Знакомство ​​с​ U​nity

**Unity​ ​как​ ​экосистема**

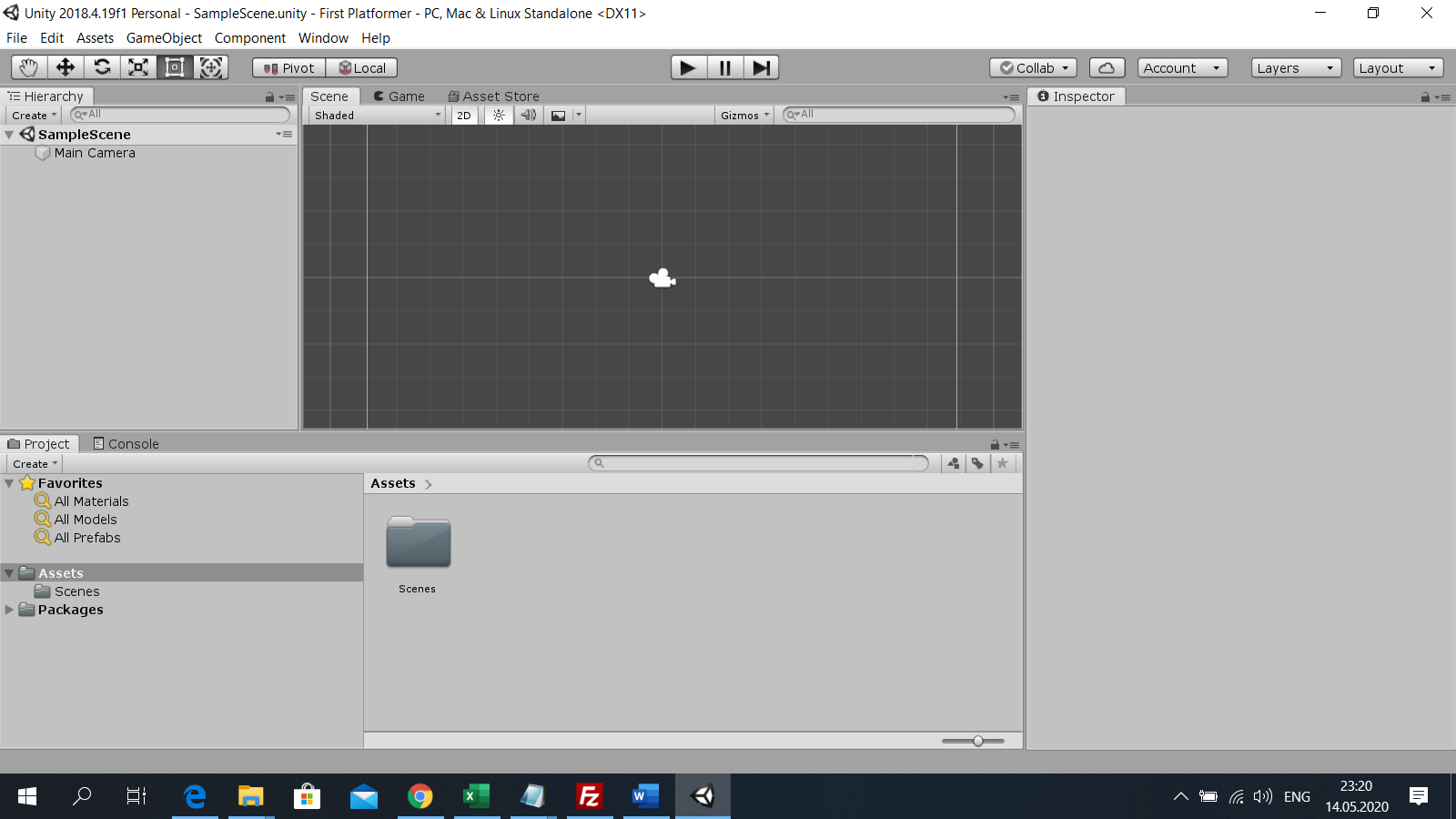
На сегодняшний день компания Unity Technologies формирует быстрое расширение мирового рынка игр и видит все возрастающее принятие движка Unity и его дополнительных игровых сервисов. Unity является доминирующим программным обеспечением для разработки игр в мире. С помощью Unity создано больше игр, чем с помощью любой другой игровой технологии. Все больше игроков играют в игры, сделанные на базе Unity, и всё больше разработчиков полагаются на инструменты и сервисы Unity​ ​для ​​развития​ с​воего​ ​бизнеса.

Unity – это не просто программа, это целая экосистема, включающая в себя несколько элементов: среду разработки (редактор), мультиплатформенность (т.е. возможность развертывания, публикации, выхода игры или приложения на множество платформ), различные сервисы Unity, Asset Store (магазин​ ​готовых ​​ресурсов​ ​и ​​плагинов),​ с​ообщество​ ​разработчиков​ и​​ ​людей,​ ​которые​ и​зучают​ ​Unity.

**Среда ​​разработки ​​(редактор​ ​Unity)**

В первую очередь, Unity предоставляет многофункциональную и в то же время, простую среду

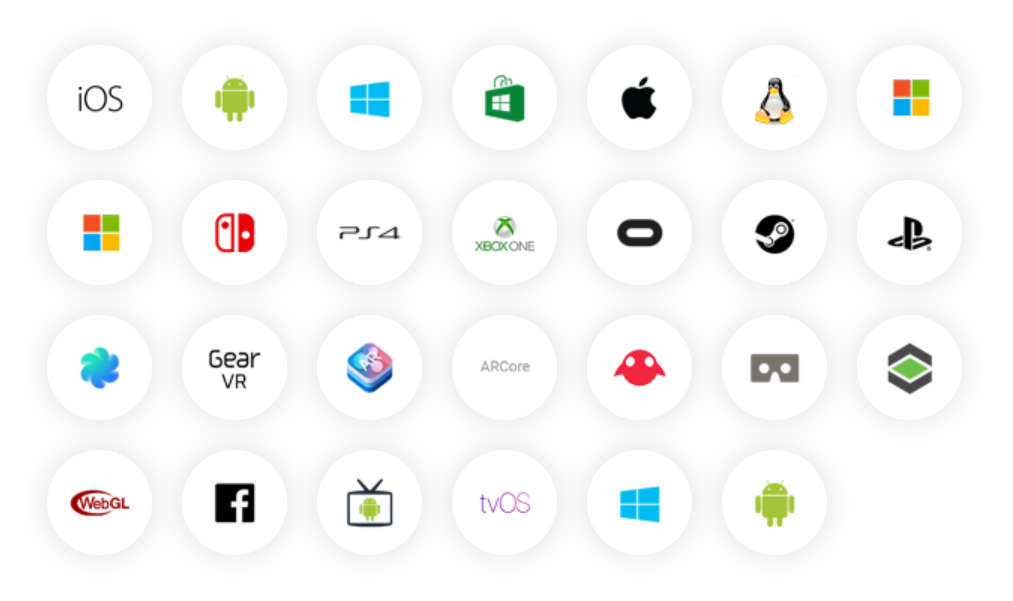
разработки ​​(редактор).



**Мультиплатформенность**

Unity​ ​известен​ ​в ​​индустрии ​​как ​​многоплатформенный​ и​гровой​ ​движок.​ Н​ а ​​данный ​​момент​ U​ nity

поддерживает​ ​24​ п​латформы,​ ​среди​ ​которых ​​можно​ ​выделить​ ​такие​ в​иды ​​платформ​ ​как: мобильные, устройства ​​виртуальной ​​и​ д​ополненной​ р​еальности, ​​настольные, ​​игровые​ ​консоли,​ в​еб-платформы, умное​ т​елевидение.



**Сервисы​ ​Unity**

Unity предоставляет ряд сервисов, помогающих разработчикам создавать игры, привлекать игроков,

удерживать​ и​х ​​в​ ​игре ​​и​ з​арабатывать. ​​На​ ​данный​ ​представлены​ с​ледующие​ ​сервисы:

●

Unity​ A​ ds​ ​–​ с​ервис​ м​ онетизации​ ​с ​​помощью​ р​екламы;

●

Unity ​​Analytics ​​–​ ​сервис ​​анализа​ в​ашей​ ​игры ​​после​ ​релиза;

●

Unity ​​Certification​ ​–​ п​рохождения ​​тестирования​ ​и ​​получения ​​сертификата​ u​nity;

●

Unity​ C​ loud ​​Build​ ​– ​​сервис​ ​автоматической​ ​сборки ​​и​ р​азвертывания​ и​гр;

●

Unity ​​Collaborate​ (​beta)​ –​​ ​сервис​ с​овместной​ ​работы​ ​над​ п​роектом;

●

Unity​ ​Everyplay​ –​​ ​возможность ​​игрокам​ з​аписывать​ г​еймплей ​​и​ д​елиться​ ​видео;

●

Unity ​​Multiplayer​ –​​ ​внедрение​ ​многопользовательского​ ​режима;

●

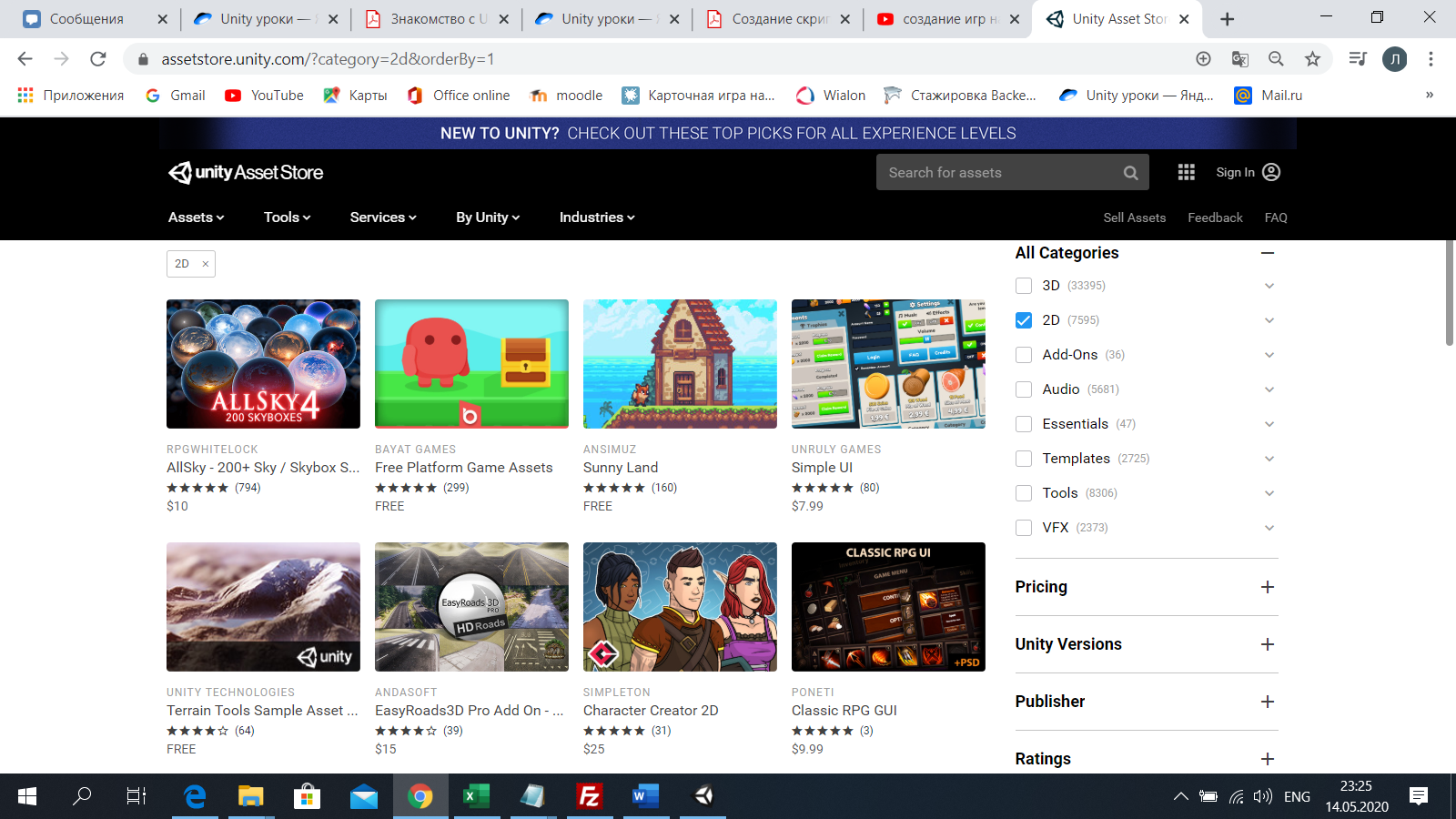
Unity ​​Performance​ ​Reporting​ ​– ​​сервис​ ​автоматического​ о​бнаружения​ о​шибок;

●

U​nity​ ​IAP​ ​- ​ ​​Встроенные​ ​покупки​ ​на​ ​разных​ п​ латформах.

**Asset​ S​tore**

Asset Store – магазин готового контента и инструментов для Unity. В магазине существуют следующие категории ассетов: 3D-модели, Анимации, Приложения, Аудио, Полностью готовые проекты, Расширения редактора, Системы частиц, Скрипты, Сервисы, Шейдеры, Текстуры и материалы, Официальные примеры Unity. Существуют как платные, так и бесплатные ассеты, а также широкой фильтр поиска. Для загрузки и использования ассетов необходимо установить Unity. Asset Store доступен через браузер и через окно в редакторе Unity. В магазине периодически проходят распродажи ,а в зависимости от лицензии могут быть специальные предложения.



Редактор​ ​Unity

# Создание​ ​нового​ ​проекта

При​ с​оздании​ ​нового​ ​проекта​ ​пользователь​ ​выбирает:

●

название проекта (это же является названием папки, в которой будет храниться содержимое

проекта);

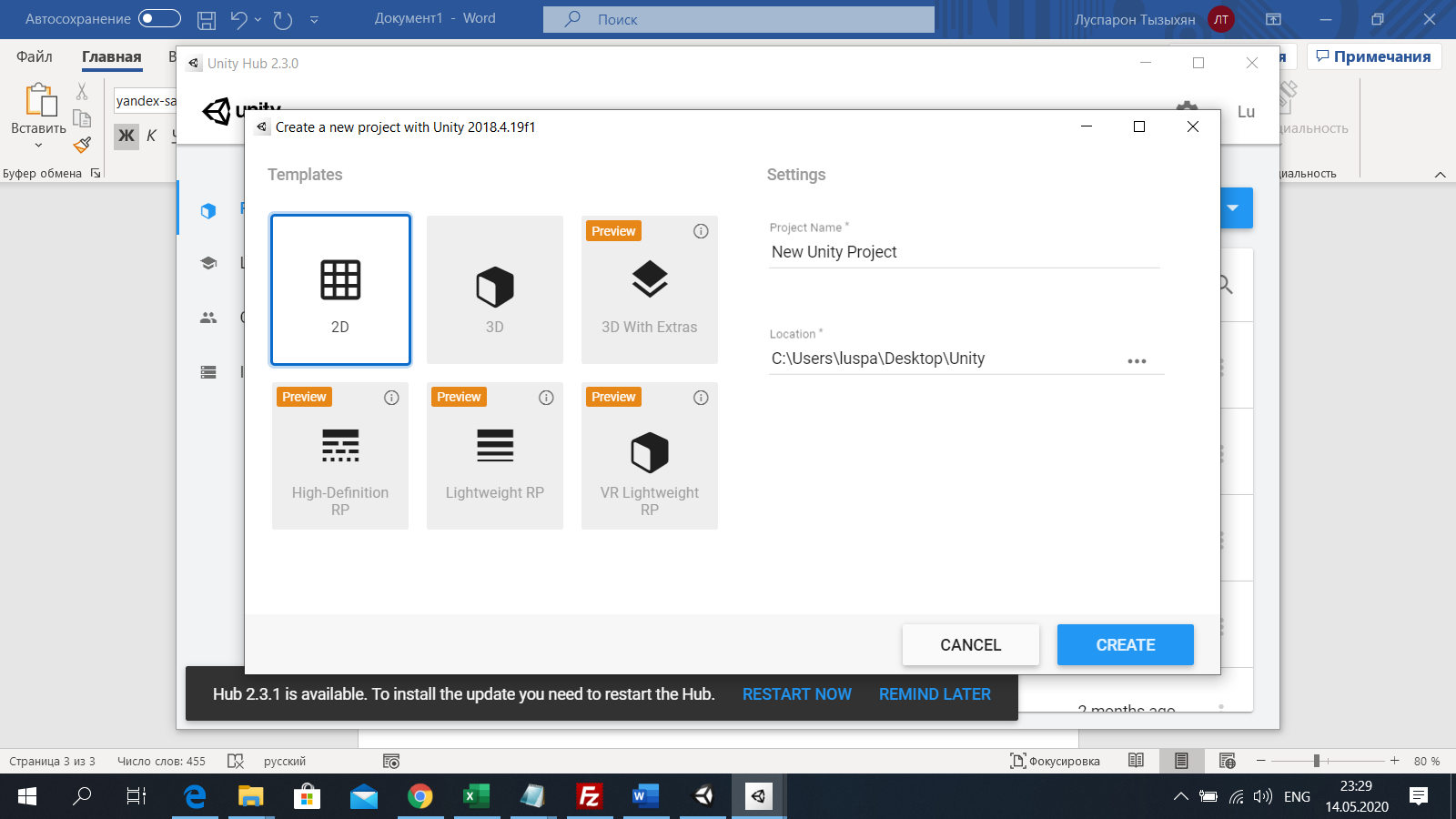
●

расположение​ ​проекта;

●

первоначальный режим редактора Unity (2D или 3D) в зависимости от того, какую игру

необходимо​ р​азработать;



# Интерфейс

Основные​ ​окна ​​редактора​ U​nity:

●

Scene – окно сцены, в котором непосредственно осуществляется сборка игровых уровней.

Сцену можно представить как один уровень в игре (за редким исключением), поэтому разработанная игра в итоге состоит из множества игровых сцен. Панель управления данного окна позволяет выбирать различные настройки просмотра сцены, влияющие на вид сцены в процессе разработки и не​ ​влияющие​ н​а ​​создаваемую​ и​гру.

Панель​ у​правления​ о​кна​ ​сцены.



Первый выпадающий список выбирает, какой режим отрисовки будет использоваться для

отображения сцены. Переключение между 2D и 3D-режимами. Включение освещения и звуков на

сцене. Второй выпадающий список включение и выключение эффектов: Skybox (текстура,

визуализируемая на заднем фоне), Fog (туман), Flares (блики от источников света), Animated Materials (анимированные материалы). Меню Gizmos – это графические иконки, которые помогают

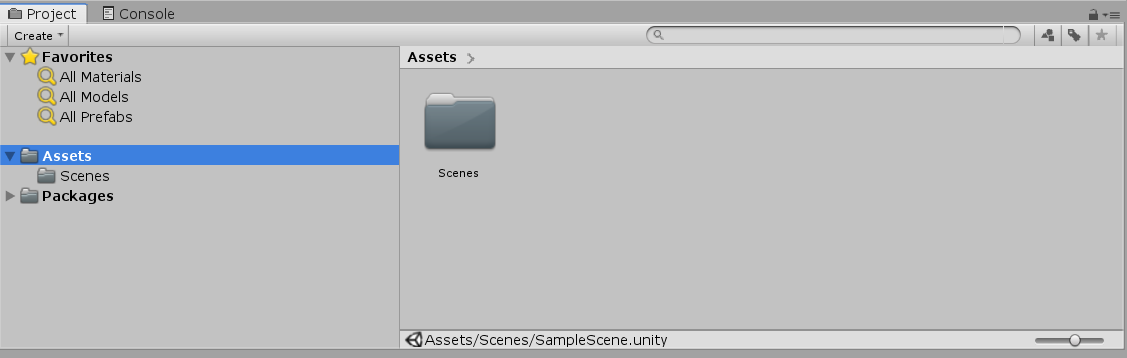
визуализировать и идентифицировать объекты на сцене (имеются встроенные иконки, но также

можно​ с​оздавать​ н​овые ​​для ​​своих​ о​бъектов).​ ​Последнее​ ​–​ э​то​ ​поле​ ​поиска ​​объектов​ ​на​ ​сцене.

●

Project – окно проекта, где отображаются все ассеты (файлы, которые физически находятся

на​ ​диске,​ ​их ​​можно​ ​найти​ ​в​ ​папке),​ ​находящиеся​ в​​ ​проекте.



Слева отображается структура проекта в виде иерархического списка, справа отображается

содержимое (также можно оставить лишь правую часть, выбрав One Column Layout в меню в правом верхнем углу). В разделе Favorites можно хранить результаты поиска ассетов в проекте. Окно проекта предоставляет мощные возможности поиска ассетов по имени, типу, метке. Для создания новых ассетов используется меню Create в левом верхнем углу окна или в контекстном меню. Кроме создания ассетов, в меню можно: показать проект в проводнике, импортировать новые ассеты (изображения, звуковые файлы и т.д.), импортировать/экспортировать пакеты (удобная возможность делиться готовыми файлами, в которых можно сохранить все взаимосвязи), поиск ссылок на ассет на сцене, выбор зависимых ассетов, обновление и реимпорт ассетов, открытие проекта в среде редактирования​ ​скриптов.

●

Inspector​ ​–​ ​окно​ ​инспектора​ ​отображает​ ​свойства,​ ​параметры​ ​выбранного​ ​объекта​ и​ли​ ​ассета.

В верхней части инспектора отображается имя объекта, слева от имени переключатель активен ли

объект, слева от переключателя для объекта может быть выбрана специальная иконка для

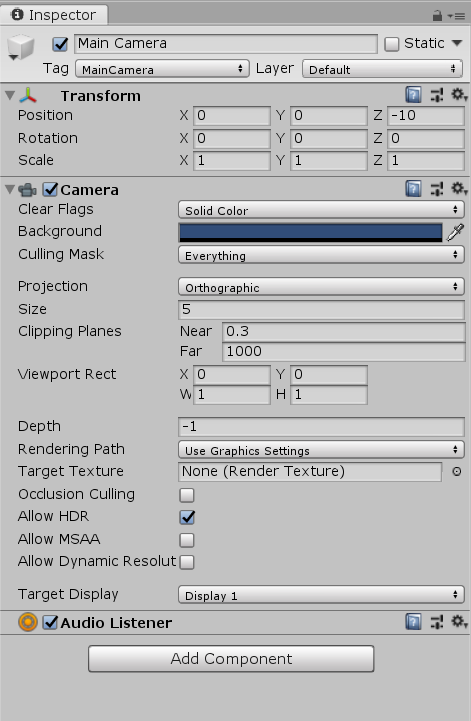
отображения его на сцене (аналог Gizmo), Tag – для объекта может быть выбран особый тэг для

удобства поиска и работы с объектом, Layer – слой, к которому относится объект, справа от имени

переключатель Static отвечает за включение всех статических настроек одновременно, далее

выпадающих список, в котором можно произвести более точную настройку (настройки Static

необходимы,​ ​например,​ ​для​ ​оптимизации).

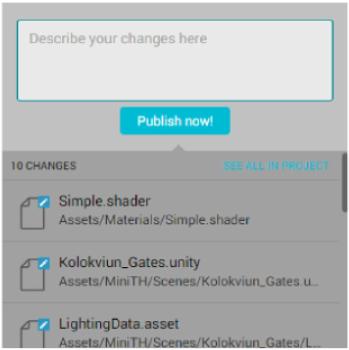


Инструменты в правой верхней части: Collab (Загрузить или отправить на сервер актуальную версию) Unity-сервисы,​ ​Аккаунт​ ​пользователя,​ ​Слои​ ​объектов,​ ​Расположение​ ​окон.



Окно Collab: В поле ввода, вы можете оставить комментарий ко внесённым вами изменениям в

проекте. ​​Ниже,​ ​вы​ м​ ожете​ ​увидеть ​​список​ в​несённых​ в​ами​ ​изменения ​​в​ ​проекте.



или​ ​увидеть ​​изменения ​​ваших​ ​коллег​ ​и​ з​агрузить​ а​ктуальную​ ​версию. Если вы нажмёте на комментарий коллеги, вам откроется окно Collab History, в котором вы сможете увидеть полный список изменений и старых версий проекта, таким образом, в случае поломки проекта,​ ​вы​ м​ожете​ о​ткатиться ​​до​ ​последней ​​рабочей​ в​ерсии.

# Игровые​ о​бъекты

Игровой объект (GameObject) – это самый важный объект при разработке игр в Unity. Любой объект в Unity является игровым объектом. По сути, GameObject – это некий контейнер, который может содержать в себе различные компоненты (о них следующая тема), и в зависимости от них является или персонажем, или землёй, домом, автомобилем, роботом, каким-либо предметом инвентаря, элементом​ и​нтерфейса, ​​или​ в​ообще​ ​объектом,​ ​который​ ​не​ и​меет​ в​изуального​ о​тображения. Пустой игровой объект – также является важным понятием, так как этот GameObject не имеет внутри себя компонентов, кроме компонента Transform. Пустые игровые объекты используются для формирования​ ​иерархии​ ​объектов​ ​и​ ​совершения ​​действий​ ​по​ ​отношению​ ​сразу​ к​​ ​группе​ ​объектов. Основные​ ​встроенные​ и​гровые​ ​объекты ​​подразделяются​ н​а​ ​следующие​ ​категории:

●

Create​ ​Empty​ ​–​ ​пустой ​​игровой ​​объект;

●

3D Object – категория объектов, к которой относятся различные 3D-примитивы (куб, капсула,

сфера ​​и​ ​другие);

●

2D​ ​Object​ ​– ​​в​ ​данной​ ​категории​ ​доступен​ ​объект​ ​Sprite;

●

Light​ ​–​ ​источники​ ​света;

●

Audio​ ​–​ ​объекты,​ с​вязанные​ ​со​ ​звуком;

●

UI ​​–​ ​категория​ ​объектов​ д​ля​ ​создания​ ​пользовательского​ ​интерфейса​ ​(UI);

●

Particle ​​System​ ​–​ ​система​ ​частиц;

●

Camera ​​–​ к​амера.

Доступ к ним осуществляется через меню GameObject на верхней панели редактора, через меню

Create и контекстное меню в окне иерархии. Также игровые объекты можно создать через скрипт.

Создаваемые​ ​игровые​ ​объект ​​содержат​ ​набор​ ​компонентов.

# Компоненты

Компоненты являются составляющими частями любого игрового объекта, они придают ему

определённый набор свойств и поведение. Обязательным компонентом, который включается по

умолчанию для любого объекта, является компонент Transform (для элементов интерфейса Rect

Transform).

Некоторые​ ​основные​ ​компоненты​ ​и ​​их ​​свойства​ ​(параметры):

●

Transform – отвечает за положение объекта на сцене (Position), поворот (Rotation), масштаб

(Scale);

●

Mesh Filter – передает mesh (набор вершин и многоугольников, определяющих форму

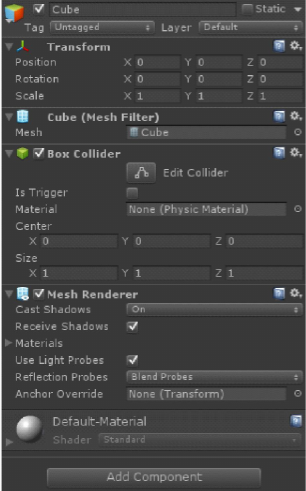
объекта)​ ​в​ ​Mesh​ ​Renderer​ ​для​ о​трисовки;

●

Box​ C​ ollider ​​–​ ​позволяет​ о​бъекту​ ​взаимодействовать​ ​с​ ​другими​ ​объектами;

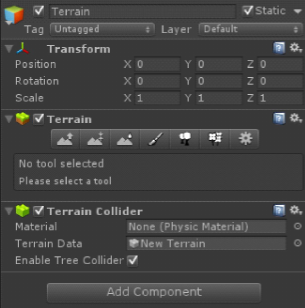
●

Mesh​ R​ enderer ​​–​ ​компонент​ о​трисовки​ ​mesh-а;



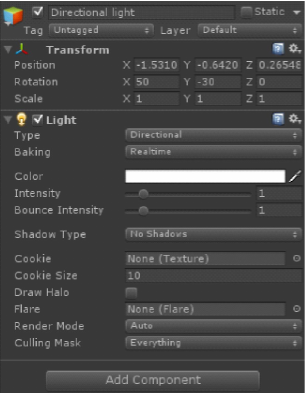
●

Terrain ​​–​ к​омпонент​ ​для ​​создания​ ​поверхности,​ р​ельефа;



●

Light​ ​– ​​компонент,​ о​пределяющих ​​тип ​​источника ​​света, ​​цвет,​ ​интенсивность​ ​и ​​т.д.;

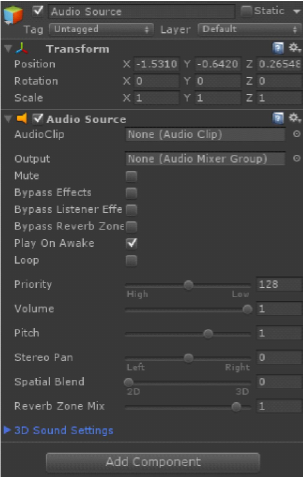


●

Audio Source – компонент для настройки и проигрывания 2D и 3D-звуков. Данный компонент

является «источником» звука, на сцене также необходим «слушатель», объект с компонентом Audio

Listener,​ о​бычно​ ​он​ ​по​ ​умолчанию​ ​находится​ ​на​ ​камере.



Компонент​ ​может​ ​быть​ ​добавлен​ ​к​ ​объекту:

●

Кнопка ​​Add​ ​Component​ в​​ ​нижней​ ​части​ ​окна​ ​инспектора;

●

Через​ ​меню​ ​Component​ ​в ​​верхней​ ​части​ ​редактора;

●

Перетащить ЛКМ из окна проекта на объект в окне иерархии или в окно инспектора

(определенные​ ​типы​ ​ассетов​ ​не​ ​могут​ ​быть​ ​добавлены ​​в​ ​качестве ​​компонентов​ т​аким​ о​бразом);

●

Через​ ​скрипт.

# Prefab​ (​префаб)

Prefab – особый тип ассетов, позволяющий хранить весь игровой объект со всеми компонентами и

значениями их свойств. Это некий шаблон для создания экземпляра хранимого объекта на сцене.

Любые изменения в префабе отражаются на его экземплярах, но при этом для каждого экземпляра в отдельности можно переопределить компоненты, не применяя изменения ко всем другим

экземплярам.

У нас есть некий объект на сцене, который повторяется множество раз, для этой цели удобно

использовать префабы и при необходимости применять изменения ко всем экземплярам префаба

одновременно. Например, в игре Марио: монетки, трубы, кирпичи – всё это могло бы быть

префабами.

Создать​ ​префаб​ ​можно​ ​следующим​ ​образом:

●

Assets​ ​–​ ​Create​ –​​ ​Prefab​ ​(верхнее​ ​меню​ ​редактора);

●

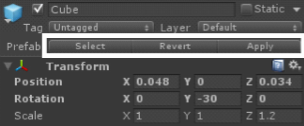
Create​ ​–​ ​Prefab​ ​(контекстное​ м​еню​ в​​ ​окне​ ​проекта);

●

И самый простой способ из текущего игрового объекта сделать префаб – это просто

перетащить​ ​его​ ​из​ ​окна​ ​иерархии​ ​в​ ​необходимую​ ​папку​ ​окна​ ​проекта. В окне иерархии префаб выделен синим цветом, а в окне инспектора у него появляются дополнительные кнопки: Select (для выбора этого префаба в окне проекта), Revert (отмена измененийу текущего экземпляра префаба до состояния исходного префаба, но это не касается позиции и поворота объекта), Apply

(применение изменений текущего экземпляра префаба ко всем экземплярам,​ ​в​ ​том​ ​числе​ исходному​ ​префабу).

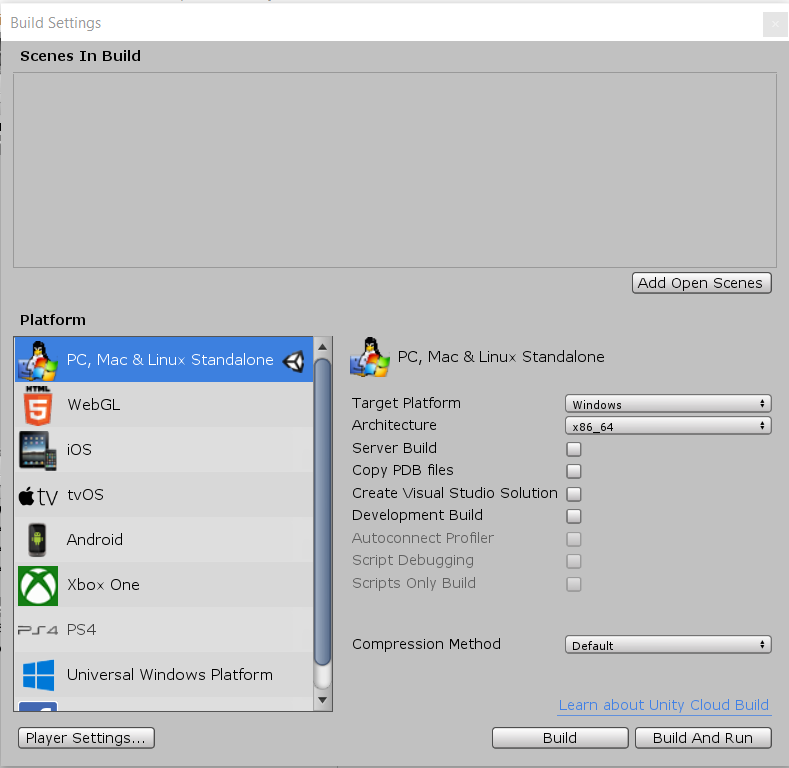


# Сборка​ ​проекта

Для того чтобы собрать проект под определённую платформу, необходимо на верхней панели

редактора выбрать File – Build Settings… Текущая выбранная платформа отмечена значком Unity. Для переключения необходимо выбрать другую платформу и нажать Switch Planform. Некоторые

платформы ​​требуют​ з​агрузки​ ​и​ ​установки​ ​дополнительных​ ​SDK​ и​ли​ м​одулей.

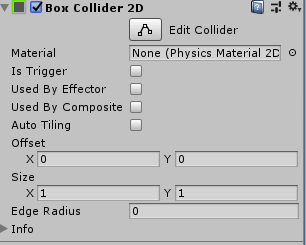


В дальнейшем, когда все настройки заданы, можно сразу нажимать Build And Run в меню File – Build And​ R​un.​ ​В​ т​аком​ ​случае,​ ​после​ ​того,​ ​как​ п​роект​ ​соберётся, ​​он ​​автоматически​ ​запустится.

# Знакомство с компонентами

## Компоненты

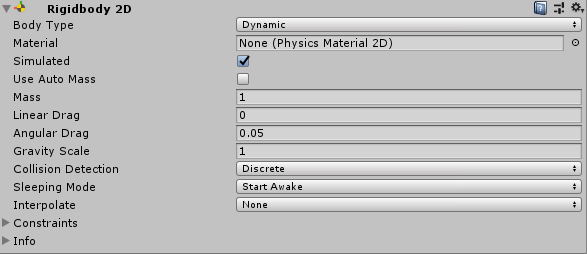
## BoxCollider2D – это прямоугольный коллайдер, используемый при работе с 2D физикой.



2D коллайдер/триггер обрабатывает события (сталкивание, пересечение и т.д.) независимо от того, где он располагается по координате Z. Все свойства коллайдера точно такие же, как и у BoxCollider, за исключением одного. Переключатель Used By Effector отвечает за то, используется ли данный коллайдер прикрепленным компонентом – Effector 2D (данные компонент бывает разного типа: точка, область, платформа и т.д.).

* Material – физический материал, который определяет свойства столкновений, такие как трение и отскок.
* Is Trigger – Является ли коллайдер триггером?
* Used by Effector – используется ли данный коллайдер прикрепленным компонентом – Effector 2D.
* Used by Composite – установите этот флажок, если вы хотите, чтобы этот коллайдер использовался прикрепленный Composite Collider 2D.
* Auto Tiling – установите этот флажок, если компонент Sprite Renderer установлен в режим Tile. Это позволит автоматически, обновлять форму коллайдера, которая будет корректироваться по форме спрайта.
* Offset – локальное смещение геометрии коллайдера.
* Size – ширина и высота прямоугольника в единицах локального пространства.
* Edge Radius – смягчает грани, делая коллайдер более круглым, по умолчанию значение равно 0.
* Info – общая информация о состоянии коллайдера.

Rigidbody2D – компонент помещает объект под контроль физического движка. В целом компонент похож на Rigidbody, но объект может двигаться только в плоскость XY, а вращаться только перпендикулярно этой плоскости.



Отличительные свойства:

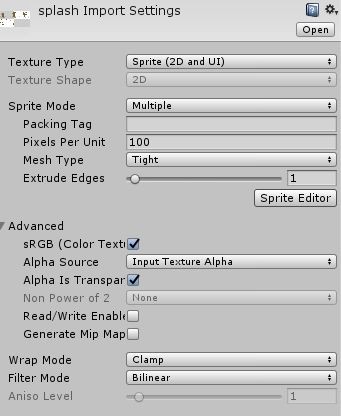
* Use Auto Mass – рассчитать массу автоматически, исходя из коллайдера;
* Gravity Scale – как гравитация воздействует на объект;
* Sleeping Mode – режим, когда объект выключается из физической симуляции (в целях оптимизации). Never Sleep – всегда активен (необходимо стараться избегать данного режима), Start Awake – активен при запуске (режим включен по умолчанию), Start Asleep

– неактивен при запуске, но может быть разбужен (столкновением или через скрипт).

# Спрайты

## Настройки спрайта (Import Settings)

## После импорта изображения в проект в 2D-режиме разработки его настройки по умолчанию выглядят следующим образом:

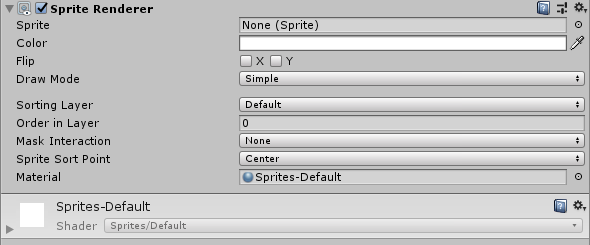
* В верхней части находится иконка – превью спрайта (также превью находится в нижней части инспектора), отображается название файла, кнопка Open, которая открывает изображение в программе просмотра изображений, установленной в системе по умолчанию.
* Texture Type – устанавливает базовые параметры в зависимости от предназначения текстуры (изображения).
* Sprite Mode – устанавливает, в каком режиме спрайт будет извлекаться из изображения, как один спрайт (Single) или как несколько раздельных спрайтов (Multiple). Второй режим может использоваться для импорта спрайтов как кадров анимации, а также как несколько спрайтов, относящихся к одному объекту. Третий режим Polygon позволяет настраивать форму, размер и pivot point для вашего спрайта.
* Packing Tag – имя атласа, в который данное изображение должно быть запаковано
* Pixels Per Unit – количество пикселей в изображении спрайта, которые соответствуют одному юниту (внутренняя единица измерения в Unity).
* Mesh Type - определяет тип меша, который создаётся для спрайта, по умолчанию это Tight.
  + Full Rect – создает квадрат по размеру спрайта,
  + Tight - создает сетку на основе альфа канала в пикселях, сгенерированная сетка повторяет форму спрайта.
* Extrude Edges – определяет область вокруг спрайта, которую нужно оставить в сгенерированной сетке.
* Pivot – точка спрайта, которая является центром локальной системы координат.

**Advanced**

* sRGB (Color Texture) – указывает, хранится ли данная текстура в гамма пространстве. Это значение должно быть у всех цветных текстур не имеющих HDR. Если вам нужно обратиться к точным значениям текстуры в шейдере (например, smoothness или metalness, то снимите флажок.)
* Alpha Source – указатель, как генерируется альфа-канал с текстуры.
  + None - не генерировать альфу, даже если текстура её имеет;
  + Input Texture Alpha - использовать альфа-канал текстуры;
  + From Grayscale - сгенерировать альфу из средних тонов.
* Alpha is Transparency – если у спрайта есть прозрачный альфа-канал, стоит включить Alpha is Transparency, чтобы увеличить радиус фильтрации, чтобы избежать артефактов на краях текстуры.
* Non Power of 2 – если размер текстуры не является степенью двойки, это будет определять поведение при масштабировании во время импорта.
  + None – размер текстуры останется неизменным.
  + To nearest – во время импорта текстура будет масштабирована в ближайший размер являющийся степенью двойки.
  + To larger – во время импорта текстура будет масштабирована в ближайший размер являющийся степенью двойки.
  + To smaller – во время импорта текстура будет масштабирована в ближайший размер являющийся степенью двойки.
* Wrap Mode – отвечает за поведение текстуры.
* Filter Mode – отвечает за то, как текстура фильтруется, если её растянуло при 3D-трансформациях:
  + Point (no filter) – вблизи текстура становится более пиксельной;
  + Bilinear – вблизи текстура становится более размытой;
  + Trilinear – как и Bilinear, но текстура также размывается и на разных мип уровнях.
* Aniso Level – увеличивает качество текстуры, при обзоре под большим углом. Хорошо подходит для пола и текстур земли. См. раздел Детали в конце страницы.
* Настройки разрешения и сжатия для всех платформ или для каждой по отдельности.
* Кнопки Revert и Apply для сохранения или отмены сделанных изменений.

## **Компонент Sprite Renderer**

Данный компонент позволяет отображать спрайты и используется как в 2D, так и в 3D-сценах.



Компонент имеет следующие свойства:

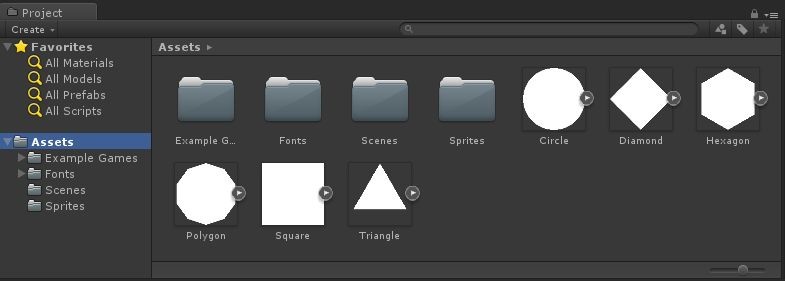
* Sprite – спрайт;
* Color – цвет;
* Flip – отображение спрайта по оси X или Y;
* Material – материал, стандартные шейдеры, которые используют для материала спрайта – Sprites/Default (не взаимодействует с освещением на сцене), Sprites/Diffuse (взаимодействует с освещением на сцене);
* Draw Mode – выберите параметр из списка Draw Mode, чтобы определить, как масштабируется спрай при изменении его параметров.
  + Simple – это значение по умолчанию. Изображение изменяется во всех направлениях, когда его размеры меняются.
  + Sliced – применяйте это, если хотите нарезать спрайт на 9 частей, чтобы эти части можно было растягивать. В этом режиме углы остаются одинаковыми, верхняя и нижняя часть тянутся в собственную высоту, а центр тянется в ширину и высоту.
  + Tiled – применяйте это если хотите нарезать спрайт на 9 частей, чтобы эти части можно было повторять. В этом режиме углы остаются одинаковыми, верхняя и нижняя часть тянутся в собственную высоту, а центр тянется в ширину и высоту.
* Sorting Layer – слой сортировки (новый слой можно создать в меню Edit – Project Settings – Layers);
* Order in Layer – порядок спрайта в слое сортировки (спрайты с большим значением отрисовываются поверх спрайтов с меньшим).

## **Sprite Creator**

С помощью этого инструмента можно создавать временные заменители спрайтов. Их можно использовать в проекте в процессе разработки, а затем, когда будет создана необходимая графика, заменить. Для создания таких спрайтов необходимо выбрать меню Assets – Create – Sprites, далее один из доступных типов спрайтов: Square, Triangle, Diamond, Hexagon, Circle, Polygon.

Через меню Assets – Create создаются не объекты, а ассеты, поэтому после создания спрайта в папке проекта появляется новый ассет, а на сцене ничего не изменяется. Далее необходимо добавить созданный спрайт на сцену.

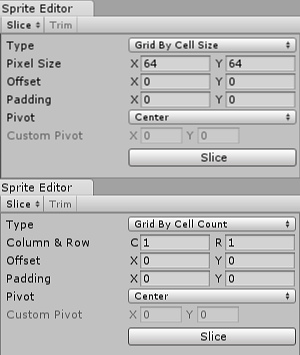
Спрайты в окне проекта:

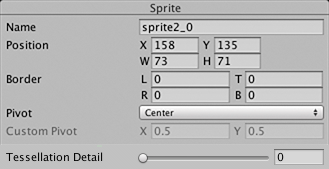


При создании спрайта Polygon открывается Sprite Editor, где можно изменить форму спрайта и некоторые его параметры.

## **Sprite Editor**

Чаще всего спрайт – это одно изображение, но иногда спрайт содержит несколько элементов, которые каким-либо образом связаны друг с другом и объединены в одно изображение. Для работы с такими спрайтами в Unity есть специальный редактор.





## 

* Name – название спрайта (редактируется только для каждого отдельного спрайта, когда включен режим Multiple);
* Position – позиция спрайта: по X, по Y, ширина, высота (редактируется только для каждого отдельного спрайта, когда включен режим Multiple). Может редактироваться прямо на изображении (отображается небольшими синими кругами);
* Border – границы спрайта: слева, справа, сверху, снизу (используется для изменения размера спрайтов в UI-элементах). Может редактироваться прямо на изображении (отображается небольшими зелёными квадратами);
* Pivot – точка спрайта, которая является центром локальной системы координат. Может быть установлен вручную прямо на изображении (отображается иконкой ), тогда автоматически включится режим Custom;
* Custom pivot – редактируется, если Pivot задан как Custom.

## **Sprite Packer**

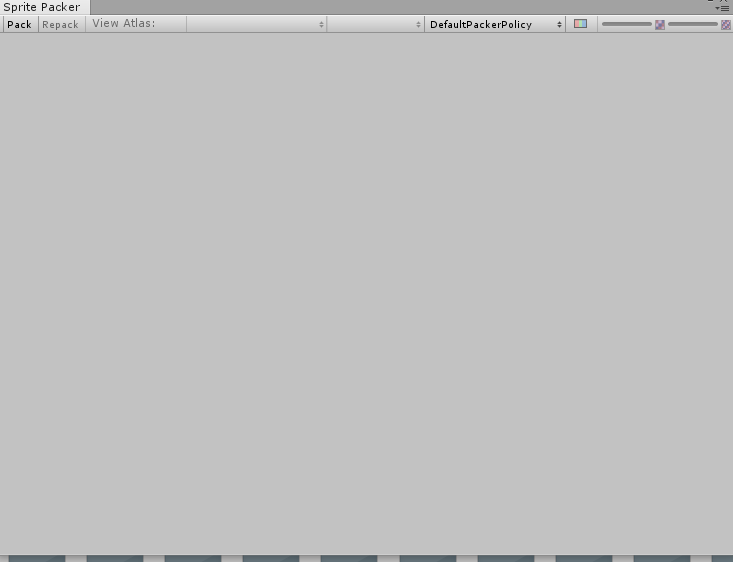
При работе со спрайтовой графикой очень важно при работе с разными персонажами работать с разными файлами текстур. Однако, значительное пространство спрайтовой текстуры будет уходить на пустые участки между графическими элементами, результатом чего станет бесполезный расход памяти при работе игры или приложения. Для достижения оптимальной производительности лучше всего будет упаковывать графические элементы из нескольких спрайтовых текстур как можно плотнее в одну текстуру, именуемую атласом. В Unity есть утилита Sprite Packer, предназначенная как раз для создания таких атласов из набора отдельных текстур.

По умолчанию упаковщик спрайтов отключён, но его работу можно настроить через настройки редактора Edit - Project Settings - Editor. Метод упаковывания спрайтов можно установить в состояние Enabled for Builds (упаковывание будет работать для сборок, но не в режиме Play) или в состояние Always Enabled (упаковывание будет работать и во время сборки и в режиме Play).

Указав для нескольких текстур (спрайтов) одинаковый Packing Tag, можно открыть Windows – Sprite Packer, нажать кнопку Pack и посмотреть расположение текстур в атласе.

Инструменты в верхней части окна упаковщика спрайтов:

* Pack – инициализирует операцию упаковывания;
* Repack – переупаковка атласа (обновление);
* View Atlas – атлас, который необходимо отобразить в окне (если при максимальном размере текстуры в неё не помещаются все спрайты, атлас может быть разбит на множество страниц);
* Page – позволят выбрать страницу атласа, которую необходимо отобразить в окне;
* Меню, следующее за номером страницы – выбор правил упаковывания использующиеся для текущего атласа;
* Далее инструменты, аналогичные редактору спрайта.



# **Основные методы класса MonoBehaviour**

**MonoBehaviour** – класс, на базе которого (наследуются) по умолчанию создаются все скрипты в Unity. Скрипты создаются через контекстное меню в окне проекта, через меню Component на верхней панели инструментов, при помощи кнопки Add Component в окне инспектора (когда выделен какой-либо объект). Скрипт работает только тогда, когда он является компонентом объекта на сцене, причём некоторые функции (методы) скрипта могут срабатывать даже тогда, когда сам скрипт выключен (неактивен).

Для редактирования скрипта после его создания необходимо дважды нажать на него ЛКМ в окне инспектора или в окне проекта. Скрипт откроется в той среде разработки, которая указана в настройках Editor – Preferences – External Tools – External Script Editor (обычно это Visual Studio или MonoDevelop). Там же можно сменить среду разработки.

Имя класса в скрипте должно совпадать с именем файла (это происходит автоматически, при создании файла). По умолчанию в скрипте создаются две функции (метода) – Start() и Update(), но существуют и другие функций, которые могут быть использованы как унаследованные от MonoBehaviour. Также можно создавать свои функции.

По умолчанию скрипт выглядит следующим образом:

using UnityEngine;

using System.Collections;

public class MyFirstScript : MonoBehaviour {

// Use this for initialization void Start () {

}

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

}

}

## Awake()

Функция, которая вызывается один раз при запуске игры до функции Start(). Вызывается в случайном порядке среди объектов на сцене, для которых она реализована. Функция срабатывает даже тогда, когда скрипт неактивен.

void Awake() {

}

## Start()

Функция, которая также вызывается один раз, но в тот момент, когда скрипт становится активным. Вызывается перед функций FixedUpdate(), Update() и т.д.

void Start () {

}

## FixedUpdate()

Функция, которая выполняется каждый фиксированный отрезок времени. Можно настроить в Edit – Project Settings – Time – Fixed Timestep. Используется для обработки физики (приложения силы, импульса к объекту и т.д.)

void FixedUpdate() {

}

Update()

Функция, которая выполняется каждый кадр. Используется для обработки игровой логики.

void Update() {

}

deltaTime – время (в секундах), прошедшее с последнего кадра. Это значением позволяет сделать игру независимой от скорости смены кадров.

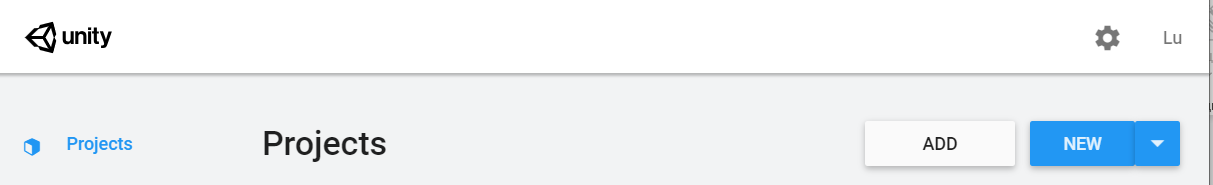
## LateUpdate()

Функция, которая выполняется каждый кадр. Вызывается после того, как все Update() функции будут вызваны. Используется для того, чтобы упорядочить выполнения действий в скрипте. Например, если камера следует за объектом, движение которого реализовано в Update() функции.

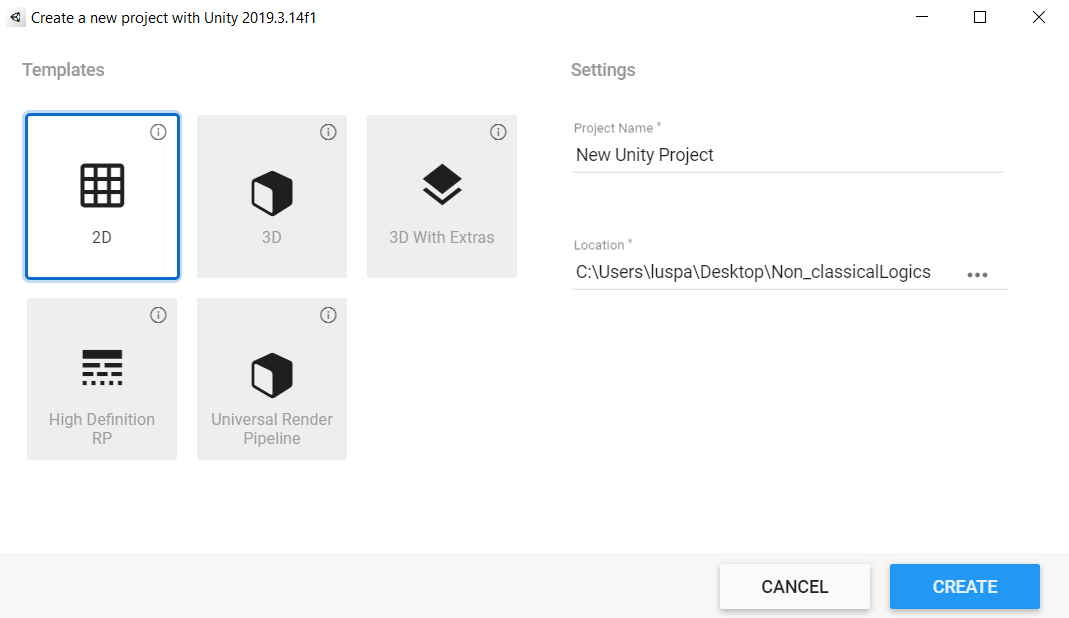
**­Создание 2D платформера с помощью Unity.**

**Создание проекта :**

В Unity Hub создаем новый проект , для этого необходимо нажать кнопку NEW во вкладке Projects.



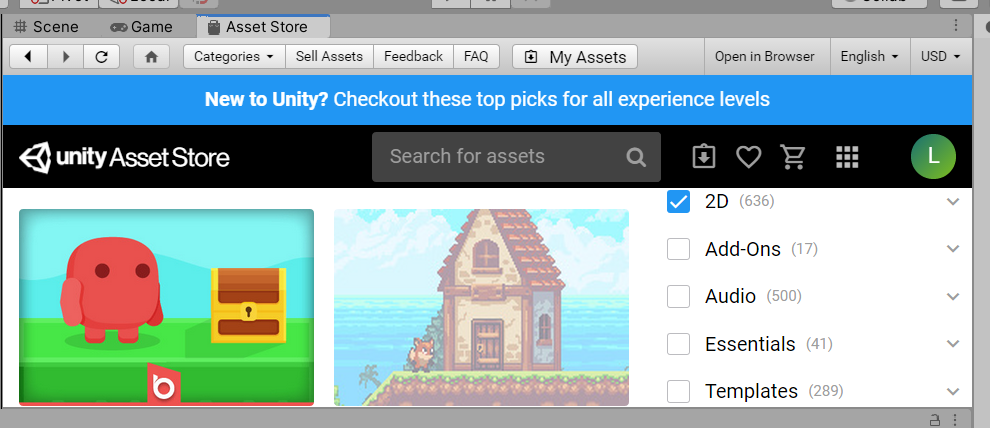
После этого открывается окно с выбором шаблона проекта . Необходимо выбрать шаблон 2D , дать имя проекту и указать его расположение. Затем нажать кнопку Create.



Готово! Проект создан.

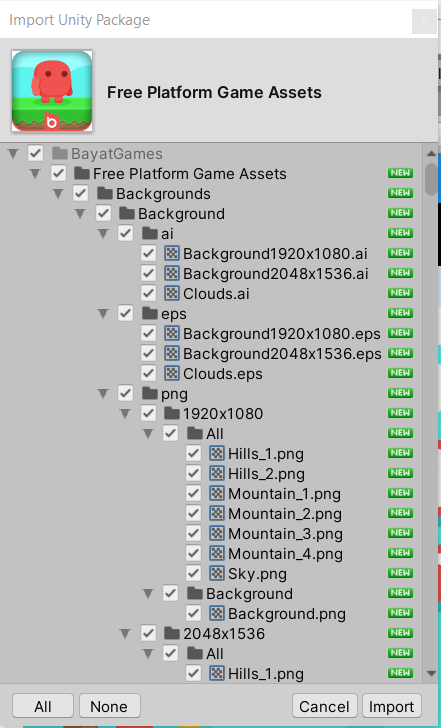
**Assets и настройка проекта** :

Далее необходимо скачать Asset-ы для нашего проекта. Переходим во вкладку Asset Store в верхней части экрана . Ставим фильтры 2D и Free Assets и выбираем понравившийся пакет.

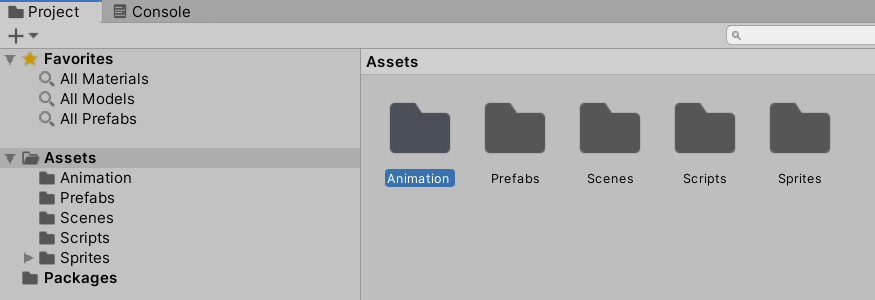


# В этом курсе используется пакет Free Platform Game Assets от разработчика BAYAT Games.

Открываем пакет и нажимаем download , затем Import для добавления ассетов к проекту. Далее вы увидите окно , где можно выбрать отдельные элементы для загрузки. По умолчанию выделяются все элементы ассета. Нажимаем кнопку Import. После завершения установки вы увидите новую вкладку в окне навигатора. Рекомендуется переименовать ее в Sprites.

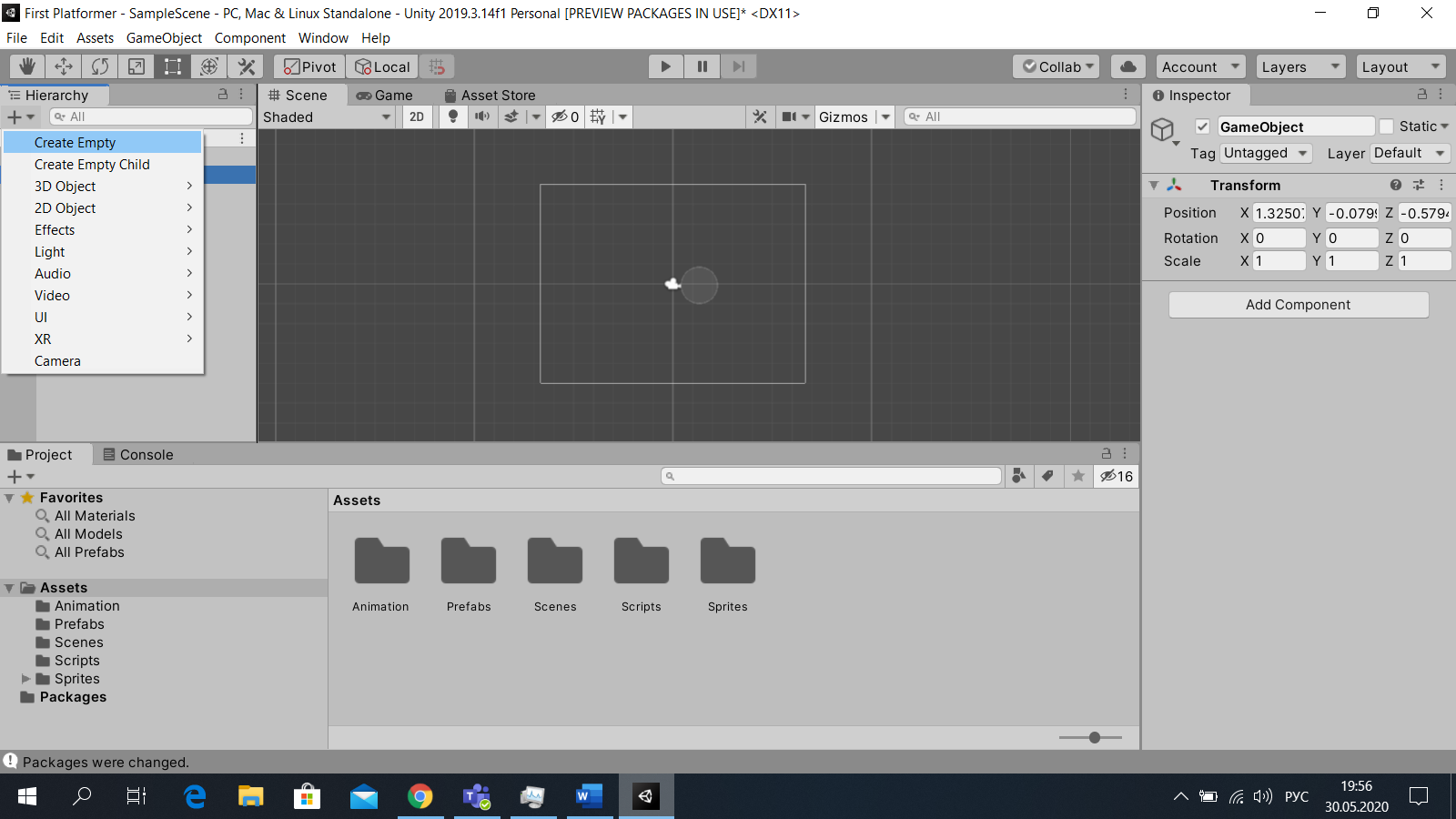


Далее необходимо создать структуру проекта. Для этого в папке Assets создаем подпапки Scripts , Prefabs ,Animation. Назначение этих подпапок понятно из их названий.



**Platforms**

Перейдем к созданию объектов. В окне иерархии нажимаем «+» или Create , в зависимости от версии Unity. Далее нажимаем Create empty для создание пустого Объекта.

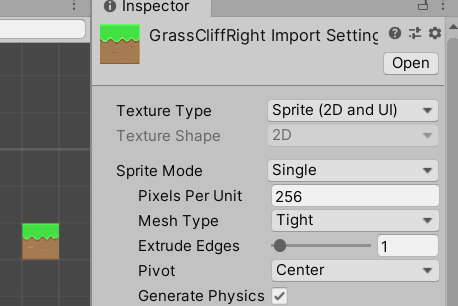


Следует переименовать объект в соответствии с его смысловой нагрузкой , например Platforms(так как , в нашем случае , это объект для платформ).

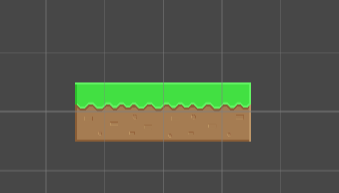
В объектном навигаторе , в папке Assets/Sprites находим необходимые тайлы (картинки). В нашем случае это файлы GrassCliffLeft , GrassCliffMiddle , GrassCliffRight .



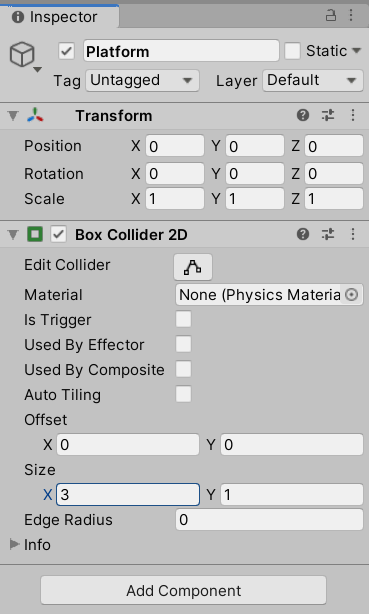
При перемещении тайла в окно сцены вы заметите что он слишком большой/маленький , поэтому необходимо настроить его размер . при нажатии на тайл в окне навигатора , справа вы увидете окно Inspector для выбранного тайла . Рассмотрим модификатор Pixel Per Unit. Нам необходимо сделать так , чтобы наш тайл совпадал с размером одного юнита . Для этого в поле Pixel Per Unit поставьте значение , соответствующее размеру одной из сторон выбранного тайла . Например в случае , если размер тайла 128Х128 , следует выставить значение 128 , и тд. Вышеупомянутые действия следует выполнить для всех тайлов.



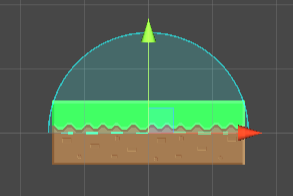
Итак , перейдем к созданию платформы. В окне иерархии создаем новый объект Platform внутри объекта Platforms. Добавляем в него наши тайлы . Далее нам необходимо их выровнять и совместить. Для этого сбросим координаты центрального тайла , нажав правой кнопкой мыши на элемент Transformв инспекторе , затем кнопку Reset. Выполнив те же действия для остальных тайлов , вы можете легко настроить их расположении по оси X (-1 для левого тайла , 1 для правого). Получили :



Следующая задача – сделать из нашего объекта префаб. Для этого добавим коллайдер на корневой файл(Platform) : выделите объект в иерархии , в окне инспектора нажмите Add Component и добавьте компонент Box Collider 2D , затем настройте размер коллайдера в соответствии с размером тайлов.

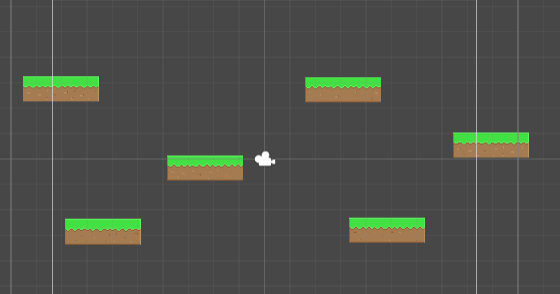


Далее добавляем компонент Platform Effector 2D и в компоненте Box Collider 2D ставим галочку напротив Used By Effector. Данный компонент определяет настройки платформы (снизу проницаема , сверху нет).

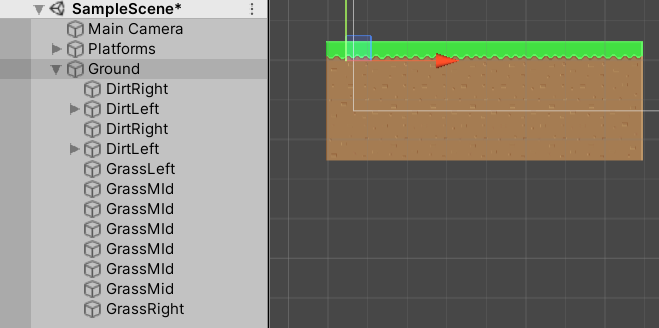


Далее переходим во вкладку навигатора Assets/Prefabs и перетаскиваем туда наш объект Platform из окна иерархии. Готово , префаб создан.

Теперь мы можем создать небольшой уровень. Перетащите на сцену какое то количество платформ (сделанных на предыдущем шаге префабов) и расположите их в хаотичном порядке. Следует следить за тем , что новые объекты находятся в объекте Platforms в окне иерархии.



Теперь создадим пол/землю. Создаем новый объект Ground и по аналогии с Platforms добавляем в него тайлы и выравниваем(в нашем случае тайлы GrassLeft , GrassMid, GrassRight , Dirt).



Затем добавляем коллайдер и настраиваем его размер.

**Character**

Перейдем к созданию персонажа. В папке Assets/Sprites найдем необходимые файлы. В нашем случае персонаж представлен в виде набора анимаций. Выберем анимацию Run и перенесем ее на сцену , но перед этим необходимо убедиться что спрайты разрезаны. Если нет , то воспользуемся инструментом Sprite Editor в инспекторе(убедитесь что Sprite Mode - Multiple). В открывшемся окне Sprite Editor нажимаем Slice – Slice , после чего перетаскиваем объект на сцену , сохраняем анимацию в Assets/Animation/Character и настраиваем размер с помощью Pixels Per Unit. В инспекторе персонажа должны появиться компоненты Sprite Renderer и Animator. Теперь при запуске игры вы можете увидеть как перcонаж “ходит на месте”.

Далее следует добавить различные компоненты к персонажу. Назначим Tag = Player , добавим коллайдер (в нашем случае Circle Collider 2D) и Rigidbody 2D.

Чтобы избежать поворотов персонажа по оси Z , ставим галочку Rigidbody 2D – Constrainst-Freeze Rotation.

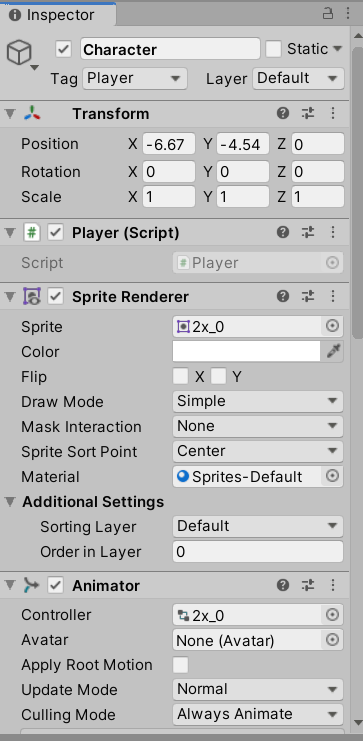
-----------

После того как персонаж попал на сцену , может возникнуть проблема со слоями (при перемещении персонажа он находиться *под* другими объектами).

**Скрипты**

Получение доступа

Перейдем к созданию скриптов для нашего персонажа. Во вкладке Assets/Scripts создаем C# скрипт с названием Player . Затем перетаскиваем скрипт в окно инспектора нашего объекта .



Далее открываем сам скрипт(по умолчанию должен открыться в Visual Studio) и видим шаблон кода.

Перед началом работы со скриптом , важно узнать основные способы получения доступа к объекту :

* ключевое слово gameobject (если скрипт является компонентом необходимого игрового объекта);

void Start () {

gameObject.SetActive(false); // Данный объект становится неактивным

}

* ссылка на другой игровой объект;

public GameObject anotherObject; void Start () {

anotherObject.SetActive(false); // Переданный в скрипт объект становится неактивным

}

* через transform (как к объекту родителю или ребенку);

становится не активным (при этом вся иерархия объектов также становится неактивной)

}

void Start () {

transform.GetChild(0).gameObject.SetActive(false); // Нулевой ребенок данного объекта становится неактивным

}

// Родительский объект данного объекта

void Start () { transform.parent.gameObject.SetActive(false);

* поиск игрового объекта по имени (операция, требующая больших ресурсов, лучше стараться её избегать);

void Start () {

GameObject.Find("GameObject").SetActive(false); // Первый найденный объект с именем "GameObject" становится неактивным

}

* поиск игрового объекта по тегу;

void Start () {

GameObject.FindGameObjectWithTag("Player").SetActive(false); // Первый найденный объект с тэгом "Player" становится неактивным, также существует метод поиска всех объектов с заданным тегом "FindGameObjectsWithTag"

}

После получения доступа к объекту, можно получить доступ к компоненту:

* через метод GetComponent<>

}

// Уничтожение компонента

void Start () {

Destroy(gameObject.GetComponent<AudioSource>()); AudioSource для данного объекта

void Start () {

gameObject.GetComponent<BoxCollider>().enabled = false; // Выключение коллайдера для данного объекта

}

Передвижение

При разработке игр часто приходится использовать такой математический объект, как вектор – направленный отрезок прямой. В Unity векторы используются для позиции объекта, направления движения и различных расчетов.

Так как мы фактически работаем в трехмерном пространстве, наши gameobject’ы имеют transform.position, который является структурой vector3, давайте рассмотрим варианты изменения позиции объекта (не связанные с физикой) :

* иерархия объектов. При перемещении родительского объекта все объекты, для которых он является родителем, будут изменять свою позицию соответственно.
* изменяем позицию через компонент Transform (простое, но далеко не оптимальное решение, так как операция new выполняется каждый кадр):

int speed = 1;

void Update ()

{

transform.position += new Vector3(Time.deltaTime\*speed, 0, 0);

}

* метод Translate. В параметрах указывается вектор направления движения или координаты по отдельности, а последним параметром (необязательный) - относительного чего происходит движение (локальная система координат, глобальная система координат, другой объект – например, камера). По умолчанию задана локальная система координат.

void Update ()

{

// Движение объекта вперед со скоростью 1 единица в секунду transform.Translate(Vector3.forward\*Time.deltaTime);

}

void Update ()

{

// Движение объекта вперед в глобальной системе координат transform.Translate(Vector3.forward\*Time.deltaTime, Space.World);

}

void Update ()

{

// Движение объекта вдоль оси Z transform.Translate(0, 0, Time.deltaTime);

}

void Update ()

{

// Движение объекта вперед относительно камеры transform.Translate(Vector3.forward\*Time.deltaTime, Camera.main.transform);

}

* метод MoveTowards. Движение объекта по прямой до указанной позиции с заданной скоростью.

public Transform target; float speed = 1;

void Update()

{

transform.position = Vector3.MoveTowards(transform.position, target.position, speed \* Time.deltaTime);

}

* Vector3.Lerp() – линейная интерполяция между двумя векторами, метод, не относящийся напрямую к движению объекта, но часто использующийся для этой цели.

В нашем проекте мы будем использовать метод MoveTowards. Создадим метод Move , отвечающий за перемещение.

private Vector3 direction;

public float speed;

private float horizontal;

void Start()

{speed = 3;}

private void Move()

{

direction = Vector3.right \* horizontal;

transform.position = Vector3.MoveTowards(transform.position, transform.position + direction, speed \* Time.deltaTime);

}

void Update()

{

horizontal = Input.GetAxis("Horizontal");

if (Input.GetButton("Horizontal"))

Move();

}

В методы Move будем осуществлять движение с помощью MoveTowards, в параметры которому передается переменная direction , отвечающая за направление движения. Direction в свою очередь обращается к переменной horizontal. Переменная horizontal получает данные в функции Update с помощью Input.GetAxis("Horizontal").Таким образом при запуске игры и нажатии клавиш A,D, -> ,<- наш персонаж будет двигаться.

Для добавления анимации разворота персонажа при изменении направления движения создадим метод Flip.

private bool right; // повернут ли игрок вправо ?

private void Flip()

{

right = !right; //меняем сторону

Vector2 sc = transform.localScale; // временный вектор для хранения Scale

sc.x \*= -1; // Изменили scale , чтобы поворнуть игрока

transform.localScale = sc;

}

Для осуществления поворота необходимо поменять значения Transform-Scale-X на противоположное . Затем в методе Move необходимо указать условия выполнения поворота , используя переменную hotizontal.

if (horizontal > 0 && right)

Flip();

else if (horizontal < 0 && !right)

Flip();

Готово! Теперь при запуске игры персонаж поворачивается при движении.

**Враги**

Перейдем к созданию префаба для врага. Находим нужный спрайт , перетаскиваем его на сцену , настраиваем размер и изменяем имя объекта на Enemy. После этого создаем тег enemy и присваиваем его к нашему объекту. Затем добавляем компоненты Box Collider 2D и RigidBody 2D(не забудьте установить галочку напротив Freeze Rotation).Создаем скрипт Enemy и добавляем его к врагу. Готовый объект добавляем в префабы.

Добавим функционал к нашему врагу.

public int health = 100;

void Update()

{

if (health <= 0) //Объект уничтожается , когда здоровье ниже нуля

Destroy(gameObject, 1f);

}

Теперь добавим генератор врагов(spawner). Создаем новый объект Spawner и добавляем к нему скрипт с таким же названием. Задача этого объекта – генерировать врага в указанном месте , если прошлый враг был уничтожен .

GameObject CurrentEnemy; //текущий враг

public GameObject enemyPref; // префаб врага

void Start()

{

//создаем врага

CurrentEnemy = Instantiate(enemyPref, transform.position, Quaternion.identity);

}

void Update()

{

if (CurrentEnemy == null) //Если враг был уничтожен , создаем нового

CurrentEnemy = Instantiate(enemyPref, transform.position, Quaternion.identity);

}

Далее удалим текущего врага с нашем сцену и перенесем объект спаунера в нужное место. Добавляем префаб врага к скрипту спаунера (перетащить префаб из навигатора в окно инспектора спаунера).

Теперь при запуске игры в указанном месте появляется враг. Если вручную удалить врага , когда игра запущена , сразу появится новый враг.

**Стрельба (коллизии и столкновения)**

Для стрельбы нужно создать оружие и пулю. К сожалению в наших ассетах нет подходящих спрайтов для оружия , но можно взять любой подходящий по форме , например цветок (который потом будет стрелять). Закидываем спрайт на сцену и добавляем в навигаторе в нашего персонажа. Переименовываем как Gun.

Теперь создадим префаб пули . Находим тайл , переносим на сцену , добавляем Circle Collider 2D , RigidBody 2D и скрипт .Важно поставить галочку в поле Is Trigger, чтобы поля проходила сквозь объекты. Переносим объект в префабы.

Организуем движение пули с помощью transform.position. Для того , чтобы на пулю не действовала гравитация установим компонент RigidBody2D-Body Type равным Kinematic.

float speed = 0.4f; //скорость пули

public int damage = 40; // Урон

public int direction; // Направление

void Update()

{

transform.position += Vector3.right \* direction \* speed;

}

Переменная direction отвечает за направление полета пули. Ей будет передаваться значение из функции Shoot , которую мы создадим позже.

Также нам необходимо удалять пулю , чтобы не захламлять память. Воспользуемся методом OnBecameInvisible , который выполняется в тот момент времени , когда объект уходит за пределы экрана.

private void OnBecameInvisible()

{

Destroy(gameObject, 3f); //удаляем пулю через 3 секунды

}

Теперь осуществим детекцию столкновения с врагом. Используем метод OnTriggerEnter2D , так как коллайдер нашего объекта находится в режиме триггера. Этот метод происходит один раз в момент, когда объект касается объекта с триггером.

private void OnTriggerEnter2D(Collider2D collision)

{

if (collision.tag == "enemy") // если попали в врага

{

GameObject temp = collision.gameObject;

temp.GetComponent<Enemy>().health -= damage; //наносим урон

Destroy(gameObject); // удаляем пулю

}

}

Важно учитывать , что коллизия будет работать , если хотя бы у одного из объекта есть RigidBody , и у обоих есть коллайдеры.

Пуля готова . Научим нашего персонажа стрелять. Для этого в скрипт персонажа добавляем метод Shoot , который будет создавать пулю и передавать ей значение direction. Для реализации метода Shoot необходимо добавить переменные BulletPref и gunPos , хранящие префаб пули и позицию оружия соответственно . Значения эти параметрам необходимо передать вручную (перетащить в окно инспектора).

public GameObject bulletPref;

public Transform gunPos;

private void Shoot()

{

GameObject temp = Instantiate(bulletPref, gunPos.position, Quaternion.identity);

temp.name = "Bullet";

temp.GetComponent<Bullet>().direction = (!right) ? 1 : -1;

}

Чтобы организовать выстрел по нажатию левой кнопки мыши , добавим условие в метод Update.

if (Input.GetKeyDown(KeyCode.Mouse0))

Shoot();

При запуске пуля не должна проходить сквозь врага , и после какого то количества выстрелов враг должен уничтожаться. Также проверьте что объекты пули уничтожаются через какое-то время.