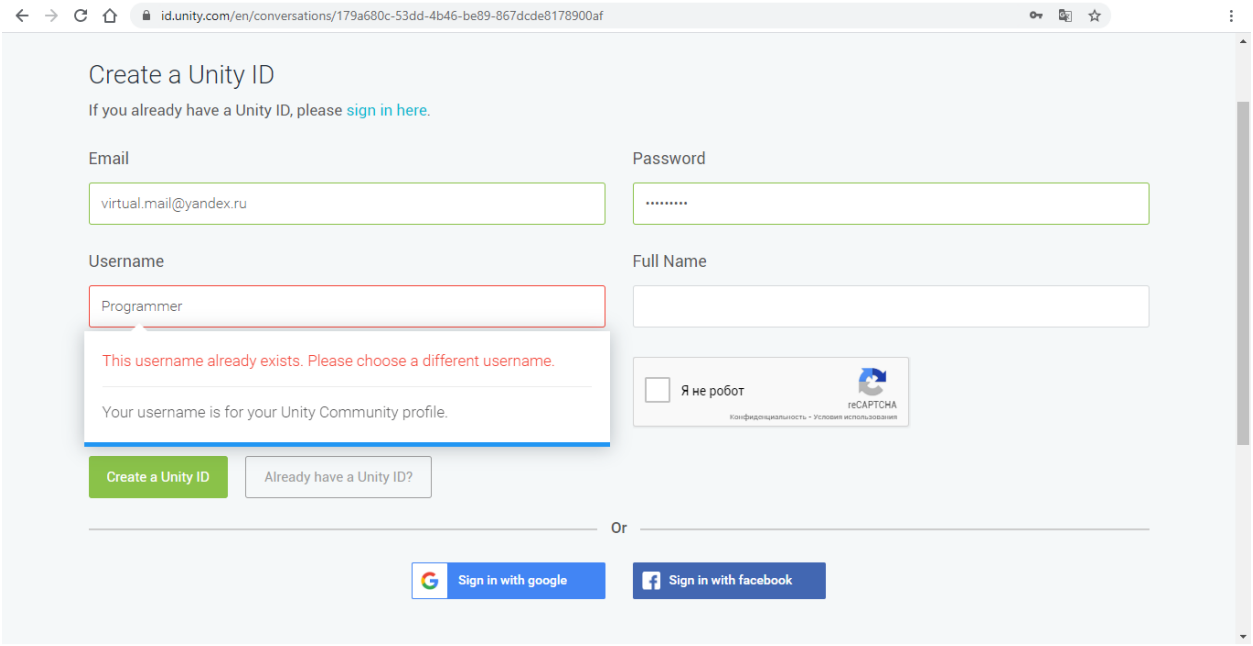
Важное замечание. Никогда не указывайте при регистрации на сайтах пароль от вашей электронной почты – это снизит риск её взлома. Возьмите за правило всегда придумывать для разных сайтов разные пароли, чтобы злоумышленники, узнав пароль от одной учётной записи, не смогли использовать его для доступа к вашим учётным записям, созданным на других сайтах.

Третье поле, которое потребуется заполнить в форме регистрации, называется «Username» («Имя пользователя»). Всплывающая надпись «**Your username is for your Unity Community profile**» поясняет, что в поле требуется ввести «Имя пользователя для вашего профиля в сообществе разработчиков Unity».



При задании имени пользователя следует использовать только английские буквы, цифры и знаки подчёркивания. При этом первым символом должна быть буква и в имени не должно быть пробелов.

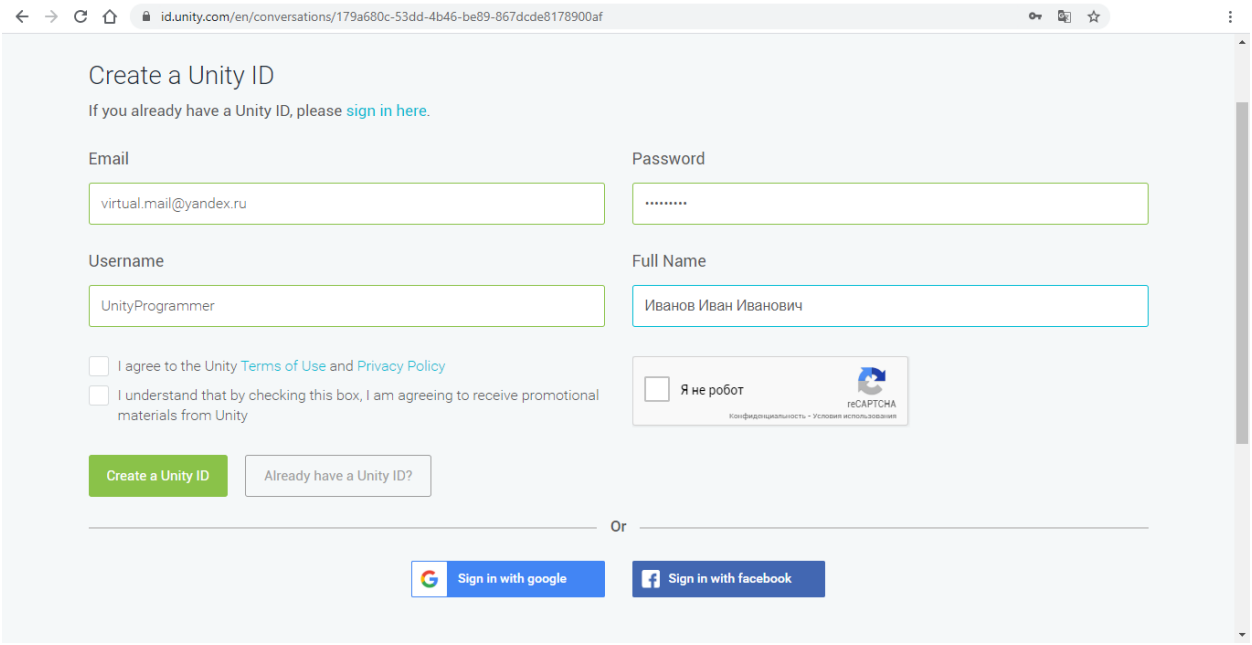
Если введённое вами имя пользователя уже кем-то занято, рамка поля ввода станет красной и всплывающая надпись отобразит красный текст «**This username already exists. Please choose different username**» («Это имя пользователя уже занято. Пожалуйста, выберите другое имя пользователя»).



Как только вы введёте незанятое и корректное имя пользователя, у поля ввода появится зелёная рамка.

Последнее поле, которое следует заполнить – «Full Name» («Полное имя»). Чтобы страница регистрации приняла ваше полное имя, рекомендуется заполнить его в формате:

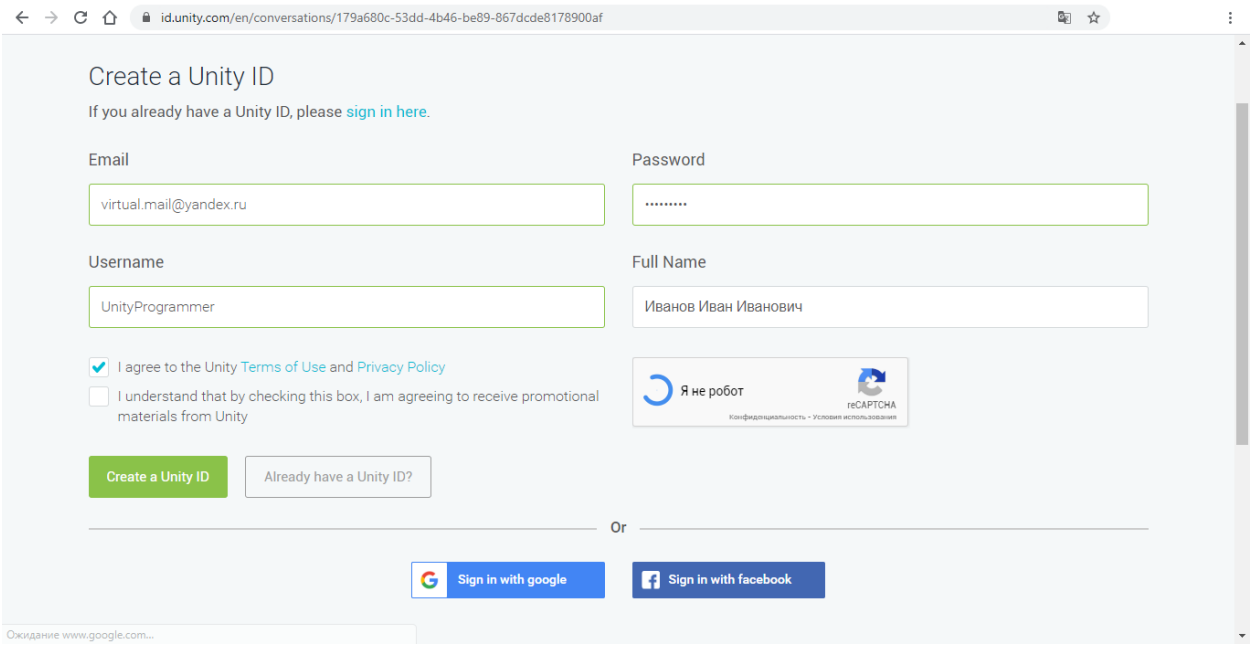
Фамилия Имя Отчество



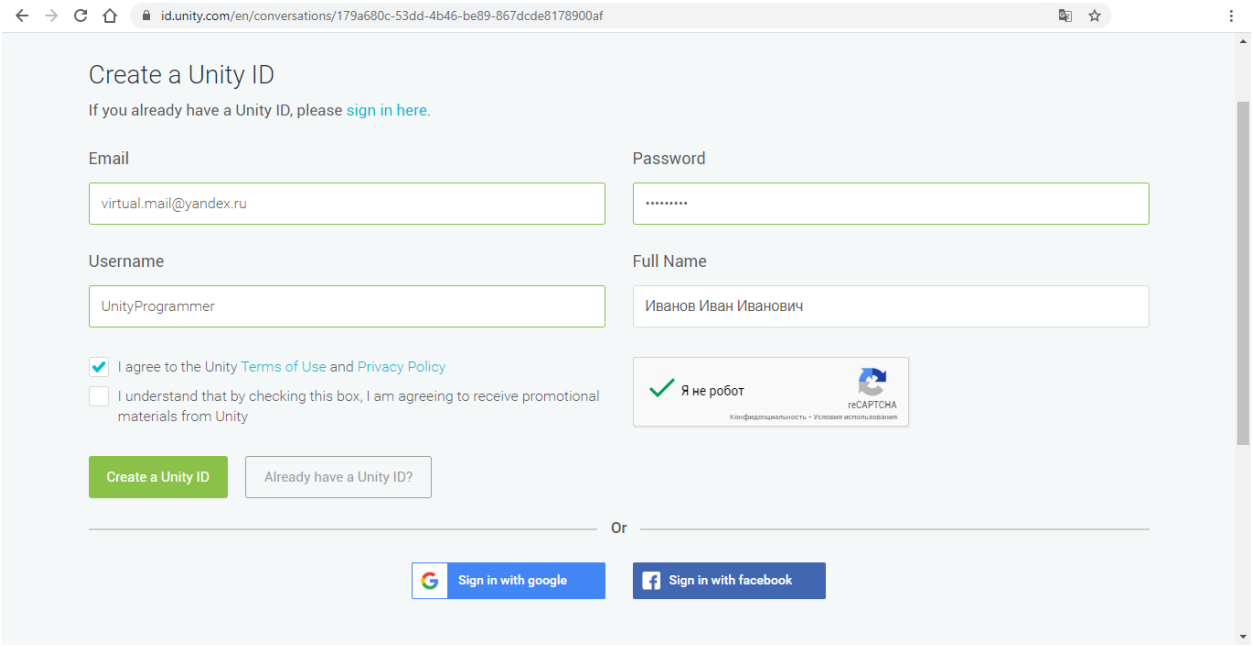
Теперь установите галочку слева от надписи «**I agree to the Unity Terms of Use and Privacy Policy**» («Я согласен с Условиями использования Unity и Политикой конфиденциальности») – это является обязательным условием регистрации.

Галочку слева от надписи «**I understand that by checking this box, I am agreeing to receive promotional materials from Unity**» («Я понимаю, что, отмечая этот пункт, я соглашаюсь получать рекламные материалы от Unity») устанавливать необязательно.

И, наконец, последнюю галочку следует установить слева от надписи «**I'm not a robot**» («Я не робот»). После этого запустится процесс проверки и, возможно, всплывёт несложная задача (например, выбрать квадратные фрагменты с изображением автомобилей), пройдя которую, вы докажете, что регистрируетесь вручную (как человек, а не автоматическая программа).



После прохождения проверки слева от надписи появится галочка.

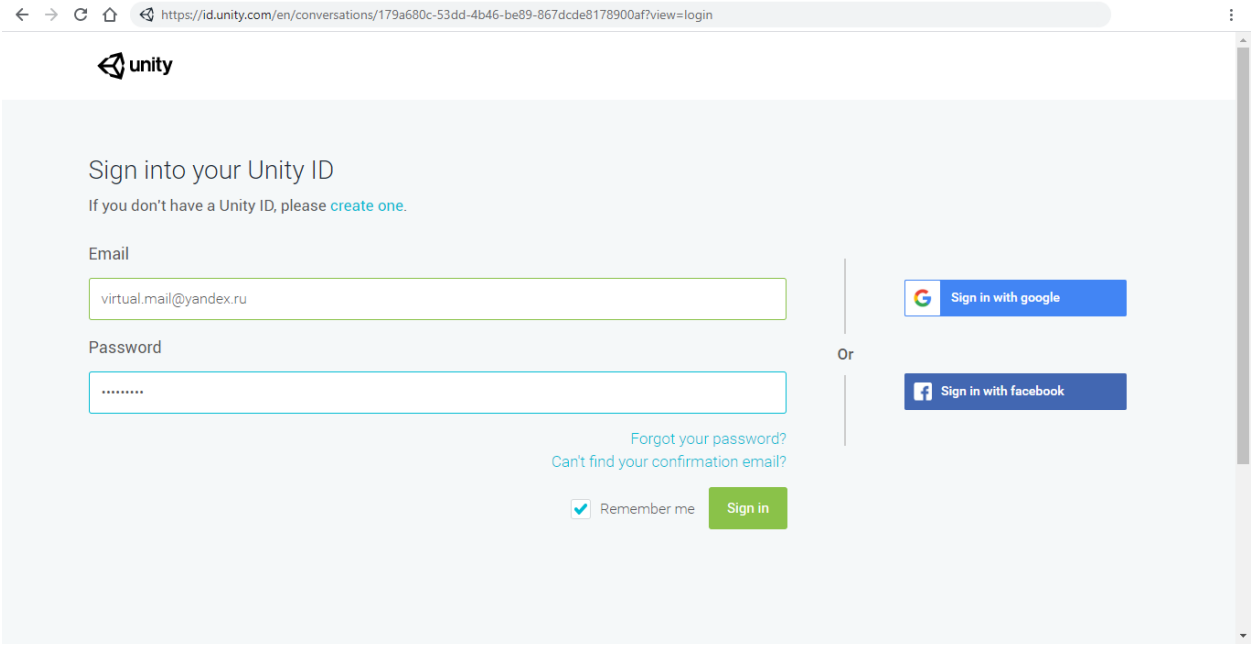


Теперь осталось нажать кнопку «Create a Unity ID» («Создать учётную запись Unity»).

После этого откройте свою электронную почту, на адрес которой вы регистрировали учётную запись Unity. На эту почту с адреса accounts@unity3d.com должно прийти письмо «Confirm your email address». В этом письме следует щёлкнуть по красной гиперссылке «Link to confirm email», чтобы завершить регистрацию.

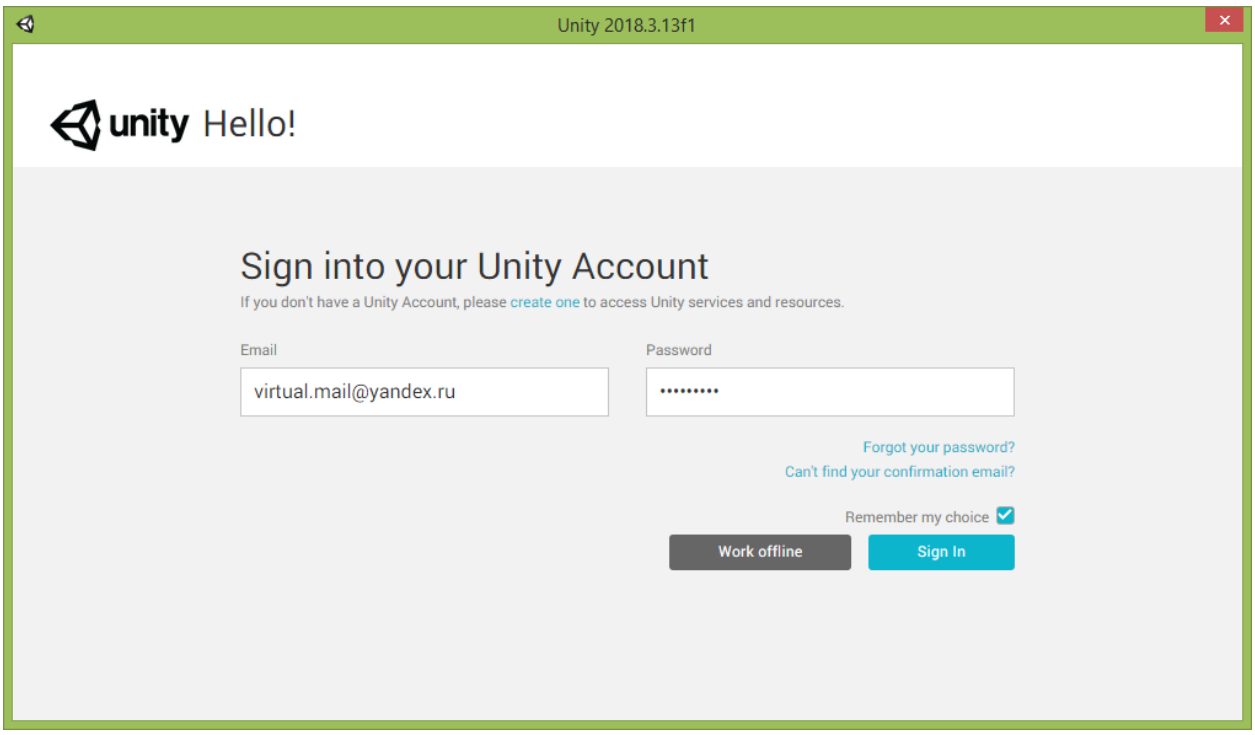
Теперь перейдите на сайт Unity по ссылке https://id.unity.com и введите в поля «Email» и «Password» адрес электронной почты и пароль, которые вы указали при регистрации. Если не хотите каждый раз при входе на сайт

повторять ввод своих учётных данных, поставьте галочку рядом с пунктом «Remember me» («Помнить меня»). Теперь нажмите кнопку «Sign in» («Войти»).



После этого вы увидите профиль своей учётной записи, в котором сможете поменять настройки и заполнить другие данные о себе (страну проживания, часовой пояс и т.д.).

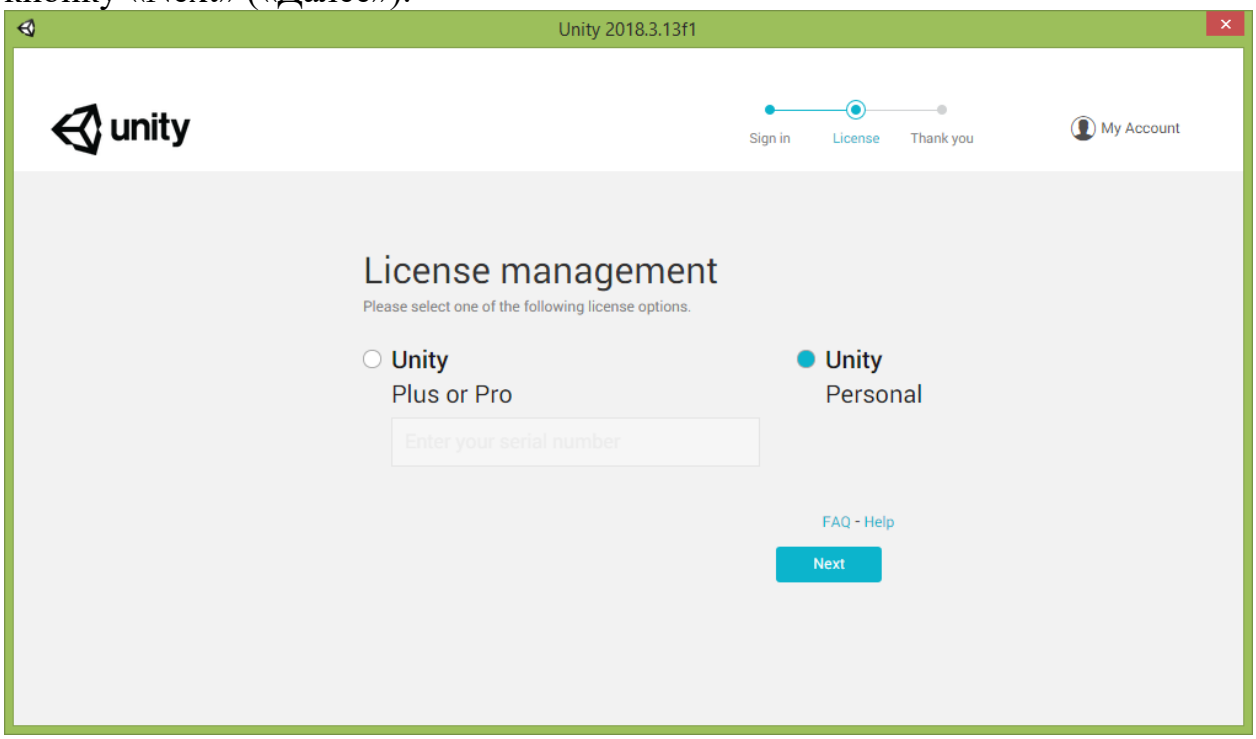
На самом деле, чтобы войти в программу Unity, не обязательно открывать указанную страницу сайта. Достаточно просто запустить программу Unity, и в окне «Sign into your Unity Account» («Войдите в свою учётную запись Unity») ввести ваши учётные данные в поля «Email» и «Password».



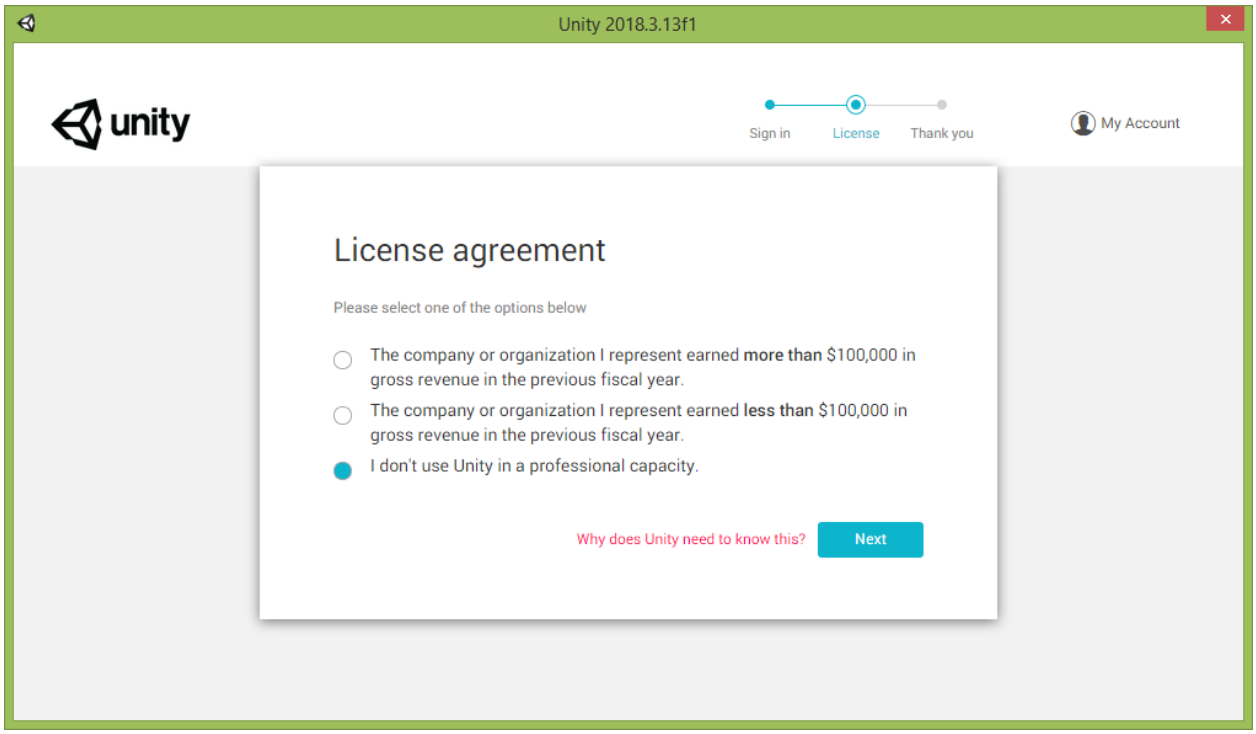
Чтобы не вводить их каждый раз при входе в программу, поставьте галочку рядом с пунктом «Remember my choice» («Запомнить мой выбор»). Далее остаётся только нажать кнопку «Sign In» («Войти»).

При первом входе в программу потребуется получить файл лицензии для работы на текущем компьютере (для этого нам и нужно было предварительно зарегистрироваться).

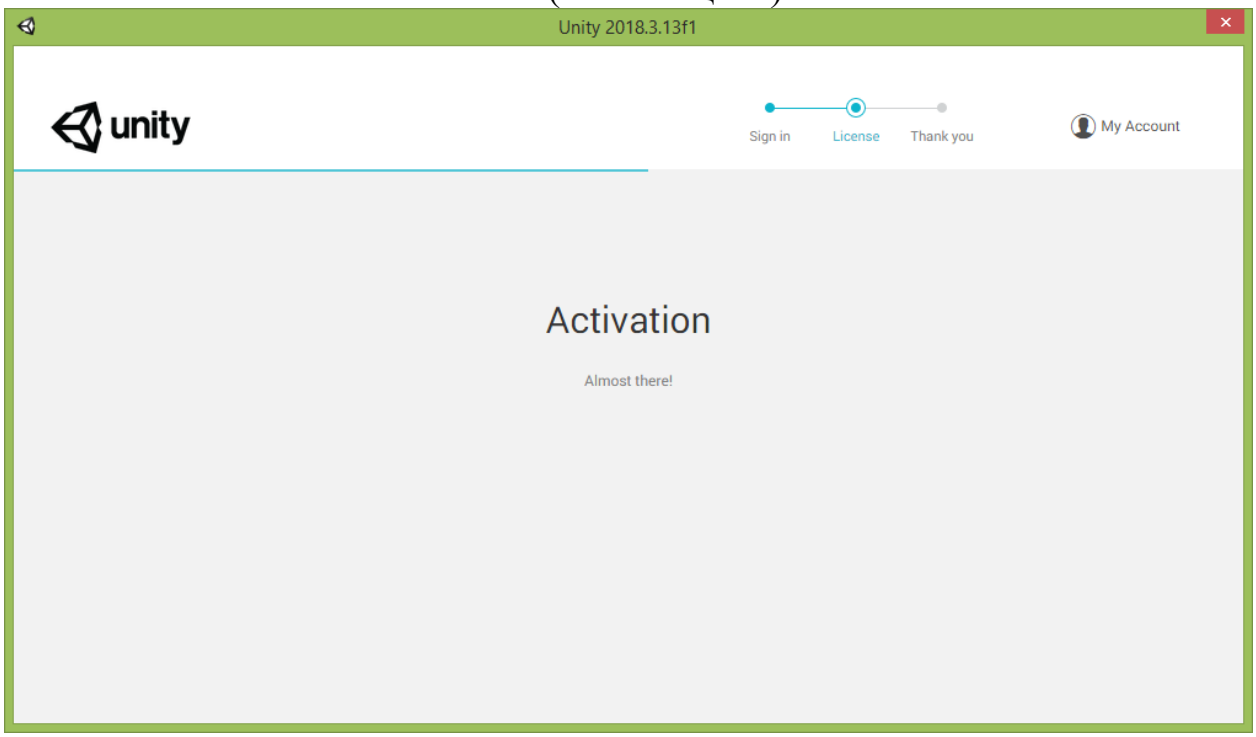
В появившемся окне «License management» («Управление лицензией») выберите пункт «Unity Personal» («Unity для личных целей») и нажмите кнопку «Next» («Далее»).



В следующем окне «License agreement» («Лицензионное соглашение») выберите пункт «I don’t use Unity in a professional capacity» («Я не использую Unity в профессиональных целях») и нажмите кнопку «Next» («Далее»).

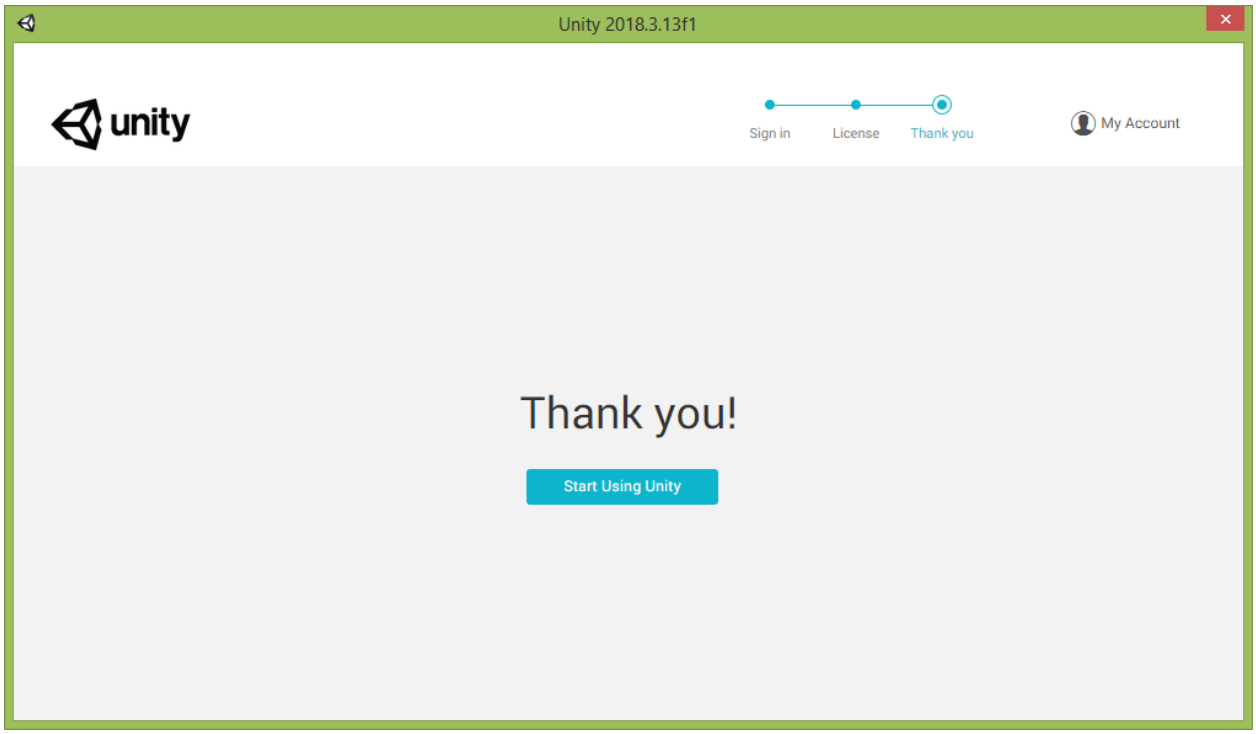


Появится окно «Activation» («Активация»).



Теперь просто ожидайте завершения активации. В ходе этого процесса через созданную вами учётную запись на компьютер устанавливается лицензионный файл Unity. Поэтому доступ к сети Интернет должен быть по-прежнему включен.

Если всё прошло хорошо, вы увидите окно с благодарностью за установку Unity. Если же что-то пошло не так, то следует вернуться назад и повторить предыдущий шаг. Обычно проблема возникает, если при активации был отключен доступ в Интернет.



После завершения активации нажмите кнопку «Start Using Unity» («Начать использование Unity»).

Задание для самостоятельной работы

1. Определите подходящую для вашего компьютера версию Unity, перебрав варианты из списка «Version» (он расположен в левом верхнем углу страницы) и познакомившись с системными требованиями каждой версии, перейдя в ветку «Working in Unity → Installing Unity → System requirements».

2. Скачайте автономный или веб-установщик выбранной вами версии и установите Unity с его помощью.

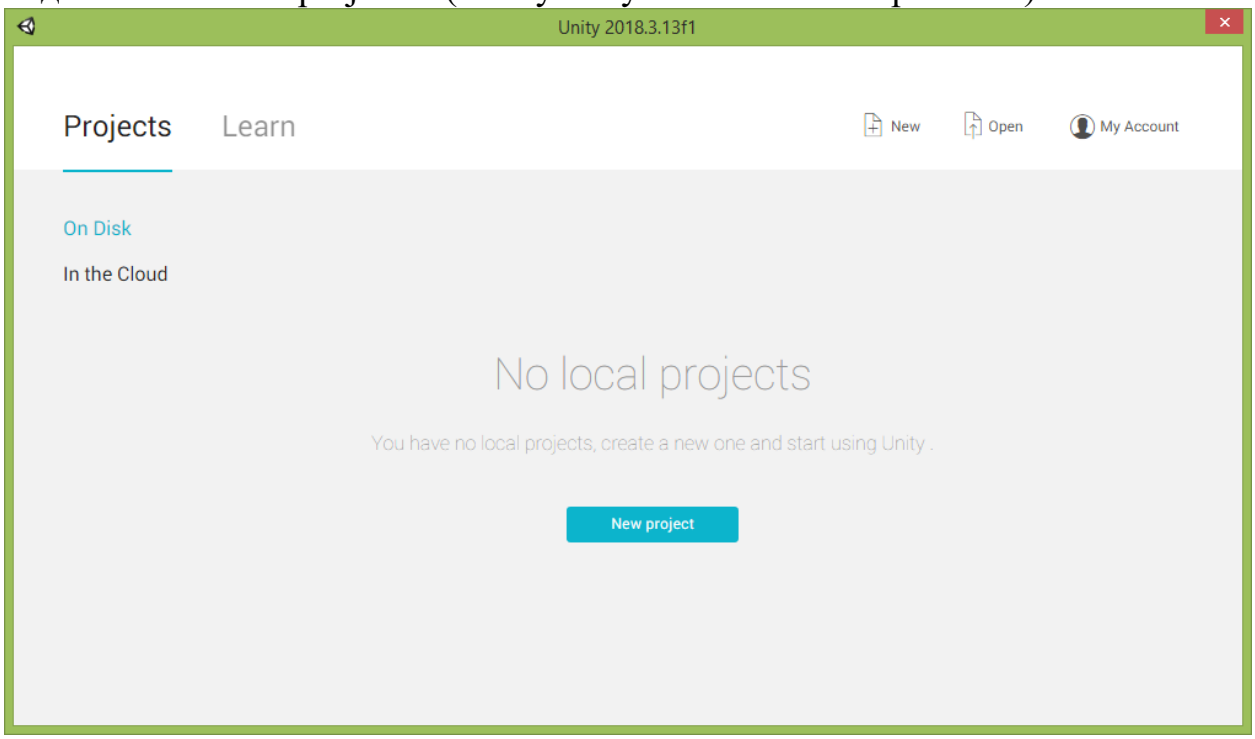
3. Пройдите процедуру регистрации учётной записи Unity (если ранее ещё этого не сделали) и зайдите под ней в приложение.

Если у вас возникнут трудности в процессе выполнения задания, обратитесь к преподавателю и обсудите с ним данную проблему.

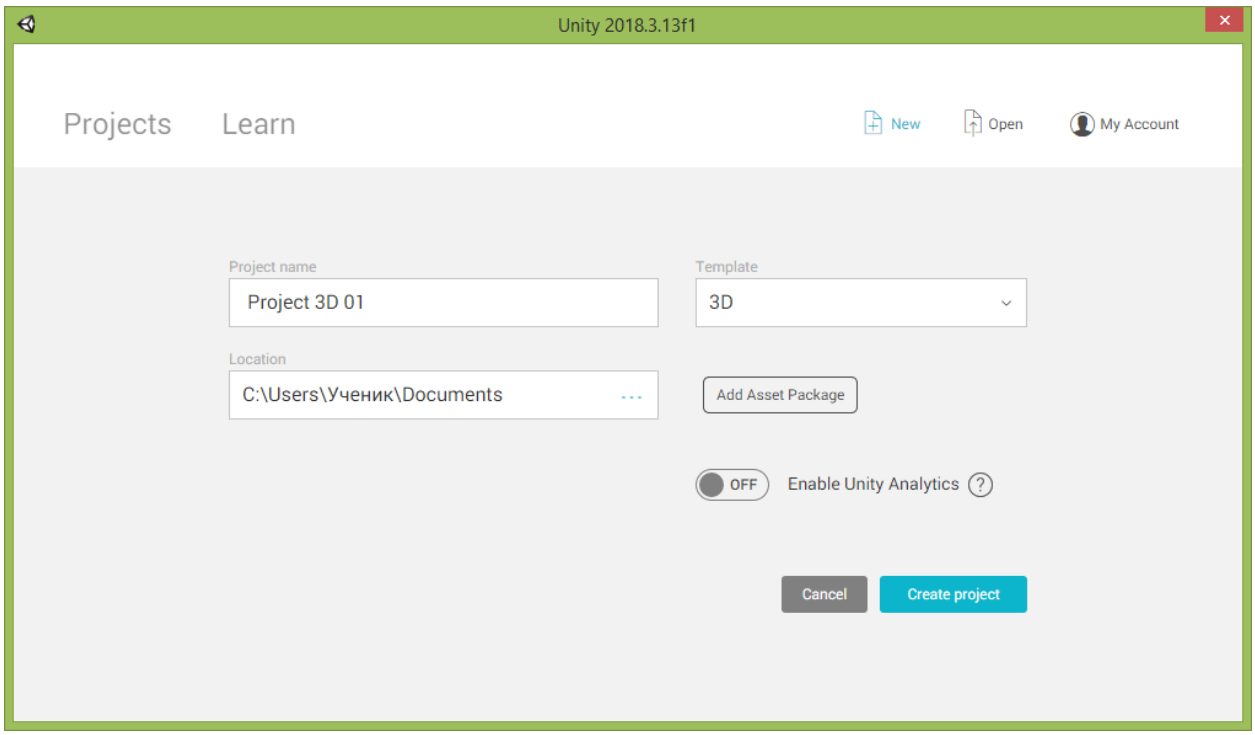
Глава 2. Создание трёхмерной сцены

2.1. Создание первого 3D-проекта

После того, как вы создали учётную запись Unity, получили файл лицензии и запустили программу Unity, появится окно со списком проектов, созданных на текущем компьютере. Изначально этот список пуст и содержит надпись «No local projects» («Отсутствуют локальные проекты»).



Нажмите кнопку «New project» («Новый проект»).
Появится форма создания проекта.



В поле «Project name» («Название проекта») задайте имя своему новому проекту. Я свой первый 3D-проект назвал «Project 3D 01».

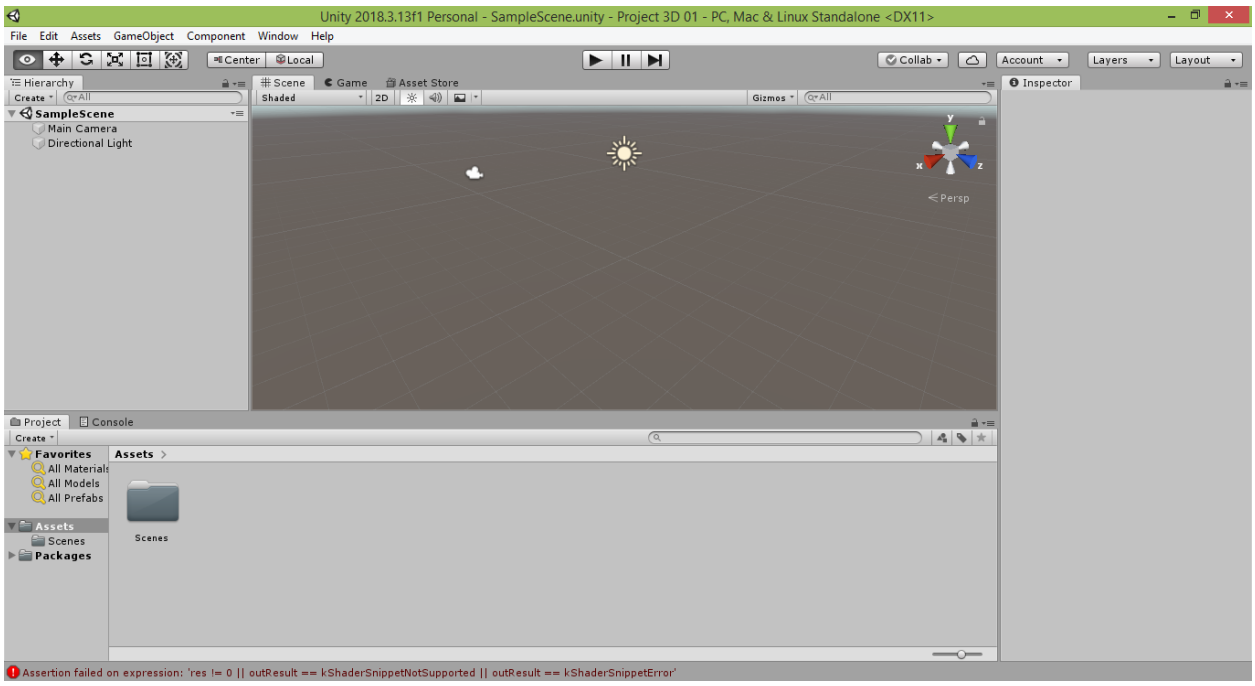
Не забудьте при этом в списке «Template» («Шаблон») выбрать пункт «3D», поскольку мы будем работать над созданием трёхмерной сцены.

В поле «Location» («Местоположение») указан адрес папки, в которой будут размещены файлы созданного вами проекта. Здесь я оставил папку по умолчанию – «Documents» («Документы»). Но рекомендую создать специальную папку для хранения ваших проектов, а затем указать путь к нему. Например, это может быть папка «Unity Projects», созданная в той же папке «Документы». Это позволит вам быстрее находить каталоги с вашими проектами – все они будут лежать отдельно в этой специальной папке, а не вперемешку с остальными вашими файлами и папками.

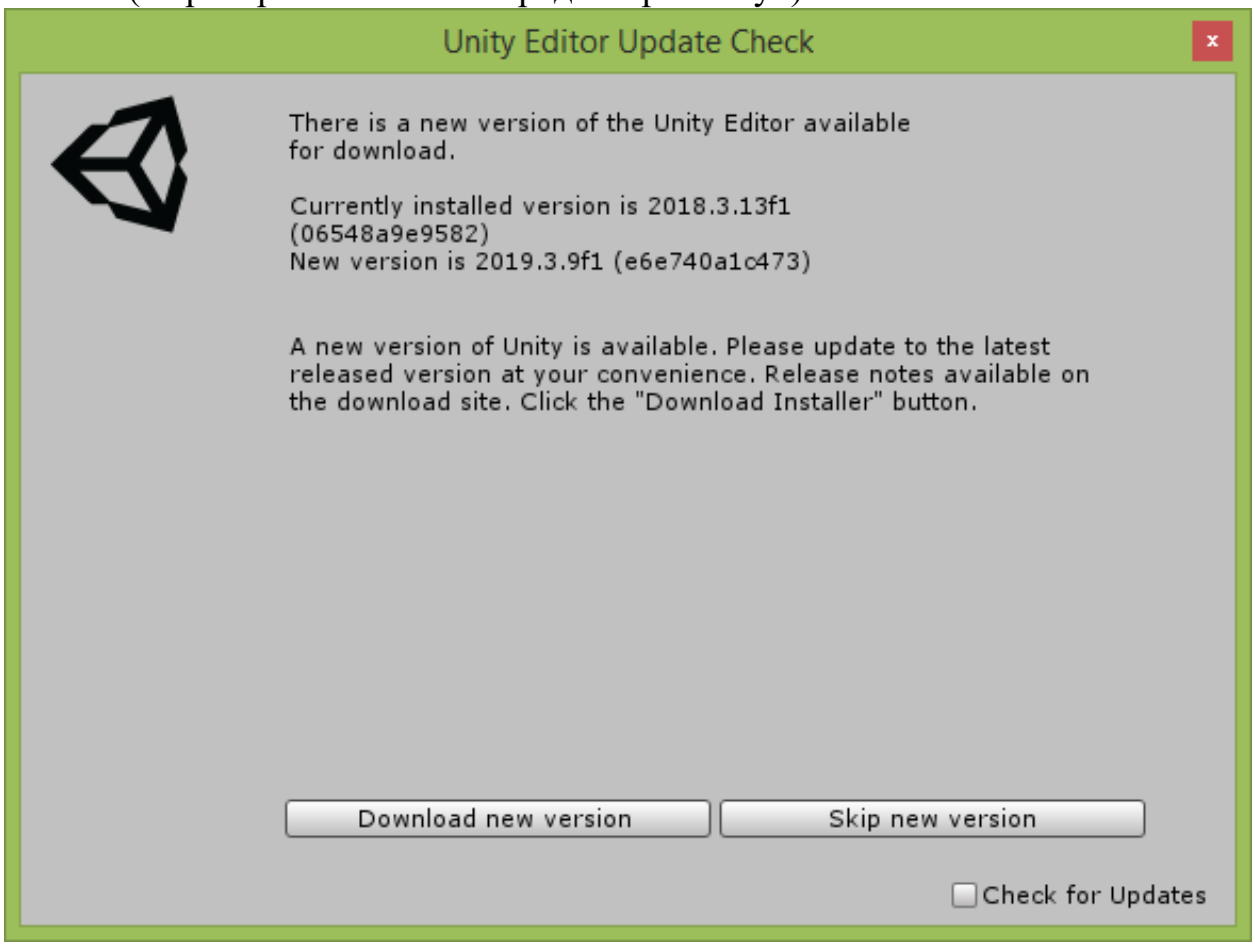
Далее обратите внимание на переключатель-движок «Enable Unity Analytics» («Задействовать аналитику Unity»). Рекомендую перевести его в положение «OFF» («Выключено»). В этом случае Unity не будет отправлять информацию о ваших действиях на сайт Unity, что несколько снизит нагрузку на вашу Интернет-сеть.

Теперь можно нажать кнопку «Create Project» («Создать проект»). Далее ожидайте, пока Unity создаст все файлы нового проекта, и запустит окно редактора. Этот процесс может длиться около минуты (за его состоянием позволяет следить полоса загрузки).

После завершения процесса создания проекта появится окно с пустой сценой.



Далее поверх этого окна может появиться окно «Unity Editor Update Check» («Проверка обновлений редактора Unity»).

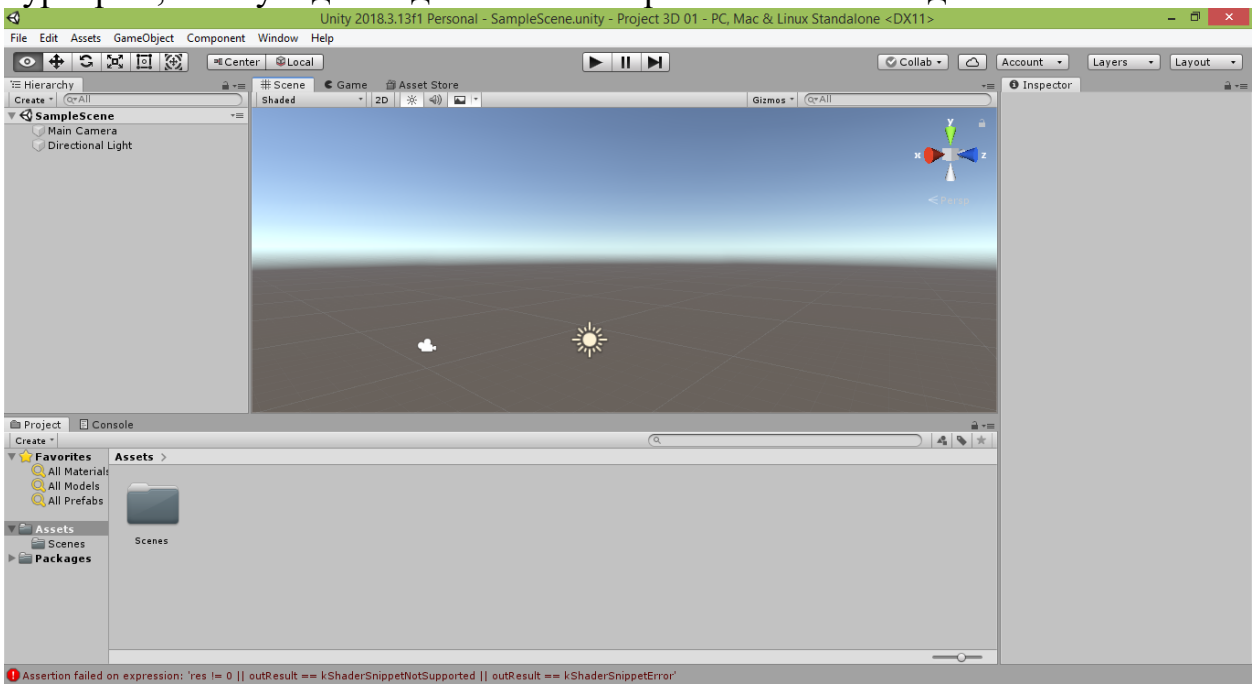


В этом окне содержится уведомление, что доступна более новая версия Unity, и предложение установить её. Вы уже знаете, как своими силами установить Unity, поэтому нажмите кнопку «Skip new version» («Пропустить

установку новой версии»). Если вы не хотите, чтобы это окно появлялось в дальнейшем, тогда перед нажатием этой кнопки снимите галочку «Check for Updates» («Проверять наличие обновлений»).

Теперь мы готовы приступить к работе со сценой. Сцена – это виртуальное пространство, на котором вы можете размещать любые объекты – геометрические тела, камеры и источники света. Также сцену увидит и пользователь, когда запустит разработанное вами приложение.

Для начала попробуем осмотреться. Для этого нажмите на пространстве сцены правой кнопкой мыши и, удерживая её нажатой, переместите курсор вверх-вниз и вправо-влево. Область обзора будет следовать за вашим курсором, и вы увидите вдали линию горизонта и небо над ней.



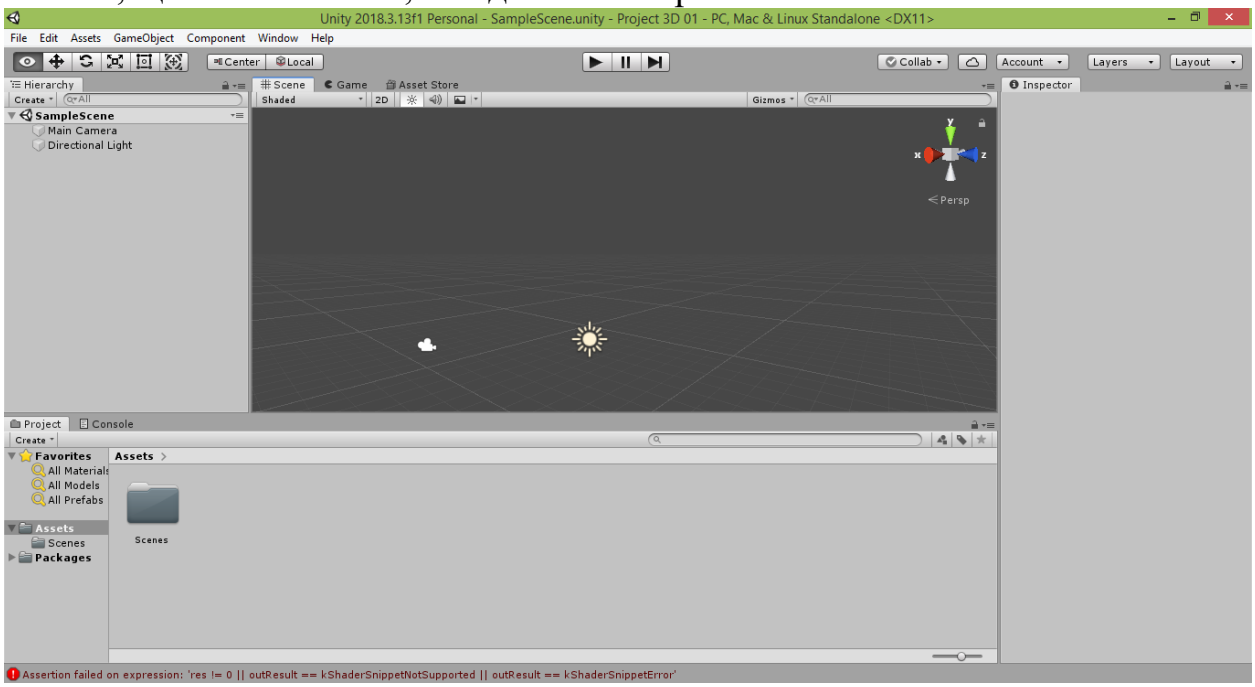
Также вы можете нажать на пространстве сцены левой кнопкой мыши и, удерживая её нажатой, переместить курсор вверх-вниз и вправо-влево. Перед этим убедитесь, что в левом верхнем углу окна редактора Unity нажата кнопка с изображением руки  (она находится прямо под пунктом меню «File» и при изменении области обзора сцены заменяется кнопкой с изображением глаза ). В результате при движении курсора вверх-вниз и вправо-влево за ним будет следовать и точка, из которой вы наблюдаете сцену.

Если вам потребуется перемещать точку обзора вправо-влево и вперёд-назад, воспользуйтесь клавишами со стрелками на своей клавиатуре.

Для того, чтобы пройтись по сцене по сложной траектории (например, подойти к объектам и осмотреть их с разных сторон), нажимайте так называемые игровые клавиши «W», «A», «S» и «D» при зажатой правой кнопке мыши. В этом случае вы сможете перемещаться боком влево и вправо (клавиши «A» и «D»), двигаться вперёд (клавиша «W») или пятиться назад (клавиша «S»), а движения мыши позволят управлять направлением взгляда,

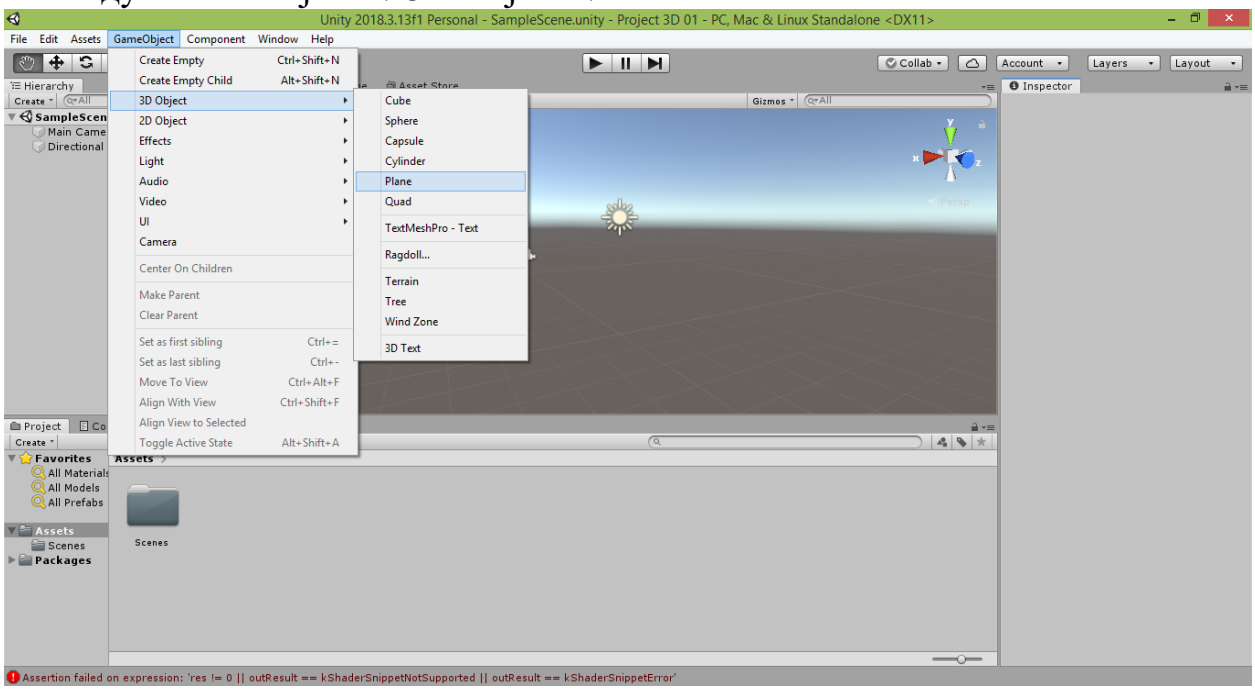
относительно линии которого и будет происходить движение вперёд-назад и вправо-влево.

В процессе движения одним из немногих ваших ориентиров будет являться линия горизонта и небо над ней. Однако возможны ситуации, когда панорама неба не будет видна (такое иногда случается при создании новых проектов). В этом случае проверьте, нажата ли небольшая кнопка с изображением пейзажа  в строке над рабочим полем сцены. Если она не нажата, щёлкните по ней, и над линией горизонта появится небо.

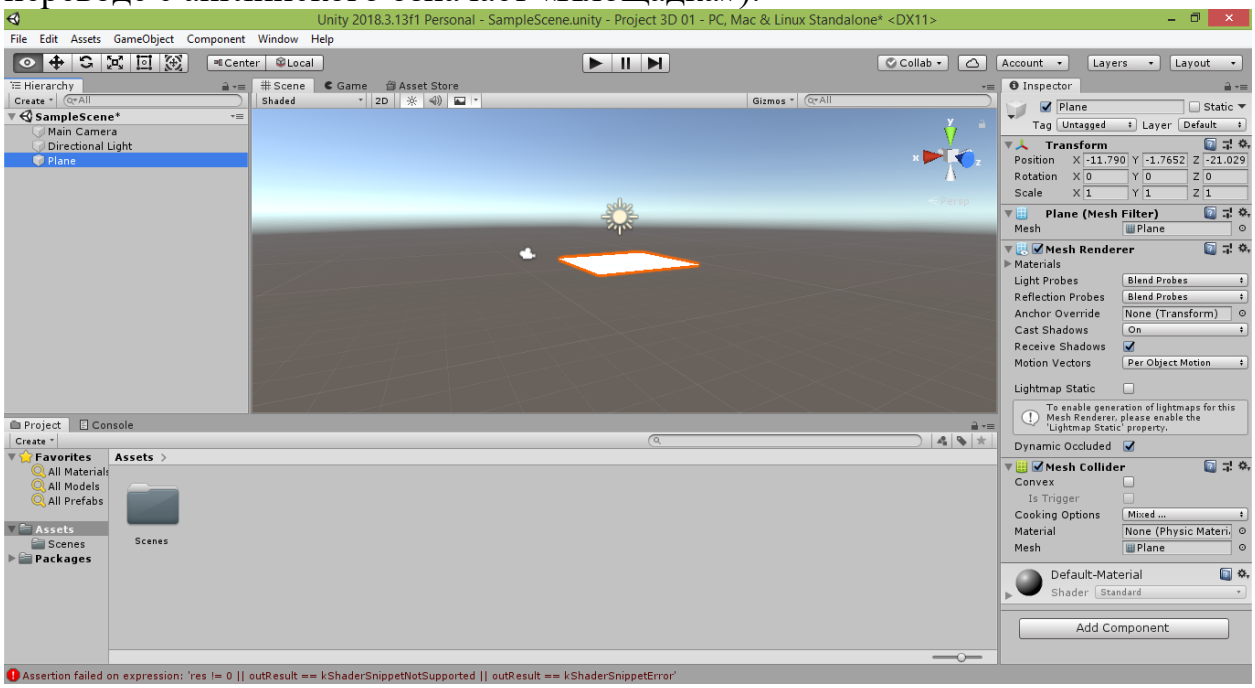


2.2. Инструменты трансформации объектов сцены

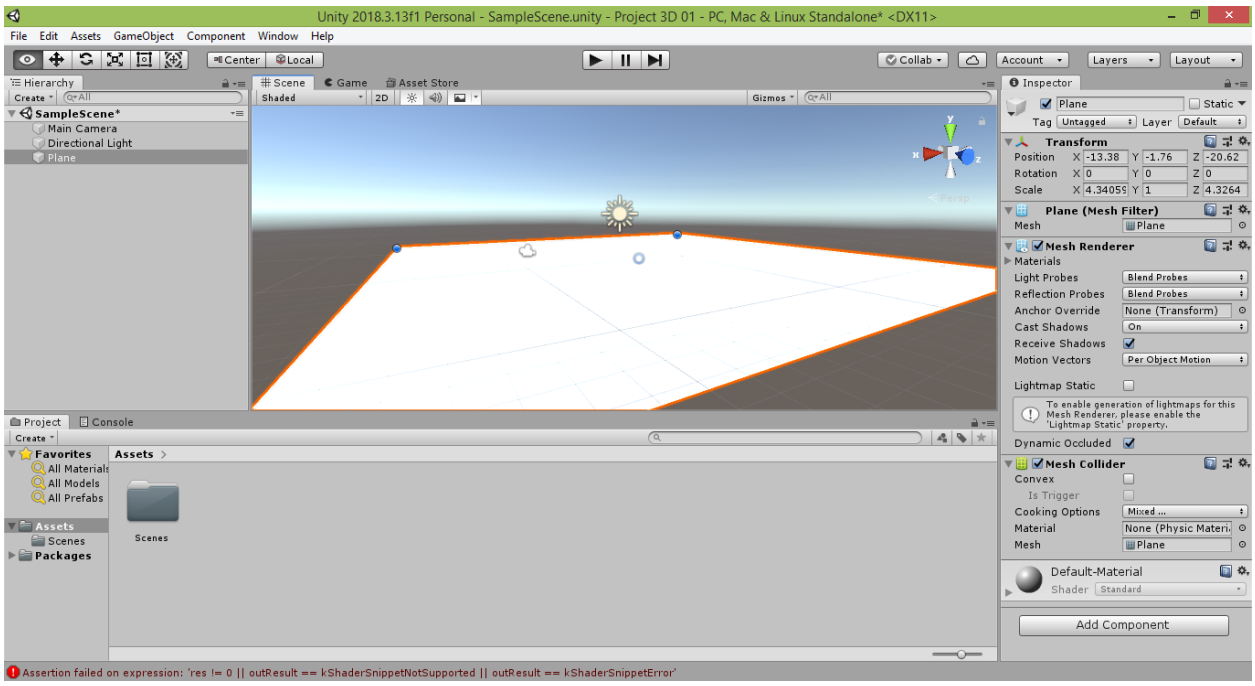
Теперь расставим объекты на нашей сцене. В меню Unity выберите команду «GameObject → 3D Object → Plane».



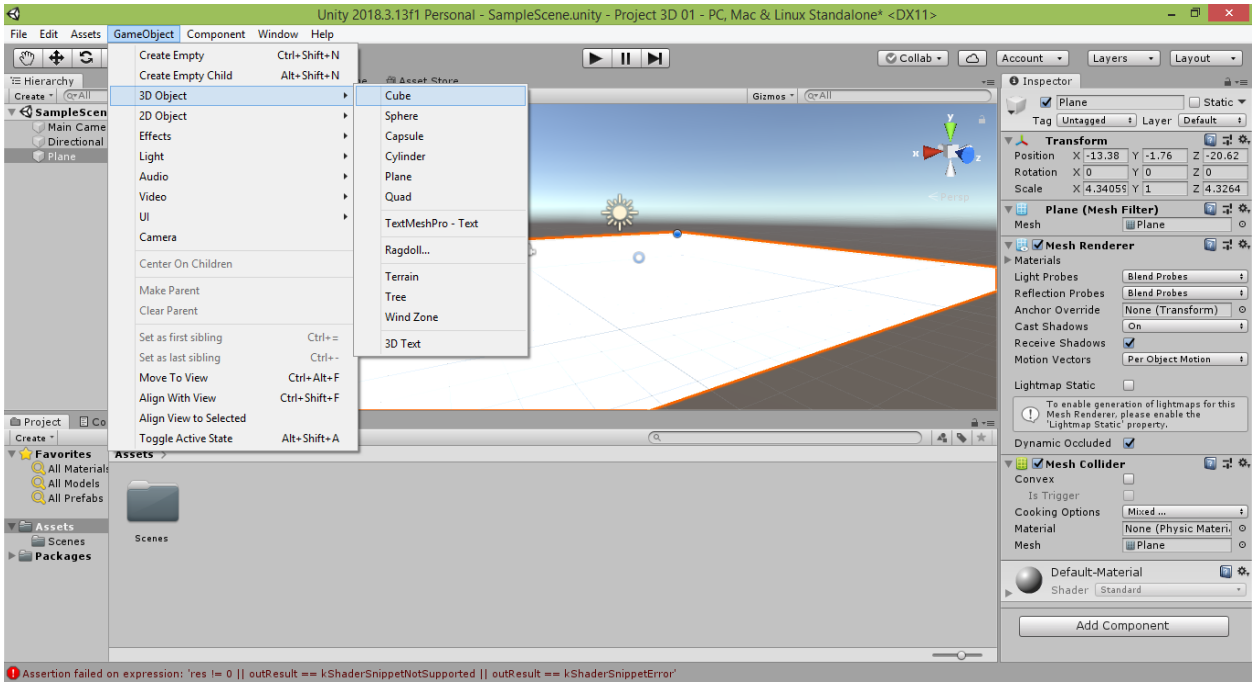
В результате на вашей сцене появится игровой 3D-объект «Plane» (в переводе с английского означает «Площадка»).



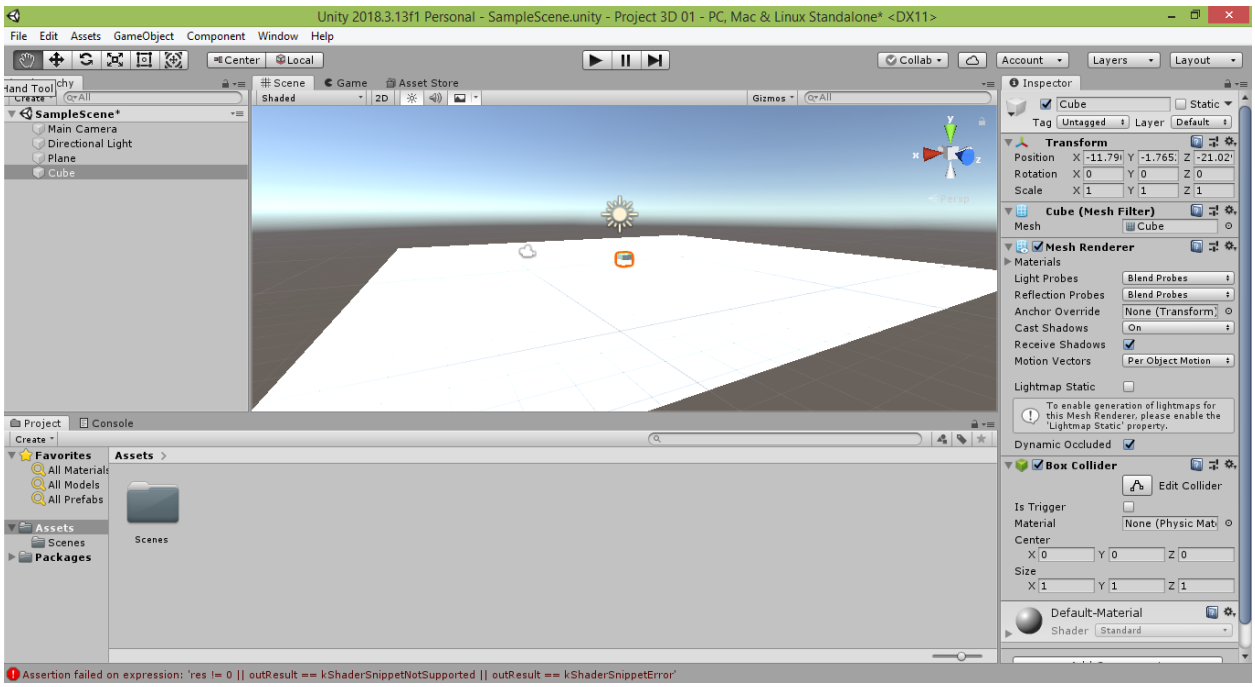
Чтобы изменить размер созданной площадки, выберите в панели инструментов (там, где вы видели кнопки с изображениями руки и глаза) кнопку «Rect Tool» («Инструмент-прямоугольник») с изображением квадратной рамки  (она находится в левом верхнем углу окна редактора Unity прямо под пунктом меню «GameObject»). После этого нажмите на площадку – у неё появится прямоугольный контур и точки в углах. Потяните за эти точки, чтобы изменить размер площадки.



Далее выберите в меню Unity команду «GameObject → 3D Object → Cube».



В результате на вашей сцене появится игровой 3D-объект «Cube» (в переводе с английского означает «Куб»).

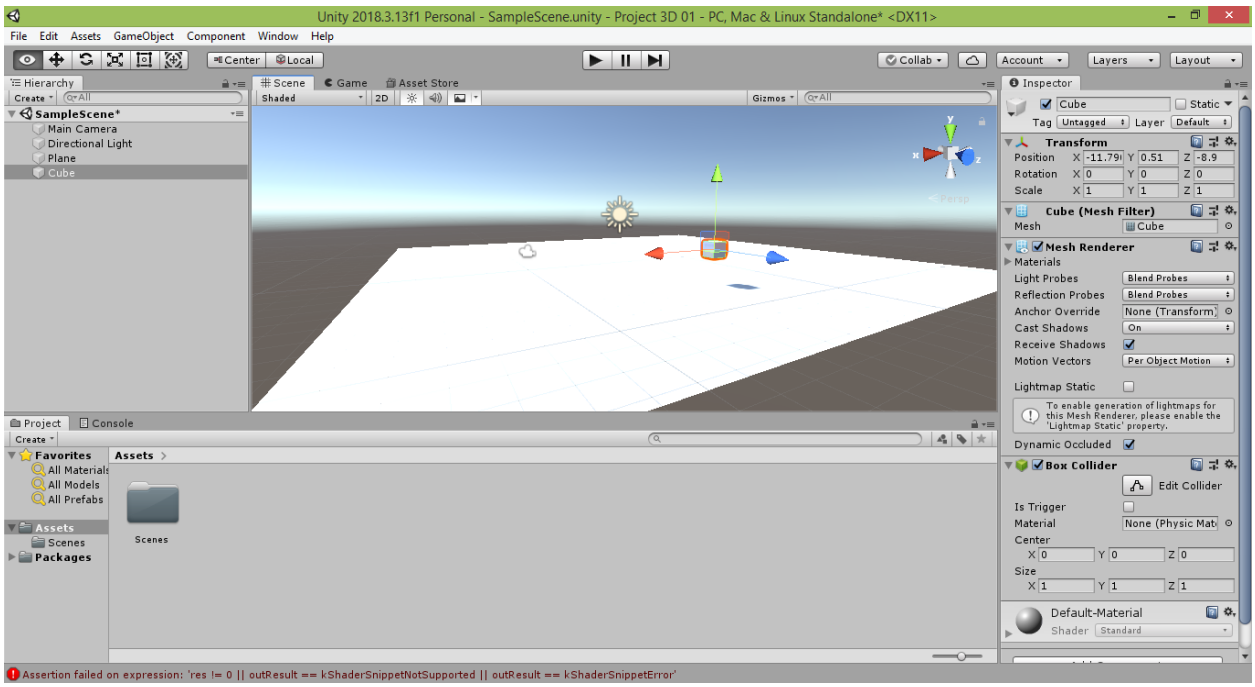


Возможна ситуация, что ваш куб будет виден частично или даже полностью не виден из-за перекрывающей его площадки «Plane». Чтобы изменить расположение куба, щёлкните в левом верхнем углу окна редактора Unity кнопку «Move Tool» («Инструмент перемещения»)  , расположенную справа от кнопки с изображением руки.

Теперь если щёлкнуть на куб или на соответствующую ему строчку «Cube» в окне «Hierarchy» («Иерархия»), у него появится система из трёх разноцветных стрелок (осей координат).

Красная стрелка показывает направление увеличения x-координаты объекта (по длине). Зелёная стрелка показывает направление увеличения y-координаты объекта (по высоте). Синяя стрелка показывает направление увеличения z-координаты объекта (по ширине).

Потянув поочерёдно за эти стрелки мышью, вы сможете переместить объект в нужном вам направлении. Если вам потребуется перемещать объект сразу по двум направлениям (в плоскости), щёлкните на квадратик между стрелками, соответствующими этим направлениями, и тяните за него его.

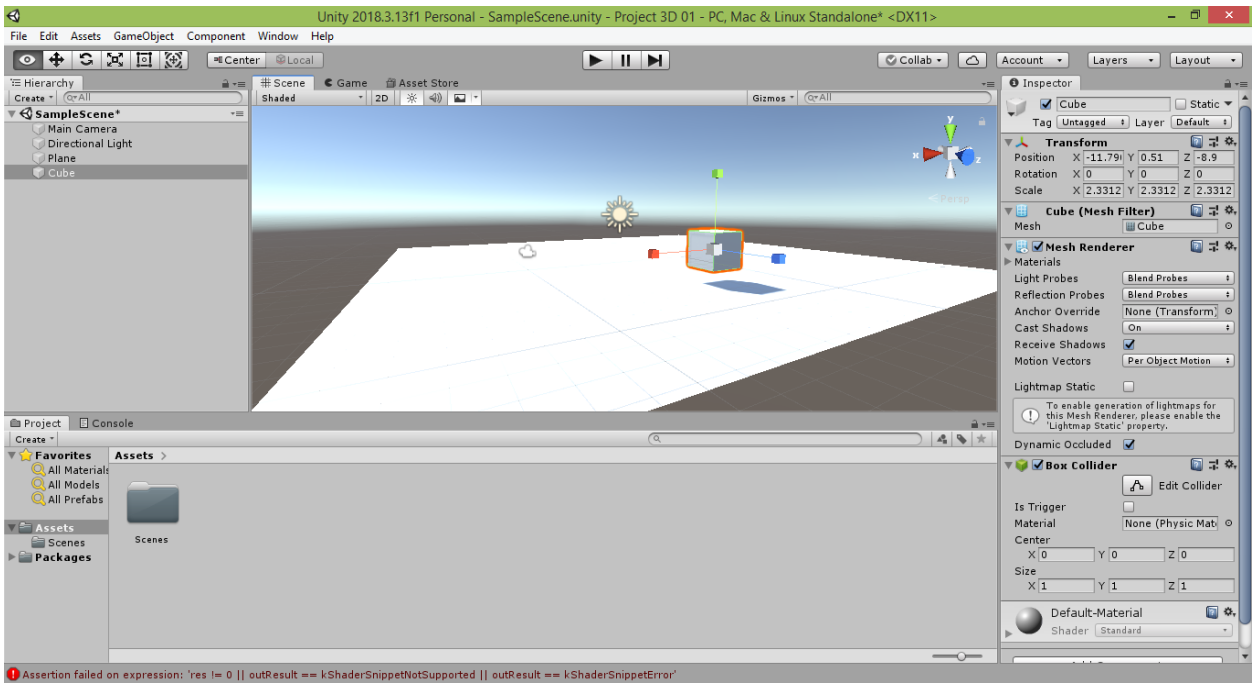


Расположите куб так над площадкой, чтобы на ней отобразилась его тень. Если вам кажется, что куб находится над площадкой, но тень от него всё равно не появляется, попробуйте переместить куб вплотную к площадке, потянув вниз за зелёную стрелку. В этом случае куб либо пристыкуется к площадке и отбросит на неё тень, либо окажется, что куб находится в стороне от площадки, и придётся перемещать.

Также вполне вероятно, что вы захотите изменить размер куба. Для этого щёлкните в левом верхнем углу окна редактора Unity кнопку «Scale Tool» («Инструмент масштабирования»)  , расположенную слева от кнопки «Rect Tool».

Теперь если щёлкнуть на куб или на соответствующую ему строчку «Cube» в окне «Hierarchy», у него появится система из трёх разноцветных рычажков с кубическими ручками на концах.

Красный рычажок обеспечивает увеличение длины объекта. Зелёный рычажок обеспечивает увеличение высоты объекта. Синий рычажок обеспечивает увеличение ширины объекта. В результате, вытянув наш куб по одной из осей, можно превратить его в параллелепипед, а если затем сжать его по другой оси, то он станет похож на плиту. Но если вы хотите растянуть или сжать куб одинаково по всем трём осям, тогда потяните его за серый кубический рычажок, расположенный в точке пересечения трёх разноцветных рычажков.



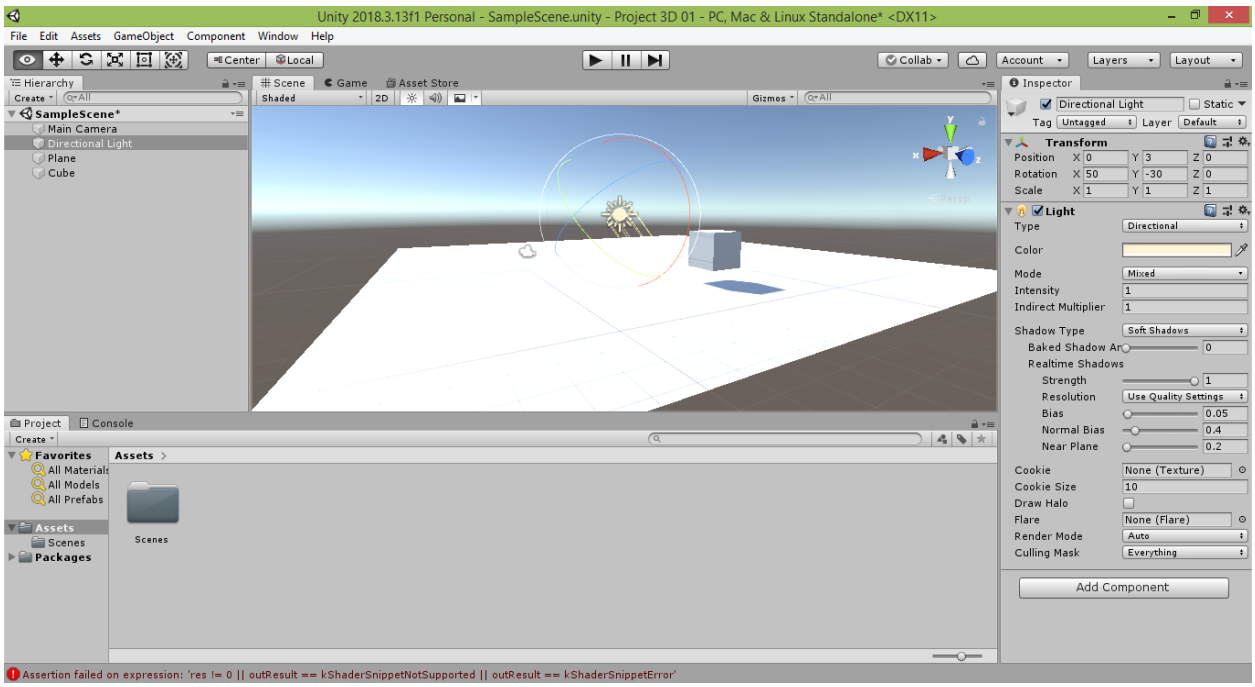
Обратите внимание, как вслед за изменением размера куба изменяется и размер его тени, отбрасываемой на площадку. Возможно, вам захочется, чтобы тень имела другую форму и размеры. Для этого потребуется изменить угол падения лучей солнца на нашу сцену. Для этого щёлкните в левом верхнем углу окна редактора Unity кнопку «Rotate Tool» («Инструмент поворота»)  , расположенную справа от кнопки «Move Tool».

Теперь если щёлкнуть на изображение солнца или на соответствующую ему строчку «Directional Light» («Направленный свет») в окне «Hierarchy», вокруг него появится система из трёх разноцветных окружностей.

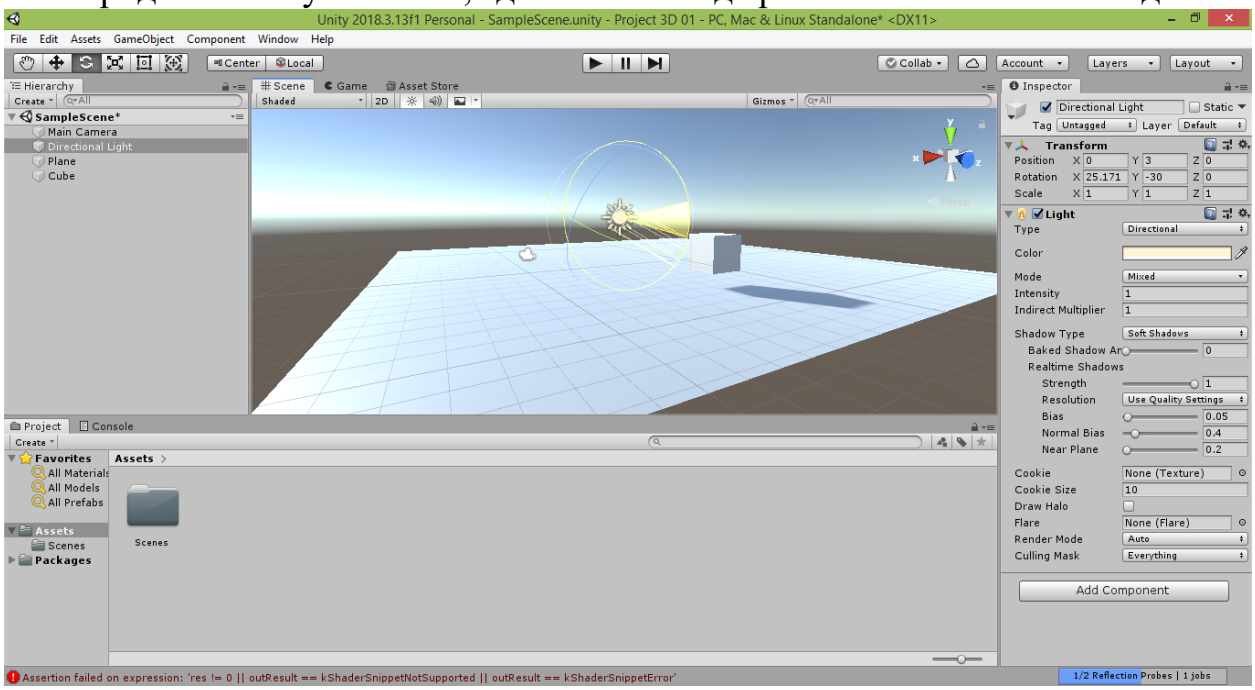
Красная окружность обеспечивает вращение вокруг красной оси. При этом остаются неизменными x-координаты точек вращаемого объекта.

Зелёная окружность обеспечивает вращение вокруг зелёной оси. При этом остаются неизменными y-координаты точек вращаемого объекта.

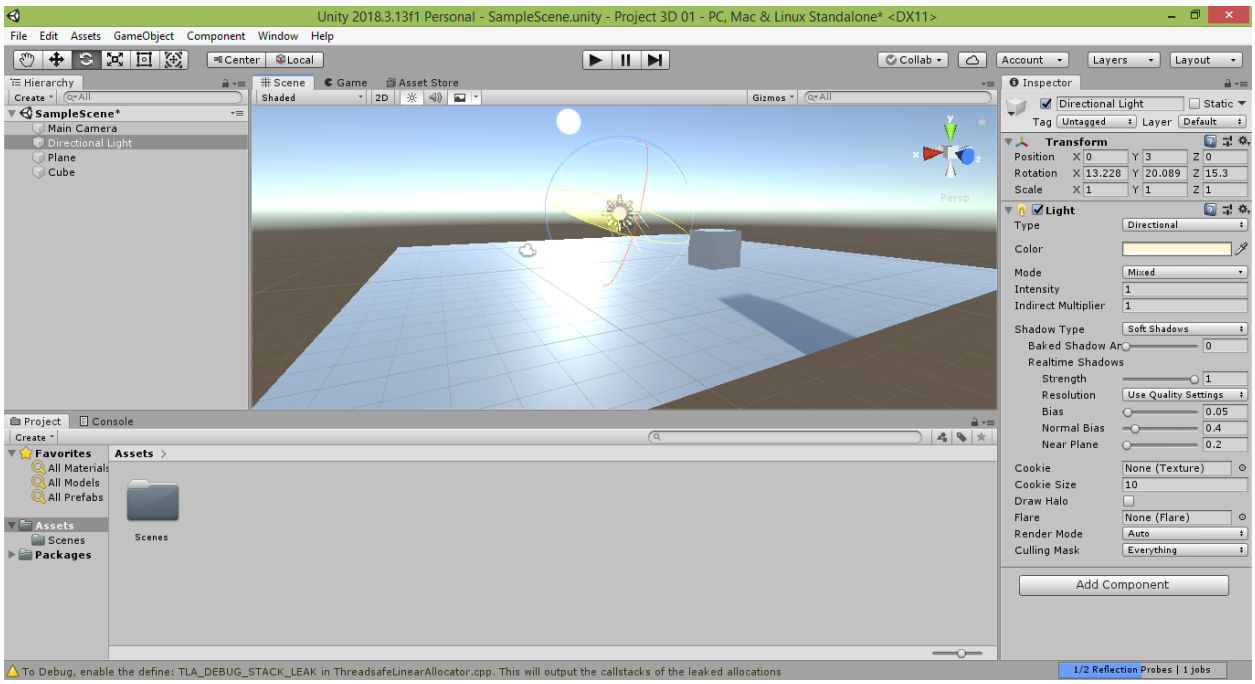
Синяя окружность обеспечивает вращение вокруг синей оси. При этом остаются неизменными z-координаты точек вращаемого объекта.



Попробуйте потянуть за красную окружность. Вы увидите, как изменится вертикальный угол падения солнечных лучей – угол поворота будет помечен сектором жёлтого цвета. При этом тень объекта изменит свою длину. Также изменят своё положение жёлтые отрезки, исходящие из значка солнца, – их продолжение указывает, где на небосводе расположен солнечный диск.



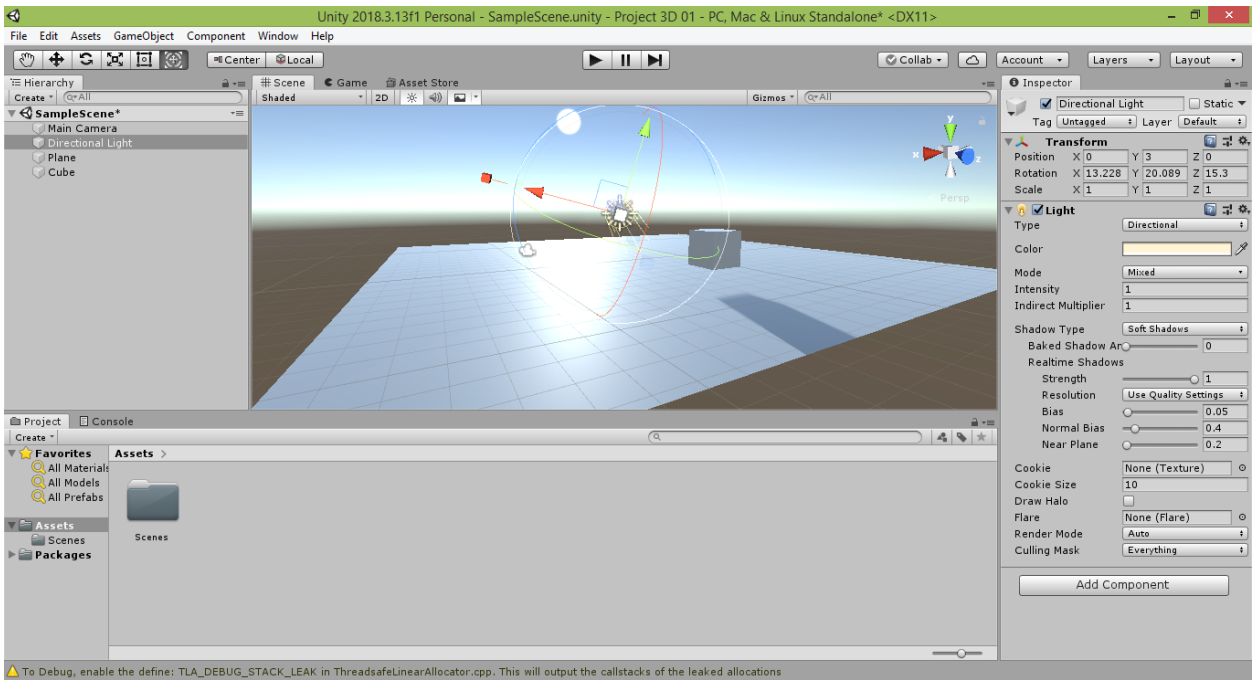
Теперь попробуйте потянуть за зелёную окружность. Вы увидите, как изменится горизонтальный угол падения солнечных лучей – угол поворота будет помечен сектором жёлтого цвета. При этом тень объекта начнёт двигаться влево или вправо – соответственно, по ходу движения часовой стрелки или против хода её движения. Вслед за ней будут менять своё положение и жёлтые отрезки, исходящие из значка солнца.



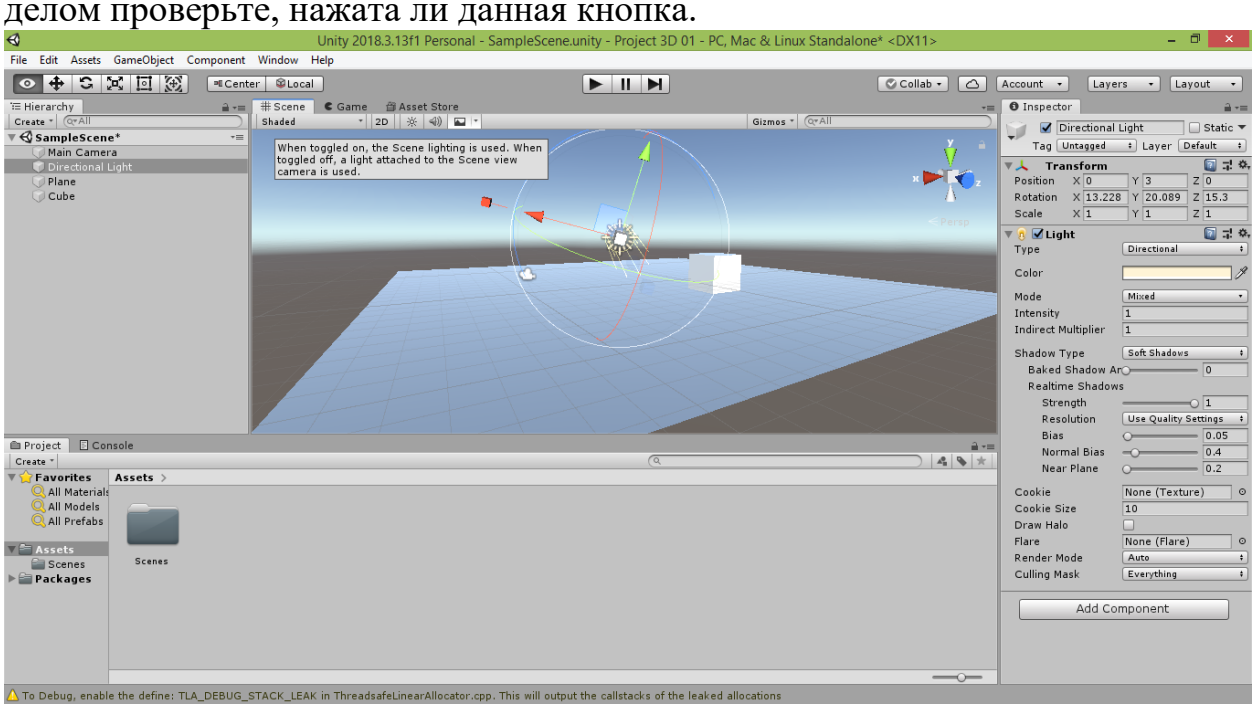
Также вы можете потянуть и за синюю окружность. При этом сразу эффекта вы не увидите, поскольку, по сути, вы вращаете круглый солнечный диск вокруг его центра. Результат изменения угла поворота проявится, если вы вновь попробуете потянуть за зелёную окружность – траектория движения солнца по небосводу будет другой. Таким образом, для направленного источника света синяя окружность позволяет задать положение траектории его движения по отношению к горизонту. С её помощью вы можете сделать так, чтобы солнце двигалось параллельно горизонту и никогда не заходило, или, наоборот, чтобы оно двигалось перпендикулярно горизонту и в результате всходило или заходило максимально быстро. Эта настройка может особенно пригодиться в тех случаях, когда вы имитируете заданное соотношение продолжительности дня и ночи в определённое время года на определённой широте.

Для других видов объектов изменение угла по синей окружности приведёт просто к их повороту вокруг синей оси.

Если вам потребуется производить сложные комбинированные изменения объекта (например, его перемещение с последующими поворотом и изменением масштаба), щёлкните в левом верхнем углу окна редактора Unity кнопку «Move, Rotate or Scale selected objects» («Переместить, повернуть или масштабировать выделенные объекты»)  , расположенную справа от кнопки «Rect Tool». В результате у объекта появятся одновременно все три системы, обеспечивающие рассмотренные виды его трансформации, – систем осей перемещения, система рычажков масштабирования и система окружностей поворота. С их помощью вы сможете быстро трансформировать объект так, как вам захочется.

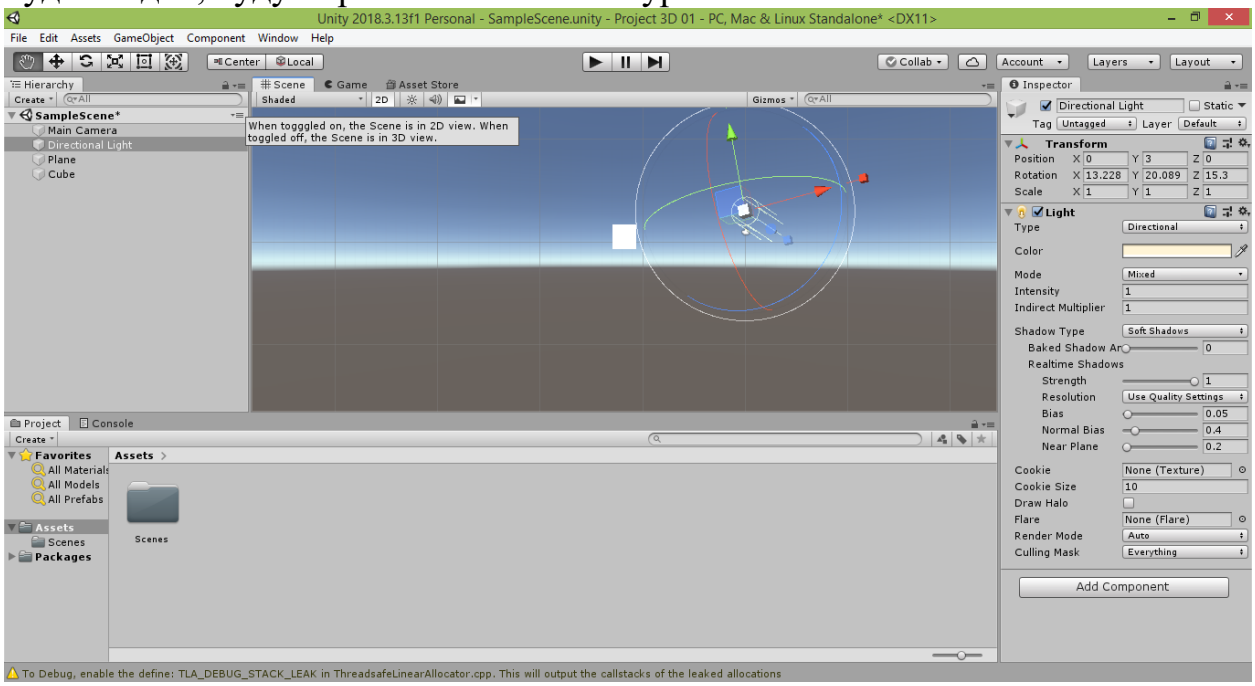


Также следует обратить внимание на небольшую кнопку с изображением солнца  в строке над рабочим полем сцены (там же, где расположена кнопка с изображением панорамы ). Попробуйте нажать на неё. В результате солнечный диск на небосводе исчезнет. Исчезнут и эффекты освещения объекта, включая отбрасываемые им тени. Чтобы вернуть их обратно, снова нажмите на эту кнопку. Таким образом, если вы столкнулись с ситуацией, когда у объектов вашей сцены отсутствуют тени и блики, первым делом проверьте, нажата ли данная кнопка.



Левее данной кнопки (также в строке над рабочим полем сцены) расположена кнопка с надписью «2D»  . Попробуйте нажать на неё. В этом случае обзор сцены изменится так, что вы будете смотреть на неё, находясь

продолжении синей оси, которая отображалась у куба при нажатии кнопки «Move Tool». Куб будет отображаться как квадрат, а плоская площадка не будет видна, будучи расположенной на уровне наших глаз.

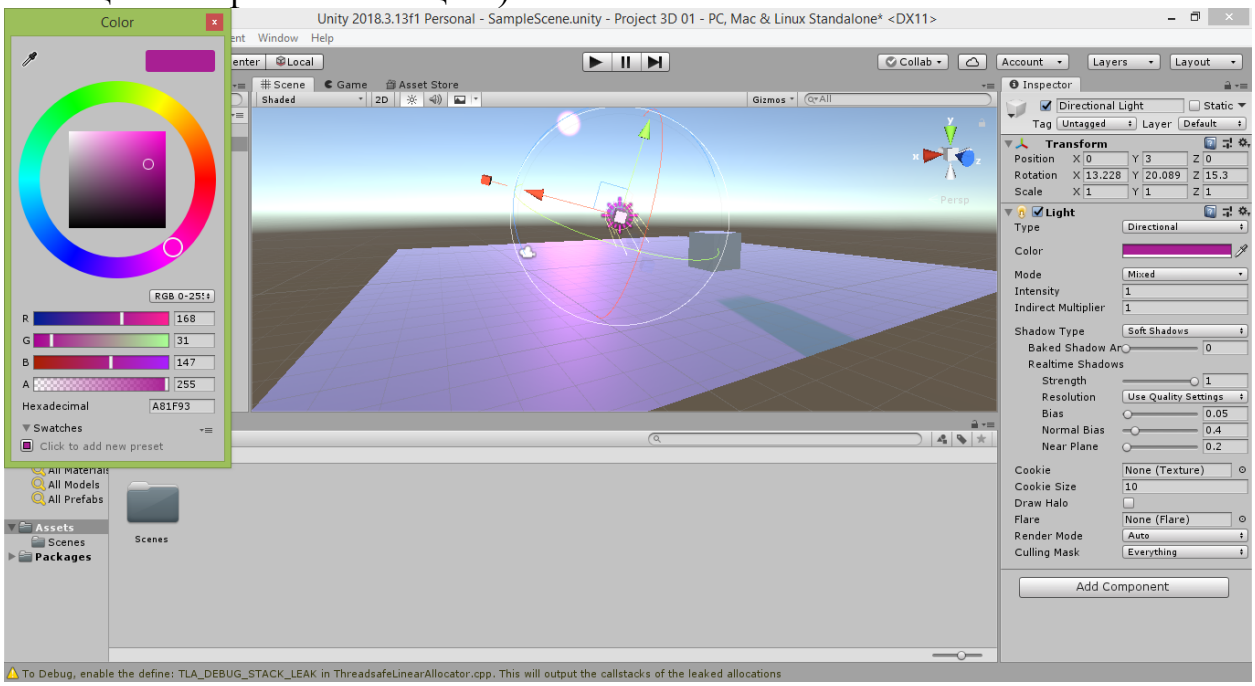


Тут следует отметить, что площадка «Plane» становится прозрачной при взгляде на неё снизу. Чтобы площадка стала непрозрачной, мы должны изменить положение точки своего обзора и посмотреть на неё сверху.

Вернитесь обратно в трёхмерный режим работы со сценой, снова щёлкнув на кнопку с надписью «2D».

2.3. Размещение объектов и настройка их освещения

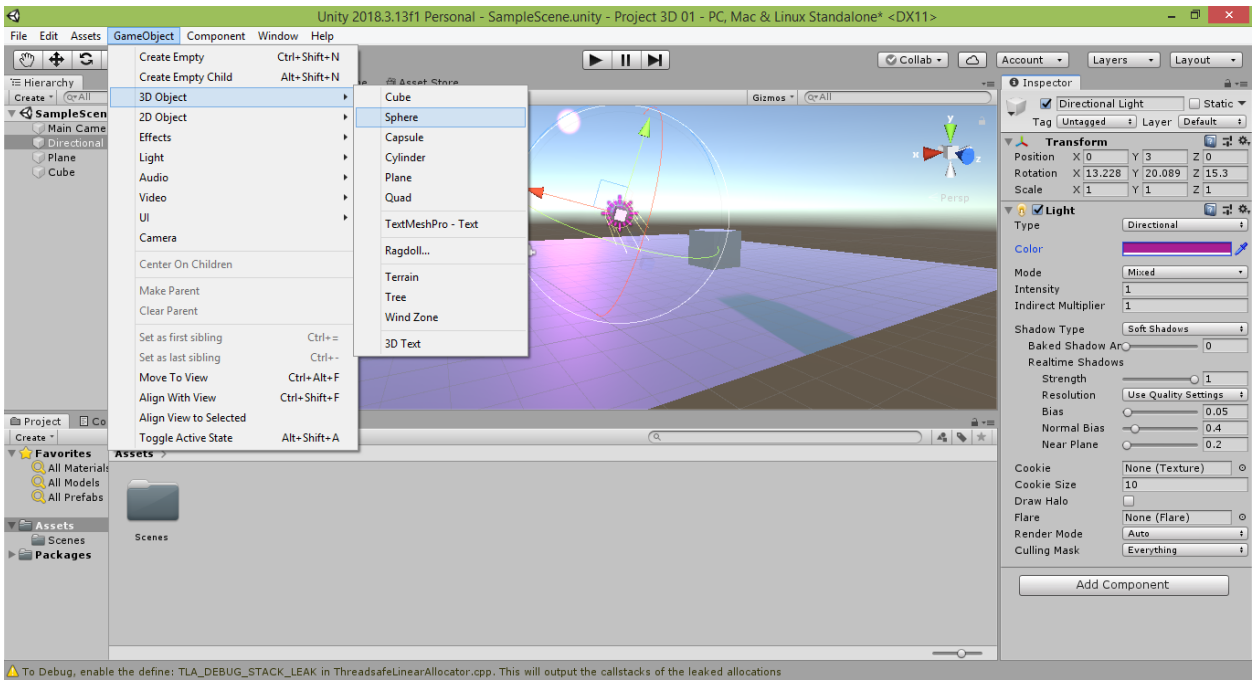
Теперь перейдём к настройке источников света нашей сцены. Для начала убедитесь, что на сцене выделен значок с изображением солнца. Если это не так, щёлкните мышкой на нём или на соответствующую ему строчку «Directional Light» («Направленный свет») в окне «Hierarchy». Справа в окне «Inspector» отобразятся свойства и настройки этого источника света. Щёлкните на прямоугольнике в свойстве «Color». В результате слева появится окно с палитрой выбора цвета. Щёлкая мышкой по окружности, вы можете выбрать нужный вам цвет, а щёлкая в области расположенного в центре квадрата – задать насыщенность выбранного цвета (я выбрал достаточно насыщенный фиолетовый цвет).



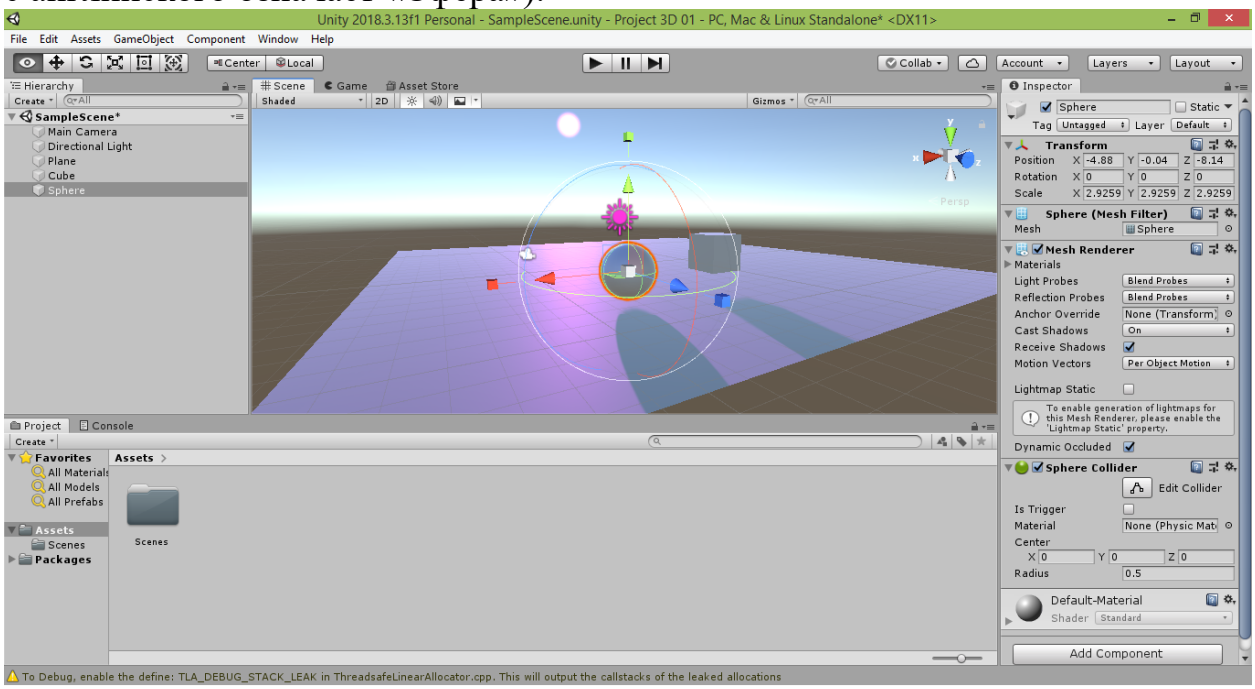
В результате вся сцена окрасится тем цветом, который вы выбрали, а вокруг солнечного диска появится небольшой контур выбранного цвета. Таким образом, свойство «Color» позволяет задать цвет испускаемого источником света.

Чтобы создаваемые нами световые эффекты были нагляднее, добавим на сцену объекты других форм.

Выберите в меню Unity команду «GameObject → 3D Object → Sphere».

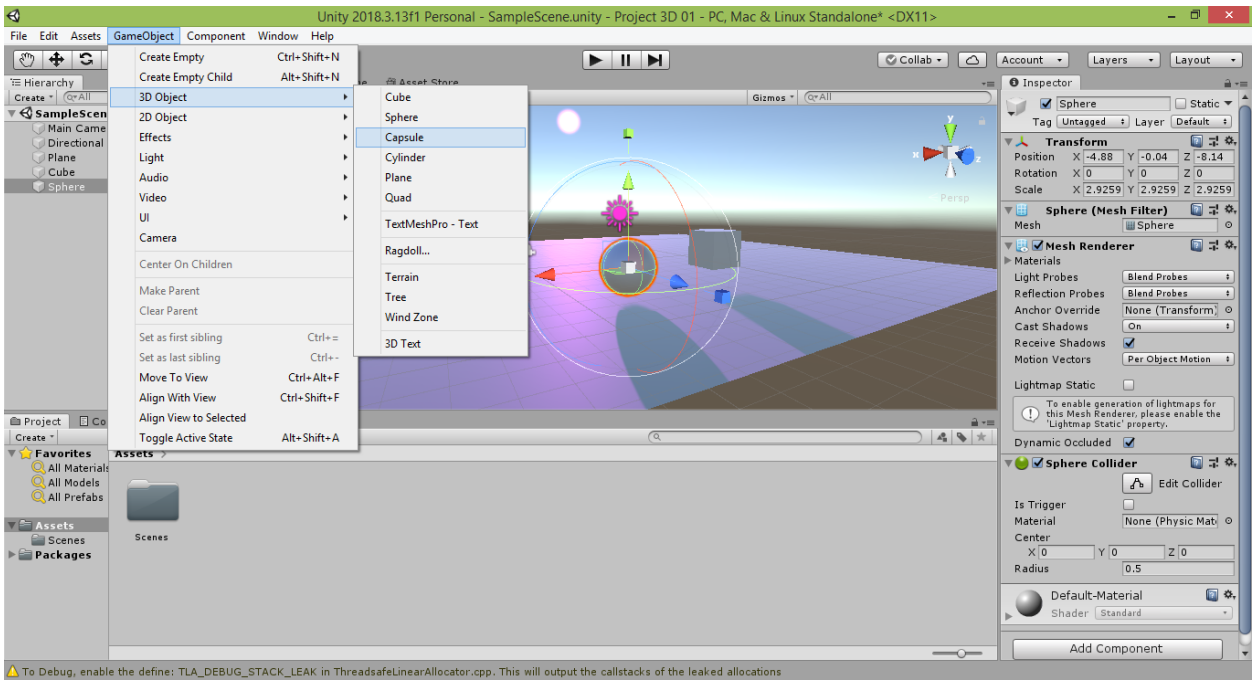


В результате на сцене появится игровой 3D-объект «Sphere» (в переводе с английского означает «Сфера»).

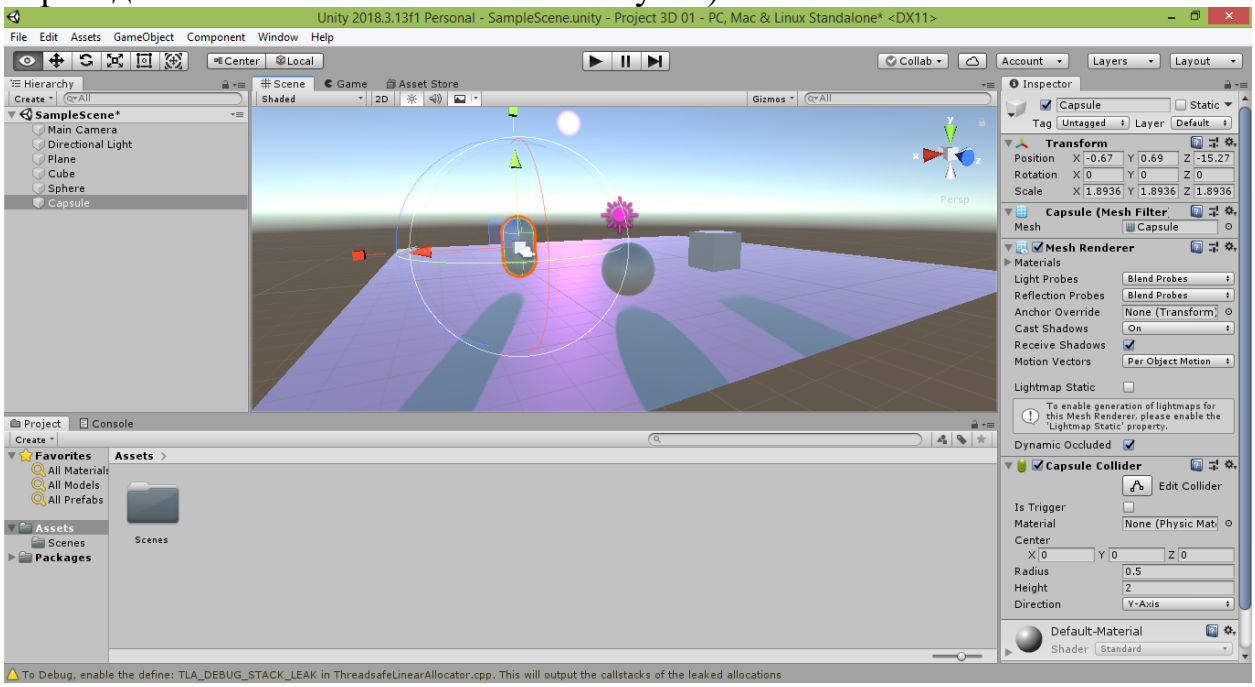


При помощи инструмента «Move, Rotate or Scale selected objects» измените положение и размер сферы так, чтобы вам было удобно видеть её.

Далее выберите в меню Unity команду «GameObject → 3D Object → Capsule».

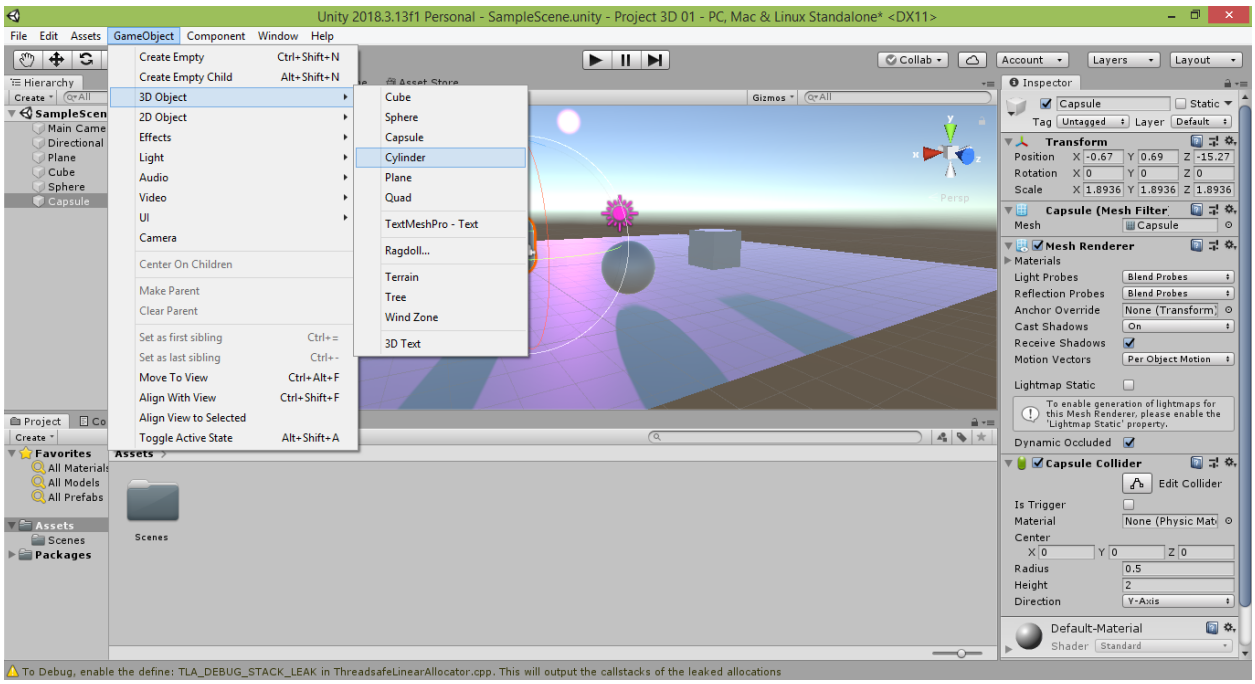


В результате на вашей сцене появится игровой 3D-объект «Capsule» (в переводе с английского означает «Капсула»).

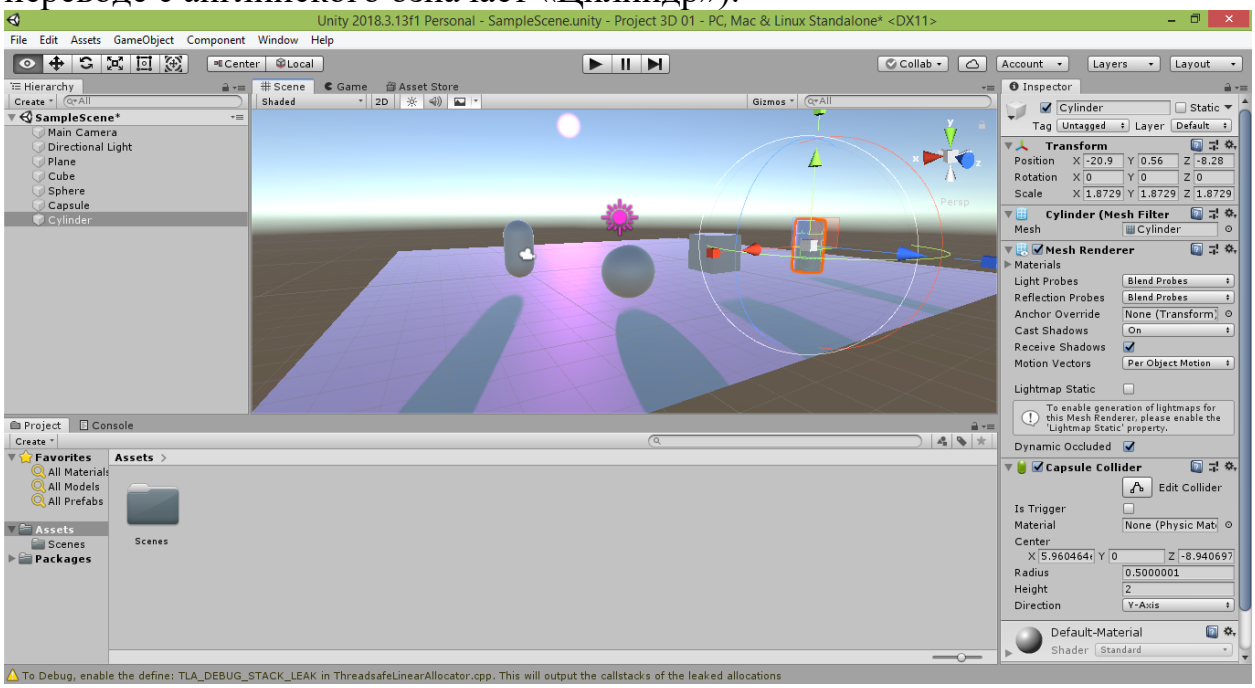


При помощи инструмента «Move, Rotate or Scale selected objects» измените положение и размер капсулы так, чтобы вам было удобно видеть её.

Теперь выберите в меню Unity команду «GameObject → 3D Object → Cylinder».

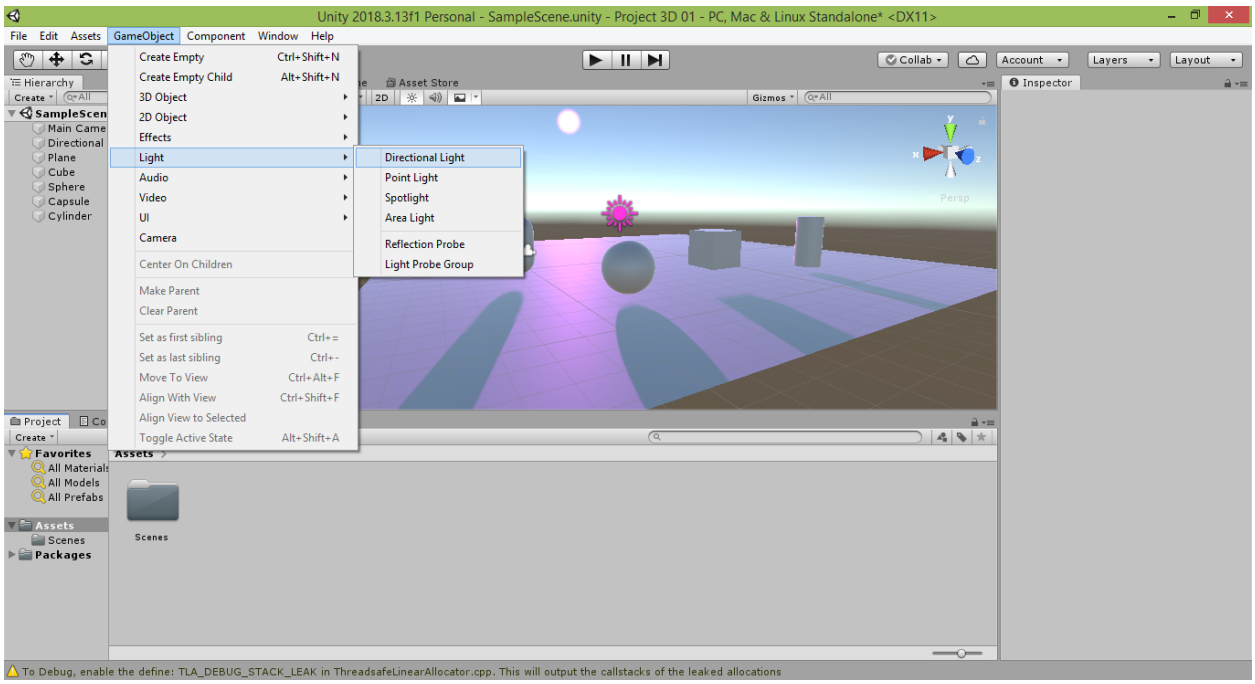


В результате на вашей сцене появится игровой 3D-объект «Cylinder» (в переводе с английского означает «Цилиндр»).

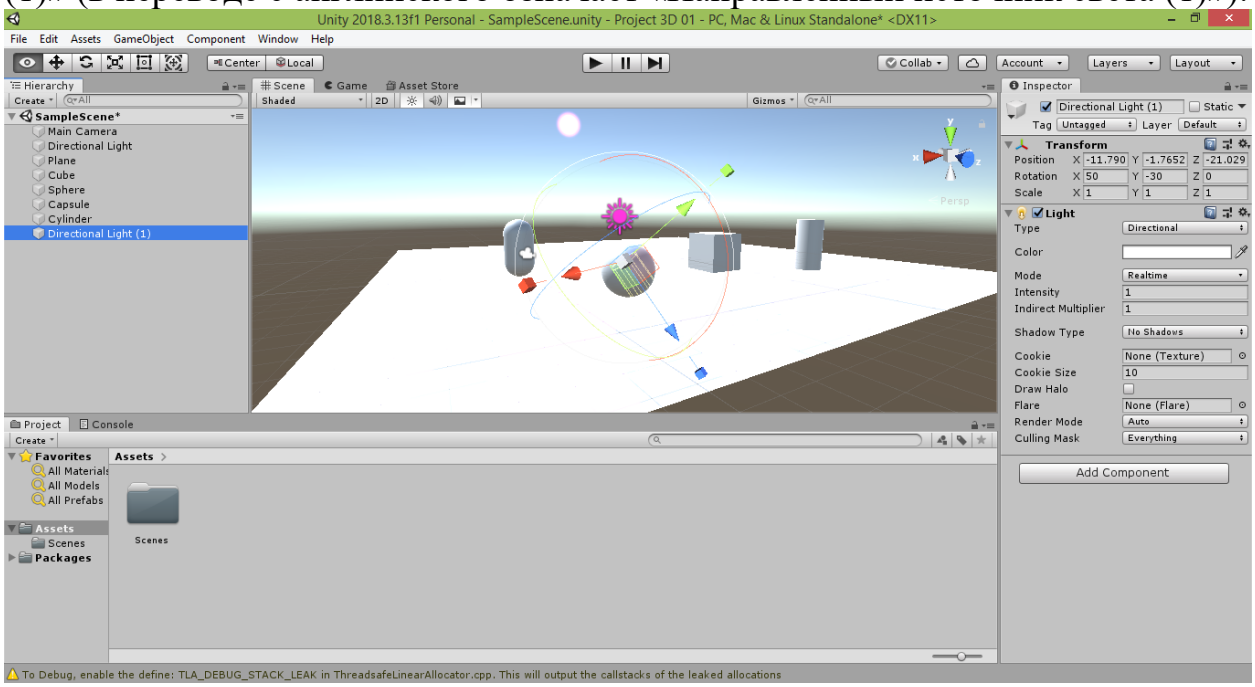


При помощи инструмента «Move, Rotate or Scale selected objects» измените положение и размер цилиндра так, чтобы вам было удобно видеть его.

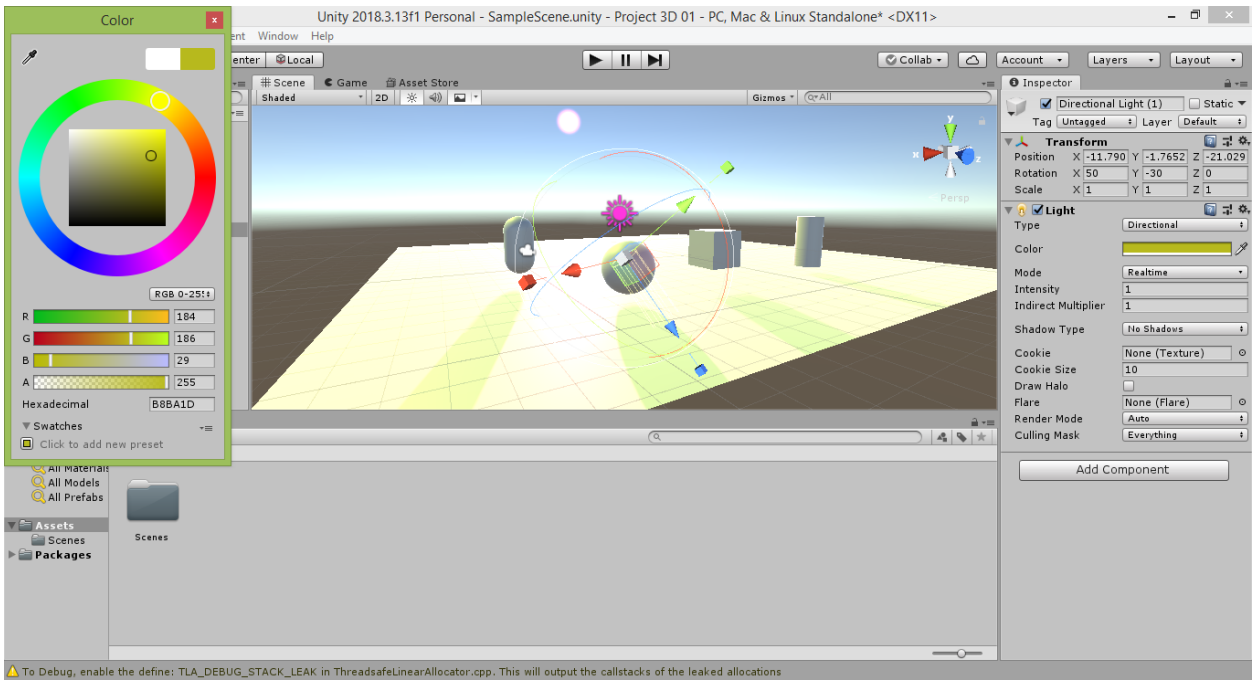
Теперь, когда объекты размещены на сцене, добавим дополнительный источник света. Выберите в меню Unity команду «GameObject → Light → Directional Light».



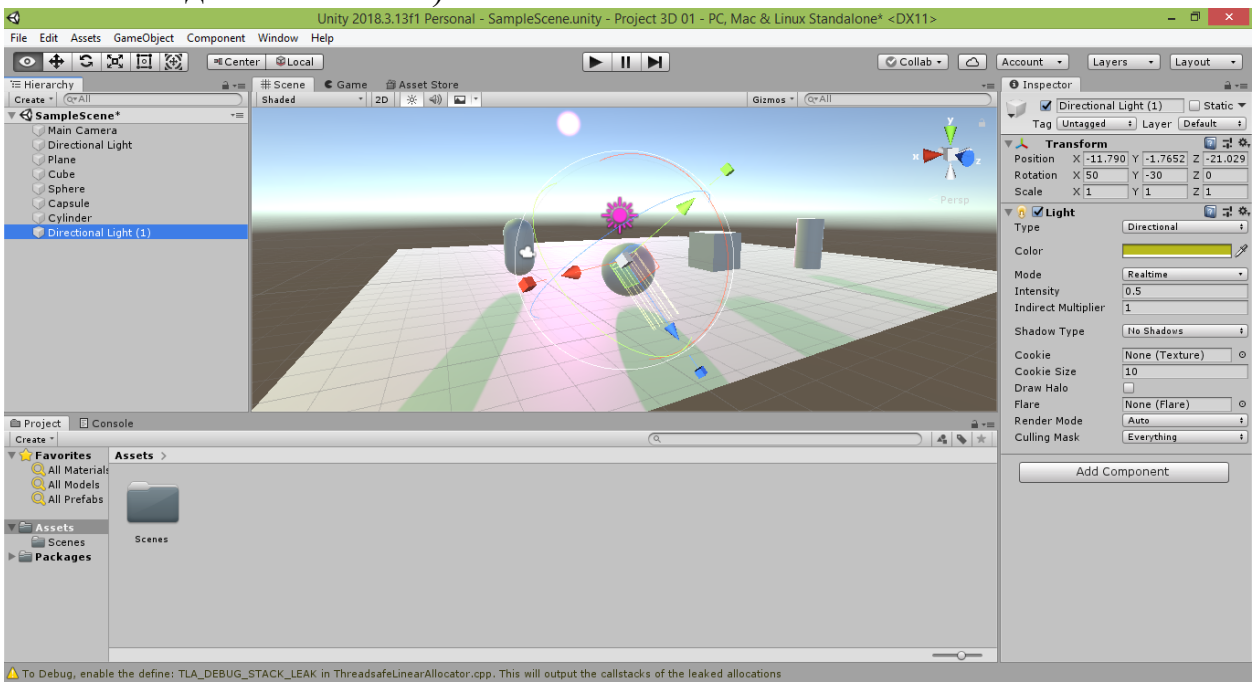
В результате на вашей сцене появится игровой объект «Directional Light (1)» (в переводе с английского означает «Направленный источник света (1)»).



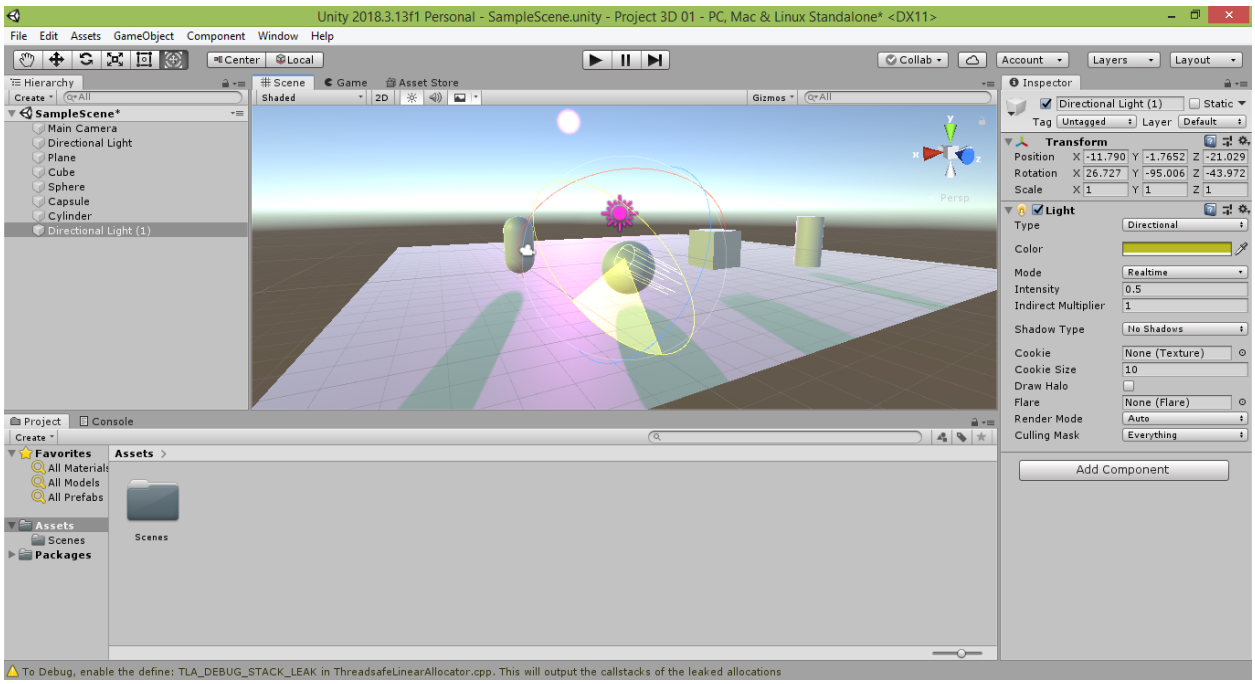
Измените его цвет в свойстве «Color» окна «Inspector» (я выбрал насыщенный жёлтый цвет).



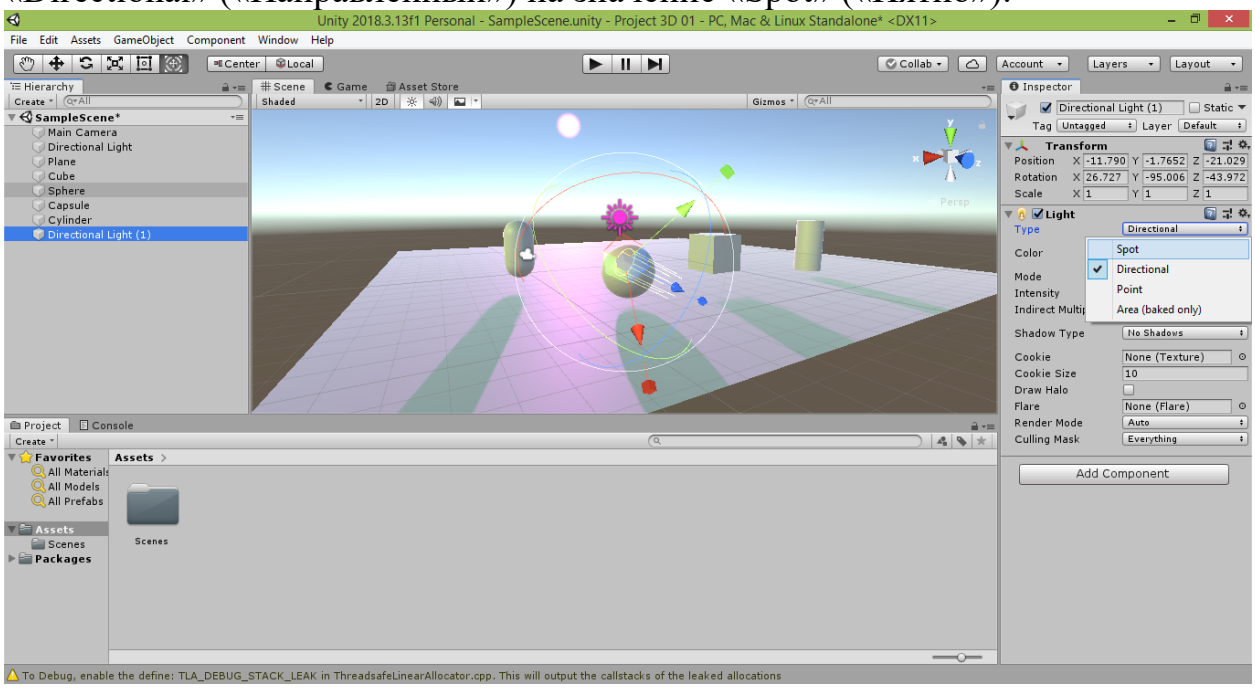
Если свет, испускаемый дополнительным источником, слишком яркий уменьшите значение свойства «Intensity» («Интенсивность») в окне «Inspector» (я уменьшил интенсивность испускаемого света в 2 раза: со значения 1 до значения 0.5).



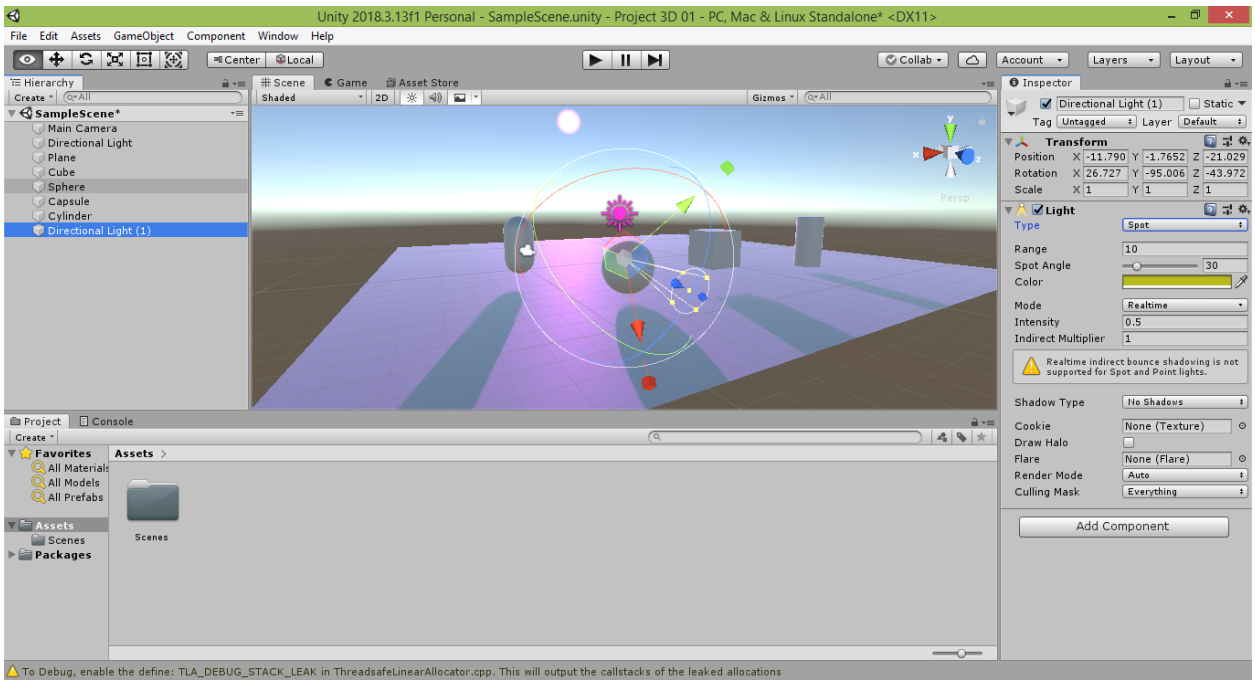
Вы увидите, что теперь объекты сцены освещаются двумя солнцами разных цветов (фиолетовым и жёлтым в моём примере). Чтобы это было более наглядно, измените угол падения лучей дополнительного источника света.



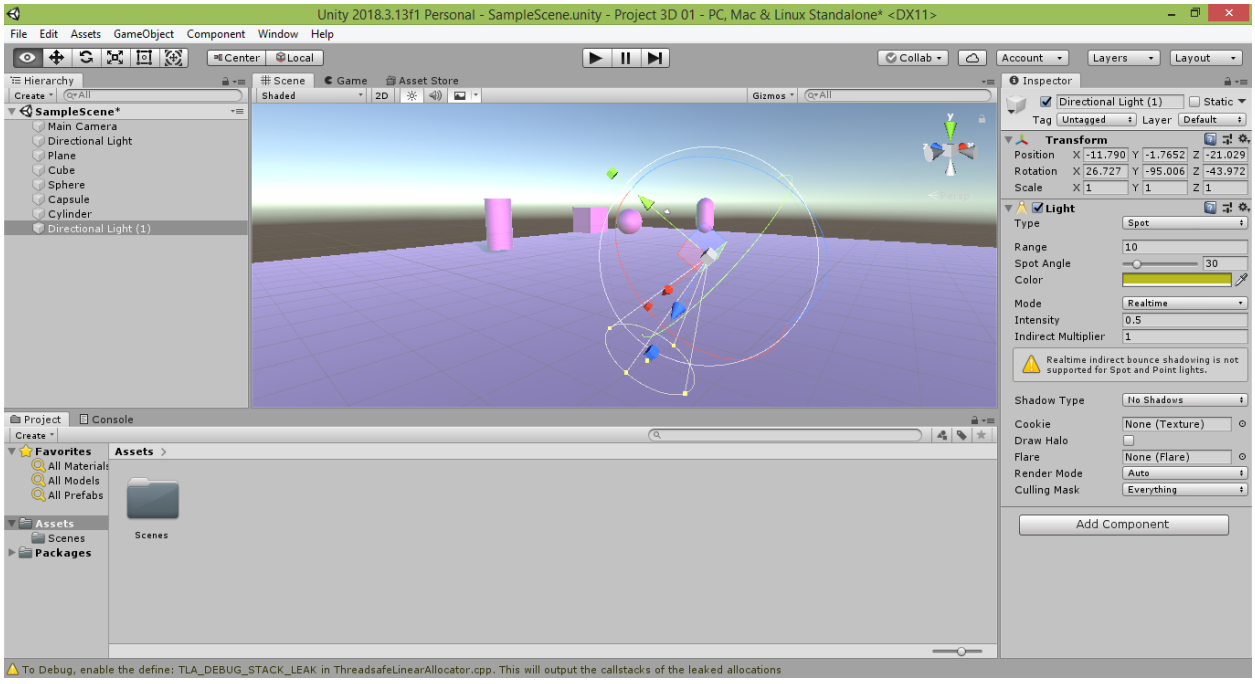
Попробуем изменить тип дополнительного источника света. Для этого в его свойстве «Type» («Тип») в окне «Inspector» поменяйте значение «Directional» («Направленный») на значение «Spot» («Пятно»).



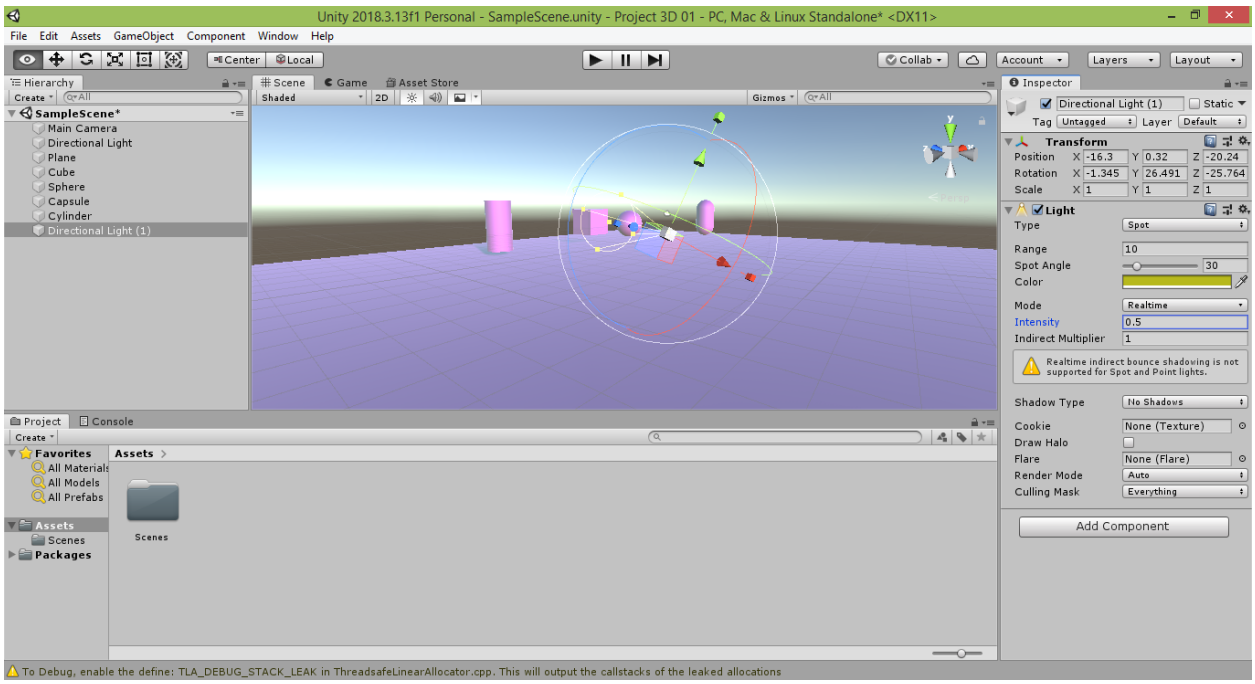
В результате направленный свет исчезнет, а вместо значка с изображением солнца и жёлтых отрезков-лучей появится значок с изображением фонарика и жёлтый конусовидный контур, схематично изображающий движение лучей от этого фонарика.



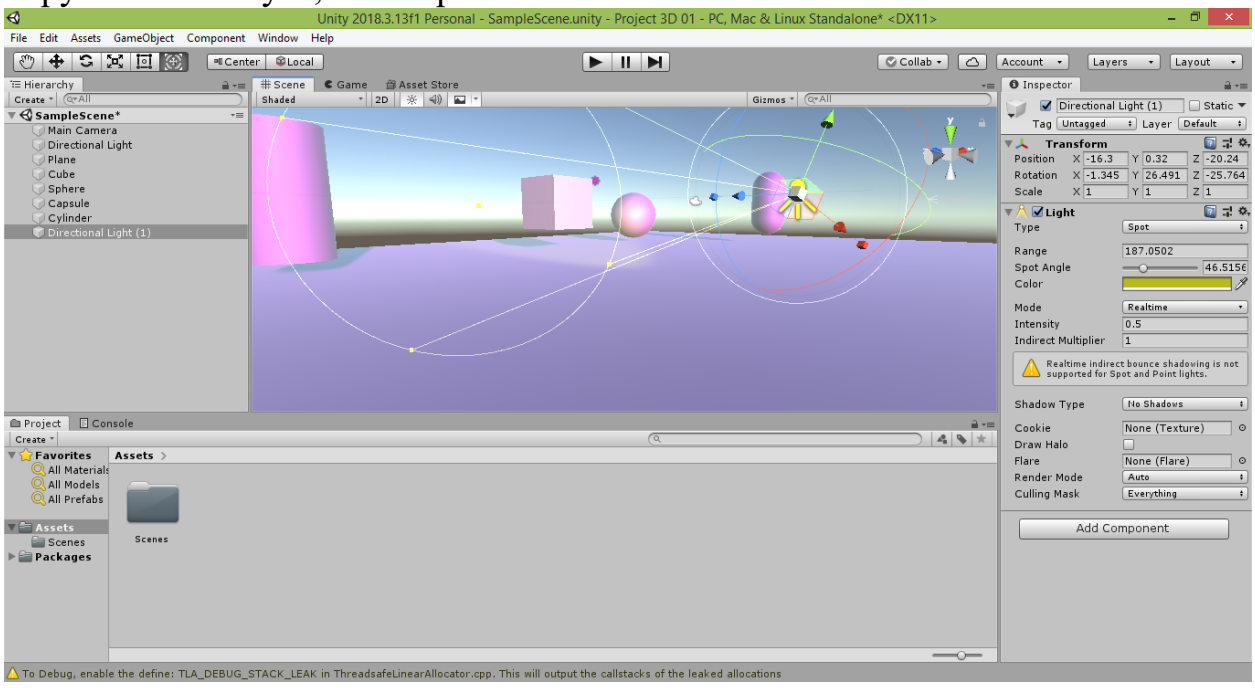
Чтобы настроить освещение объектов сцены этим фонариком переместитесь по сцене и переместите дополнительный источник света поближе к этим объектам.



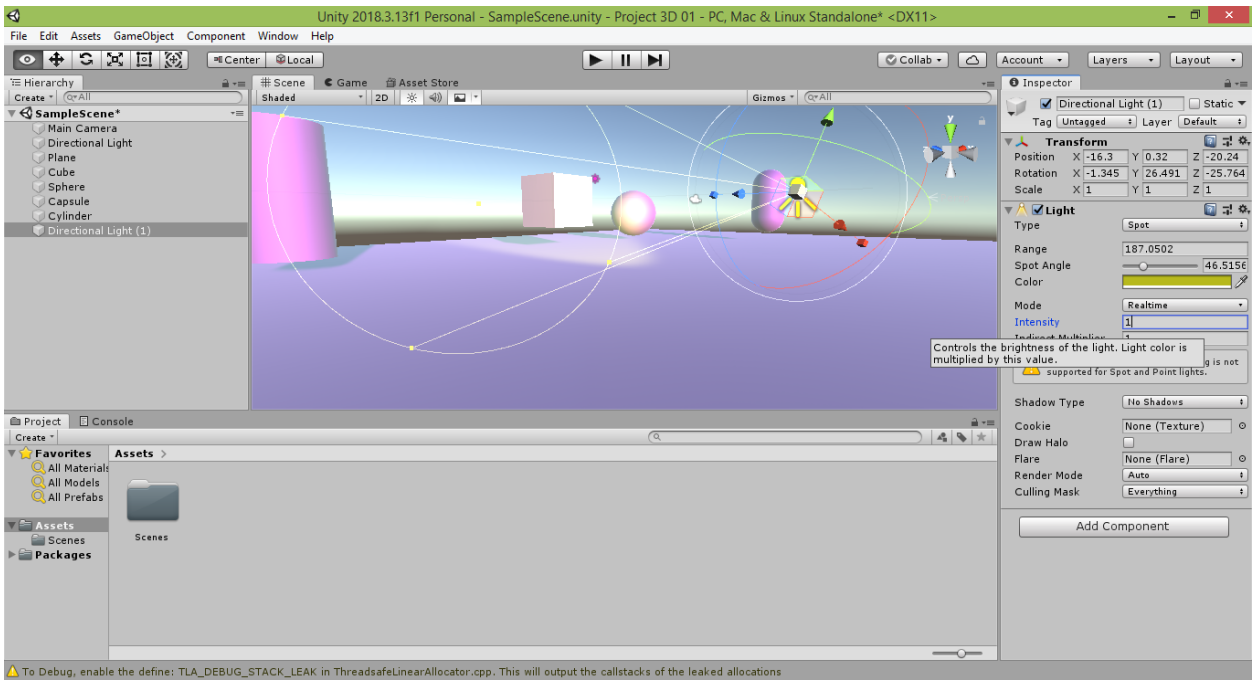
После этого измените угол поворота дополнительного источника света так, чтобы его лучи были направлены в сторону объектов сцены.



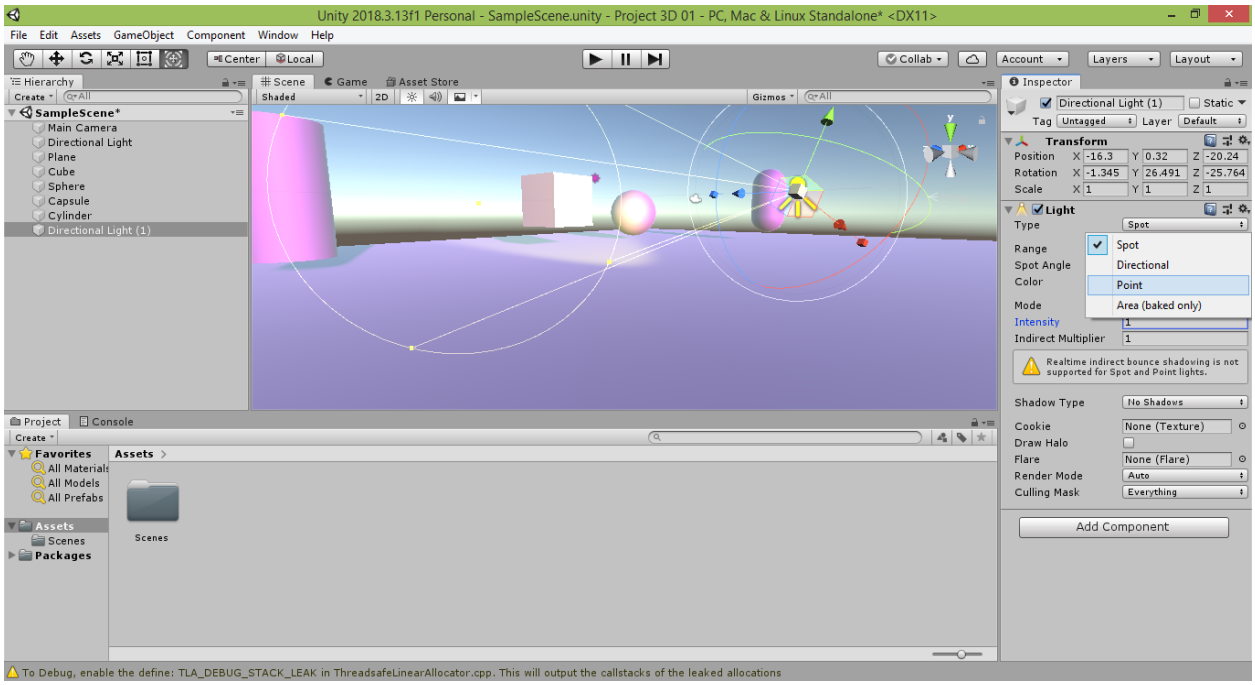
Чтобы световое пятно фонарика стало крупнее, потяните за любой из жёлтых квадратиков окружности конуса. Чтобы лучи фонарика доставали до объектов сцены, потяните за жёлтый квадратик, расположенный в центре окружности конуса, по направлению к объектам.



В результате вы увидите, как на объектах сцены появляется световое пятно, как если бы они освещались лучами фонарика. Чтобы усилить свет, идущий от фонарика, попробуйте увеличить значение свойства «Intensity» в окне «Inspector» (я увеличил интенсивность испускаемого света в 2 раза: со значения 0.5 обратно до значения 1). В результате световое пятно станет более чётким.



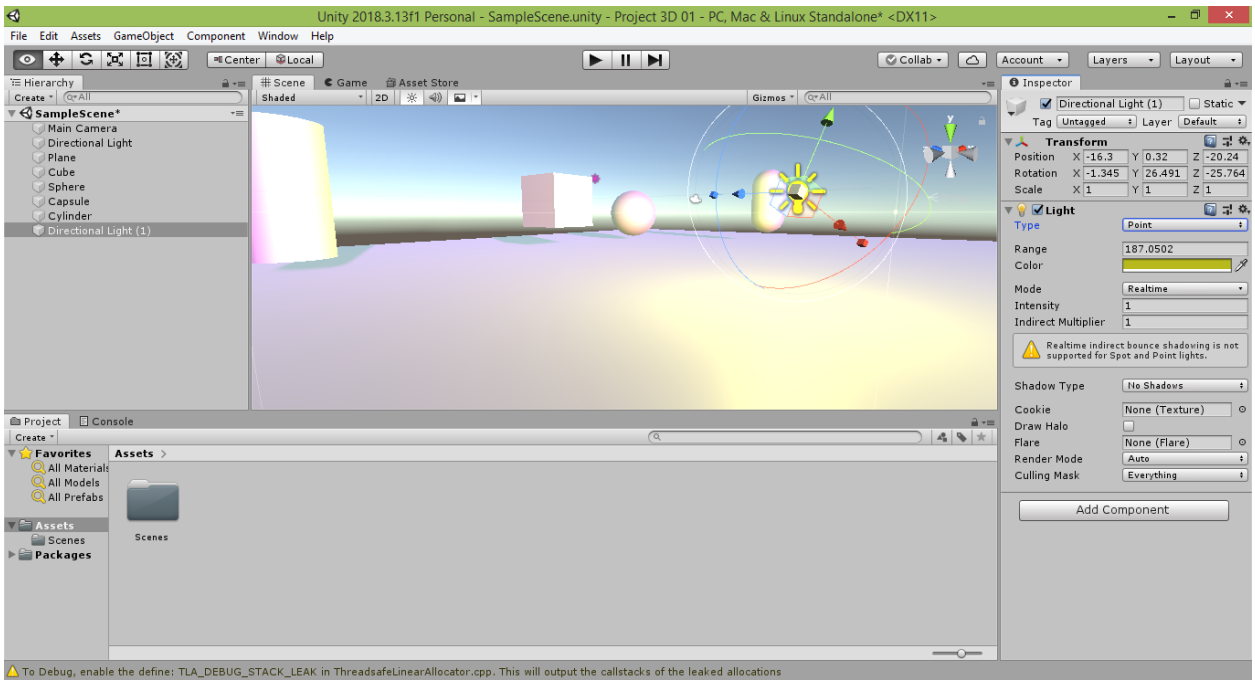
Теперь попробуйте снова изменить тип дополнительного источника света.



Для этого в его свойстве «Type» в окне «Inspector» поменяйте значение «Spot» («Пятно») на значение «Point» («Точка»).

В результате источник света будет освещать объекты сцены уже не в одном направлении, а во всех, как будто вместо фонарика на сцене появилась висящая в воздухе лампочка, испускающая световые лучи во всех направлениях.

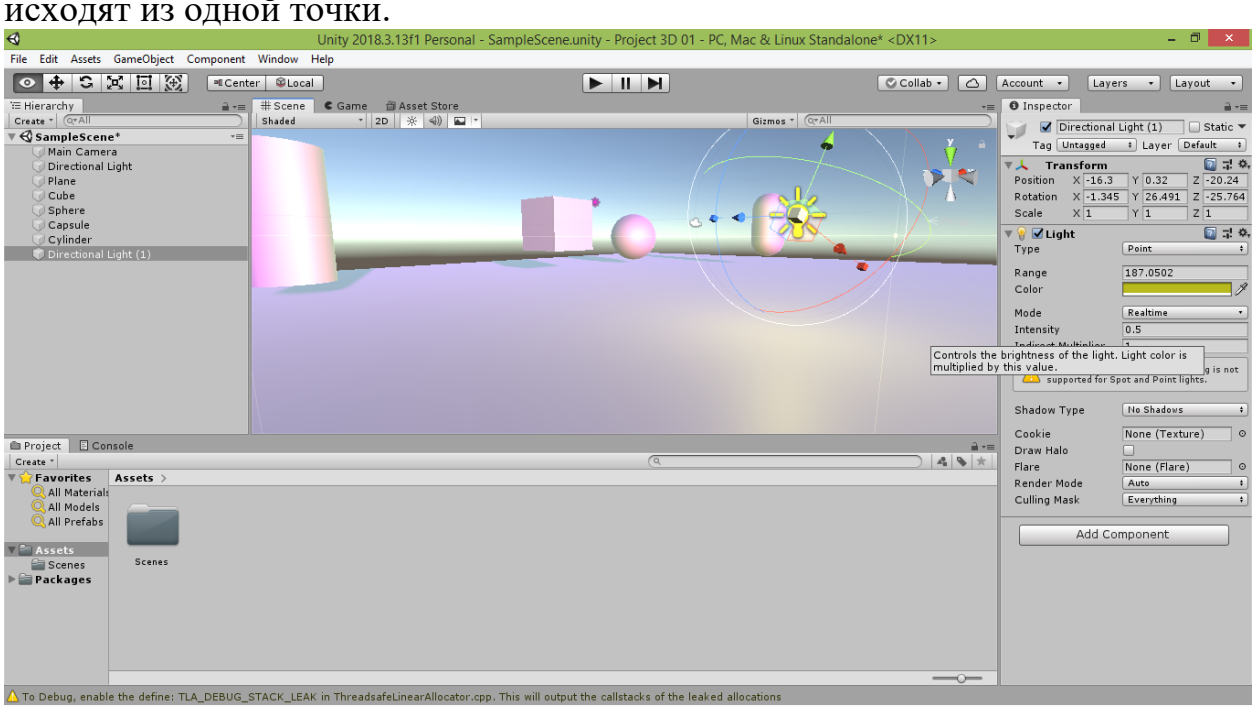
Значок источника света при смене его типа будет изменён с изображения фонарика на изображение лампочки.



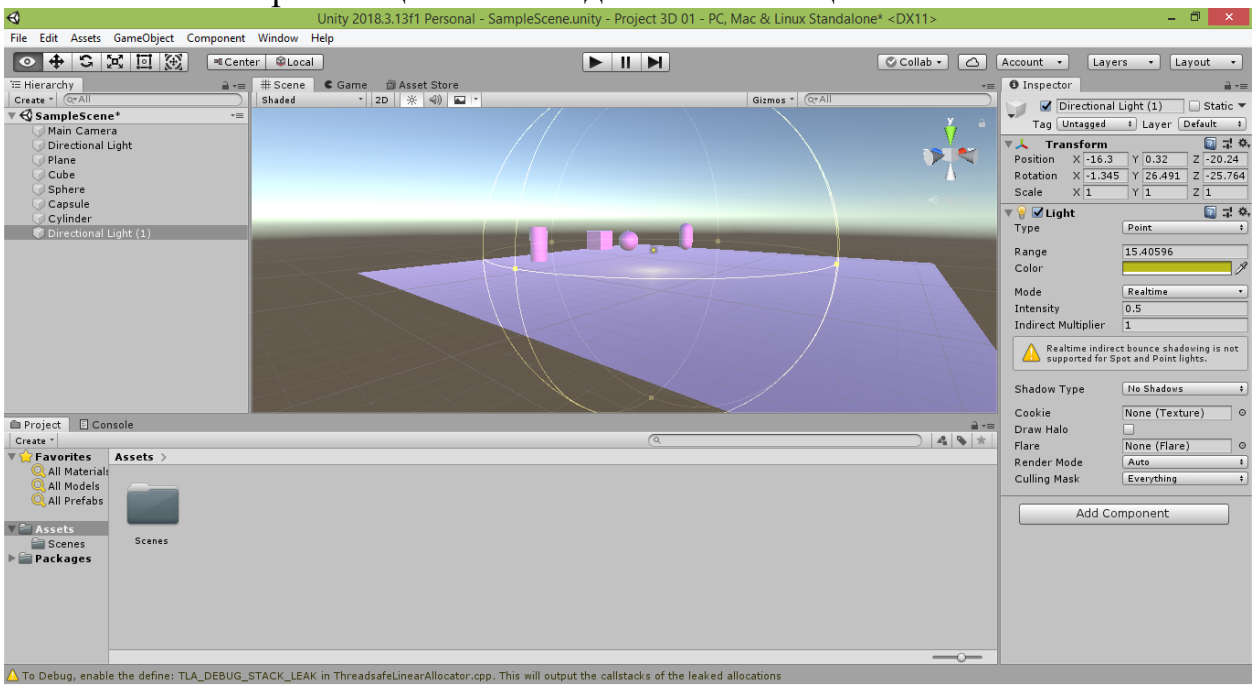
Если потребуется ослабить свет, идущий от лампочки, попробуйте уменьшить значение свойства «Intensity» в окне «Inspector» (я снова уменьшил интенсивность испускаемого света в 2 раза: со значения 1 до значения 0.5).

В результате вы увидите, как лампочка слегка подсвечивает объекты сцены, которые при этом одновременно освещаются более интенсивным и равномерным светом лучей солнца.

Солнце как источник света имеет тип «Directional». Это означает, что оно (в отличие от типов «Spot» и «Point») имеет не точечный размер, а более крупный (область светящихся точек) и при этом находится достаточно далеко в пространстве. В результате лучи солнца идут параллельно друг другу, а не исходят из одной точки.



Также вы можете расширить дальность действия лучей лампочки, если потянете за любой из жёлтых квадратиков сферы, окружающей лампочку и схематично изображающей область действия освещения лампочки.

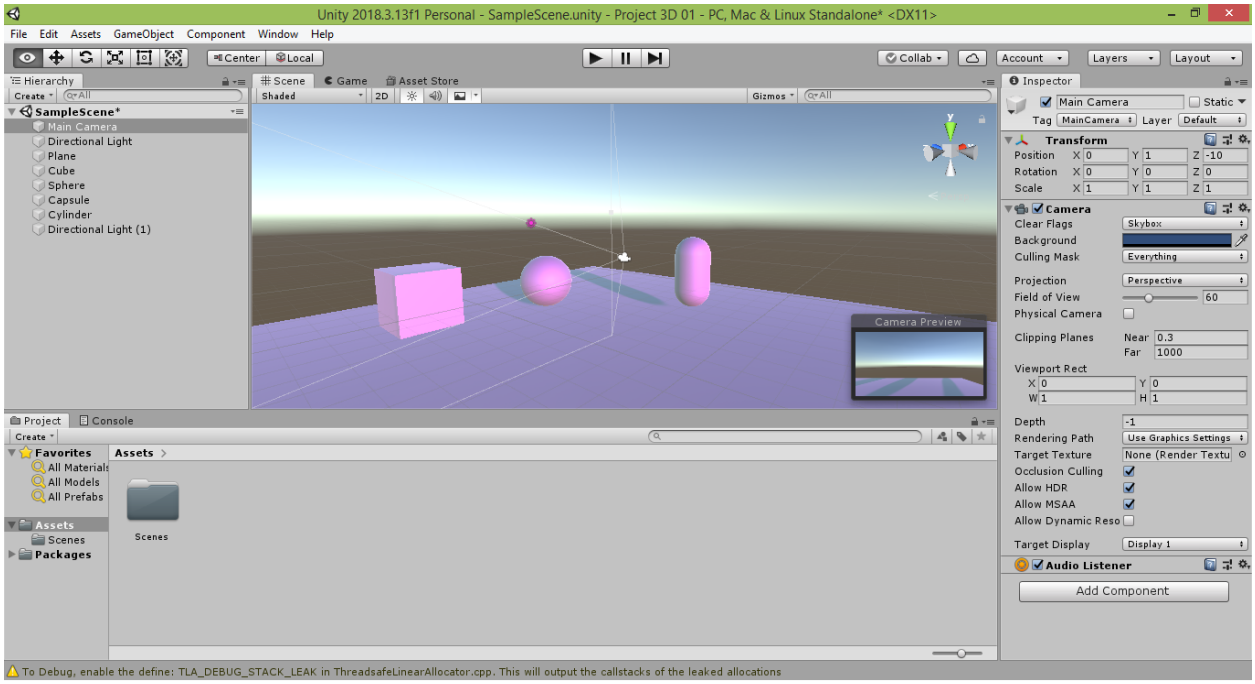


2.4. Размещение и настройка камер

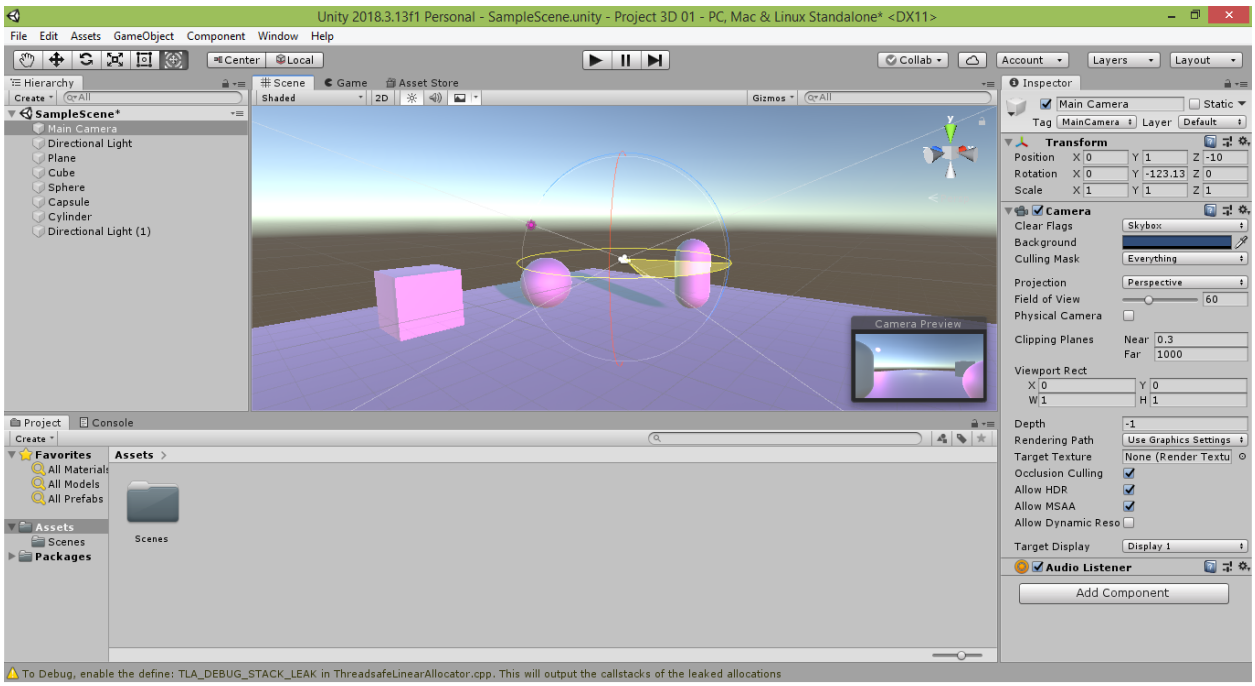
Теперь, когда мы разместили объекты на сцене и настроили их освещение, попробуем отрегулировать камеры, чтобы лучше видеть результат наших действий.

Щёлкните мышкой на значок камеры на сцене или на соответствующую ему строчку «Main Camera» («Основная камера») в окне «Hierarchy».

Вы увидите в правом нижнем углу сцены экран «Camera Preview» («Вид с камеры»), на который транслируется изображение сцены, получаемое с камеры. При этом область охвата сцены камерой схематично показана белыми линиями, исходящими из значка камеры.

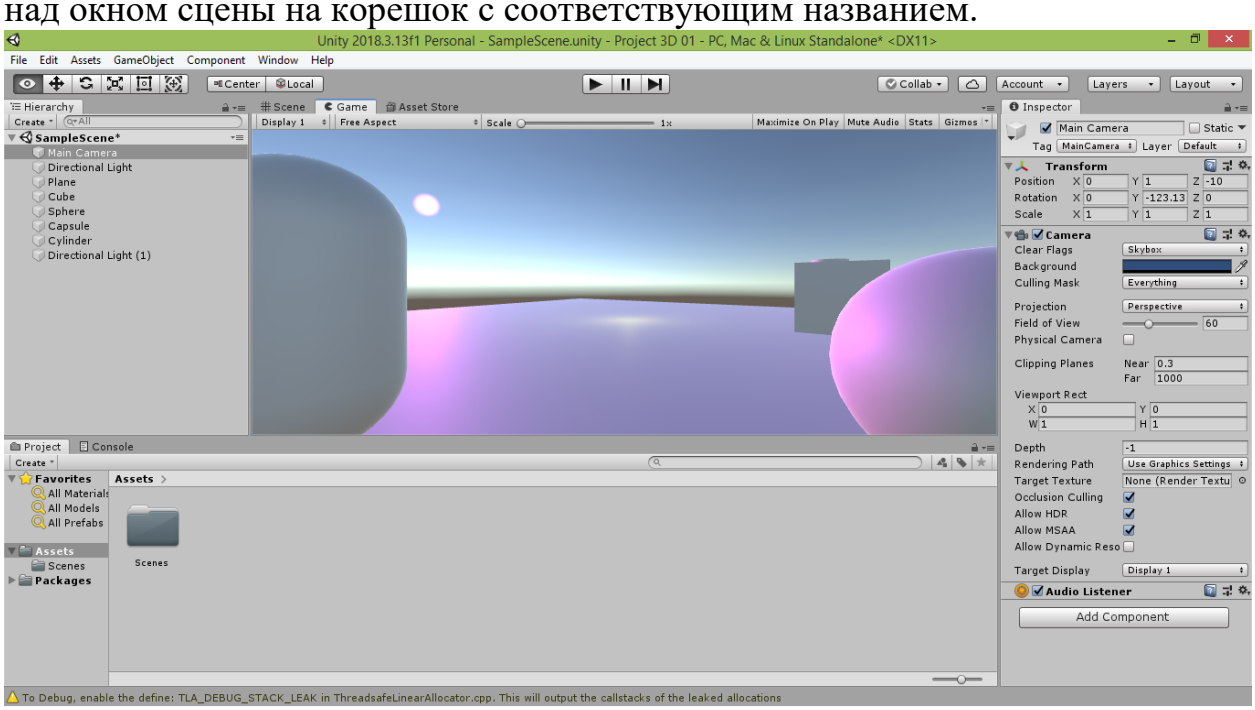


Чтобы настроить удобный для вас ракурс камеры, нажмите на кнопку «Rotate Tool».



Разверните камеру, потянув за разноцветные окружности вращения. В процессе изменения углов поворота камеры будет плавно меняться и вид в окне «Camera Preview».

Если хотите посмотреть, как пользователь будет видеть вашу сцену в процессе её проигрывания, перейдите на вкладку «Game» («Игра»), щёлкнув над окном сцены на корешок с соответствующим названием.

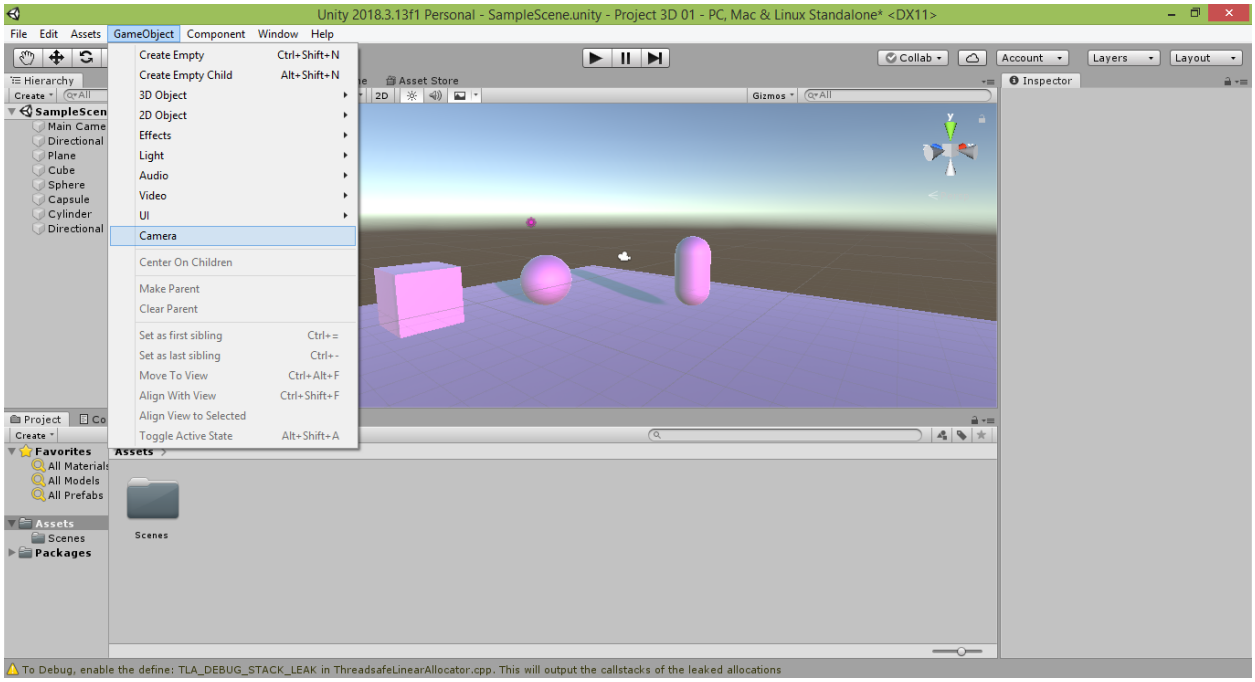


Вы увидите то же самое изображение, что транслировалось в окне «Camera Preview», но в полноэкранном формате.

Нажмите на корешок вкладки «Scene» («Сцена»), чтобы вернуться в окно редактирования сцены.

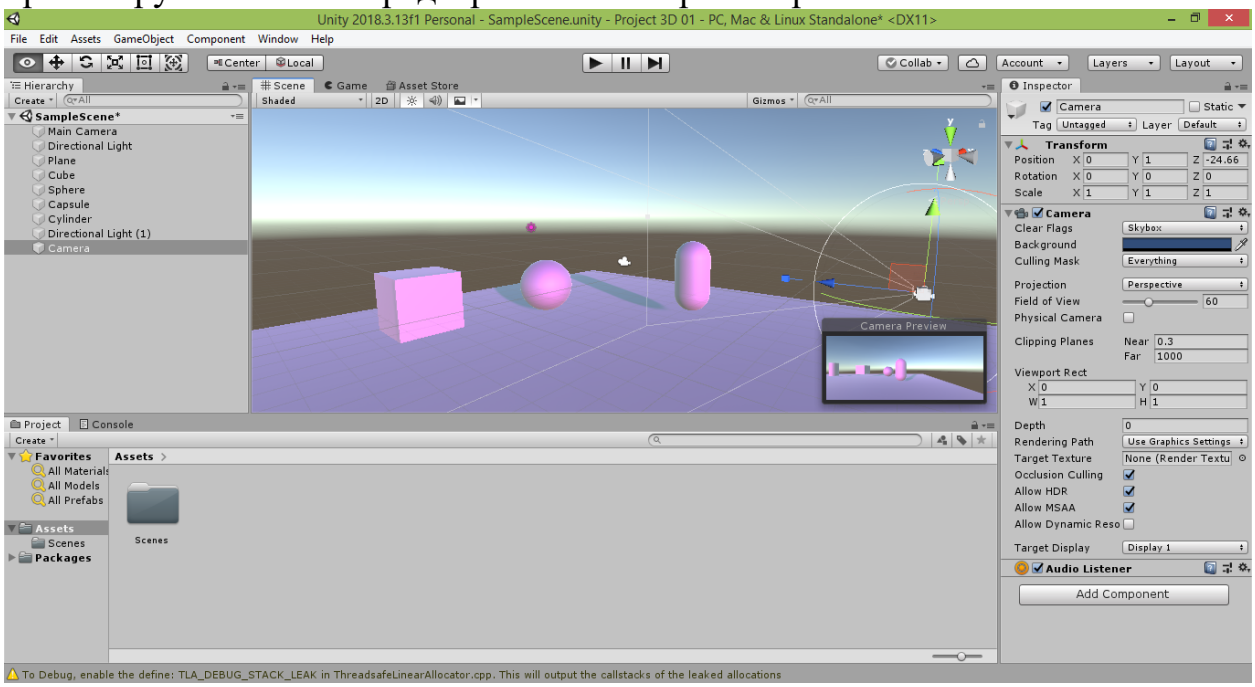
Добавим на сцену дополнительную камеру.

Для этого выберите в меню Unity команду «GameObject → Camera».



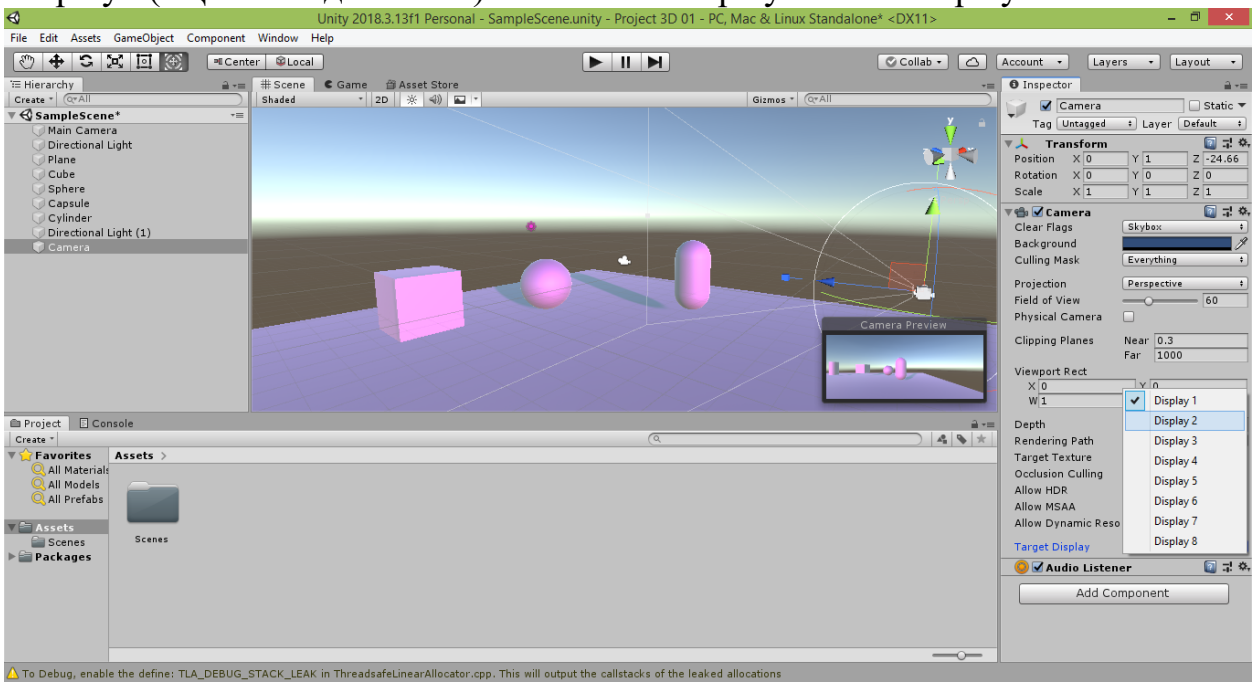
В результате на вашей сцене появится новый игровой объект «Camera» (в переводе с английского означает «Камера»).

Если новая камера появится в той же точке, что и основная камера, переместите её в другую точку сцены и разверните в нужном для вас ракурсе, ориентируясь на окно предварительного просмотра «Camera Preview».



Однако при переходе на вкладку «Game» будет доступен для просмотра только вид с дополнительной камеры, в то время как вид с основной камеры исчезнет. Это происходит поскольку две камеры не могут одновременно отображать картинку на одном дисплее. Чтобы переключаться между видами с камер, следует назначить каждой камере отдельный дисплей. Для этого

измените у дополнительной камеры в окне «Inspector» в свойстве «Target Display» («Целевой дисплей») значение «Display 1» на «Display 2».

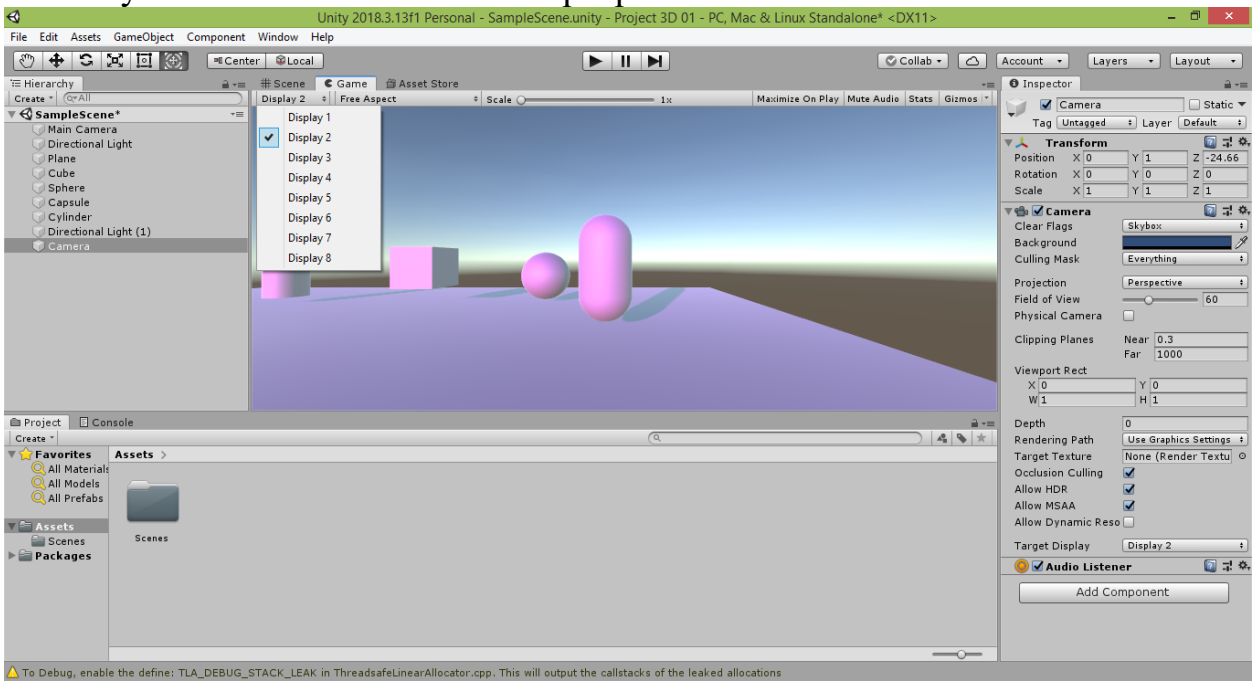


Снова перейдите на вкладку «Game». Вы увидите вид с основной камеры, поскольку у неё в свойстве «Target Display» осталось значение «Display 1», устанавливаемое по умолчанию при создании любой камеры.

Откройте выпадающий список в левом верхнем углу окна вкладки «Game» и выберите вместо значения «Display 1» значение «Display 2».

Произойдёт переключение вида игры на второй дисплей, и вы увидите изображение сцены, транслируемое на второй дисплей с дополнительной камеры.

В системе Unity доступно 8 дисплеев. Таким образом, вручную вы можете настроить до 8 различных видов вашей сцены. При необходимости можно увеличить это количество программным способом.



Для сохранения выполненного вами проекта выберите в меню программы Unity пункт «File → Save» или нажмите комбинацию клавиш CTRL+S. Даже если вы забудете это сделать, Unity при закрытии напомнит вам, что в проект были внесены изменения, и спросит, следует ли их сохранить. Если вы согласны на фиксацию изменений, произведённых с проектом, нажмите «Save» («Сохранить»), в противном случае – откажитесь, нажав «Don't save» («Не сохранять»).

Важное замечание. Рекомендуется не ждать закрытия программы, а периодически производить сохранения промежуточных состояний вашего проекта, чтобы не потерять их в случае возникновения внештатной ситуации (например, внезапного выключения электричества).

P.S.: В процессе длительного написания данного руководства у автора произошло «зависание» Unity (видимо, вследствие большой нагрузки на оперативную память сразу нескольких приложений – Unity, Visual Studio, Word и других). В результате пришлось экстренно завершить работу Unity через Диспетчер задач, после чего проект при открытии оказался пустым – в нём остались только файлы скриптов, а несохранённая сцена, которую вы видели на иллюстрациях, была полностью утеряна.

Вывод: как можно чаще сохраняйте свои работы!

Задание для самостоятельной работы

1. Измените форму и размер куба на параллелепипед, растянув его по одной или нескольким осям.

2. Измените форму и размер сферы на эллипсоид, растянув её по одной или нескольким осям.

3. Измените форму и размер цилиндра, растянув его по одной или нескольким осям.

4. Измените форму и размер капсулы, растянув её по одной или нескольким осям.

5. Увеличьте размер и расположение площадки, чтобы на ней были видны тени вышеуказанных объектов.

6. Разместите объекты на сцене так, чтобы при обзоре сцены с камеры они не перекрывали друг друга.

7. Добавьте третий источник света типа «Spot». В результате у вас на сцене должны быть размещены источники света трёх типов: «Directional», «Spot» и «Point».

8. Настройте цвет, интенсивность, угол и дальность действия всех источников света, чтобы был виден вклад каждого из них в освещение сцены.

9. Добавьте на сцену третью камеру и разместите все три камеры так, чтобы каждая из них наглядно демонстрировала действие одного из трёх источников света. Каждая камера должна отображать сцену на отдельном дисплее. Если потребуется, можете добавить на сцену больше трёх камер.

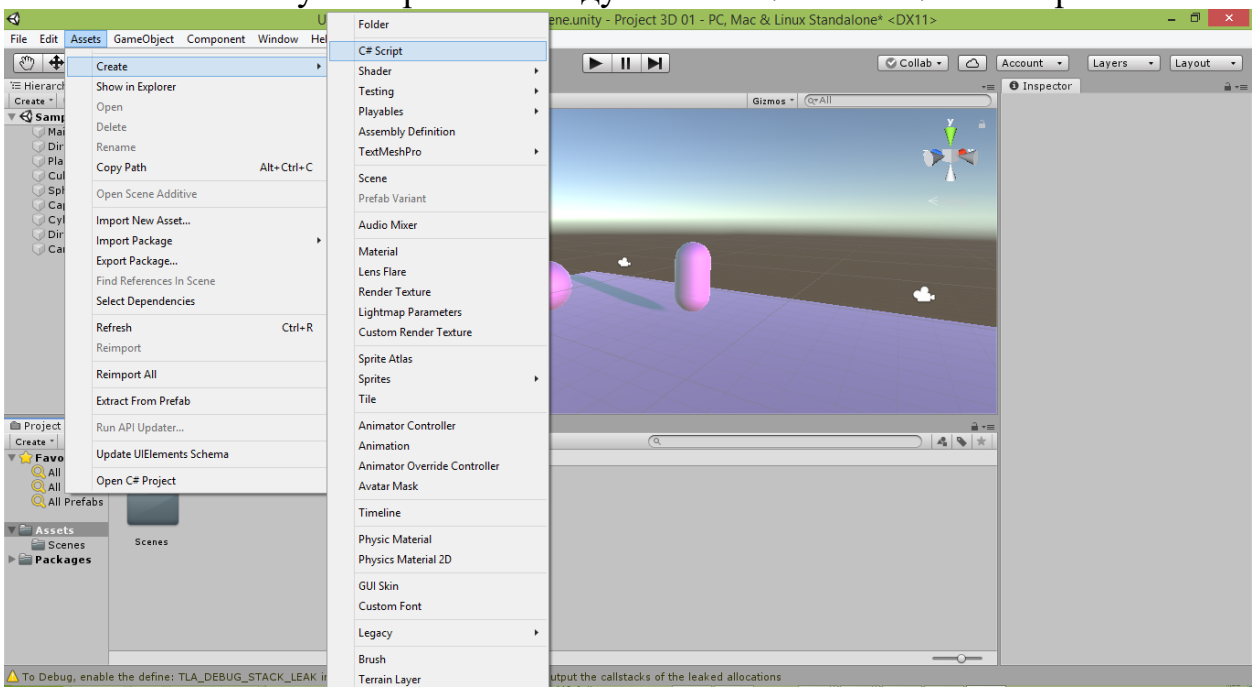
10. Скопируйте в отдельные файлы результаты вашей работы, сделав скриншот сцены (вкладка «Scene») и скриншот вида с каждой из камер (вкладка «Game»). Итого должно получиться 4 скриншота (их может быть и больше, если вы добавляли больше трёх камер).

Результаты вашей работы рекомендуется показать преподавателю и обсудить с ним.

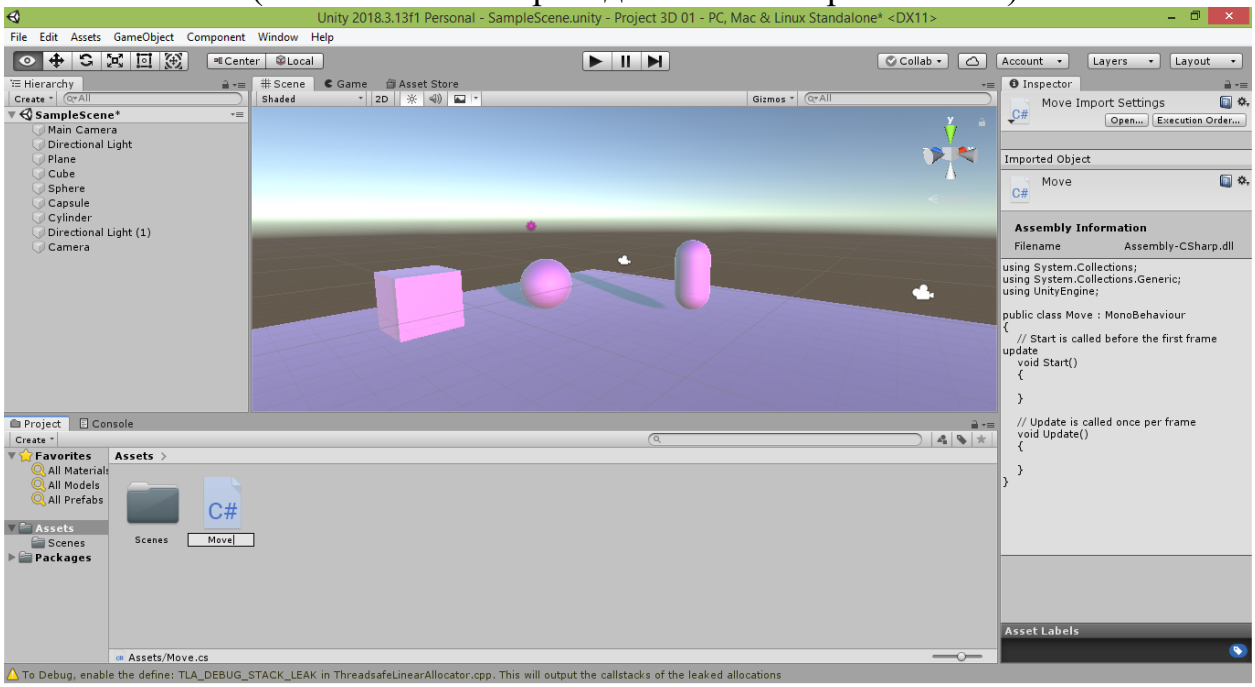
Глава 3. Программное изменение объектов

3.1. Создание скрипта (программного сценария)

В меню Unity выберите команду «Assets → Create → C# Script».

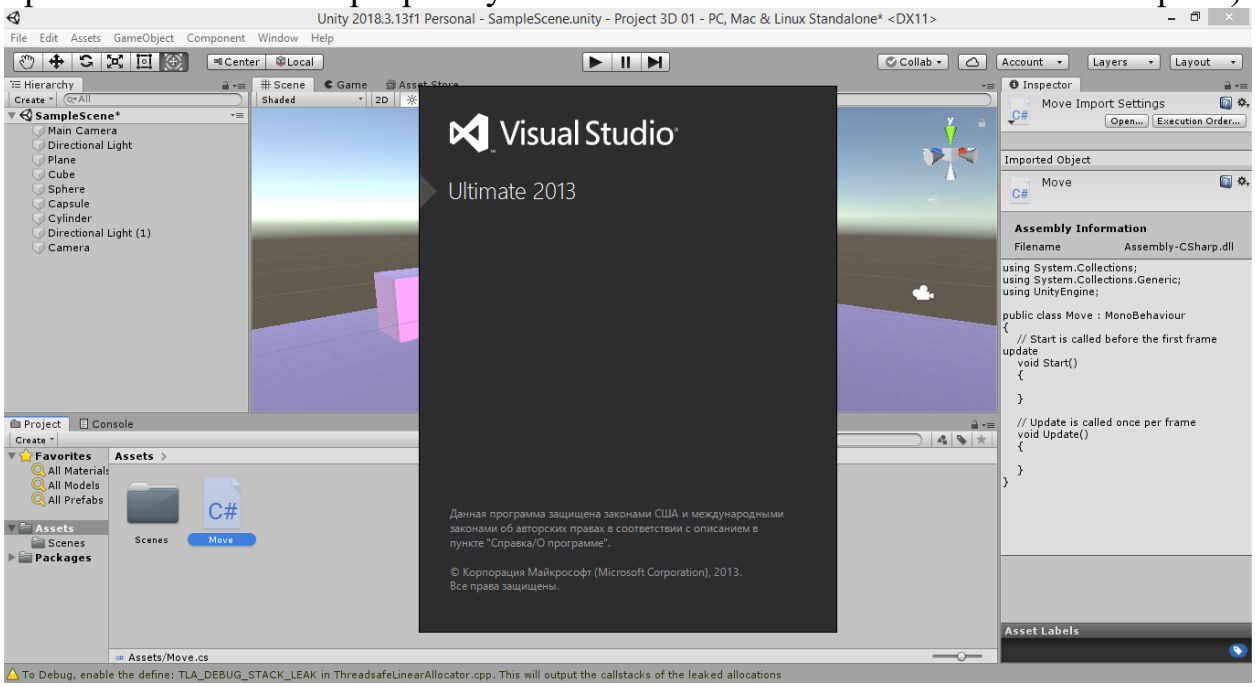


Внизу появится значок файла скрипта. Созданный файл скрипта назовите Move (с английского переводится как «Переместить»).

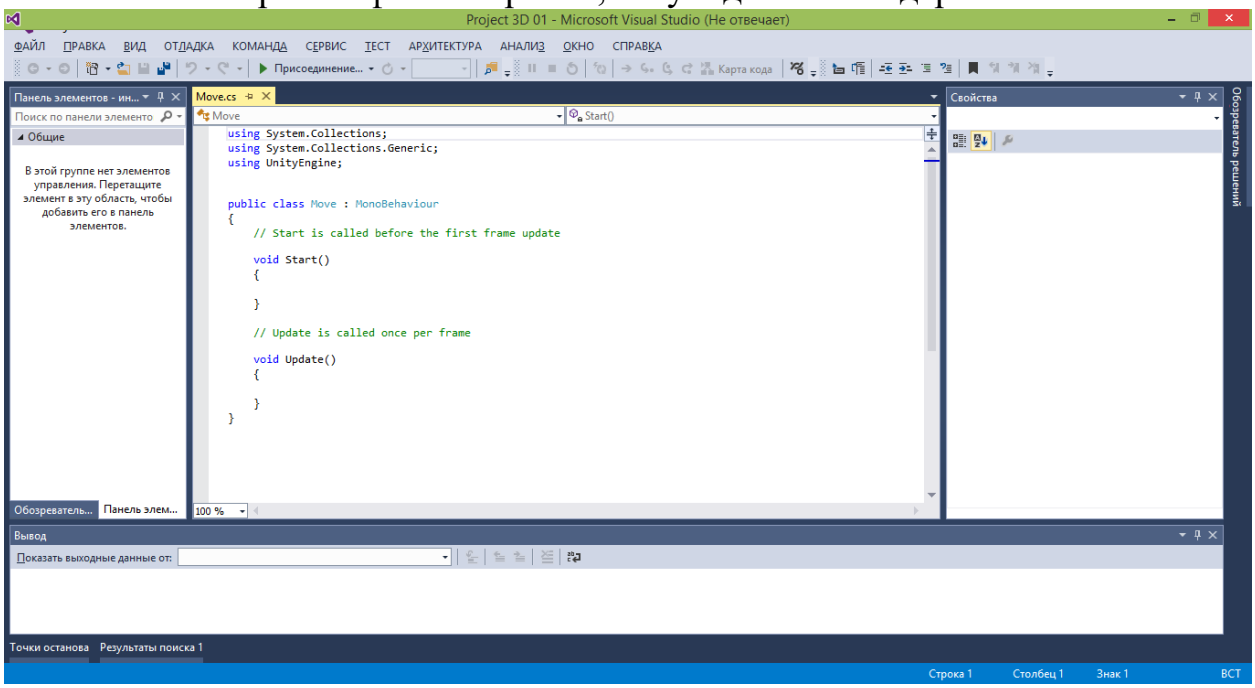


Закончив переименование, щёлкните два раза на значке файла скрипта или нажмите клавишу «Enter». Запустится редактор Visual Studio (если вдруг

он у вас не установлен, выберите в открывшемся списке доступных приложений программу «Блокнот» / «Notepad»).



После открытия файла скрипта, вы увидите его содержимое.



По умолчанию содержимое нового скрипта выглядит следующим образом:

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

public class Move : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
```

```
        {

        }

        // Update is called once per frame
        void Update()
        {

        }
    }
```

Первые строки, начинающиеся со слова using, менять не следует – они отвечают за подключение различных возможностей (в частности, модуль UnityEngine отвечает за возможность управления объектами сцены в Unity).

Строка

```
public class Move : MonoBehaviour
```

описывает класс – программный шаблон.

В случае с Unity, в классах категории `MonoBehaviour` («однотипное поведение») описывается шаблон сценария поведения любого объекта, с которым мы свяжем наш скрипт.

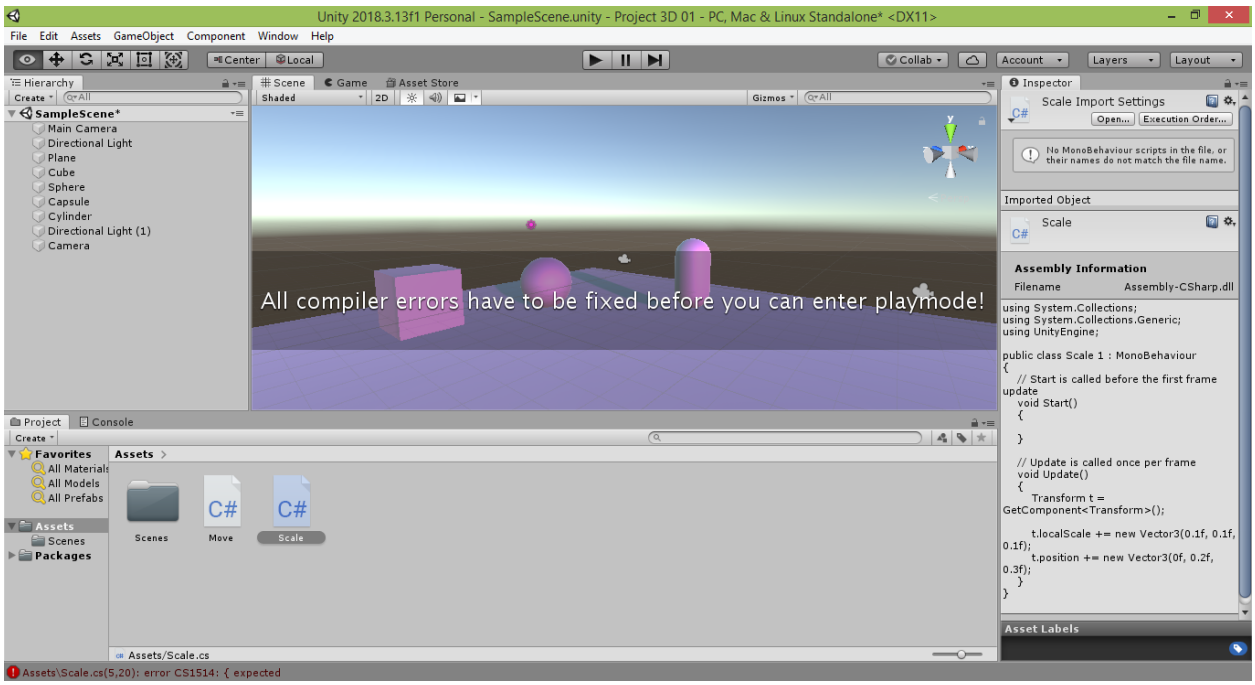
Важное замечание. Обращайте внимание на названия, которые вы даёте своим скриптам. Основные правила:

1. Название всегда должно начинаться с буквы
2. В названии не должно быть пробелов.
3. В названии допускается использовать цифры и знак подчёркивания (вместо пробела)
4. В названии лучше не использовать русские или другие неанглоязычные буквы.
5. Название скрипта и класса должны совпадать (в нашем примере используется `Move`)
6. Большие и маленькие буквы считаются разными (если назвать наш класс `move`, то возникнет ошибка, поскольку файл скрипта называется `Move`, и для её устранения следует либо переименовать файл скрипта в `move`, либо переименовать класс обратно в `Move`).

Если одно из указанных правил было нарушено, то, скорее всего, после создания скрипта ваш Unity-проект не запустится и выдаст сообщение об ошибке:

«All compiler errors have to be fixed before you can enter playmode!»

(«Все ошибки в коде должны быть устранены перед запуском игры!»)



Далее в теле класса скрипта идёт описание двух подпрограмм.

1. `void Start()`

Данная подпрограмма сопровождается комментарием (эта зелёная строка не влияет на работу скрипта, её можно вообще удалить):

```
// Start is called before the first frame update
// Start вызывается перед обновлением первого кадра
```

Таким образом, в начале работы вашего Unity-проекта произойдёт срабатывание кода, описанного в данной подпрограмме. В этом плане `void Start()` аналогична подпрограмме `void setup()` в скрипте Arduino.

2. `void Update()`

Данная подпрограмма сопровождается комментарием (эта зелёная строка также не влияет на работу скрипта и может быть удалена):

```
// Update is called once per frame
// Update вызывается в каждом кадре
```

Из описания следует, что код, прописанный в данной подпрограмме, срабатывает каждый раз, когда обновляется кадр сцены на экране вашего дисплея. В этом плане `void Update()` схожа с подпрограммой `void loop()` в скрипте Arduino. Отличие заключается в том, что скорость изменений сцены будет зависеть от настроек видеокарты компьютера. Допустим, ваша видеокарта обновляет изображение на экране 120 раз в секунду. Если вы в настройках видеокарты понизите обновление изображения до 60 кадров в секунду, то ваша сцена (например, 3D-видеоролик) будет меняться в 2 раза медленнее. Однако пока мы не будем обращать внимание на такие детали и начнём изучение основ программирования в Unity.

3.2. Изменение местоположения объекта

Первым делом научимся перемещать объекты. Для этого напишите в подпрограмме (чаще её называют методом) `Update()` следующие три строки кода:

```
void Update()
{
    Transform t;

    t = GetComponent<Transform>();

    t.position = t.position + new Vector3(0.1f, 0f, 0f);
}
```

1. В первой строке данного кода создаётся переменная (ячейка памяти), имеющая тип `Transform` и предназначенная для хранения инструментов изменения (трансформации) любого объекта. Мы назвали нашу переменную `t` (название может быть любым, но должно начинаться с буквы и далее может содержать буквы, цифры и знаки подчёркивания). Переменные выполняют в памяти роль временных хранилищ подобно тому, как ящички в тумбочке нужны для временного хранения предметов.

2. Теперь мы должны получить и положить в наш ящичек с названием `t` набор инструментов для трансформации объекта. Для этого мы во второй строке вызываем метод (команду)

```
GetComponent<Transform>()
```

который выдаёт нам компонент, представляющий набор инструментов, описываемых классом `Transform` и предназначенных для трансформации объектов.

Осталось положить полученный набор инструментов в ящик `t`, чтобы можно было в любой момент взять их оттуда. Для этого мы записываем (присваиваем) ячейке памяти `t` значение, выданное командой `GetComponent`:

```
t = GetComponent<Transform>();
```

3. В третьей строке происходит самое интересное. Теперь, когда мы имеем набор инструментов для изменения объекта, в переменной `t`, мы обращаемся к ней и говорим, что хотим задать новое значение местоположению нашего объекта (свойство `position`). Для этого мы запрашиваем текущее местоположение объекта `t.position` и прибавляем к нему вектор смещения в 3D-пространстве. Создать новый такой вектор можно командой

```
new Vector3(0.1f, 0f, 0f)
```

Здесь `Vector3` – это название класса, описывающего набор из трёх чисел. Первое число (координата x), задаёт изменение по красной оси (длина), второе число (координата y) задаёт изменение по зелёной оси (высота), третье

число (координата z) задаёт изменение по синей оси (ширина или глубина). В рассматриваемом нами примере изменение будет заключаться в перемещении объекта вдоль этих осей.

Элементы, составляющие вектор, являются дробными числами и потому имеют тип **float** (переводится как «плавающий», что означает число с плавающей точкой – десятичную дробь). В языках C++, C# и других родственных для хранения дробного значения по умолчанию используется тип **double** (дробное число двойной точности), под которое выделяется больше памяти для хранения, чем под значение типа **float**.

Поэтому такая запись нашего вектора выдаст ошибку:

```
new Vector3(0.1, 0, 0)
```

Дело в том, что каждое из трёх заданных нами значений по умолчанию будет иметь тип **double** и не поместится в ячейку памяти, отведённую под значение типа **float**. Поэтому мы вынуждены указывать после десятичных букву «f», тем самым говоря программе, что хотим подрезать наши десятичные двойной точности до размера значений типа **float**:

```
new Vector3(0.1f, 0f, 0f)
```

На самом деле, для числа 0 не обязательно дописывать букву «f» в конце, но лучше это делать, чтобы не забыть про неё, когда будете менять 0 на какое-либо другое значение.

Теперь вернёмся к нашей последней строке:

```
t.position = t.position + new Vector3(0.1f, 0f, 0f);
```

Справа от знака «=» мы запросили текущее местоположение объекта и добавили к нему вектор-смещение. В результате местоположение объекта должно измениться вдоль красной оси на одну десятую шага (поскольку значение x-координаты вектора равно 0.1f) и не изменится вдоль зелёной и синей осей (значения смещения равны `0f`). Чтобы объект получил новое (рассчитанное) местоположение мы должны записать его вместо старого. Для этого мы и используем операцию присвоения (задания) нового значения свойству:

```
t.position =
```

На самом деле, последнюю строку кода можно было записать короче, используя оператор «+=» для увеличения свойства `t.position` на вектор:

```
t.position += new Vector3(0.1f, 0f, 0f);
```

Точно так же можно было превратить первые две строки в одну, сохранив полученный набор инструментов в переменную `t` сразу после её создания:

```
Transform t = GetComponent<Transform>();
```

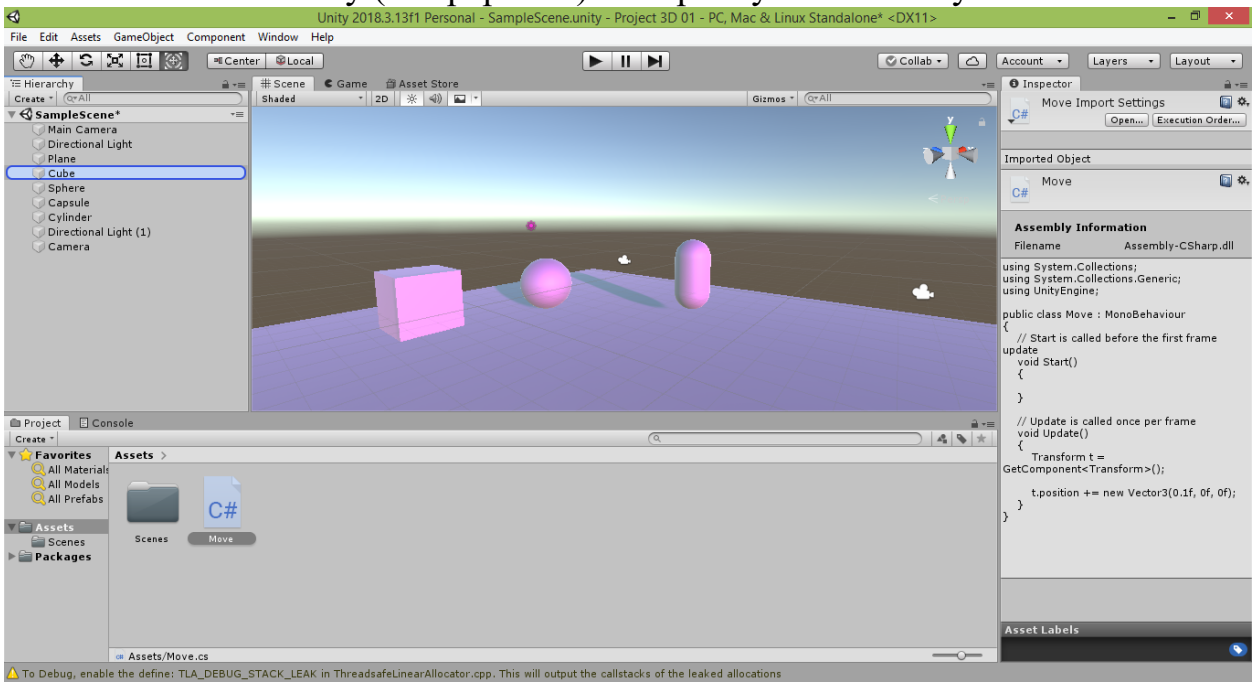
В результате наш программа приобрела более компактный вид:

```
void Update()
{
    Transform t = GetComponent<Transform>();
```

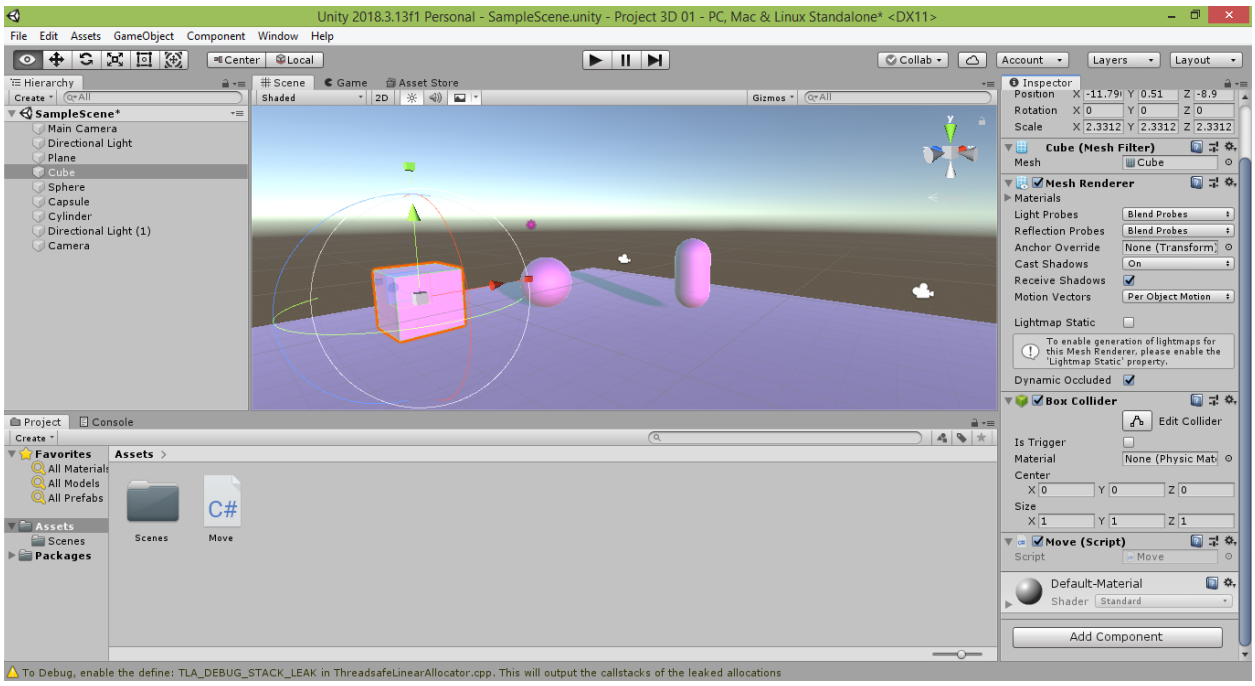
```
        t.position += new Vector3(0.1f, 0f, 0f);
    }
```

Теперь осталось сохранить код. Для этого нажмите вверху значок квадратной синей дискеты или выберите в меню команду «Файл → Сохранить Move.cs» («File → Save Move.cs» в английской версии Visual Studio). Также для сохранения можно использовать комбинацию клавиш CTRL+S.

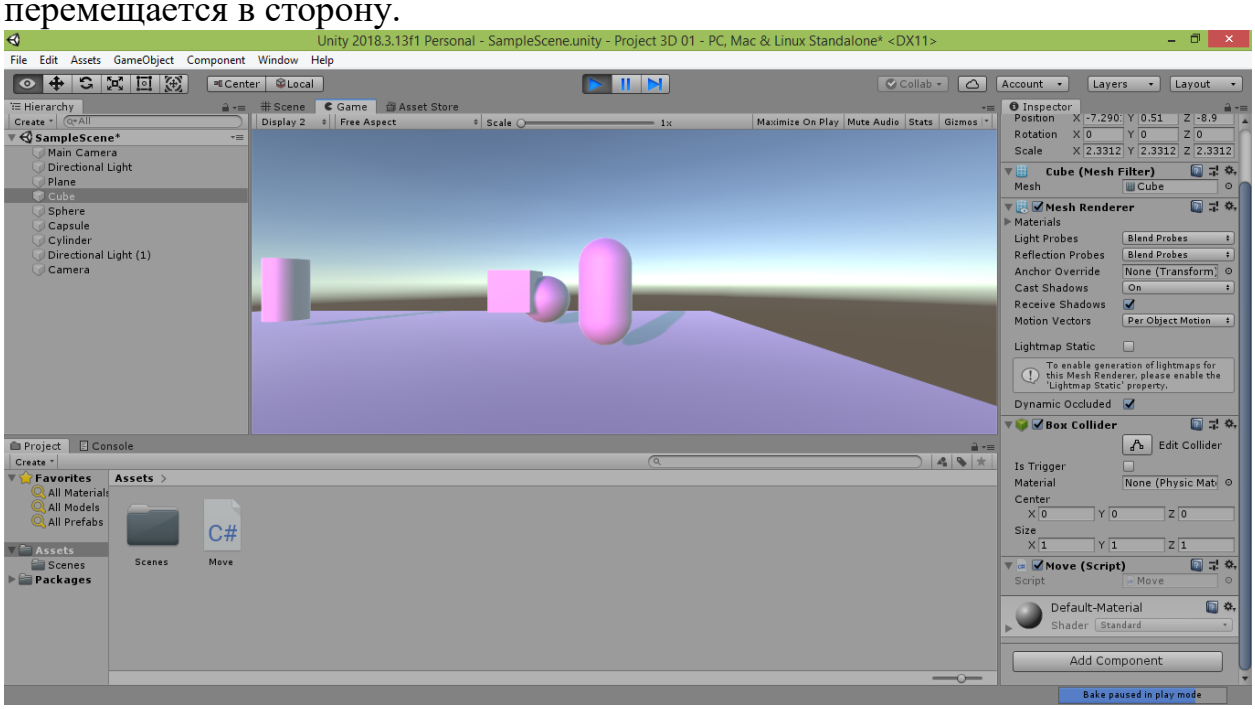
Пока не закрывайте окно редактора Visual Studio (мы ещё много раз будем к нему обращаться). Перейдите обратно в окно Unity и убедитесь, что написанный вам код отображается справа в окне Inspector («Инспектор»), когда вы выделяете файл скрипта Move. После этого перетяните файл скрипта влево в окно Hierarchy («Иерархия») на строчку Cube и отпустите.



Теперь если выделить строку Cube, то вы увидите в окне Inspector среди свойств созданного нами куба пункт Move (Script). Это означает, что наш скрипт был привязан к кубу.



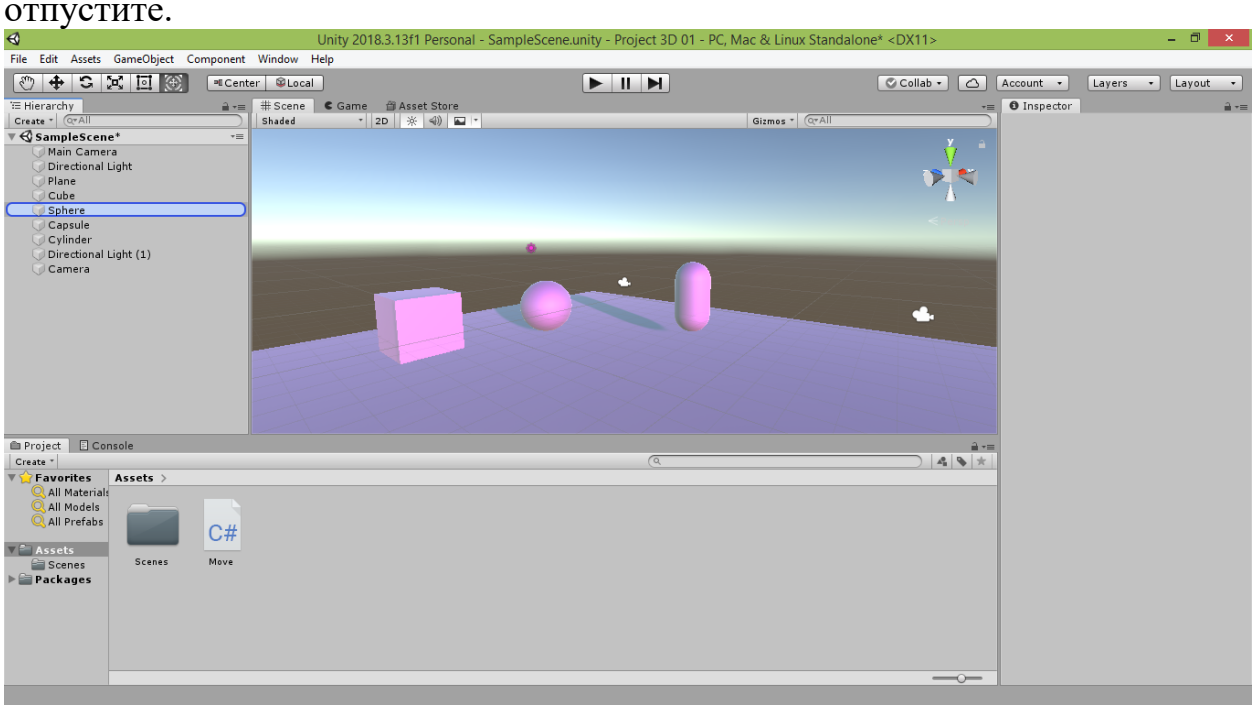
Теперь можно запустить наш проект в игровом режиме, нажав в самом верху кнопку «Play» со значком чёрного треугольника, указывающего вправо. Если в поле зрения вашей камеры попадает куб, вы увидите, как он плавно перемещается в сторону.



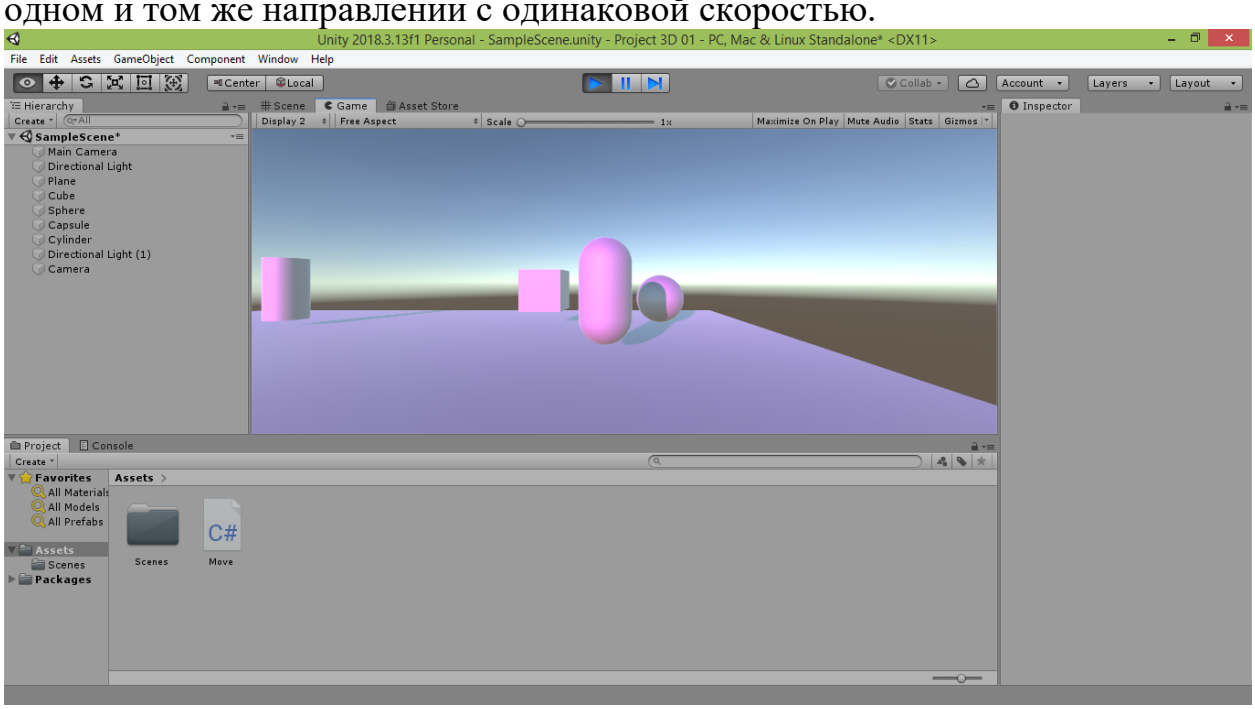
Нажмите ещё раз кнопку «Play», чтобы остановить проигрывание сцены.

Важное замечание. Не забывайте останавливать проигрывание, иначе все сделанные вами далее изменения сцены исчезнут после выхода из режима проигрывания. То есть настройки сцены можно менять во время режима проигрывания, но в этом случае все они считаются временными (пробными, что удобно при тестировании игр).

Большим преимуществом скриптов является то, что один и тот же скрипт можно привязывать сразу к нескольким объектам. Пока не закрывайте окно редактора Visual Studio (мы ещё много раз будем к нему обращаться). Перетяните файл скрипта Move влево в окно Hierarchy на строчку Sphere и отпустите.



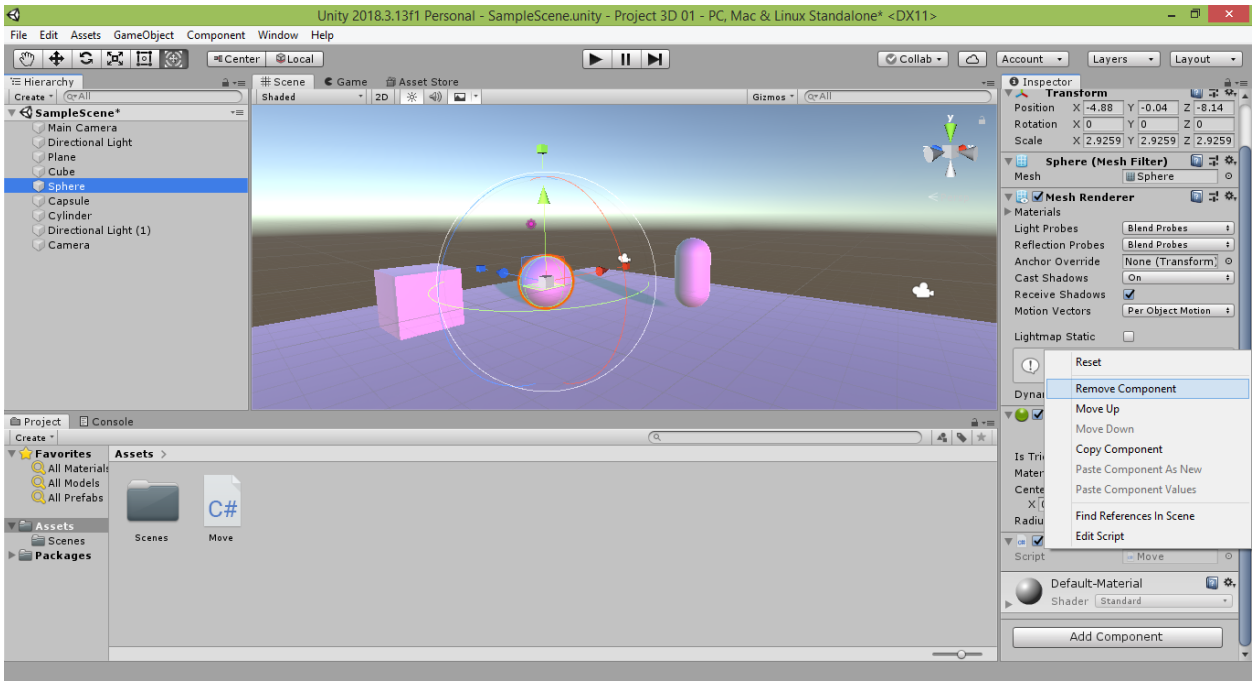
Теперь если выделить строку Sphere, то вы также увидите в окне Inspector среди свойств созданного нами шара пункт Move (Script). Это означает, что наш скрипт был привязан не только к кубу, но и к шару. Снова запустите проект в игровом режиме, нажав в самом верху кнопку «Play». Теперь куб и шар, управляемые одним и тем же скриптом, двигаются вместе в одном и том же направлении с одинаковой скоростью.



Нажмите ещё раз кнопку «Play», чтобы остановить проигрывание сцены.

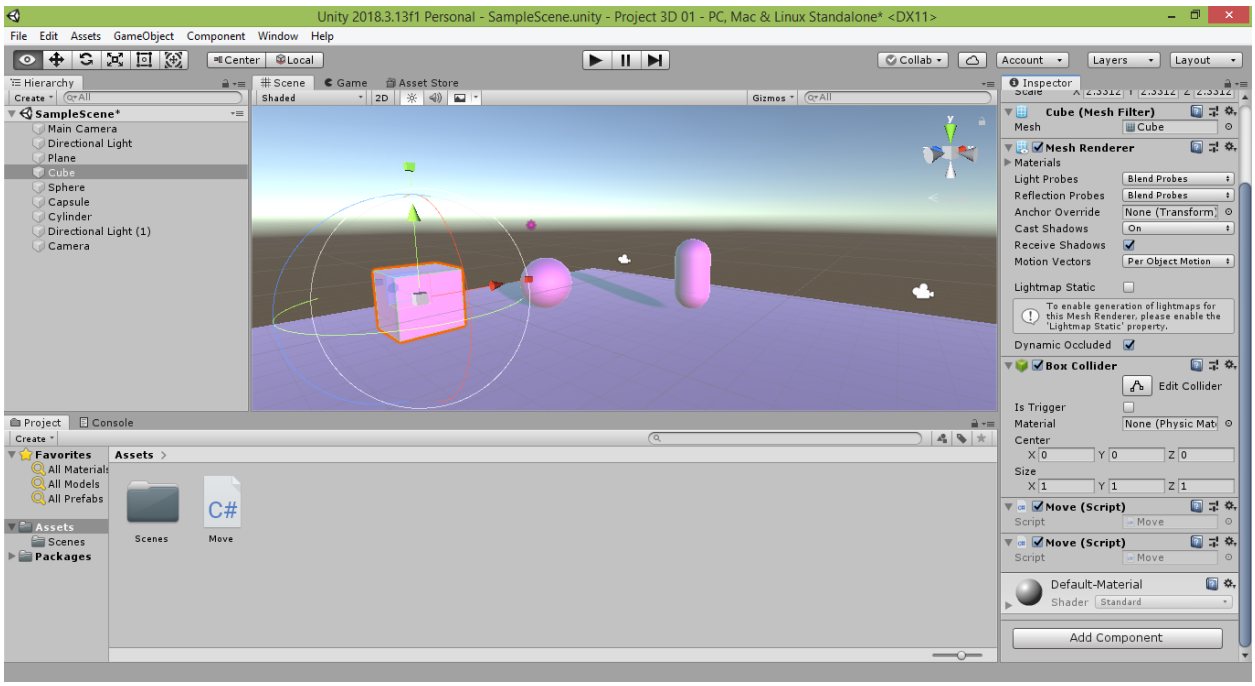
Если вы захотите выключить движение сферы, можно удалить привязку скрипта к ней. Для этого следует щёлкнуть на значке шестерни в правом верхнем углу пункта Move (Script) и выбрать пункт «Remove Component». В этом случае строка Move (Script) исчезнет из списка свойств объекта в окне Inspector.

Если же вы не хотите удалять привязку скрипта к объекту, можно просто снять галочку слева от пункта Move (Script) – скрипт деактивируется для этого объекта.

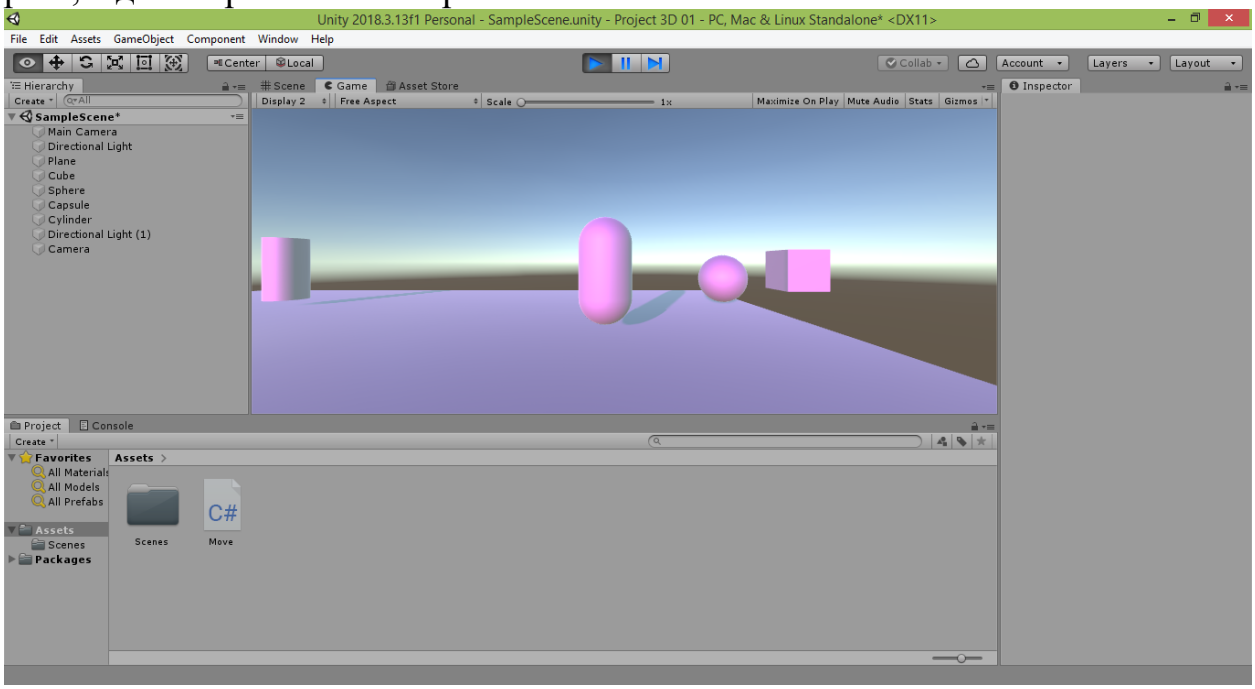


Также одни и тот же скрипт можно несколько раз привязывать к одному и тому же объекту. В этом случае его программа будет повторяться за один кадр столько раз, сколько привязок данного скрипта было сделано к объекту.

Воспользуемся этой возможностью для ускорения движения куба. Снова перетяните файл скрипта влево в окно Hierarchy на строчку Cube и отпустите. Выделите строку Cube и убедитесь, что в окне Inspector среди свойств куба теперь присутствуют два пункта Move (Script). Это означает, что к нашему кубу привязаны два экземпляра одного и того же скрипта.



Запустите проект в игровом режиме, нажав в самом верху кнопку «Play». Теперь куб двигается в 2 раза быстрее и обгоняет шар, поскольку для куба за один кадр написанная нами программа перемещения объекта срабатывает 2 раза, а для шара – только 1 раз.



Нажмите ещё раз кнопку «Play», чтобы остановить проигрывание сцены.

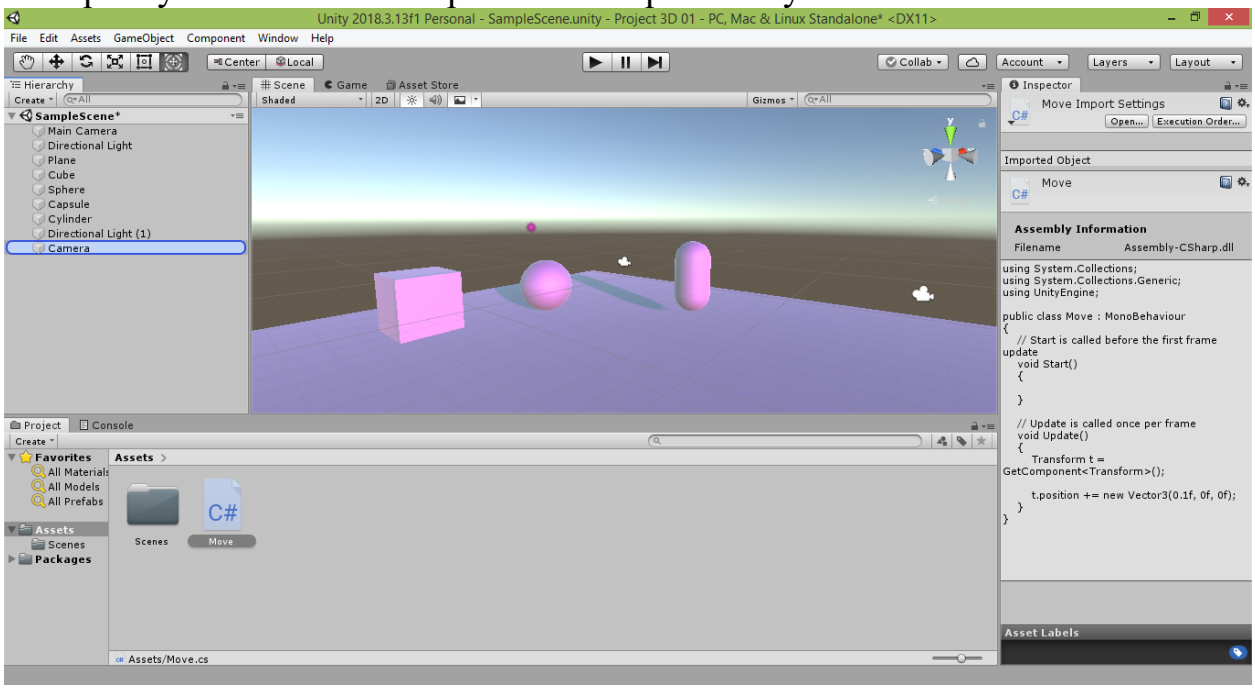
Если вы захотите ещё больше ускорить движение куба, можете снова перетянуть скрипт на строку Cube в окне Hierarchy столько раз, сколько потребуется.

При необходимости, можно убрать привязку лишних экземпляров скрипта к кубу, используя команду «Remove Component» в настройках

экземпляра скрипта, или деактивировать эти экземпляры, сняв галочку рядом с их пунктом.

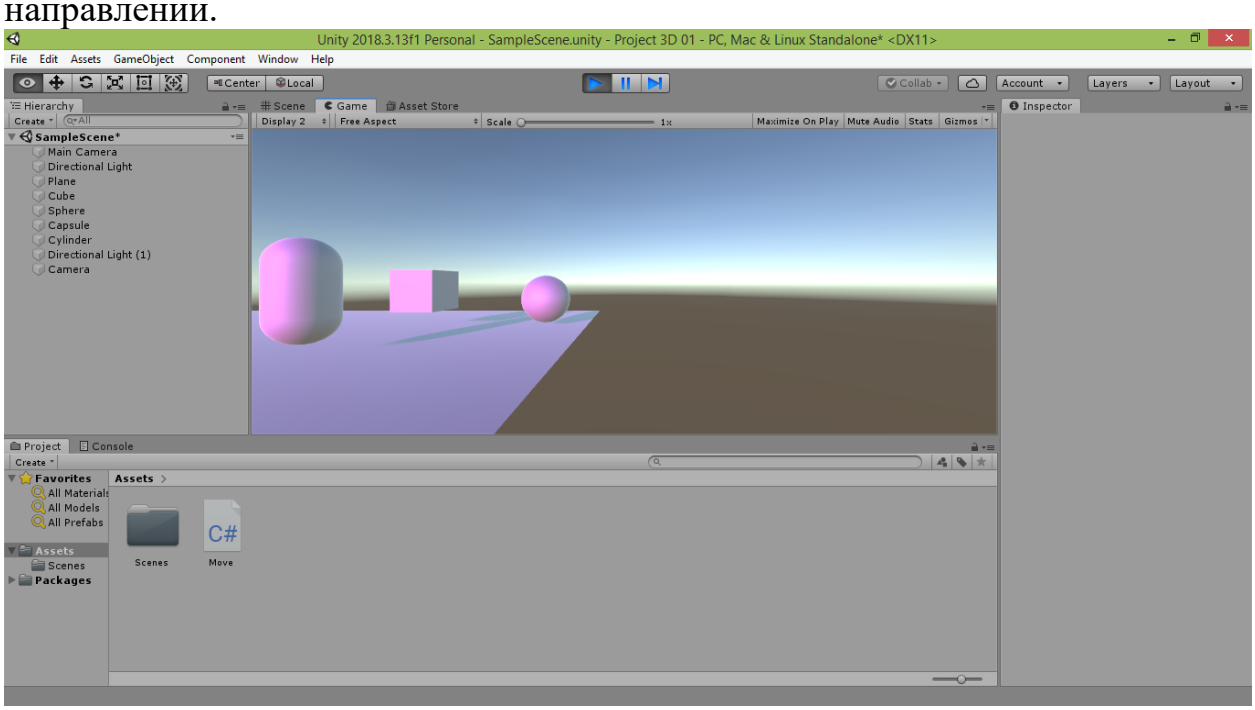
Следует отметить, что скрипты могут быть привязаны и к камерам. Это позволяет создать эффект перемещения по сцене.

Выберите камеру, перетяните файл скрипта Move влево в окно Hierarchy на строчку с названием выбранной камеры и отпустите.



Выделите строку с названием камеры и убедитесь, что в окне Inspector среди свойств камеры присутствует пункт Move (Script). Это означает, что наш скрипт был привязан к выбранной камере.

Запустите проект в игровом режиме, нажав в самом верху кнопку «Play». Теперь камера следует по сцене за кубом и сферой в одном с ними направлении.



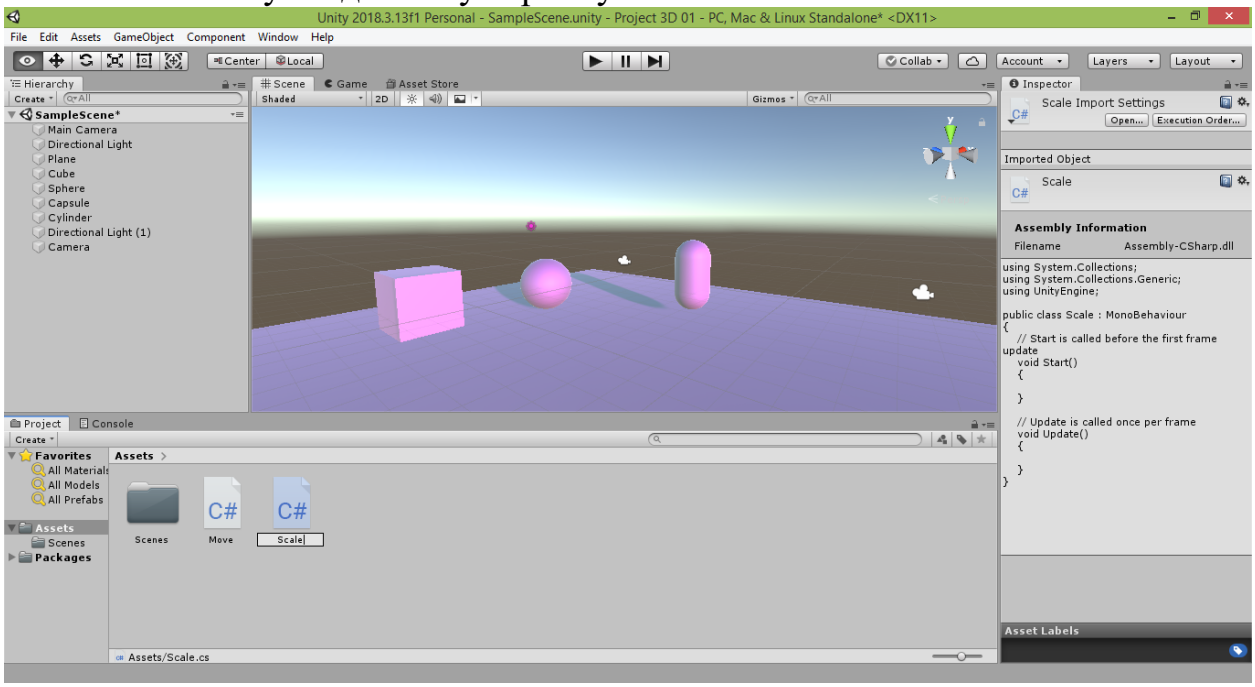
Нажмите ещё раз кнопку «Play», чтобы остановить проигрывание сцены.

Удалите или деактивируйте привязку скрипта к камере, чтобы её движение не мешало дальнейшему просмотру сцены при дальнейшей работе с объектами.

Аналогичным способом можно управлять движением источников света, создавая на поверхности объектов интересные эффекты, состоящие из разноцветных переливов или движущихся бликов (например, света из окон вагонов проезжающего мимо поезда). Если вас заинтересовала такая возможность, можете самостоятельно создать несколько источников света, различающихся по типу, цвету и интенсивности освещения (можете также настроить и другие их свойства). Расположите источники света в разных местах сцены. Далее создайте один или несколько скриптов с различными направлениями и скоростью движения, а затем попробуйте различные комбинации связок этих скриптов и источников света. Полученные эффекты могут быть достаточно интересными и неожиданными.

3.3. Изменение масштаба объекта

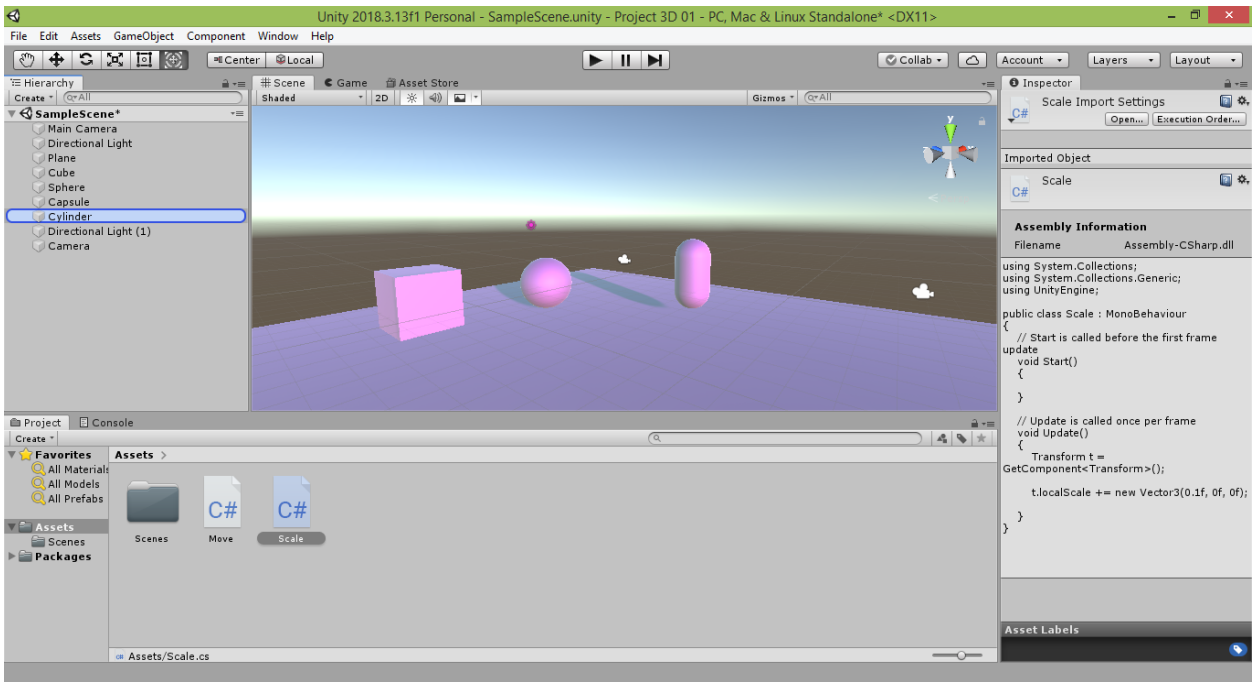
Выберем в меню Unity команду «Assets → Create → C# Script» и дадим имя Scale новому созданному скрипту.



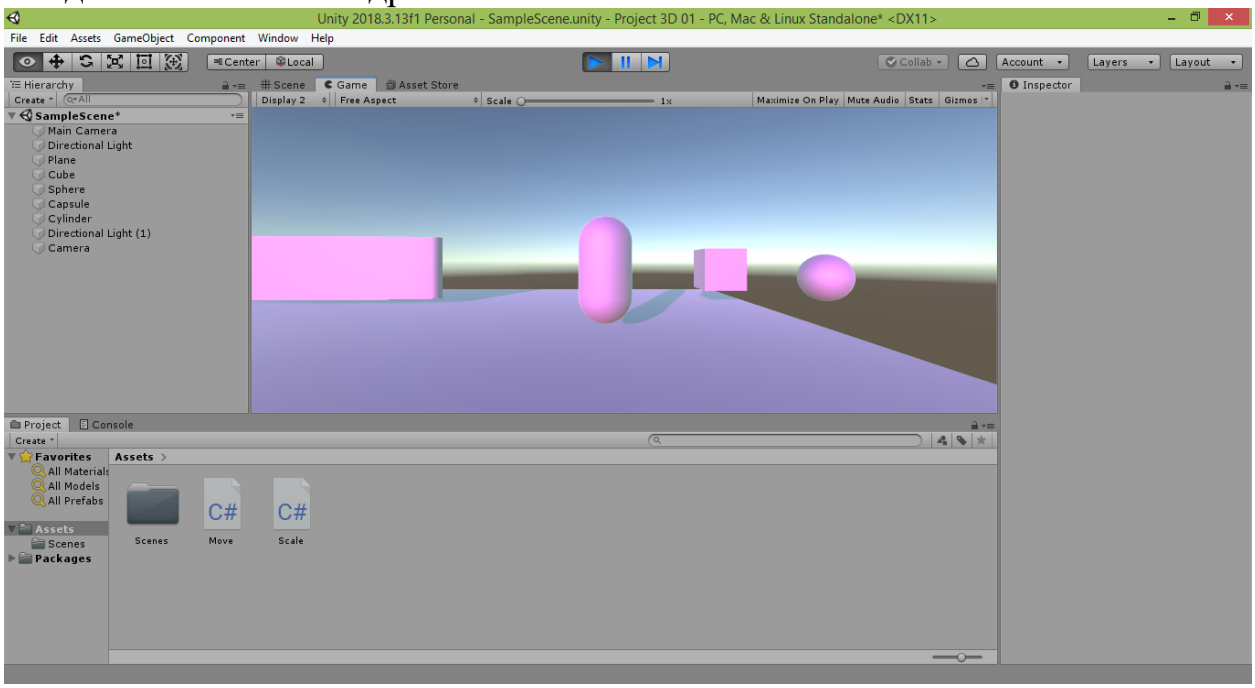
Откроем его двойным щелчком или нажатием клавиши «Enter» в редакторе Visual Studio. Напишем код, похожий на код предыдущего скрипта Move. Отличие будет состоять лишь в том, что вместо параметра position, отвечающего за местоположение, мы будем изменять параметр масштаба localScale, чтобы растянуть объект в направлении красной оси. Для этого увеличим масштаб длины объекта на одну десятую шага, задав значение 0.1f в первом параметре команды `new Vector3`:

```
void Update()
{
    Transform t = GetComponent<Transform>();

    t.localScale += new Vector3(0.1f, 0f, 0f);
}
```

Сохраните код, нажав комбинацию клавиш CTRL+S, нажав вверху значок квадратной синей дискеты или выбрав в меню команду «Файл → Сохранить Scale.cs» («File → Save Scale.cs» в английской версии Visual Studio). Не закрывая окно редактора Visual Studio, перейдите обратно в окно Unity и убедитесь, что написанный вам код отображается справа в окне Inspector, когда вы выделяете файл скрипта Scale. После этого перетяните файл скрипта влево в окно Hierarchy на строчку Cylinder и отпустите.



Теперь если выделить строку Cylinder, то вы увидите в окне Inspector среди свойств созданного нами цилиндра пункт Scale (Script). Это означает, что наш скрипт был привязан к цилиндру. Теперь можно запустить наш проект в игровом режиме, нажав в самом верху кнопку «Play» со значком чёрного треугольника, указывающего вправо. Если в поле зрения вашей камеры попадает цилиндр, вы увидите, как он будет растягиваться по своей длине при каждом обновлении кадра.



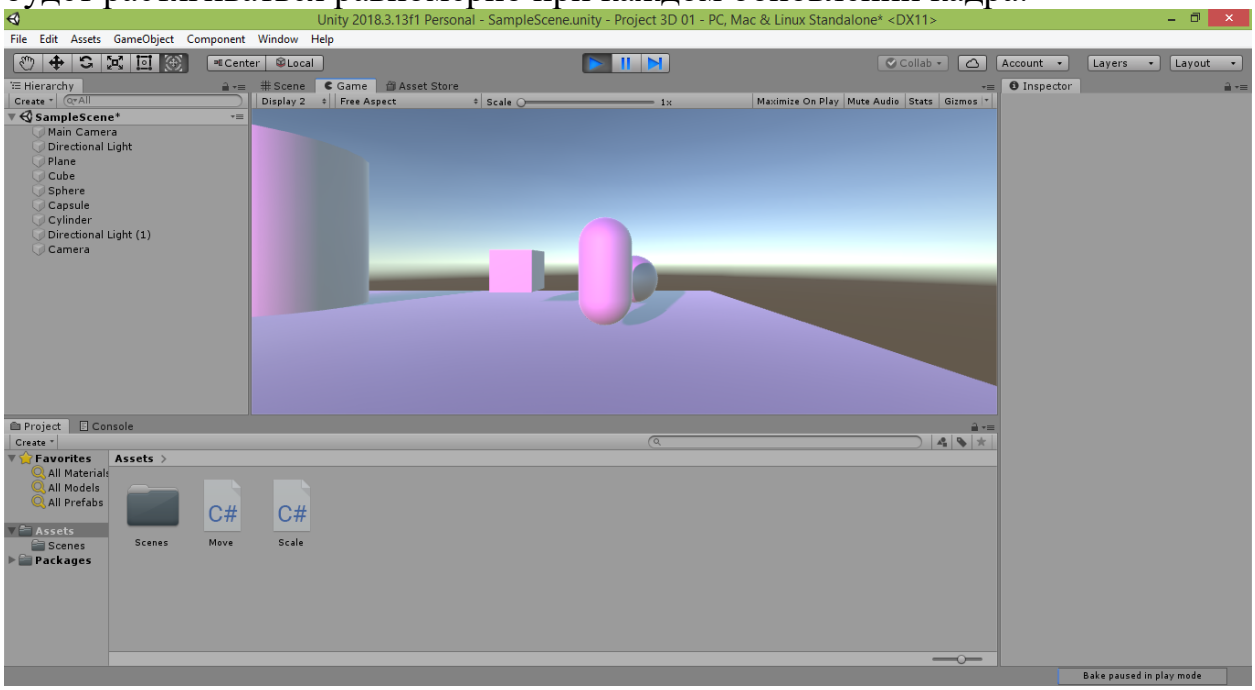
Нажмите ещё раз кнопку «Play», чтобы остановить проигрывание сцены.

Теперь вернёмся в редактор Visual Studio и изменим код, чтобы цилиндр растягивался с одинаковой скоростью по всем трём направлениям – длине

(первая координата x – красная ось), высоте (вторая координата y – зелёная ось) и ширине (третья координата z – синяя ось):

```
void Update()
{
    Transform t = GetComponent<Transform>();

    t.localScale += new Vector3(0.1f, 0.1f, 0.1f);
}
```

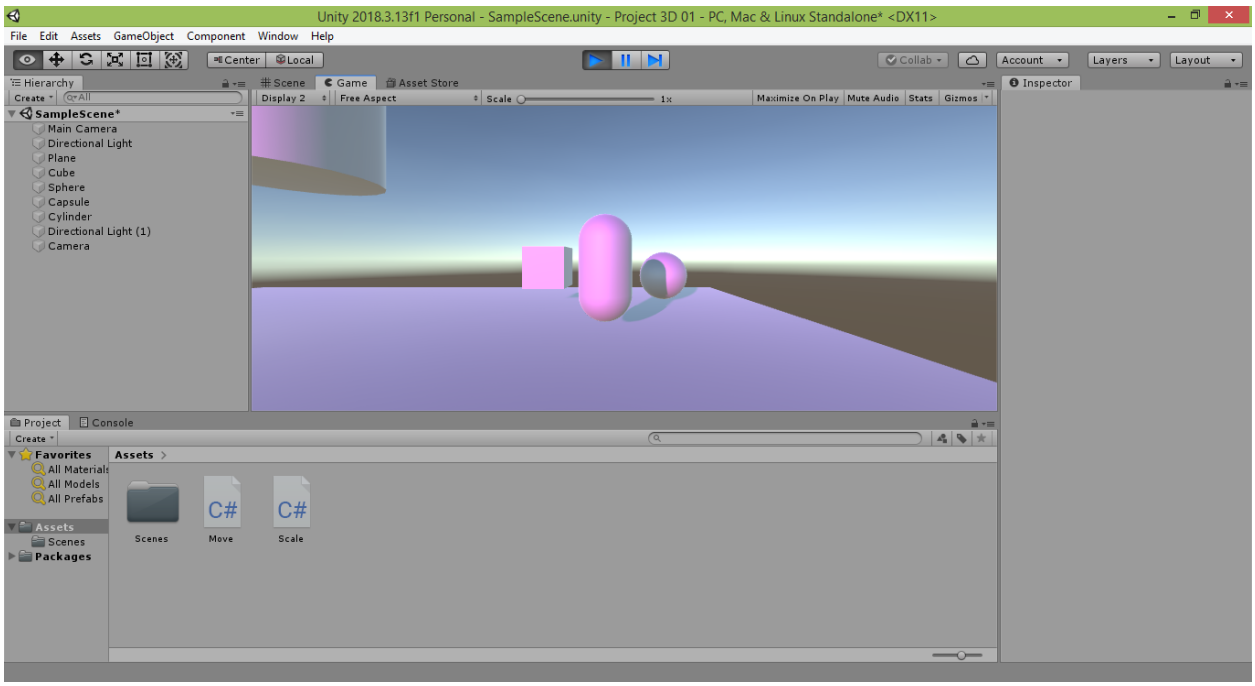
Сохраните код, нажав комбинацию клавиш CTRL+S, и, нажав кнопку «Play» в окне Unity, проверьте, как поведёт себя цилиндр. Теперь цилиндр будет растягиваться равномерно при каждом обновлении кадра.



Остановите проигрывание сцены и вернитесь в редактор Visual Studio. Добавьте изменение позиции цилиндра по высоте. Для этого сместите текущую позицию объекта на вектор, у которого второй элемент, отвечающий за перемещение по высоте вдоль зелёной оси, равен 0.2f:

```
void Update()
{
    Transform t = GetComponent<Transform>();

    t.localScale += new Vector3(0.1f, 0.1f, 0.1f);
    t.position += new Vector3(0f, 0.2f, 0f);
}
```

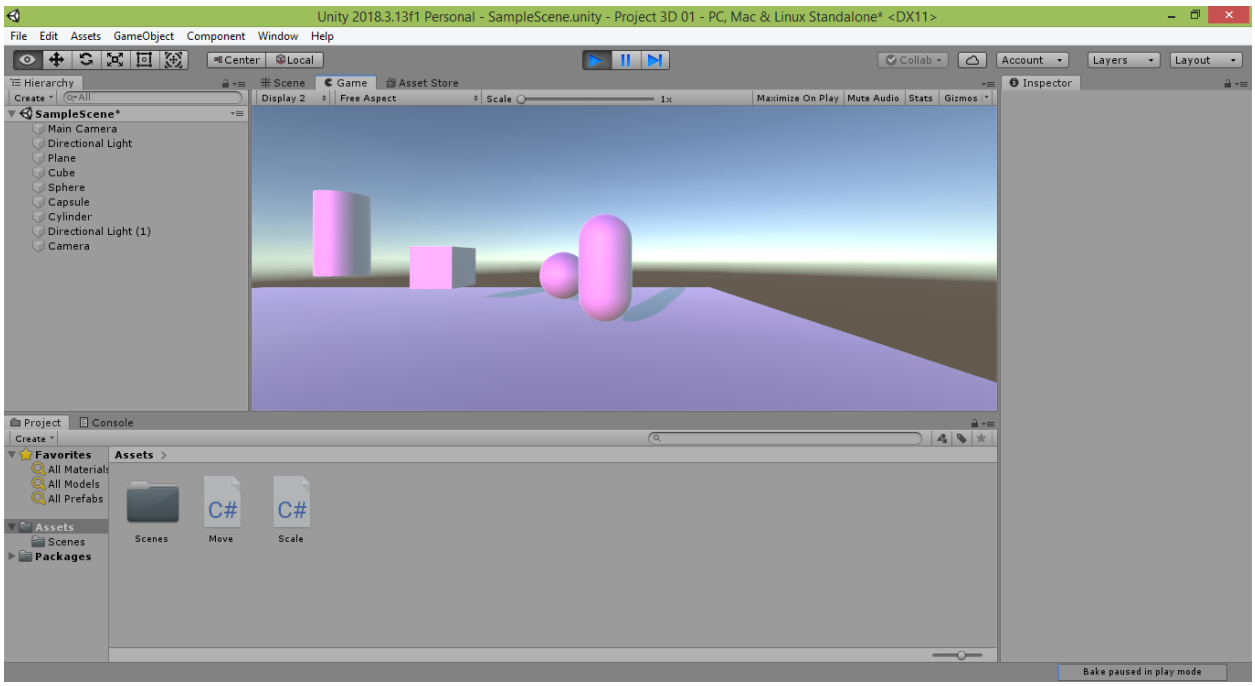
Сохраните код, нажав комбинацию клавиш CTRL+S, и, нажав кнопку «Play» в окне Unity, проверьте, как поведёт себя цилиндр. Цилиндр, увеличиваясь, будет одновременно подниматься вверх.



Остановите проигрывание сцены и вернитесь в редактор Visual Studio. Добавьте изменение позиции цилиндра по ширине (синяя ось). Для этого задайте третьему элементу вектора значение 0.5f:

```
void Update()
{
    Transform t = GetComponent<Transform>();

    t.localScale += new Vector3(0.1f, 0.1f, 0.1f);
    t.position += new Vector3(0f, 0.2f, 0.5f);
}
```

Сохраните код, нажав комбинацию клавиш CTRL+S, и, нажав кнопку «Play» в окне в окне Unity, проверьте, как поведёт себя цилиндр. Цилиндр, увеличиваясь и поднимаясь вверх, будет одновременно удаляться от нас (в какой-то момент может возникнуть ощущение, что он прекратил расти и завис в воздухе, но это лишь оптическая иллюзия – цилиндр находится в постоянном движении и увеличивается в размере).



Важное замечание. Если в вашем проекте присутствует много объектов, для которых предполагается использовать различные комбинации движений, поворотов и других программных изменений, рекомендуется написать несколько простых скриптов, каждый из которых отвечает за одно простейшее изменение объекта. Например, скрипты, отвечающие за перемещение на 0.1 шага по длине, перемещение на 0.1 шага по высоте и перемещение на 0.1 шага по ширине (то же самое – для изменения масштаба по каждой из трёх осей координат). После этого вы сможете комбинировать эти скрипты в одном настраиваемом объекте, делая его поведение настолько сложным, насколько вам потребуется (например, организовать движение с масштабированием, как у нашего цилиндра). Если потребуется увеличить скорость работы скрипта, его можно связать с объектом два и более раз (с помощью этого приёма мы ранее увеличили скорость куба в 2 раза, повторно привязав к нему созданный нами простой скрипт Move).

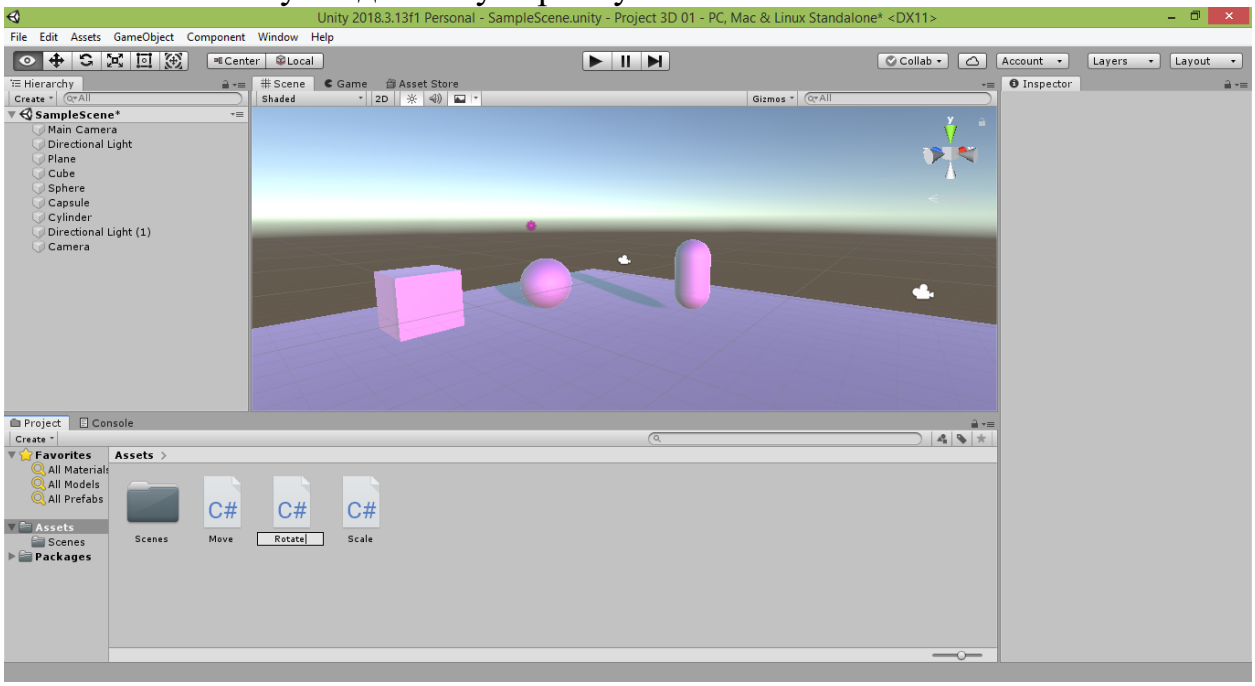
Исходя из указанных соображений, удалим последнюю строку кода из скрипта Scale, чтобы он отвечал только за масштабирование объекта (поскольку для управления перемещением объекта у нас уже есть скрипт Move):

```
void Update()
{
    Transform t = GetComponent< Transform >();

    t.localScale += new Vector3(0.1f, 0.1f, 0.1f);
}
```

Сохраним код, нажав комбинацию клавиш CTRL+S.

3.4. Изменение углов поворота объекта

Выберем в меню Unity команду «Assets → Create → C# Script» и дадим имя Rotate новому созданному скрипту.



Откроем его двойным щелчком или нажатием клавиши «Enter» в редакторе Visual Studio.

Код, который мы напишем, будет немного сложнее кода предыдущих скриптов Move и Scale:

```
public class Rotate : MonoBehaviour
{
    float x = 0, y = 0, z = 0;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        Transform t = GetComponent<Transform>();

        x = x + 1;
        y = y + 1;
        z = z + 1;

        t.rotation = Quaternion.Euler(x, y, z);
    }
}
```

Во-первых, мы добавили в самом начале класса строку

```
float x = 0, y = 0, z = 0;
```

В ней мы создали три переменные x, y и z, определяющие углы текущего поворота объекта вокруг соответствующих осей координат (x – красная ось, y – зелёная ось, z – синяя ось). Начальные значения этих углов мы задали равными 0.

Обратите внимание, что все три переменные имеют тип float – мы уже знаем, что значения именно этого типа принимают используемые нами команды.

Второй ключевой момент – строки в методе Update():

```
x = x + 1;
y = y + 1;
z = z + 1;
```

Нам нужно, чтобы объект не просто повернулся один раз на некоторые углы и остановился, а чтобы он вращался непрерывно. Именно поэтому мы увеличиваем каждую переменную на 1 (соответствующий ей угол поворота в этом случае увеличится на 1 градус) и делаем это в методе Update() – в результате увеличение будет происходить при каждом обновлении кадра сцены.

И, наконец, в методе после всех настроек в методе Update() происходит программное изменение поворота объекта:

```
t.rotation = Quaternion.Euler(x, y, z);
```

Здесь не используется изменение по вектору, как это было в случае со свойствами t.position и t.localScale. Вместо этого через свойство t.rotation сразу (моментально) задаётся новое положение объекта. Поэтому используется оператор задания значения «=» вместо оператора увеличения текущего значения «+=» (увеличение углов мы производим через изменение значения переменных x, y и z). Само получение нового положения объекта после поворота обеспечивается командой Euler из класса Quaternion, которая использует три угла поворота, хранящиеся в переменных x, y и z. Интересно отметить, что слово «кватернион», давшее название классу Quaternion, обозначает окружности, помогающие человеку сориентироваться в том, как повёрнут объект в пространстве (эти окружности-кватернионы – красную, зелёную и синюю – вы использовали, когда, например, вращали источники света). Команда Euler получила своё название в честь известного математика Леонарда Эйлера, который, в частности, изучал геометрию углов поворота.

Кстати говоря, написанный нами код можно сделать чуть более кратким:

```
void Update()
{
    Transform t = GetComponent<Transform>();

    x++;
    y++;
    z++;
```

```
    t.rotation = Quaternion.Euler(x, y, z);
}
```

Запись `x++` называется инкрементом (увеличением). В C++, C# и родственных им языках она означает то же самое, что и запись `x = x + 1`. Именно в честь неё язык программирования C++ получил своё название. Также существует операция `x--`, называемая декрементом (уменьшением), которая представляет собой краткую запись выражения `x = x - 1`.

На самом деле, C-подобные языки (C++, C#, Arduino и другие) позволяют свернуть написанный нами код в две строчки:

```
void Update()
{
    Transform t = GetComponent<Transform>();

    t.rotation = Quaternion.Euler(x++, y++, z++);
}
```

В нём после вызова команды `Euler` и поворота объекта произойдёт увеличение углов, хранящихся в переменных `x`, `y` и `z`. То есть фактически в развёрнутом виде код будет таким:

```
void Update()
{
    Transform t = GetComponent<Transform>();

    t.rotation = Quaternion.Euler(x, y, z);

    x++;
    y++;
    z++;
}
```

Возникает вопрос, как, используя компактную запись, произвести сначала увеличение углов, а уже затем – поворот объекта с использованием новых значений?

Для этого в C-подобных языках предусмотрена операция `++x`, которая тоже работает аналогично `x = x + 1`, но указывает, что инкремент (увеличение значения переменной) нужно произвести в первую очередь, до начала работы других команд. Также существует аналогичная операция декременнта (уменьшения значения переменной) с повышенным приоритетом, которая записывается как `--x` и эквивалентна записи `x = x - 1`, записанной до вызова команды.

Таким образом, наш код в компактной записи будет выглядеть так:

```
void Update()
{
    Transform t = GetComponent<Transform>();
```

```
        t.rotation = Quaternion.Euler(++x, ++y, ++z);
    }
```

В этом случае углы, хранящиеся в переменных `x`, `y` и `z`, сначала будут увеличены, а уже затем – использованы методом `Euler`.

Однако я предлагаю оставить в скрипте прежнюю (некомпактную) запись:

```
    void Update()
    {
        Transform t = GetComponent<Transform>();

        x = x + 1;
        y = y + 1;
        z = z + 1;

        t.rotation = Quaternion.Euler(x, y, z);
    }
```

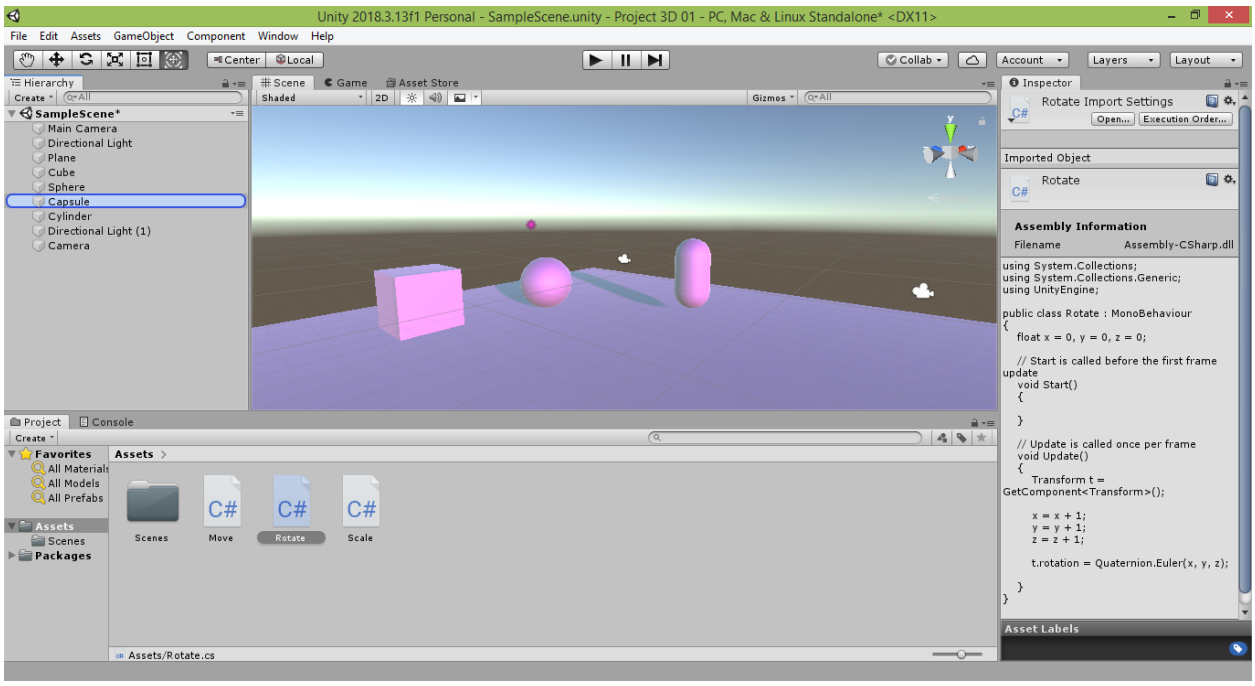
Дело в том, что в этом случае мы сможем увеличивать переменные `x`, `y` и `z` на любые другие величины. Например, чтобы увеличить угол поворота вокруг красной оси на 8 градусов, мы запишем:

```
x = x + 8;
```

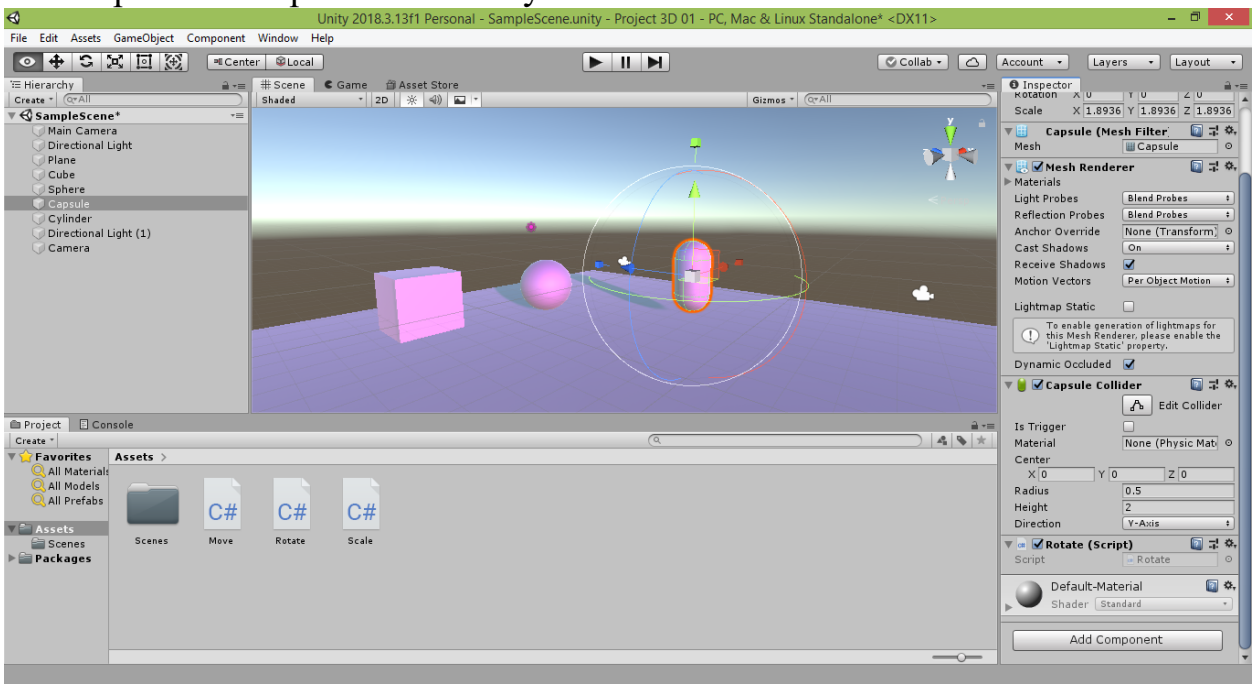
Это позволит нам задавать различающиеся скорости поворота вокруг осей координат.

Во-первых, отличие будет состоять в том, что мы будем менять параметр rotation, отвечающий за угол поворота объекта в разных плоскостях.

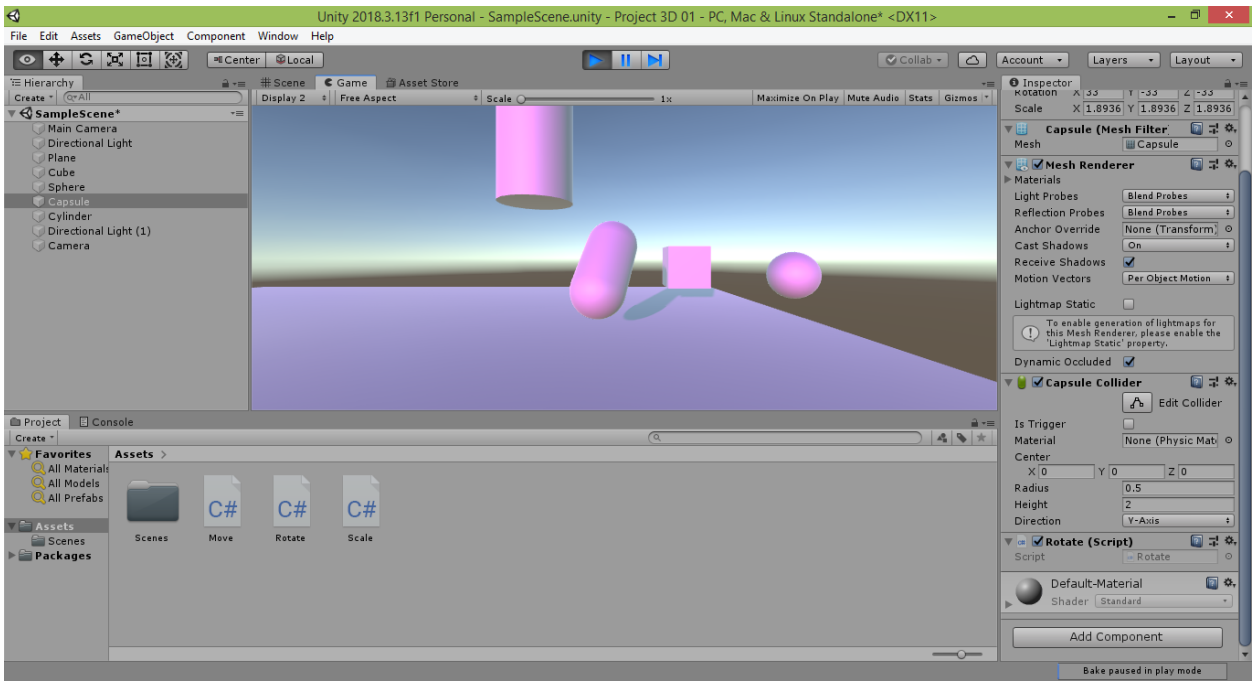
Теперь, когда программа написана, сохраните код, нажав комбинацию клавиш CTRL+S, нажав вверху значок квадратной синей дискеты или выбрав в меню команду «Файл → Сохранить Rotate.cs» («File → Save Rotate.cs» в английской версии Visual Studio). Не закрывая окно редактора Visual Studio, перейдите обратно в окно Unity и убедитесь, что написанный вам код отображается справа в окне Inspector, когда вы выделяете файл скрипта Rotate. После этого перетяните файл скрипта влево в окно Hierarchy на строчку Capsule и отпустите.



Теперь если выделить строку Capsule, то вы увидите в окне Inspector среди свойств созданной нами капсулы пункт Rotate (Script). Это означает, что наш скрипт был привязан к капсуле.



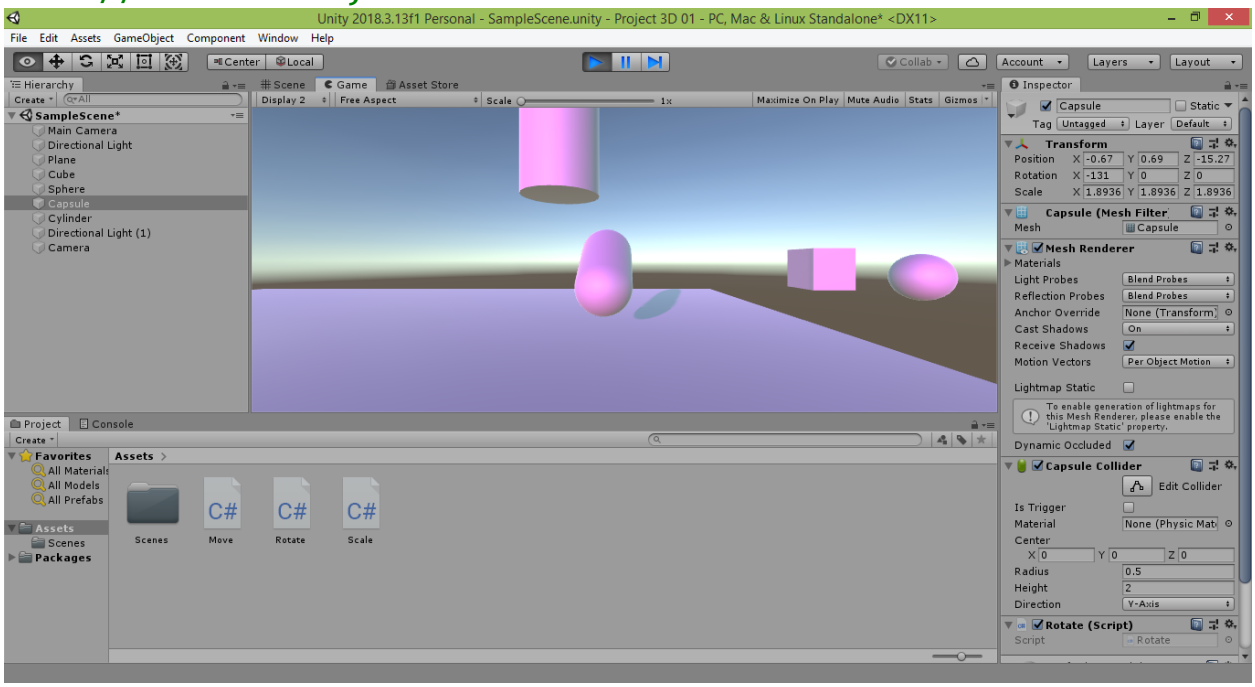
Теперь можно запустить наш проект в игровом режиме, нажав в самом верху кнопку «Play». Если в поле зрения вашей камеры попадает капсула, вы увидите, как она будет вращаться одновременно по всем трём осям координат.



Нажмите ещё раз кнопку «Play», чтобы остановить проигрывание сцены.

Если вас не устраивает такой вариант вращения, вы можете удалить или закомментировать (используя `//` в начале строки) в скрипте любые одну-две строки. Например, нижеприведённый код обеспечит вращение объекта только вокруг красной оси:

```
x = x + 1;
// y = y + 1;
// z = z + 1;
```



3.5. Изменение объекта по нажатию клавиш

Обычно в проектах Unity (если только это не видеоролики), пользователь не является сторонним наблюдателем, а имеет возможность управлять поведением всех или некоторых объектов (например, в играх). Для этого они может использовать такие устройства как мышь и клавиатура. Поскольку клавиатура предоставляет гораздо больше управляющих воздействий, чем мышь, рассмотрим, как управлять созданными нами объектами при помощи клавиш.

Для этого нам пригодится ещё один класс, который называется `Input` (в переводе с английского означает «Ввод»). В этом классе есть полезная команда, которая называется `GetKey`. В качестве параметра она принимает код клавиши и выдаёт одно из двух так называемых булевских значений:

- **true** («истина», «да», «верно»)
- **false** («ложь», «нет», «неверно»)

Чтобы не запоминать числовой код каждой клавиши, можно воспользоваться перечислением `KeyCode`, при обращении к которому через точку указывается название отслеживаемой клавиши.

Таким образом, чтобы отследить, что была нажата клавиша «Пробел», нужно использовать код:

```
Input.GetKey(KeyCode.Space)
```

Чтобы по нажатию клавиши выполнить команду, следует воспользоваться условным оператором if , который пропустит команду на выполнение только в том случае, если полученный им параметр имеет значение **true**. Таким образом, если мы хотим задать, например, изменение угла поворота объекта вокруг красной оси `x` по нажатию пользователем клавиши «X», мы должны использовать следующую строку кода:

```
if ( Input.GetKey(KeyCode.X) ) x = x + 1;
```

Пропишем в скрипте Scale, чтобы вращение объекта вокруг осей `x`, `y` и `z` происходило по нажатию клавиш «X», «Y» и «Z», соответственно:

```
void Update()
{
    Transform t = GetComponent<Transform>();

    if (Input.GetKey(KeyCode.X)) x = x + 1;
    if (Input.GetKey(KeyCode.Y)) y = y + 1;
    if (Input.GetKey(KeyCode.Z)) z = z + 1;

    t.rotation = Quaternion.Euler(x, y, z);
}
```

Сохраните код, нажав комбинацию клавиш CTRL+S, и, нажав кнопку «Play» в окне Unity, проверьте, как поведёт себя капсула после нажатия клавиш «X», «Y» и «Z». Теперь она вращается только под вашим управлением.

Теперь перейдите в скрипт Move и пропишите движение связанного с ним объекта по клавише «Стрелка вправо» («RightArrow»):

```
void Update()
{
    Transform t = GetComponent<Transform>();

    if (Input.GetKey(KeyCode.RightArrow))
        t.position = t.position + new Vector3(0.1f, 0f, 0f);
}
```

Возникает закономерный вопрос: как программно организовать движение объекта в противоположном направлении? Для этого нужно не увеличивать, а уменьшать координату объекта по красной оси x, придав отрицательное значение первому элементу вектора смещения. Сделаем это по нажатию на клавишу «Стрелка влево» («LeftArrow»):

```
void Update()
{
    Transform t = GetComponent<Transform>();

    if (Input.GetKey(KeyCode.RightArrow))
        t.position = t.position + new Vector3(0.1f, 0f, 0f);

    if (Input.GetKey(KeyCode.LeftArrow))
        t.position = t.position + new Vector3(-0.1f, 0f, 0f);
}
```

Сохраните код, нажав комбинацию клавиш CTRL+S, и, нажав кнопку «Play» в окне Unity, проверьте, как поведёт себя куб. Теперь наш куб может двигаться не только вправо, но и влево.

Движение вперёд и назад (вдоль синей оси z) обеспечивается путём задания ненулевого значения третьему элементу вектора смещения. Организуем это по нажатию клавиш «Стрелка вверх» («UpArrow») и «Стрелка вниз» («DownArrow»), соответственно:

```
void Update()
{
    Transform t = GetComponent<Transform>();

    if (Input.GetKey(KeyCode.RightArrow))
        t.position = t.position + new Vector3(0.1f, 0f, 0f);

    if (Input.GetKey(KeyCode.LeftArrow))
        t.position = t.position + new Vector3(-0.1f, 0f, 0f);

    if (Input.GetKey(KeyCode.UpArrow))
        t.position = t.position + new Vector3(0f, 0f, 0.1f);

    if (Input.GetKey(KeyCode.DownArrow))
        t.position = t.position + new Vector3(0f, 0f, -0.1f);
```

```
}
```

Сохраните код, нажав комбинацию клавиш CTRL+S, и, нажав кнопку «Play» в окне Unity, убедитесь, что куб теперь может перемещаться по клавишам-стрелкам во всех четырёх горизонтальных направлениях.

Часто в играх для перемещения персонажей и объектов вместо или наравне со стрелками используют группу так называемых игровых клавиш WASD, где для перемещения влево используется клавиша «W», вверх – клавиша «A», вниз – клавиша «S», вправо – клавиша «D». Вы можете использовать группу игровых клавиш и группу клавиш со стрелками, чтобы, например, в вашем приложении с одной клавиатуры можно было одновременно управлять двумя персонажами (режим игры для двух пользователей).

Важное замечание. Как и код в редакторе Visual Studio, проект в Unity требует сохранения внесённых в него изменений. Для сохранения вашего проекта выберите в меню программы Unity пункт «File → Save» или нажмите комбинацию клавиш CTRL+S. Даже если вы забудете это сделать, Unity при закрытии напомнит вам, что в проект были внесены изменения, и спросит, следует ли их сохранить. Если вы согласны на фиксацию изменений, произведённых с проектом, нажмите «Save» («Сохранить»), в противном случае – откажитесь, нажав «Don’t save» («Не сохранять»).

Также рекомендуется не ждать закрытия программы, а периодически производить сохранения промежуточных состояний вашего проекта, чтобы не потерять их в случае возникновения внештатной ситуации (например, внезапного выключения электричества).

P.S.: В процессе длительного написания данного руководства у автора произошло «зависание» Unity (видимо, вследствие большой нагрузки на оперативную память сразу нескольких приложений – Unity, Visual Studio, Word и других). В результате пришлось экстренно завершить работу Unity через Диспетчер задач, после чего проект при открытии оказался пустым – в нём остались только файлы скриптов, а несохранённая сцена, которую вы видели на скриншотах, была полностью утеряна. Чаще сохраняйте свои работы!

Задание для самостоятельной работы

1. Выберите клавиши и организуйте возможность перемещения куба по вертикали (вверх и вниз).

2. Выберите клавиши и организуйте возможность изменения масштаба цилиндра отдельно по каждой из трёх осей координат.

3. Попробуйте скомбинировать созданные три скрипта, привязав их к шару, и изменить его местоположение, форму и углы поворота, используя заданные в скриптах клавиши.

4. Скопируйте в отдельные файлы результаты вашей работы:

- программный код всех трёх скриптов:
    1. В редакторе Visual Studio выделите весь код скрипта комбинацией клавиш CTRL+A или выбором команды «Правка → Выделить все» / «Edit → Select all»
    2. Скопируйте выделенный код комбинацией клавиш CTRL+C, выбором в меню команды «Правка → Копировать» / «Edit → Copy» или щелчком правой кнопкой мыши на выделенном коде и выбором команды «Копировать» / «Copy».
    3. В программе «Блокнот» / «Notepad» вставьте скопированный код комбинацией клавиш CTRL+V или щелчком правой кнопкой мыши на пустом пространстве и выбором команды «Вставить» / «Paste».
    4. Сохраните txt-файл с кодом комбинацией клавиш CTRL+S или выбором в меню команды «Файл → Сохранить» / «File → Save».
- скриншот сцены с шаром, изменённым при помощи скриптов:
    1. При открытом окне со сценой Unity нажмите клавишу PrtSc (Print Screen) на клавиатуре.
    2. В программе Paint вставьте скриншот окна сцены комбинацией клавиш CTRL+V или щелчком правой кнопкой мыши на чистом пространстве и выбором команды «Вставить» / «Paste».
    3. Сохраните png-файл со скриншотом комбинацией клавиш CTRL+S или выбором в меню команды «Файл → Сохранить» / «File → Save».

Результаты вашей работы рекомендуется показать преподавателю и обсудить с ним.

Глава 4. Настройка пользовательского интерфейса

4.1. Пользовательский интерфейс (User Interface)

Все мы с вами пробовали играть в компьютерные игры. И первое, на что обращали внимание при запуске игрового приложения – это набор всевозможных кнопок, переключателей и других графических элементов, с помощью которых можно задать настройки игры (например, ввести имя персонажа, включить/выключить звуки и фоновую музыку) или управлять ходом самой игры (ставить игру на паузу, открывать магазин, задавать количество покупаемых вещей и т.д.). Все вместе эти элементы объединяются общий понятием «пользовательский интерфейс». По-английски это понятие пишется, как «User Interface» или «Graphic User Interface» («графический пользовательский интерфейс»). Также в целях сокращения этих английских словосочетаний используются их аббревиатуры – «UI» и «GUI». Слово «interface» в переводе с английского означает «связь», «средство связи». Схожим с ним понятием является слово «interaction» – «взаимодействие». Таким образом, пользовательский интерфейс – это средство связи пользователя с компьютерной программой, которое позволяет ему взаимодействовать с ней. Сегодня наиболее распространёнными элементами пользовательского интерфейса в компьютерных программах являются кнопки, флажки, переключатели, числовые счётчики, выпадающие списки, ползунки и полосы прокрутки. Они имитируют реально существующие объекты, которые вы встречали на панелях управления электрических приборов. Например, кнопки, переключатели и ползунки есть у радиоприёмника, пылесоса, станка на заводе, автомобиля, самолёта и у многих других технических устройств.

Однако следует также подчеркнуть, что пользовательский интерфейс со временем может претерпевать существенные изменения. Например, до 1990-х годов взаимодействие пользователя с компьютерной программой происходило зачастую в текстовом формате: человек с клавиатуры вводил текстовые команды, которые указывали программе, как управлять работой компьютера. В ответ программа показывала на экране компьютера текст или картинку и ожидала следующей команды пользователя. Сегодня в текстовом формате с программами взаимодействуют преимущественно только их разработчики – программисты. Однако даже они активно используют средства графического интерфейса для быстрого и удобного создания оконных приложений.

Также, возможно, вы слышали о таком понятии, как «нейроинтерфейс». Это перспективный пользовательский интерфейс, которому сегодня стараются найти применение в управлении беспилотными летательными аппаратами – дронами. Оператор, управляющий беспилотным устройством, надевает на голову шапочку с датчиками и, после некоторой адаптации программы к его мозговой деятельности, начинает усилием мысли посылать

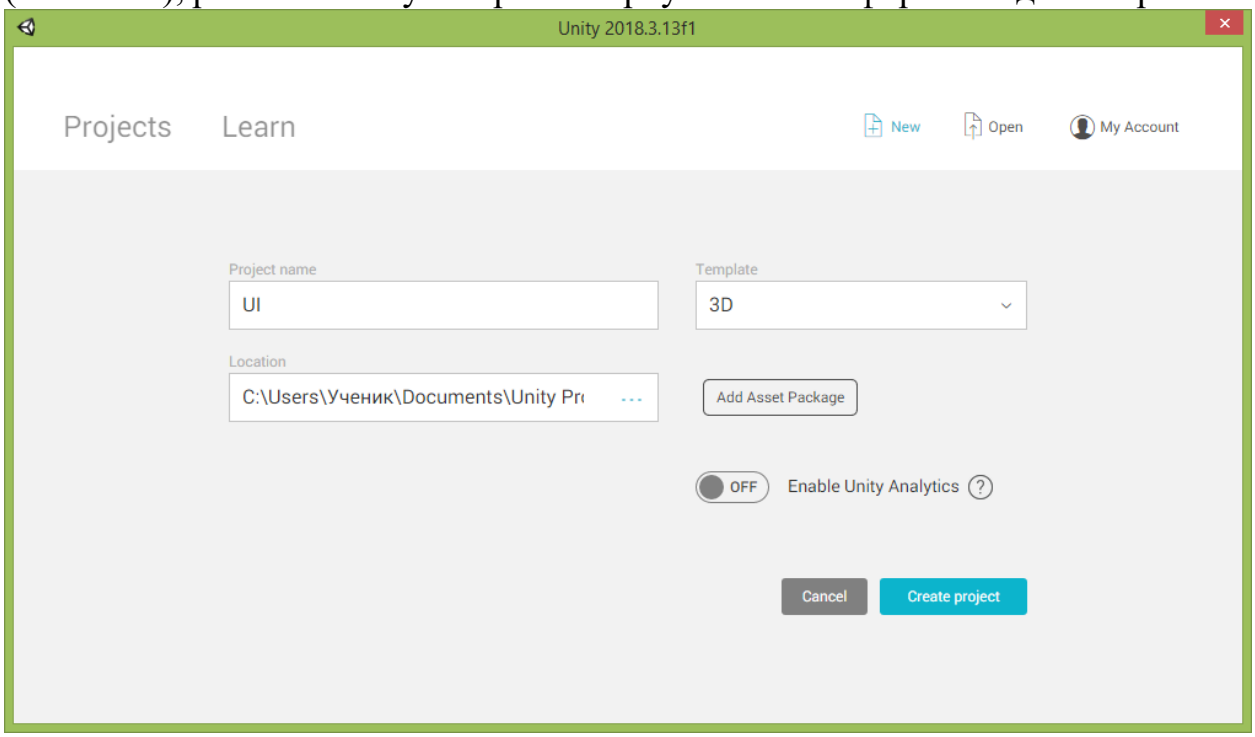
ей команды. В свою очередь, программа, получая от оператора команды в форме импульсов от различных областей мозга, приводит в движение техническое устройство. Вполне вероятно, что в недалёком будущем все известные нам программы будут управляться (и разрабатываться) без использования мышки и клавиатуры – лишь одним усилием человеческой мысли.

А пока мы на примере среды для разработки компьютерных игр Unity познакомимся поближе с элементами классического пользовательского интерфейса. Сегодня трудно представить компьютерную игру, которая бы не предоставляла пользователю удобные и красочные элементы управления собой. Изучив основы работы с элементами пользовательского интерфейса, вы тоже сможете создавать подобные графические приложения.

4.2. Создание сцены и настройка объекта «Canvas» («Полотно»)

Первым делом создадим в Unity новый 3D-проект.

Запустите Unity и в открывшемся окне нажмите кнопку «New» («Новый»), расположенную справа вверху. Появится форма создания проекта.



В поле «Project name» («Название проекта») задайте имя своему новому проекту. Я свой 3D-проект назвал просто «UI».

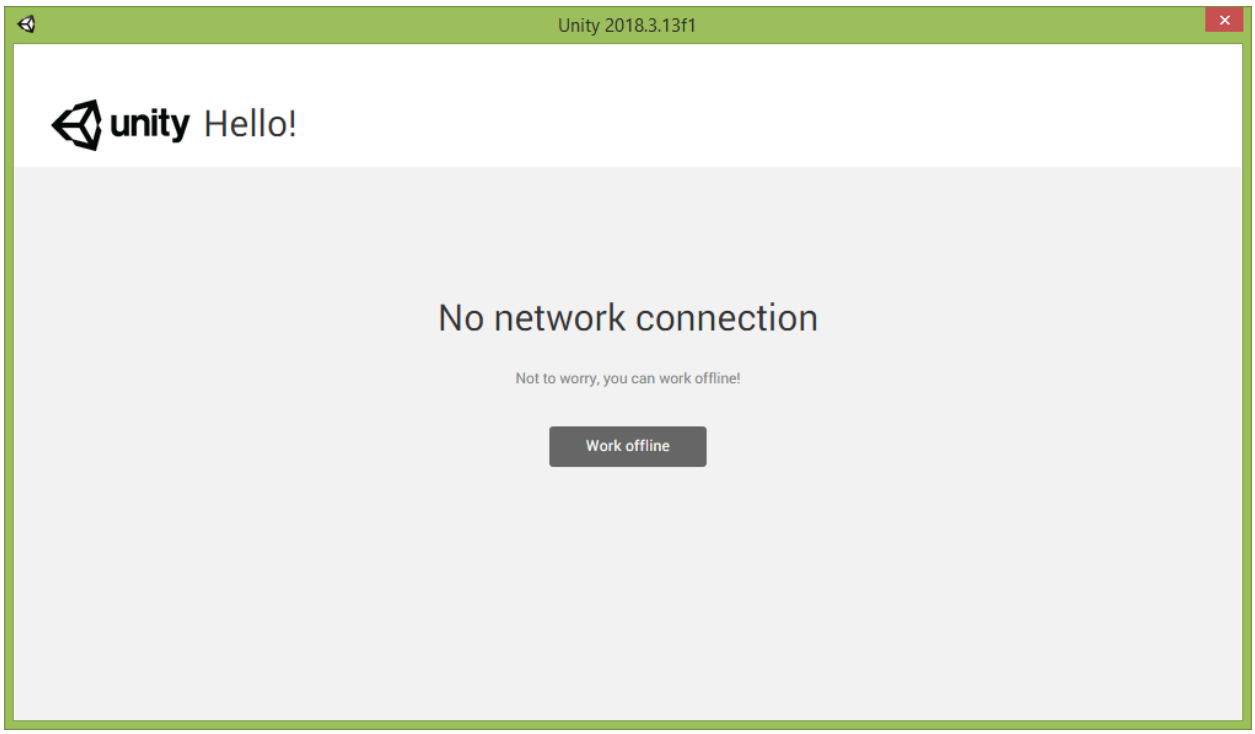
Убедитесь, что в списке «Template» («Шаблон») выбран пункт «3D».

В поле «Location» («Местоположение») укажите адрес папки, где вы хотели бы разместить файлы вашего проекта. Для своих проектов я создал у себя на компьютере папку «Unity Projects» в стандартной папке «Documents» («Документы») и теперь указал путь к ней.

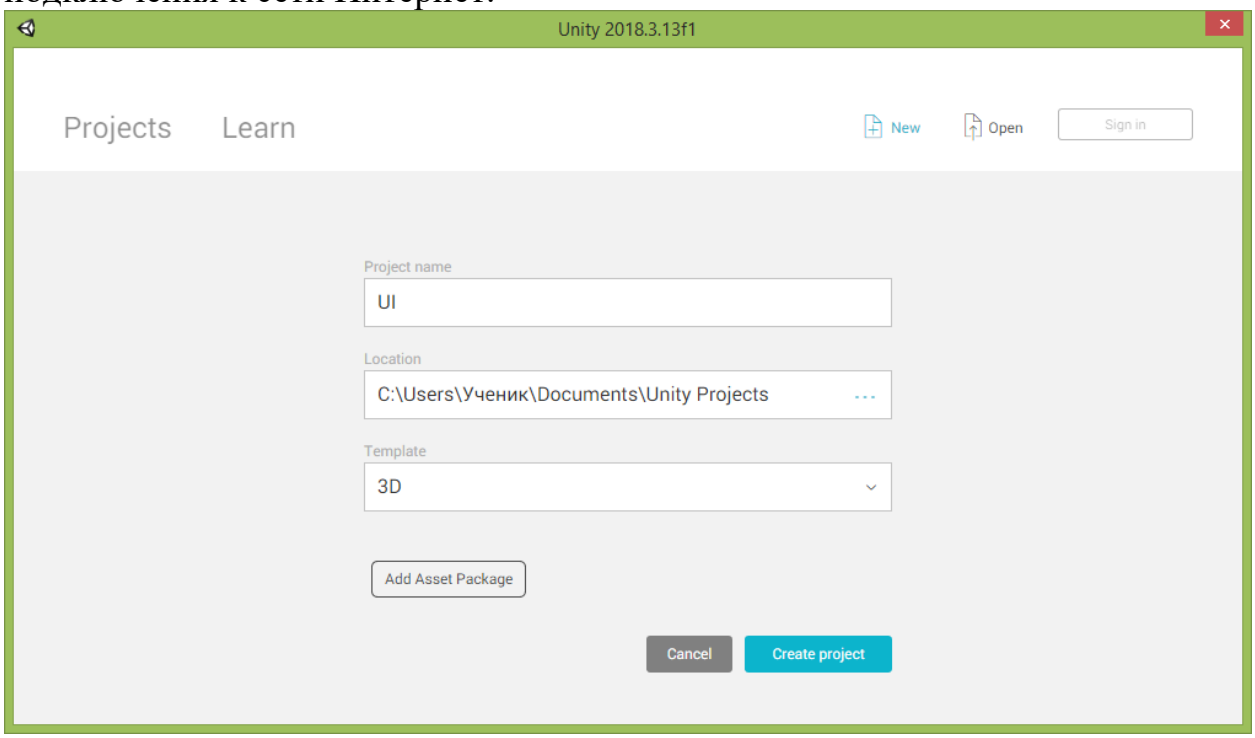
Если вы подключены к сети Интернет и вошли под учётной записью Unity, то в окне создания проекта вы также увидите переключатель-движок «Enable Unity Analytics» («Задействовать аналитику Unity»). Переведите его в положение «OFF» («Выключено»), чтобы Unity не отправляла информацию о ваших действиях на сайт Unity.

Если же доступ к сети Интернет у вас выключен, то при запуске среды Unity на экране появится уведомление: «No network connection» («Отсутствует подключение к сети»)Ниже под этой надписью будет небольшое пояснение: «Not to worry, you can work offline!» («Не волнуйтесь, вы можете работать без сети!»).

Нажмите на кнопку «Work offline» («Работать без сети») под этой надписью.



При создании проекта в режиме «оффлайн» форма создания проекта будет похожа на показанную ранее. На ней будут отсутствовать лишь упомянутый переключатель-движок «Enable Unity Analytics» и кнопка «Add Asset Package» («Добавить пакет компонентов»), которые требуют наличия подключения к сети Интернет.



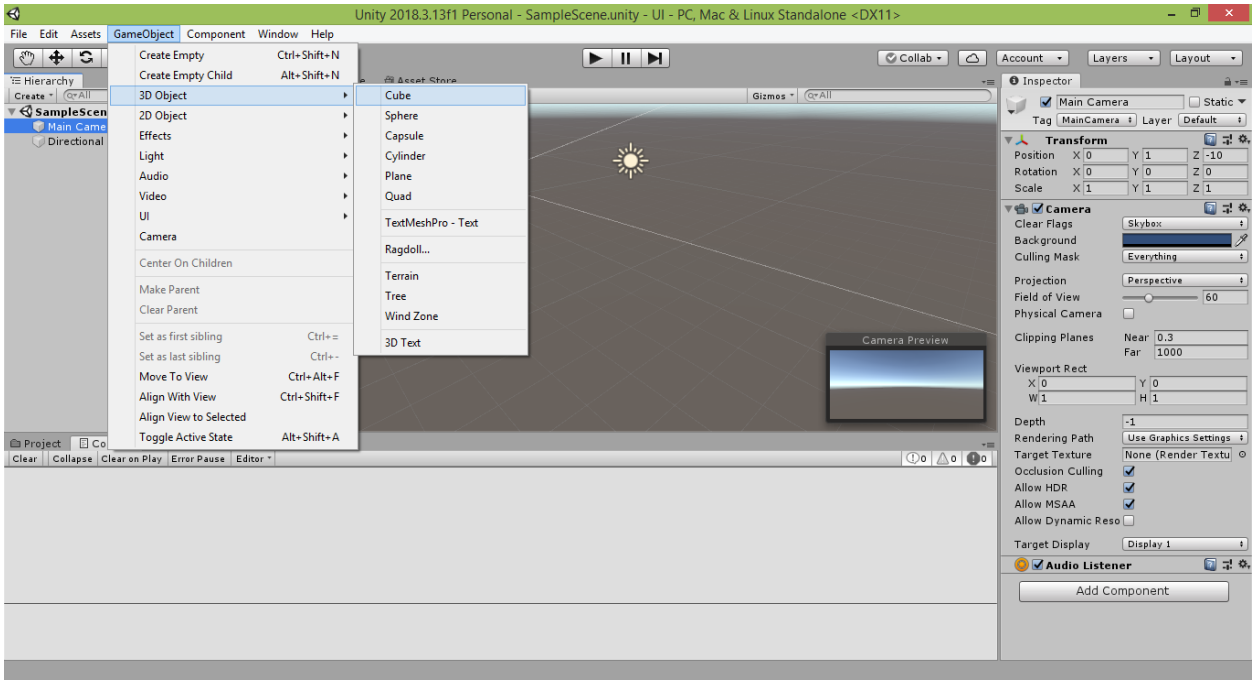
После того, как вы заполнили все поля, нажмите кнопку «Create Project» («Создать проект») и ожидайте, пока Unity создаст файлы нового проекта, и запустит окно редактора.

После завершения процесса создания проекта появится окно с пустой сценой, на которой вы увидите только источник направленного света «Directional Light» и камеру «Main Camera».

Приложение, которое мы создадим, позволит пользователю управлять объектами сцены при помощи элементов пользовательского интерфейса (в Unity чаще всего вы будете встречать сокращение «UI»).

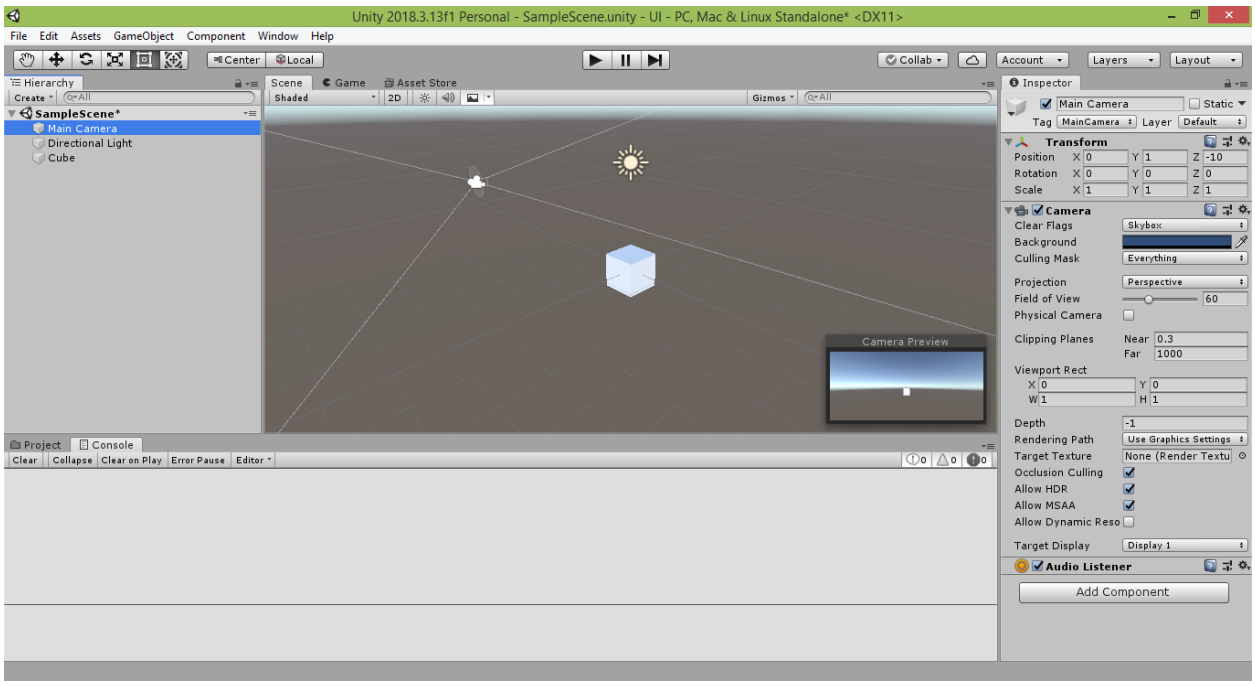
Первым делом добавим на сцену куб.

Выберите в меню Unity команду «GameObject → 3D Object → Cube».

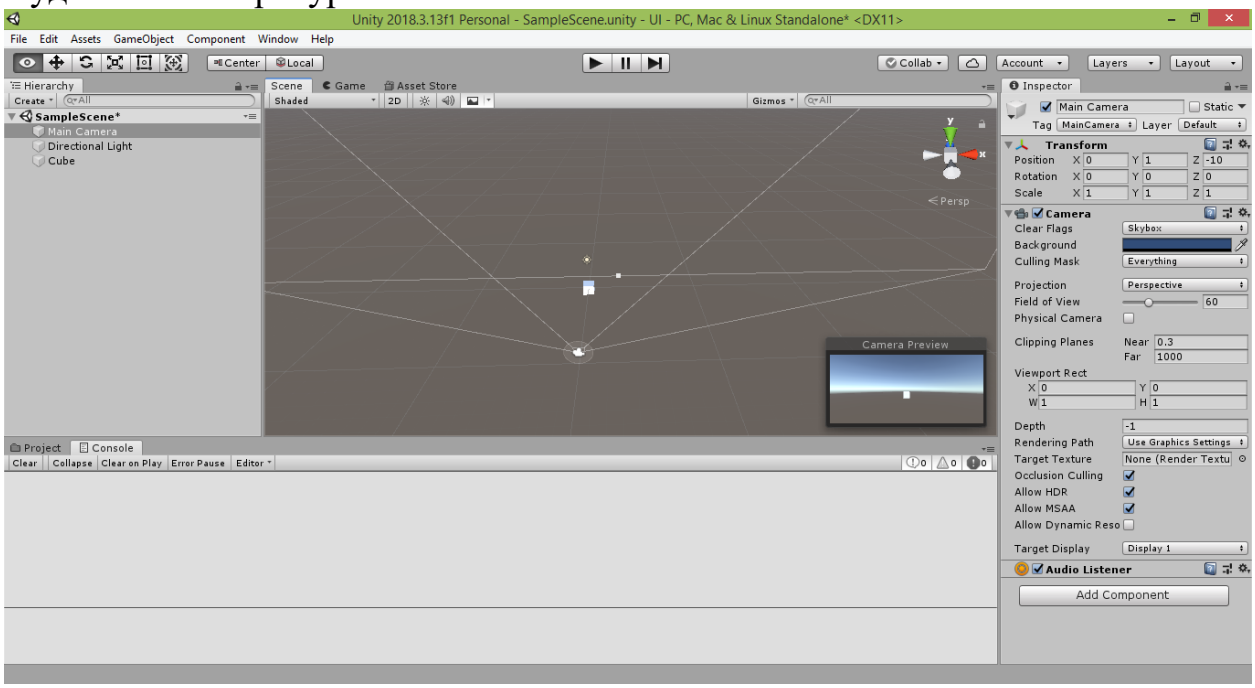


В результате в центре сцены появится игровой 3D-объект «Cube».

Нажмите в левом окне Hierarchy («Иерархия») на строчку «Main Camera», чтобы в правом нижнем углу появилось окно «Camera Preview» («Вид с камеры»), а у значка камеры на сцене появились белые линии, схематично показывающие область охвата сцены камерой. Скорее всего, камера будет направлена в центр сцены прямо на созданный вами куб, и изображение куба будет транслироваться справа внизу в окне предпросмотра.

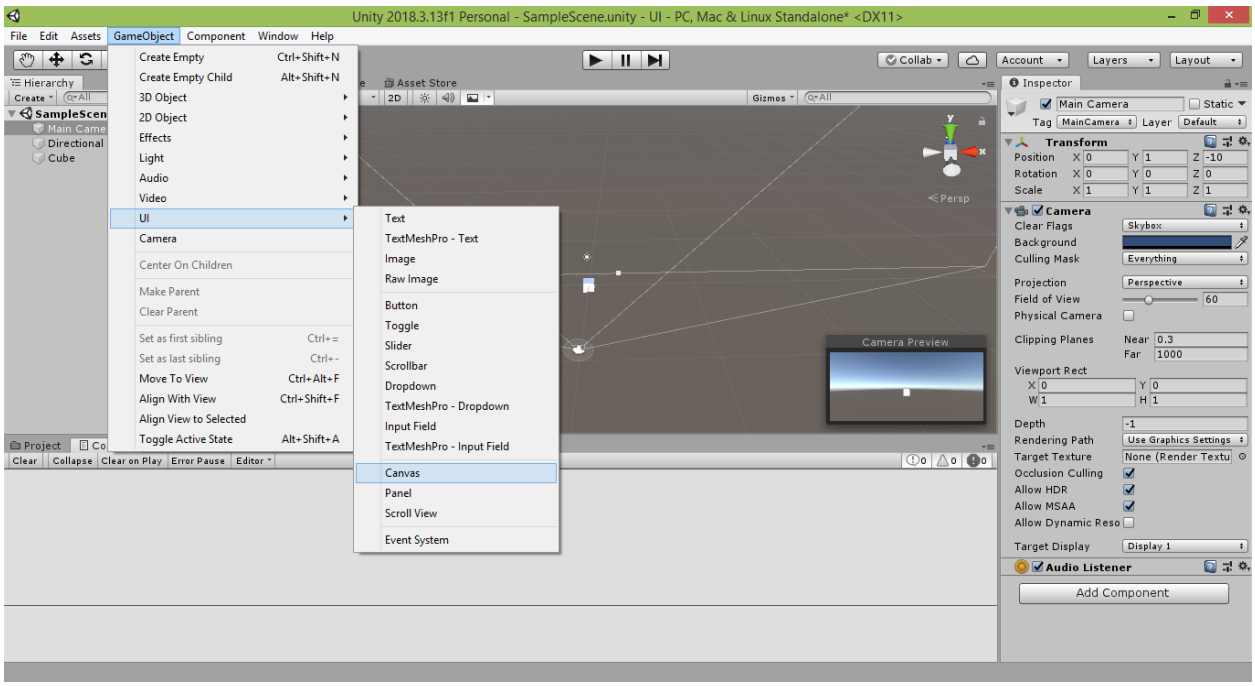


Зажав правую кнопку мыши и используя игровые клавиши «W», «A», «S» и «D», переместите точку своего наблюдения, чтобы видеть камеру и куб с удобного вам ракурса.

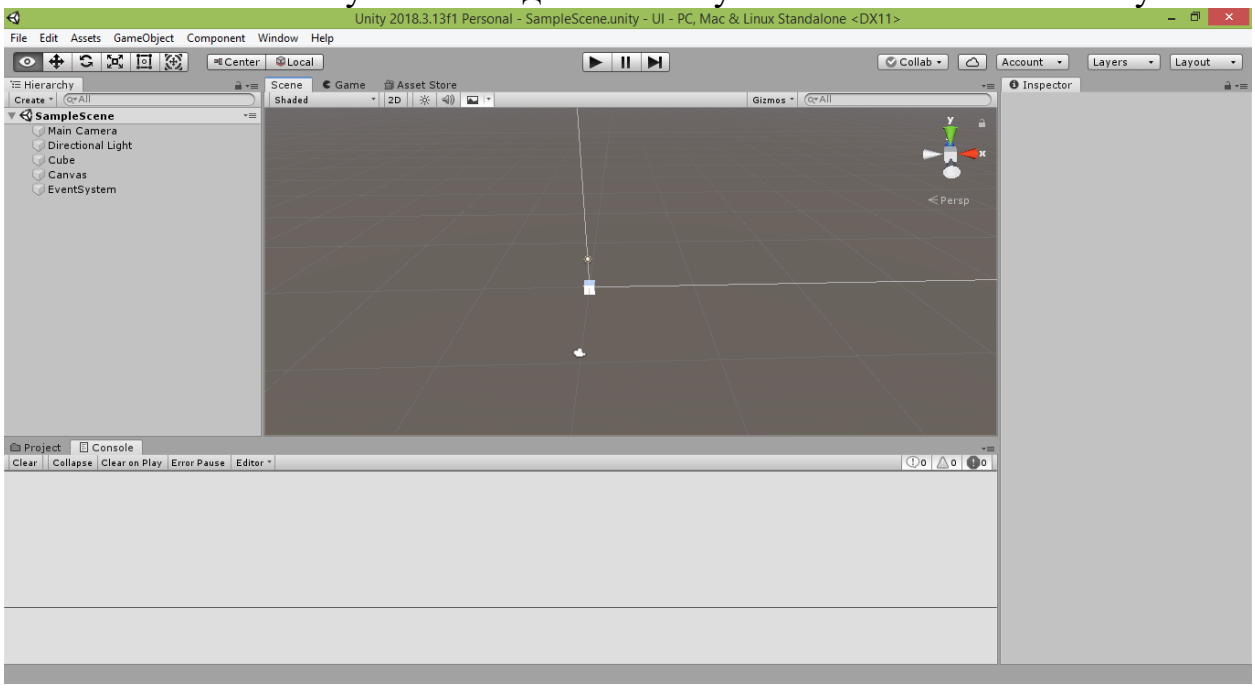


Также вы можете перейти на вкладку «Game» («Игра»), чтобы увидеть сцену с камеры в полноэкранном формате.

Теперь добавим на сцену первый компонент пользовательского интерфейса, именуемый «Canvas» (в переводе с английского означает «Полотно», «Холст», «Канва»). Данный компонент будет являться площадкой, на которой мы разместим элементы управления нашим приложением (подобно тому, как художник создаёт объекты на полотне своей картины). В меню Unity выберите команду «GameObject → UI → Canvas».



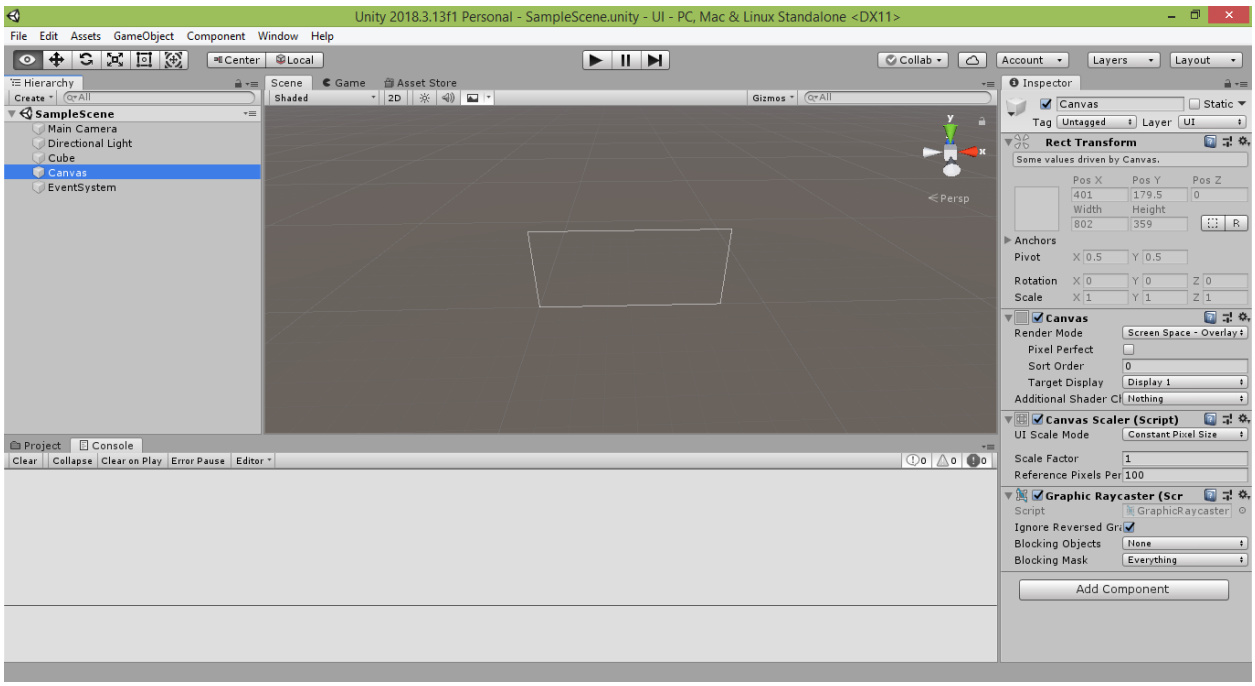
В окне Hierarchy появятся два новых пункта: «Canvas» и «EventSystem».



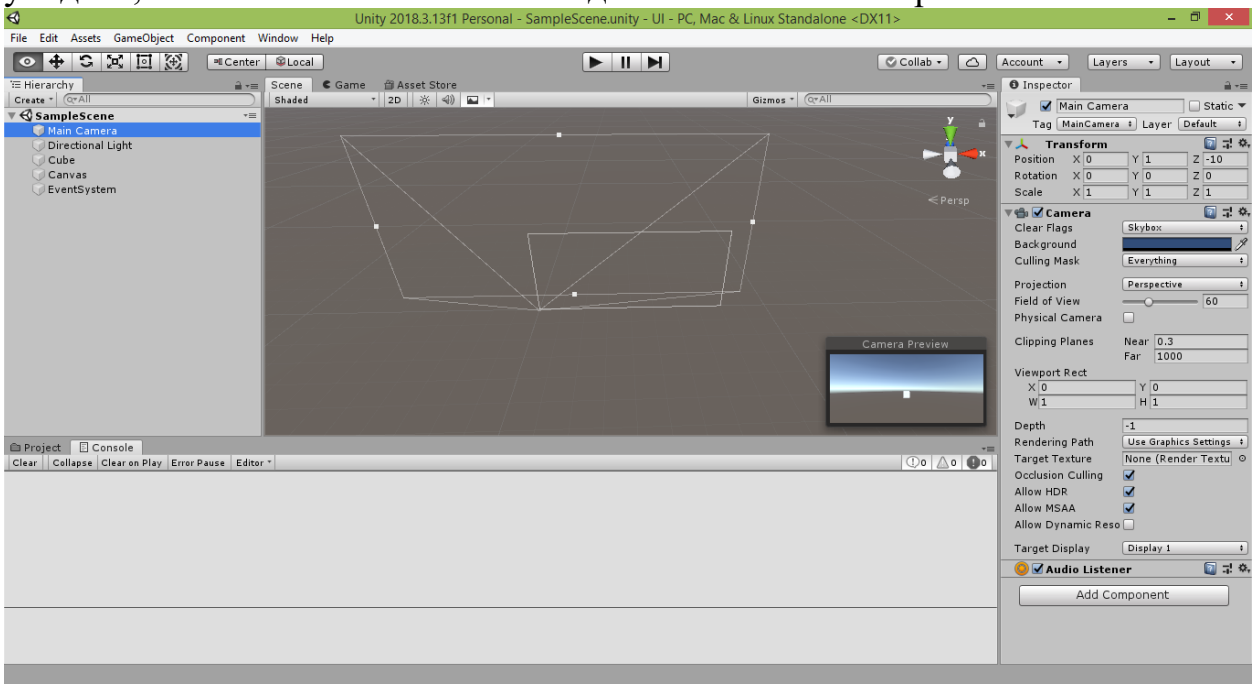
Объект «Canvas» изначально пуст и на сцене представлен прямоугольной рамкой.

По мере разработки нашего приложения объект «Canvas» будет дополняться новыми объектами – элементами управления пользовательского интерфейса.

Чтобы увидеть контуры полотна полностью, щёлкните два раза на строке «Canvas» в окне Hierarchy.



Теперь если вы щёлкните на строку «Main Camera» в окне Hierarchy, вы увидите, как соотносится область видимости вашей камеры и полотно.



Здесь следует отметить, что размер полотна «Canvas», отображаемого в процессе работы вашего приложения, будет определяться значением свойства «Render Mode» («Режим представления»), которое можно настроить в окне Inspector («Инспектор»).

Здесь доступны три варианта значения:

1) «Screen Space – Overlay» («Экранное пространство – верхний слой») означает, что все элементы управления вашим приложением будут видны поверх сцены независимо от близости её расположения к наблюдателю. В этом режиме вы не сможете изменить размер полотна – оно автоматически растянется на весь экран при запуске приложения.

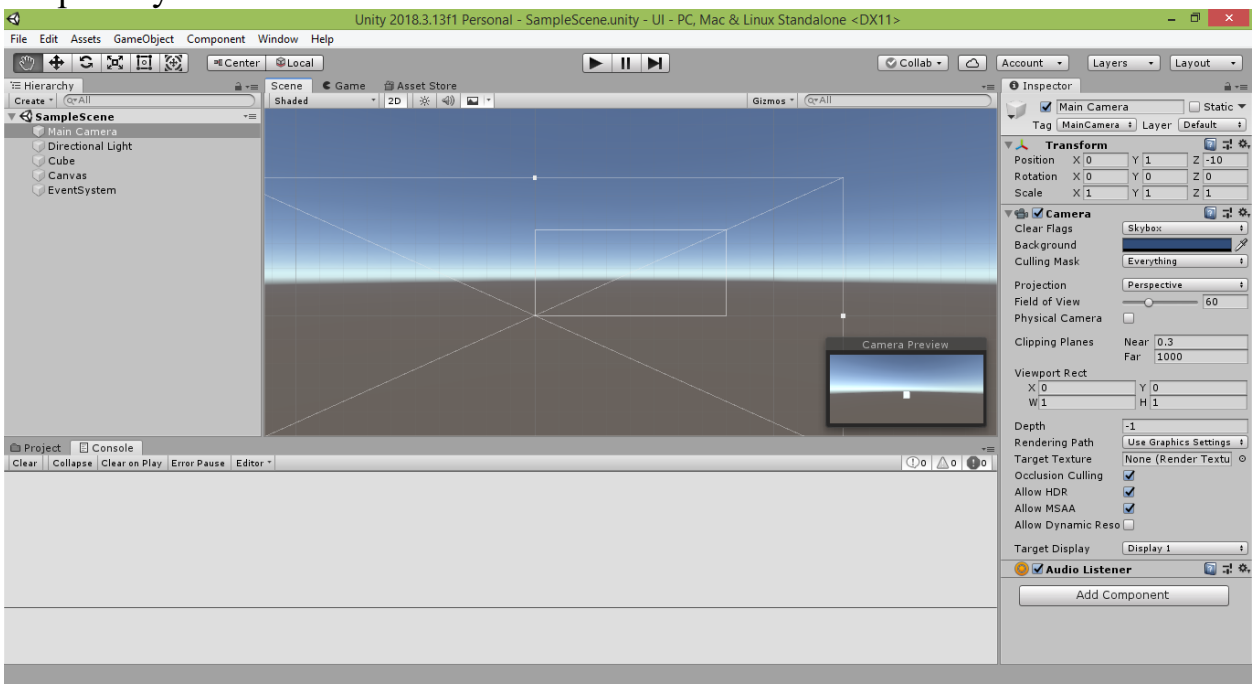
2) «Screen Space – Camera» («Экранное пространство – камера») означает, что все элементы управления вашим приложением будут размещены за сценой, независимо от дальности ей расположения от наблюдателя. В этом режиме вы тоже не сможете изменить размер полотна – оно автоматически растянется на весь экран при запуске приложения.

3) «World Space» («В мировом пространстве») означает, что ваше полотно будет подобно обычному игровому объекту «Panel» («Панель») – вы сможете придать ему нужные вам размеры, переместить в нужную вам точку сцены, повернуть на нужный вам угол. Полотно в этом режиме может как перекрывать другие объекты сцены, так и само перекрываться ими – всё будет зависеть от их взаимного расположения. Кроме того, пользователь увидит только ту часть полотна, которая попадёт в область видимости камеры. Данный режим удобен, когда нужно использовать полотно в качестве одного из объектов созданного вами мира (например, это может быть панель с полоской индикации здоровья над головой персонажа игры).

На текущий момент наше полотно ещё не заполнено. Поэтому пока оставим выбранным пункт «Screen Space – Overlay». Это значение свойства «Render Mode» уже выставлено по умолчанию при создании полотна.

Теперь в строке прямо над рабочим полем сцены щёлкните на кнопке с надписью «2D» 

. Обзор сцены изменится так, что вы будете смотреть прямо на полотно, находясь на продолжении синей оси, вдоль которой отсчитывается координата z объектов сцены. В режиме 2D мы и будем создавать интерфейс нашего приложения. Также, если в окне Hierarchy всё ещё выделена строка «Main Camera», вы увидите на сцене область её охвата камерой, схематично очерченную белыми линиями.

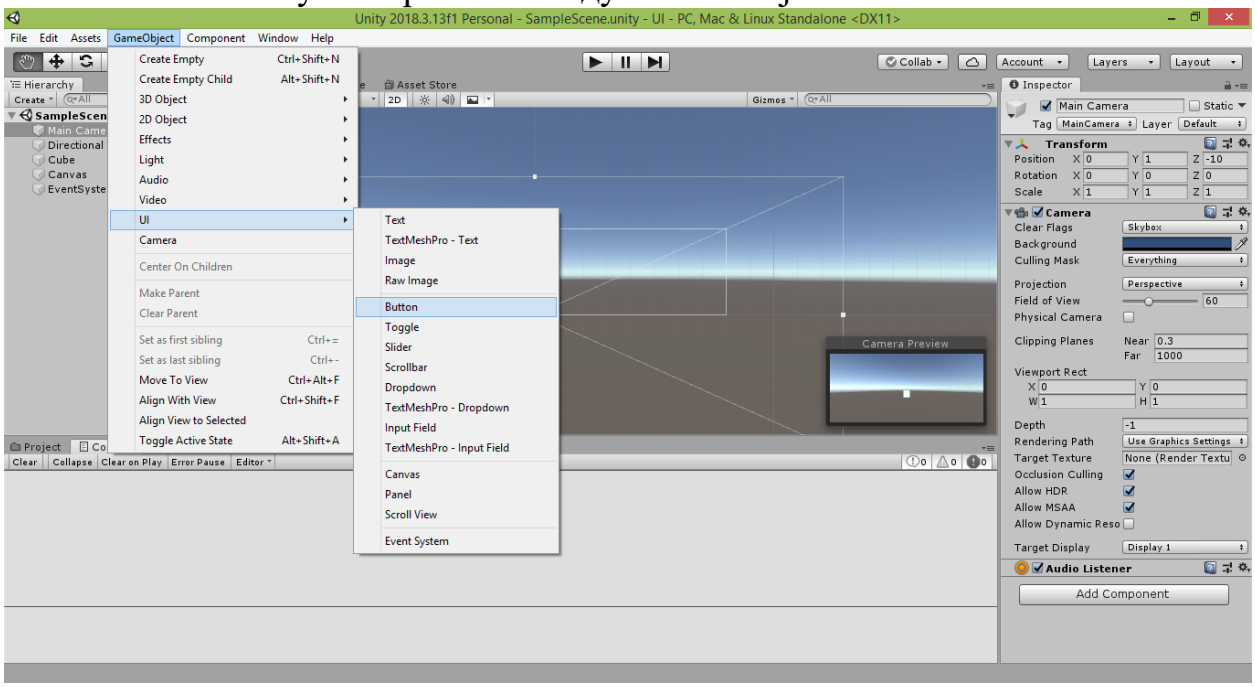


Также следует сказать несколько слов об объекте «EventSystem» («Система событий»), появившемся в окне Hierarchy при создании полотна «Canvas». Данный объект отвечает за отслеживание событий, происходящих с объектами сцены. Например, он позволяет вовремя отреагировать на нажатие пользователем переключателя или кнопки, на изменение положения ползунка или выбора одного из пунктов выпадающего списка. Именно с помощью этого объекта наше приложение сможет реагировать на действия пользователя. Поэтому удалять его не следует. Более того, объекты типа «EventSystem» можно добавлять на сцену вручную через меню «GameObject» (но сейчас нам это не требуется).

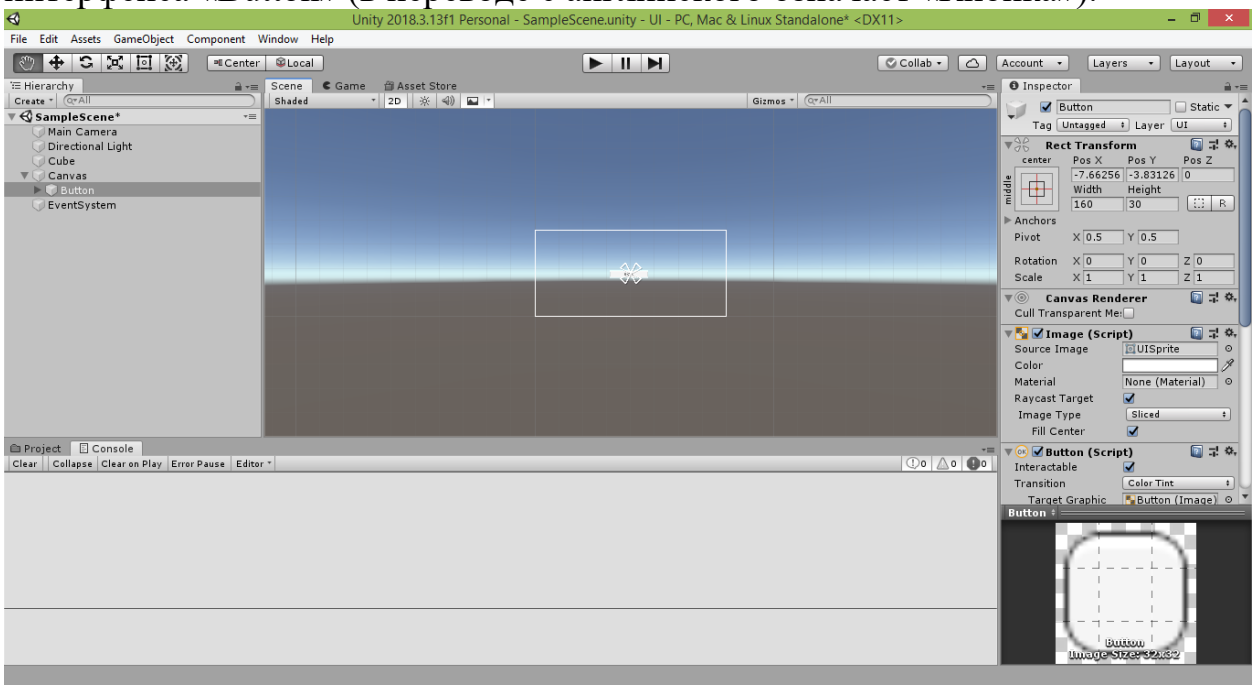
Теперь перейдём к самому интересному – управлению нашим кубиком при помощи элементов пользовательского интерфейса.

4.3. Создание и настройка объекта «Button» («Кнопка»)

В меню Unity выберите команду «GameObject → UI → Button».



В результате на вашей сцене появится объект пользовательского интерфейса «Button» (в переводе с английского означает «Кнопка»).



Кнопка является наиболее популярным элементом пользовательского интерфейса компьютерных программ, поскольку с её помощью пользователь может заставить приложение выполнить определённую команду простым щелчком мышки.

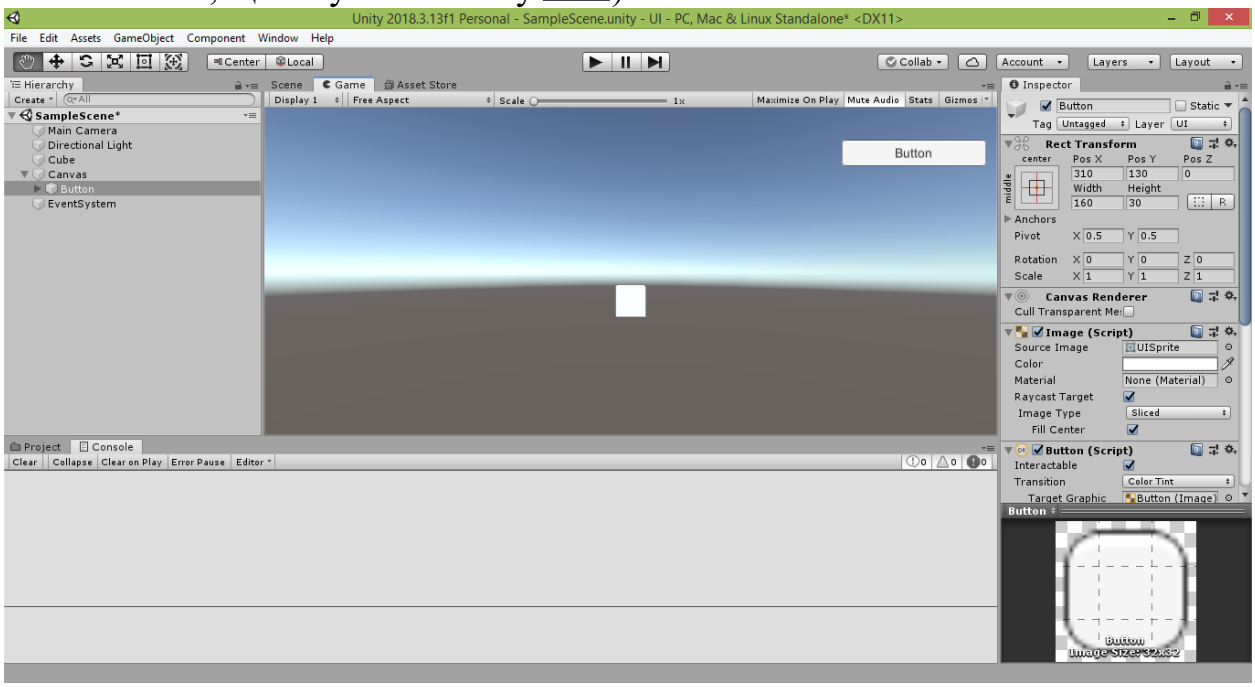
Созданная нами кнопка будет изменять цвет куба на красный.

Прежде чем приступить к написанию программы для кнопки, настроим несколько её свойств.

Для этого выделим нашу кнопку. Сделать это можно щелчком по ней на сцене, но удобнее всего воспользоваться списком объектов в окне Hierarchy. При создании кнопки в окне Hierarchy появилась строка «Button», подчинённая строке «Canvas». И это неудивительно, поскольку все элементы пользовательского интерфейса должны располагаться на площадке-полотне. Щёлкните на строке «Button» в окне Hierarchy. Справа в окне Inspector появится список свойств выбранной нами кнопки.

Измените значение свойства «Pos X» на 310, а значение свойства «Pos Y» – на 130. Мы задали новые координаты расположения нашей кнопки. Перейдите на вкладку «Game» и проверьте, видна ли ваша кнопка в правом верхнем углу игрового окна.

Если этого не произошло, значит, кнопка оказалась за пределами рамки полотна «Canvas». В этом случае подберите значения свойств «Pos X» и «Pos Y», чтобы кнопка оказалась в пределах рамки полотна «Canvas» (для перемещения кнопки по полотну можно также воспользоваться инструментом «Move Tool», щёлкнув на кнопку ).

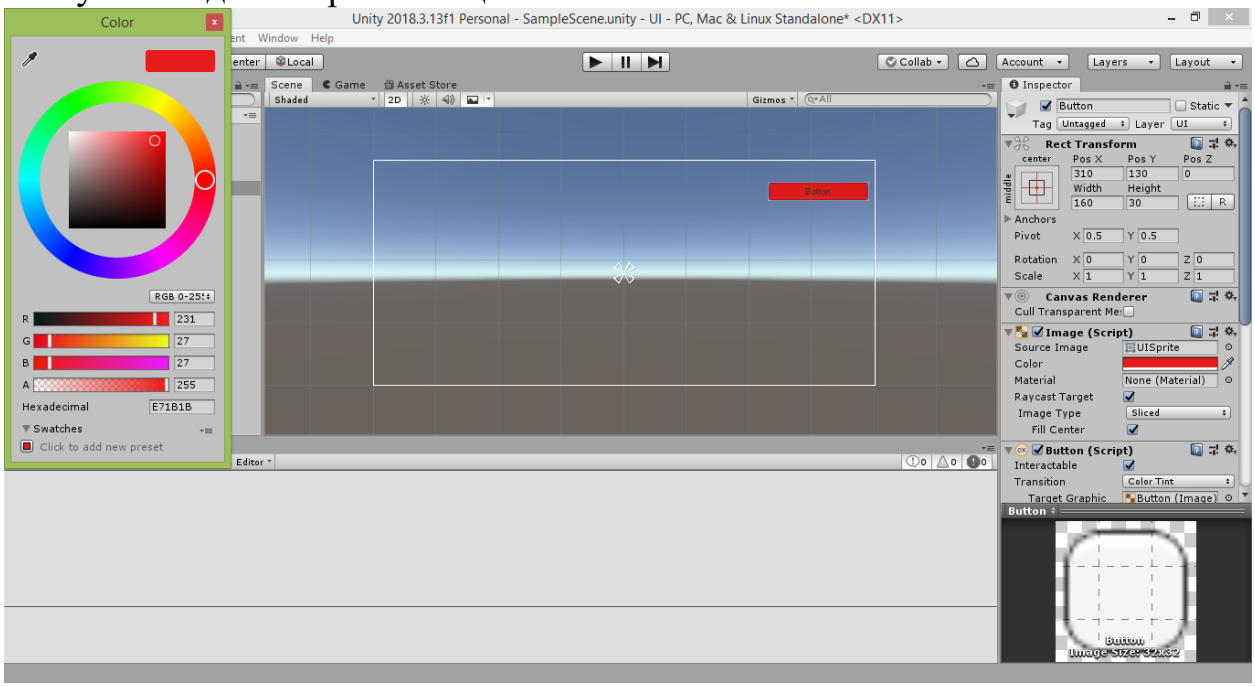


Если запустить проект в игровом режиме, нажав в самом верху кнопку «Play» со значком чёрного треугольника, указывающего вправо, то вы даже сможете пощёлкать на созданную вами кнопку (хотя пока это и не будет приводить к какому-нибудь результату).

Выйдите из игрового режима и вернитесь на вкладку «Scene» («Сцена»).

Щёлкните в окне Inspector на белом прямоугольнике в свойстве «Color» («Цвет»). В результате слева появится окно с палитрой выбора цвета. Щёлкая мышкой по окружности, вы можете выбрать нужный вам цвет, а щёлкая в области расположенного в центре квадрата – задать насыщенность

выбранного цвета (я выбрал достаточно насыщенный фиолетовый цвет). Задайте кнопке красный цвет. Если кнопка выглядит мелкой, приблизьте к ней точку наблюдения при помощи колёсика мышки.



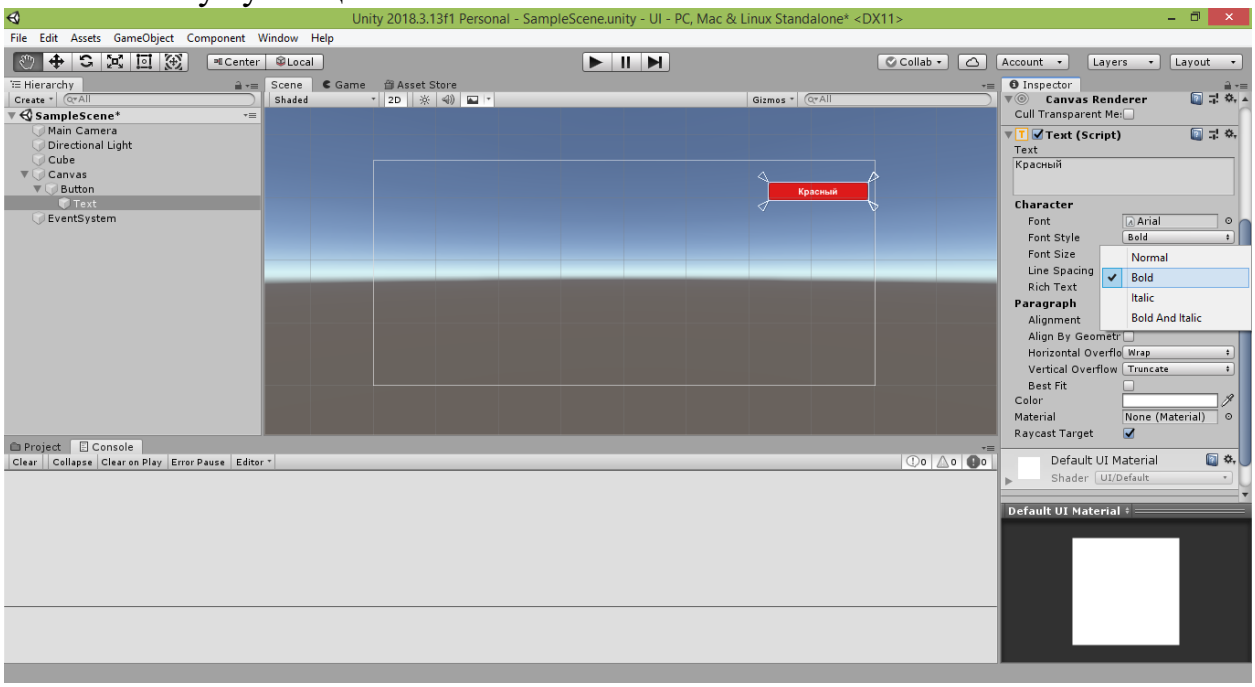
Не пугайтесь, если на вкладке «Scene» вы не увидели ранее созданный куб. Он никуда не пропал – о его наличии свидетельствует строка «Cube» в окне Hierarchy. Дело в том, что в режиме представления «Screen Space – Overlay» полотно «Canvas» со всеми размещёнными на нём элементами управления будет автоматически наложено на вид с камеры. Поэтому на сцене оно может располагаться отдельно от других объектов. Достаточно будет щёлкнуть строку «Main Camera», чтобы увидеть куб в окне «Camera Preview», или перейти на вкладку «Game». Главное условие видимости куба – он должен находиться в поле обзора камеры.

Теперь в окне Hierarchy щёлкните на треугольник слева от строки «Button». Строка «Button» раскроется и ниже появится подчинённая ей строка «Text». Дело в том, что многие элементы управления (даже такие простые по виду, как кнопка) в действительности являются составными. В частности, кнопка включает в себя компонент класса «Text», который имеет набор свойств, характерных для текстовой надписи. Воспользуемся им и первым делом изменим текст подписи кнопки. Щёлкните на строке «Text» кнопки и найдите справа в окне Inspector в группе «Text (Script)» одноимённое поле «Text», в котором написано слово «Button». Измените его на слово «Красный». Вы сразу увидите, как такое же слово появится в качестве подписи у вашей кнопки.

Однако чёрный текст на красном фоне выглядит не очень контрастно. Найдите в окне Inspector в группе «Text (Script)» свойство «Color» и, щёлкнув на его тёмном прямоугольнике, выберите белый цвет в появившемся слева окне с палитрой. В результате цвет подписи кнопки станет белым.

Чтобы добиться ещё большего контраста, откройте в группе «Text (Script)» выпадающий список у свойства «Font Style» («Стиль шрифта») и

выберите пункт «Bold» («Полужирный»). В результате буквы у подписи кнопки станут утолщёнными.



В дополнение к созданной красной кнопке нам потребуются ещё две – зелёного и синего цветов. Добавить и настроить их можно аналогичным способом. Однако в окне Hierarchy они все будут обозначены как «Button». Чтобы в дальнейшем не путаться в них, мы будем давать каждой кнопке уникальное имя. Переименуем строку «Button», соответствующую красной кнопке, в «ButtonRed». Для этого щёлкните на этой строке в окне Hierarchy. Далее переименовать её можно несколькими способами:

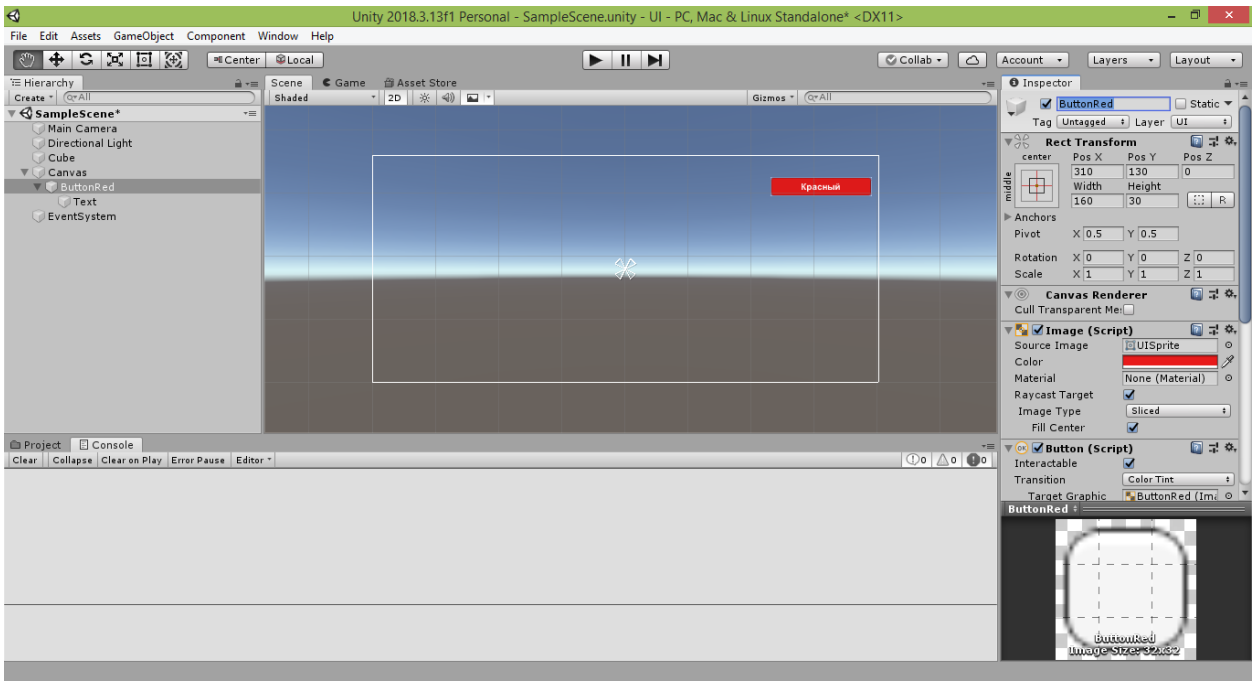
– сделать два отдельных щелчка на выделенной строке;

– нажать после выделения строки клавишу F2;

– щёлкнуть на строке правой кнопкой мыши и выбрать в открывшемся контекстном меню команду «Rename» («Переименовать»);

– зайти в окно Inspector и изменить в самом первом (верхнем) поле текст «Button» на «ButtonRed».

На мой взгляд, последний способ является самым удобным. Однако рекомендую попробовать все варианты переименования. В результате строка «Button» в окне Hierarchy получит новое название «ButtonRed».

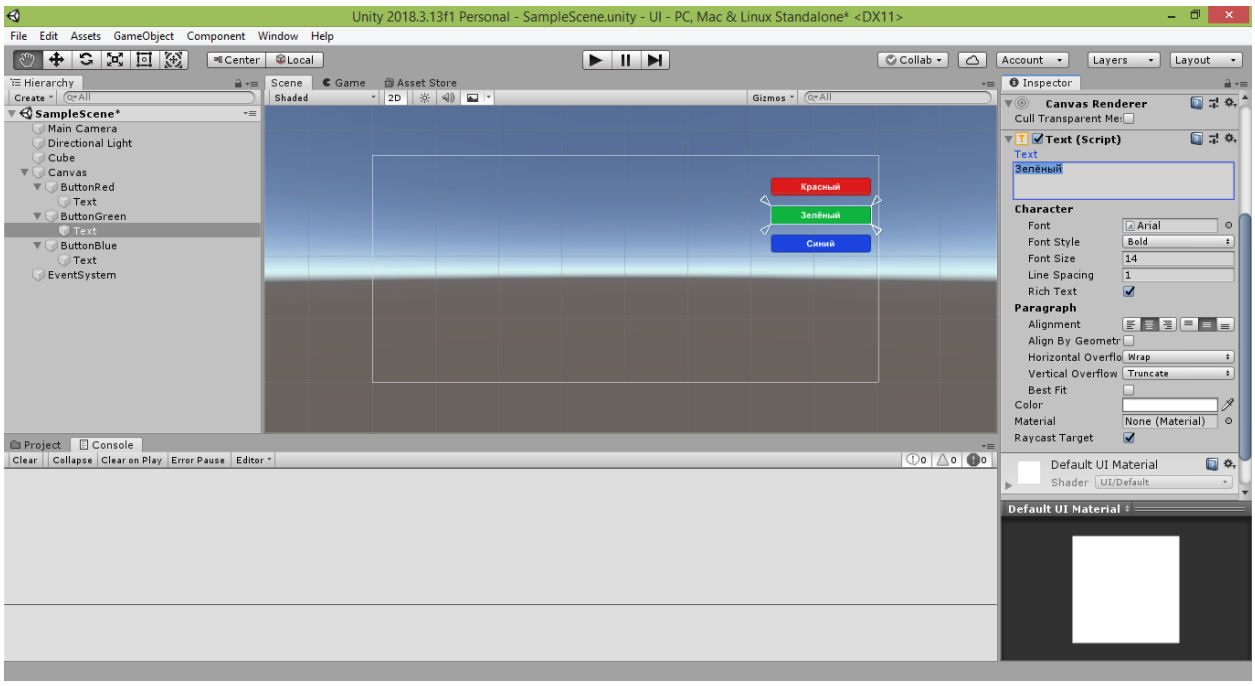


Используя в меню Unity команду «GameObject → UI → Button», добавьте ещё две кнопки. Одной дайте имя «ButtonGreen», другой «ButtonBlue».

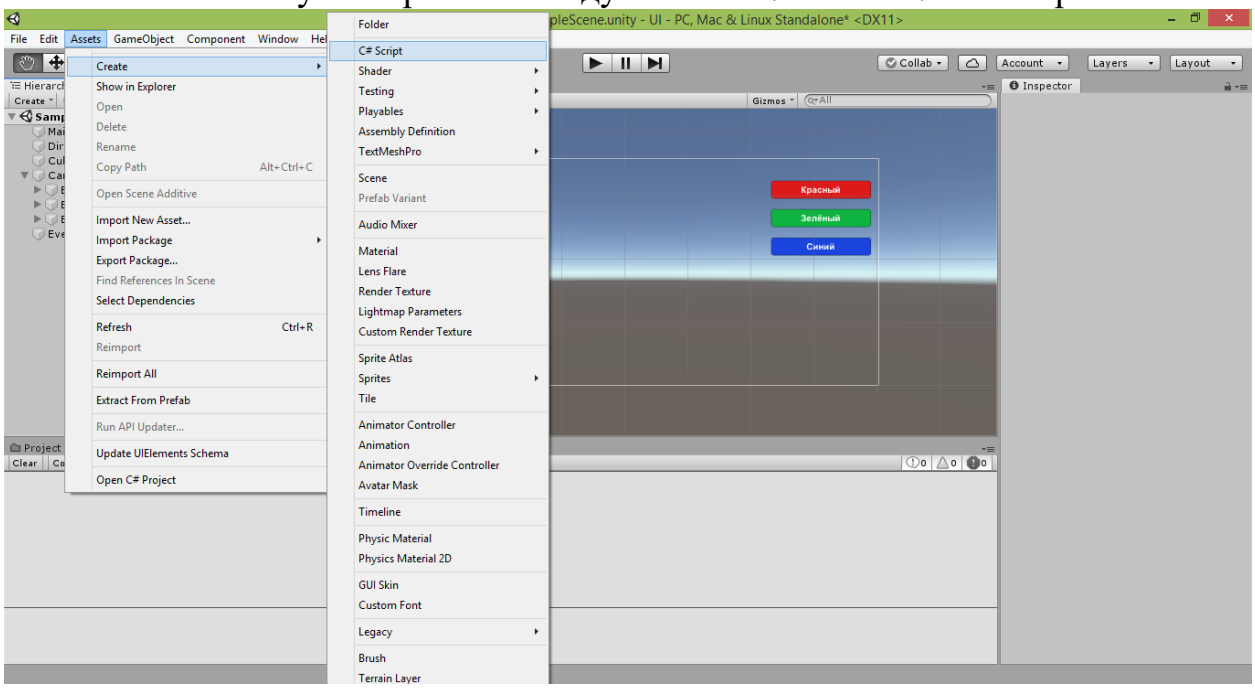
У кнопки «ButtonGreen» измените значение свойства «Pos X» на 310, а значение свойства «Pos Y» – на 85. Свойству «Color» кнопки задайте зелёный цвет. Далее, щёлкнув слева в окне Hierarchy на строку «Text» этой кнопки, задайте справа в окне Inspector свойству «Text» значение «Зелёный», свойству «Color» – белый цвет, полю «Font Style» – значение «Bold».

У кнопки «ButtonBlue» измените значение свойства «Pos X» на 310, а значение свойства «Pos Y» – на 40. Свойству «Color» кнопки задайте синий цвет. Далее, щёлкнув слева в окне Hierarchy на строку «Text» этой кнопки, задайте справа в окне Inspector свойству «Text» значение «Синий», свойству «Color» – белый цвет, полю «Font Style» – значение «Bold».

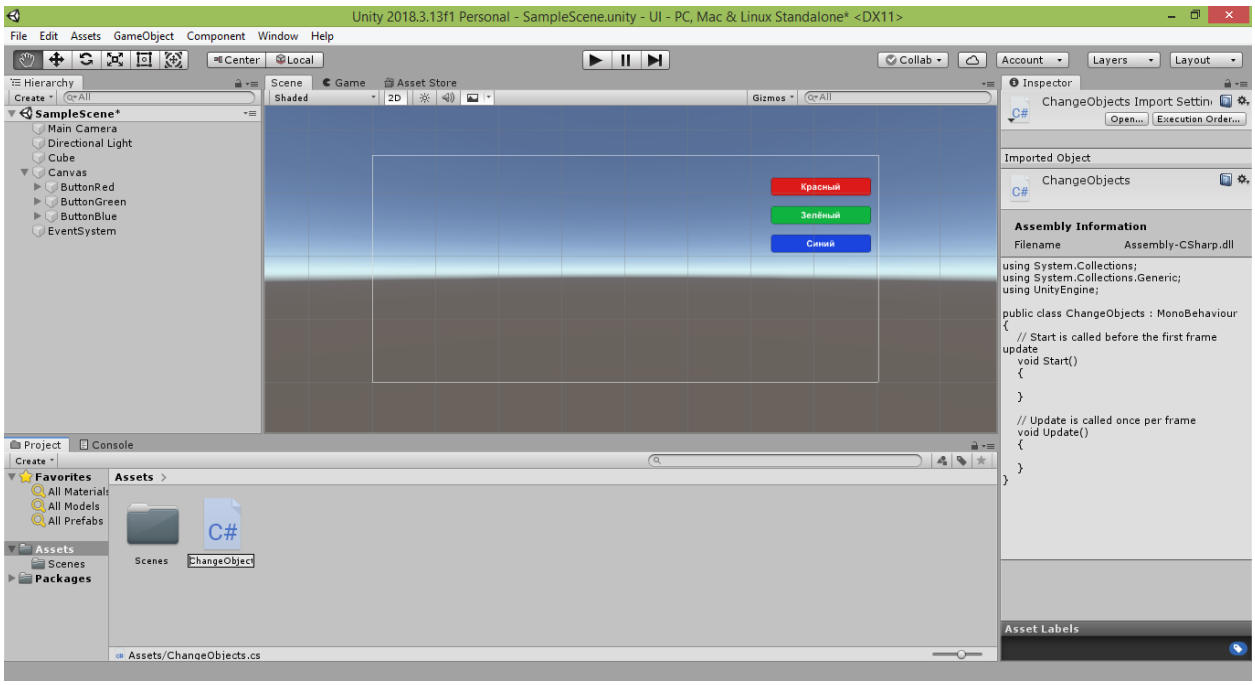
Ниже представлен результат выполнения всех описанных настроек.



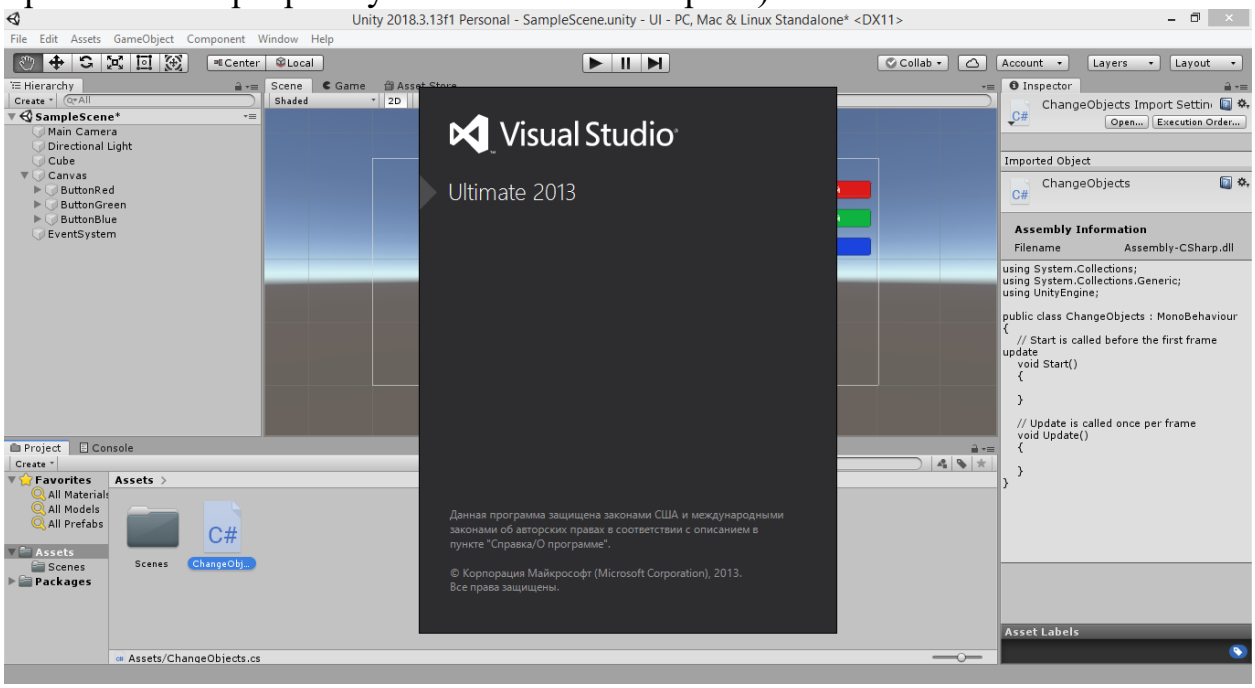
Теперь приступим к программированию созданных кнопок.
Сначала создадим скрипт, в котором и будем программировать.
В меню Unity выберите команду «Assets → Create → C# Script».



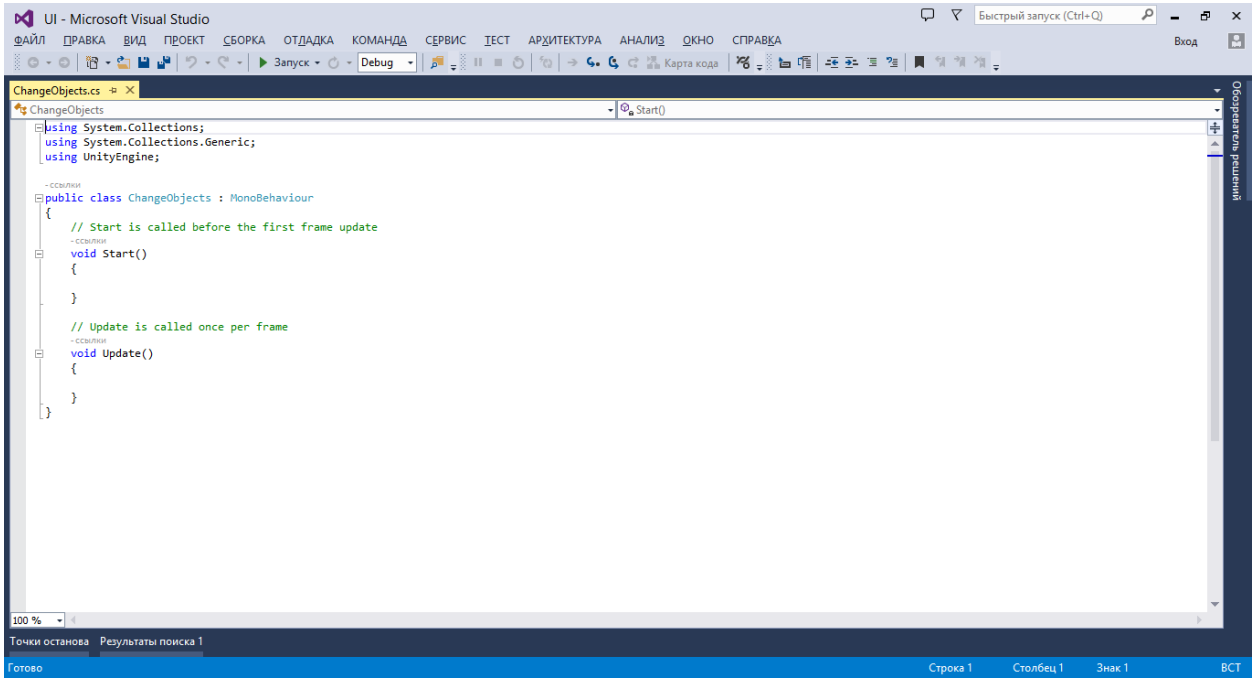
Внизу появится значок файла скрипта. Созданный файл скрипта назовите ChangeObjects (с английского переводится как «Изменить объекты»). Хочу отдельно подчеркнуть, что это имя не является названием какой-либо команды – мы выбрали его сами для нашего скрипта.



Закончив переименование, щёлкните два раза на значке файла скрипта или нажмите клавишу «Enter». Запустится редактор Visual Studio (если вдруг он у вас не установлен, выберите в открывшемся списке доступных приложений программу «Блокнот» / «Notepad»).



После открытия файла скрипта, вы увидите его содержимое.



По умолчанию содержимое нового скрипта выглядит следующим образом:

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

public class ChangeObjects : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Первое, что нам потребуется сделать, – это создать открытую переменную, через которую мы сможем управлять кубом на нашей сцене. Сразу подчеркну, что переменные в языке C# можно условно разделить на две категории: открытые (`public`) и скрытые (`private`).

Предположим, в программе вы решили объявить числовую переменную `i`:

```
public class ChangeObjects : MonoBehaviour
{
    int i = 7;
```

Фактически программа считает, что вы написали следующее:

```
public class ChangeObjects : MonoBehaviour
{
    private int i = 7;
```

Это означает, что вы создали скрытую (`private`) переменную, которая видна только внутри класса `ChangeObjects` . То есть только этот класс является владельцем переменной `i` и только внутри него она может быть изменена.

Теперь предположим, что вы объявили переменную `i` по-другому:

```
public class ChangeObjects : MonoBehaviour
{
    public int i = 7;
```

В этом случае вы создали открытую (`public`) переменную, которая видна за пределами класса `ChangeObjects` и может быть изменена извне. То есть теперь доступом к считыванию и записи информации в переменную `i` обладает любой сторонний класс.

Ключевые слова `private` и `public` называются модификаторами доступа. Они позволяют задавать разные права доступа к различным элементам класса (не только к переменным, но и к методам класса).

Приведу наглядный пример. Предположим, вы решили спроектировать автомобиль. Разумеется, у автомобиля будет двигатель и будут элементы управления этим двигателем (педали газа и тормоза, коробка передач). Но вы не хотите, чтобы в процессе работы двигателя автомобиля кто-нибудь залез в него, поскольку человек в этом случае получит травмы. Поэтому вы прячете двигатель внутри корпуса автомобиля, а доступ к прямой работе с двигателем (например, для его починки) будет иметь только ваш завод-изготовитель. То есть фактически вы делаете двигатель скрытым (`private`). В то же время, владелец автомобиля должен иметь возможность управлять работой двигателя: заводить, глушить, увеличивать или снижать обороты. Для этого в салоне проектируемого автомобиля вы предусматриваете ключ зажигания (для запуска и глушения двигателя), выводите рычаг коробки передач, педали газа и тормоза (чтобы изменять скорость оборотов двигателя). То есть фактически вы создаёте открытые (`public`) элементы управления работой рассматриваемого нами механизма. В результате никто посторонний не сможет напрямую вмешаться во внутреннюю работу автомобиля, однако любой человек сможет управлять им, пользуясь теми устройствами в салоне автомобиля, которые специально для этого были созданы.

Аналогичную работу по разграничению доступа мы сейчас проделываем при проектировании программы на языке C#. Поэтому давайте разберёмся, какое отношение сказанное имеет к теме пользовательского интерфейса Unity. Первой строкой в классе нашего скрипта объявим `private`-переменную `cube1`, через которую мы будем работать с нашим кубом:

```
public class ChangeObjects : MonoBehaviour
```

```
{
    private GameObject cube1;
```

`cube1` – это придуманное нами имя переменной (цифра «1» в конце добавлена на случай, если мы захотим расширить сцену, добавив на неё ещё кубы).

`GameObject` – класс из пространства имён `UnityEngine`, описывающий в общем виде любой игровой объект сцены (куб, сферу, цилиндр и т.д.).

Сохраним код скрипта, нажав вверху значок квадратной синей дискеты или выбрав в меню команду «Файл → Сохранить ChangeObjects.cs» («File → Save ChangeObjects.cs» в английской версии Visual Studio). Также для сохранения можно использовать комбинацию клавиш CTRL+S.

Не закрывая окно редактора Visual Studio, перейдите обратно в окно Unity.

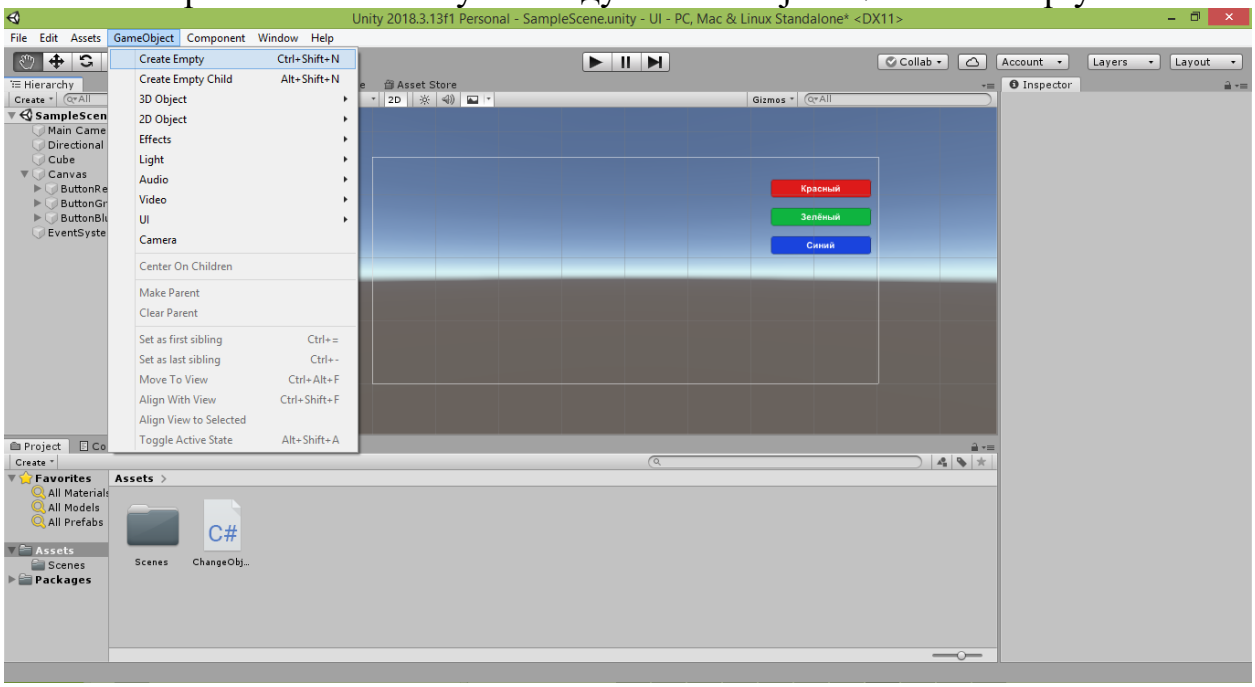
Теперь осталось связать созданный нами скрипт с объектом сцены.

Поскольку у нас на сцене имеется только куб, выбор, казалось бы, очевиден.

Однако в дальнейшем наша сцена пополнится новыми объектами и всеми ими мы будем управлять из нашего скрипта. К тому же, нет гарантии, что, привязав скрипт к одному из объектов, мы в дальнейшем не удалим этот объект в процессе редактирования сцены.

Поэтому мы поступим по-другому – свяжем скрипт с пустым игровым объектом.

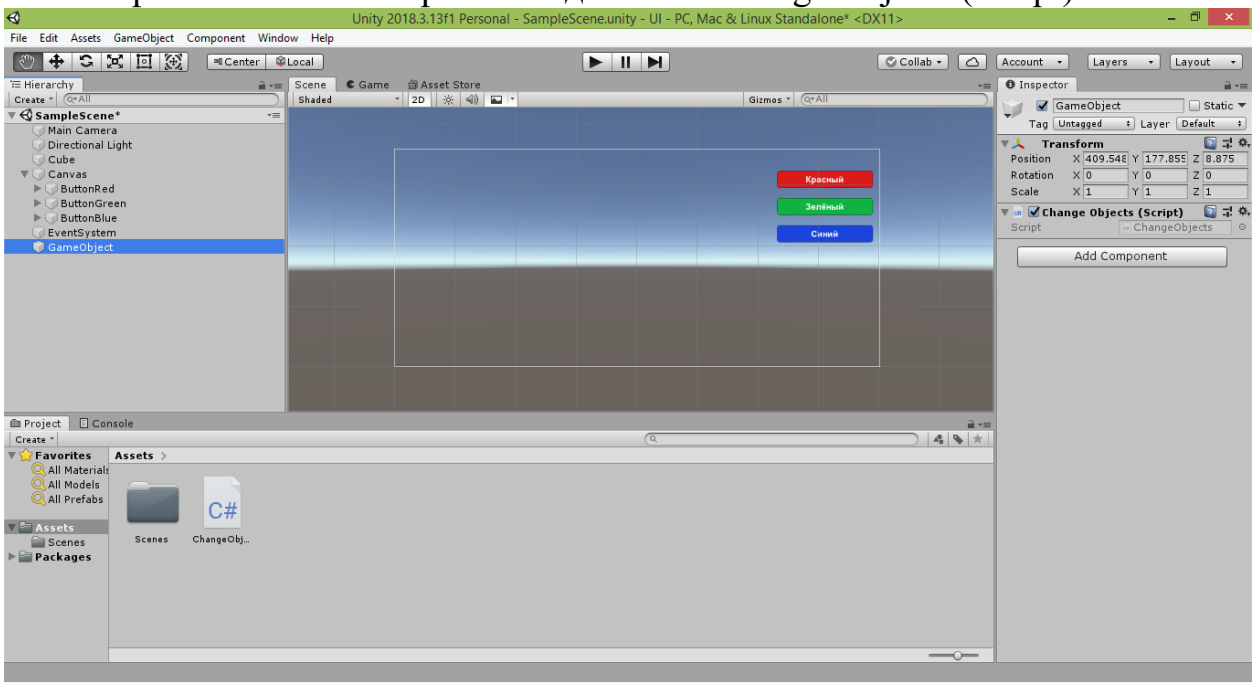
Выберите в меню Unity команду «GameObject → Create Empty».



В окне Hierarchy появится строка «GameObject», указывающая, что наша сцена пополнилась пустым игровым объектом.

Теперь нажмите на эту строку и перетяните скрипт ChangeObjects вправо в окно Inspector, чтобы связать скрипт с пустым объектом. Справа в

окне Inspector появится строка с надписью «Change Objects (Script)».



Как видите, у добавленного компонента есть только недоступное для редактирования поле «Script», в котором указано «ChangeObjects».

Теперь вернёмся в редактор Visual Studio и изменим доступность переменной `cube1` с `private` на `public` :

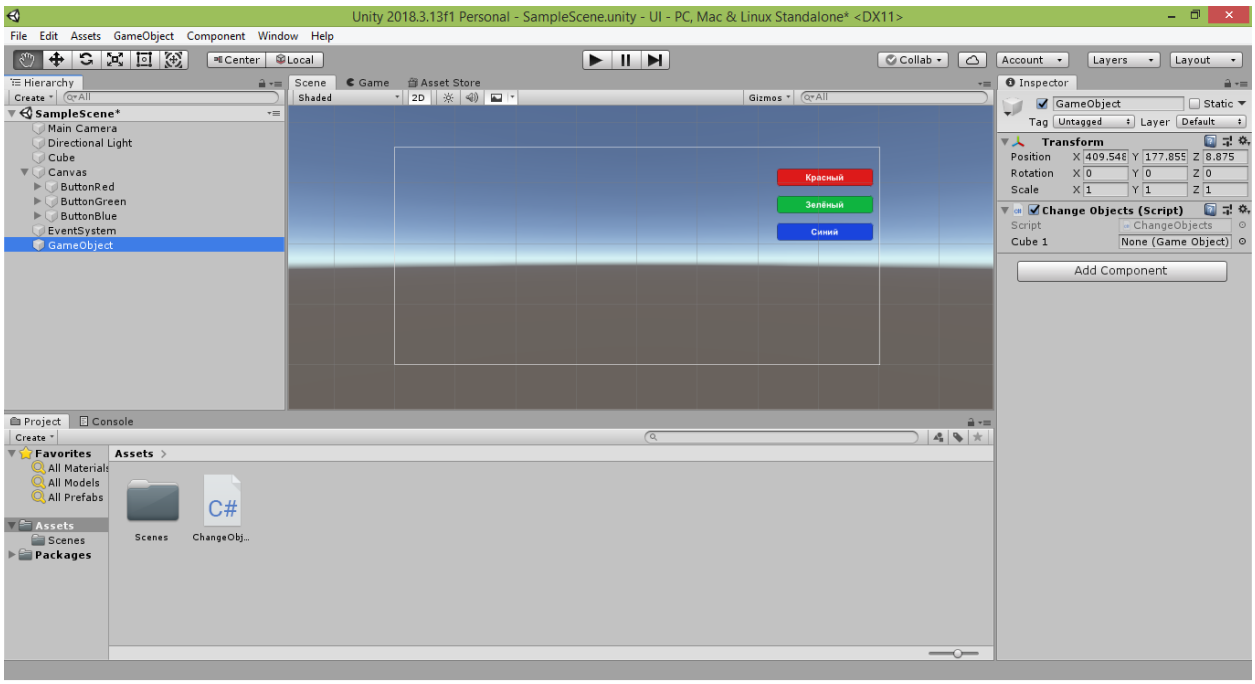
```
public class ChangeObjects : MonoBehaviour
{
    public GameObject cube1;
```

Сохраните код, нажав комбинацию клавиш CTRL+S, и перейдите обратно в окно Unity.

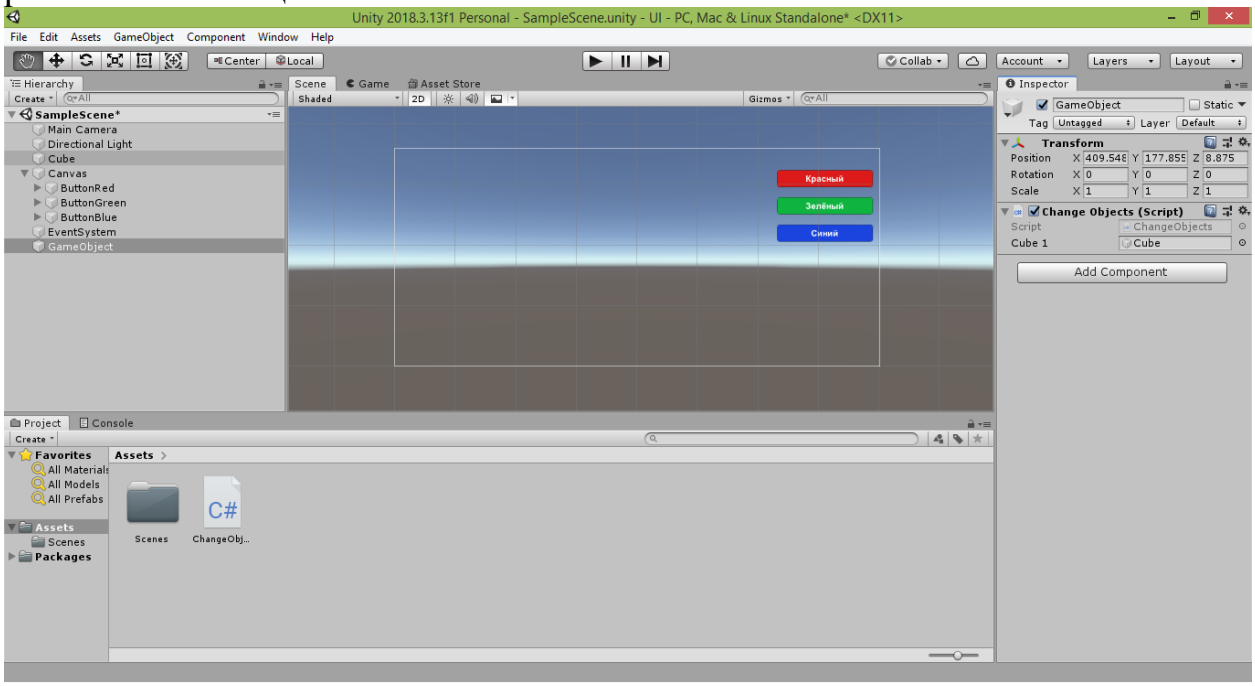
Выделив в окне Hierarchy строку «GameObject», посмотрите, что изменилось в окне Inspector (возможно, придётся подождать 5-10 секунд, пока Unity обновит состояние скрипта).

Вы обнаружите, что у пустого объекта появилось новое поле с подписью «Cube 1», в котором указано значение «None (Game Object)» («Отсутствует (игровой объект)»).

Это поле допускается редактировать, и среда Unity ожидает, когда мы укажем объект сцены, с которым хотим связать переменную `cube1` нашего скрипта.



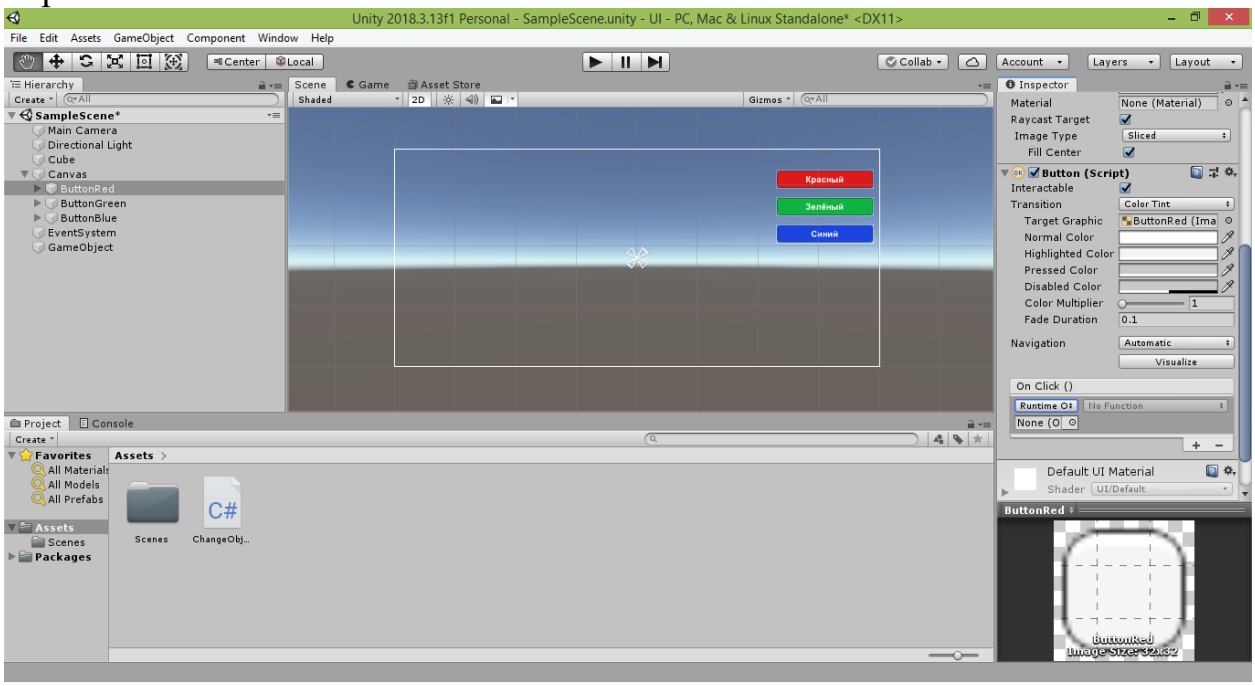
Перетяните в это поле строку «Cube» из окна Hierarchy. В результате открытая переменная `cube1` окажется связанной с кубом, который расположен на сцене.



Теперь выделите строку «ButtonRed» в окне Hierarchy и справа в окне Inspector в группе «Button (Script)» найдите поле «On Click ( )», в котором написано «List is Empty» («Список пуст»).

В это поле мы добавим метод (подпрограмму), который будет вызываться при щелчке на красной кнопке. То есть в момент щелчка кнопки произойдёт стандартное событие «OnClick» (что переводится с английского как «При щелчке») и произойдёт вызов связанной с событием программы, в качестве которой будет выступать написанный нами метод. Мы назовём этот метод «PaintToRed» («Покрасить в красный»).

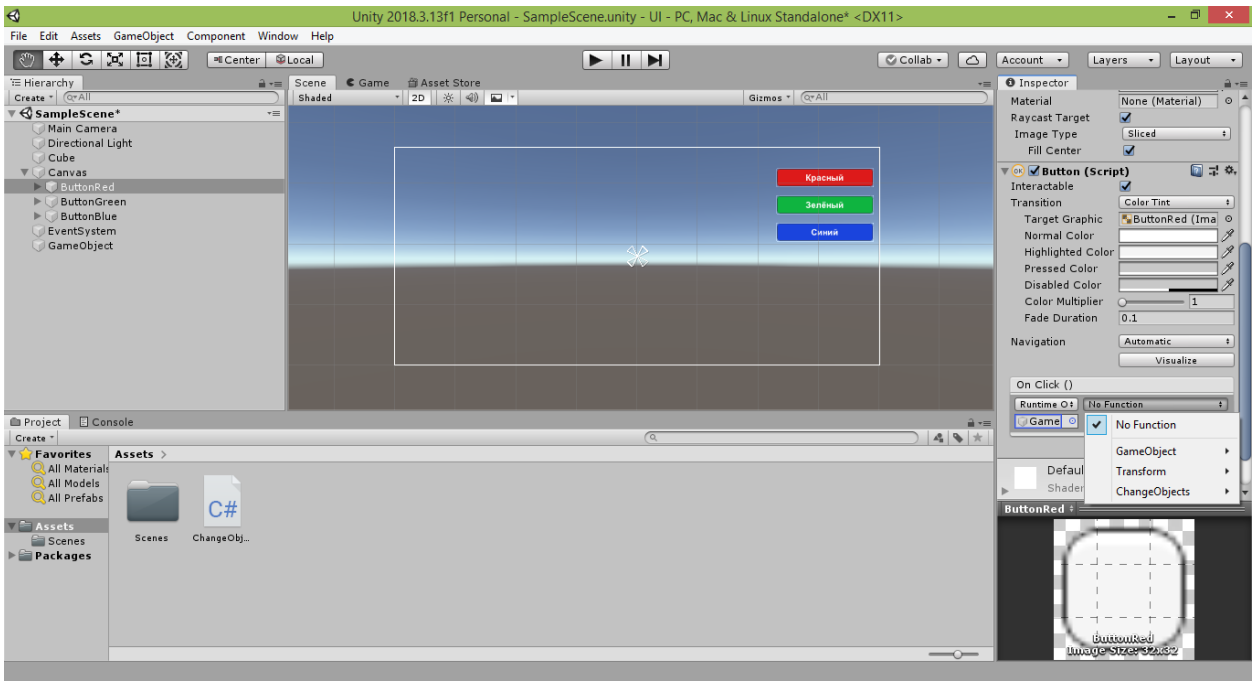
Но прежде чем создать метод, щёлкните на кнопке «+» внизу списка «On Click ( )». В списке появится новый блок, позволяющий задать и настроить обработчик события.



Таких блоков можно добавить сколько угодно и все указанные в них программы-обработчики будут вызываться при наступлении события, указанного в заголовке списка (для кнопки это событие «OnClick» – «При щелчке»). Нам достаточно одного обработчика, поэтому приступим к настройке добавленного блока.

Прежде всего требуется указать объект, к которому привязан скрипт с нужной нам программой (как вы помните, наш скрипт мы привязывали к пустому объекту с именем «GameObject»). Сейчас поле привязки объекта в блоке события содержит надпись «None (Object)» («Отсутствует (объект)»). Потяните строку «GameObject» из окна Hierarchy в это поле. Оказавшись над полем, курсор мыши изменится – у него появится значок «+». Теперь отпустите перетягиваемую строку. В поле появится новая надпись: «GameObject». При этом станет доступен для изменения выпадающий список с надписью «No Function» («Отсутствует функция»), расположенный рядом с полем.

Если раскрыть этот список, то в нём вы увидите группы (подменю) «GameObject», «Transform» и «ChangeObjects». Каждая группа соответствует классу, связанному с объектом.



Класс «GameObject» содержит описание свойств и методов, общих для всех объектов сцены. Если вы помните, ранее объявленной переменной `cube1` мы как раз задали тип `GameObject` , поскольку куб тоже является игровым объектом и описывается этим классом.

Класс «Transform» уже знаком нам. Он содержит описание свойств и методов, необходимых для изменения (трансформирования) объекта – перемещения, масштабирования, поворота и т.д.

Класс «ChangeObjects» – это класс, описанный в созданном нами скрипте. В него мы и добавим сейчас метод, обрабатывающий событие нажатия кнопки.

Если посмотреть состав пунктов в каком-либо из трёх подменю, вы увидите в первых строчках список содержащихся в классе переменных (у них указан тип, например, `int` , `bool` , `string` , `Transform` и т.д.), а далее – список описанных в классе методов (они имеют скобки после названия, в которых указан допустимый тип параметра, влияющего на результат работы данного метода).

Переменные и методы, которые видны в любом подменю, являются открытыми, то есть они описаны с использованием модификатора доступа `public` . Кроме них, в указанных классах, скорее всего, также содержатся скрытые переменные и методы, описанные с использованием модификатора доступа `private` (или они вообще описаны без указания модификатора доступа, что также сделает их скрытыми). Скрытые переменные и методы не отображаются в подменю, поскольку они предназначены для обеспечения внутренней работы класса, а не для прямого вызова по желанию пользователя.

Теперь вернитесь в редактор Visual Studio и добавьте открытый метод «PaintToRed» (например, сразу после стандартного метода «Update»):

```
    void Update()
    {
```

```
    }

    public void PaintToRed()
    {
        cube1.GetComponent<Renderer>().material.color = Color.red;
    }

}
```

Тип `void` означает, что наш метод не будет выдавать какое-либо значение после завершения своей работы, а модификатор доступа `public` – что метод будет виден извне (например, в среде Unity).

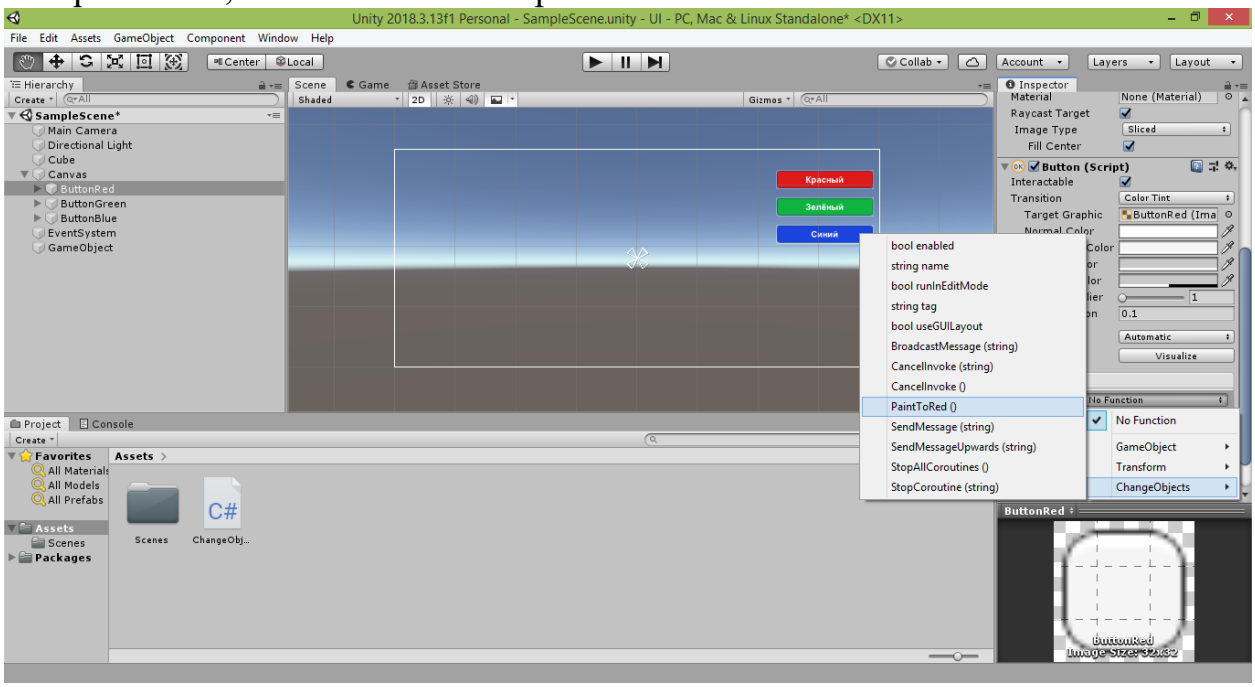
В созданном нами методе содержится всего одна строка. В этой строке у объекта, на который ссылается переменная `cube1` , мы при помощи команды `GetComponent` получаем компонент класса `Renderer` , отвечающий за представление (оформление) объекта на сцене. В частности, в этом классе объявлена составная переменная `material`, позволяющая настроить характеристики материала объекта. Одной из таких характеристик является свойство `color`. Этому свойству мы и присваиваем значение `Color.red` . `Color` – это структура из пространства имён `UnityEngine` (подключенного в самом начале нашего скрипта командой `using`). Данная структура содержит список названий базовых цветов. Мы выбрали красный цвет, указав после точки название `red` (если хотите увидеть весь список доступных цветов, установите курсор сразу после точки и нажмите комбинацию клавиш CTRL+Пробел).

Важное замечание. Обращайте внимание на заглавные и строчные буквы, поскольку в языке C# учитывается регистр символов. То есть если вы напишите `Color.Red` , такая запись даст ошибку в коде, поскольку красному цвету соответствует название `red` , начинающееся со строчной буквы. Чтобы узнать, как правильно пишется название команды или переменной, можно воспользоваться выпадающим списком, открывающимся комбинацией клавиш CTRL+Пробел. Также ориентируйтесь на приводимый мной код – он проверен в работе и не содержит ошибок.

Сохраните изменённый код, нажав комбинацию клавиш CTRL+S, и перейдите обратно в окно Unity.

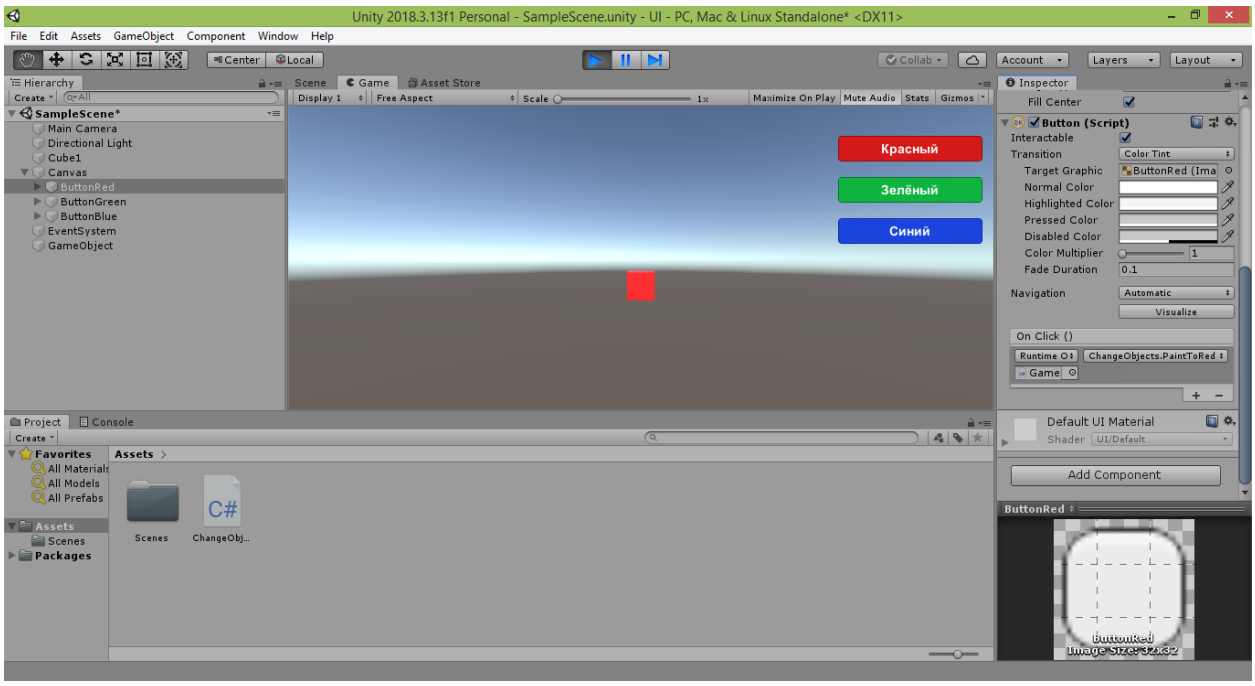
Теперь, если зайти в список «On Click ( )» элемента управления «ButtonRed» и раскрыть в созданном нами блоке список функций, вы увидите в подменю «ChangeObjects» пункт с названием «PaintToRed ( )». Этот пункт соответствует созданному нами открытому методу (пустые скобки означают, что наш метод не использует параметры в своей работе). Если пункт не появился сразу, подождите 5-10 секунд, пока Unity обновит список доступных переменных и методов. В крайнем случае, можно удалить созданный блок,

щёлкнув на кнопке «–» внизу списка «On Click ( )», а затем заново добавить и настроить его, как было описано ранее.



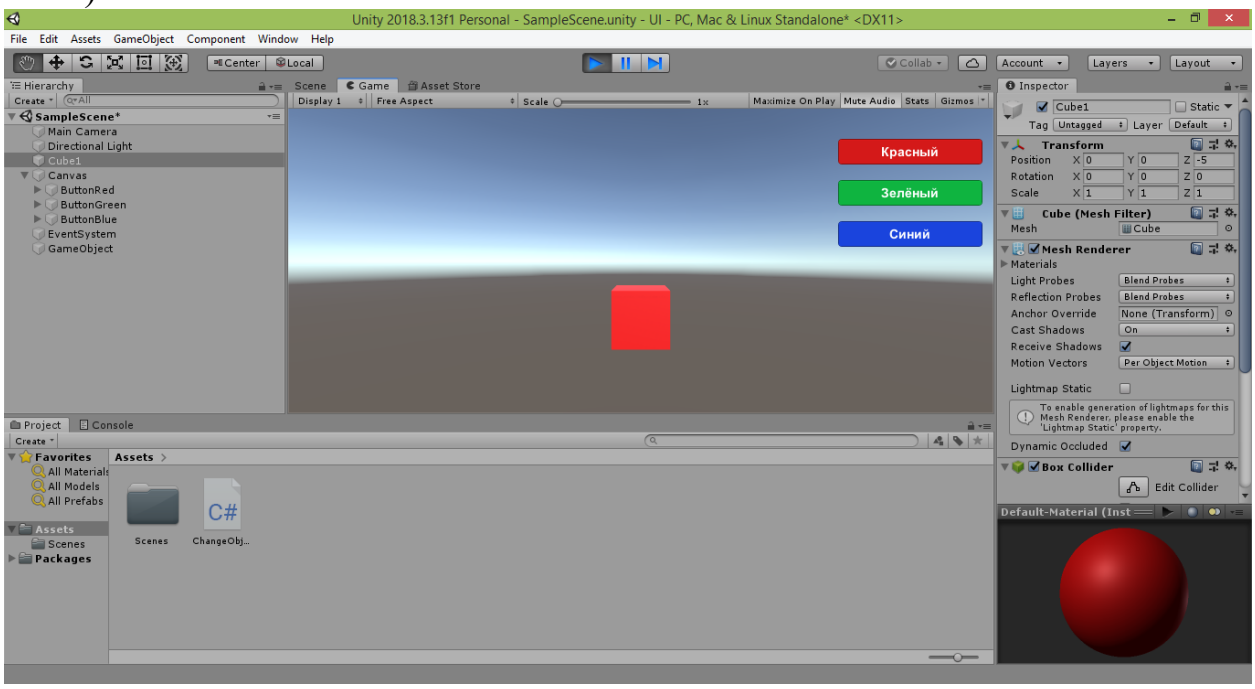
Важное замечание. Если вы указали метод (например, «PaintToRed»), срабатывающий по событию элемента управления, а затем поменяли в коде название или список параметров этого метода (например, переименовали его в «ClickToRed»), то ссылка на данный метод исчезнет после сохранения изменений. Вместо его названия в списке появится надпись: <Missing ChangeObjects.PaintToRed> (что переводится как «Утерян ChangeObjects.PaintToRed»). В этом случае следует снова зайти в подменю со списком переменных и методов и выбрать пункт с обновлённым названием метода (или просто удалить блок обработки события, а затем заново создать его и настроить).

Теперь запустите проект в игровом режиме, нажав в самом верху кнопку «Play» со значком чёрного треугольника, указывающего вправо. Попробуйте щёлкнуть на созданной нами красной кнопке. В результате белый куб в кадре окрасится в красный цвет.



Нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Если куб слишком мелкий, попробуйте в окне Inspector изменить значение свойства «Pos Z» с 0 на -5 (или другое отрицательное число, близкое к этому). Куб переместится в направлении, обратном направлению стрелки синей оси координат (то есть по направлению к пользователю, поскольку при создании проекта камера смотрит на куб в направлении, указываемом синей осью).



Таким образом, нажатие красной кнопки пользователем привело к возникновению события «OnClick». В качестве обработчика события «OnClick» нами был указан метод «PaintToRed» из класса «ChangeObjects», описанного в одноимённом скрипте. Сам скрипт с классом перед этим мы привязали к пустому объекту «GameObject», ссылку на который указали в

блоке обработки события (что позволило затем выбрать метод «PaintToRed» в качестве обработчика).

Если ещё более просто сформулировать принцип настройки элементов пользовательского интерфейса в Unity, то он заключается в том, чтобы прописать в скрипте управляющую программу (метод) и связать его с событием элемента управления. Однако Unity не позволяет напрямую привязать скрипт к событию. Поэтому следует использовать объект сцены в качестве посредника (мы в качестве посредника использовали неотображаемый пустой объект «GameObject», чтобы случайно не удалить его при редактировании рабочего пространства сцены).

Таким образом, скрипт мы привязали к объекту, а объект привязали к событию. И только после этого у нас появилась возможность выбрать для события метод-обработчик из нашего скрипта.

При этом не следует забывать, что, если вы хотите при помощи скрипта управлять объектами сцены, следует объявить в классе соответствующие им `public`-переменные, имеющие тип `GameObject` (так мы поступили, создав переменную `cube1` для управления кубом на сцене). Задать в переменную ссылку на объект сцены можно, перетянув его в поле, появившееся у объекта, с которым связан скрипт (мы перетянули куб в поле «Cube 1», появившееся у пустого объекта «GameObject» после объявления открытой переменной `cube1` и сохранения связанного с ним скрипта).

Итак, красная кнопка работает.

Но у нас пока ещё остались незадействованными две другие кнопки – зелёная и синяя. Перейдите в редактор Visual Studio и добавьте после метода «PaintToRed» ещё два открытых метода: «PaintToGreen» и «PaintToBlue»:

```
    public void PaintToRed()
    {
        cube1.GetComponent<Renderer>().material.color = Color.red;
    }

    public void PaintToGreen()
    {
        cube1.GetComponent<Renderer>().material.color = Color.green;
    }

    public void PaintToBlue()
    {
        cube1.GetComponent<Renderer>().material.color = Color.blue;
    }

}
```

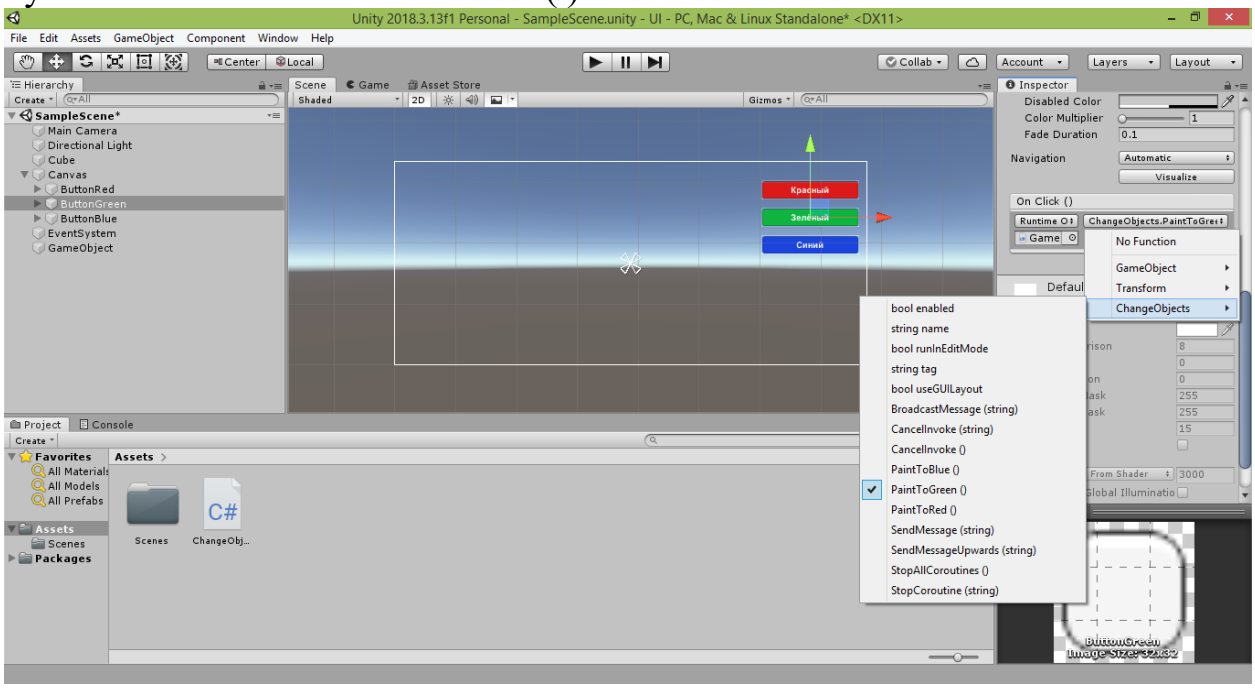
Сохраните код, нажав комбинацию клавиш CTRL+S, и перейдите обратно в Unity.

Теперь осталось выполнить действия, аналогичные тем, что мы делали с кнопкой «ButtonRed». Выделите строку «ButtonGreen» в окне Hierarchy и справа в окне Inspector в группе «Button (Script)» внизу пустого списка «On Click ( )» щёлкните на кнопке «+». В списке появится новый блок для задания обработчика события.

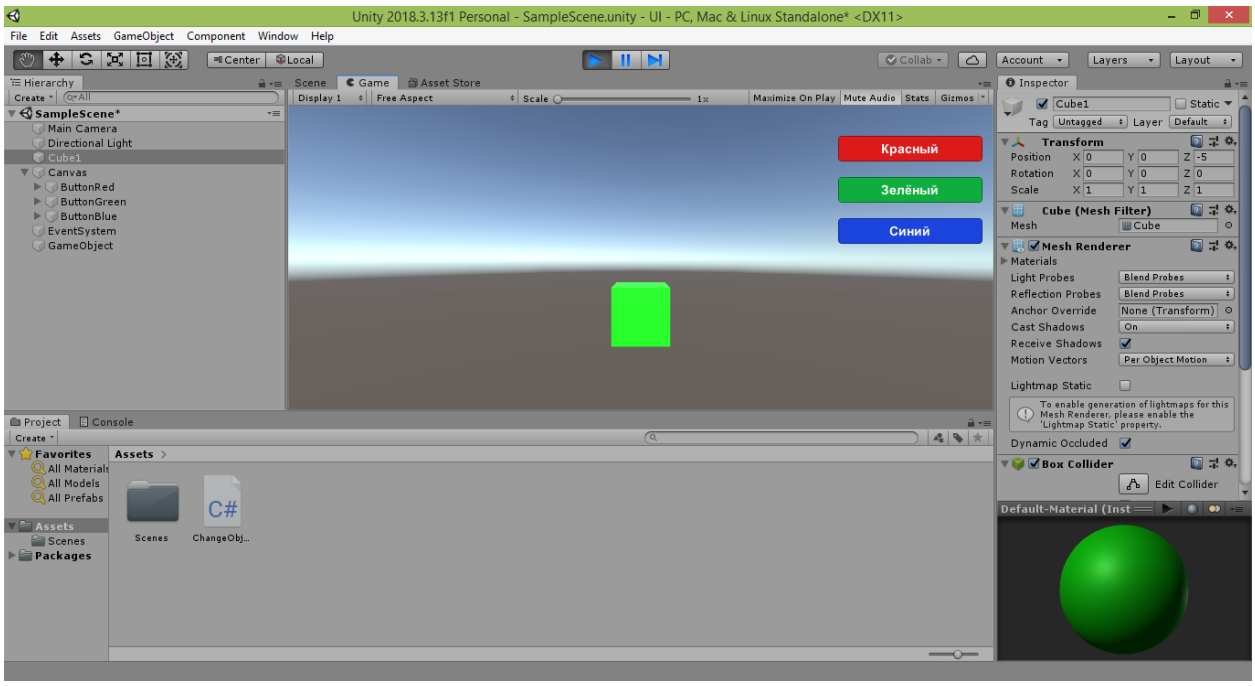
Перетяните строку «GameObject» из окна Hierarchy в поле блока, где написано «None (Object)», и отпустите. В поле появится новая надпись: «GameObject».

При этом станет доступен для изменения выпадающий список с надписью «No Function», расположенный рядом с полем.

Раскройте выпадающий список и в подменю «ChangeObjects» выберите пункт с названием «PaintToGreen ( )».



Запустите проект в игровом режиме, нажав кнопку «Play», и попробуйте щёлкнуть на зелёной кнопке. В результате куб в кадре окрасится в зелёный цвет. Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.



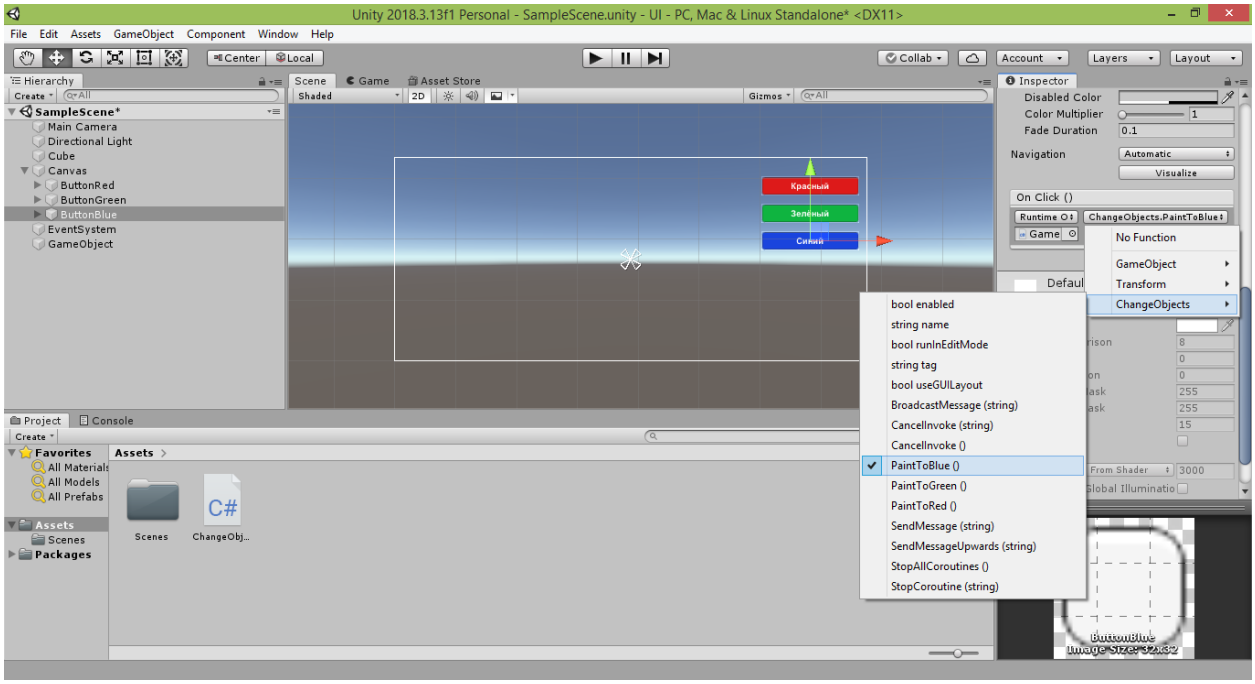
Далее настроим синюю кнопку.

Выделите строку «ButtonBlue» в окне Hierarchy и справа в окне Inspector в группе «Button (Script)» внизу пустого списка «On Click ( )» щёлкните на кнопке «+». В списке появится новый блок для заданий обработчика события.

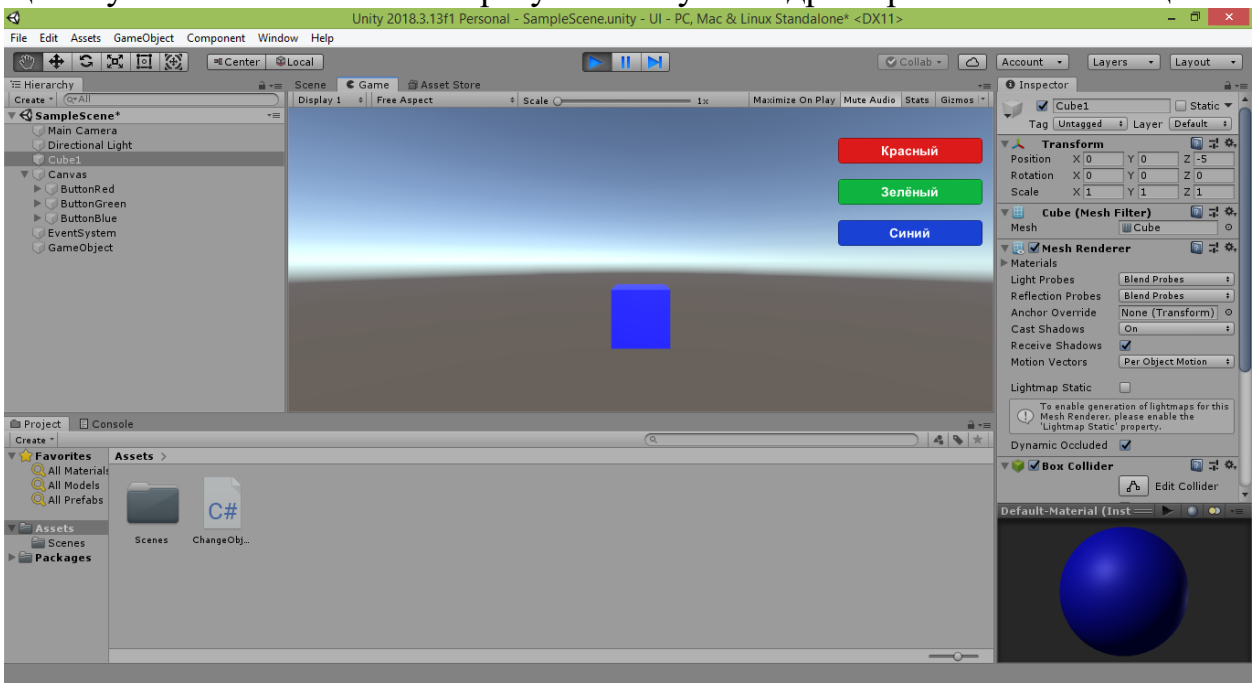
Перетяните строку «GameObject» из окна Hierarchy в поле блока, где написано «None (Object)», и отпустите. В поле появится новая надпись: «GameObject».

При этом станет доступен для изменения выпадающий список с надписью «No Function», расположенный рядом с полем.

Раскройте выпадающий список и в подменю «ChangeObjects» выберите пункт с названием «PaintToBlue ( )».



Запустите проект в игровом режиме, нажав кнопку «Play», и попробуйте щёлкнуть на синей кнопке. В результате куб в кадре окрасится в синий цвет.



Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Отлично! Все три кнопки работают!

С их помощью вы сможете изменять цвет куба на любой из представленных.

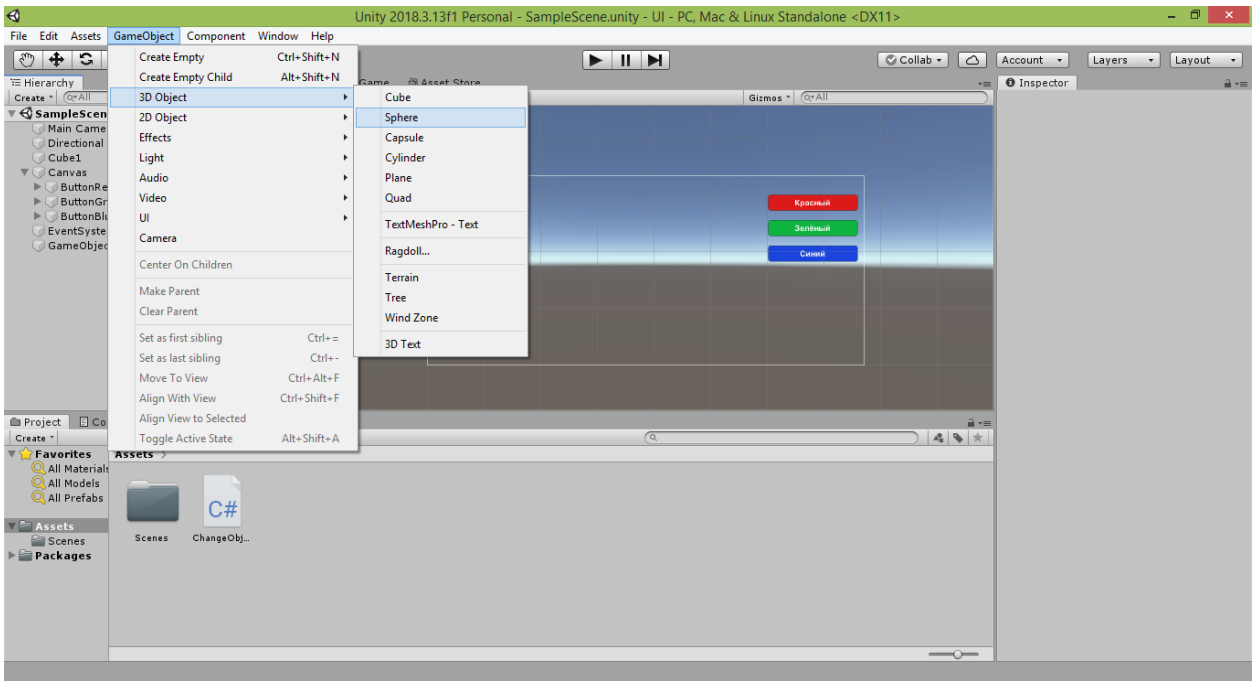
Также вы можете дополнить пользовательский интерфейс вашего приложения кнопками с другими цветами.

Или поменять цвет в методах уже добавленных кнопок (не забывайте только менять подписи и цвета кнопок, чтобы не запутаться в сделанных вами изменениях).

Теперь попробуем при помощи созданного нами пользовательского интерфейса управлять не одним, а уже двумя объектами сцены.

Для этого в дополнение к кубу добавим на сцену сферу.

Выберите в меню Unity команду «GameObject → 3D Object → Sphere».



В результате на сцене появится игровой 3D-объект «Sphere».

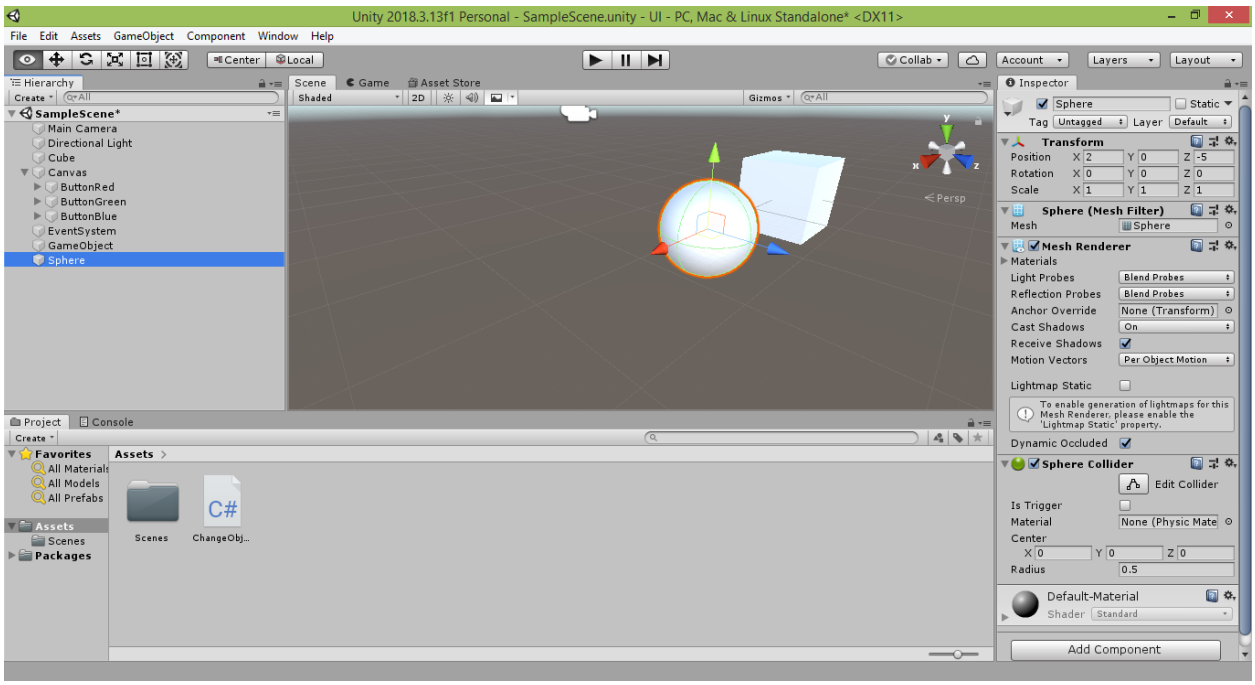
Расположите созданную сферу рядом с кубом. В моём примере куб изначально появился в начале системы координат, а затем я его подвинул на 5 условных делений к наблюдателю вдоль синей оси. Поэтому сейчас куб имеет равные нулю координаты в свойствах «Pos X» и «Pos Y» и равную -5 координату в свойстве «Pos Z».

Сферу я размещу правее куба, для чего сдвину её в положительном направлении красной оси, вдоль которой отсчитывается координата X. Для этого у сферы в окне Inspector я задам значение свойства «Pos X» равным 2.

Значение свойства «Pos Y» я оставлю равным 0, чтобы сфера была на той же высоте, что и куб.

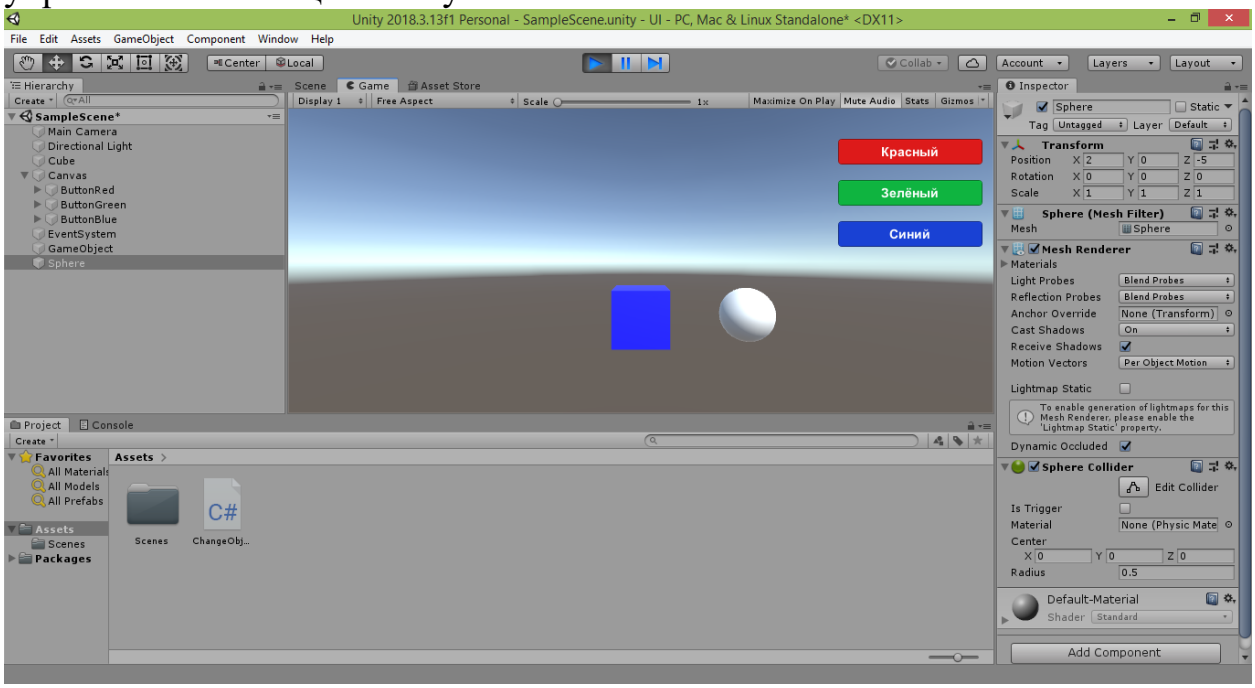
Значение свойства «Pos Z» я изменю с 0 на -5, чтобы сфера приблизилась к пользователю (камере) и находилась от него том же расстоянии, что и куб.

Вы тоже можете передвинуть сферу, задав значения её координат (как это сделал я), а можете воспользоваться инструментом перемещения «Move Tool», щёлкнув на кнопку  и потянув сферу вдоль каждой из трёх осей координат. Для более удобного управления перемещением можете временно выключить режим плоского отображения сцены, щёлкнув в строке над рабочим полем на кнопке с надписью «2D»  .



После того, как вы настроите размещение сферы на сцене, вернитесь в режим 2D-представления сцены.

Запустите проект в игровом режиме, нажав кнопку «Play», и убедитесь, что куб и сфера видны пользователю и находятся примерно в центре экрана. Также можете пощёлкать на наших кнопках и убедиться, что они пока управляют только цветом куба.



Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Теперь перейдите в редактор Visual Studio и добавьте в класс «ChangeObjects» новую открытую переменную `sphere1` , с которой мы свяжем созданную нами сферу:

```
public class ChangeObjects : MonoBehaviour
{
```

```
    public GameObject cube1, sphere1;
```

Также в ранее созданные методы «PaintToRed», «PaintToGreen» и «PaintToBlue» добавьте строки кода, изменяющие цвет сферы, соответственно, на красный, зелёный и синий:

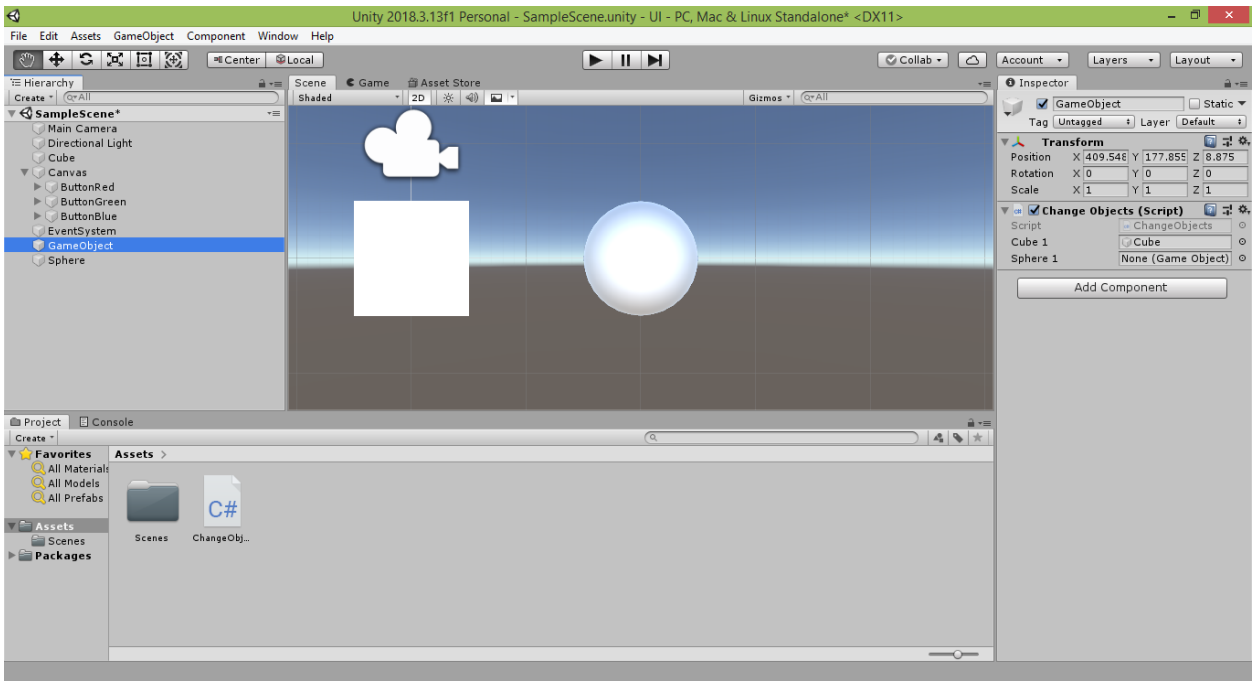
```
    public void PaintToRed()
    {
        cube1.GetComponent<Renderer>().material.color = Color.red;
        sphere1.GetComponent<Renderer>().material.color = Color.red;
    }

    public void PaintToGreen()
    {
        cube1.GetComponent<Renderer>().material.color = Color.green;
        sphere1.GetComponent<Renderer>().material.color = Color.green;
    }

    public void PaintToBlue()
    {
        cube1.GetComponent<Renderer>().material.color = Color.blue;
        sphere1.GetComponent<Renderer>().material.color = Color.blue;
    }

}
```

Сохраните код, нажав комбинацию клавиш CTRL+S.

Теперь осталось перейти обратно в окно Unity и связать переменную `sphere1` с объектом сцены.

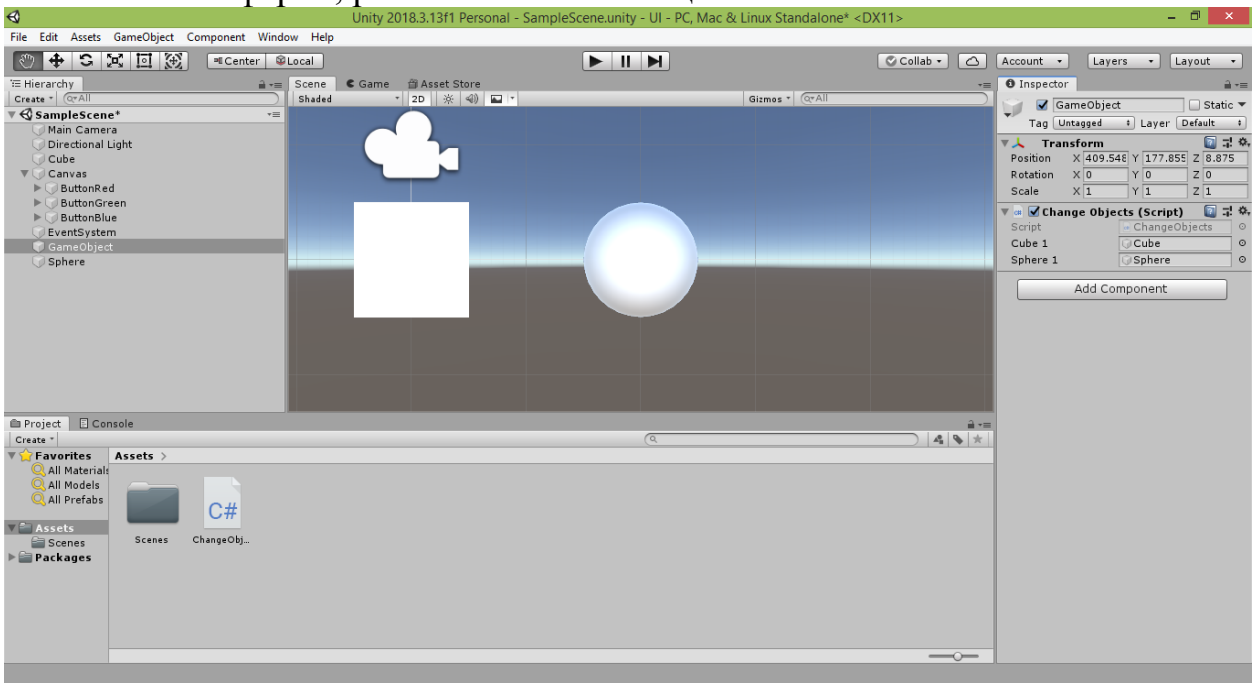
Выделите строку «GameObject» в окне Hierarchy и справа в окне Inspector вы увидите, что ниже поля «Cube 1» появилось новое поле с подписью «Sphere 1», в котором указано значение «None (Game Object)».

Если этого сразу не произошло, попробуйте подождать 5-10 секунд, пока Unity обновит состояние скрипта.

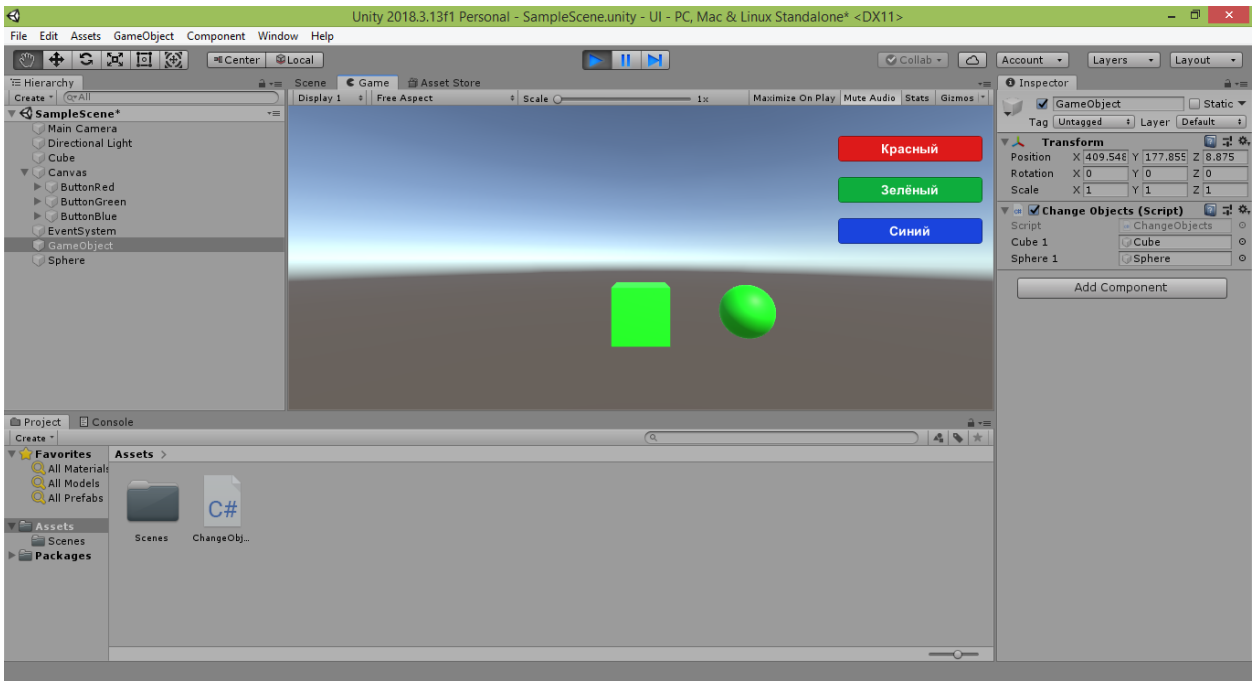


Перетяните в это поле строку «Sphere» из окна Hierarchy.

В результате открытая переменная `sphere1` из нашего скрипта окажется связанной со сферой, расположенной на сцене.



Запустите проект в игровом режиме, нажав кнопку «Play», и попробуйте пощёлкать на созданных нами кнопках. Вы увидите, что и куб, и сфера принимают цвет той кнопки, на которой вы щёлкнули.



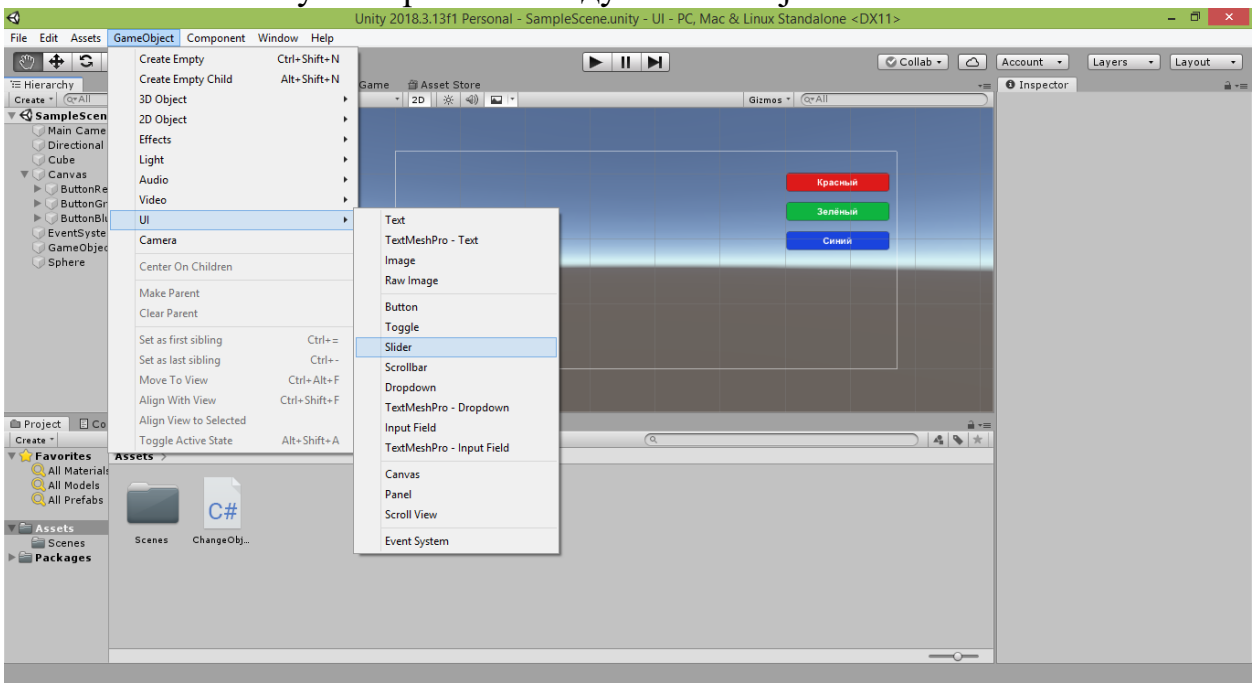
Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Как видите, после создания общего пользовательского интерфейса и написания работающего скрипта для управления одним объектом последующие добавление и программная настройка новых объектов происходят гораздо быстрее.

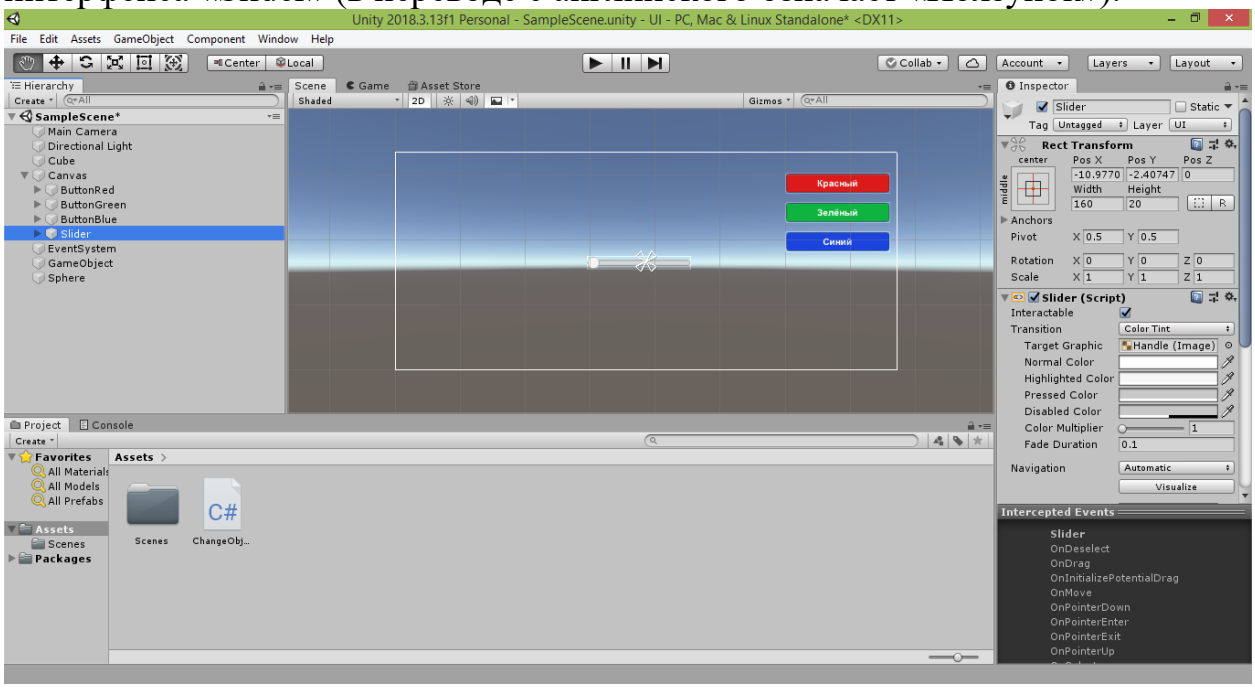
Если захотите потренироваться, можете самостоятельно добавить на сцену и настроить дополнительные объекты (например, цилиндр, капсулу или другие кубы и сферы).

4.4. Создание и настройка объекта «Slider» («Ползунок»)

В меню Unity выберите команду «GameObject → UI → Slider».



В результате на вашей сцене появится объект пользовательского интерфейса «Slider» (в переводе с английского означает «Ползунок»).



Ползунок позволяет плавно или пошагово изменять значение связанного с ним параметра. Данный объект интерфейса вы чаще всего можете встретить в проигрывателях музыки и видео, где ползунки используются для настройки громкости звука и выбора момента воспроизведения медиафайла. Также ползунки используются в дисплеях и графических редакторах для настройки характеристик изображения (яркость, насыщенность, контрастность и т.д.).

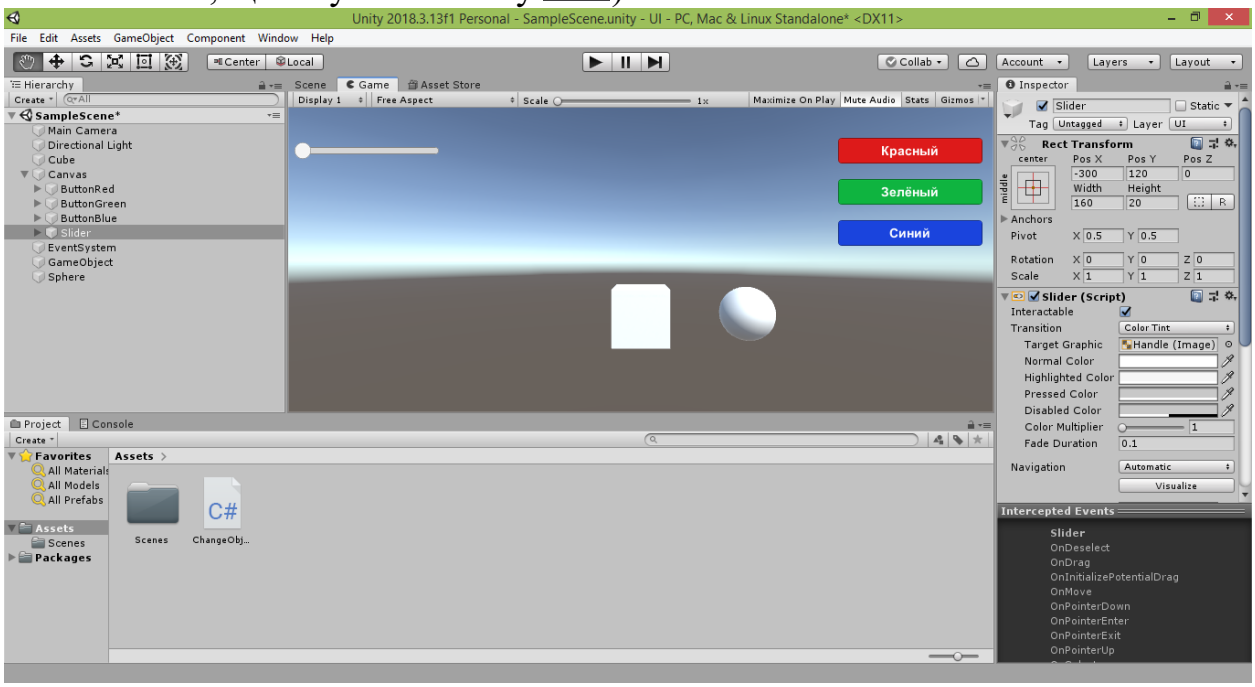
Созданный нами ползунок будет обеспечивать перемещение объектов сцены вдоль красной оси (изменять координату X местоположения объектов).

Прежде чем приступить к написанию программы для ползунка, настроим несколько его свойств.

Для этого выделим наш полузнок. Сделать это можно щелчком по нему на сцене, но удобнее всего воспользоваться списком объектов в окне Hierarchy. При создании ползунка в окне Hierarchy появилась строка «Slider», подчинённая строке «Canvas». Щёлкните на строке «Slider» в окне Hierarchy. Справа в окне Inspector появится список свойств выбранного нами ползунка.

Измените значение свойства «Pos X» на -300, а значение свойства «Pos Y» – на 120. Мы задали новые координаты расположения нашего ползунка. Перейдите на вкладку «Game» и проверьте, виден ли ваш ползунок в левом верхнем углу игрового окна.

Если этого не произошло, значит, ползунок оказался за пределами рамки полотна «Canvas». В этом случае подберите значения свойств «Pos X» и «Pos Y», чтобы кнопка оказалась в пределах рамки полотна «Canvas» (для перемещения кнопки по полотну можно также воспользоваться инструментом «Move Tool», щёлкнув на кнопку ).

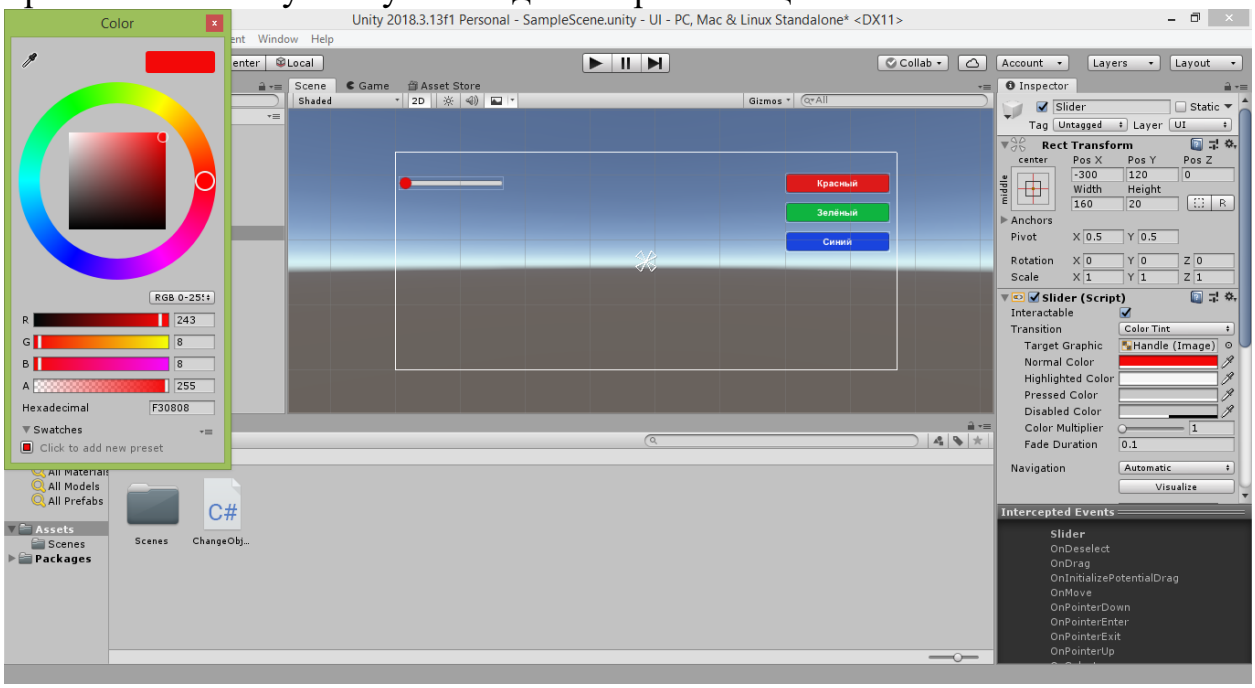


Если запустить проект в игровом режиме, нажав в самом верху кнопку «Play» со значком чёрного треугольника, указывающего вправо, то вы даже сможете передвигать ручку ползунка (пока что это не будет приводить к какому-либо результату).

Нажмите кнопку «Play», чтобы выйти из игрового режима, и вернитесь на вкладку «Scene» («Сцена»).

Щёлкните в окне Inspector на белом прямоугольнике в свойстве «Normal Color» («Цвет в обычном состоянии»). В результате слева появится окно с палитрой выбора цвета. Щёлкая мышкой по окружности, вы можете выбрать

нужный вам цвет, а щёлкая в области расположенного в центре квадрата – задать насыщенность выбранного цвета. Задайте ползунку красный цвет. В результате ручка ползунка станет красным. Если ползунок выглядит мелким, приблизьте к нему точку наблюдения при помощи колёсика мышки.



Теперь настроим положение ручки ползунка. В окне Inspector прокрутите немного вниз список свойств ползунка. Вы увидите следующие свойства:

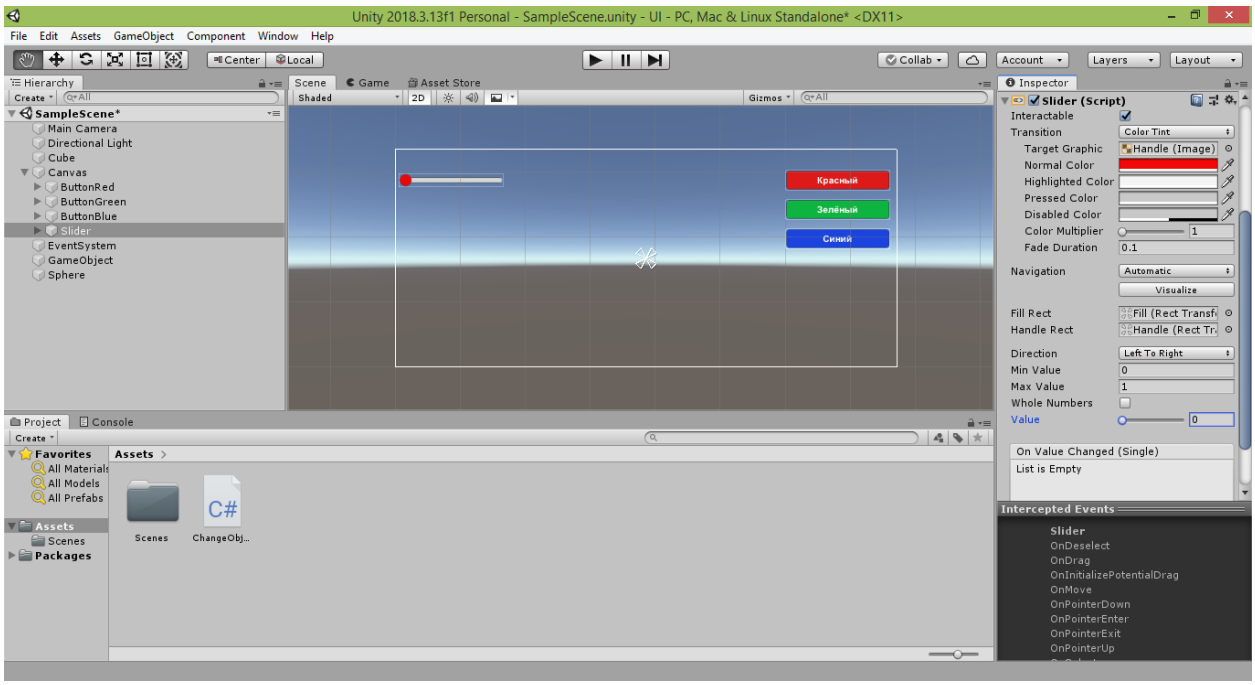
- выпадающий список «Direction» со значением «Left To Right»;
- поле «Min Value» со значением 0;
- поле «Max Value» со значением 1;
- флажок «Whole Numbers» со значением 1;
- ползунок «Value» со значением 0.

Данные свойства отвечают за режима прокрутки ручки ползунка.

Свойство «Direction» (переводится как «Направление») определяет, в каком направлении будет двигаться ручка ползунка.

Значение «Left To Right» («Слева направо») означает, что слева будет находиться начало отсчёта позиции ручки ползунка, а саму ручку пользователь будет перемещать вправо.

Такой вариант нас устраивает, поэтому не будет изменять это свойство.



Поля «Min Value» («Минимальное значение») и «Max Value» («Максимальное значение») определяют минимальное и максимальное значения, соответствующие крайним положениям ручки ползунка.

Изначально при крайнем левом положении ручки ползунок будет выдавать значение 0, а при крайнем правом – значение 1.

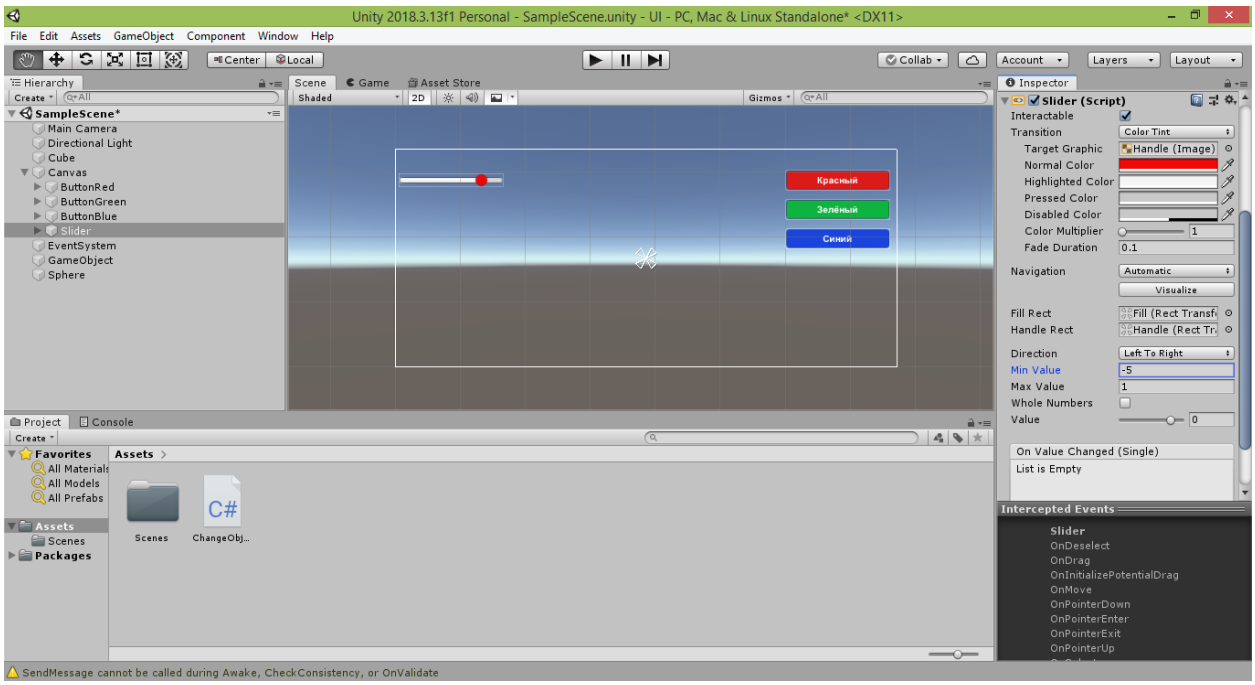
Мы будем использовать числовые значения, выдаваемые ползунком, чтобы определять, насколько сдвинуть объекты сцены вдоль красной оси относительно их начального местоположения.

Сейчас получается, что ручка ползунка может перемещаться только вправо и наши объекты тоже будут двигаться только в этом направлении.

Чтобы появилась возможность при помощи ползунка уменьшать координату X, значение в поле «Min Value» должно быть отрицательным.

Попробуйте установить значение свойства «Min Value» равным -5.

В результате ручка ползунка сдвинется вправо.



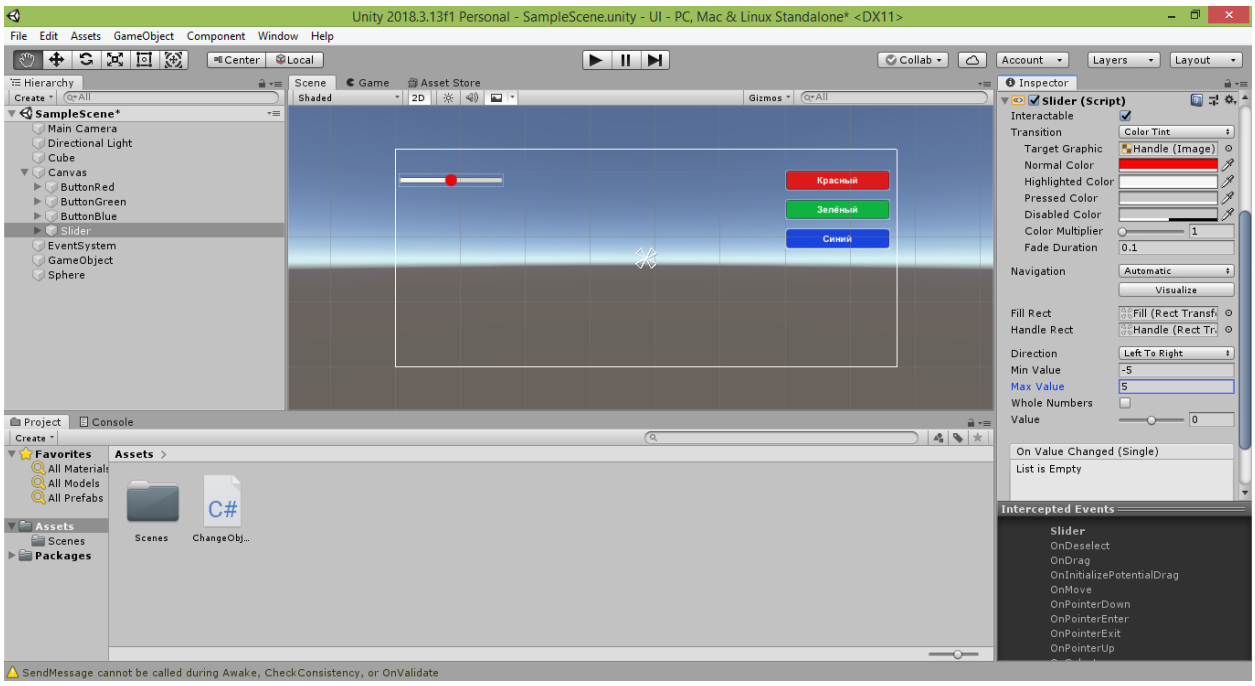
Дело в том, что мы пока не меняли значение свойства «Value» («Значение»), которое задаёт текущее местоположение ручки. Оно осталось равным 0 (каким и было при создании ползунка). Но изначально диапазон значений ручки составлял от 0 до 1, и при значении свойства «Value», равном 0, ручка находилась в крайнем левом положении.

Сейчас диапазон значений изменился, и составляет от -5 до +1.

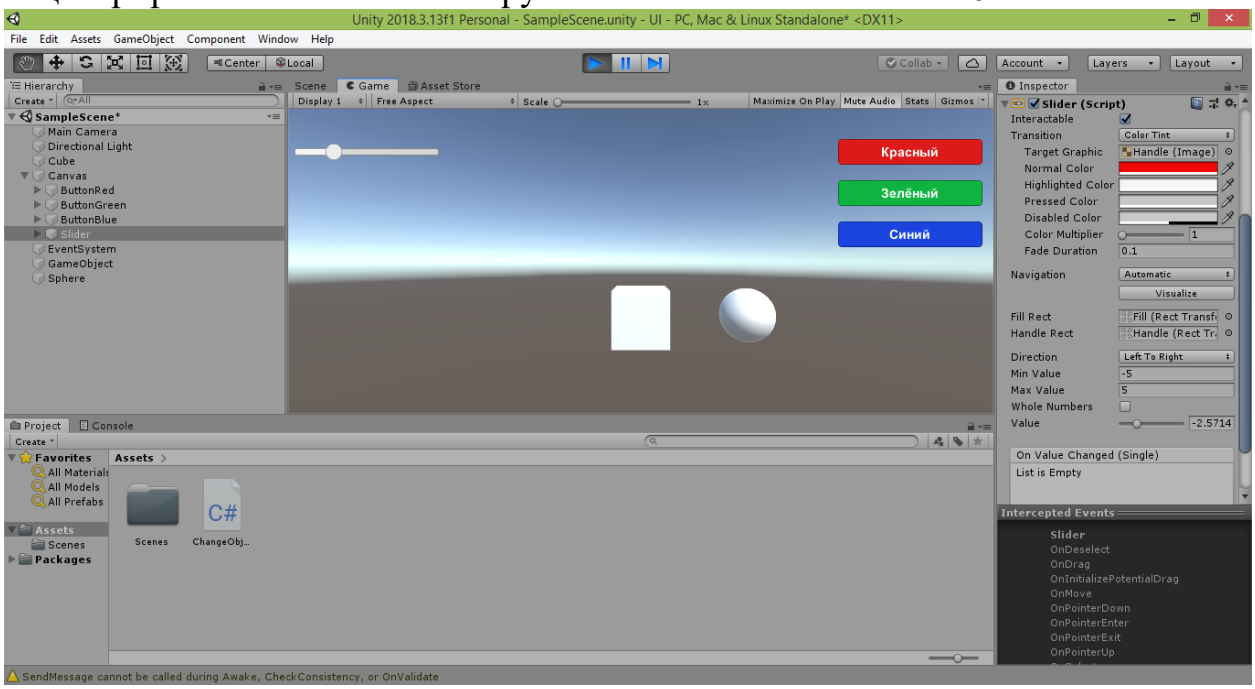
В результате длина отрицательного поддиапазона (от -5 до 0) ручки ползунка стала в 5 раз больше длины положительного поддиапазона (от 0 до 1). Поэтому и получилось, что при значении свойства «Value», равном 0, ручка ползунка сместилась вправо.

Чтобы ручка ползунка при диапазоне от -5 до +1 снова оказалась в центре, мы должны сделать значение свойства «Value» равным -2, поскольку расстояние от -5 до -2 равно расстоянию от -2 до +1.

Но нам проще изменить значение свойства «Max Value», сделав его равным 5. Тогда диапазон значений ползунка будет составлять от -5 до +5, а значение свойства «Value», равное 0, будет соответствовать центральному положению ручки. Одновременно это обеспечит равенство максимальных перемещений объектов сцены влево и вправо.



Можете попробовать запустить проект в игровом режиме и проверить, как работает ползунок. Если при это справа останется открытым окно Inspector, то вы увидите, как в поле «Value» появляются числа, соответствующие текущем положению ручки ползунка. В крайнем левом положении ручки в поле появится число -5, в крайнем правом – число 5. При отцентрированном положении ручки в поле появится число 0.

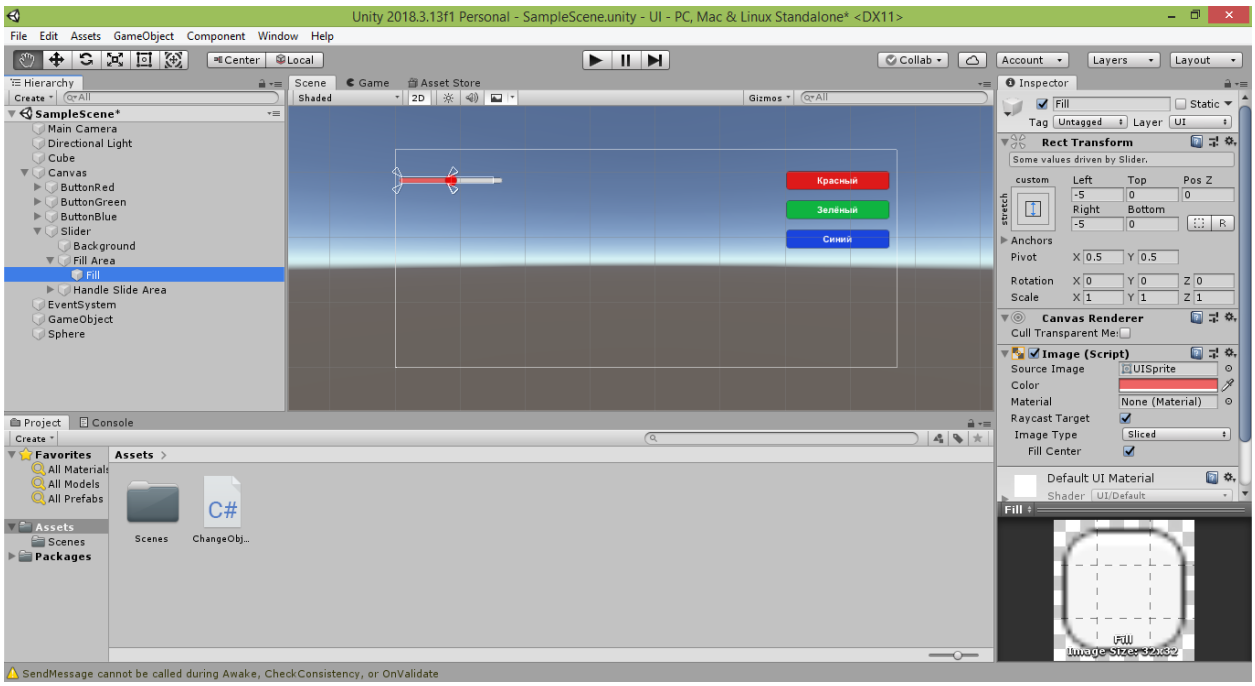


Выйдите из игрового режима и вернитесь на вкладку «Scene» («Сцена»).

Теперь попробуйте в окне Inspector установить флажок в свойстве «Whole Numbers» («Целые числа»), расположенном сразу над свойством «Value». Запустите проект в игровом режиме и проверьте, как теперь будет работать ползунок. Вы увидите, что ручка ползунка теперь стала

перемещаться скачками, а в свойстве «Value» теперь появляются только целые числовые значения (без дробной части): -5, -4, 3, -2, -1, 0, 1, 2, 3, 4, 5.

В ряде случае такая настройка может оказаться полезной. Но мы не хотим, чтобы наши объекты перемещались по сцене скачками. Поэтому выйдите из игрового режима и уберите флажок в свойстве «Whole Numbers» нашего ползунка, чтобы вернуть его ручке плавный режим прокрутки.



Следует заметить, что элемент управления «Slider», как и элемент управления «Button», является составным. Щёлкните в окне Hierarchy на треугольник слева от строки «Slider».

Строка «Slider» раскроется и ниже появятся подчинённые ему строки «Background» («Фон»), «Fill Area» («Область заполнения») и «Handle Slide Area» («Область ручки прокрутки»).

Щёлкните на строке «Fill Area» ползунка.

Справа в окне Inspector вы увидите относительно небольшой список свойств.

Но нас будет интересовать не он, а свойства объекта, подчинённого объекту «Fill Area».

Дело в том, что объект «Fill Area» тоже является составным.

Чтобы убедиться в этом, щёлкните в окне Hierarchy на треугольник слева от строки «Fill Area» нашего ползунка.

Строка «Fill Area» раскроется и ниже появится подчинённая ей строка «Fill» («Заполнение»).

Щёлкните на строке «Fill» ползунка.

Справа в окне Inspector вы увидите большое количество свойств.

Найдите в группе «Image (Script)» свойство «Color» («Цвет»).

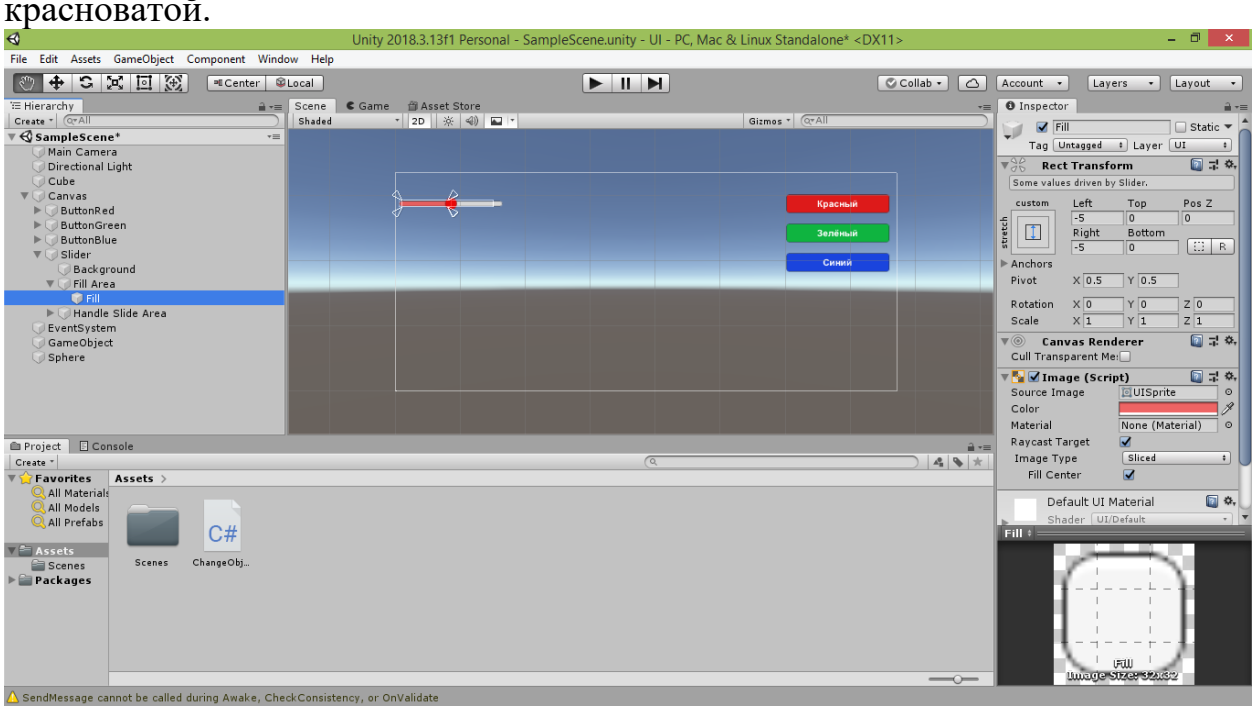
Оно задаёт цвет той части диапазона, которую пробежала ручка ползунка.

Щёлкните на белом прямоугольнике в свойстве «Color».

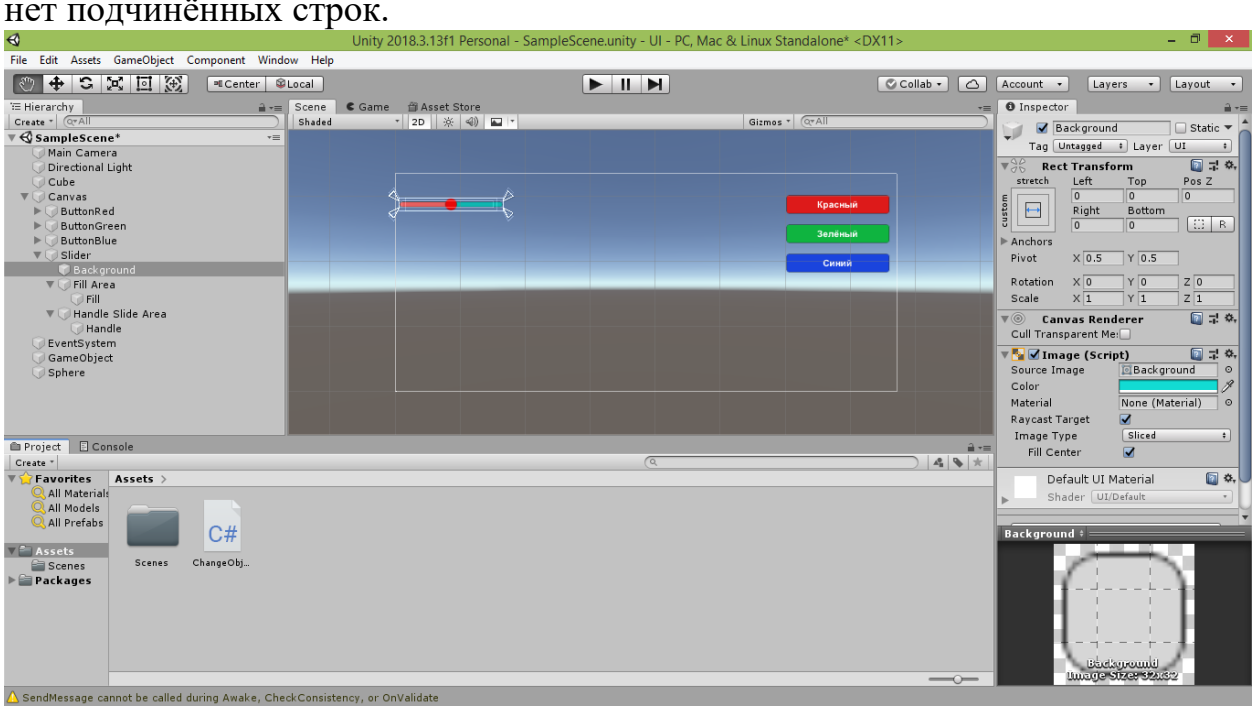
В результате слева появится окно с палитрой выбора цвета.

Выберите не слишком насыщенный оттенок красного цвета.

В результате вы увидите, как область левее ручки ползунка станет красноватой.



Свойство «Color» также имеется в списке свойств объекта «Background», подчинённого объекту «Fill Area», и отвечает за закраску правой части ползунка, пока ещё не пройденной его ручкой. Сам объект «Background» является простым, поэтому в окне Hierarchy у соответствующей ему строки нет подчинённых строк.



Что касается объекта «Fill Handle Area», подчинённого объекту «Fill Area», то он является составным. В его состав входит объект «Handle», у которого свойство «Color» задаёт цвет ручки ползунка в момент её нажатия пользователем (в обычном состоянии этот цвет смешивается с цветом, заданным в объекте «Fill»).

Если интересно, можете поэкспериментировать с настройками цвета разных частей ползунка. Я же оставлю изменёнными только цвет ручки ползунка и цвет пройденной ею области.

Чтобы иметь возможность перемещать объекты не только вдоль красной, но и вдоль зелёной и синей осей, в дополнение к созданному красному ползунку нам потребуются ещё два – зелёного и синего цветов. Добавить и настроить их можно аналогичным способом. Однако в окне Hierarchy они все будут обозначены как «Slider». Чтобы в дальнейшем не путаться в них, мы зададим каждому ползунку уникальное имя. Для начала переименуем строку «Slider», соответствующую красному ползунку, в «SliderMoveRed». Для этого щёлкните на строке «Slider» в окне Hierarchy. Переименовать её можно несколькими способами:

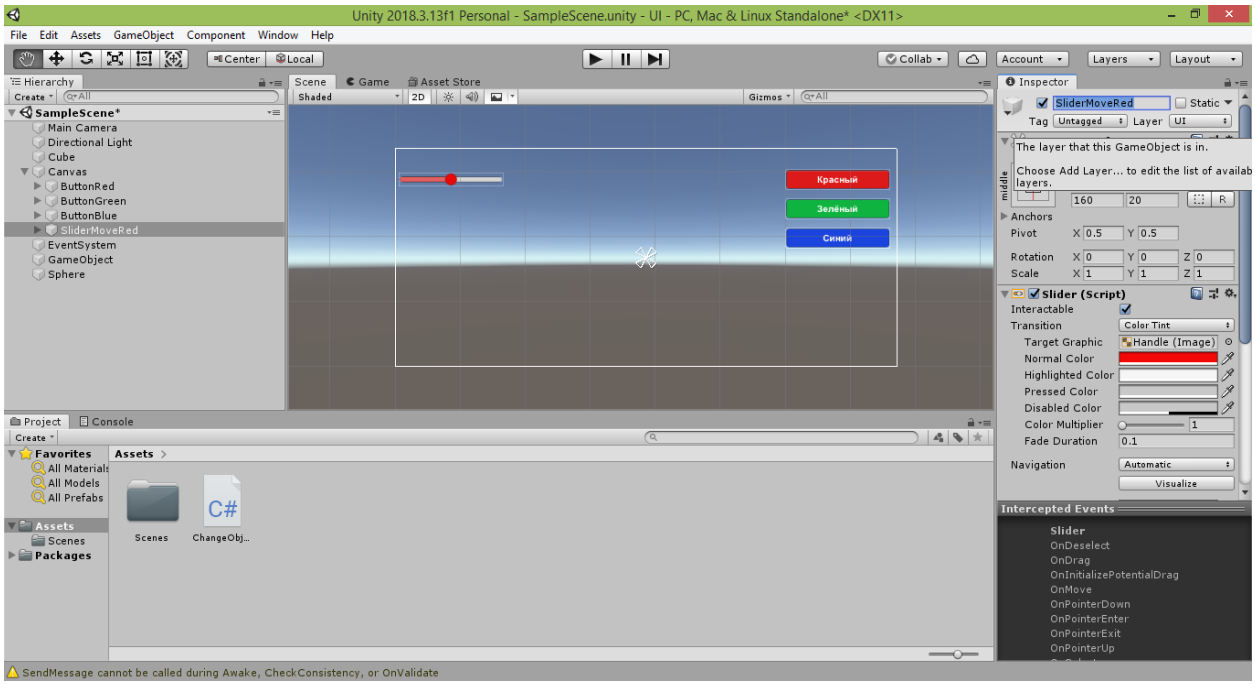
– сделать два отдельных щелчка на выделенной строке;

– нажать после выделения строки клавишу F2;

– щёлкнуть на строке правой кнопкой мыши и выбрать в открывшемся контекстном меню команду «Rename» («Переименовать»);

– зайти в окно Inspector и изменить в самом первом (верхнем) поле текст «Slider» на «SliderMoveRed».

После переименования строка «Slider» в окне Hierarchy получит новое название «SliderMoveRed».

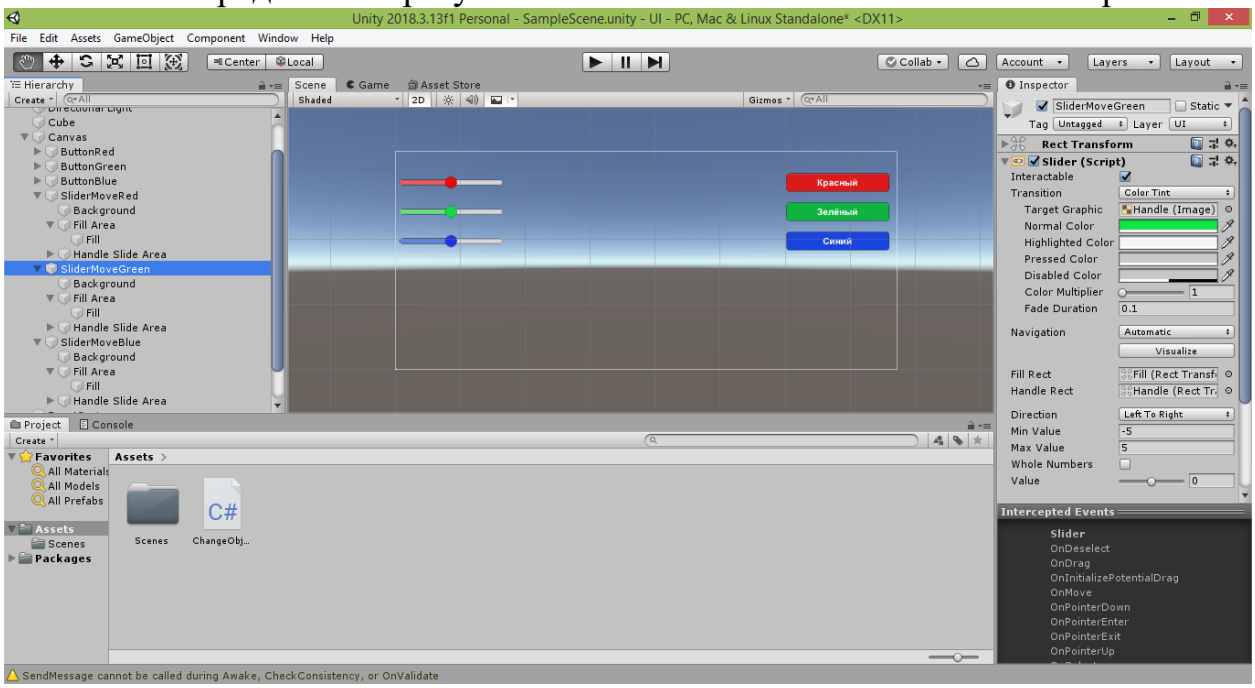


Используя в меню Unity команду «GameObject → UI → Slider», добавьте ещё два ползунка. Одному дайте имя «SliderMoveGreen», другому – «SliderMoveBlue».

У ползунка «SliderMoveGreen» измените значение свойства «Pos X» на -300, а значение свойства «Pos Y» – на 75. Свойству «Normal Color» ползунка задайте зелёный цвет. Свойству «Min Value» задайте значение -5, а свойству «Max Value» задайте значение 5. Далее, найдя слева в окне Hierarchy строку «Fill Area», подчинённую этому ползунку, раскройте её и уже у её подчинённой строки «Fill» справа в окне Inspector выберите в палитре у свойства «Color» не слишком насыщенный оттенок зелёного цвета.

У ползунка «SliderMoveBlue» измените значение свойства «Pos X» на -300, а значение свойства «Pos Y» – на 30. Свойству «Normal Color» ползунка задайте синий цвет. Свойству «Min Value» задайте значение -5, а свойству «Max Value» задайте значение 5. Далее, найдя слева в окне Hierarchy строку «Fill Area», подчинённую этому ползунку, раскройте её и уже у её подчинённой строки «Fill» справа в окне Inspector выберите в палитре у свойства «Color» не слишком насыщенный оттенок синего цвета.

Ниже представлен результат выполнения всех описанных настроек.



Теперь приступим к программированию созданных ползунков.

Делать это мы будет в уже созданном нами скрипте ChangeObjects, в котором мы ранее программировали кнопки (на самом деле, для ползунков можно было создать отдельный скрипт, но нам проще все изменения объектов запрограммировать в одном скрипте-классе, поскольку для этой цели он и создавался).

Зайдя в скрипт в редакторе Visual Studio (или в любом текстовом редакторе наподобие программы «Блокнот» / «Notepad»), добавим новый открытый метод в конец нашего скрипта (после метода PaintToBlue):

```
    public void MoveRed(Slider RedSlider)
    {
        cube1.transform.position += new Vector3(RedSlider.value, 0f,
0f);
        sphere1.transform.position += new Vector3(RedSlider.value, 0f,
0f);
    }
```

Метод мы назвали «MoveRed», чтобы обозначить, что он отвечает за перемещение объектов вдоль красной оси.

В отличие от методов, которые мы создавали для кнопок, у этого метода круглые скобки, стоящие после названия, не являются пустыми, а содержат параметр `Slider RedSlider`.

Слово `Slider` является названием класса, описывающим элемент управления «Slider». Сейчас слово `Slider` не подсвечено цветом, а подчёркнуто красной волнистой линией, поскольку программа не знает, что это за класс. Дело в том, что описание этого класса содержится в пространстве имён UI, которое мы пока не подключили в нашем скрипте. Данное пространство имён вложено в более крупное пространство имён UnityEngine. Поэтому, чтобы подключить вложенное пространство имён UI, пропишите в самом начале скрипта строку `using UnityEngine.UI;` .

Таким образом, самые первые строки вашего скрипта будут такими:

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

Теперь вернитесь в метод «MoveRed» и убедитесь, что слово «Slider» в круглых скобках перестало подчёркиваться красным, а стало подсвечиваться цветом морской волны: `Slider` .

Следующее за ним слово `RedSlider` было придумано нами. Оно является названием параметра, который передаётся в метод и будет влиять на результат его работы. Поскольку `RedSlider` имеет тип `Slider`, в качестве параметра метода будет выступать ползунок. По положению ручки ползунка мы будем определять, на какое расстояние требуется переместить объекты сцены.

В теле созданного метода мы прописали две строки, в которых обращаемся к свойству `position` наших куба и сферы.

Куб и сфера ранее были связаны с созданными нами открытыми переменными `cube1` и `sphere1` .

Поэтому мы можем использовать обращения к их местоположению `cube1.transform.position` и `sphere1.transform.position` .

Обратите внимание, что здесь мы получаем доступ к местоположению конкретных объектов сцены через входящий в их состав компонент `transform`.

Ранее мы с вами пробовали менять местоположение объектов через скрипт:

```
Transform t;
t = GetComponent<Transform>();
t.position = t.position + new Vector3(0.1f, 0f, 0f);
```

В нём мы получали компонент класса `Transform` в общем виде и не привязывали его к конкретному объекту сцены.

Вместо этого мы использовали связь в общем виде – с переменной `t` этого же класса.

Сама привязка к конкретному объекту сцены происходила в тот момент, когда мы переходили в Unity и связывали скрипт с этим объектом (кубом, сферой, цилиндром, капсулой, камерой и т.д.).

Теперь же мы получаем доступ к компоненту класса `Transform` конкретных объектов сцены через ссылающиеся на них переменные.

То есть тем самым мы указываем, что хотим перемещать только те объекты, на которые ссылаются переменные `cube1` и `sphere1` .

Поэтому не имеет значения, сколько других объектов расположено на нашей сцене и какой из трёх ползунков будет использовать созданный нами метод.

Перемещаться будут только те два объекта, на которые ссылаются переменные `cube1` и `sphere1` .

Ещё раз напомню, что ссылки на объекты настраиваются в созданном нами пустом объекте «GameObject» , к которому мы и привязали скрипт ChangeObjects .

То есть, чтобы программно управлять объектами сцены, не обязательно напрямую привязывать к ним скрипт.

Это можно делать через промежуточный (например, пустой) объект, в свойствах которого в окне Inspector можно связать открытые переменные скрипта с объектами сцены.

Теперь, когда мы получили доступ к компоненту `transform` объекта сцены, отвечающему за его трансформацию, мы обращаемся к свойству `position` этого компонента, чтобы изменить текущее местоположение этого объекта:

```
cube1.transform.position
sphere1.transform.position
```

Далее, используя операцию «+=», мы добавляем к текущему местоположению объекта новый вектор смещения `new Vector3(RedSlider.value, 0f, 0f)` .

Как видите, изменения затронут только координату X объекта.

К координатам Y и Z будут добавлены нули, и они не изменятся.

Смещение вдоль красной оси определяется числовым значением, соответствующим текущему положению ручки ползунка.

Чтобы получить это число, мы обращаемся к параметру `RedSlider` , который мы задали у метода «MoveRed» в круглых скобках.

Поскольку этим параметром является ползунок (на это указывает тип параметра – `Slider`), мы можем обратиться через точку к его свойству «Value».

Важное замечание. Путаницу может вызвать, что в окне «Unity» у ползунка свойство «Value» пишется с большой буквы, а в коде название переменной `value` пишется с маленькой буквы.

Дело в том, что в окне Unity автоматически повышается регистр первой буквы каждого слова, входящего в название открытой переменной.

Разделителями слов в названии переменной считаются заглавные буквы и цифры.

Возможно, вы обратили внимание, что когда мы создали открытые переменные `cube1` и `sphere1` , в окне Unity в свойствах пустого объекта «GameObject» для них появились поля «Cube 1» и «Sphere 1».

То есть среда Unity создала эти подписи на основе названий переменных, повысив регистр первых букв входящих в них слов, а также разделив слова пробелами.

Поэтому на будущее просто возьмите за правило обращаться в коде ко всем компонентам, вложенным в объект, со строчной буквы.

Теперь, когда мы разобрались, что происходит в созданном методе, нажмите комбинацию клавиш CTRL+S, не закрывая окно редактора Visual Studio, перейдите обратно в окно Unity.

Здесь мы назначим созданный метод «MoveRed» красному ползунку.

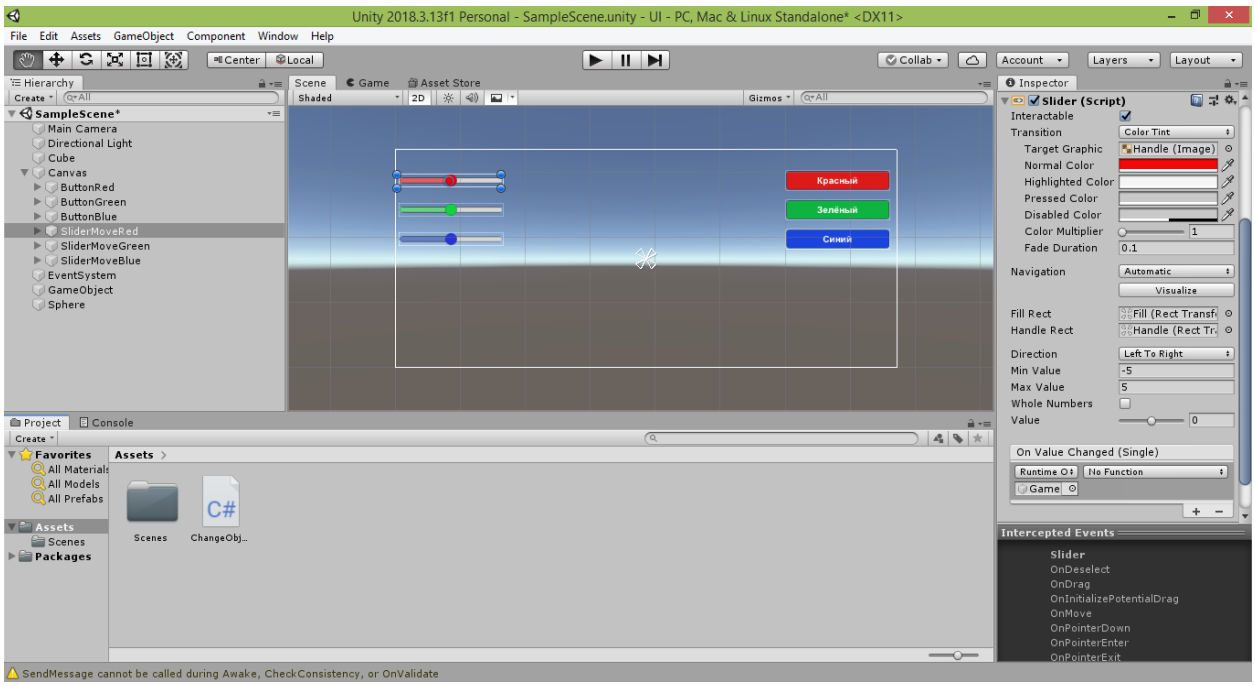
Выделите строку «SliderMoveRed» в окне Hierarchy и справа в окне Inspector найдите группу свойств «Slider (Script)». Внизу этой группы имеется пустой список «On Value Changed (Single)», в котором сейчас написано «List is Empty» («Список пуст»). В данном списке можно задать программные обработчики для события «OnValueChanged» (что переводится с английского как «При изменении значения»), которое возникает, когда пользователь перемещает ручку ползунка.

Щёлкните на кнопке «+» внизу под списком.

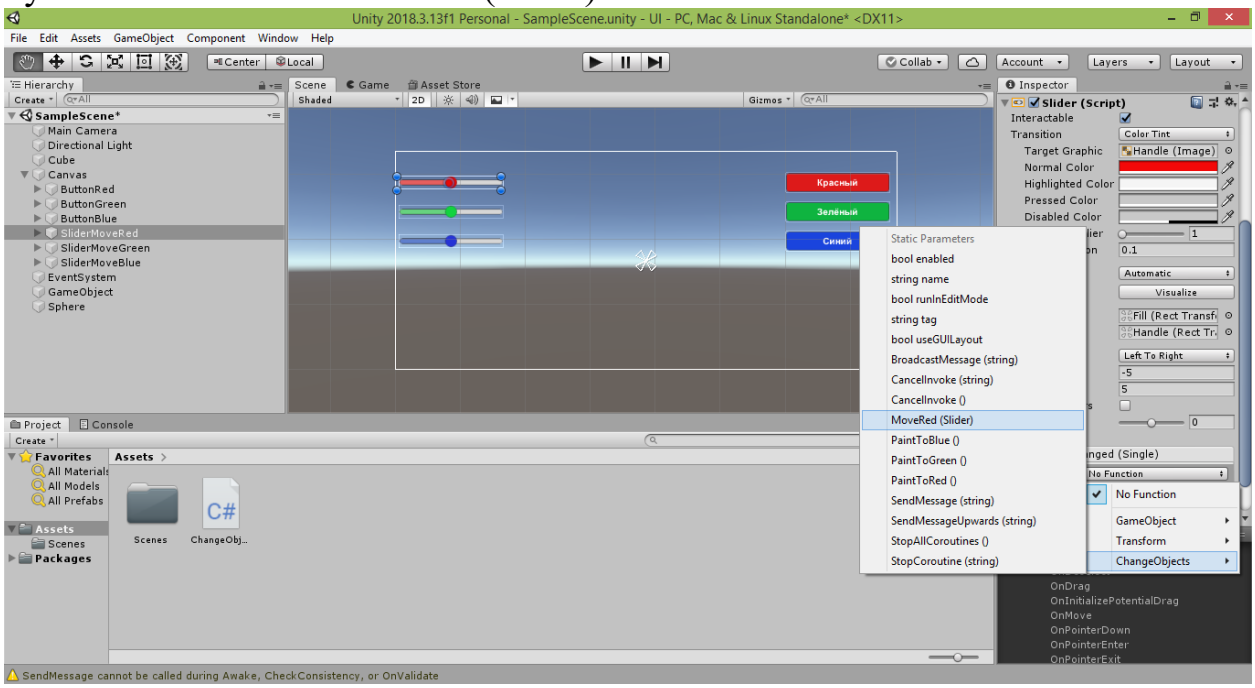
В результате в списке появится новый блок для задания обработчика события.

Перетяните строку «GameObject» из окна Hierarchy в поле блока, где написано «None (Object)», и отпустите. В поле появится новая надпись: «GameObject».

При этом станет доступен для изменения выпадающий список с надписью «No Function», расположенный рядом с полем.



Раскройте выпадающий список и в подменю «ChangeObjects» выберите пункт с названием «MoveRed (Slider)».



После этого под выпадающим списком появится новое поле с надписью «None (Slider)» («Отсутствует (Ползунок)»).

Если вы помните, ранее при задании методов-обработчиков для кнопок никаких дополнительных полей не появлялось. Это было связано с тем, что у этих методов обработки щелчка отсутствовали параметры (круглые скобки этих методов были пустыми).

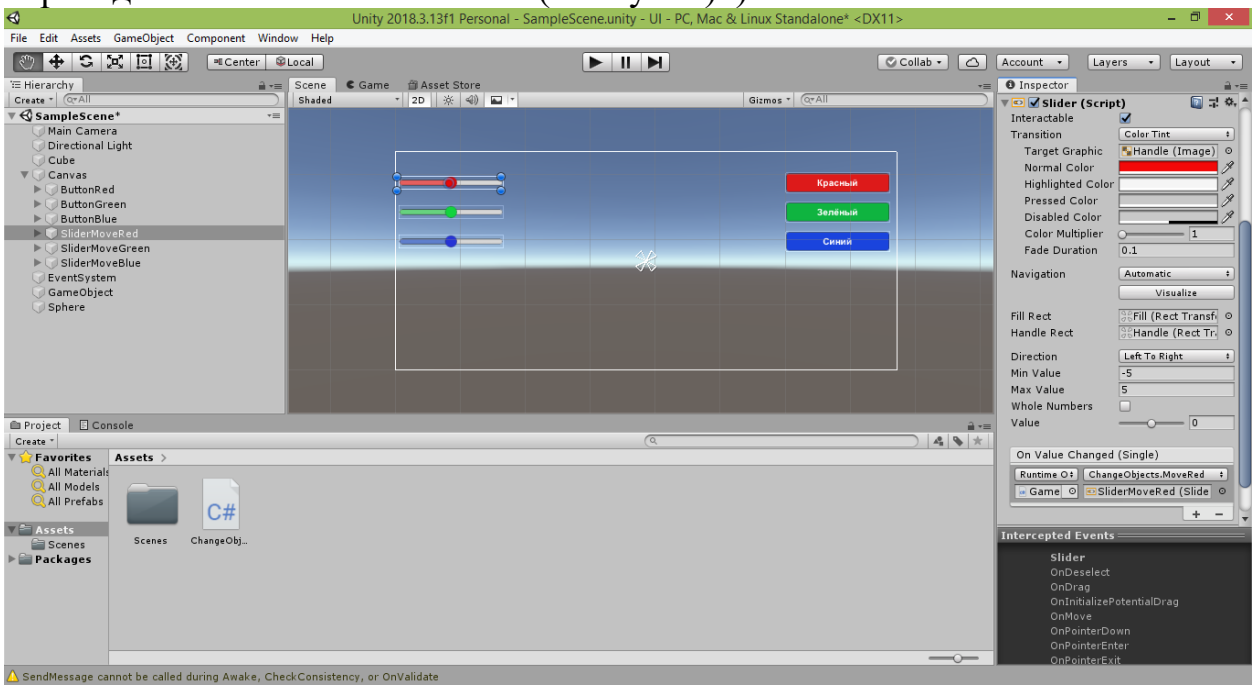
Но в скобках у метода «SliderMoveRed», выбранного в качестве обработчика, нами был указан параметр `Slider RedSlider`.

И при вызове метода мы должны задать его значение.

В качестве значения будет выступать красный ползунок, у которого метод будет считывать положение ручки для перемещения куба и сферы.

Перетяните строку «SliderMoveRed» из окна Hierarchy в поле, где надписано «None (Slider)», и отпустите.

В поле появится новая надпись: «SliderMoveRed (Slider)» (что переводится как «SliderMoveRed (Ползунок)»).



Таким образом, в блоке обработки события мы задали, что при изменении положения ручки красного ползунка будет вызываться метод «MoveRed», который в качестве параметра будет использовать данные о положении ручки этого же ползунка для перемещения куба и сферы.

Следует отметить, что в качестве параметра метода мы могли бы задать любой другой ползунок (например, зелёный или синий).

В этом случае при движении ручки красного ползунка для перемещения объектов сцены использовались бы данные о положении ручки другого ползунка, выбранного в качестве параметра.

Однако поскольку вызов метода «MoveRed» произойдёт, когда пользователь начнёт управлять красным ползунком, логично использовать данные о положении ручки красного ползунка, чтобы перемещать объекты по сцене.

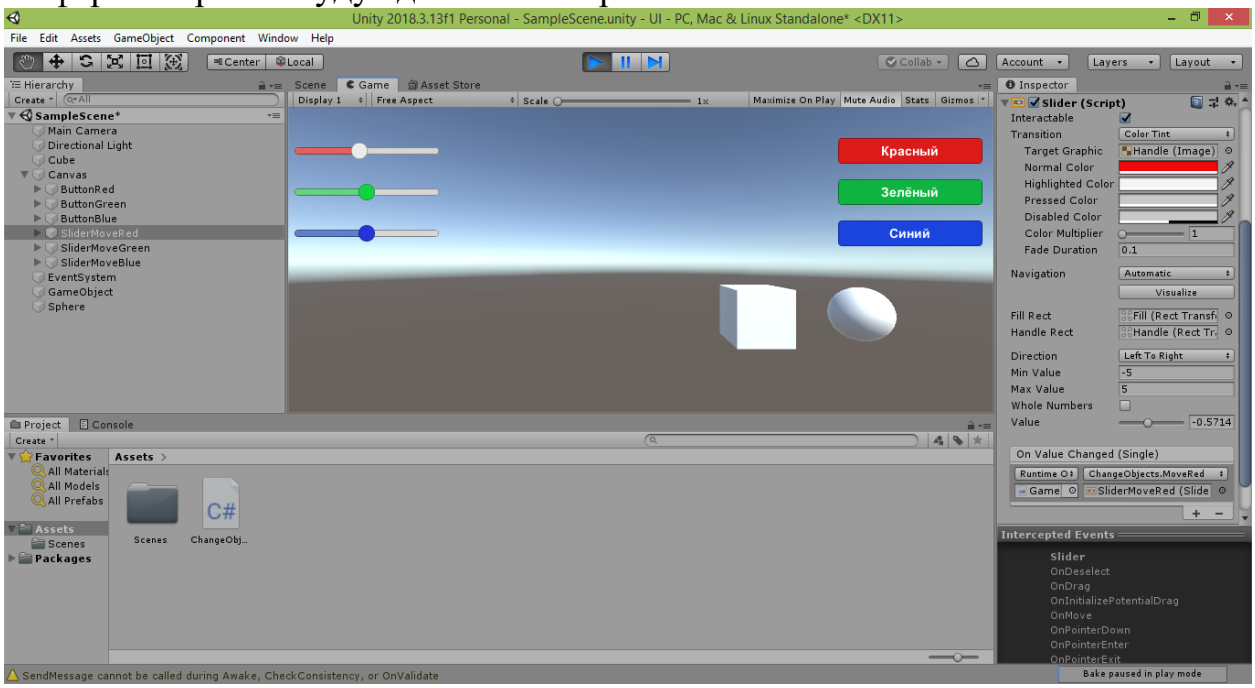
Теперь, когда мы настроили для красного ползунка обработку события «OnValueChanged», запустите проект в игровом режиме, нажав кнопку «Play».

Попробуйте переместить вправо ручку ползунка на два-три небольших шага (не торопитесь делать много движений ручкой ползунка, иначе объекты могут уйти из поля зрения камеры).

В результате куб и сфера начнут двигаться вправо.

Однако двигаются они неравномерно: каждый шаг становится больше.

Кроме того, если сдвинуть ручку ползунка на небольшой шаг влево, куб и сфера всё равно будут двигаться вправо.



Это связано с тем, что в коде мы постоянно прибавляем значение, соответствующее положению ручки красного ползунка, к текущей координате X объектов.

Проще говоря, пока ручка ползунка находится в диапазоне значений от 0 до 5, координата X будет увеличиваться.

Например, если мы переместим ручку со значения 5 влево на значение 3, в этом случае метод «MoveRed» увеличит на 3 координату X куба и сферы.

Влево объекты начнут двигаться в том случае, когда ручка ползунка попадёт в отрицательный диапазон значений (от -5 до 0).

Но и здесь будет аналогичная проблема: если мы, например, передвинем ручку ползунка из крайнего левого положения (значение -5) вправо на значение -2, то куб и сфера всё равно переместятся влево, поскольку метод «MoveRed» добавит к их координате X значение -2, уменьшив её и тем самым переместив объекты влево.

Как же быть?

Относительно простым методом решения этой проблемы является прибавление значения, соответствующего положению ручки ползунка, к координате X начального положения объектов сцены.

Проще говоря, после каждого движения ручки ползунка мы будем считать, что объекты заняли своё исходное местоположение, к которому мы и будем применять смещение, чтобы определить новое (на этот раз корректное) местоположение объектов.

Данный подход потребует от нас небольшого изменения кода скрипта.

Нажмите в Unity кнопку «Play», чтобы остановить проигрывание сцены.

Перейдите в редактор Visual Studio и объявите две скрытые переменные `startPositionCube1` и `startPositionSphere1` сразу после строки, где были объявлены открытые переменные `cube1` и `sphere1` :

```
public GameObject cube1, sphere1;
private Vector3 startPositionCube1, startPositionSphere1;
```

Названия переменных `startPositionCube1` и `startPositionSphere1` мы выбрали сами (они переводятся как «Начальное местоположение Куба 1» и «Начальное местоположение Сферы 1»).

Скрытыми (`private`) эти переменные мы объявили, потому что они будут использоваться только внутри скрипта (мы не будем переходить в Unity и привязывать к ним объекты сцены).

Ещё раз напомню, что можно было вообще не указывать модификатор доступа `private` – в таких случаях переменные также считаются скрытыми :

```
Vector3 startPositionCube1, startPositionSphere1;
```

В качестве типа объявленных переменных мы указали класс `Vector3` .

Дело в том, что местоположение объектов сцены, хранящееся в свойстве `position`, представлено в виде вектора (набора) из трёх координат (X, Y и Z) и имеет тип `Vector3` .

Поэтому начальное местоположение куба и сферы мы сможем сохранить только в переменных типа `Vector3` .

Само запоминание начального местоположения должно происходить при запуске сцены.

Для этого воспользуемся стандартным методом `Start`, в котором присвоим новым переменным текущие местоположения соответствующих им объектов:

```
void Start()
{
    startPositionCube1 = cube1.transform.position;
    startPositionSphere1 = sphere1.transform.position;
}
```

Теперь изменим код в методе «MoveRed».

В развёрнутом виде старый код этого метода выглядит так:

```
public void MoveRed(Slider RedSlider)
{
  cube1.transform.position = cube1.transform.position + new Vector3(RedSlider.value,
0f, 0f);

  sphere1.transform.position = sphere1.transform.position + new
Vector3(RedSlider.value, 0f, 0f);
}
```

Мы условились, что теперь будем отсчитывать смещение не от текущего, а от начального местоположения объекта. Поэтому скорректируем код метода:

```
public void MoveRed(Slider RedSlider)
{
    cube1.transform.position = startPositionCube1 + new Vector3(RedSlider.value,
0f, 0f);

    sphere1.transform.position = startPositionSphere1 + new
Vector3(RedSlider.value, 0f, 0f);
}
```

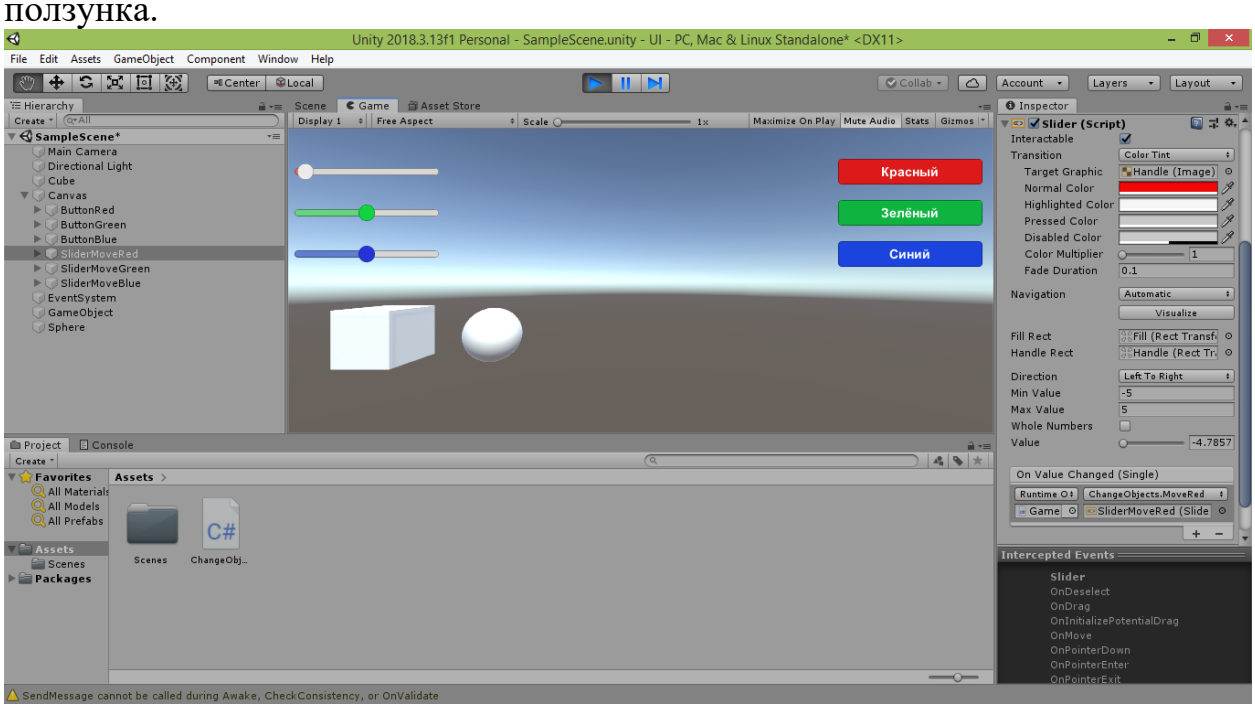
Похоже на старый вариант, но результат будет отличаться.

Теперь мы задаём каждому объекту сцены новое местоположение путём добавления к его начальному местоположению вектора смещения в соответствии с положением ручки красного ползунка.

Проверим работу метода, сохранив его нажатием комбинации клавиш CTRL+S и перейдя в окно Unity.

Запустите проект в игровом режиме, нажав кнопку «Play», и проверьте, как работает ползунок.

Теперь куб и сфера двигаются в соответствии с перемещением ручки ползунка.



Работает!

Однако мы планировали двигать объекты не только вдоль красной оси, но и вдоль двух других.

Для этого потребуется аналогичным образом настроить зелёный и синий ползунки.

Нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Перейдите в редактор Visual Studio и добавьте после метода «MoveRed» ещё два открытых метода: «MoveGreen» и «MoveBlue»:

```
    public void MoveRed(Slider RedSlider)
    {
```

```
        cube1.transform.position = startPositionCube1 + new Vector3
(RedSlider.value, 0f, 0f);

        sphere1.transform.position = startPositionSphere1 + new
Vector3
(RedSlider.value, 0f, 0f);
    }

    public void MoveGreen(Slider GreenSlider)
    {
        cube1.transform.position = startPositionCube1 + new Vector3
(0f, GreenSlider.value, 0f);

        sphere1.transform.position = startPositionSphere1 + new
Vector3
(0f, GreenSlider.value, 0f);
    }


    public void MoveBlue(Slider BlueSlider)
    {
        cube1.transform.position = startPositionCube1 + new Vector3
(0f, 0f, BlueSlider.value);
        sphere1.transform.position = startPositionSphere1 + new
Vector3
(0f, 0f, BlueSlider.value);
    }
```

Как видите, все три метода очень похожи друг на друга.

Одно из отличий заключается в разных названиях параметров.

Но методы работают независимо друг от друга и не могут использовать параметры друг друга.

К тому же, имена параметров – вещь условная (их придумывали мы сами).

Поэтому мы могли бы использовать одно и то же название параметра (например, `Slider MySlider`) во всех трёх методах.

Другое отличие методов заключается в том, что они используют разные векторы смещения:

```
new Vector3(RedSlider.value, 0f, 0f);
new Vector3(0f, GreenSlider.value, 0f);
new Vector3(0f, 0f, BlueSlider.value);
```

Вектор в методе «MoveRed» производит изменение координаты X.

Вектор в методе «MoveGreen» производит изменение координаты Y.

Вектор в методе «MoveBlue» производит изменение координаты Z.

Таким образом, у нас появилась возможность осуществлять программное перемещение объектов сцены вдоль любой из трёх осей координат.

А для того, чтобы такое перемещение мог осуществлять и пользователь нашего приложения, необходимо настроить два оставшихся ползунка.

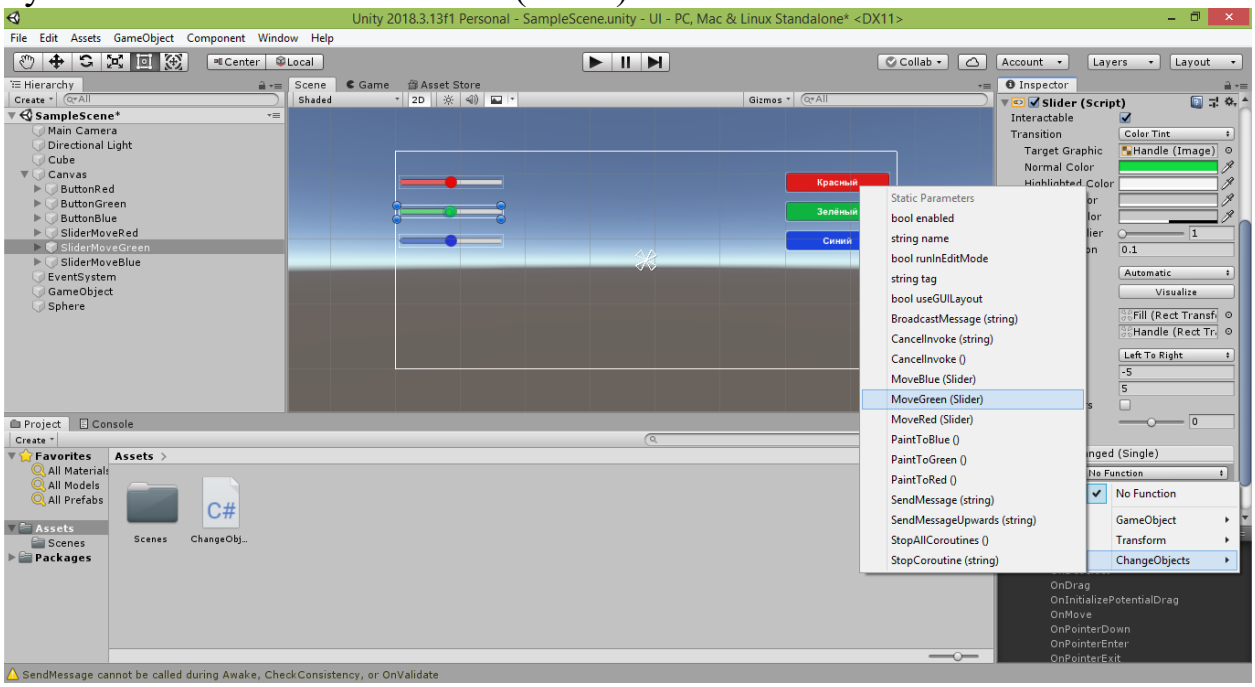
Сохраните код, нажав комбинацию клавиш CTRL+S, и перейдите в окно Unity.

Выполним действия, аналогичные тем, что делали с ползунком «SliderMoveRed». Выделите строку «SliderMoveGreen» в окне Hierarchy и справа в окне Inspector в группе «Slider (Script)» внизу пустого списка «On Value Changed (Single)» щёлкните на кнопке «+». В списке появится новый блок для задания обработчика события.

Перетяните строку «GameObject» из окна Hierarchy в поле блока, где написано «None (Object)», и отпустите. В поле появится новая надпись: «GameObject».

При этом станет доступен для изменения выпадающий список с надписью «No Function», расположенный рядом с полем.

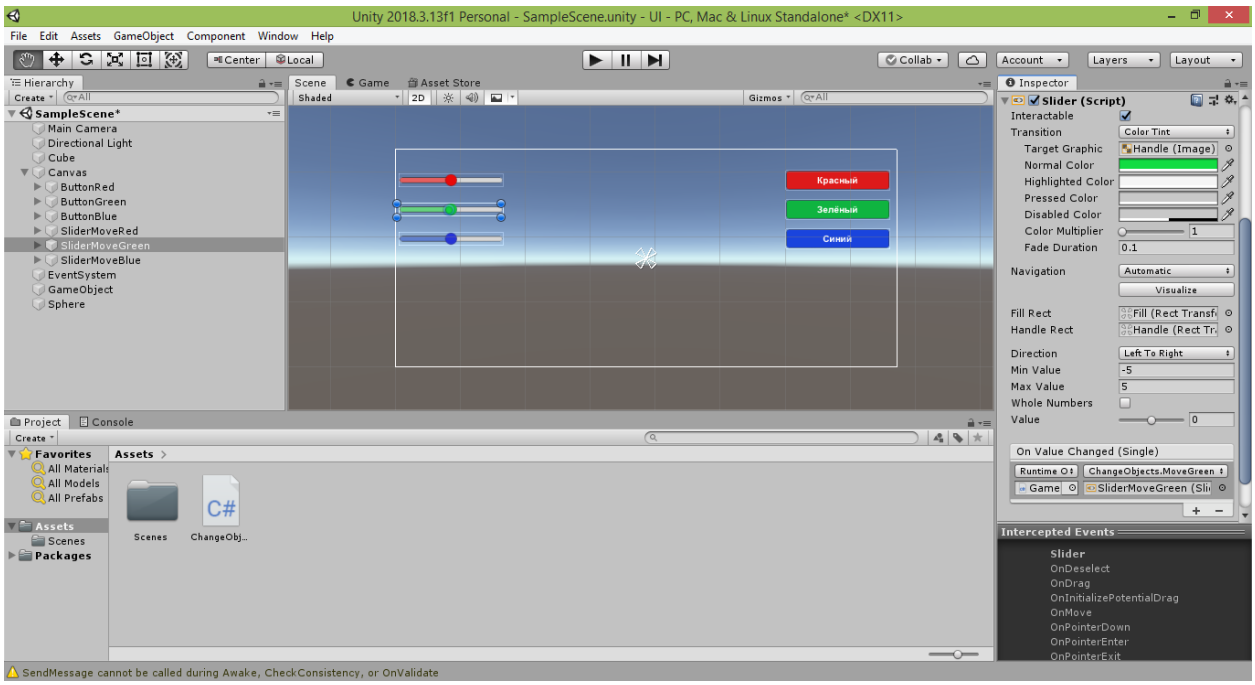
Раскройте выпадающий список и в подменю «ChangeObjects» выберите пункт с названием «MoveGreen (Slider)».



После этого под выпадающим списком появится новое поле с надписью «None (Slider)».

Перетяните строку «SliderMoveGreen» из окна Hierarchy в это поле и отпустите.

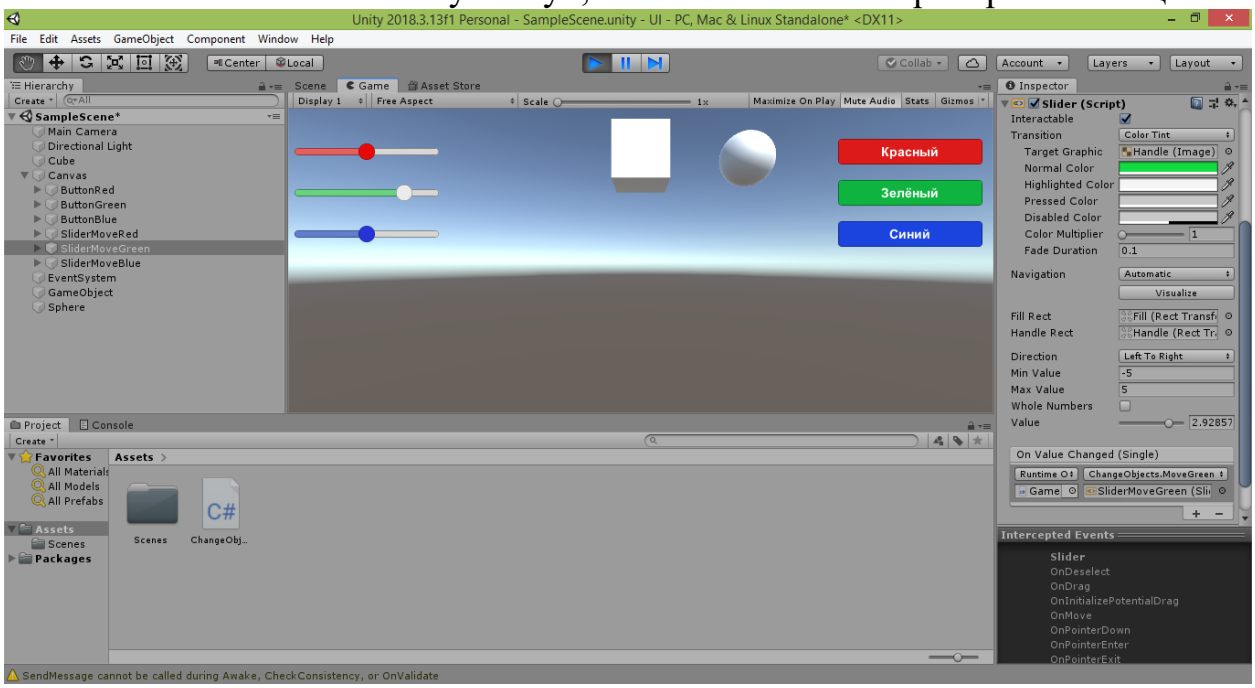
В поле появится новая надпись: «SliderMoveGreen (Slider)».



Запустите проект в игровом режиме, нажав кнопку «Play», и попробуйте переместить ручку зелёного ползунка.

В результате куб и сфера будут двигаться по вертикали (вдоль зелёной оси) в соответствии с перемещением ручки зелёного ползунка.

Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.



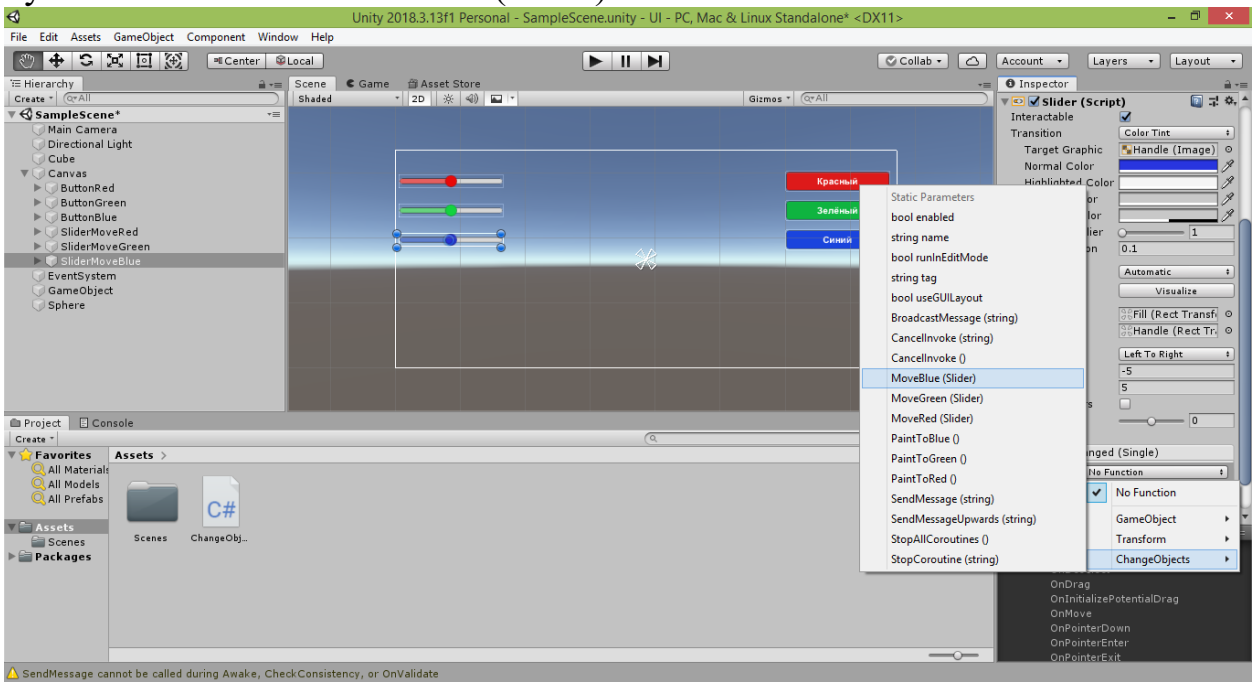
Далее настроим синий ползунок.

Выделите строку «SliderMoveBlue» в окне Hierarchy и справа в окне Inspector в группе «Slider (Script)» внизу пустого списка «On Value Changed (Single)» щёлкните на кнопке «+». В списке появится новый блок для задания обработчика события.

Перетяните строку «GameObject» из окна Hierarchy в поле блока, где написано «None (Object)», и отпустите. В поле появится новая надпись: «GameObject».

При этом станет доступен для изменения выпадающий список с надписью «No Function», расположенный рядом с полем.

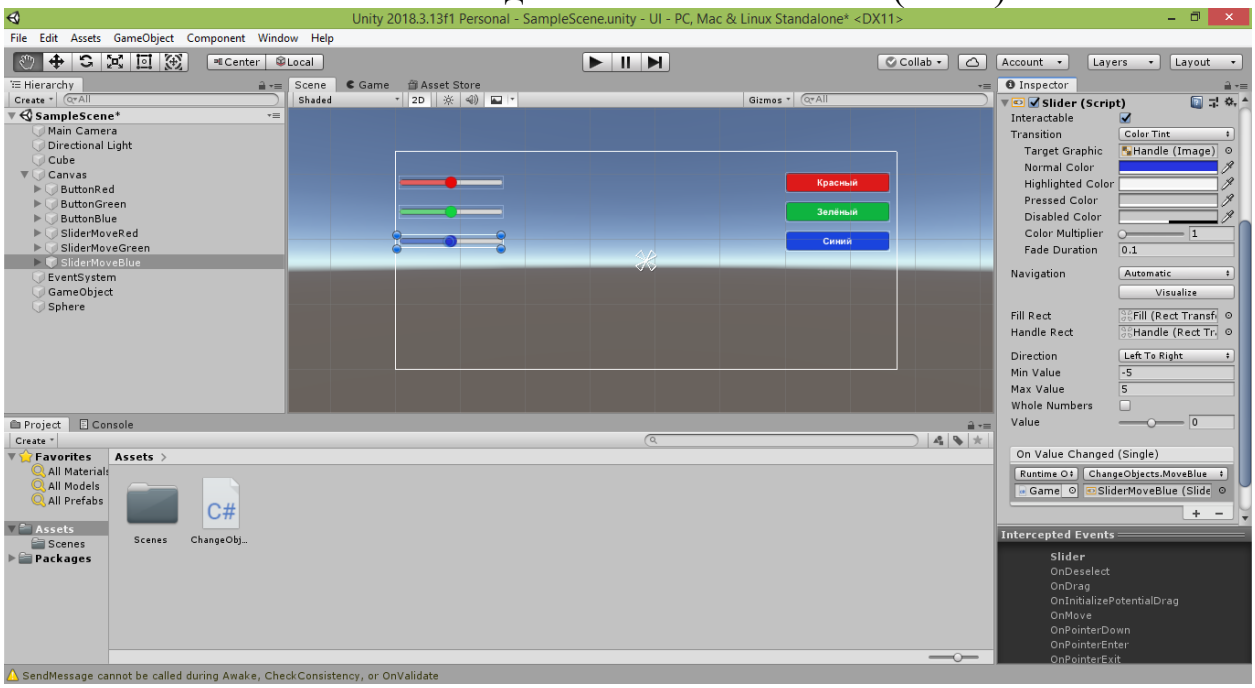
Раскройте выпадающий список и в подменю «ChangeObjects» выберите пункт с названием «MoveBlue (Slider)».



После этого под выпадающим списком появится новое поле с надписью «None (Slider)».

Перетяните строку «SliderMoveBlue» из окна Hierarchy в это поле и отпустите.

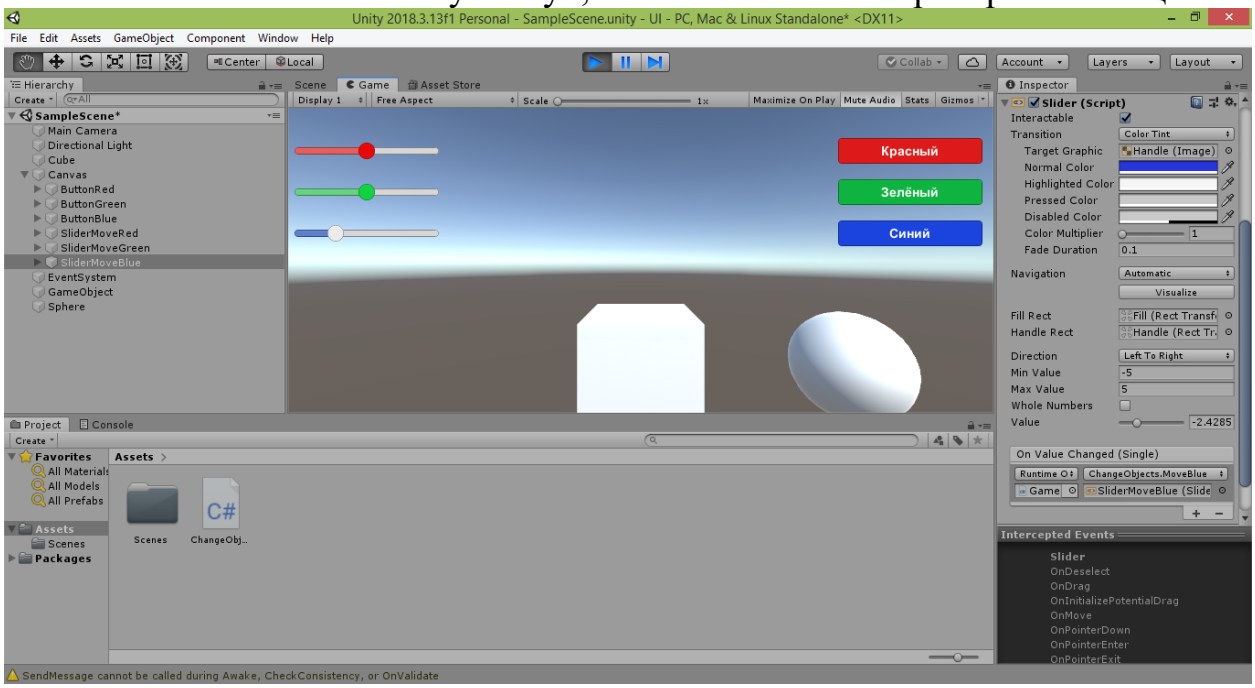
В поле появится новая надпись: «SliderMoveBlue (Slider)».



Запустите проект в игровом режиме, нажав кнопку «Play», и попробуйте переместить ручку синего ползунка.

В результате куб и сфера будут двигаться вперёд-назад (вдоль синей оси) в соответствии с перемещением ручки синего ползунка.

Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.



Отлично! Теперь мы можем не только менять цвет объектов, но и перемещать их!

Единственный недостаток нашей реализации (вероятно, вы обратили на него внимание), заключается в том, что каждый ползунок при движении его ручки сбрасывает изменения, выполненные ранее с помощью других ползунков.

Это связано с тем, что объекты сцены смещаются вдоль текущей оси, стартуя из точки их изначального местоположения.

Из-за этого теряются прежние смещения по остальным осям, которые до этого были заданы объектам.

Исправить такую ситуацию возможно, если программа нашего скрипта будет располагать информацией о положении ручек всех трёх ползунков.

Попробуем организовать это.

Один из вариантов – объявить три скрытые переменные, имеющие тип `float`, и записывать в каждую из них значение соответствующего ползунка при движении его ручки.

Однако вместо трёх переменных можно воспользоваться одной – вектором класса `Vector3` .

Напомню ещё раз, что вектор – это набор из нескольких чисел.

Класс `Vector3` описывает набор из трёх чисел.

В таком наборе мы и будем хранить информацию о положении ручек ползунков.

Добавим новую переменную-вектор в список уже существующих:

```
    private Vector3 startPositionCube1, startPositionSphere1,
slidersMove;
```

Переменная `slidersMove` (что можно перевести как «Ползунки перемещения») будет обновляться каждый раз при изменении положения ручек ползунков.

В самом начале работы программы ручки ползунков центрированы и их положению соответствуют значения, равные нулю.

Поэтому начальным значением для переменной `slidersMove` станет вектор, состоящий из нулей.

Его мы присвоим переменной в стандартном методе `Start` , с которого наше приложение начнёт свою работу:

```
    void Start()
    {
        startPositionCube1 = cube1.transform.position;
        startPositionSphere1 = sphere1.transform.position;
        slidersMove = new Vector3(0f, 0f, 0f);
    }
```

Обратите внимание, что если название переменной или метода состоит из нескольких слов, то каждое слово в названии удобно начинать с заглавной буквы.

Некоторые компьютерные системы (Unity, 1С и другие) используют заглавные буквы в названии, чтобы разбивать его на слова.

Вы наблюдали это процесс с пустым объектом «GameObject», у которого подписи к полям «Cube 1» и «Sphere 1» формировались автоматически на основе названий открытых переменных `cube1` и `sphere1` , объявленных в связанном с ним скрипте.

Также обратите внимание, что все переменные я называю со строчной буквы, а методы – с заглавной.

Это не обязательное правило, но профессиональные программисты его придерживаются, чтобы переменные и методы было проще отличать по их названию при чтении кода.

После того, как мы подготовили переменную для хранения информации о положении ручек ползунков, обновим содержимое методов, отвечающих за перемещение:

```
    public void MoveRed(Slider RedSlider)
    {
        slidersMove.x = RedSlider.value;

        cube1.transform.position = startPositionCube1 + slidersMove;
        sphere1.transform.position = startPositionSphere1 +
slidersMove;
```

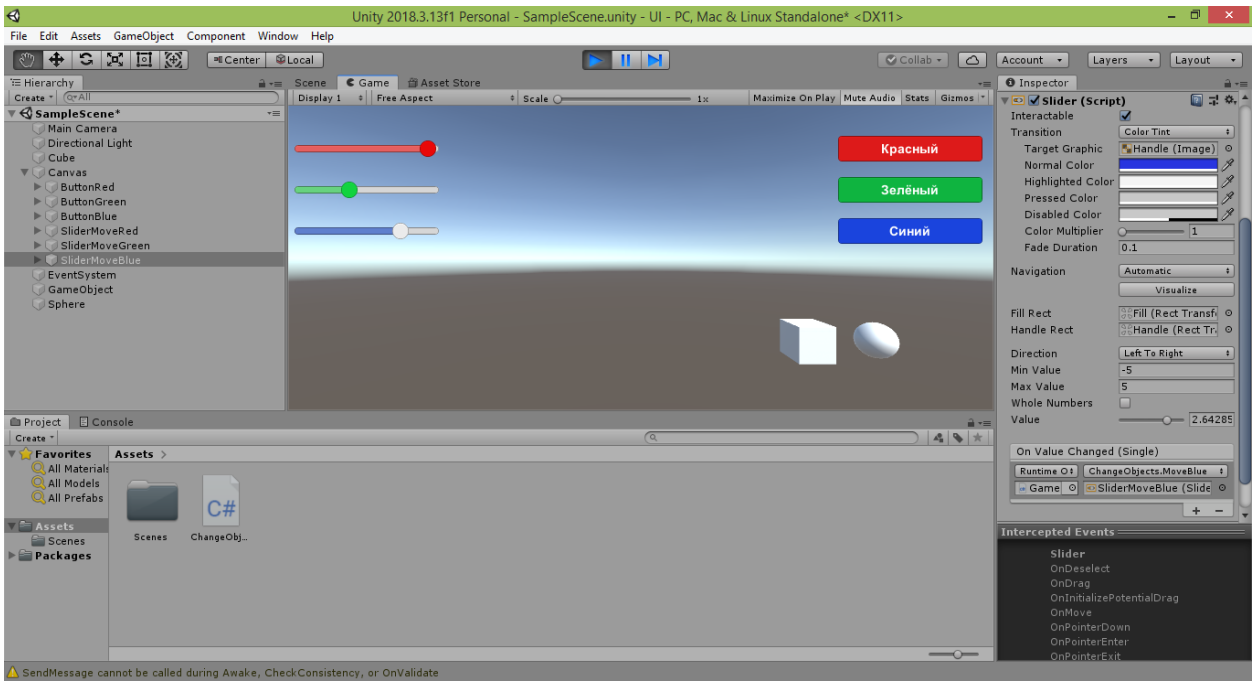
```
    }


    public void MoveGreen(Slider GreenSlider)
    {
        slidersMove.y = GreenSlider.value;

        cube1.transform.position = startPositionCube1 + slidersMove;
        sphere1.transform.position = startPositionSphere1 +
slidersMove;
    }


    public void MoveBlue(Slider BlueSlider)
    {
        slidersMove.z = BlueSlider.value;

        cube1.transform.position = startPositionCube1 + slidersMove;
        sphere1.transform.position = startPositionSphere1 +
slidersMove;
    }
```

Поскольку переменная `slidersMove` имеет тип, описываемый классом `Vector3` , у неё доступны три компоненты `x` , `y` и `z`. В них мы записываем новые значения положения ручек красного, зелёного и синего ползунков, соответственно. Запись (обновление) значения происходит для того ползунка, ручка которого пришла в движение.

Оставшиеся строки отвечают за расчёт нового местоположения (`position`) объектов сцены, которое определяется путём сложения двух векторов: к компонентам `x` , `y` и `z` вектора начального местоположения объекта прибавляются соответствующие компоненты (положения ручек ползунков) вектора смещения `slidersMove` :

```
        cube1.transform.position = startPositionCube1 + slidersMove;
        sphere1.transform.position = startPositionSphere1 +
slidersMove;
```

Проверим работу нового варианта кода, сохранив его нажатием комбинации клавиш CTRL+S и перейдя в окно Unity. Запустите проект в игровом режиме, нажав кнопку «Play», и измените положение ручек всех трёх ползунков.



Теперь куб и сфера не возвращаются в исходное местоположение, когда пользователь перемещает их по очередной оси координат.

Таким образом, ползунки больше не конкурируют друг с другом за смещение по осям координат, а действуют согласованно, позволяя переместить объекты в любую точку сцены (например, в один из углов).

Нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Обратите внимание, что последние две строки, которые мы прописали, у всех трёх методов, одинаковы.

В профессиональном программировании копирование кода не приветствуется, поскольку код программы становится более громоздким и трудночитаемым.

Наличие повторяющегося кода является признаком того, что его можно сократить, заменив обобщённым шаблоном.

В роли шаблона обычно выступают вспомогательные методы.

Предлагаю сделать наш код более компактным и читабельным, добавив в него новый метод:

```
    void MoveObjects()
    {
        cube1.transform.position = startPositionCube1 + slidersMove;
        sphere1.transform.position = startPositionSphere1 +
slidersMove;
    }
```

Данный метод включает в себя те самые две строки кода, которые мы растиражировали в методах `MoveRed` , `MoveGreen` и `MoveBlue` .

Теперь в каждом из этих методов мы можем убрать две строки, заменив их одной:

```
    public void MoveRed(Slider RedSlider)
    {
```

```
        slidersMove.x = RedSlider.value;
        MoveObjects();
    }


    public void MoveGreen(Slider GreenSlider)
    {
        slidersMove.y = GreenSlider.value;
        MoveObjects();
    }


    public void MoveBlue(Slider BlueSlider)
    {
        slidersMove.z = BlueSlider.value;
        MoveObjects();
    }
```

Как видите, код стал более лаконичным.

Мы видим, что в каждом методе у переменной `slidersMove` происходит обновление информации о положении ручки ползунка, вызвавшего метод.

После этого происходит перемещение объектов по сцене (за это отвечает метод `MoveObjects`).

Сохраните код, нажав комбинацию клавиш CTRL+S.

Затем перейдите в окно Unity и убедитесь, что приложение по-прежнему работает корректно – ползунки согласованно управляют перемещением объектов по сцене.

Обратите внимание, что метод `MoveObjects` я объявил без модификатора доступа `public`, что равносильно его объявлению с модификатором доступа `private` (скрытый).

Как уже было сказано, данный метод является вспомогательным в нашем скрипте.

Мы не собираемся связывать его с какими-либо элементами пользовательского интерфейса.

Поэтому нет смысла делать его открытым – это только добавит лишнюю строку в список доступных обработчиков событий.

Несмотря на его скрытость, метод `MoveObjects` доступен внутри класса `ChangeObjects` нашего скрипта, в чём мы только что убедились, использовав его в трёх других методах.

По этой же причине в пределах всего класса `ChangeObjects` доступна переменная `slidersMove` , которую мы также объявили скрытой, но смогли напрямую использовать в нескольких методах без передачи в качестве параметра.

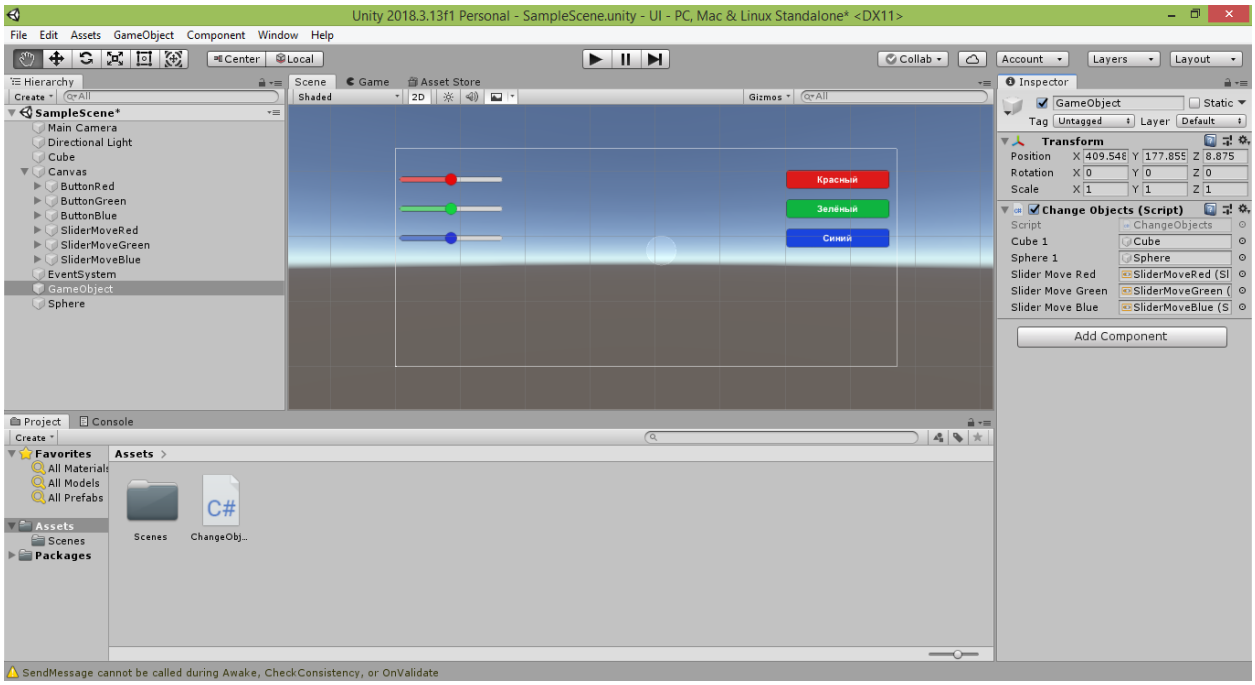
Важное замечание. Переменная `slidersMove` оказалась доступной всем методам, поскольку мы объявили её за их пределами. Такие переменные называются глобальными.

Если бы мы объявили переменную `slidersMove` внутри одного из методов (например, в методе `MoveObjects`), то она превратилась бы в локальную переменную.

Локальные переменные доступны только в пределах того метода, в котором они были объявлены (попытка их использования в других методах приведёт к ошибке).

Следует заметить, что ещё одним способом передавать в программу скрипта информацию о положении ручек ползунков, является объявление трёх открытых переменных типа `Slider` , которые затем следует связать с соответствующими ползунками через три новых поля, появившихся в свойствах пустого объекта «GameObject»:

```
public GameObject cube1, sphere1;
public Slider sliderMoveRed, sliderMoveGreen, sliderMoveBlue;
```

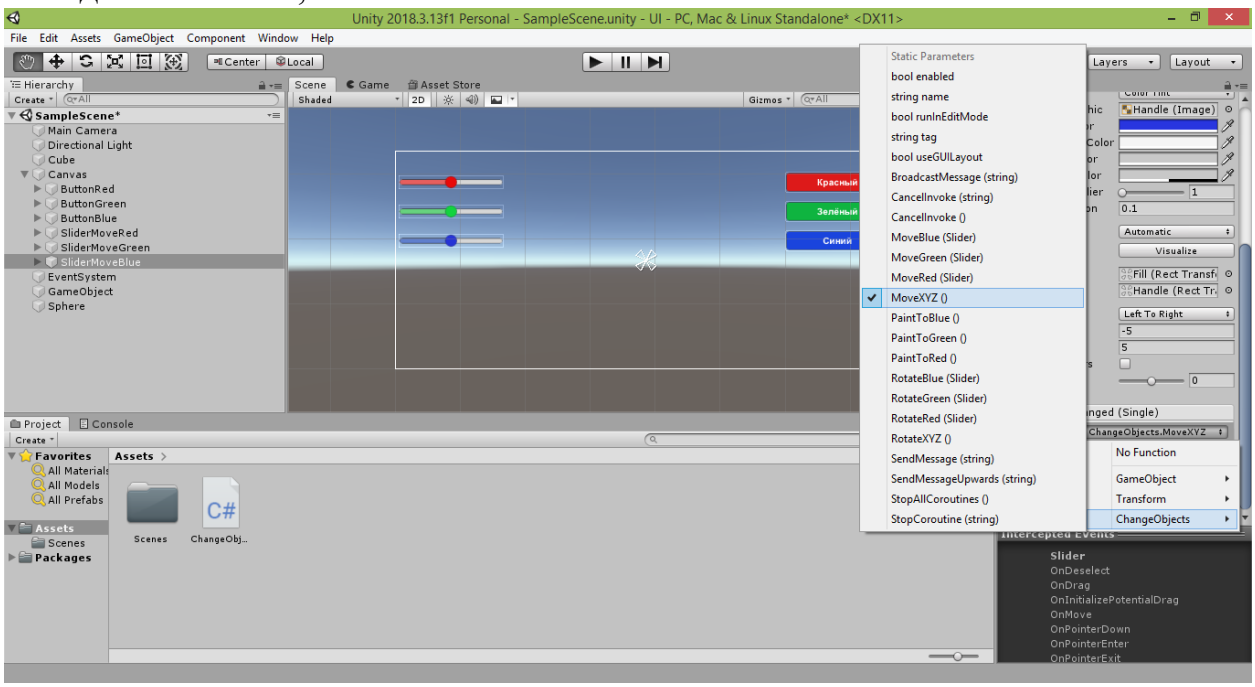


В этом случае нам даже не потребуются методы `MoveRed` , `MoveGreen` и `MoveBlue`. Вместо них мы создадим единый универсальный метод `MoveXYZ` , который будет заносить в переменную `slidersMove` информацию о текущем состоянии ручек ползунков, а затем вызывать метод `MoveObjects` для перемещения объектов:

```
public void MoveXYZ()
{
    slidersMove.x = sliderMoveRed.value;
```

```
        slidersMove.y = sliderMoveGreen.value;
        slidersMove.z = sliderMoveBlue.value;

        MoveObjects();
    }
```

Метод `MoveXYZ` был создан открытым (`public`), чтобы его можно было использовать в качестве обработчика события «OnValueChanged» у каждого из трёх ползунков. Зададим ползункам данный метод обработки вместо методов `MoveRed` , `MoveGreen` и `MoveBlue` .



После этого можно запустить проект в игровом режиме и убедиться, что новый вариант кода управляет объектами сцены так же, как и раньше.

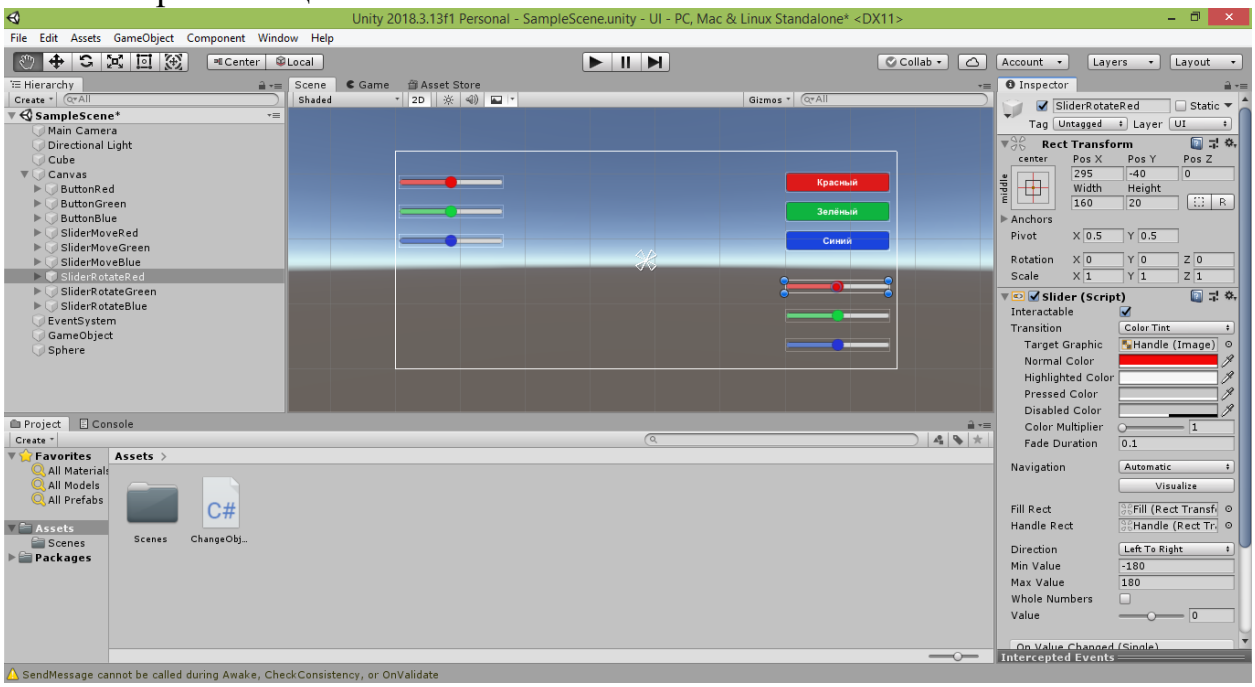
Прежде чем завершить рассмотрение приёмов работы с ползунками, попробуем запрограммировать ещё один их набор для управления поворотом объектов.

Добавьте на полотно три ползунка путём копирования ранее созданных или путём добавления новых при помощи команды «GameObject → UI → Slider» в меню Unity.

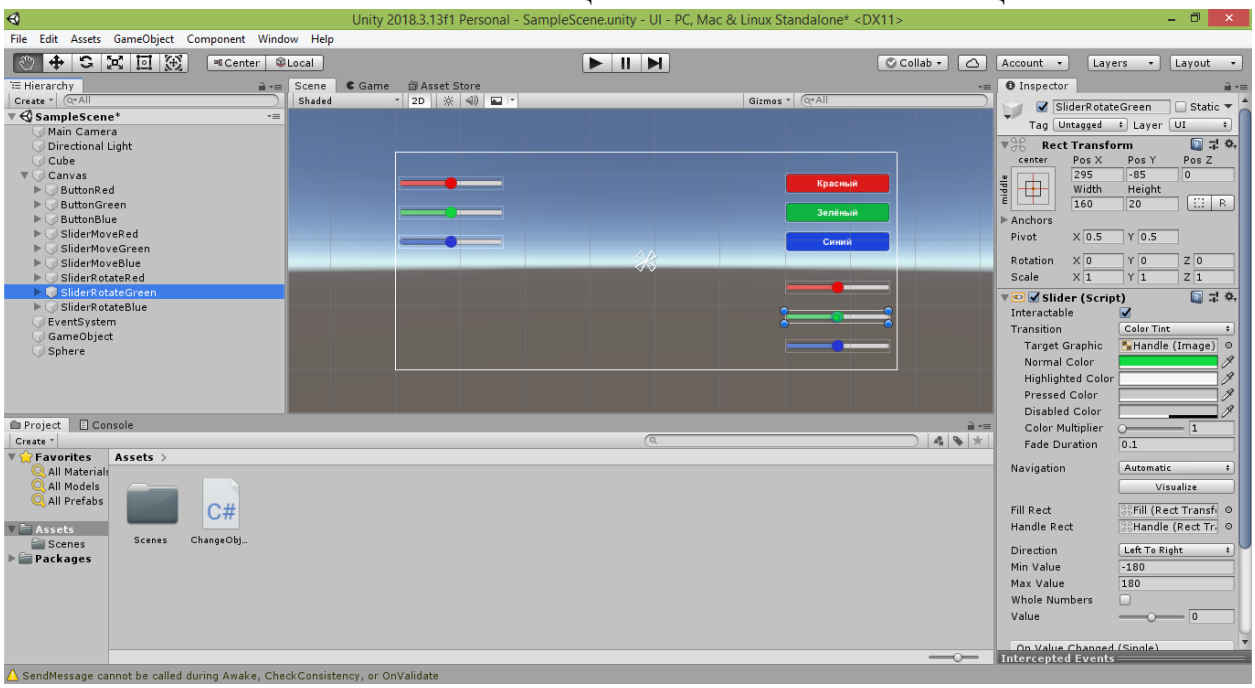
Одному ползунку дайте имя «SliderRotateRed», другому – «SliderRotateGreen», третьему – «SliderRotateBlue».

У ползунка «SliderRotateRed» измените значение свойства «Pos X» на 295, а значение свойства «Pos Y» – на 120. Свойству «Normal Color» ползунка задайте красный цвет. Свойству «Min Value» задайте значение -180, а свойству «Max Value» задайте значение 180, значение свойства «Value» оставьте равным 0. Далее, найдя слева в окне Hierarchy строку «Fill Area», подчинённую

этому ползунку, раскройте её и уже у её подчинённой строки «Fill» справа в окне Inspector выберите в палитре у свойства «Color» не слишком насыщенный оттенок красного цвета.

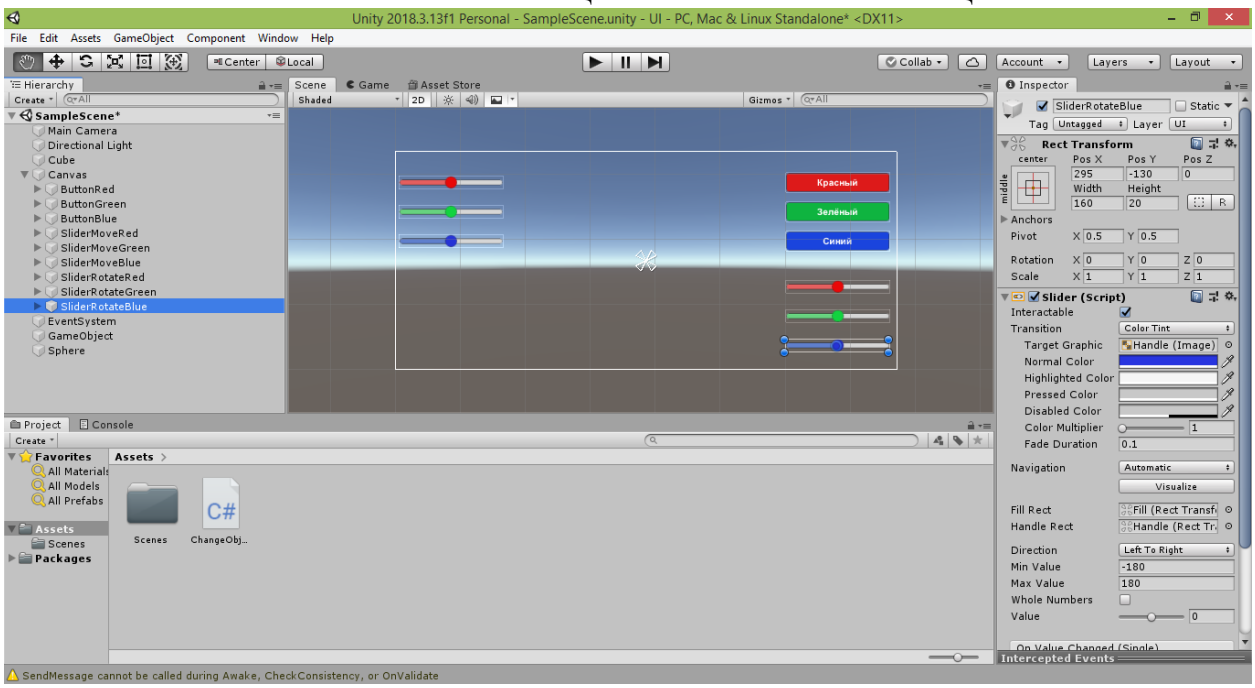


У ползунка «SliderRotateGreen» измените значение свойства «Pos X» на 295, а значение свойства «Pos Y» – на 75. Свойству «Normal Color» ползунка задайте зелёный цвет. Свойству «Min Value» задайте значение -180, а свойству «Max Value» задайте значение 180. Далее, найдя слева в окне Hierarchy строку «Fill Area», подчинённую этому ползунку, раскройте её и уже у её подчинённой строки «Fill» справа в окне Inspector выберите в палитре у свойства «Color» не слишком насыщенный оттенок зелёного цвета.



У ползунка «SliderRotateBlue» измените значение свойства «Pos X» на 295, а значение свойства «Pos Y» – на 30. Свойству «Normal Color» ползунка

задайте синий цвет. Свойству «Min Value» задайте значение -180, а свойству «Max Value» задайте значение 180. Далее, найдя слева в окне Hierarchy строку «Fill Area», подчинённую этому ползунку, раскройте её и уже у её подчинённой строки «Fill» справа в окне Inspector выберите в палитре у свойства «Color» не слишком насыщенный оттенок синего цвета.



Теперь приступим к программированию созданных ползунков.

Делать это мы будем в том же скрипте ChangeObjects, в котором мы программировали предыдущие три ползунка и кнопки, отвечавшие за перемещение объектов по сцене.

Как ранее уже отмечалось, для новых ползунков можно было бы создать отдельный скрипт, но для данной задачи проще все изменения объектов программировать в одном скрипте-классе, не разбивая программу на части.

Зайдя в скрипт в редакторе Visual Studio (или в любом текстовом редакторе наподобие программы «Блокнот» / «Notepad»), добавим новую переменную-вектор `slidersRotate` в список уже существующих:

```
private Vector3 startPositionCube1, startPositionSphere1, slidersMove, slidersRotate;
```

Переменная `slidersRotate` (что можно перевести как «Ползунки поворота») будет обновляться каждый раз при изменении положения ручек новых ползунков.

В самом начале работы программы ручки ползунков центрированы и их положению соответствуют значения, равные нулю.

Поэтому для переменной `slidersMove` начальным значением станет вектор, состоящий из нулей.

Его мы присвоим переменной в стандартном методе `Start` , с которого наше приложение начнёт свою работу:

```
    void Start()
    {
        startPositionCube1 = cube1.transform.position;
        startPositionSphere1 = sphere1.transform.position;
        slidersMove = new Vector3(0f, 0f, 0f);

        slidersRotate = new Vector3(0f, 0f, 0f);
    }
```

Также нам потребуется объявить две новые скрытые переменные `startAnglesCube1` и `startAnglesSphere1`, имеющие тип `Quaternion`:

```
private Vector3 startPositionCube1, startPositionSphere1, slidersMove, slidersRotate;
private Vector3 startAnglesCube1, startAnglesSphere1;
```

Названия переменных `startAnglesCube1` и `startAnglesSphere1` мы выбрали сами (они переводятся как «Начальные углы Куба 1» и «Начальные углы Сферы 1»).

Скрытыми (`private`) эти переменные мы объявили, потому что они будут использоваться только внутри скрипта (мы не будем переходить в Unity и привязывать к ним объекты сцены).

В качестве типа объявленных переменных мы указали класс `Vector3`, что позволит нам хранить в них наборы из трёх углов, на которые оказались повёрнуты объекты сцены в самом начале работы программы.

Само запоминание начального поворота каждого объекта должно происходить при запуске сцены.

Поэтому в стандартном методе `Start` мы будем задавать новым переменным текущие повороты соответствующих им объектов:

```
    void Start()
    {
        startPositionCube1 = cube1.transform.position;
        startPositionSphere1 = sphere1.transform.position;
        slidersMove = new Vector3(0f, 0f, 0f);


        startAnglesCube1 = new Vector3(cube1.transform.rotation.x, cube1.transform.rotation.y, cube1.transform.rotation.z);
        startAnglesSphere1 = new Vector3(sphere1.transform.rotation.x, sphere1.transform.rotation.y, sphere1.transform.rotation.z);

        slidersRotate = new Vector3(0f, 0f, 0f);
    }
```

Как видите, в переменные `startAnglesCube1` и `startAnglesSphere1` записываются новые вектора, составленные из углов поворота объектов вокруг трёх осей координат. Узнать значения указанных углов (`x` , `y` и `z`) можно через свойство `rotation` компонента `transform` объекта сцены.

Приведённый код является достаточно громоздким.

Можно сделать его более читабельным, если начальные углы поворота объектов записывать в переменные-вектора не напрямую, а посредством вспомогательных переменных `startRotationCube1` и `startRotationSphere1`, имеющих тип `Quaternion`:

```
    void Start()
    {
        startPositionCube1 = cube1.transform.position;
        startPositionSphere1 = sphere1.transform.position;
        slidersMove = new Vector3(0f, 0f, 0f);


        Quaternion startRotationCube1 = cube1.transform.rotation;
        Quaternion startRotationSphere1 = sphere1.transform.rotation;

        startAnglesCube1 = new Vector3(startRotationCube1.x,
startRotationCube1.y, startRotationCube1.z);
        startAnglesSphere1 = new Vector3(startRotationSphere1.x,
startRotationSphere1.y, startRotationSphere1.z);

        slidersRotate = new Vector3(0f, 0f, 0f);
    }
```

Переменная `startRotationCube1` была добавлена лишь для того, чтобы заменить цепочку `cube1.transform.rotation` одним словом.

Аналогично переменная `startRotationSphere1` была добавлена для того, чтобы заменить одним словом цепочку `sphere1.transform.rotation`.

Названия переменных `startRotationCube1` и `startRotationSphere1` мы выбрали сами (они переводятся как «Начальный поворот Куба 1» и «Начальный поворот Сферы 1»).

В качестве типа объявленных переменных мы указали класс `Quaternion`.

Дело в том, что поворот объектов сцены, хранящийся в свойстве `rotation`, представлен в виде набора из трёх кватернионов (окружностей поворота вокруг осей координат) и имеет тип `Quaternion`.

Поэтому набор начальных поворотов объекта сцены мы сможем сохранить только в переменную, имеющую тип `Quaternion`.

Далее в коде мы обращаемся к объявленным вспомогательным переменным, чтобы получить значения углов поворота `x`, `y`, `z` и сформировать из них вектора, которые сохраняются в переменные `startAnglesCube1` и `startAnglesSphere1`.

После того, как мы запомнили начальные углы поворота, добавим три открытых метода, которые обеспечат поворот объектов вокруг осей координат:

```
    public void RotateRed(Slider RedSlider)
    {
        cube1.transform.rotation = Quaternion.Euler(startAnglesCube1 +
new
Vector3(RedSlider.value, 0f, 0f));

        sphere1.transform.rotation =
Quaternion.Euler(startAnglesSphere1 + new
Vector3(RedSlider.value, 0f, 0f));
    }

    public void RotateGreen(Slider GreenSlider)
    {
        cube1.transform.rotation = Quaternion.Euler(startAnglesCube1 +
new
Vector3(0f, GreenSlider.value, 0f));

        sphere1.transform.rotation =
Quaternion.Euler(startAnglesSphere1 + new
Vector3(0f, GreenSlider.value, 0f));
    }


    public void RotateBlue(Slider BlueSlider)
    {
        cube1.transform.rotation = Quaternion.Euler(startAnglesCube1 +
new
Vector3(0f, 0f, BlueSlider.value));

        sphere1.transform.rotation =
Quaternion.Euler(startAnglesSphere1 + new
Vector3(0f, 0f, BlueSlider.value));
    }
```

Код, прописанный в методах «RotateRed», «RotateGreen» и «RotateBlue», является аналогом первого варианта кода методов «MoveRed», «MoveGreen» и «MoveBlue».

Как вы помните, тот код приводил к рассогласованной работе ползунков: объекты начинали перемещаться со своего первоначального местоположения, сбрасывая перемещения, выполненные при помощи других ползунков.

В случае с поворотом объектов будет происходить примерно то же самое.

Давайте проанализируем, почему.

Все три метода поворота – «RotateRed», «RotateGreen» и «RotateBlue» – очень похожи друг на друга.

Одно из отличий заключается в том, что их параметрам мы дали разные имена: `RedSlider` , `GreenSlider` , и `BlueSlider` .

Однако мы могли бы использовать и одинаковые имена параметров (например, `Slider MySlider`), поскольку методы работают независимо друг от друга и не видят параметры друг друга.

Так что указанное различие методов условно и не является существенным.

Более существенным различием является то, что в методах используются разные наборы углов поворота:

```
new Vector3(RedSlider.value, 0f, 0f);
new Vector3(0f, GreenSlider.value, 0f);
new Vector3(0f, 0f, BlueSlider.value);
```

Вектор в методе «RotateRed» производит поворот объекта вокруг красной оси.

Вектор в методе «RotateGreen» производит поворот объекта вокруг зелёной оси.

Вектор в методе «RotateBlue» производит поворот объекта вокруг синей оси.

Указанные векторы складываются с векторами `startAnglesCube1` и `startAnglesSphere1` , в которых содержатся углы начального поворота куба и сферы. При сложении двух векторов образуется итоговый вектор поворота, который и передаётся в команду `Euler` , вычисляющую новый набор углов поворота.

В свою очередь, команда `Euler` возвращает набор кватернионов – значение, описываемое классом `Quaternion` .

Данный набор записывается в свойство `rotation` .

В результате происходит изменение одного из углов поворота объекта (остальные два угла не изменяются, поскольку к ним прибавляются нули).

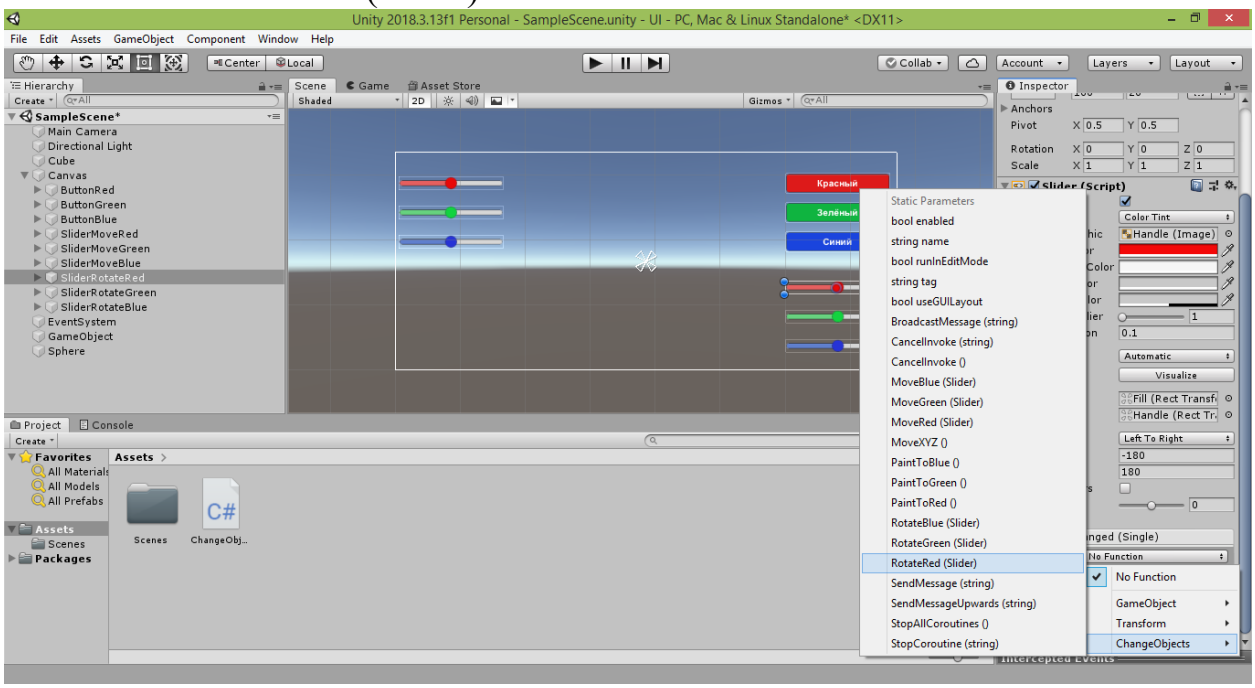
Чтобы проверить работу кода, потребуется задать созданные открытые методы в качестве обработчиков событий изменения положения ручек ползунков.

Сохраните код, нажав комбинацию клавиш CTRL+S, и перейдите в окно Unity.

Выделите строку «SliderRotateRed» в окне Hierarchy и справа в окне Inspector в группе «Slider (Script)» внизу пустого списка «On Value Changed (Single)» щёлкните на кнопке «+». В списке появится новый блок для задания обработчика события.

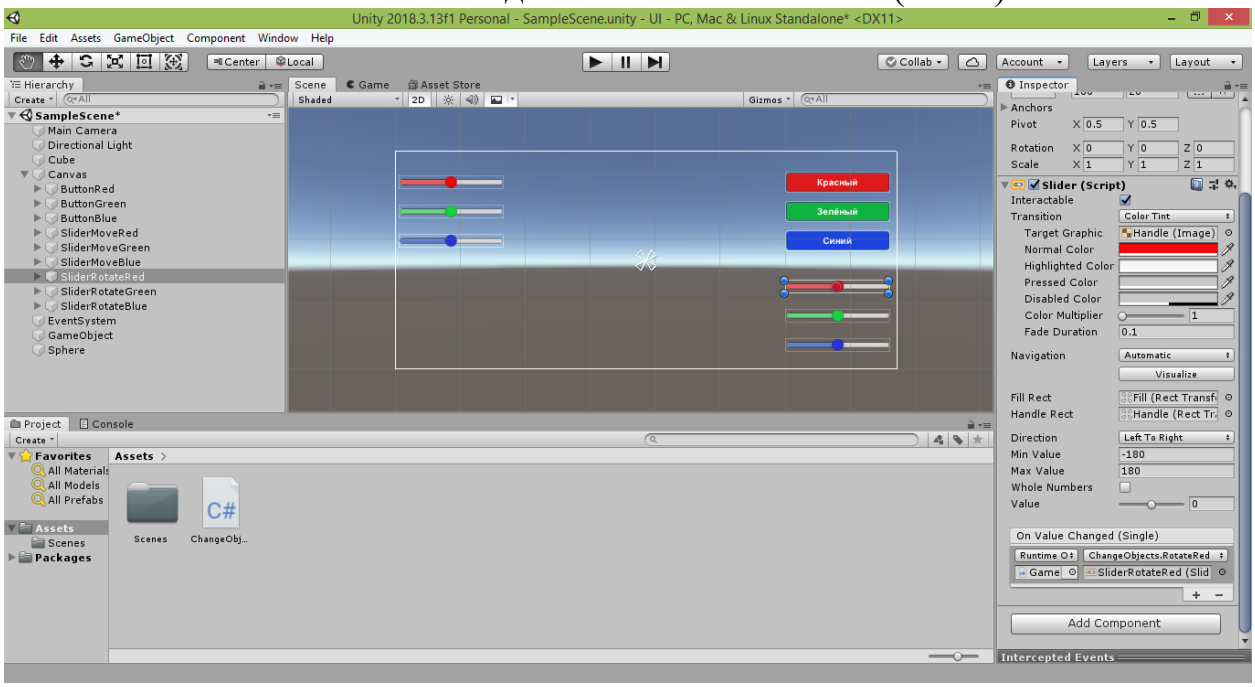
Перетяните строку «GameObject» из окна Hierarchy в поле блока, где написано «None (Object)», и отпустите. В поле появится новая надпись: «GameObject».

При этом станет доступен для изменения выпадающий список с надписью «No Function», расположенный рядом с полем. Раскройте выпадающий список и в подменю «ChangeObjects» выберите пункт с названием «RotateRed (Slider)».



После этого под выпадающим списком появится новое поле с надписью «None (Slider)». Перетяните строку «SliderRotateRed» из окна Hierarchy в это поле и отпустите.

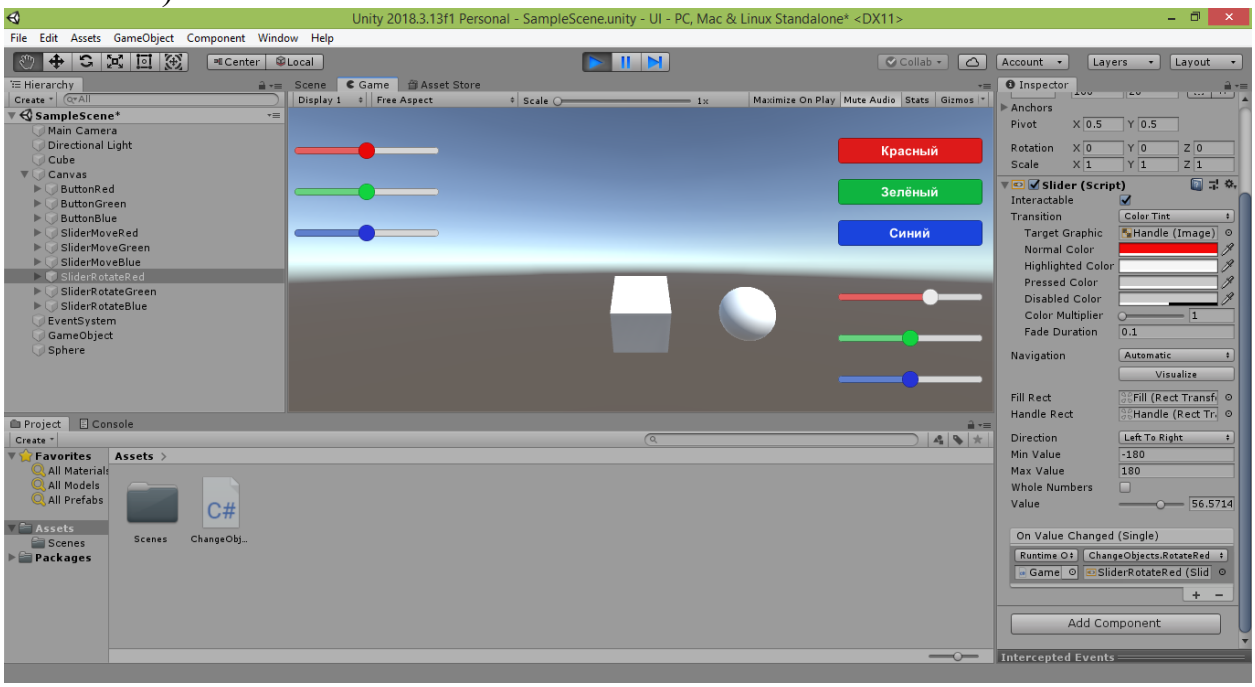
В поле появится новая надпись: «SliderRotateRed (Slider)».



Запустите проект в игровом режиме, нажав кнопку «Play», и попробуйте переместить ручку красного ползунка.

В результате куб и сфера будут поворачиваться вокруг красной горизонтальной оси в соответствии с перемещением ручки красного ползунка.

Следует заметить, что вращение сферы видно не будет (чтобы его увидеть, нужно превратить сферу в эллипсоид, вытянув её вдоль зелёной или синей оси).



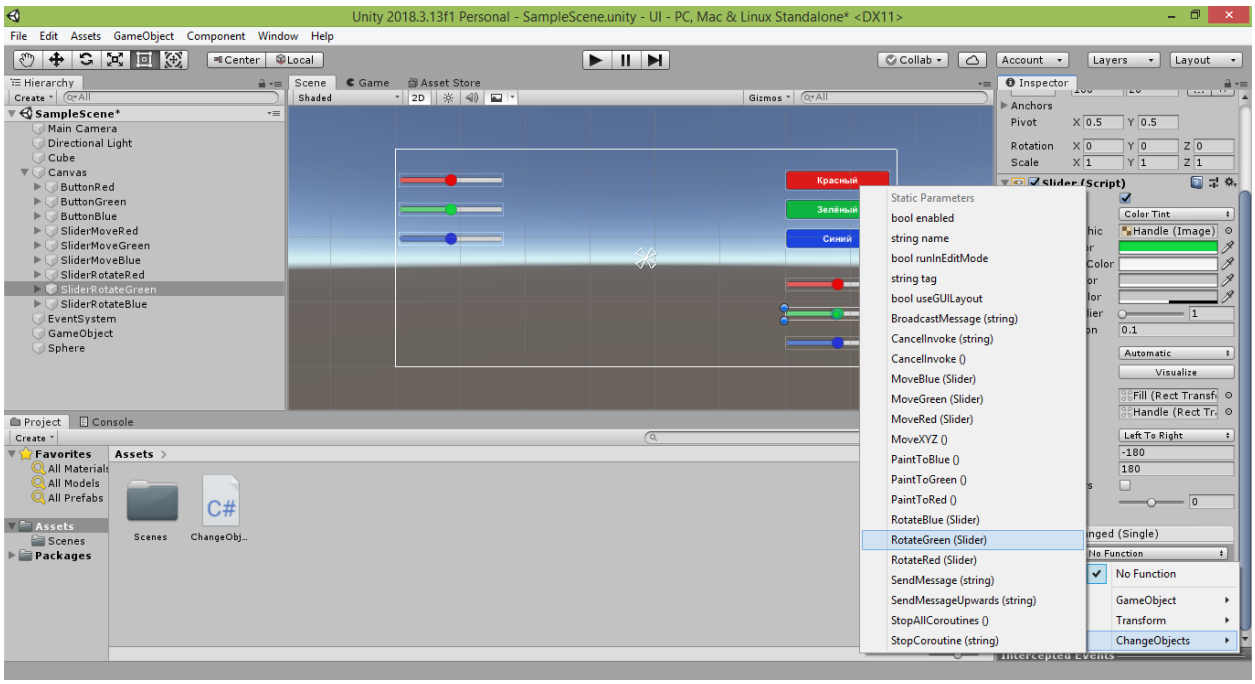
Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Выделите строку «SliderRotateGreen» в окне Hierarchy и справа в окне Inspector в группе «Slider (Script)» внизу пустого списка «On Value Changed (Single)» щёлкните на кнопке «+». В списке появится новый блок для задания обработчика события.

Перетяните строку «GameObject» из окна Hierarchy в поле блока, где написано «None (Object)», и отпустите. В поле появится новая надпись: «GameObject».

При этом станет доступен для изменения выпадающий список с надписью «No Function», расположенный рядом с полем.

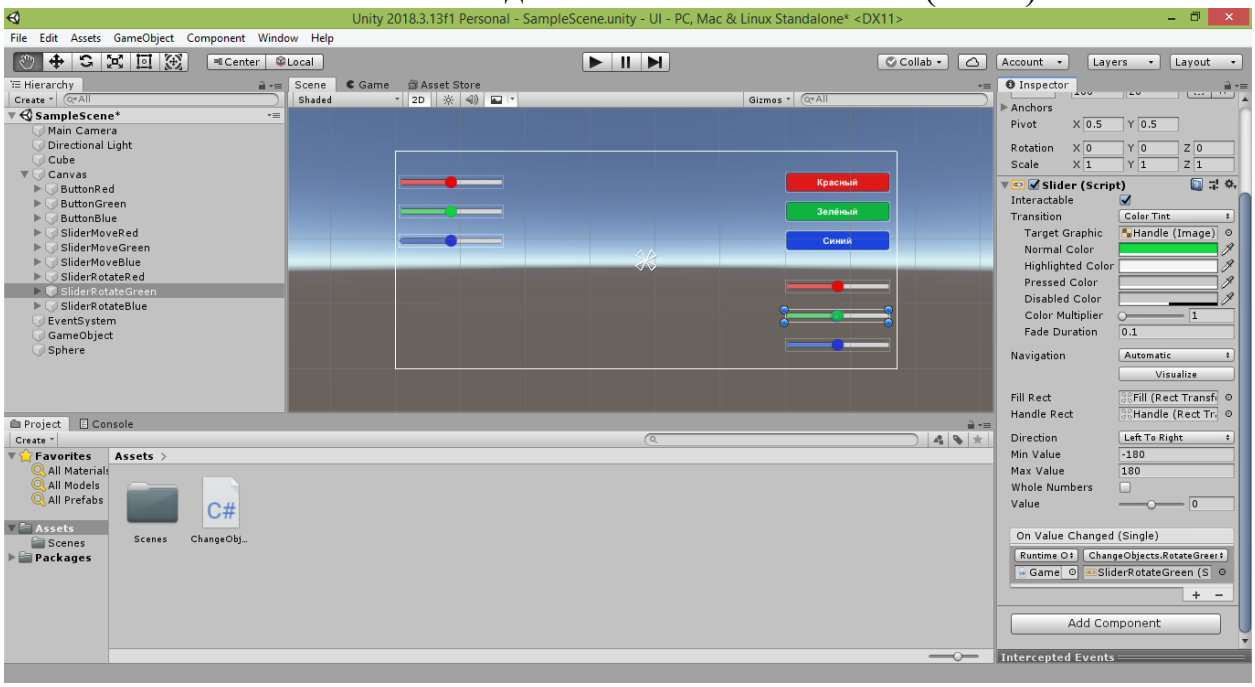
Раскройте выпадающий список и в подменю «ChangeObjects» выберите пункт с названием «RotateGreen (Slider)».



После этого под выпадающим списком появится новое поле с надписью «None (Slider)».

Перетяните строку «SliderRotateGreen» из окна Hierarchy в это поле и отпустите.

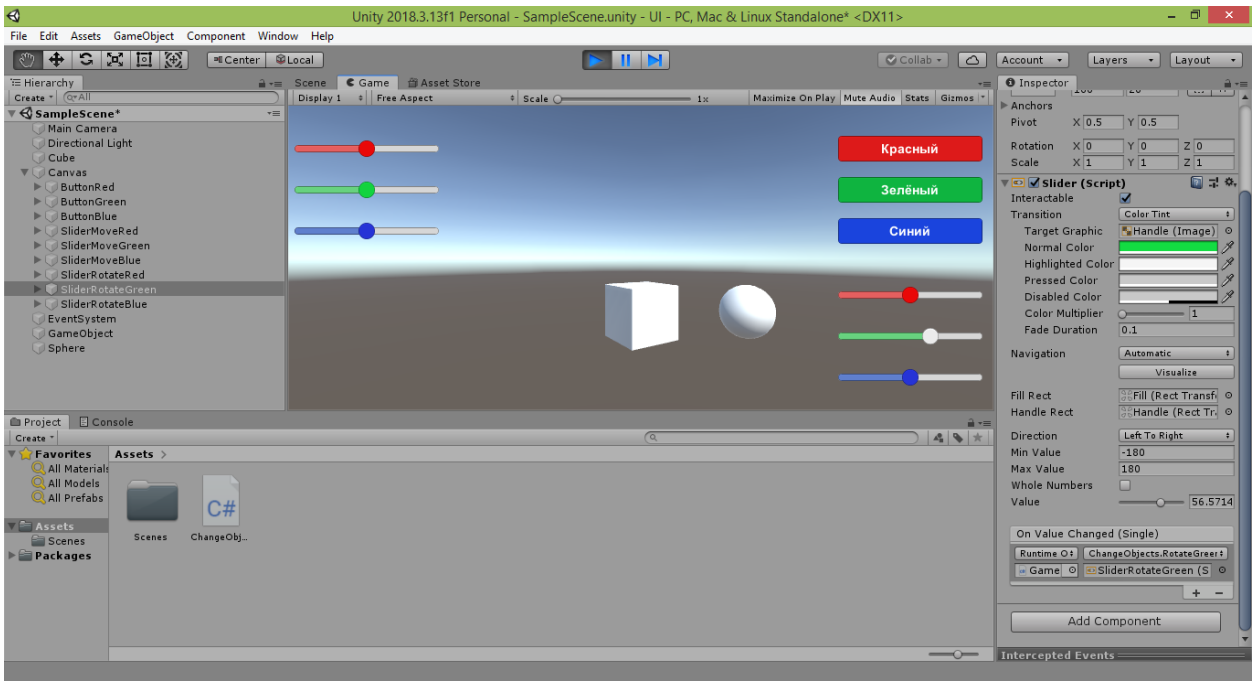
В поле появится новая надпись: «SliderRotateGreen (Slider)».



Запустите проект в игровом режиме, нажав кнопку «Play», и попробуйте переместить ручку зелёного ползунка.

В результате куб и сфера будут поворачиваться вокруг зелёной вертикальной оси в соответствии с перемещением ручки зелёного ползунка.

Следует заметить, что вращение сферы видно не будет (чтобы его увидеть, нужно превратить сферу в эллипсоид, вытянув её вдоль красной или синей оси).



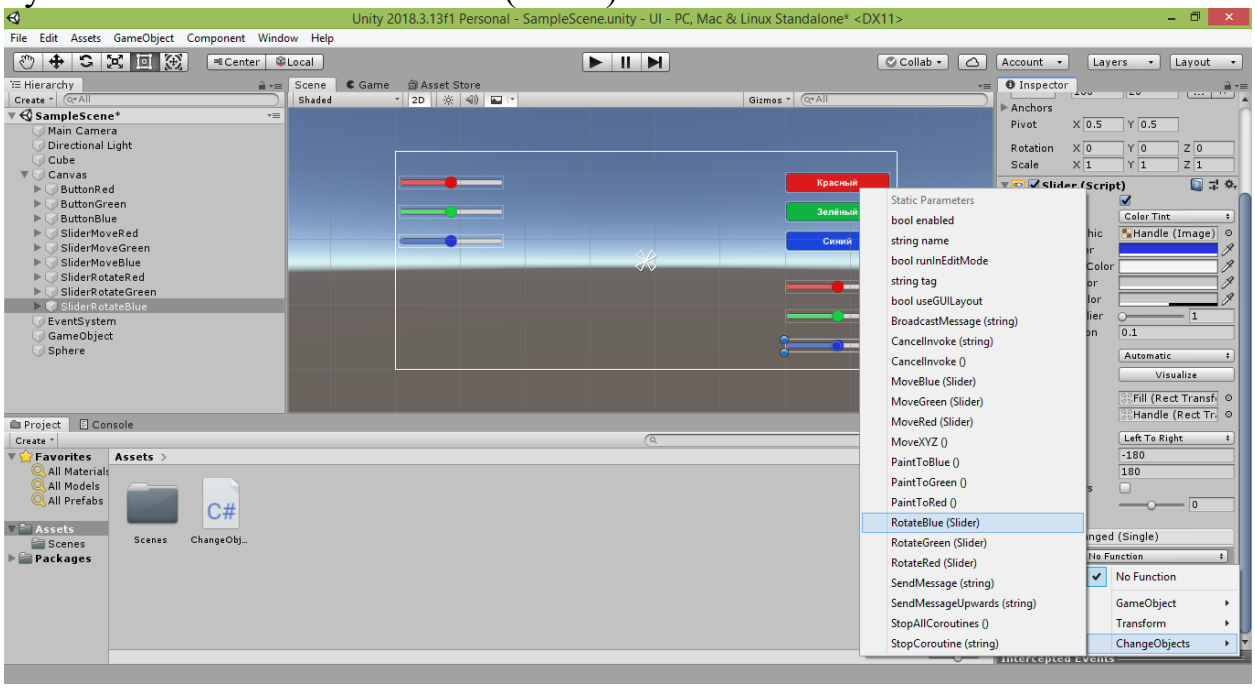
Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Выделите строку «SliderRotateBlue» в окне Hierarchy и справа в окне Inspector в группе «Slider (Script)» внизу пустого списка «On Value Changed (Single)» щёлкните на кнопке «+». В списке появится новый блок для задания обработчика события.

Перетяните строку «GameObject» из окна Hierarchy в поле блока, где написано «None (Object)», и отпустите. В поле появится новая надпись: «GameObject».

При этом станет доступен для изменения выпадающий список с надписью «No Function», расположенный рядом с полем.

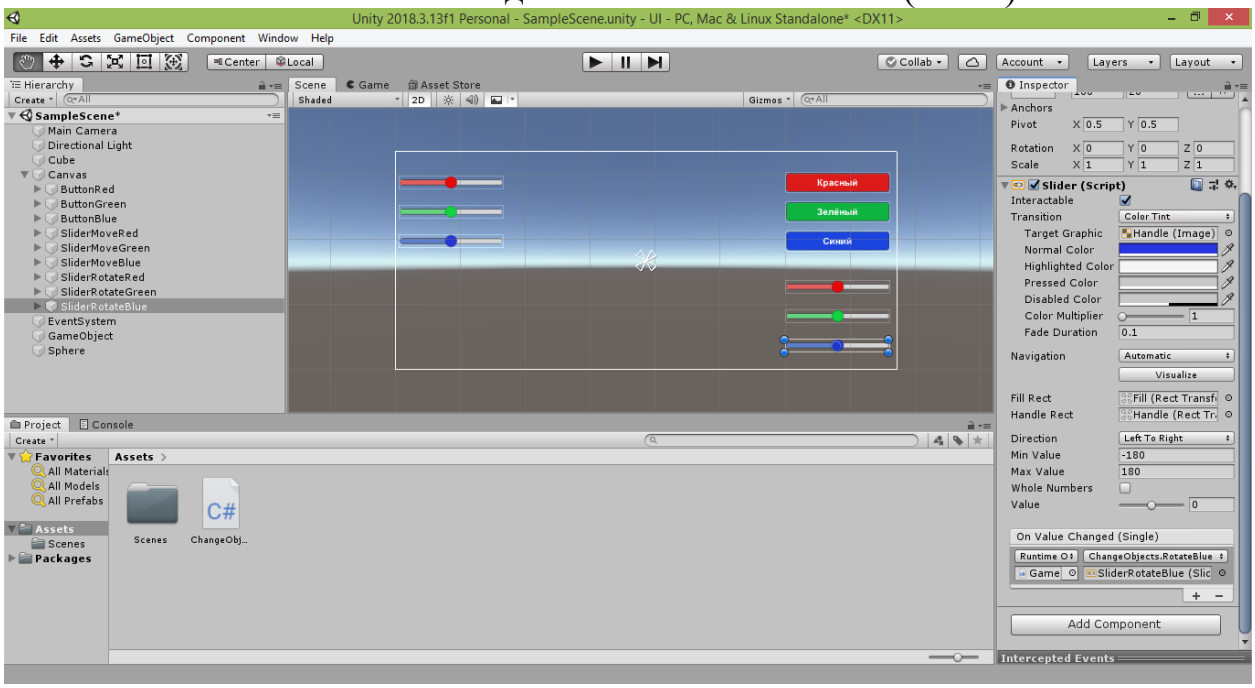
Раскройте выпадающий список и в подменю «ChangeObjects» выберите пункт с названием «RotateBlue (Slider)».



После этого под выпадающим списком появится новое поле с надписью «None (Slider)».

Перетяните строку «SliderRotateBlue» из окна Hierarchy в это поле и отпустите.

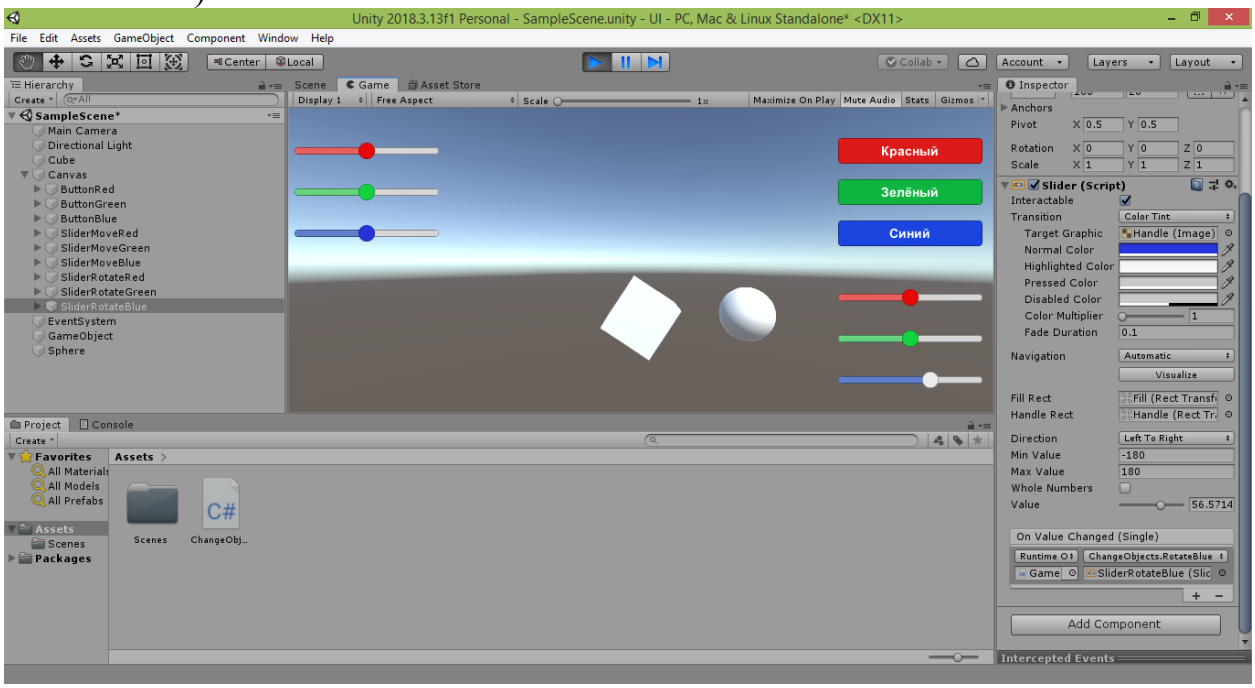
В поле появится новая надпись: «SliderRotateBlue (Slider)».



Запустите проект в игровом режиме, нажав кнопку «Play», и попробуйте переместить ручку синего ползунка.

В результате куб и сфера будут поворачиваться вокруг синей горизонтальной оси в соответствии с перемещением ручки синего ползунка.

Следует заметить, что вращение сферы видно не будет (чтобы его увидеть, нужно превратить сферу в эллипсоид, вытянув её вдоль красной или зелёной оси).



Снова нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Теперь мы можем поворачивать объекты сцены!

Но, как уже было сказано, недостаток данной реализации заключается в том, что каждый ползунок при движении его ручки сбрасывает изменения, выполненные ранее с помощью других ползунков.

Убедиться в этом можно, если изменить положение ручки одного из ползунков, а затем изменить положение ручки другого ползунка – куб сначала вернётся в исходное положение, а уже затем начнёт поворачиваться из него по другой оси.

Данную ситуацию мы исправим при помощи того же приёма, которым мы воспользовались при программировании перемещения объектов – будем запоминать положение ручек всех трёх ползунков в переменной, имеющей тип `Vector3` .

Именно для этого мы и объявили переменную `slidersRotate` .

В методе `Start` её составляющие получают равные нулю значения, поскольку при начале работы программы ручки ползунков центрированы.

Чтобы задействовать в коде переменную `slidersRotate` , обновим содержимое методов, отвечающих за поворот:

```
    public void RotateRed(Slider RedSlider)
    {
        slidersRotate.x = RedSlider.value;

        cube1.transform.rotation = Quaternion.Euler(startAnglesCube1 +
slidersRotate);
        sphere1.transform.rotation =
Quaternion.Euler(startAnglesSphere1 + slidersRotate);
    }


    public void RotateGreen(Slider GreenSlider)
    {
        slidersRotate.y = GreenSlider.value;

        cube1.transform.rotation = Quaternion.Euler(startAnglesCube1 +
slidersRotate);
        sphere1.transform.rotation =
Quaternion.Euler(startAnglesSphere1 + slidersRotate);
    }


    public void RotateBlue(Slider BlueSlider)
    {
        slidersRotate.z = BlueSlider.value;
```

```
        cube1.transform.rotation = Quaternion.Euler(startAnglesCube1 +
slidersRotate);
        sphere1.transform.rotation =
Quaternion.Euler(startAnglesSphere1 + slidersRotate);
    }
```

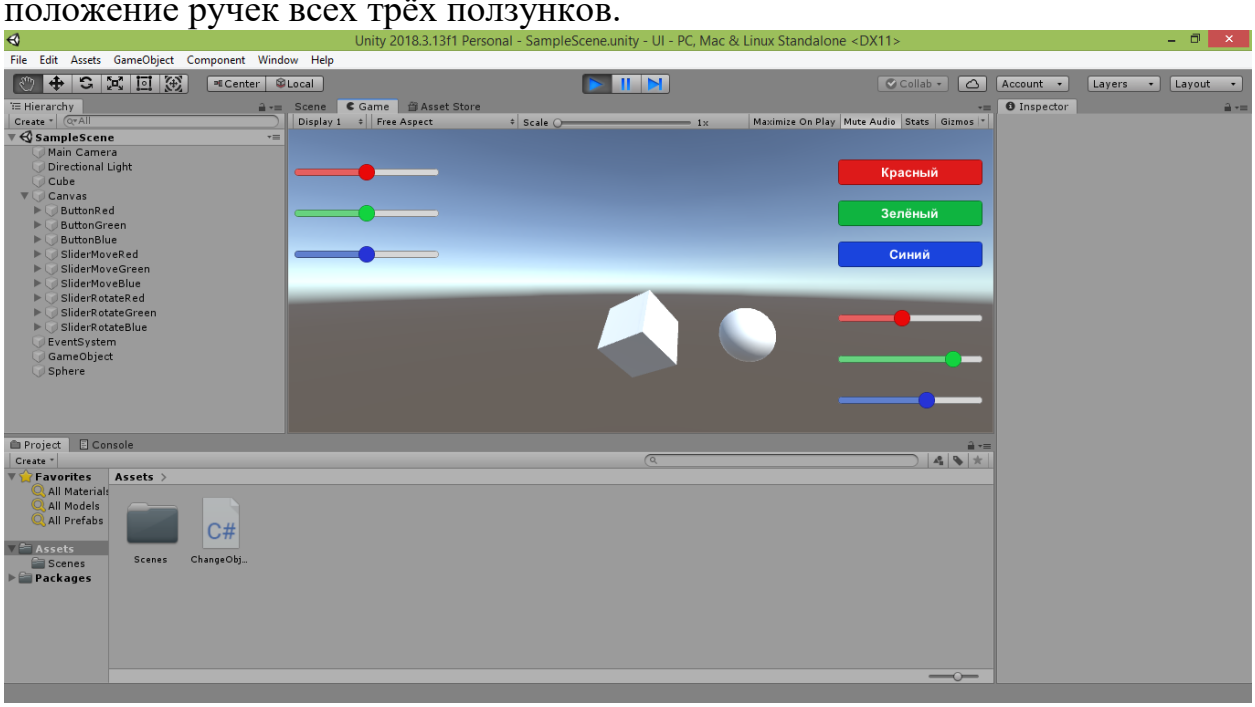
В компоненты `x` , `y` и `z` переменной `slidersRotate` мы записываем новые значения положения ручек красного, зелёного и синего ползунков поворота, соответственно. Запись (обновление) значения происходит для того ползунка, ручка которого пришла в движение.

Оставшиеся строки отвечают за расчёт новых углов поворота (`rotation`) объектов сцены, которое определяется путём сложения двух векторов: к компонентам-углам `x` , `y` и `z` вектора начального поворота объекта прибавляются соответствующие компоненты (положения ручек ползунков) вектора `slidersRotate` . Результат передаётся в команду `Euler` для записи итоговых кватернионов в свойство `rotation` объекта :

```
        cube1.transform.rotation = Quaternion.Euler(startAnglesCube1 +
slidersMove);
        sphere1.transform.rotation =
Quaternion.Euler(startAnglesSphere1 + slidersMove);
```

Проверим работу нового варианта кода, сохранив его нажатием комбинации клавиш CTRL+S и перейдя в окно Unity.

Запустите проект в игровом режиме, нажав кнопку «Play», и измените положение ручек всех трёх ползунков.



Теперь куб и сфера не возвращаются к исходным углам поворота, когда пользователь поворачивает их вокруг очередной оси координат.

Таким образом, ползунки больше не конкурируют друг с другом за повороты по своим осям координат, а действуют согласованно, позволяя задать объектам любую комбинацию углов поворота.

Нажмите кнопку «Play», чтобы остановить проигрывание сцены.

Обратите внимание, что последние две строки, которые мы прописали у всех трёх методов, одинаковы.

Это означает, что их можно оформить в виде шаблона, для чего потребуется создать новый метод:

```
    void RotateObjects()
    {
        cube1.transform.rotation = Quaternion.Euler(startAnglesCube1 +
slidersRotate);
        sphere1.transform.rotation =
Quaternion.Euler(startAnglesSphere1 + slidersRotate);
    }
```

Теперь в каждом из методов `RotateRed` , `RotateGreen` и `RotateBlue` мы можем убрать указанные две строки, заменив их одной:

```
    public void RotateRed(Slider RedSlider)
    {
        slidersRotate.x = RedSlider.value;
        RotateObjects();
    }


    public void RotateGreen(Slider GreenSlider)
    {
        slidersRotate.y = GreenSlider.value;
        RotateObjects();
    }


    public void RotateBlue(Slider BlueSlider)
    {
        slidersRotate.z = BlueSlider.value;
        RotateObjects();
    }
```

В результате код стал более компактным и читабельным.

Мы видим, что в каждом методе у переменной `slidersRotate` происходит обновление информации о положении ручки ползунка, вызвавшего метод.

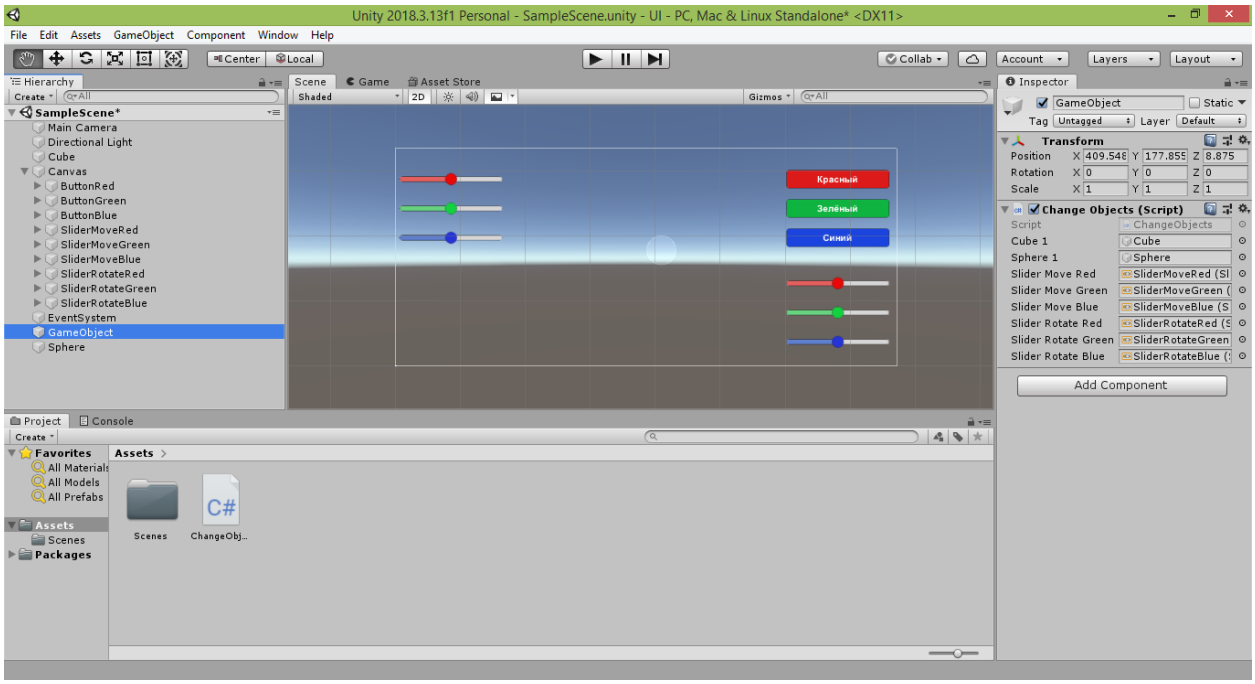
После этого происходит поворот объектов согласно новой комбинации углов (за это отвечает метод `RotateObjects`).

Сохраните код, нажав комбинацию клавиш CTRL+S.

Затем перейдите в окно Unity и убедитесь, что приложение по-прежнему работает корректно – ползунки согласованно управляют поворотом объектов.

Повторюсь, что ещё одним способом передавать в программу скрипта информацию о положении ручек ползунков, является объявление трёх открытых переменных типа `Slider` , которые затем следует связать с соответствующими ползунками через три новых поля, появившихся в свойствах пустого объекта «GameObject»:

```
    public GameObject cube1, sphere1;
    public Slider sliderRotateRed, sliderRotateGreen,
sliderRotateBlue;
```



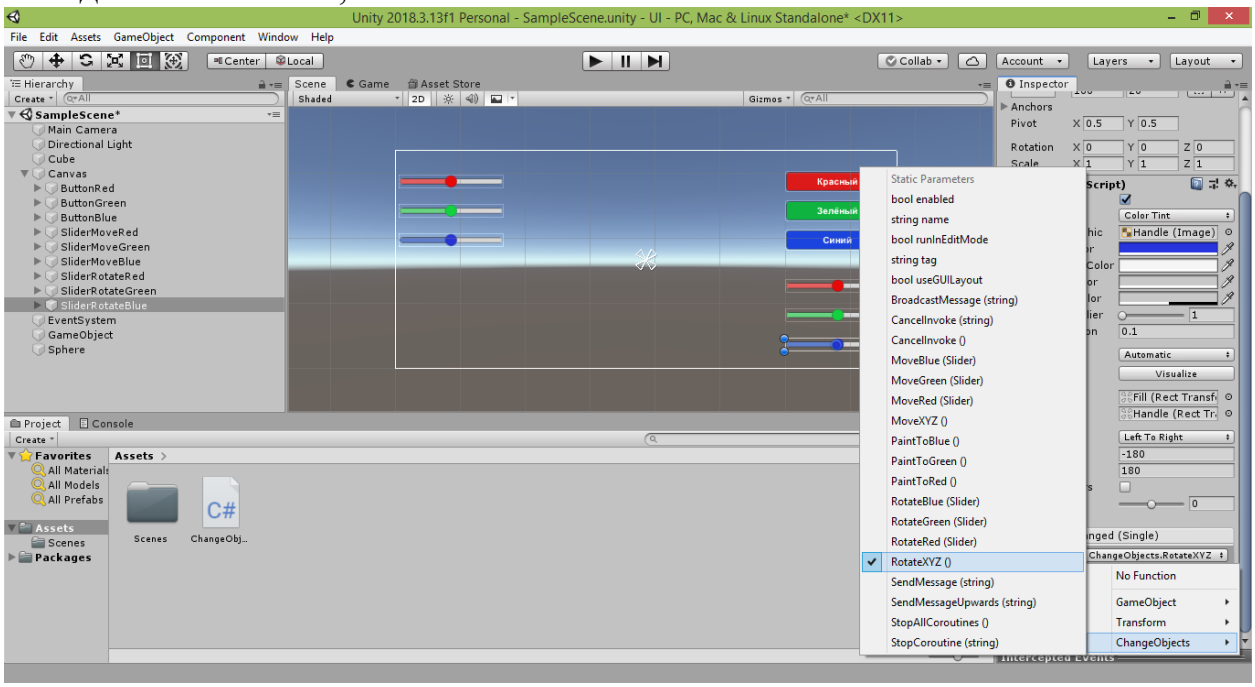
В этом случае нам уже не потребуются методы `RotateRed` , `RotateGreen` и `RotateBlue`.

Вместо них мы создадим единый универсальный метод `RotateXYZ` , который будет заносить в переменную `slidersRotate` информацию о текущем состоянии ручек ползунков, а затем вызывать метод `RotateObjects` для поворота объектов:

```
    public void RotateXYZ()
    {
        slidersRotate.x = sliderRotateRed.value;
        slidersRotate.y = sliderRotateGreen.value;
        slidersRotate.z = sliderRotateBlue.value;

        RotateObjects();
    }
```

Метод `RotateXYZ` был создан открытым (`public`), чтобы его можно было использовать в качестве обработчика события «OnValueChanged» у каждого из трёх ползунков.

Зададим ползункам метод `RotateXYZ` в качестве обработчика вместо методов `RotateRed` , `RotateGreen` и `RotateBlue` .



После этого можно запустить проект в игровом режиме и убедиться, что новый вариант кода управляет объектами сцены так же, как и раньше.

В завершение темы настройки ползунков рассмотрим способы их переориентации на полотне.

Предлагаю ползунки, отвечающие за управление поворотом объектов, расположить перпендикулярно осям, вокруг которых осуществляется поворот.

В результате красный ползунок должен занять вертикальное положение, поскольку красная ось изначально направлена вправо относительно наблюдателя.

Зелёный ползунок примет горизонтальное положение, поскольку зелёная ось направлена вверх.

Синий ползунок предлагаю разместить с наклоном под углом в 45 градусов, поскольку при создании проекта синяя ось направлена на наблюдателя (камера находится на её линии).

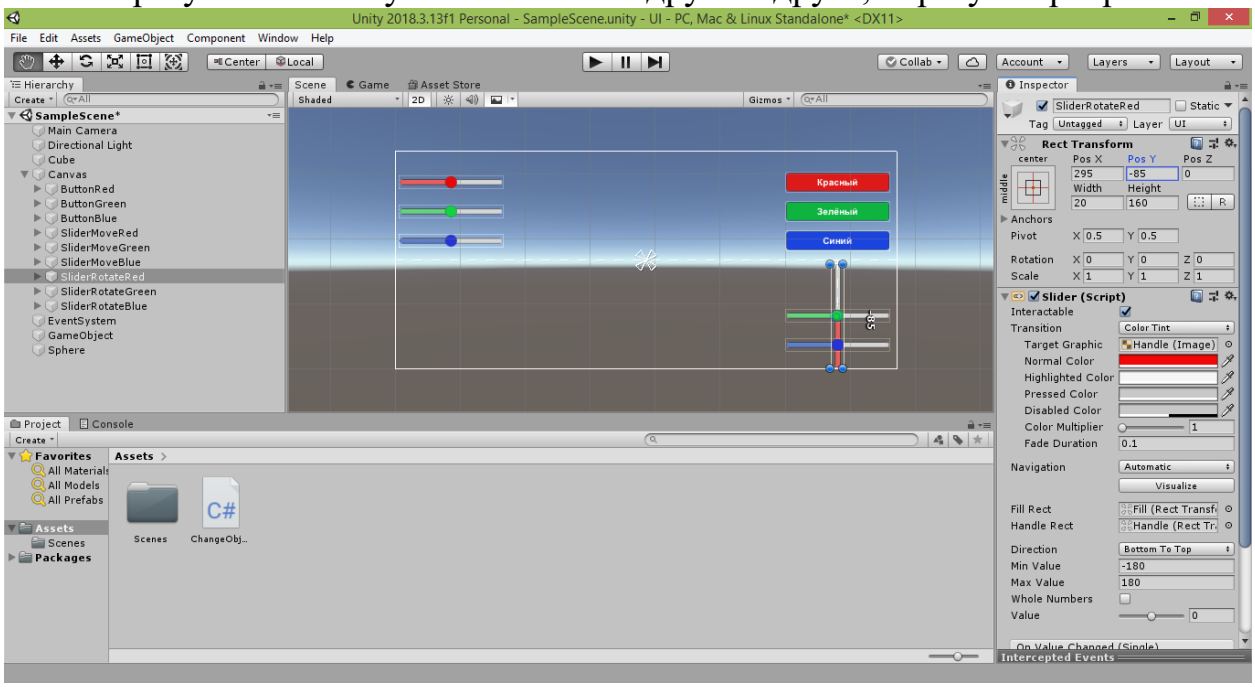
Начнём с красного ползунка. Выделите строку «SliderRotateRed» в окне Hierarchy и справа в окне Inspector в группе «Slider (Script)» выберите в выпадающем списке «Direction» (что переводится как «Направление») значение «Bottom To Top» (что переводится как «Снизу вверх»).

В результате красный ползунок примет вертикальное положение.

Теперь совместим центр красного ползунка с центром зелёного.

Для этого в свойстве «Pos Y» красного ползунка задайте значение -85.

В результате ползунки наложатся друг на друга, образуя перекрестие.



Зелёный ползунок в переориентации не нуждается, поскольку он изначально имеет требуемое расположение.

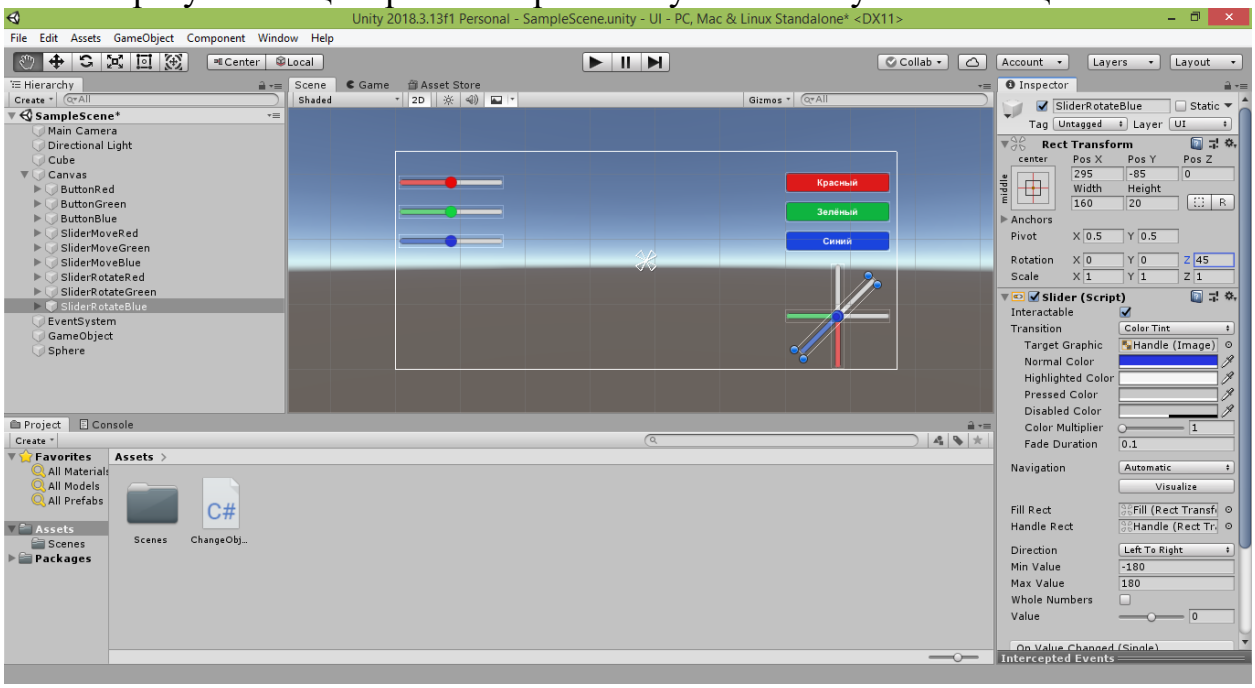
Для настройки синего ползунка выделите строку «SliderRotateBlue» в окне Hierarchy и справа в окне Inspector в группе «Slider (Script)» в свойстве «Rotation» задайте значение 45 в поле «Z».

В результате синий ползунок примет диагональное положение.

Теперь совместим центр синего ползунка с центром зелёного.

Для этого в свойстве «Pos Y» синего ползунка задайте значение -85.

В результате центры всех трёх ползунков окажутся совмещены.



После этого можно запустить проект в игровом режиме и протестировать новый вариант расположения элементов пользовательского интерфейса.

Теперь управление поворотом объектов стало интуитивно понятным, поскольку теперь оно имитирует расположение окружностей кватернионов (исключением является синий ползунок, который пришлось расположить с наклоном).

Одним из недостатков такой системы управления является то, что ручки ползунков могут накладываться друг на друга. Это осложняет пользователю выбор нужного ему ползунка, поскольку есть риск, что будет изменено положение другой ручки, перекрывающей целевую.

Другим, более существенным, недостатком является то, что при повороте объекта вокруг любой оси поворачивается вся связанная с ним (локальная) система координат.

Например, если вы повернёте куб на 90 градусов при помощи синего ползунка, то зелёная ось, направленная вверх, повернётся вместе с кубом и примет горизонтальное положение.

То же самое касается и остальных двух осей координат.

Поэтому отклонение от центра положения ручек двух или трёх ползунков в результате может привести к такому повороту локальной системы координат объекта, что заданные нами положения ползунков начнут вводить в заблуждение пользователя приложения.

Проще говоря, комбинация поворотов может поменять местами оси координат, но положение управляющих ими ползунков при этом всё равно останется прежним, что будет только дезориентировать пользователя.

Поэтому, если вас больше устраивает первоначальный вариант горизонтального расположения ползунков поворота, вы можете самостоятельно вернуться к нему, отменив сделанные изменения нажатием комбинации клавиш CTRL+Z или же просто выставив прежние значения изменённых свойств ползунков.

Задание для самостоятельной работы

1. Добавьте на сцену два дополнительных объекта: цилиндр и капсулу.

2. Расположите все объекты так, чтобы они не перекрывали друг друга при обзоре сцены с камеры.

3. Измените код скрипта (в том числе в методах «PaintToRed», «PaintToGreen» и «PaintToBlue») так, чтобы при нажатии кнопок цилиндр и капсула становились окрашенными в соответствующие цвета. Не забывайте, что после объявления открытых переменных следует перейти в Unity и связать их с соответствующими объектами сцены.

4. Измените код скрипта (в том числе в методах «Start» и «MoveObjects») так, чтобы изменение положения ручек горизонтальных ползунков приводило к перемещению цилиндра и капсулы по соответствующим осям координат.

5. Измените код скрипта (в том числе в методах «Start» и «RotateObjects») так, чтобы изменение положения ручек перекрещивающихся ползунков приводило к повороту цилиндра и капсулы вокруг соответствующих осей координат.

6. Добавьте на полотно «Canvas» новую группу, состоящую из красного, зелёного и синего ползунков (можете разместить их вертикально). Измените код скрипта так, чтобы изменение положения ручек данных ползунков приводило к масштабированию куба, сферы, цилиндра и капсулы вдоль соответствующих осей координат.

7. Чтобы облегчить пользователю поиск элементов управления, добавьте на полотно «Canvas» текстовые надписи «Покраска», «Перемещение», «Поворот», «Масштабирование», выбрав в меню Unity команду «GameObject → UI → Text». После настройки текста и внешнего вида надписей расположите каждую из них вблизи соответствующей группы элементов управления.

8. Скопируйте в отдельные файлы результаты вашей работы:

- программный код скрипта, обеспечивающего работу пользовательского интерфейса;
- скриншот сцены с объектами, у которых при помощи элементов управления были изменены цвет, местоположение, угол поворота и масштаб.

Результаты вашей работы рекомендуется показать преподавателю и обсудить с ним.

Рекомендуемая литература

1. Бонд Дж.Г. Unity и C#. Геймдев от идеи до реализации / Дж.Г. Бонд. – СПб.: Питер, 2019. – 928 с.
2. Гейг М. Разработка игр на Unity 2018 за 24 часа / М. Гейг. – М.: Эксмо, 2020. – 464 с.
3. Ламмерс К. Шейдеры и эффекты в Unity. Книга рецептов / К. Ламмерс. – М.: ДМК Пресс, 2014. – 274 с.
4. Линовес Дж. Виртуальная реальность в Unity / Дж. Линовес. – М.: ДМК Пресс, 2016. – 316 с.
5. Мэннинг Дж. Unity для разработчика. Мобильные мультиплатформенные игры / Дж. Мэннинг, П. Батфилд-Эддисон. – СПб.: Питер, 2018. – 304 с.
6. Паласиос Х. Unity 5.x. Программирование искусственного интеллекта в играх / Х. Паласиос. – М.: ДМК Пресс, 2017. – 272 с.
7. Торн А. Искусство создания сценариев в Unity / А. Торн. – М.: ДМК Пресс, 2016. – 360 с.
8. Торн А. Основы анимации в Unity / А. Торн. – М.: ДМК Пресс, 2016. – 176 с.
9. Ферроне Х. Изучаем C# через разработку игр на Unity / Х. Ферроне. – СПб.: Питер, 2022. – 400 с.
10. Хокинг Дж. Unity в действии. Мультиплатформенная разработка на C# / Дж. Хокинг. – СПб.: Питер, 2019. – 352 с.

Заключение

В настоящем учебно-методическом пособии были рассмотрены основы создания, настройки и программирования объектов трёхмерных сцен, а также элементов пользовательского интерфейса графических приложений, разрабатываемых с использованием системы Unity. Конечно, перечень тем, которые требуется изучить, чтобы решать прикладные задачи в данной области на профессиональном уровне, выходит далеко за рамки представленного пособия. Однако знакомство с изложенной информацией позволит учащимся получить те базовые знания, которые необходимы для дальнейшего самостоятельного освоения нового материала в сфере разработки графических приложений. Педагогам пособие поможет определить ключевые темы и подобрать материалы, с рассмотрения которых следует начать погружение своих учеников в данную отрасль информационных технологий.