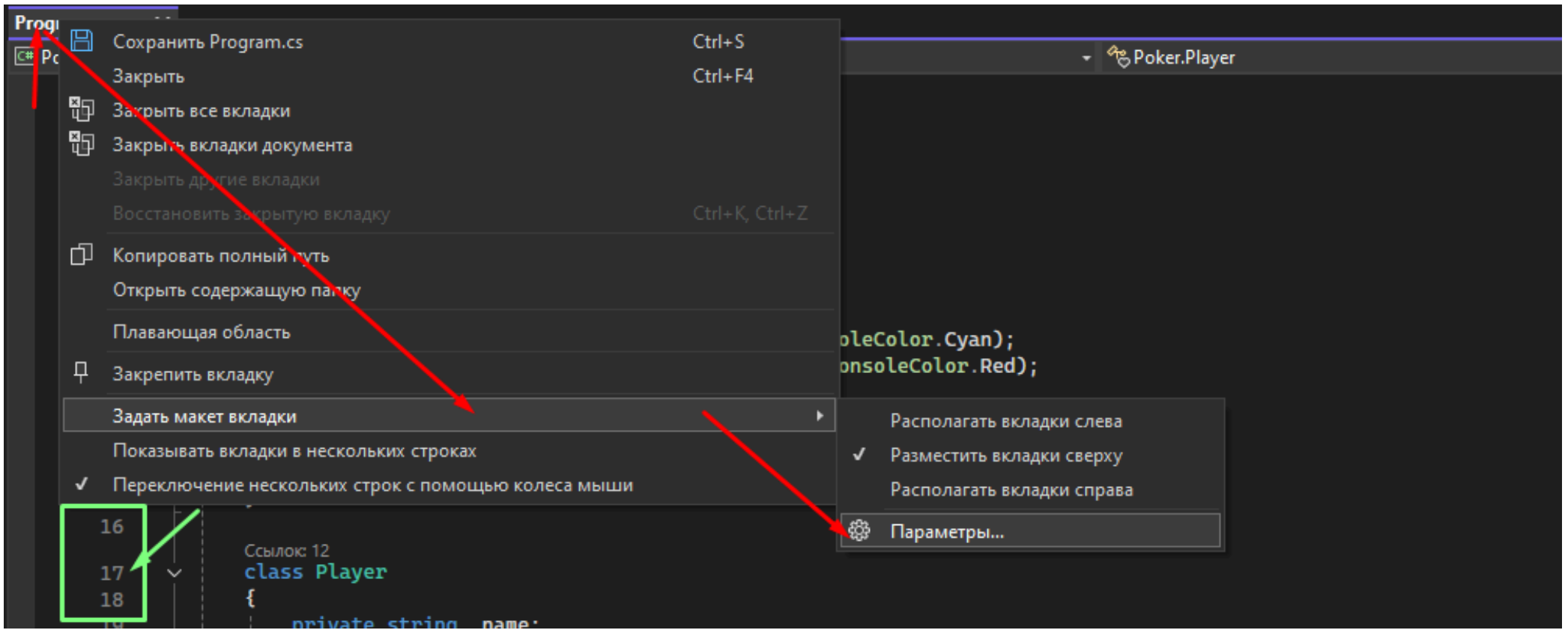
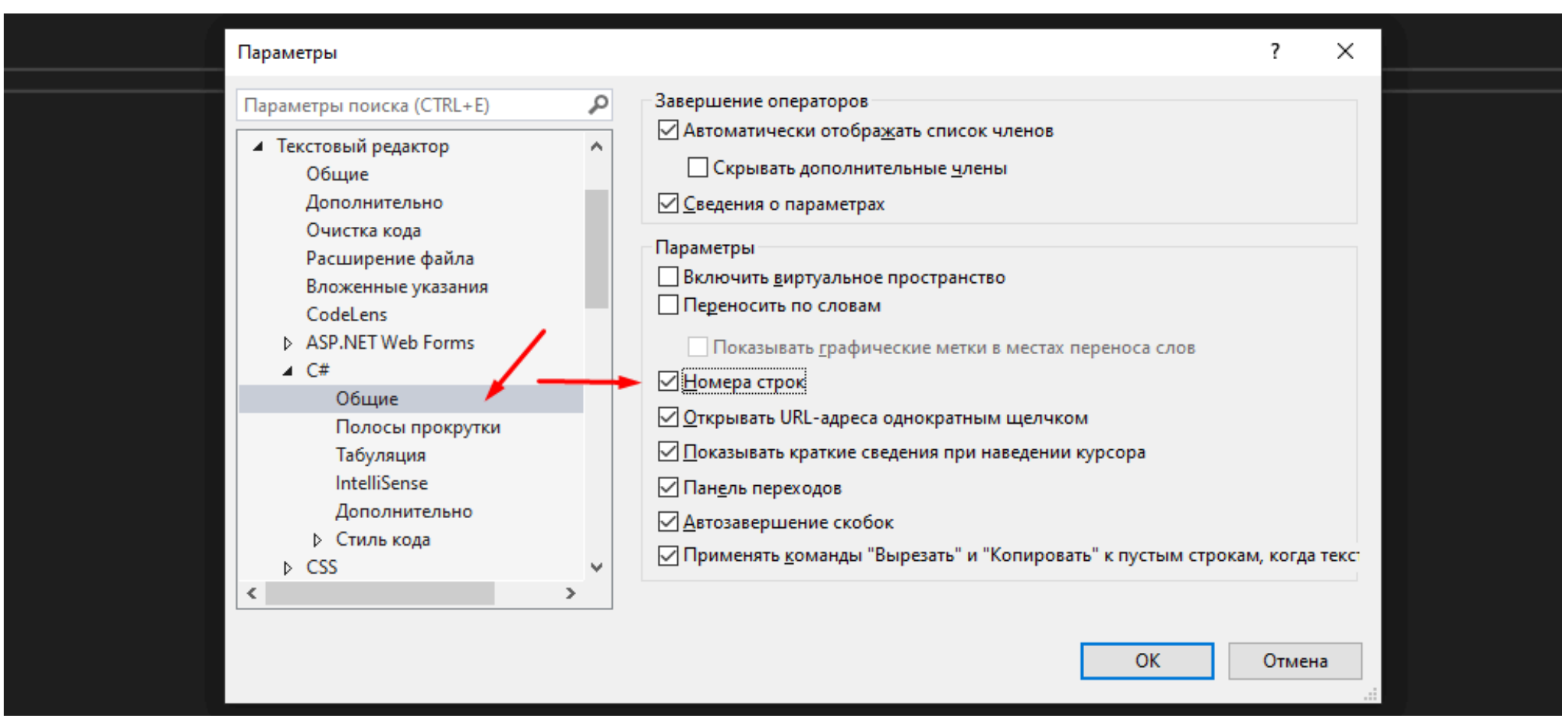
# Unity - II

# Visual Studio

## ※ Включить отображение номера строки

В контекстном меню окна с кодом Задать макет вкладки / Параметры /




Unity необходима **Visual Studio** для компиляции проекта, даже если рабочим редактором кода будет другая программа.

## ※ Горячие клавиши

**Хоткеи для Visual Studio**

**TAB** - автоматически “допечатать” текст, который предлагает среда при анализе команд пользователя
**Alt + стрелка вверх/вниз** - перемещает текущую строку кода вверх/вниз
**Ctrl + D** - продублировать текущую строку кода (по умолчанию может быть иное сочетание клавиш)
**Ctrl + L** - удалить текущую строку
**Ctrl + K, Ctrl + D** - автоформатирование всего кода в открытом файле - отступы, пробелы
**Ctrl + R, Ctrl + R** - переименовать выделенную переменную во всем коде
**Ctrl + K, Ctrl + C** - закомментировать выделенные строки кода (поставить // в начале)
**Ctrl + K, Ctrl + U** - раскомментировать выделенные строки кода (удалить // в начале)
**Ctrl + пробел** - раскрыть окно выбора функций/методов после точки: **random.**

## Скрипты

При создании скрипта имя **класса** будет взято из имени **файла** скрипта. Если потом переименовать скрипт, то имя класса в скрипте НЕ изменится автоматически. Скрипт будет работать, но для избежания путаницы надо менять имя класса в скрипте на новое имя скрипта.

## ※ Важные классы

**MonoBehaviour** - базовый класс для всех новых скриптов (сценариев), предоставляет список всех функций и событий, доступных для стандартных сценариев, прикрепленных к игровым объектам.

**Transform** - у каждого объекта есть позиция, ротация, масштаб. Все эти характеристики определяются компонентом Transform, и для взаимодействия с ними надо пользоваться этим классом.

**Rigidbody** - для большинства элементов физический движок предоставляет самый простой набор инструментов для перемещения объектов, обнаружения триггеров и столкновений. Данный класс предоставляет все свойства и функции, необходимые для работы с физикой в Unity.

## ※ Атрибуты

Атрибуты это специальные маркеры в коде перед классом, полями (переменными, свойствами) и методами (функциями), необходимые для указания на особое поведение. Некоторые из них:

[**Header**] - заголовок

[**HideInInspector**] - не показывать в инспекторе, даже если стоит модификатор доступа public.

[**SerializeField**] - показывает в инспекторе, даже если поля приватные.

[**Range**] - значение будет ограничено диапазоном, регулируется в инспекторе ползунком.

[**Min**] - ограничивает минимальное значение границей (Max).

[**Tooltip**] - устанавливает подсказку для отображения в инспекторе.

[**RequireComponent**] - добавляет компонент указанного типа объекту, если его еще нет на объекте. Нужно для гарантии что необходимый тип ресурса всегда будет на настраиваемом объекте на сцене.

```
[RequireComponent(typeof(AudioSource))] // этот атрибут надо указать до объявления класса в скрипте

[Header("Пример заголовка")]

[HideInInspector]
public int Strength;

[SerializeField]
private float _scale = 1f;

[Tooltip ("Пример текстовой подсказки")]
[Range (10, 30)]
public float Duration;

[Min (10)]
public float Speed;
```

## ※ Обработка исключений

Тема обработки исключений не так нужна в начале знакомства с Unity, но её все равно надо будет пройти, когда-нибудь...

В контексте Unity надо обрабатывать исключение **NullReferenceException**, которое происходит при попытке получить доступ к ссылочной переменной, которая не ссылается ни на один объект. Если ссылочная переменная не ссылается на объект, она будет рассматриваться как **null**. Среда сообщит что получить доступ невозможно и выбросит исключение **NullReferenceException**.

```
using UnityEngine;

public class ExampleScriptException : MonoBehaviour
{
    private void Start()
    {
        try // в блок try положить код, который возможно вызовет ошибку
        {
            GameObject player = GameObject.Find("player ");
            Debug.Log(player.name);
        }
        catch (NullReferenceException ex)
        {
            Debug.Log("Нет объекта с именем player ");
        }
    }
}
```

## ※ Основные методы

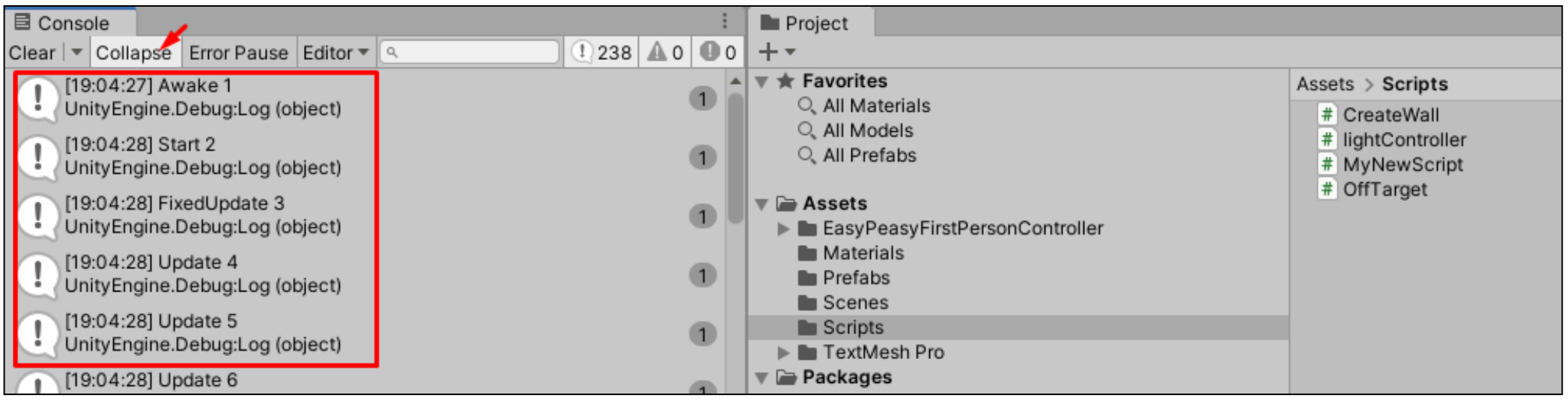
**Awake()** - выполняется сразу после создания объекта на сцене, выполняется один раз и всегда до метода **Start()**.
**Start()** - выполняется только один раз, при запуске сцены.
**Update()** - выполняется каждый кадр, частота выполнения зависит от FPS.
**FixedUpdate()** - выполняется каждый фиксированный отрезок времени.
**LateUpdate()** - выполняется после **Update()** и **FixedUpdate()**, используется для расчета положения камеры (повернуть взгляд персонажа).
**OnEnable()** - выполняется при включении объекта - установке флага **SetActive(true)**. Окно [**Главное меню**] включают на время.
**OnDisable()** - выполняется при выключении объекта - установке флага **SetActive(false)**. Окно [**Настройки**] выключает [**Главное меню**].
**OnDestroy()** - выполняется один раз при уничтожении объекта.
**OnGUI()** - в этом методе вызывается код системы отрисовки элементов интерфейса поверх всего остального (GAME OVER).

В меню редактора **Edit / Project Settings / Time / Fixed Timestep** можно установить интервал вызова метода **FixedUpdate()**.

Проверить порядок запуска описанных методов можно создав скрипт с кодом, и добавив скрипт любому объекту на сцене:

```
using UnityEngine;

public class MyNewScript : MonoBehaviour
{
    private int i = 1;

    private void Start()
    {
        Debug.Log($"Start {i++}");
    }

    private void Update()
    {
        Debug.Log($"Update {i++}");
    }

    private void Awake()
    {
        Debug.Log($"Awake {i++}");
    }

    private void FixedUpdate()
    {
        Debug.Log($"FixedUpdate {i++}");
    }
}
```

В окне [**Console**] включить опцию [**Collapse**]. При запуске здесь будет отображаться вывод сообщений в лог с количеством повторов:



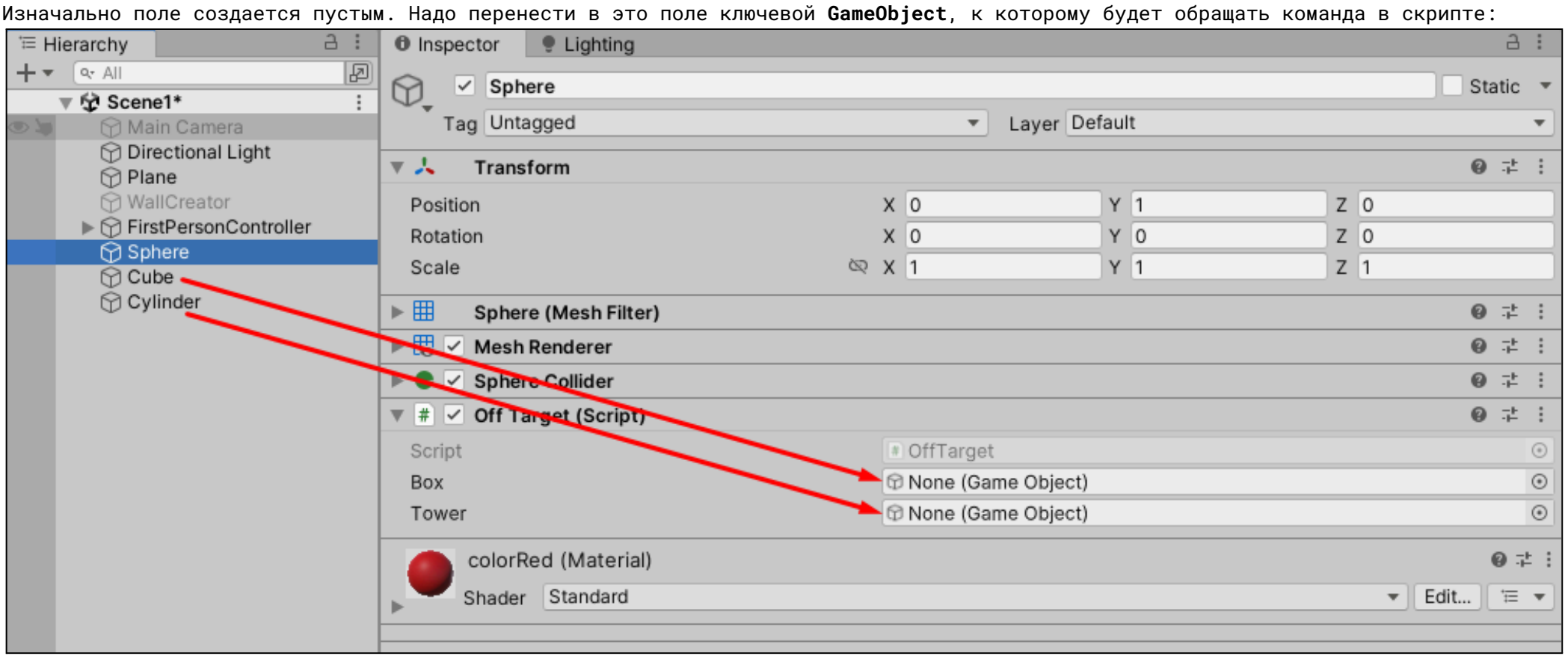

Если скрипт добавлен объекту (стал компонентом), то обратиться к этому объекту в скрипте можно через ключевое слово **gameObject**.
Команда **SetActive(false)** сделает ключевой объект неактивным - аналог снятия галочки в свойствах объекта (окно инспектор):

```
    private void Start()
    {
        gameObject.SetActive(false); // при выключении объекта в его скрипте также выполняется метод OnDisable()
    }

    private void OnDisable()
    {
        print($"{gameObject.name} :: Безобразие! Меня отключают!");
    }
```

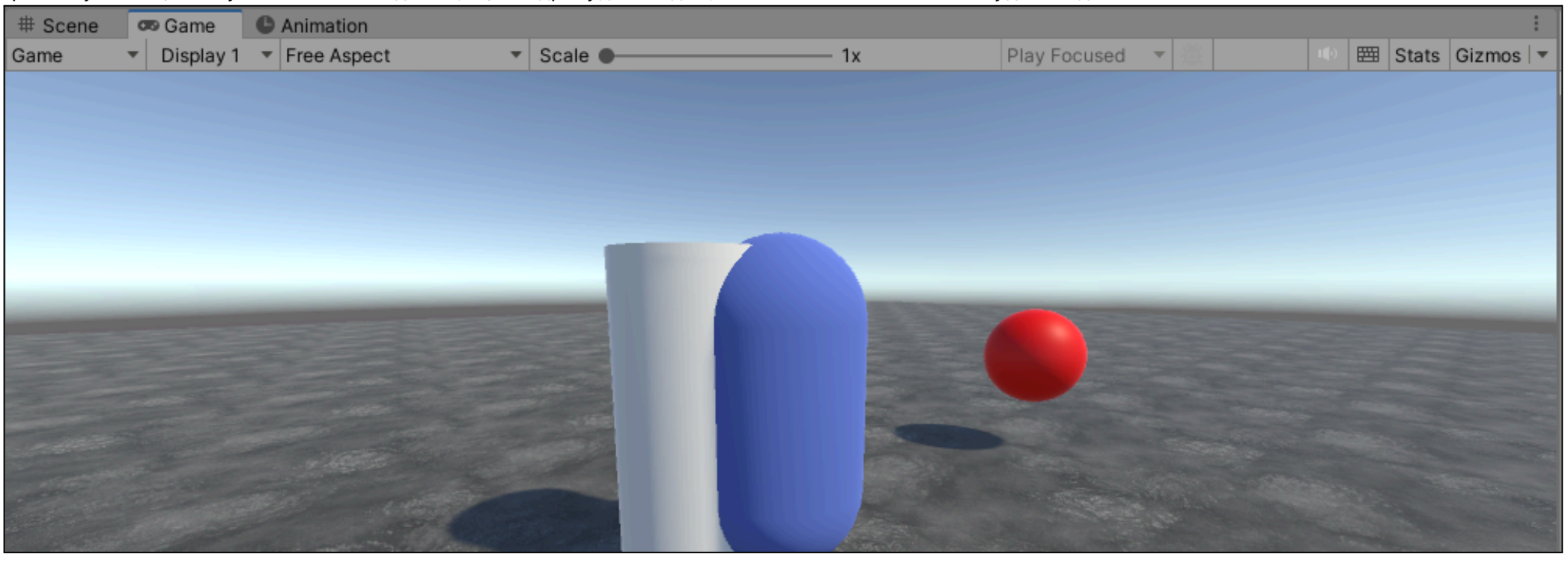

Если надо получить доступ к другому объекту на сцене, необходимо сперва создать ссылку на этот объект в скрипте.
Для создания ссылки объявить публичную переменную типа **GameObject**.

```
using UnityEngine;

public class OffTarget : MonoBehaviour
{
    public GameObject box; // создаст ссылку с именем Box
    public GameObject tower; // создаст ссылку с именем Tower

    private void Start()
    {
        Debug.Log(gameObject.activeSelf); // вывести в лог статус активности объекта, на котором висит скрипт (включен или нет)

        box.SetActive(false); // обращаемся к объекту box (Cube) и через оператор "." выключаем его - метод SetActive(false)

        // после получения доступа к объекту можно обращаться к его компонентам через GetComponent<Component>(), пример:
        Destroy(tower.GetComponent<CapsuleCollider>()); // уничтожаем компонент CapsuleCollider для объекта tower (Cylinder)
    }
}
```

После создания **публичной** (**public**) ссылки в скрипте, в Unity у компоненте скрипта появится поле с именем этой ссылки (**Box** и **Tower**). Изначально поле создается пустым. Надо перенести в это поле ключевой **GameObject**, к которому будет обращать команда в скрипте:



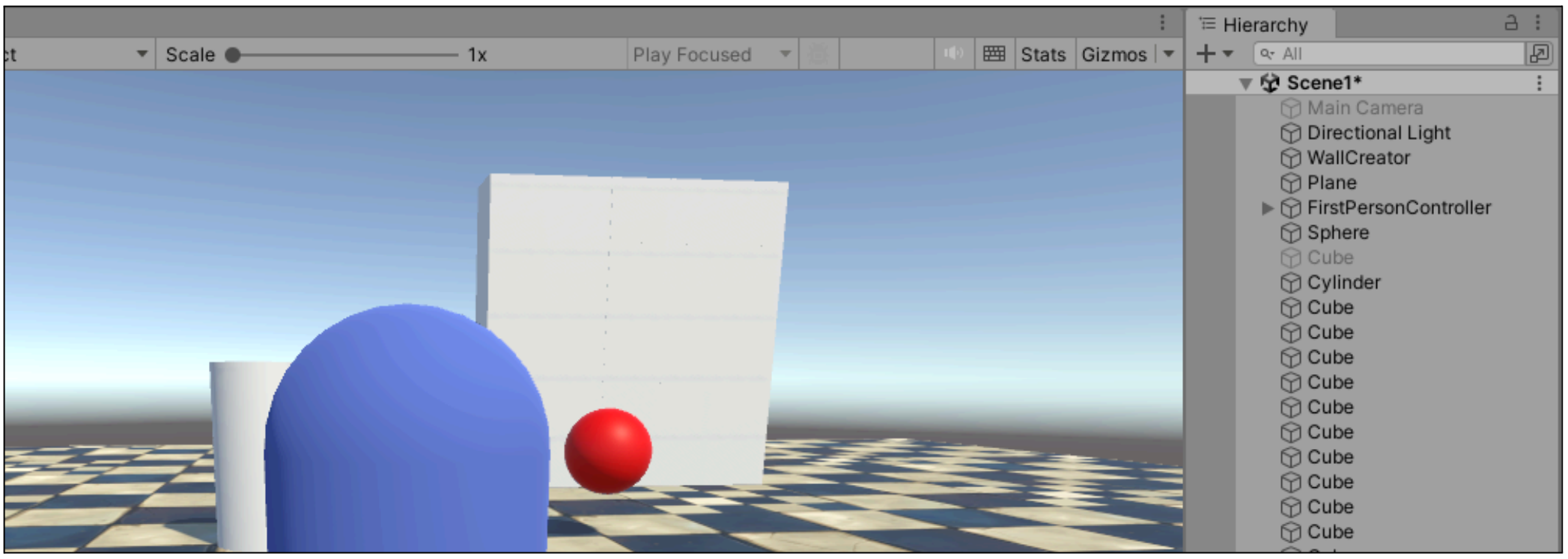

При запуске сцены куб станет невидимым, цилиндр будет видно, но сквозь него можно будет ходить:



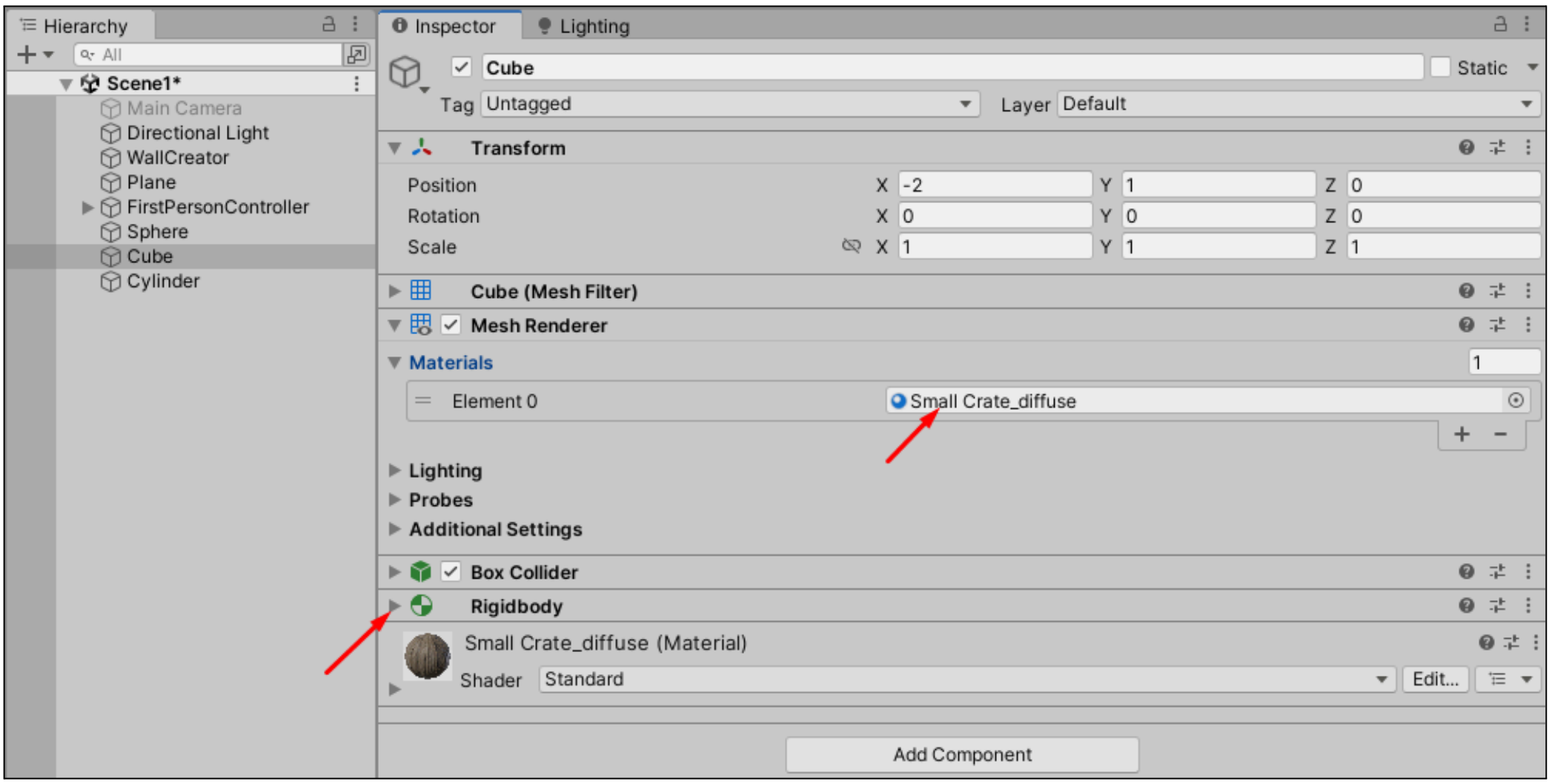
Через скрипты можно создать на сцене полноценный объект. Создать скрипт **CreateWall**:

```
using UnityEngine;

public class CreateWall : MonoBehaviour
{
    private void Start()
    {
        for (int y = 0; y < 5; y++)
        {
            for (int x = 0; x < 5; x++)
            {
                GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube); // создать новый куб
                cube.transform.position = new Vector3(x, y + 0.5f, 5); // положить куб в заданные координаты
                // размер куба 1x1x1 поэтому надо поднять его на половину его высоты - на +0.5 от пола, пол на высоте 0
                cube.AddComponent<Rigidbody>(); // добавить кубу физическое тело - он сможет падать
            }
        }
    }
}
```

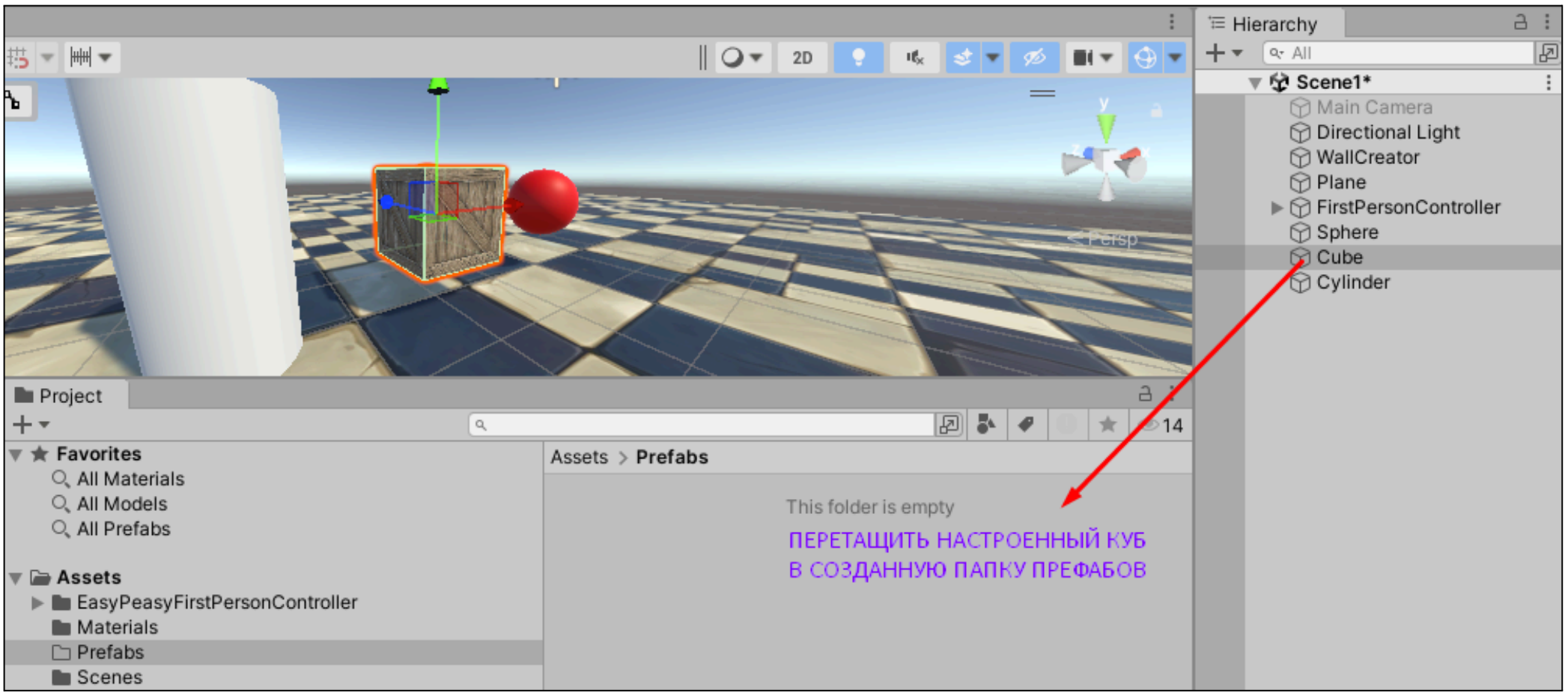
Создать на сцене пустой объект (в примере это **WallCreator**), повесить на него созданный скрипт, запустить сцену - в итоге при запуске будет построена стена из белых кубов 5x5:



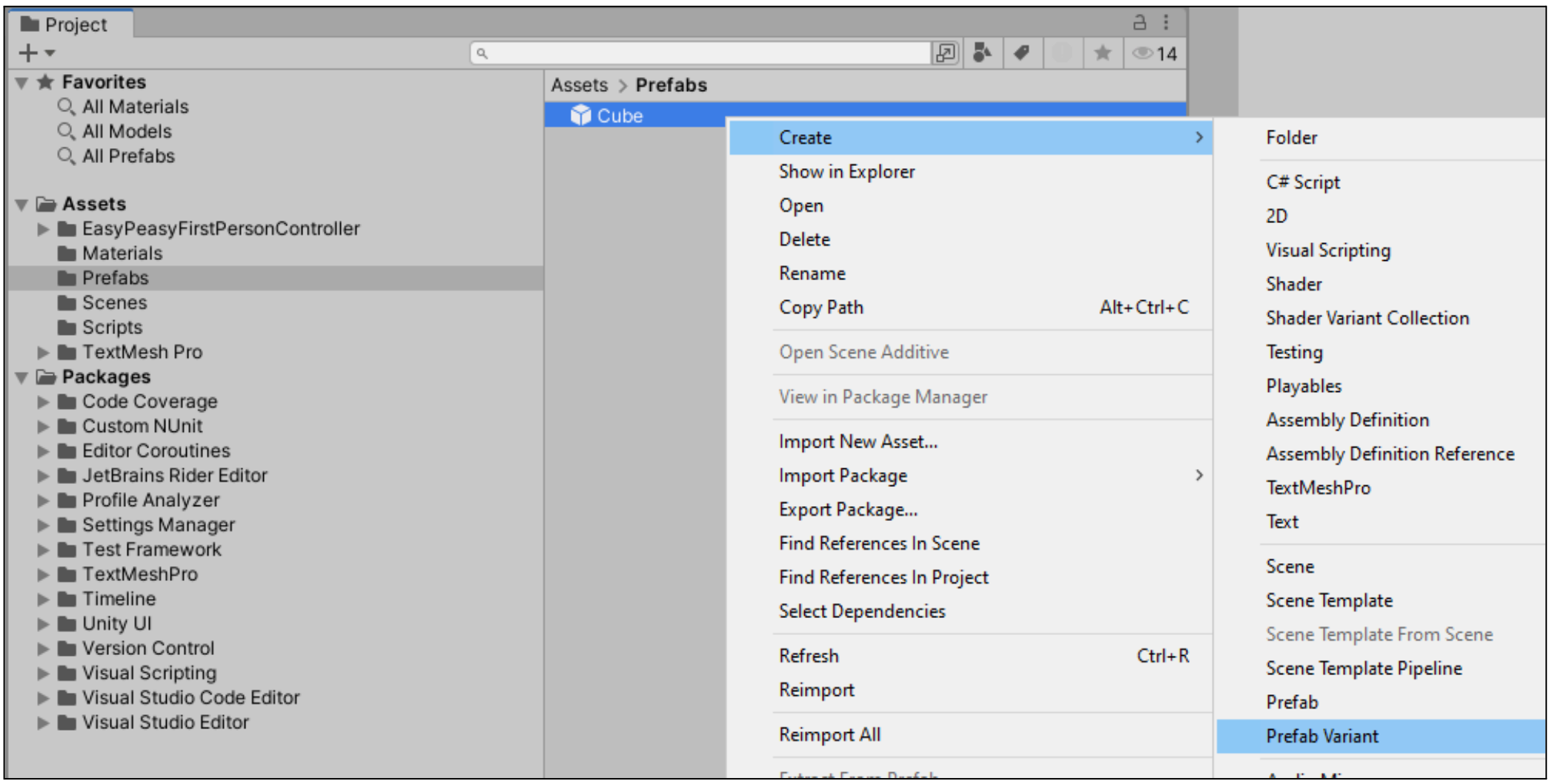

Чтобы не добавлять каждому новому объекту через код необходимые компоненты, лучше создать префаб объекта, на котором уже будут настроены все нужные компоненты. Добавить размещенному на сцене кубу любую текстуру, и компонент **Rigidbody** (физическое тело):

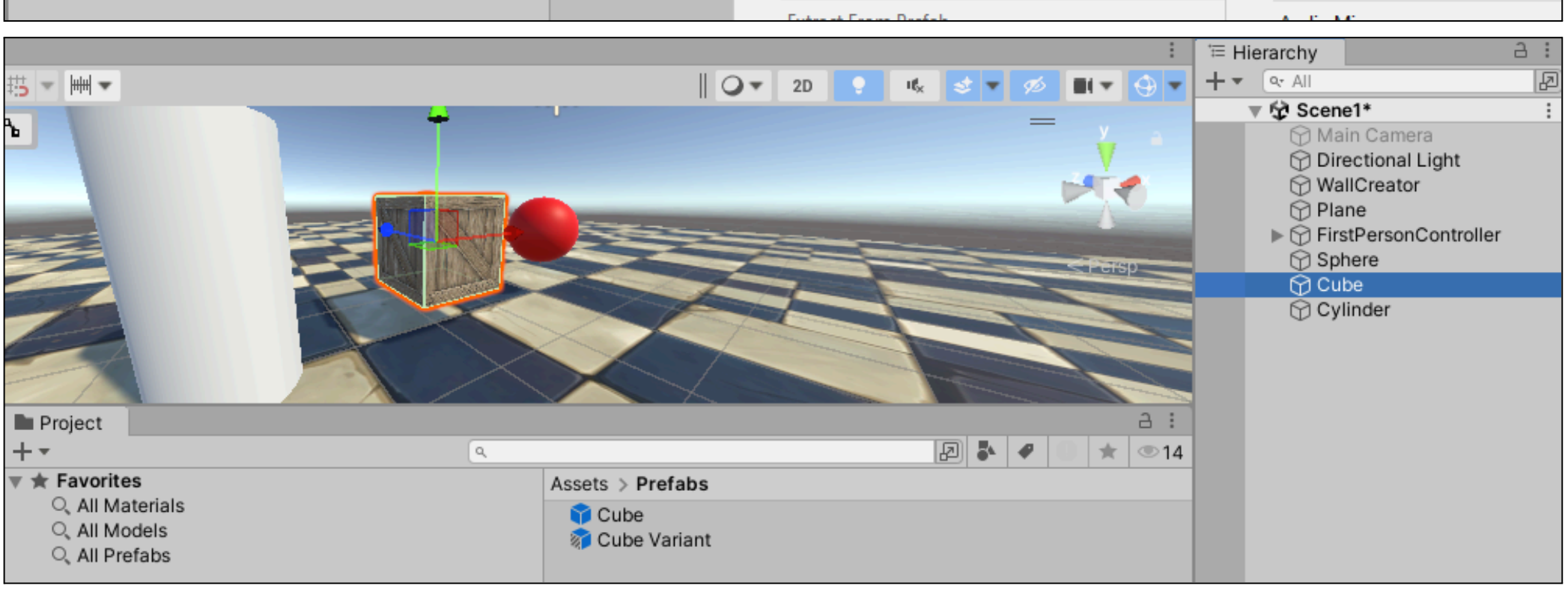



Создать в структуре проекта папку для префабов. Перетащить в нее настроенный куб из окна иерархии объектов.
В папке появится префаб куба (с синим маркером), а куб на сцене станет ссылкой на этот префаб и тоже посинеет.
Куб на сцене удалить или распаковать в обычный объект (в контекстном меню объекта сцены - **Prefab** - **Unpack**).




Все копии префаба будут иметь одни и те же исходные параметры. Для того чтобы сохранять различия у экземпляров на сцене, надо создавать варианты префаба. Конкретно в этом примере это не обязательно, но для ознакомления и демонстрации сделано таким образом. Создать вариант префаба (**Prefab Variant**) можно из контекстного меню этого префаба:

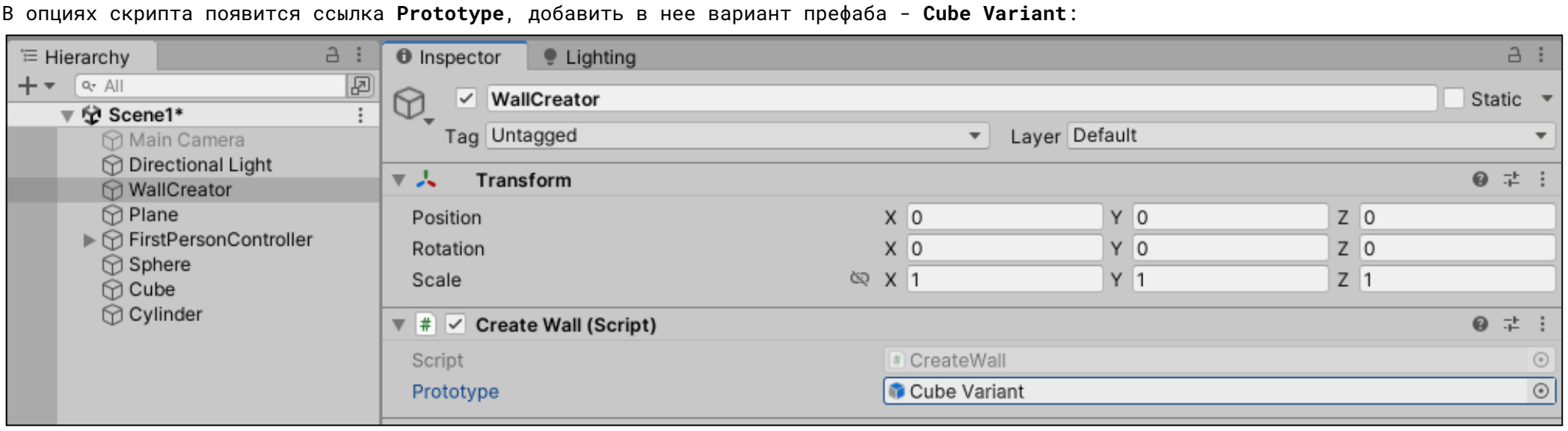


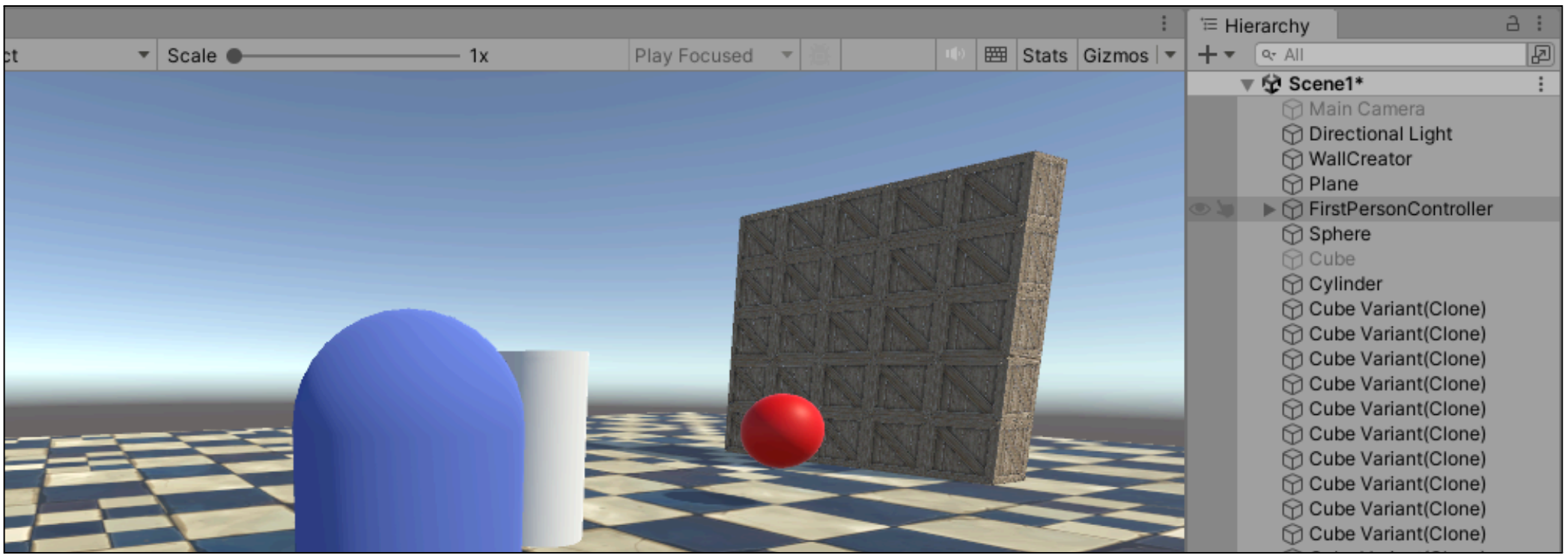

Теперь скрипт по созданию объектов будет выглядеть короче, но нужно получить ссылку на объект, который будет создаваться в скрипте - на вариант префаба. Добавить в скрипт публичную переменную **prototype** и изменить код метода **Start()**:

```
using UnityEngine;

public class CreateWall : MonoBehaviour
{
    public GameObject prototype;

    private void Start()
    {
        for (int y = 0; y < 5; y++)
            for (int z = 0; z < 5; z++)
                Instantiate(prototype, new Vector3(5, y + 0.5f, z), Quaternion.identity);
    }
}
```

В опциях скрипта появится ссылка **Prototype**, добавить в нее вариант префаба - **Cube Variant**:




Теперь после запуска сцены, будет создана стенка из уже настроенных кубов:




## ※ Перемещение объектов

Новый скрипт **MovableObject** считывает нажатые кнопки. Скрипт добавить сфере, которая будет перемещаться по сцене.

```
using UnityEngine;

public class MovableObject : MonoBehaviour
{
    private void Update()
    {
        // Translate будет двигать объект на единицу расстояния, срабатывать раз в кадр, и чтобы объект не улетел далеко
        // надо это компенсировать долями от секунды между текущим кадром и прошлым - Time.deltaTime, при 40 кадрах доля 1/40
        // направление движения по осям будет задавать множитель -1 или +1 из с кнопок управления осями Input.GetAxis()
        transform.Translate(Vector3.forward * Time.deltaTime * Input.GetAxis("Vertical")); // GetAxis("Vertical") - WS
        transform.Translate(Vector3.right * Time.deltaTime * Input.GetAxis("Horizontal")); // GetAxis("Horizontal") - AD
        // forward / back - ось Z, right / left - ось X, up / down - ось Y
        // Time.deltaTime - разница во времени между графическими кадрами
        // Time.fixedDeltaTime - разница во времени между физическими кадрами, подробнее о кадрах в разделе о RigidBody
    }
}
```

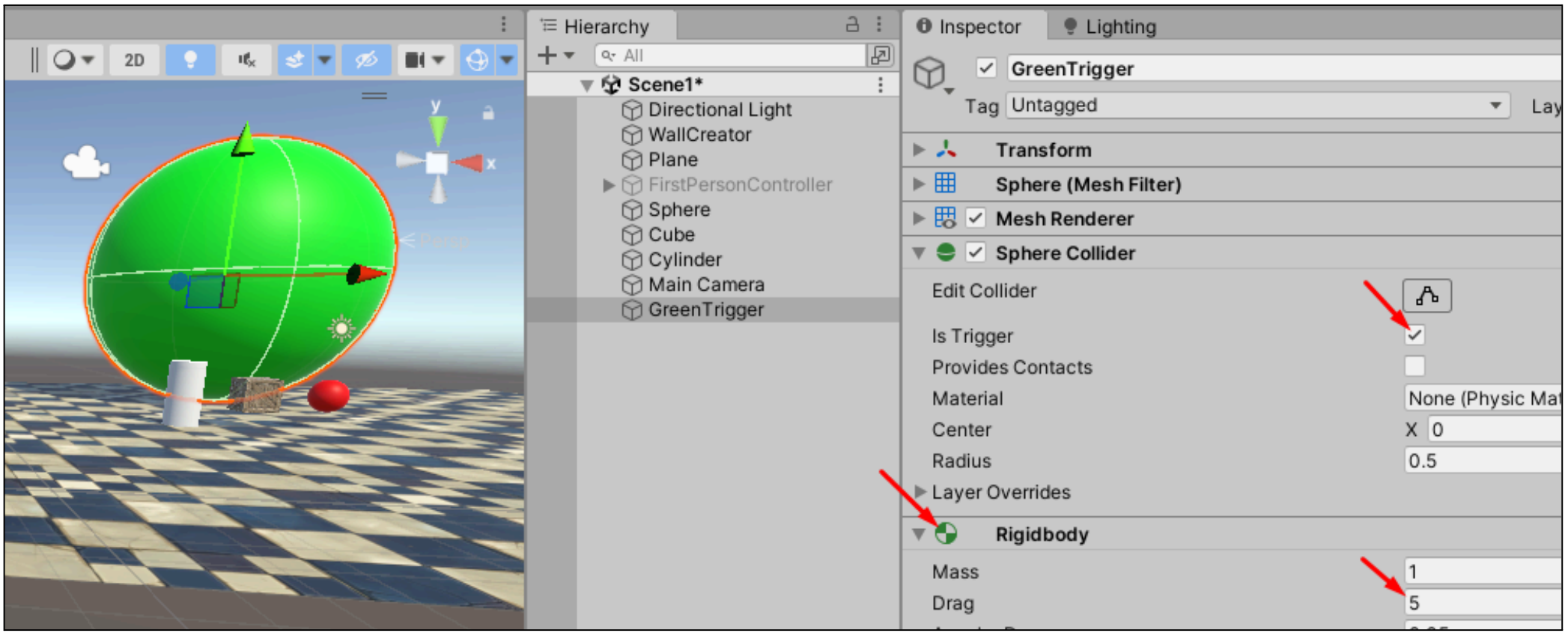
Контролер игрока в сцене надо отключить, включить камеру, и выставить ей координаты для удобного наблюдения за сценой.

Так как у сферы нет физического тела, она свободно парит и не падает. Чаще всего она будет проходить кубы в стенке кубов насквозь, лишь иногда отталкивая их.

## ※ Триггер

Объект с коллайдером в режиме триггера (**Is Trigger**) будет вызывать у себя в скриптах метод **OnTriggerEnter()** каждый раз, когда его коллайдер будет пересекать другой объект. Один из объектов взаимодействия всегда должен обладать компонентом **Rigidbody** (физическим телом) - либо триггер, либо заходящий в него объект. Это относится и к обработке столкновений / соприкосновений объектов.

На сцену добавлен новый объект - сфера, которая переименована в **GreenTrigger**. Она будет коллайдером, работающим в режиме триггера, с компонентом **Rigidbody**, у которого настроено сопротивление воздуха **Drag** = 5.




Этой сфере добавлен новый скрипт **Trigger** с кодом:

```
using UnityEngine;

public class Trigger : MonoBehaviour
{
    private void OnTriggerEnter(Collider other) // other - объект, который вызывает срабатывание триггера
    {
        int x = 0;

        if (other.name == "Sphere") // проверка имени объекта, игрока можно проверять по тегу: other.CompareTag("Player")
            x = 10;

        transform.Translate(Vector3.up * (10 + x));
    }
}
```

Если зеленый шар, с коллайдером в режиме триггера, будет соприкасаться с полом, то зеленый шар телепортируется вверх на 10 метров.
Если зеленый шар коснется с объекта с именем = Sphere (управляемый красный шар), то зеленый шар телепортируется на 20 метров.

**OnTriggerEnter(Collider** other) срабатывает в момент входа объекта other в коллайдер.
**OnTriggerExit(Collider** other) срабатывает в момент выхода объекта other из коллайдера.
**OnTriggerStay(Collider** other) срабатывает каждый кадр пока происходит пересечение коллайдера объектом other.

**Методы обрабатывающие соприкосновения:**
**OnCollisionEnter(Collision** collision) срабатывает в момент установления касания с объектом collision.
**OnCollisionExit(Collision** collision) срабатывает в момент прекращения касания с объектом collision.
**OnCollisionStay(Collision** collision) срабатывает каждый кадр, пока продолжается соприкасание с объектом collision.
Соприкосновения обрабатываются только пока объекты движутся. В целях оптимизации обработка этих событий прекращается как только движение сильно ослабевает или объект прекращает движение (затухание).

Коллайдеры взаимодействуют друг с другом в зависимости от того, как настроены их компоненты **Rigidbody**. Три важные конфигурации: **Статический** - без **Rigidbody**, **Динамический** - есть **Rigidbody**, **Кинематический** - есть **Rigidbody** с флагом **Is Kinematic** (не падает).

## Физическое тело

Компонент **Rigidbody** добавляет объектам физические свойства - массу, плотность, инерцию. Получать импульс, крутящий момент.

Движок Unity состоит из двух частей - графического движка и физического движка. Графический рисует картинку, физический рассчитывает физику.

Метод **Update()** срабатывает с частотой отрисовки кадров графическим движком.

Метод **FixedUpdate()** срабатывает с частотой расчета физических кадров - частота работы физического движка.

**Mass** - масса определяет силу взаимодействия при столкновении (масса + скорость = сила эффекта).

**Drag** - плотность окружающей среды вокруг объекта - для имитации сопротивления воздуха или воды.

**Angular Drag** - плотность окружающей среды вокруг объекта - для имитации сопротивления вращению в воздухе и в воде.

**Automatic Center Of Mass** - точка равновесия определяется движком, если отключить, то можно настроить смещение центра массы.

**Automatic Tensor** - автоматический расчет инерции - если отключить, то можно настроить как трудно повернуть объект вокруг осей.

**Use Gravity** - применение гравитации.

**Is Kinematic** - если включено объект перестает взаимодействовать с физикой, его нельзя толкнуть, заставить крутиться, но об него можно удариться. Перемещением объекта можно управлять только через компонент **Transform**. Это полезно для перемещения платформ.

**Кинематические Rigidbody** должны использоваться у объектов, которые могут двигаться, или включаться/выключаться. Например, дверь. Обычно дверь неподвижна (ее нельзя двигать, если в нее упереться), но она сама может двигаться когда ее открывают / закрывают.

**Interpolate** - точность вычисления физики на объекте, опция применяется при наличии тряски в процессе перемещения:

- **None** - не применять интерполяцию
- **Interpolate** - нахождение точки между двумя точками (вычисление следующего кадра на основе 2 предыдущих кадров - среднее)
- **Extrapolate** - продолжение точки на основе предыдущих (вычисление-прогнозирование следующего кадра на основе предыдущего)

**Interpolate** сглаживает результат работы графической части движка Unity, когда расчетов физических взаимодействий в кадре мало.

**Collision Detection** - отслеживание (обнаружение) столкновений и точность их обработки:

- **Discrete** - самый простой и наименее надежный (объект на высоких скоростях может проходить сквозь другие объекты)
- **Continuous** - точность столкновения рассчитывает препятствие на пути другого движущегося объекта
- **Continuous Dynamic** - столкновение обрабатываются на всем пути движения объекта (один объект не может проскочить другой)
- **Continuous Speculative** - минимизирует вероятность "проскальзывания" или "проваливания" объектов сквозь друг друга

**Discrete** - расчет столкновений происходит прерывисто, отдельно в каждом физическом кадре. Поэтому если в физическом кадре объекты не пересекаются, то нет столкновения, и поэтому быстро движущийся объект может пролететь другой.

**Continuous** - расчет столкновений происходит непрерывно, т.е. тело учитывает свое положение в предыдущем физическом кадре. Если между текущим положением и прошлым есть физическое тело, значит столкновение произошло. Подходит для быстрых объектов - пуль, но этот режим не рассчитает столкновение двух летящих пуль.

**Continuous Dynamic** - рассчитает столкновение двух быстро движущихся физических тел.

**Constraints** - блокировка физики по отдельным осям - движения и вращения.

**Layer Overrides** - матрица столкновений.

Объекты с компонентом **Rigidbody** (с физическими телами) некорректно двигать, принудительно меняя их координаты в компоненте **Transform**. Логичнее прикладывать к ним воздействующие силы, а дальше все рассчитает движок.

## ※ Физические силы

**Гравитация:** Unity имитирует гравитацию, притягивающую объекты вниз. Компонент **Rigidbody**, когда прикреплен к объекту, реагирует на гравитацию. Можно настраивать параметры гравитации, такие как направление и сила.

**Силы и Торки:** Силы и торки могут быть применены к объектам с помощью компонента **Rigidbody**. Силы используются для изменения линейной скорости, а торки - для изменения угловой скорости объекта.

**Сопротивление Воздуха и Трение:** Unity позволяет включать сопротивление воздуха и трение для объектов, что делает их движение более реалистичным. Эти параметры могут быть настроены в компоненте **Rigidbody**.

**Импульсы и Моменты:** Импульсы представляют собой резкое изменение количества движения объекта. Моменты применяются для изменения угловой скорости. Импульсы и моменты могут быть использованы для создания взрывов, ударов и других резких изменений.

Комбинирование этих элементов и настройка параметров физики позволяют создавать разнообразные и увлекательные сцены с реалистичным и интересным движением объектов.

Физические силы надо обрабатывать в методе **FixedUpdate()**. Новый скрипт для демонстрации использования физики **PhysicsTester**:

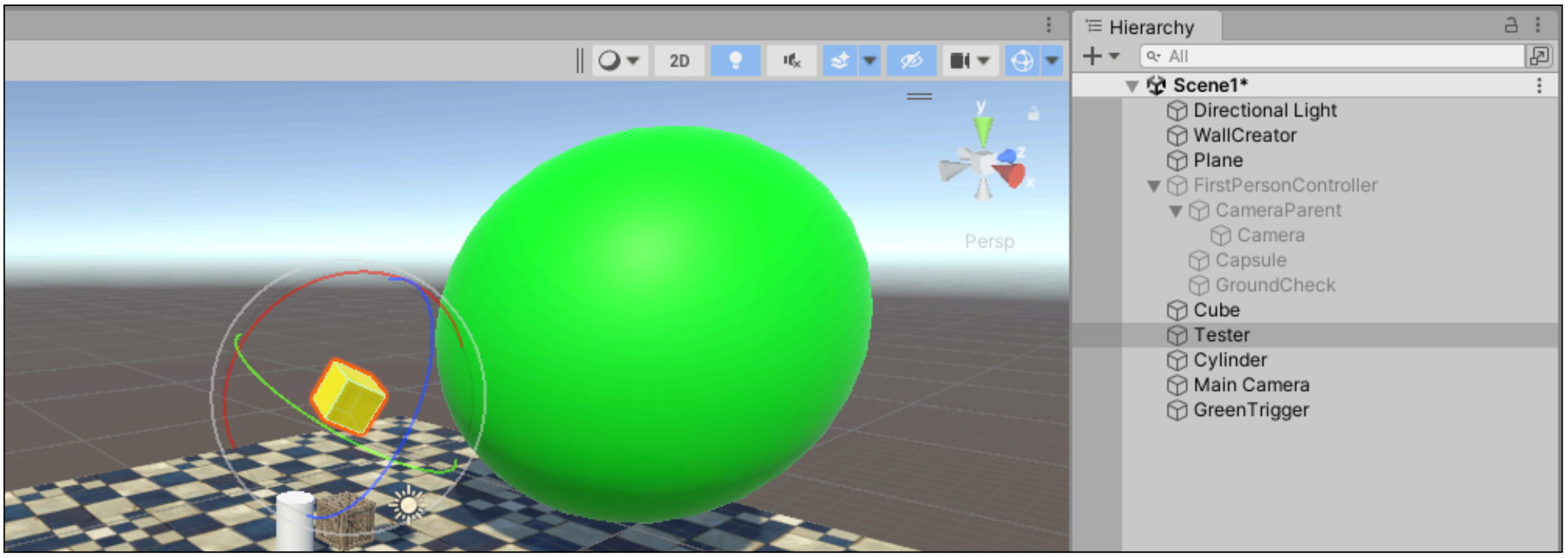
```
using UnityEngine;

public class PhysicsTester : MonoBehaviour
{
    private Rigidbody rigidbody;

    private void Start()
    {
        rigidbody = GetComponent<Rigidbody>(); // GetComponent тяжелая операция, оптимизация - кэширование объекта в переменной
    }

    private void FixedUpdate()
    {
        rigidbody.AddForce(0.5f, 0, 0, ForceMode.Acceleration); // координаты - куда будет направлена сила
        rigidbody.AddRelativeTorque(0.5f, 0.5f, 0);             // ForceMode - характер воздействия:

    }
}
```

Координаты, задающие направление движения работают по разному в зависимости от использованного метода.

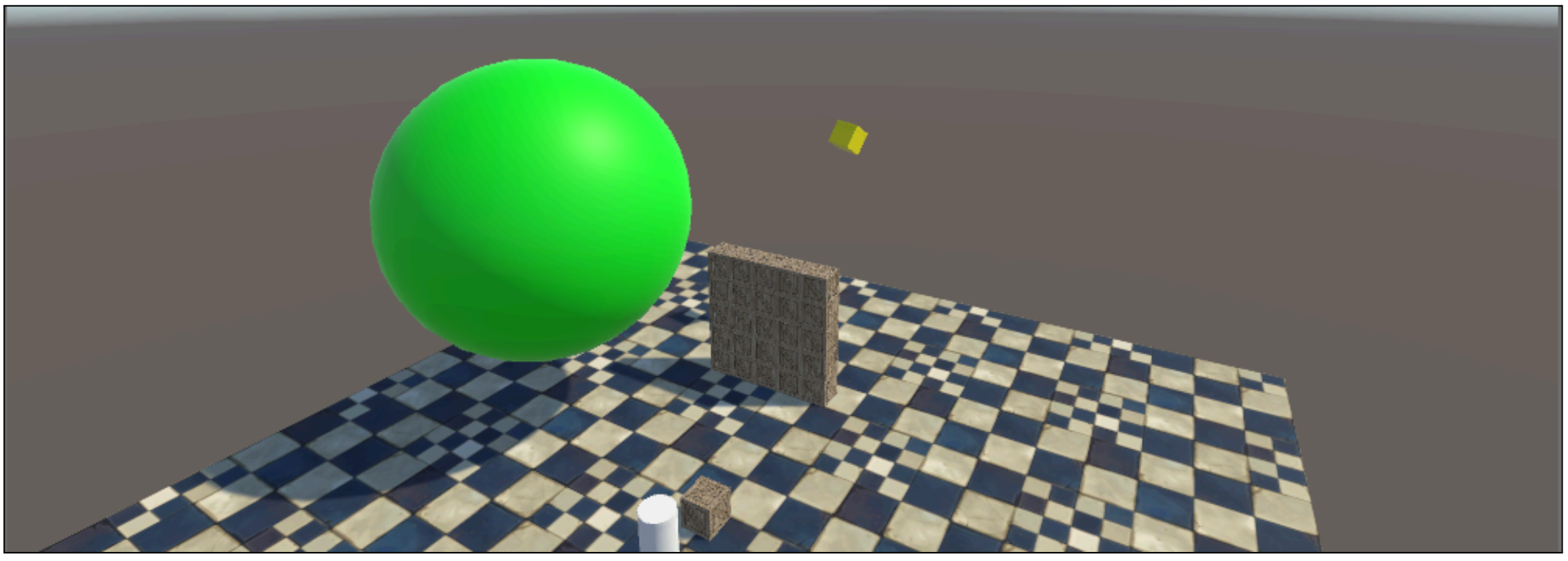
**AddForce()** прикладывает силу в глобальных координатах, объект будет двигаться по глобальным осям координат.
**AddRelativeForce()** прикладывает силу в локальных координатах, объект будет двигаться по своим осям, учитывая свои углы поворота.
**AddTorque()** задает вращение в глобальных координатах.
**AddRelativeTorque()** задает вращение в локальных координатах.

Для оптимизации надо кэшировать ссылки на объекты, это удобно делать в методах **Start()** и **Awake()**.

Новый объект для опытов - куб **Tester**. Он должен обладать компонентом **Rigidbody, но с отключенной гравитацией,** и скриптом выше:




Куб улетает со сцены, вращаясь в полете:



## ※ Постоянные силы

Постоянное физическое воздействие на объект можно получить добавлением компонента [**Constant Force**], в котором:

**Force** - вектор силы в глобальных координатах.

**Relative Force** - вектор силы в локальных координатах.

**Torque** - вектор крутящего момента в глобальных координатах.

**Relative Torque** - вектор крутящего момента в локальных координатах.

## ※ Физический материал

Физический материал частично определяет взаимодействие объектов с физическими телами. Он используется для настройки трения и упругости (отталкивания) при столкновениях. Физический материал задается в компоненте коллайдер [**Collider**].

**Dynamic Friction** - трение во время движения, когда объект двигается - замедлить объекты при столкновении друг с другом.

**Static Friction** - трение в состоянии покоя, когда объект неподвижен - не дать объекту двигаться без воздействия физической силы.

**Bounciness** - упругость.

**Friction Combine** - как комбинируется между собой трение двух объектов.

- **Average** - значения двух трений усредняются
- **Minimum** - из двух значений берется меньшее
- **Maximum** - из двух значений берет максимальное
- **Multiply** - значения двух трений умножаются

**Bounciness Combine** - как комбинируется упругость двух сталкивающихся объектов.

При контакте двух объектов к каждому из них отталкивание и трение применяется индивидуально в зависимости от их параметров. Движок Unity настроен на производительность, поэтому симуляция физики происходит только приблизительно.

## Сочленения

Объекты можно соединять между собой специальной связью, придающей соединению определенные свойства. Для установления такой связи оба соединяемых объекта должны обладать компонентом физическое тело (**Rigidbody**).

Компонент [**Character Joint**] в основном используются для эффектов **тряпичной куклы** (**Ragdoll**). Это шароподобное соединение, которое позволяет ограничивать движение по каждой оси.

Компонент [**Fixed Joint**] (неподвижное соединение) ограничивает движение объекта, связывая его с другим объектом. Это похоже на определение дочернего объекта по отношению к другому - родительскому, но реализовано с помощью физики, а не иерархии компонентов **Transform**. Используется если надо соединить или разъединить два объекта без необходимости изменения иерархии. Когда надо двигать объекты вместе (временно или постоянно), этот компонент упрощает реализацию, но надо добавлять компонент **Rigidbody**.

Компонент [**Configurable joint**] - включают в себя все возможности других видов соединений, поэтому его настройки самые обширные. С включенной опцией **Configured In World Space** движения будут ограничены осями мира, а не локальными осями объекта.

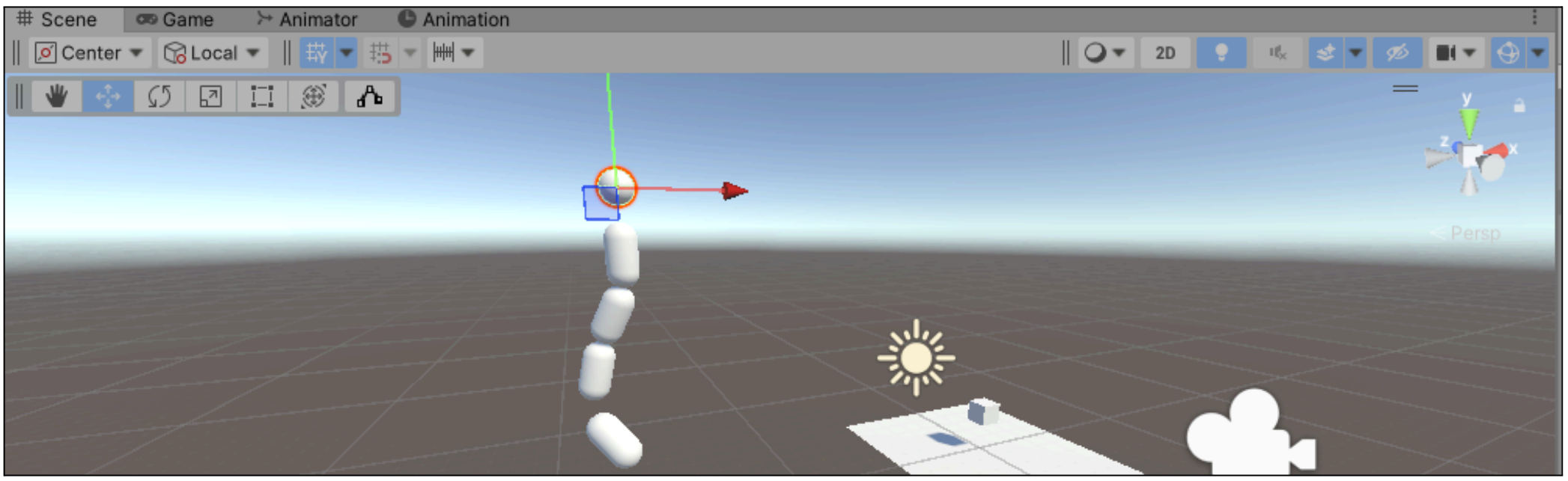
Компонент [**Hinge Joint**] - имитирует соединение с помощью дверных петель.

Компонент [**Spring Joint**] - имитирует соединение пружиной.

Для наглядной демонстрации можно собрать в тестовой сцене колбаску из элементов:

- сфера, с физическим телом, и флагом Is Kinematic;
- капсула 1, с физическим телом, которая крепится через соединение к **Fixed Joint** к сфере;
- капсула 2, с физическим телом, которая крепится через соединение к **Character Joint** к капсуле 1;
- капсула 3, с физическим телом, которая крепится через соединение к **Hinge Joint** к капсуле 2;
- капсула 4, с физическим телом, которая крепится через соединение к **Spring Joint** к капсуле 3.

Запустить **Play**, переключиться на окно сцены и подвигать сферу в пространстве, наблюдая как двигаются прикрепленные к ней капсулы:



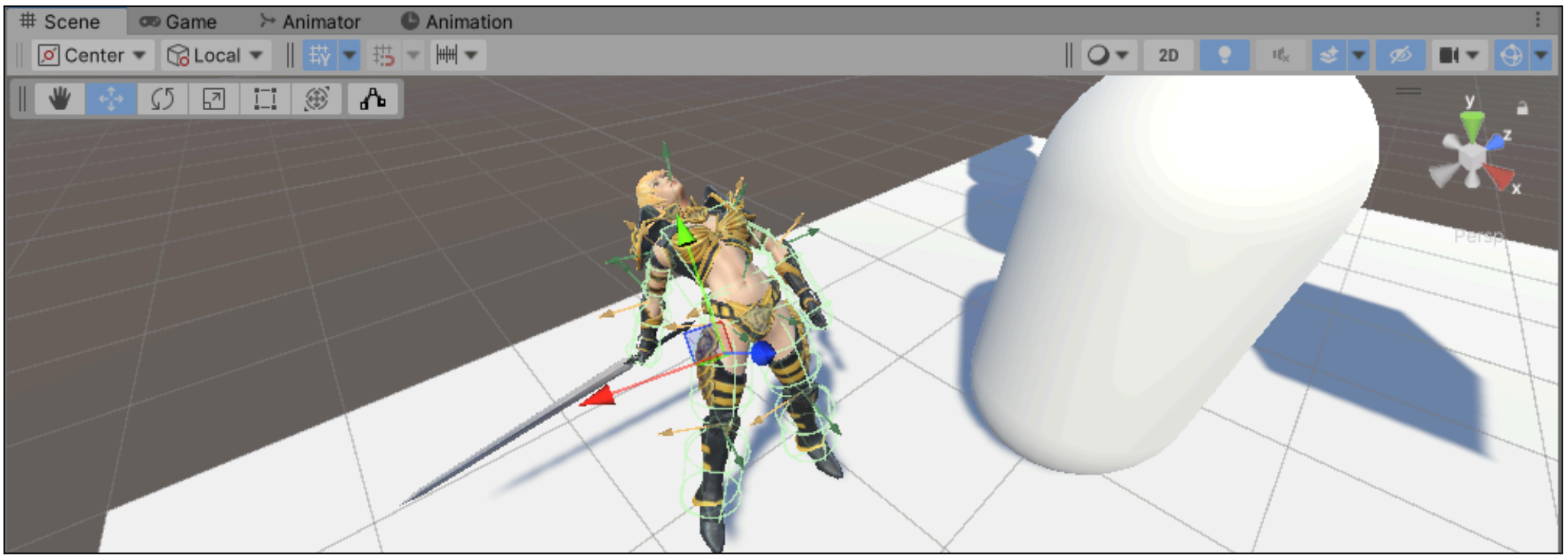

## ※ Тряпичная кукла

Команда **Ragdoll** вызывает мастер установки сочленений между компонентами. Итоговая композиция на основе физических взаимодействий может заменить анимации объектов или частично осуществить взаимодействие с физикой мира. Тряпичная кукла использует меши с привязкой к костям (**skinned meshes**), т.е. меш персонажа, оснащенный костями в программе 3D моделирования.

**Ragdoll** вызывается в окне иерархии объектов - т.е. в рамках сцены, и работает только с объектами, размещенными на сцене. Поэтому перед вызовом мастера создания куклы надо добавить на сцену нужную модель НПС и распаковать ее в объект. Затем в окне иерархии объектов развернуть древовидную структуру, чтобы можно было выделять части для удобства дальнейшего сопоставления.

При создании объекта **Ragdoll** надо установить ссылки на участки модели персонажа, которые будут соединены в соответствии с их положением в теле. Когда все ссылки и поля будут заполнены - завершить создание [**Create**].

Если все соответствия частей тела будут установлены верно, то модель НПС будет вести себя как тряпичная кукла:



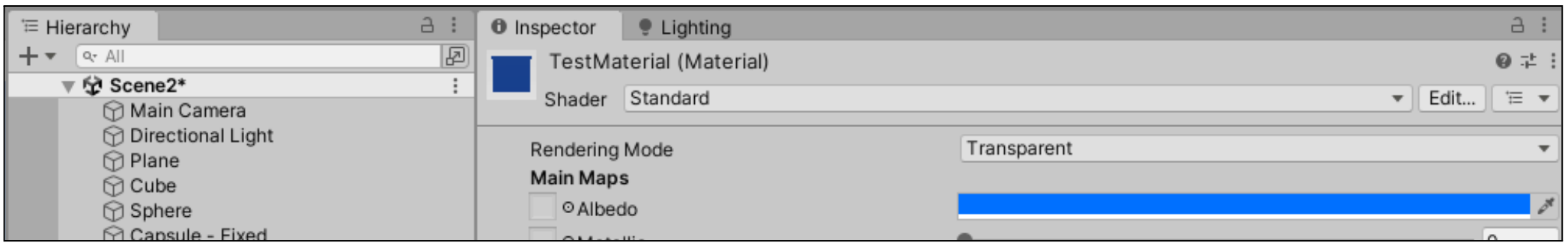

Если результат положительный, следует сохранить объект как префаб / вариант префаба.
Тряпичная кукла может быть использована для анимации убитых НПС - когда они падают на землю в нелепых позах, и т.п.

## Корутины

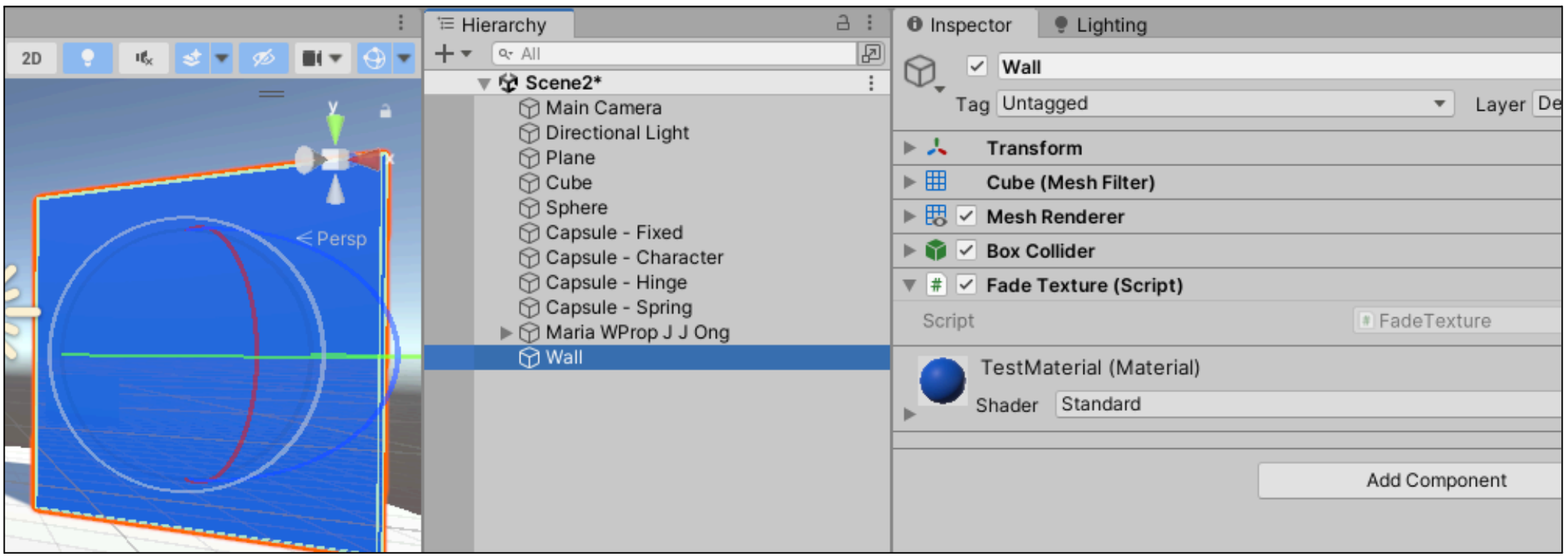
Функции должны выполняться до конца, чтобы вернуть результат. Вызов функций происходит в каждом кадре, и в течении кадра функция должна выполниться до конца. Но некоторые процессы продолжительны по времени, поэтому обычные функции к ним неприменимы (звук, процедурная анимация, визуальные эффекты). В этих случаях используются сопрограммы - корутины (**Coroutines**).

## ※ Изменение материала

В качестве примера - изменение прозрачности объекта. Создан новый материал, которому установлен режим рендеринга **Transparent**:




На сцене создан куб **Wall**, которому придана форма стены, назначен новый материал и новый скрипт **FadeTexture**:



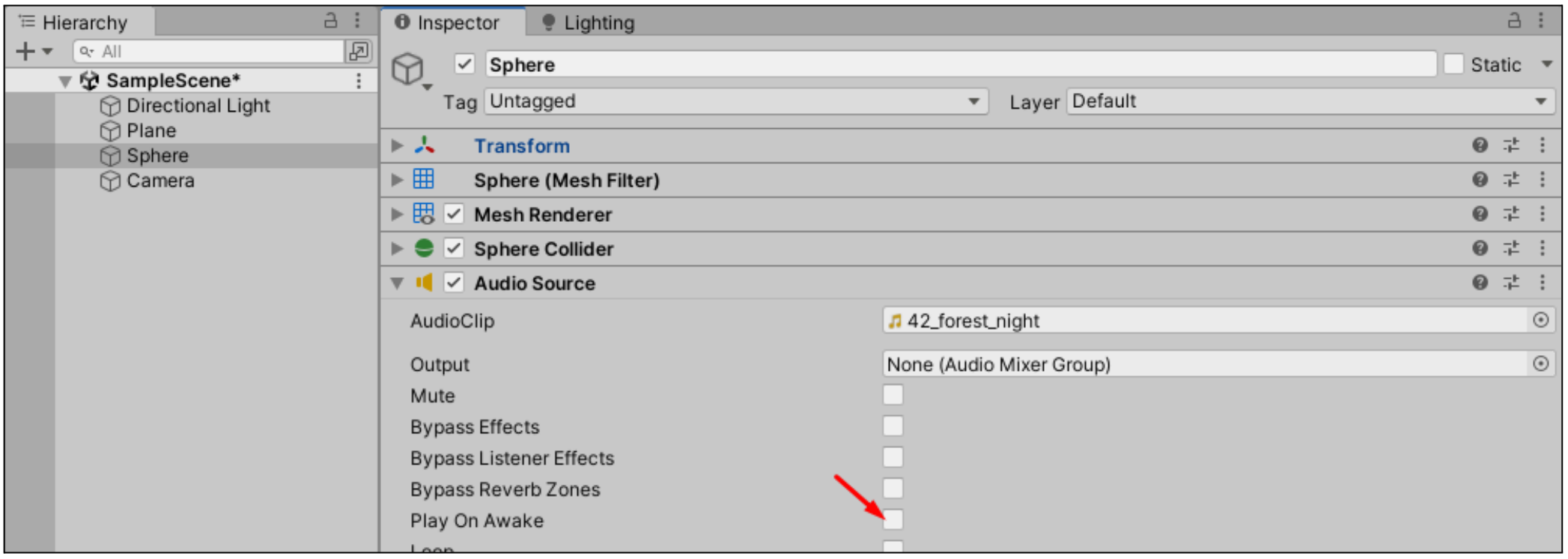

Код скрипта будет реагировать на нажатие кнопки [**F**] во время **Play** мода, и постепенно делать куб почти полностью прозрачным.

```
using UnityEngine;

public class FadeTexture : MonoBehaviour
{
    private Material material;
    private Color color;

    private void Start()
    {
        material = gameObject.GetComponent<Renderer>().material;
        color = material.color;
    }

    private void Update()
    {
        if(Input.GetKeyDown("f")) // считывает нажатие кнопки 1 раз, но не ее удержание
            StartCoroutine(Fade());
            // StartCoroutine("Fade"); // так тоже работает
    }

    private IEnumerator Fade()
    {
        var wait = new WaitForSeconds(0.2f);

        for (float f = 1f; f >= 0; f -= 0.05f)
        {
            color.a = f;
            material.color = color;

            yield return wait;
        }
    }
}
```

## ※ Проигрывание звука

В примере рассмотрено управление воспроизведением звука через команды в скрипте.

Создать тестовую сцену, создать объект с компонентом **AudioSource**. Назначить звуковой файл и выключить флаг **Play On Awake**.




Создать скрипт с кодом и добавить к объекту. При запуске на экране будет отрисованы две кнопки - поставить на паузу и продолжить:

```
using UnityEngine;

public class PlaySound : MonoBehaviour
{
    private AudioSource audioData;

    private void Start()
    {
        audioData = GetComponent<AudioSource>();
        audioData.Play();
    }

    private void OnGUI()
    {
        if (GUI.Button(new Rect(100, 70, 150, 30), "Pause")) // 100 отступ слева, 70 отступ сверху, 150 ширина, 30 высота кнопки
            audioData.Pause();

        if (GUI.Button(new Rect(100, 170, 150, 30), "Continue"))
            audioData.UnPause();
    }
}
```

Предыдущий скрипт отключить. Создать новый скрипт и добавить какому-либо объекту на сцене (например, к пустому объекту, в который будут вложены новые звуки). Объекту с новым скриптом добавить новые звуки. Скрипт будет проигрывать случайный клип из списка.

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

public class MusicMaster : MonoBehaviour
{
    public List<AudioClip> audioClips;

    private void Start()
    {
        StartCoroutine(PlayRandomClip());
    }

    private IEnumerator PlayRandomClip()
    {
        while (true)
        {
            int randomIndex = Random.Range(0, audioClips.Count);
            float duration = audioClips[randomIndex].length;
            float wait = duration + Random.Range(20, 60); // вычислить время следующего запуска клипа

            AudioSource.PlayClipAtPoint(audioClips[randomIndex], transform.position); // играть звуковой клип

            yield return new WaitForSeconds(wait); // следующий клип будет проигрываться после паузы в 20-60 сек
        }
    }
}
```

## Анимация

Анимация строится из клипов. **Анимационные клипы** собираются в общую структуру - **анимационный контроллер** [**Animator Controller**]. Он отслеживает какая анимация должна проигрываться в данный момент, когда анимации должны меняться или смешиваться.

При открытии анимационного контроллера отобразится схема **состояний**. **Состояния модели** отображаются как **узлы**, соединенные линиями.

Модель персонажа имеет особуют конфигурацию **костей** - **скелет** (**Rig**), которые собираются вместе в структуру внутри **аватара** [**Avatar**].

Для назначения анимации модели персонажа к персонажу добавляется **компонент** [**Animator**]. Аниматору назначается [**Animator Controller**] и [**Avatar**], ссылка на аватар нужна только для анимации человекоподобных существ. Опция **Apply Root Motion** - анимация объекта становится связанной с размерами и положением в пространстве (обычно не надо включать для анимации объектов).

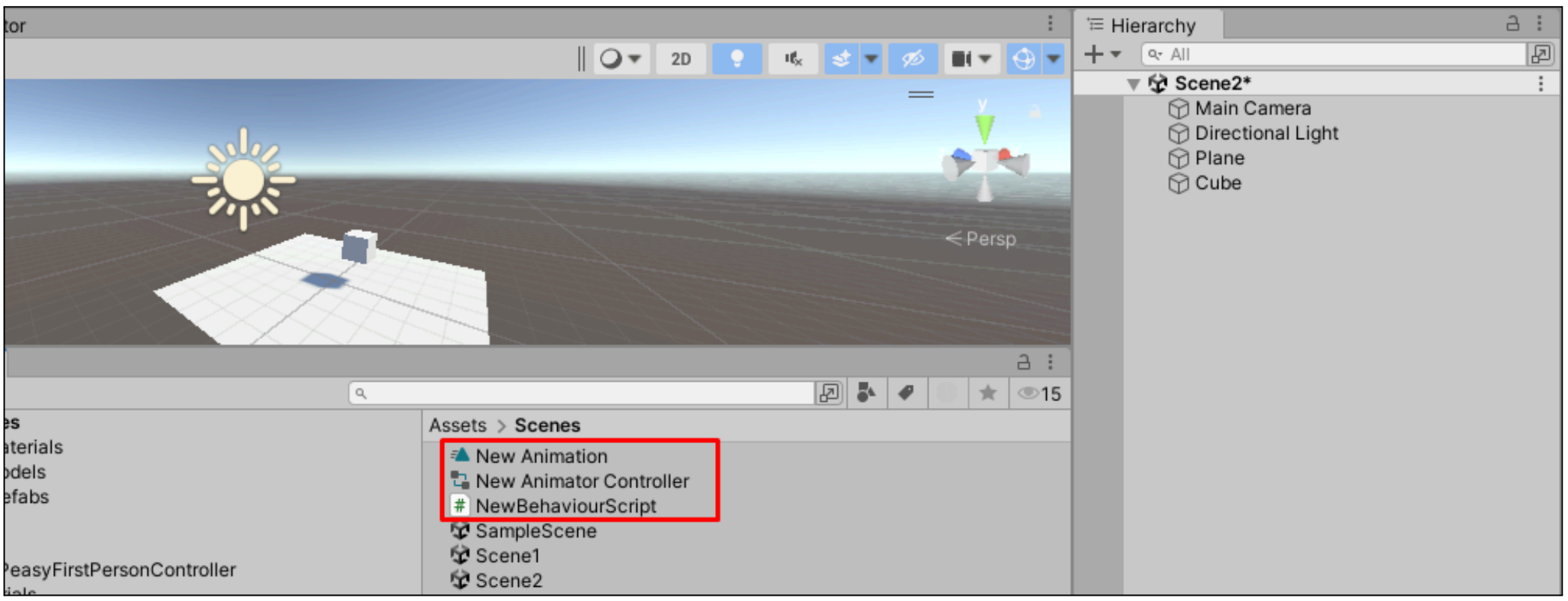
## ※ Анимация объектов

Создать новую сцену, создать на сцене объект **Cube** (Cube).

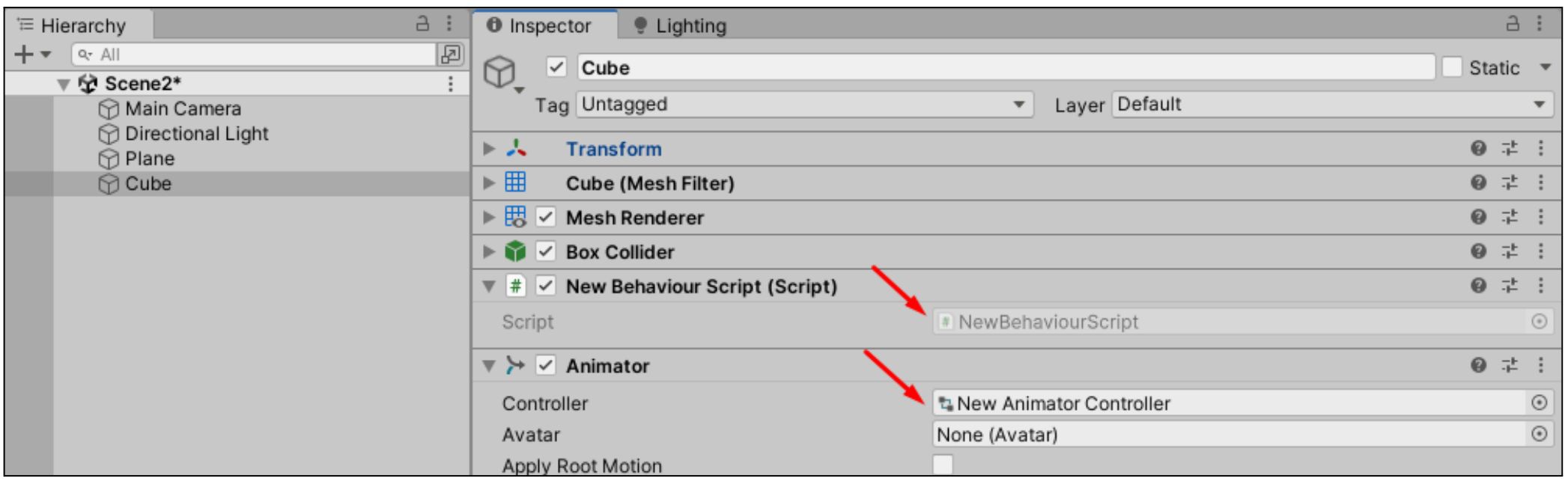
Создать **Animator Controller** (New Animator Controller).

Создать **Animation** (New Animation).

Создать **Script** (NewBehaviourScript).




Назначить кубу созданный скрипт, и добавить компонент [**Animator**], и в нем назначить контроллер **New Animator Controller**.



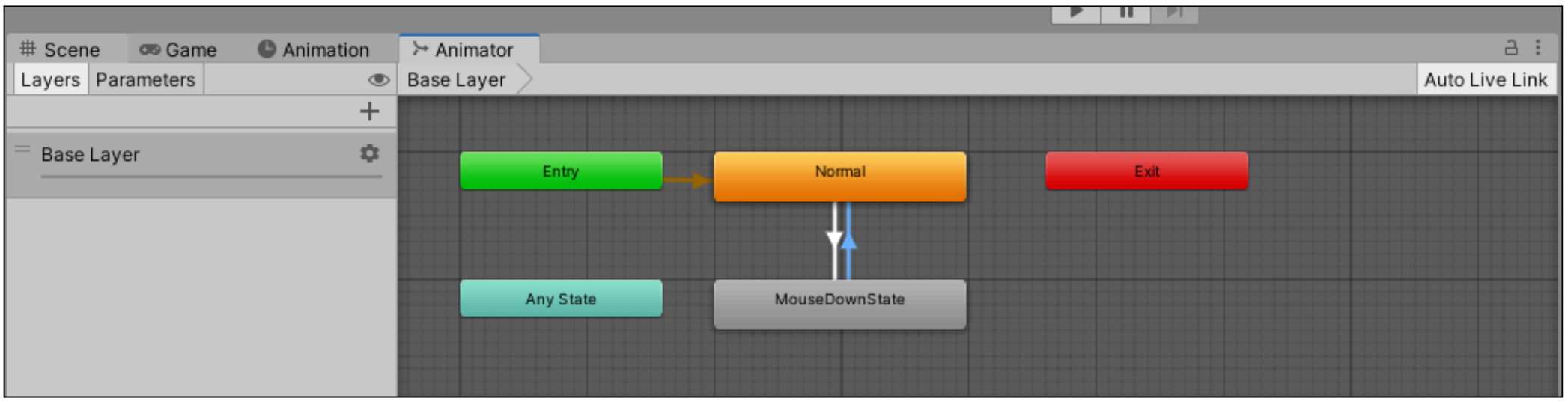

Открыть **New Animator Controller** - развернется **окно** [**Animator**]. Это окно можно открыть только в одном экземпляре.

Создать новое состояние (**Create State** - **Empty**), переименовать его в **Normal**. Связь от элемента **Entry** будет создана автоматически.

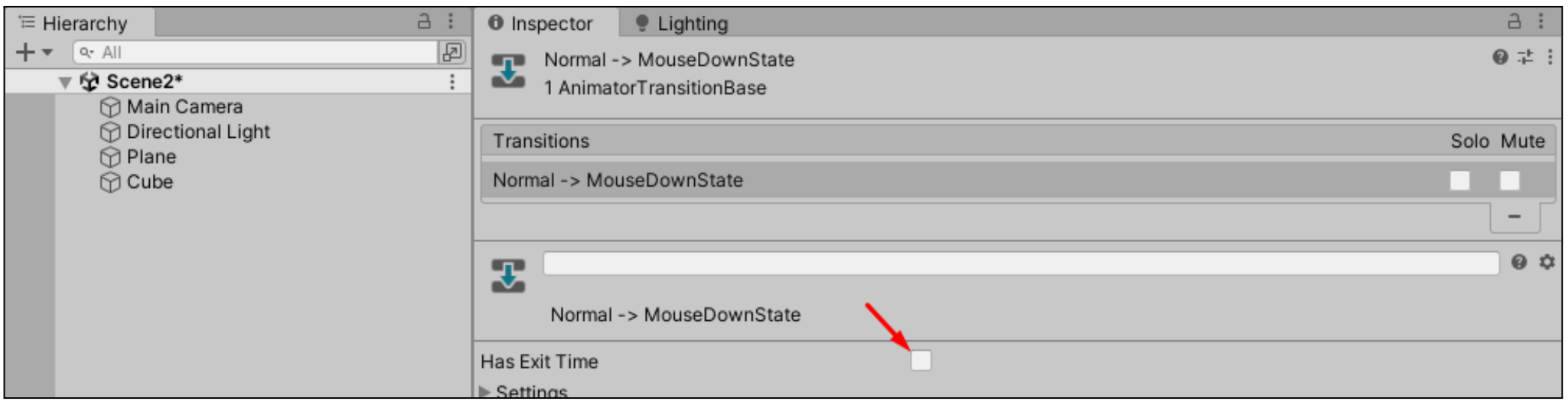
Создать новое состояние (**Create State** - **Empty**), переименовать его в **MouseDownState**.

От **Normal** через контекстное меню создать связь с **MouseDownState** - выбрать **Make Transition** и указать на элемент.

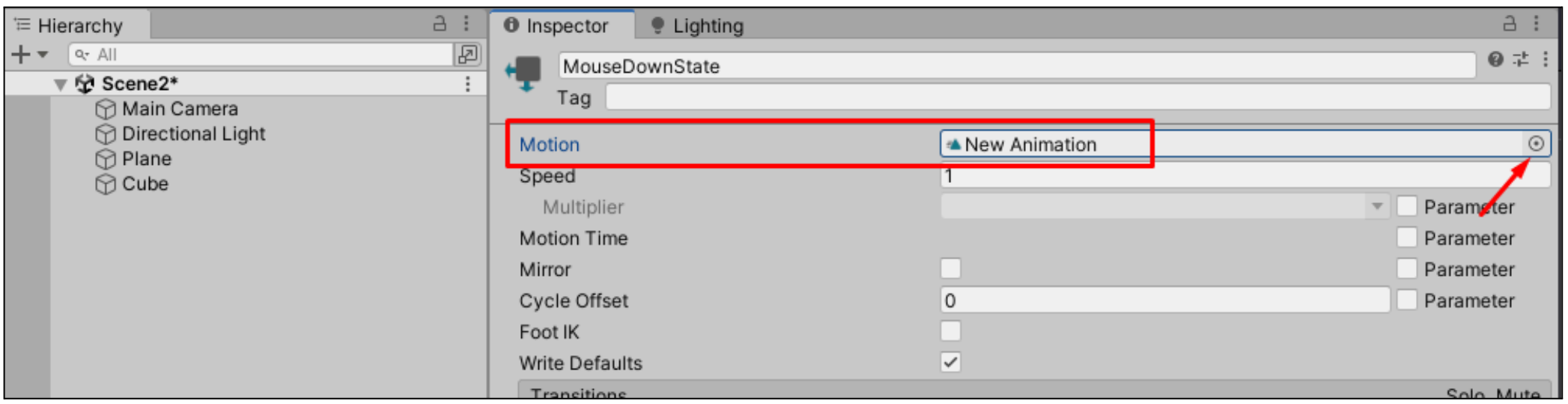
От **MouseDownState** через контекстное меню создать связь с **Normal** - выбрать **Make Transition** и указать на элемент.



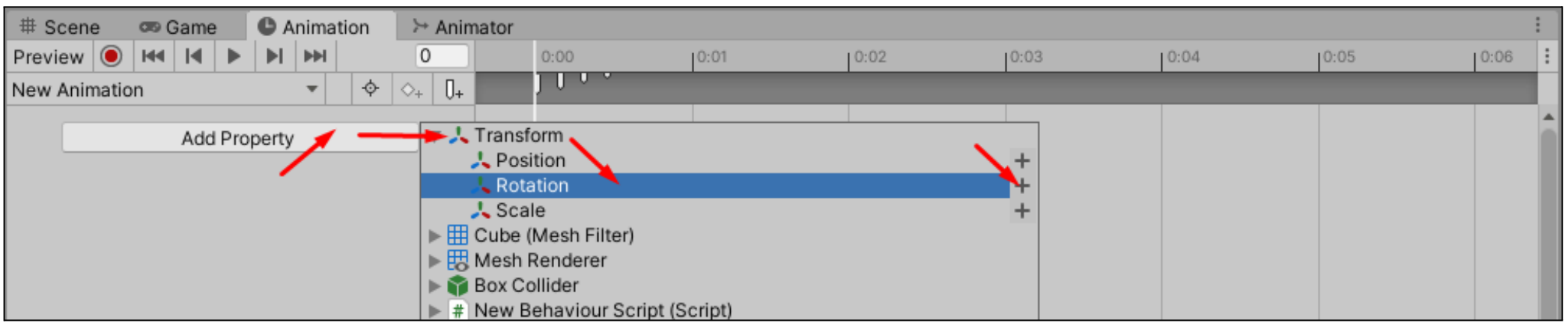
Выбрать связь от **Normal** к **MouseDownState** и снять флаг **Has Exit Time** (не прерывать предыдущую анимацию). Этот флаг надо убирать с анимации ходьбы, покоя и прочих подобных, чтобы их можно было прерывать в любой момент, а с цепочками связанных анимаций наоборот.



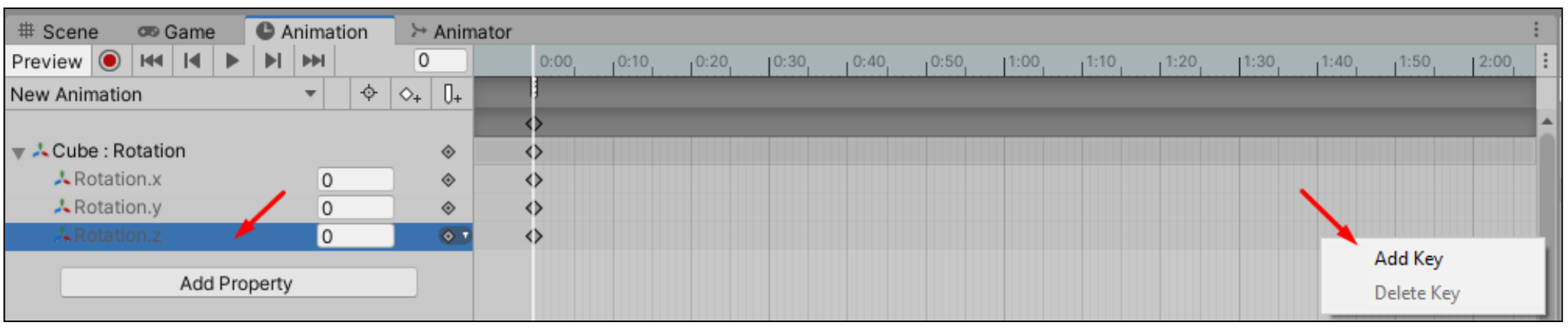

Выбрать **MouseDownState** и назначить в **Motion** созданную анимацию **New Animation**:



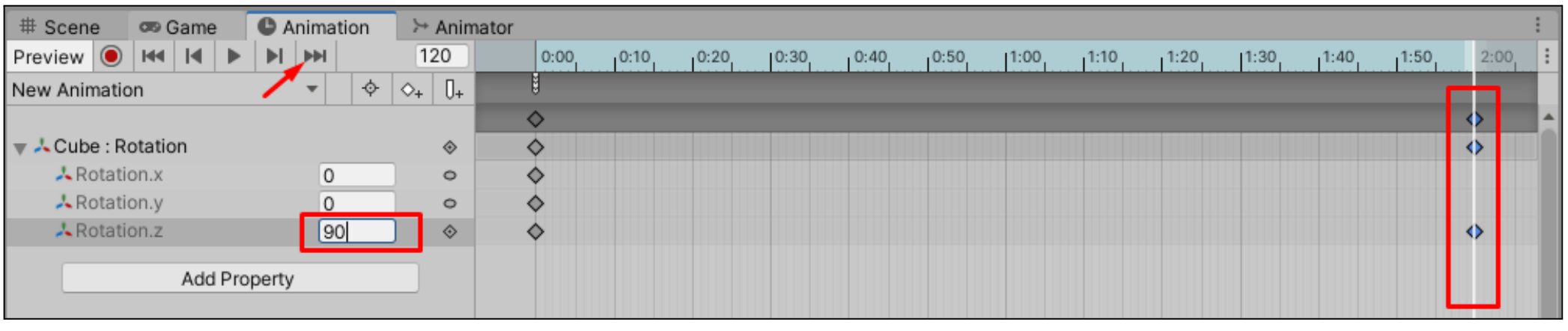

Выбрать куб на сцене, **Ctrl + 6** - откроется **окно** [**Animation**]. Добавить свойство - [**Add Property**] / **Transform** / **Rotation** / [**+**].




Выбрать ось Z и на таймлайне через контекстное меню создать точку - **Add Key**:




Передвинуть позицию настройки (белая линия) стрелкой перемотки к созданной точке и выставить в ней поворот по **Z** на 90 градусов:



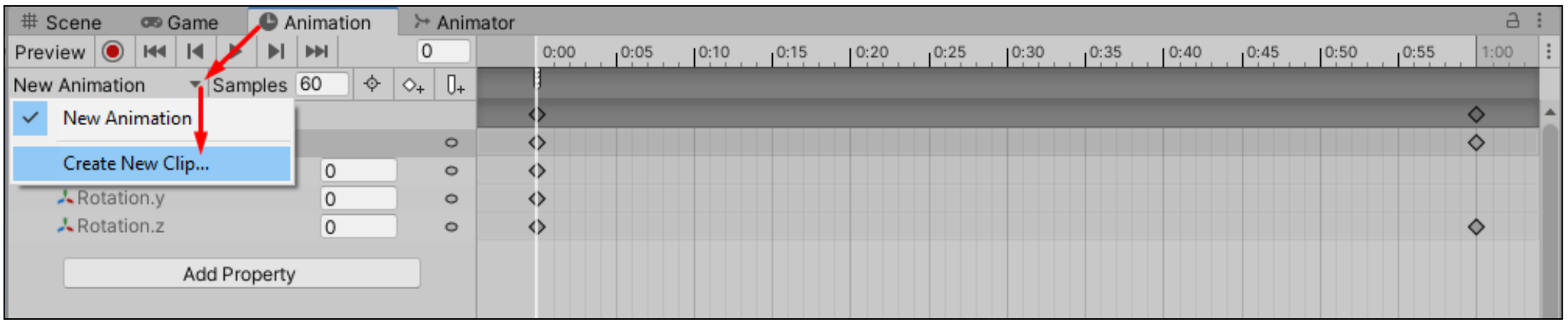

Внести изменения в созданный скрипт **NewBehaviourScript**:

```
using UnityEngine;
```

```
public class NewBehaviourScript : MonoBehaviour
{
    private void Update()
    {
        if (Input.GetMouseButtonDown(0)) // определить нажата ли левая кнопка мышки
        {
            var ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit raycastHit;

            if (Physics.Raycast(ray, out raycastHit, 100))
            {
                var colliderHit = raycastHit.collider; // получить коллайдер, по которому кликнули
                var gameObjectHit = colliderHit.gameObject; // получить объект игры, к которому привязан коллайдер
                var animator = gameObjectHit.GetComponent<Animator>(); // получить аниматор объекта

                animator.Play("MouseDownState"); // воспроизвести анимацию по указанному названию
            }
        }
    }
}
```

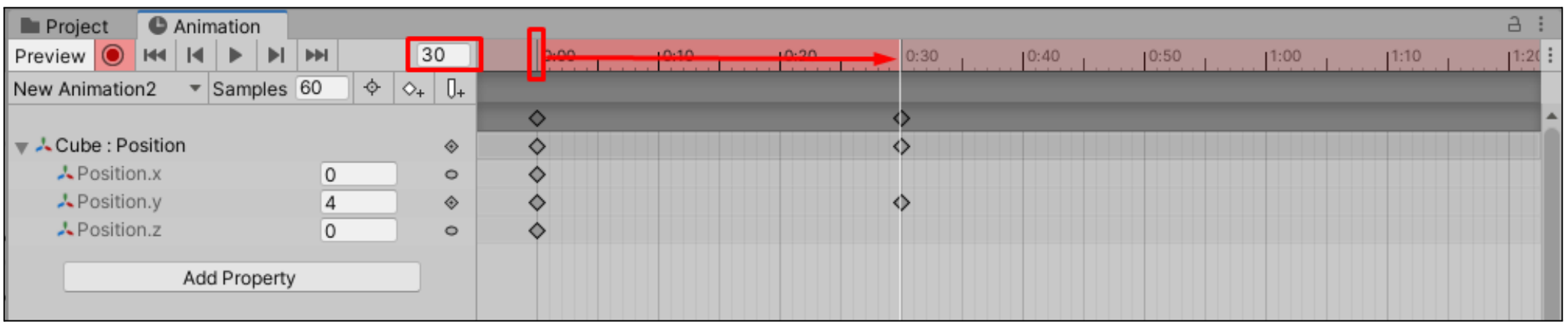
Проделанных изменений достаточно чтобы куб на сцене поворачивался при клику по нему. Так как анимация возврата в исходное положение не добавлена, то после завершения поворота куб автоматически вернется в исходное положение рывком - в состояние **Normal**.

Альтернативный вариант создания анимации - **запись изменений объекта в окне сцены**. Выбрать куб, в окне анимации создать новый клип:

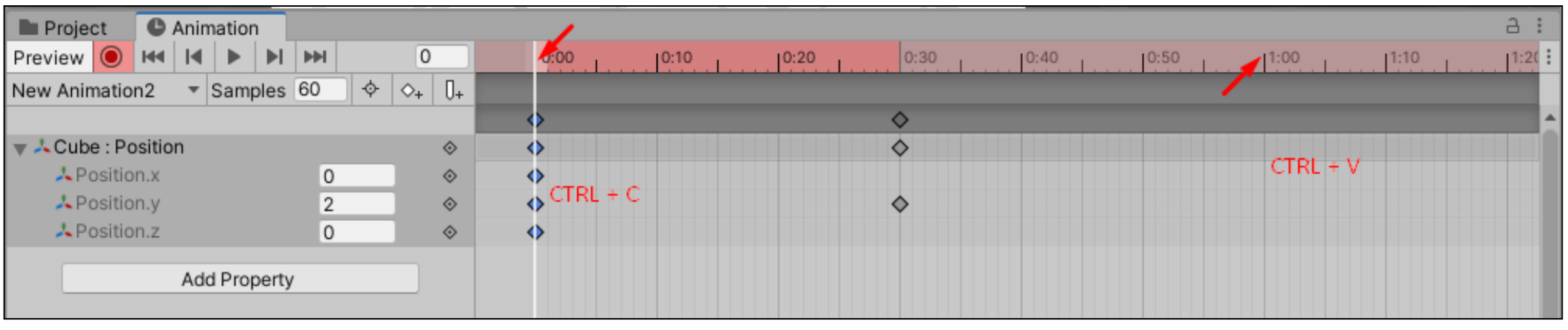



В этом примере создан файл **New Animation2.anim**. Далее включить запись изменений по кнопке, и выбрать стартовую позицию куба:


В анимацию будут записано свойство со стартовыми координатами. Переместить позицию текущего кадра (белая линия) на таймлайне (или указать номер кадра). Перейти окно сцены и переместить куб на новое место, чтобы в свойства записались новые координаты объекта:

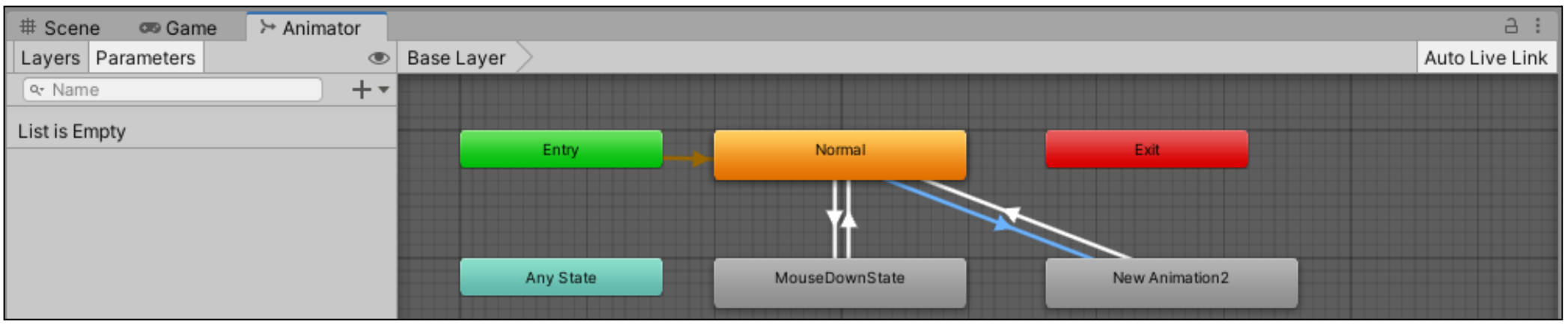



Если выбрать позицию кадра с маркерами в виде ромба, то можно будет скопировать координаты, и вставить их на таймлайне далее:




Остановить запись анимации. При проигрывании анимации будет видно, как куб из стартовой точки перемещается в конечную и назад. Новый анимационный клип при создании был автоматически подключен к новому созданному состоянию.

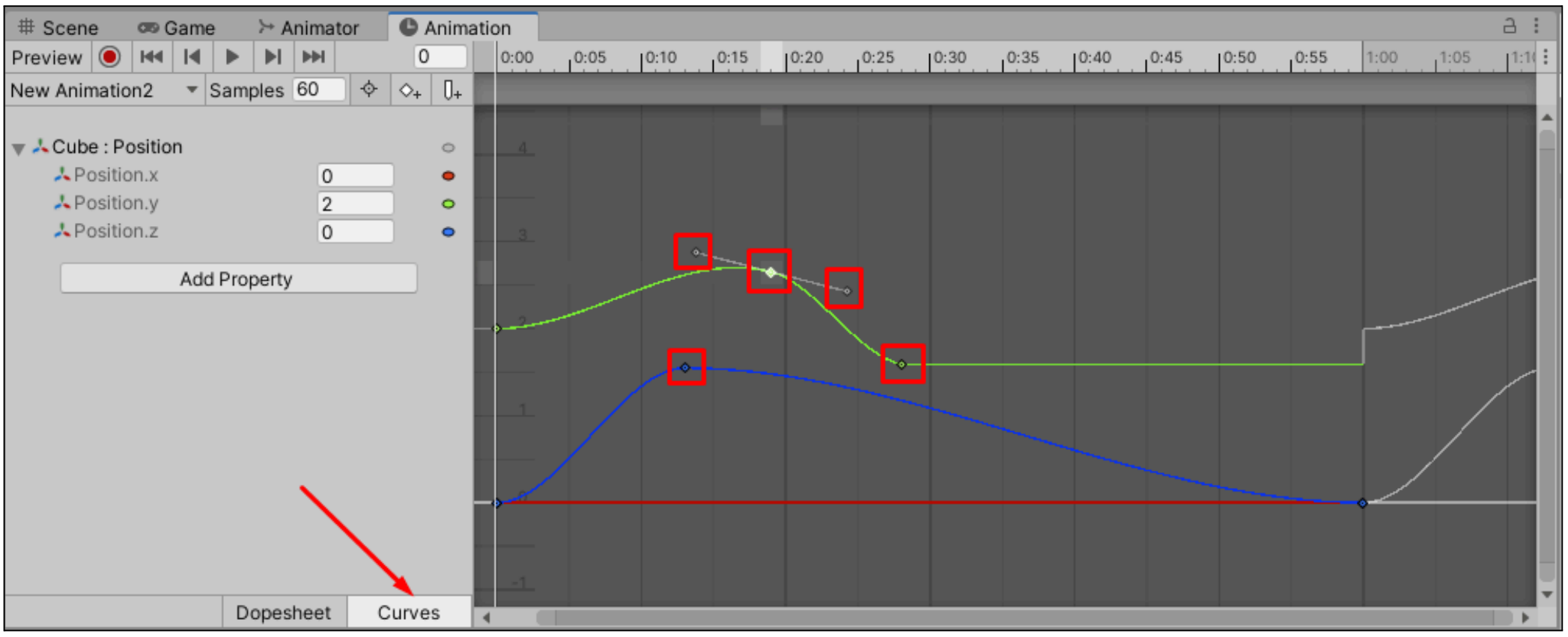
В окне просмотра состояний [**Animator**] надо создать связи между новым состоянием и состоянием **Normal**.
С обеих новых связей не снимать флаг **Has Exit Time**, анимация будет проигрываться в цикле, без внешней активации и без остановки:




Однако при клике по кубу он все еще будет возвращаться в исходную позицию и запускать поворот, так как анимационный клип сохранил в себе работу с глобальными координатами, а не с локальными.

В окне [**Animation**] можно переключиться на вкладку [**Curves**]. Добавляя в таймлайн новые позиции для ключевых кадров (**Add Key**) или удаляя существующие, регулируя значения и плавность кривых можно более детально управлять анимацией движений.

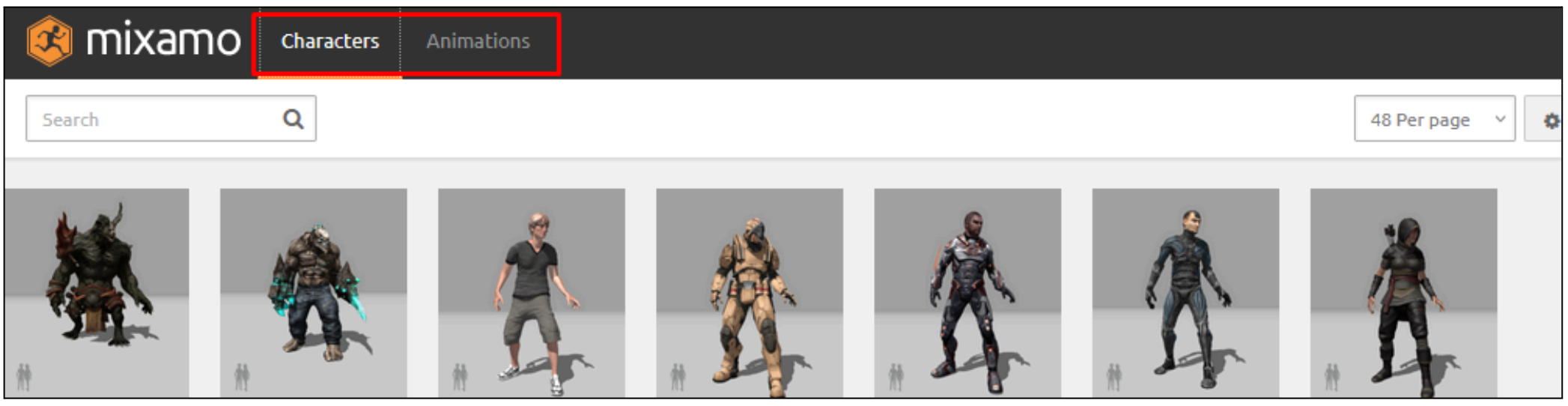
Так можно менять не только координаты, поворот и размер объекта, но и значения других параметров - цвет материала и т.д. Анимация персонажей устроена аналогично, только там анимация работает с движениями (координатами) костей аватара.



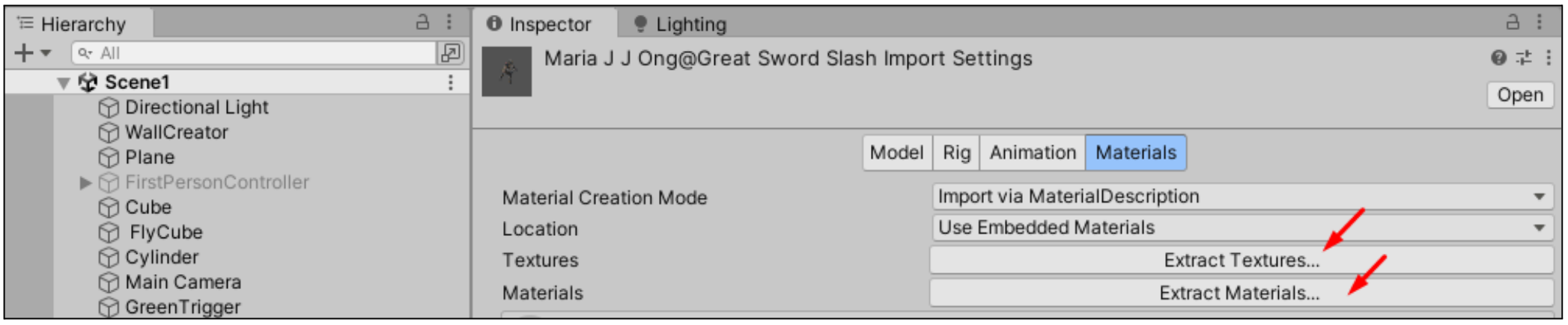

> Импортированные анимации нельзя редактировать, только смотреть / читать.

## ※ Импорт анимации

Готовую анимацию можно загрузить из магазина ассетов и других сайтов. Пример - https://www.mixamo.com - выбрать модель и анимацию:




Выгрузка анимации [**Download**] делается вместе с моделью, или без модели. **Формат - FBX для Unity**. Импорт в проект Unity как обычно - перетащить файл в окно проекта. Если модель загрузилась серой, распаковать текстуры и материалы:



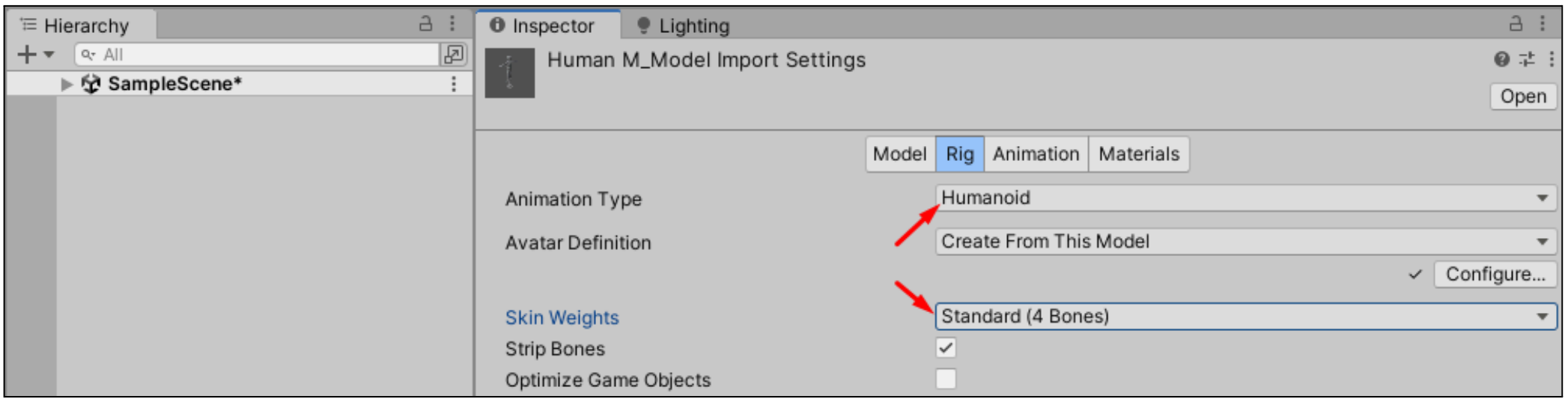
## Структура модели

Для демонстрации процесса создан новый проект, из магазина ассетов импортирован бесплатный набор - с моделями и анимациями:
https://assetstore.unity.com/packages/p/human-melee-animations-free-165785
В комплекте с ним хорошо подойдет еще один набор (в пакете общие анимации), но его лучше скачать после изучения темы этой главы.
https://assetstore.unity.com/packages/3d/animations/human-basic-motions-free-154271

Префаб модели персонажа собирается как конструктор из компонентов:
**Mesh** + **Material** отвечают за визуальную составляющую, образуя **скин**.
**Скин** связан со **скелетом** модели, для гуманоидных существ **скелет** упакован в **Аватар**.
**Скелет** собран из **костей** в иерархическую структуру. **Анимация частей тела** персонажа связана с движением **костей**.
**Кость** это условно-абстрактная сущность, кость может быть волосах, глазах, ушах, хвостах, элементах одежды и т.д.
[**Rig**] - **Тип анимации** модели и **Аватар** (нужен только для гуманоидов).
**Типы анимации:**

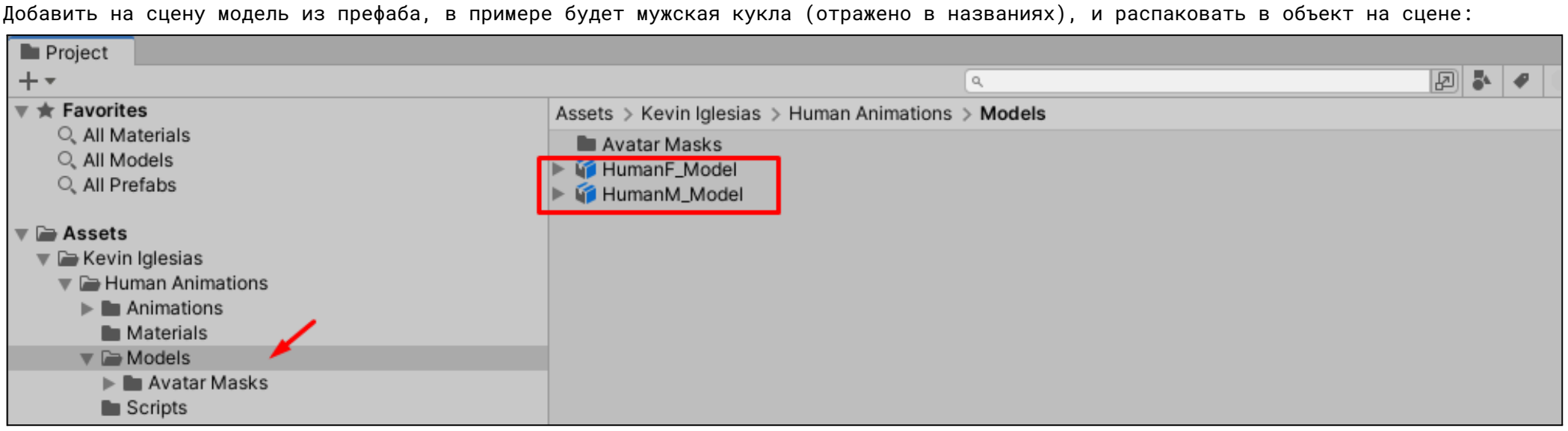
- **Legacy** - старый формат, который нужен в очень старых проектах и при экстремальной оптимизации
- **Generic** - применяется в основном к объектам и не гуманоидным существам, иногда к гуманоидным
- **Humanoid** - применяется к гуманоидным персонажам, аАнимации работают в связке с аватарами

[**Animation**] - сборник анимационных клипов (в данном примере пусто).
[**Materials**] - используемые материалы (в данном примере пусто).

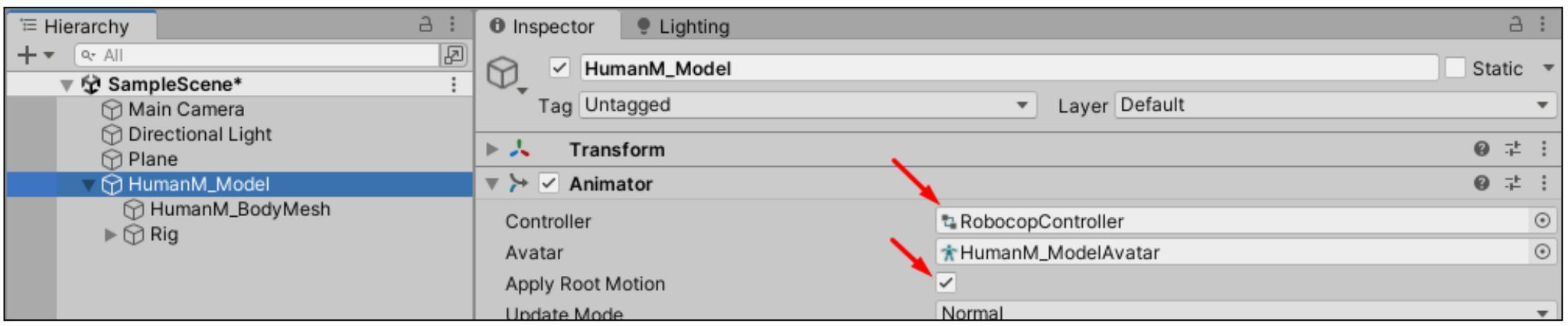



Обычно не требуется в импортированных моделях влезать в аватар и что-то в нем менять. Сами модели, со скелетом делают в программах 3D моделирования. Там же можно создавать анимации.

Добавить на сцену модель из префаба, в примере будет мужская кукла (отражено в названиях), и распаковать в объект на сцене:




У распакованного на сцене объекта есть компонент **Animator** со ссылкой на **Avatar**, но он без анимационного контроллера.
Создать новый анимационный контроллер **Animator Controller** (RobocopController) и добавить его в **Animator** модели. В примере будут использоваться анимации, способные двигать персонажа на сцене, поэтому опция **Apply Root Motion** должна быть включена:

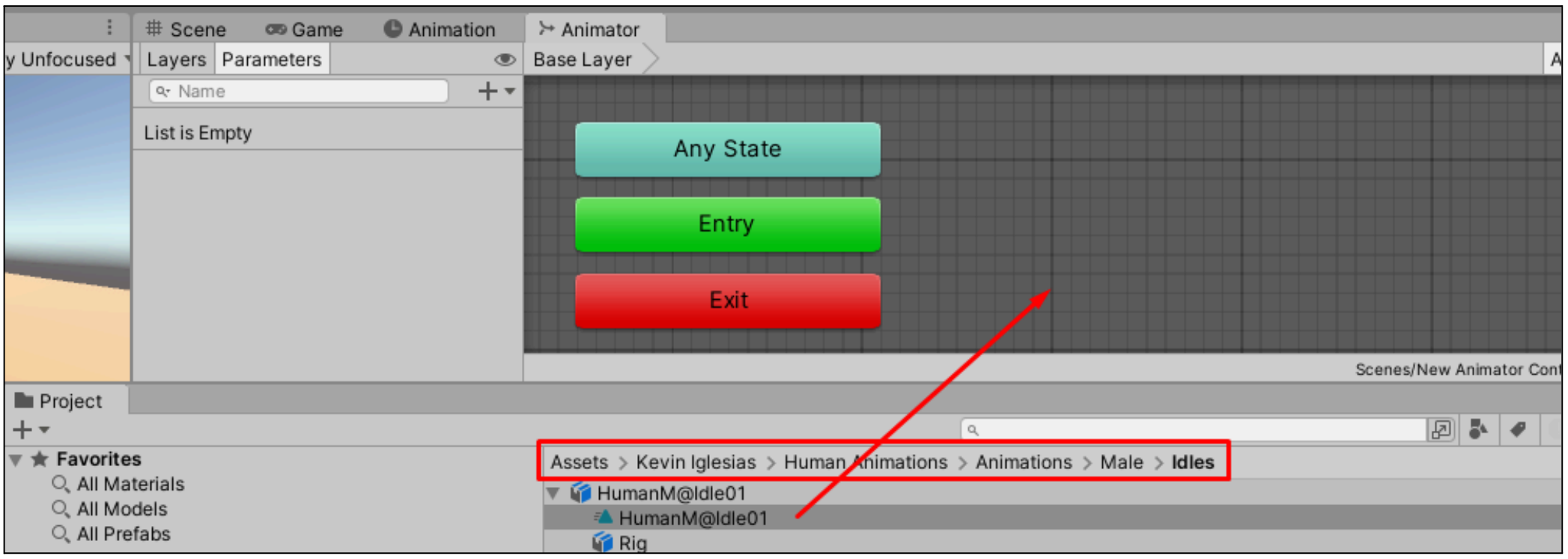


## ※ Анимационный контроллер

Анимации делят на категории, соответствующие некоторому состоянию, некоторые из них:

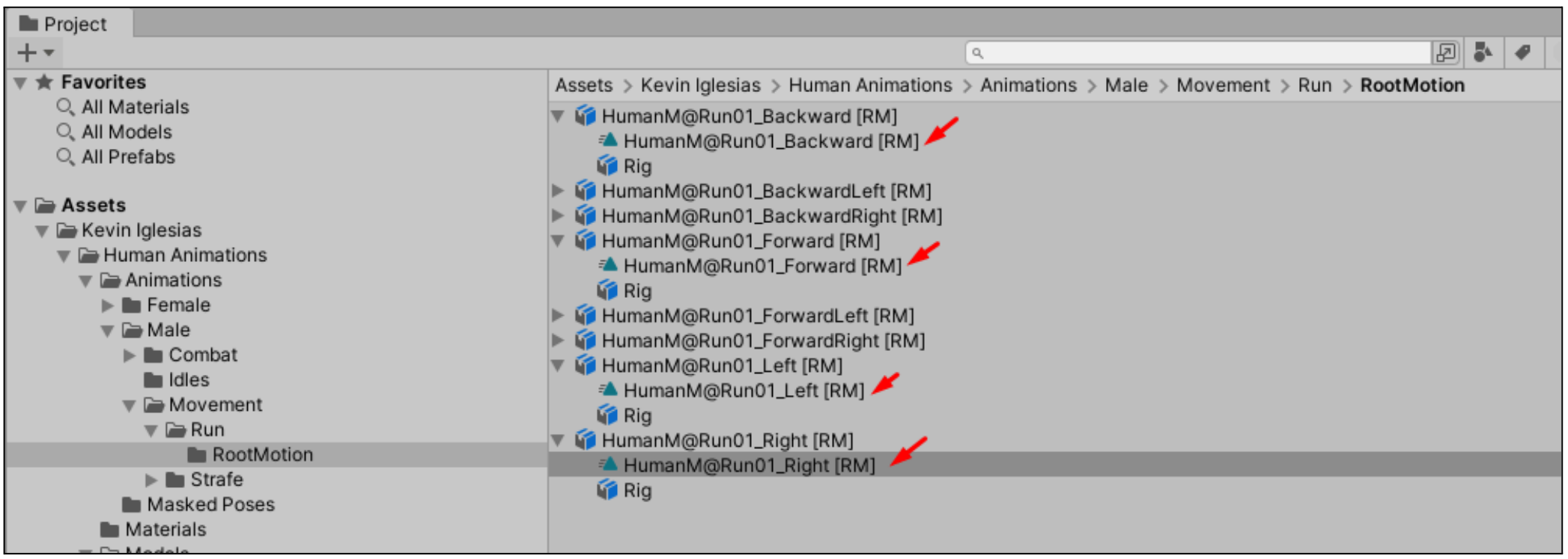
- idle - стоять
- walk - идти
- run - бежать
- jump - прыгать

Открыть **анимационный контроллер** и добавить туда из импортированного ассета анимацию ожидания - **HumanM@Idle01**:

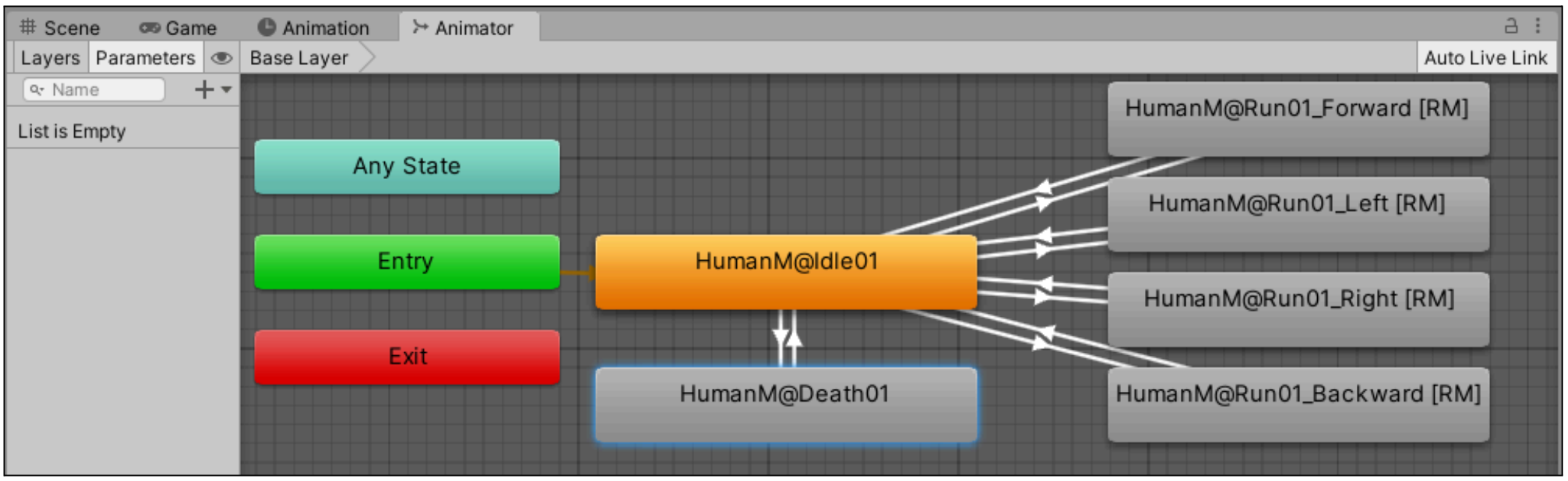



Чтобы персонаж перемещался по сцене надо добавить в схему анимации передвижения. У используемых вариантов в окне предпросмотра анимации под куклой движется сетка (окно предпросмотра в нижнем правом углу, оно может быть уменьшено в размерах). Добавить клипы:

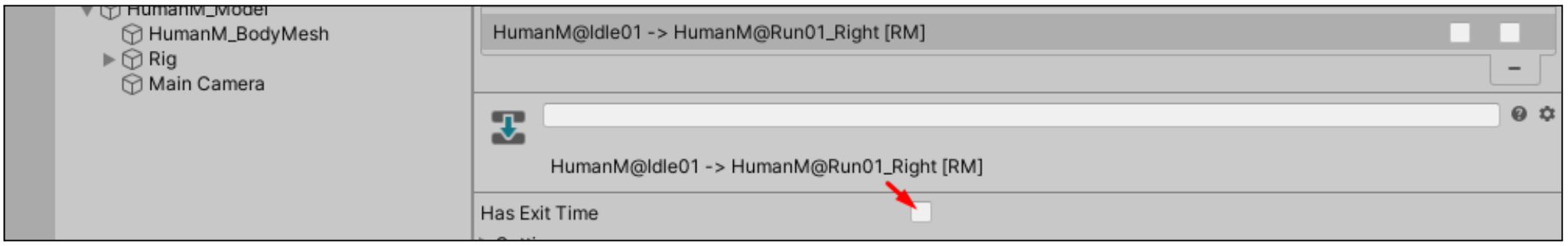
- **HumanM@Run01_Forward [RM]**
- **HumanM@Run01_Left [RM]**
- **HumanM@Run01_Right [RM]**
- **HumanM@Run01_Backward [RM]**




Анимации прыжка в этом комплекте нет (она во втором), добавочно взята анимация **HumanM@Death01**. Построены связи с состоянием покоя:

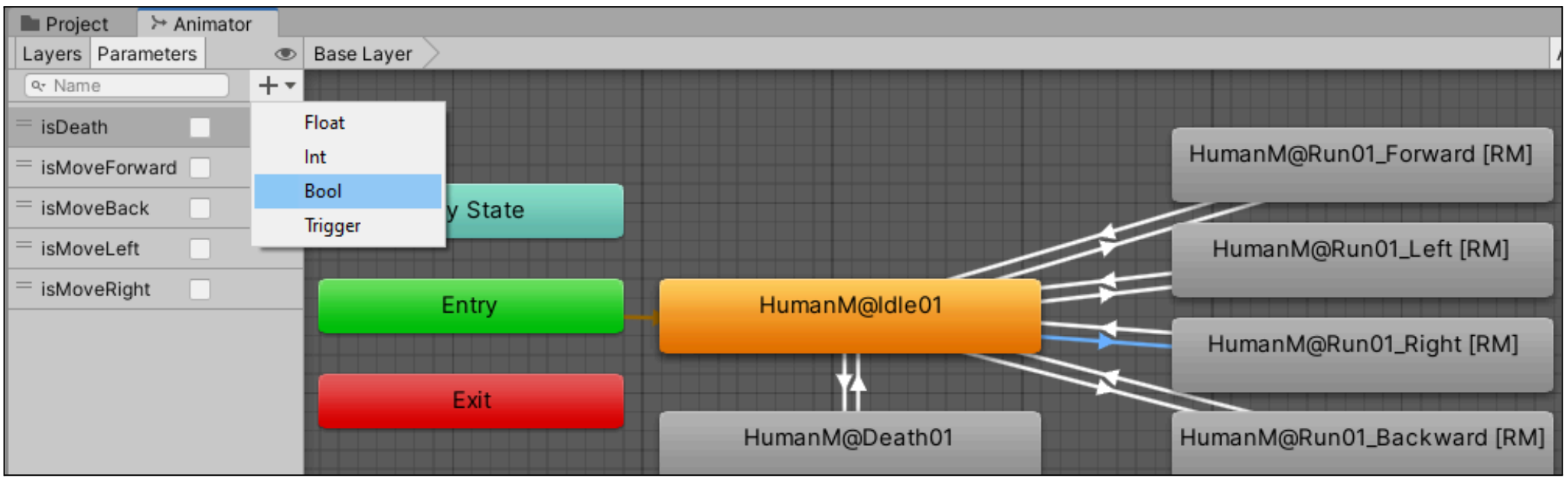



В рамках описанного здесь примера у всех созданных связей между анимациями надо снять флаг **Has Exit Time**. Включенный флаг на этой опции запрещает анимации прерывать предыдущую анимацию до завершения, тут это не нужно.



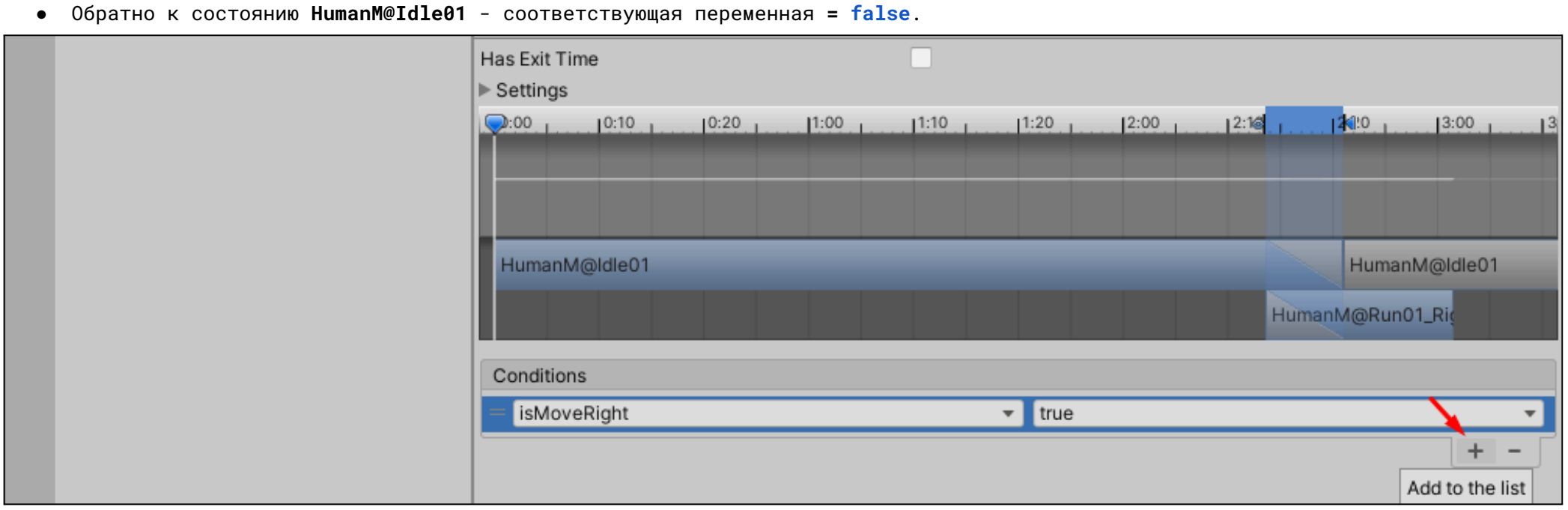

Анимационному контроллеру добавлены переменные типа **Bool**:

- isDeath
- isMoveForward
- isMoveBack
- isMoveLeft
- isMoveRight




У всех созданных связей **настроены условия воспроизведения** (**Conditions**):

- От состояния **HumanM@Idle01** к состояниям бега - соответствующая переменная **=** **true**.
- Обратно к состоянию **HumanM@Idle01** - соответствующая переменная **=** **false**.




## ※ Контроллер персонажа

**Информация из первой части:**

Компонент **Character Controller** дает персонажу простой коллайдер в форме капсулы, который всегда находится в вертикальном положении, частично имитирует физику (импульс игнорируется). У него особые функции для назначения скорости и направления объекта, и он не требует наличия компонента **Rigidbody** (физическое тело).

Персонаж с этим компонентом не может проходить сквозь **статические коллайдеры** в сцене - не проваливается сквозь пол, и не ходит сквозь стены. При движении он может отталкивать объекты с **Rigidbody**, но когда они толкают его, персонаж не получает импульс.

Персонаж будет подниматься по лестницам, если высота ступеньки ниже значения свойства **Step Offset**,

Персонаж будет  подниматься по склону если угол наклона у склона меньше значения свойства **Slope Limit**.

Свойство **Skin Width** позволяет другим объектам (коллайдерам) слегка пересекать контроллер. Это уменьшает тряску при соприкосновениях и защищает от застревания. Рекомендуемое значение более **0.01** и больше **Radius / 10**.

Уменьшить тряску при движении позволяет увеличение **Min Move Distance**, но чаще всего его оставляют 0 или близком значении.

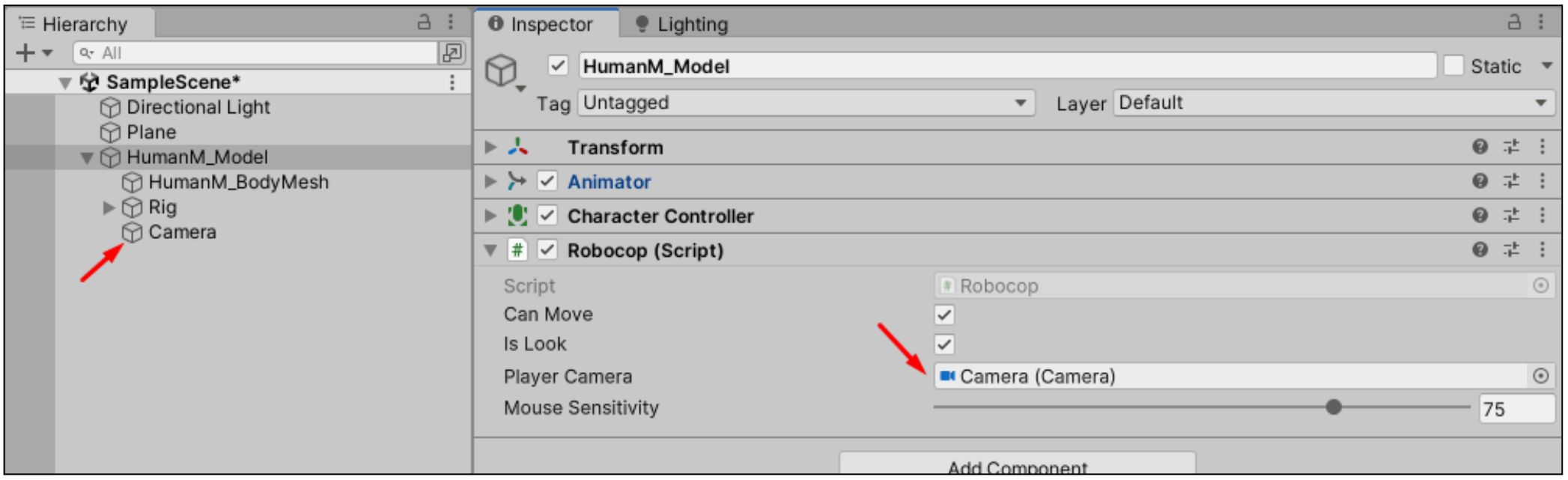
Чтобы управляемый персонаж мог толкать другие объекты, обладающие **Character Controller** или **Rigidbody**, надо применять к ним силы через скрипты при помощи функции **OnControllerColliderHit()**.

Добавить кукле компонент **Character Controller** и отрегулировать в его настройках расположение центра на высоте от 1.

Для удобства слежения за персонажем-куклой камеру на сцене надо перетащить в объект персонажа и закрепить в удобных координатах.

## ※ Скрипт управления

Для управления куклой создан скрипт **Robocop**, который добавлен объекту куклы на сцене. Скрипт передает управляющие сигналы в **анимационный контроллер** персонажа. Скрипт предусматривает добавление ссылки на камеру игрока.

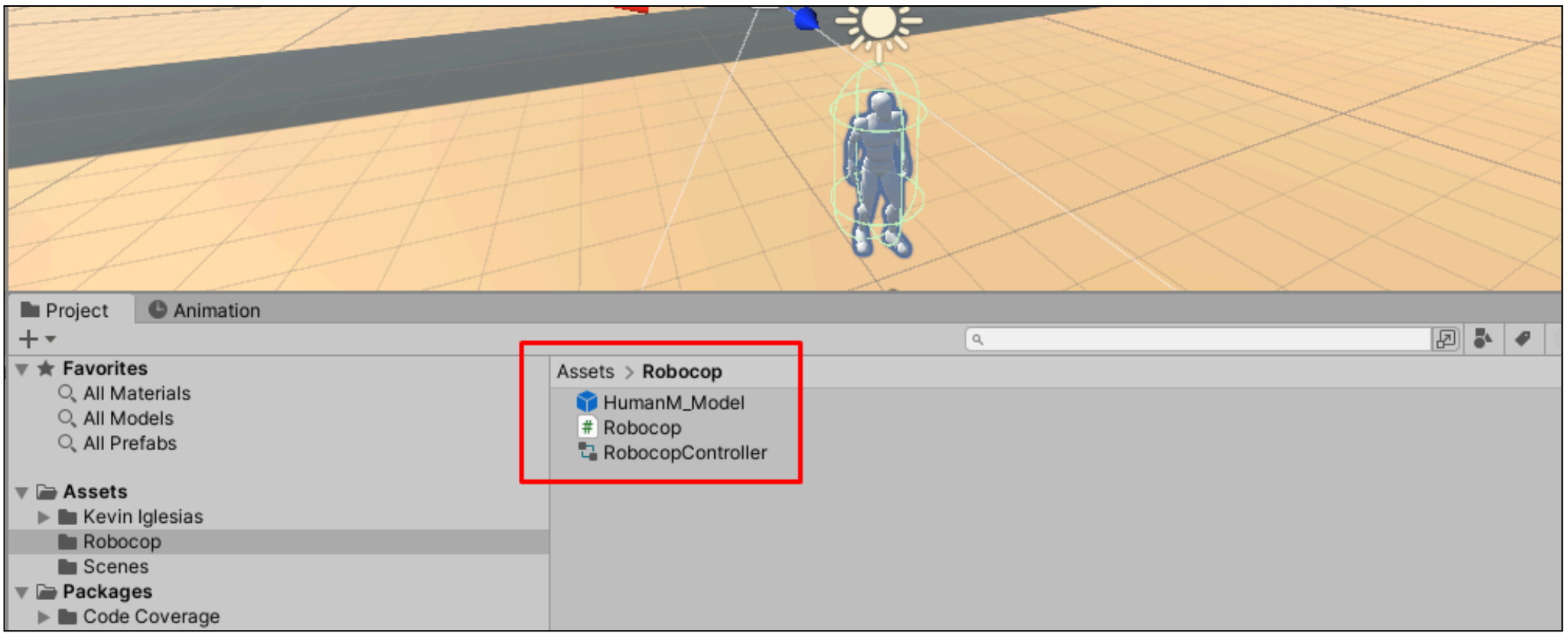



Для возможности управлять взглядом камеры и поворотами через мышку в скрипт добавлено чтение данных с виртуальных осей - **GetAxis**.

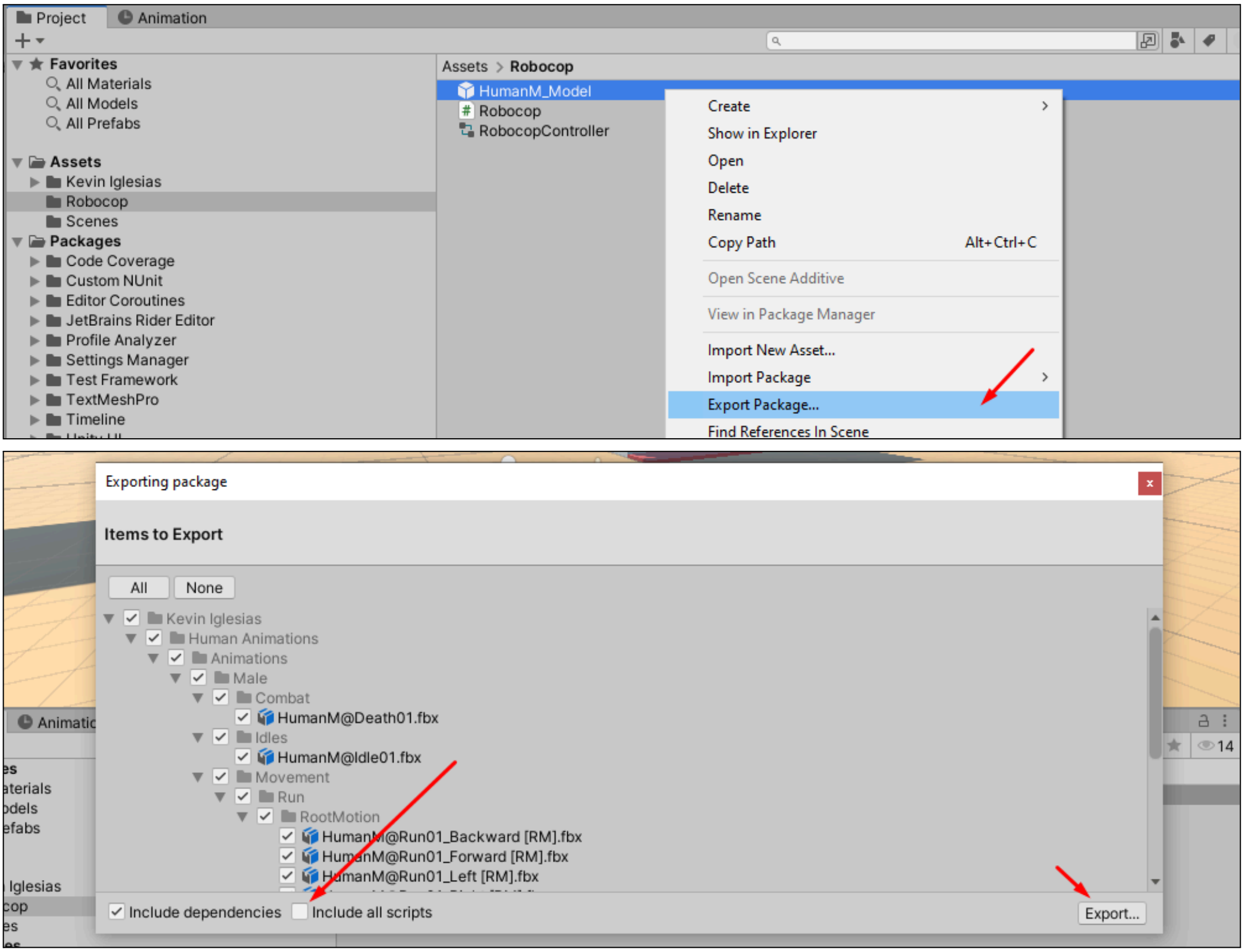
```
using UnityEngine;

public class Robocop : MonoBehaviour
{
    public bool canMove = true;
    //public bool isLook = true;
    public Camera playerCamera; // в настройках появится слот под камеру - надо создать ссылку на камеру игрока
    [Range(0, 100)] public float mouseSensitivity = 75f;
    [Range(0f, 200f)] private float snappiness = 100f;
    private float xRotation, yRotation;
    private float xVelocity, yVelocity;
    private float currentTiltAngle = 0f;
    private float tiltVelocity = 0f;
    private Animator animator;

    private void Start()
    {
        animator = GetComponent<Animator>();
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.I))
        {
            animator.SetBool("isDeath", !animator.GetBool("isDeath"));
            canMove = !canMove;
        }

        if (!Cursor.visible)
        {
            float mouseX = Input.GetAxis("Mouse X") * 10 * mouseSensitivity * Time.deltaTime;
            float mouseY = Input.GetAxis("Mouse Y") * 10 * mouseSensitivity * Time.deltaTime;

            xRotation += mouseX;
            yRotation -= mouseY;
            yRotation = Mathf.Clamp(yRotation, -90f, 90f);

            xVelocity = Mathf.Lerp(xVelocity, xRotation, snappiness * Time.deltaTime);
            yVelocity = Mathf.Lerp(yVelocity, yRotation, snappiness * Time.deltaTime);

            currentTiltAngle = Mathf.SmoothDamp(currentTiltAngle, 0f, ref tiltVelocity, 0.2f);
            playerCamera.transform.localRotation = Quaternion.Euler(yVelocity - currentTiltAngle, 0f, 0f);
            transform.rotation = Quaternion.Euler(0f, xVelocity, 0f);
        }
    }

    private void FixedUpdate()
    {
        animator.SetBool("isMoveForward", Input.GetKey(KeyCode.W) && canMove);
        animator.SetBool("isMoveBack", Input.GetKey(KeyCode.S) && canMove);

        animator.SetBool("isMoveLeft", Input.GetKey(KeyCode.A) && canMove);
        animator.SetBool("isMoveRight", Input.GetKey(KeyCode.D) && canMove);
    }
}
```

Обычно принято использовать не множество переменных с разными названиями (isMoveForward, isMoveBack и т.д.), а одну - state, но с разными числовыми значениями для анимаций. Тогда установка значения аниматору будет выглядеть как:
`Input.GetKey(KeyCode.D) ? animator.SetInteger("state", 5); // хотя это тоже говнокод, все таки не гайд по программированию`
Соответственно по другому будут выглядеть и условия включения анимаций. О возможностях чтения виртуальных осей справка далее.

Полученный набор надо экспортировать. Для этого в проекте создать папку. Перетащить в новую папку из окна сцены настроенную модель (с компонентом аниматор, внутри которого указан анимационный контроллер и аватар, и с управляющим скриптом). В эту же папку переместить управляющий скрипт и анимационный контроллер.




И экспортировать префаб куклы, ссылки на скрипты **Include all scripts** при экспорте отключить:



## ※ Input Manager

Виртуальные оси и кнопки могут быть созданы в **Input Manager**, а конечные пользователи могут настраивать ввод с клавиатуры на экране конфигурационного диалогового окна.

> Это мини-раздел носит ознакомительно справочный характер, примеры работы с этими данными встречаются по ходу дела.

Виртуальные оси доступны из скриптов по их именам. При создании проекта по умолчанию создаются следующие оси ввода по умолчанию:
**Horizontal** привязана к клавишам [**W**] **(+1)**, [**S**] **(-1)**
**Vertical** привязана к клавишам [**A**] **(-1)**, [**D**] **(+1)**
Ось может иметь значение от -1 до +1, на нейтральное положение указывает 0.
**Fire1** - клавиша **Ctrl** (Control)
**Fire2** - клавиша **Alt** (Option)
**Fire3** - клавиша Command (macOS)
**Mouse X** и **Mouse Y** привязаны к перемещениям мыши
**Window Shake X** и **Window Shake Y** привязаны к перемещению окна

Можно добавить новые виртуальные оси в меню редактора **Edit / Project Settings / Input Manager**.
В Input Manager можно изменить настройки каждой оси. Ось привязывается к двум кнопкам на джойстике, мыши или клавиатуре.

| Свойство | Функция |
|---|---|
| Name | Имя, используемое для доступа к оси из скрипта. |
| Descriptive Name | Имя положительного значения, отображаемое на вкладке Input диалогового окна Configuration в автономны сборках. |
| Descriptive Negative Name | Имя отрицательного значения, отображаемое на вкладке Input диалогового окна Configuration в автономны сборках. |
| Negative Button | Кнопка, используемая для смещения значения оси в отрицательном направлении. |
| Positive Button | Кнопка, используемая для смещения значения оси в положительном направлении. |
| Alt Negative Button | Альтернативная кнопка, используемая для смещения значения оси в отрицательном направлении. |
| Alt Positive Button | Альтернативная кнопка, используемая для смещения значения оси в положительном направлении. |
| Gravity | Скорость в единицах в секунду, с которой ось возвращается в нейтральное положение, когда кнопки не нажаты. |
| Dead | Размер аналоговой мертвой зоны. Все значения аналоговых устройств, попадающие в этот диапазон, считаются нейтральными. |
| Sensitivity | Скорость в единицах в секунду, с которой ось движется к заданному значению. Только для цифровых устройств. |
| Snap | Если включено, значение оси будет сбрасываться в ноль при нажатии кнопки в противоположном направлении. |
| Invert | Если включено, Negative Buttons будут выдавать положительные значения, а Positive отрицательные. |
| Type | Тип ввода, который будет управлять осью. Например, **Key or Mouse Button**, означает, что этой осью будет управлять или кнопка клавиатуры или кнопка мыши. Значение **Mouse Movement** означает, что осью будет управлять перемещение мыши, а значение **Joystick Axis** - управление будет осуществляться через джойстик. |
| Axis | Ось подключенного устройства, которая будет управлять этой осью. |
| Joy Num | Подключенный джойстик, который будет управлять этой осью. |

Перемещение по горизонтальной оси [**AD**], по вертикальной [**WS**]. Нейтральное положение 0.

Когда игрок пойдет влево, то значение горизонтальной оси будет уменьшаться.
Соответственно смотрим какая клавиша назначена для **Negative Button** и **Alt Negative Button**.
Это будут кнопки [**left**] (стрелка влево) и кнопка [**a**][**A**].

Когда игрок пойдет вправо, то значение горизонтальной оси будет увеличиваться.
Соответственно смотрим какая клавиша назначена для **Positive Button** и **Alt Positive Button**.
Это будут кнопки [**right**] (стрелка вправо) и кнопка [**d**][**D**].

Запросить текущее состояние из скрипта (вернет дробное float значение от -1 до 1):

```
value = Input.GetAxis("Horizontal"); // вернет от -1 до 1
```

Ось может иметь значение от -1 до 1, нейтральное положение указывает 0.

Получить подтверждение нажатия кнопки в скрипте (вернет булево значение):

```
value = Input.GetKey("a"); // вернет true или false
```

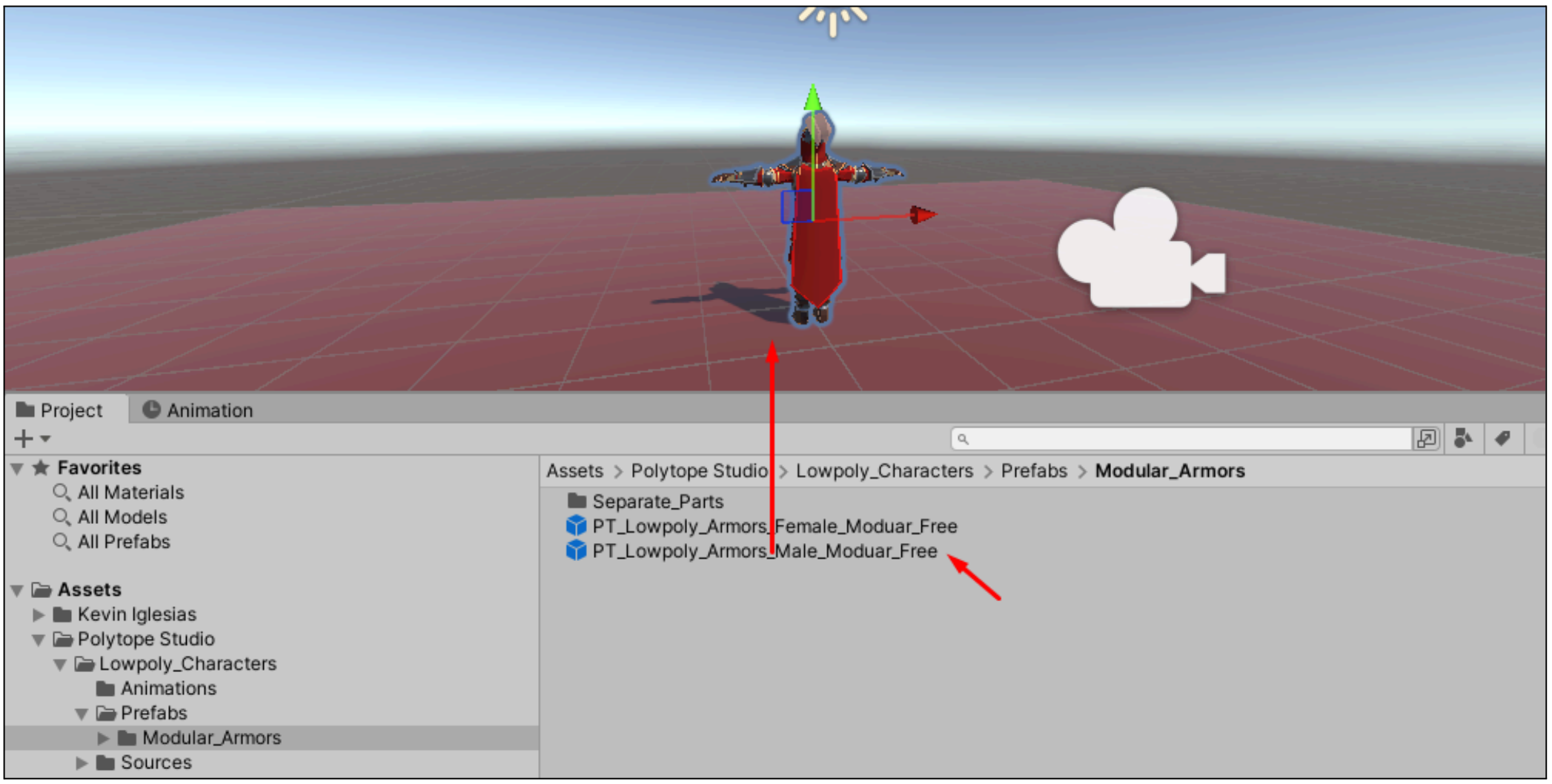
Это в случае джойстика или клавиатуры. Изменение осей **Mouse** и **Window Shake** показывает насколько мышь или окно сдвинулось по сравнению с последним кадром, это значит что они могут быть больше 1 или меньше -1, если пользователь быстро двигает мышь.

| Тип кнопок | Значение |
|---|---|
| Обычные кнопки | a..z |
| Цифровые кнопки | 0..9 |
| Стрелки | up, down, left, right |
| Цифровой блок | 1,2,3,+,equals |
| Специальные комбинации | right shift, left shift, right ctrl, left ctrl, right alt, left alt, right cmd, left cmd |
| Мышка | mouse0, mouse1, mouse2 |
| Джойстик | joystick button 0, joystick button 1, joystick button 2.. |
| Заданный джойстик | joystick 1 button 0, joystick 1 button 1, joystick 2 button 0.. |
| Специальные кнопки | backspace, tab, escape, return, space, delete, enter, insert, home, end, page up, rage down |
| Функциональные кнопки | f1, f2, f3.. |

## Изменение модели

Создать новую сцену и подгрузить бесплатный ассет (если сцена в новом проекте - импортировать созданный ранее пакет с куклой): https://assetstore.unity.com/packages/3d/characters/lowpoly-modular-armors-free-medieval-fantasy-series-199890

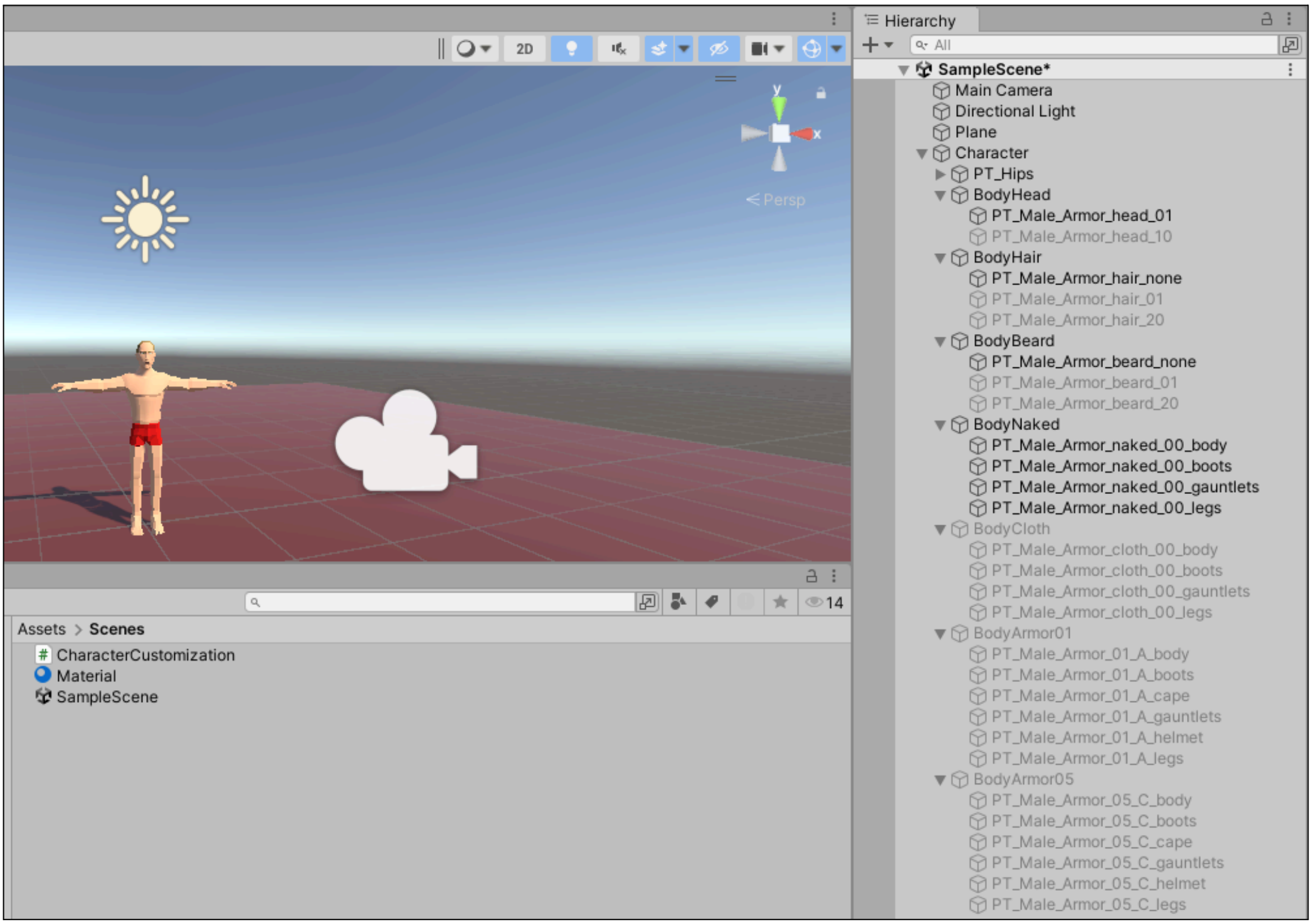
На новую цену поместить модульную модель персонажа (в примере модель мужская) и распаковать в окне иерархии в объект:




Создать новый скрипт (**CharacterCustomization**). Скрипт добавить размещенному на сцене персонажу. В скрипте заменить код:

```
using UnityEngine;

public class CharacterCustomization : MonoBehaviour
{
    public GameObject PLAYER_CHARACTER;
    public GameObject[] PT_Male_Heads;
    public GameObject[] PT_Male_Hairs;
    public GameObject[] PT_Male_Beards;
    public GameObject[] PT_Male_Armors;
    private int currentHead = 0;
    private int currentHair = 0;
    private int currentBeard = 0;
    private int currentArmor = 0;
    private bool isRotateModel = false;

    private void Update()
    {
        if (Input.GetKeyUp(KeyCode.R)) // начать крутить модель персонажа
            isRotateModel = !isRotateModel;

        if (isRotateModel)
            PLAYER_CHARACTER.transform.Rotate(new Vector3(0, 1, 0), 30 * Time.deltaTime);

        if (Input.GetKeyUp(KeyCode.Alpha1)) // кнопка 1 Голова
            ChangeHead();

        if (Input.GetKeyUp(KeyCode.Alpha2)) // кнопка 2 Волосы
            ChangeHair();

        if (Input.GetKeyUp(KeyCode.Alpha3)) // кнопка 3 Борода
            ChangeBeard();

        if (Input.GetKeyUp(KeyCode.Alpha4)) // кнопка 4 Броня
            ChangeArmor();
    }

    private void OnGUI()
    {
        if (GUI.Button(new Rect(50, 100, 150, 30), "R - Поворот модели"))
            isRotateModel = !isRotateModel;

        if (GUI.Button(new Rect(50, 150, 150, 30), "1 - Голова"))
            ChangeHead();

        if (GUI.Button(new Rect(50, 200, 150, 30), "2 - Волосы"))
            ChangeHair();

        if (GUI.Button(new Rect(50, 250, 150, 30), "3 - Борода"))
            ChangeBeard();

        if (GUI.Button(new Rect(50, 300, 150, 30), "4 - Броня"))
            ChangeArmor();
    }

    private void ChangeHead()
    {
        PT_Male_Heads[currentHead].SetActive(false);

        if (++currentHead == PT_Male_Heads.Length)
            currentHead = 0;

        PT_Male_Heads[currentHead].SetActive(true);
    }

    private void ChangeHair()
    {
        PT_Male_Hairs[currentHair].SetActive(false);
        if (++currentHair == PT_Male_Hairs.Length)
            currentHair = 0;
```

```
        PT_Male_Hairs[currentHair].SetActive(true);
    }

    private void ChangeBeard()
    {
        PT_Male_Beards[currentBeard].SetActive(false);

        if (++currentBeard == PT_Male_Beards.Length)
            currentBeard = 0;

        PT_Male_Beards[currentBeard].SetActive(true);
    }

    private void ChangeArmor()
    {
        PT_Male_Armors[currentArmor].SetActive(false);

        if (++currentArmor == PT_Male_Armors.Length)
            currentArmor = 0;

        PT_Male_Armors[currentArmor].SetActive(true);

        HideHair(currentArmor <= 1); // если персонаж голый или в одежде, то волосы должны быть видны
    }

    private void HideHair(bool value)
    {
        for (int i = 1; i < PT_Male_Hairs.Length; i++)
            PT_Male_Hairs[i].GetComponent<SkinnedMeshRenderer>().enabled = value;
    }
}
```



**Что было сделано в моделью персонажа на сцене:**

Для группировки деталей были созданы пустые объекты **Create Empty** и переименованы соответственно теме группировки:

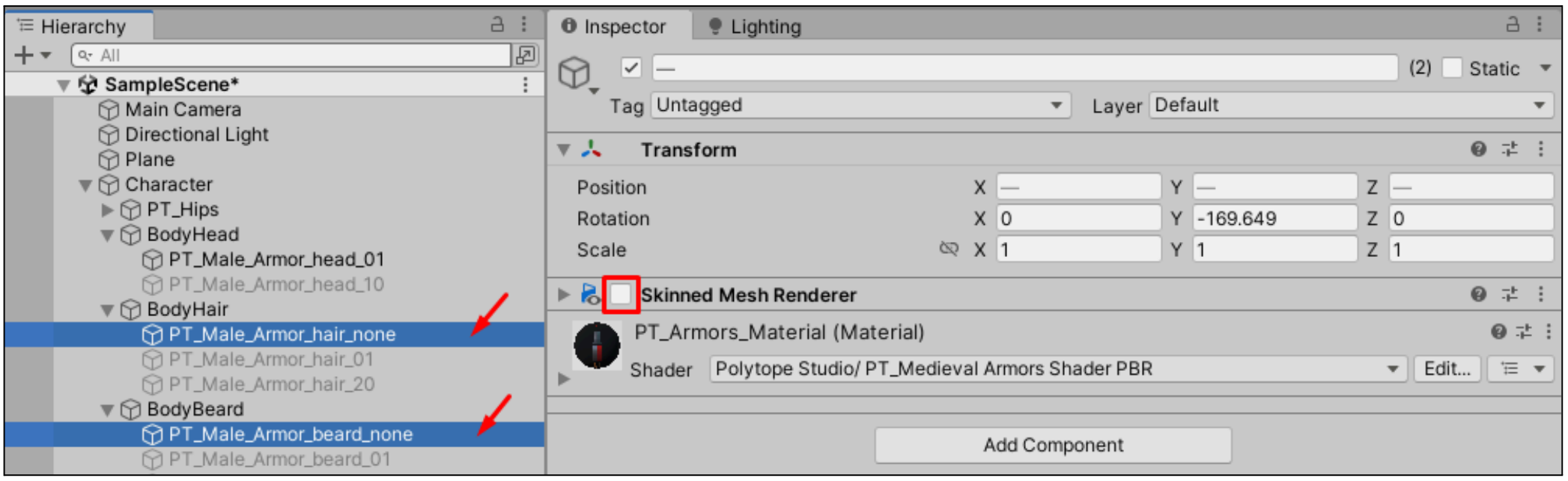
- **BodyHead**
- **BodyHair**
- **BodyBeard**
- **BodyNaked**
- **BodyCloth**
- **BodyArmor01**
- **BodyArmor05**

В объекты для группировки сложены детали модели с небольшим нюансом:

Деталь **PT_Male_Armor_hair_01** продублирована, дубль переименован в **PT_Male_Armor_hair_none**.

Деталь **PT_Male_Armor_beard_01** продублирована, дубль переименован в **PT_Male_Armor_beard_none**.

У дублированных деталей **PT_Male_Armor_hair_none** и **PT_Male_Armor_beard_none** отключен **mesh**, получился вариант без бороды / волос:

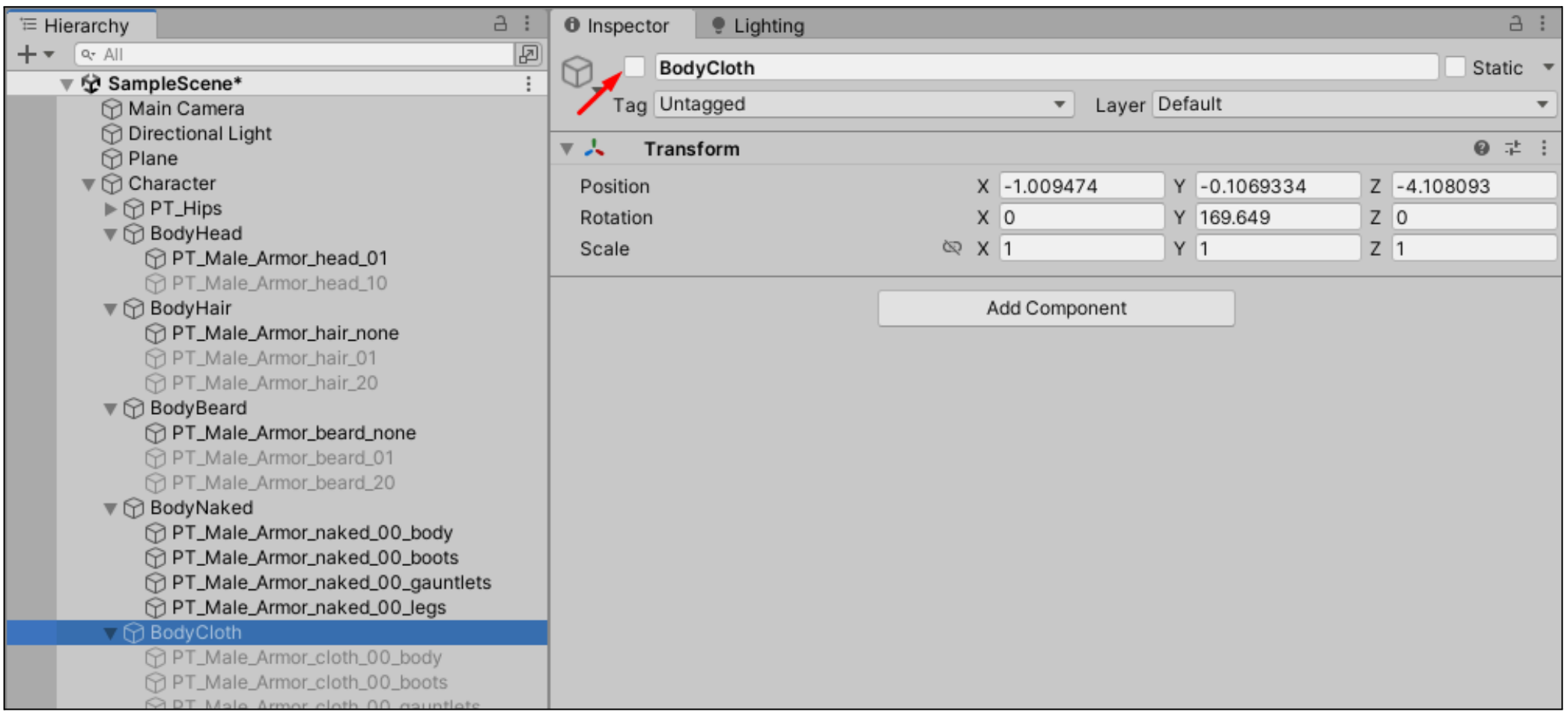



У всех остальных деталей **mesh** включен.

У всех объектов-деталей модели включена активность, кроме деталей из списка:

- **PT_Male_Armor_head_10**
- **PT_Male_Armor_hair_01**
- **PT_Male_Armor_hair_20**
- **PT_Male_Armor_beard_01**
- **PT_Male_Armor_beard_20**

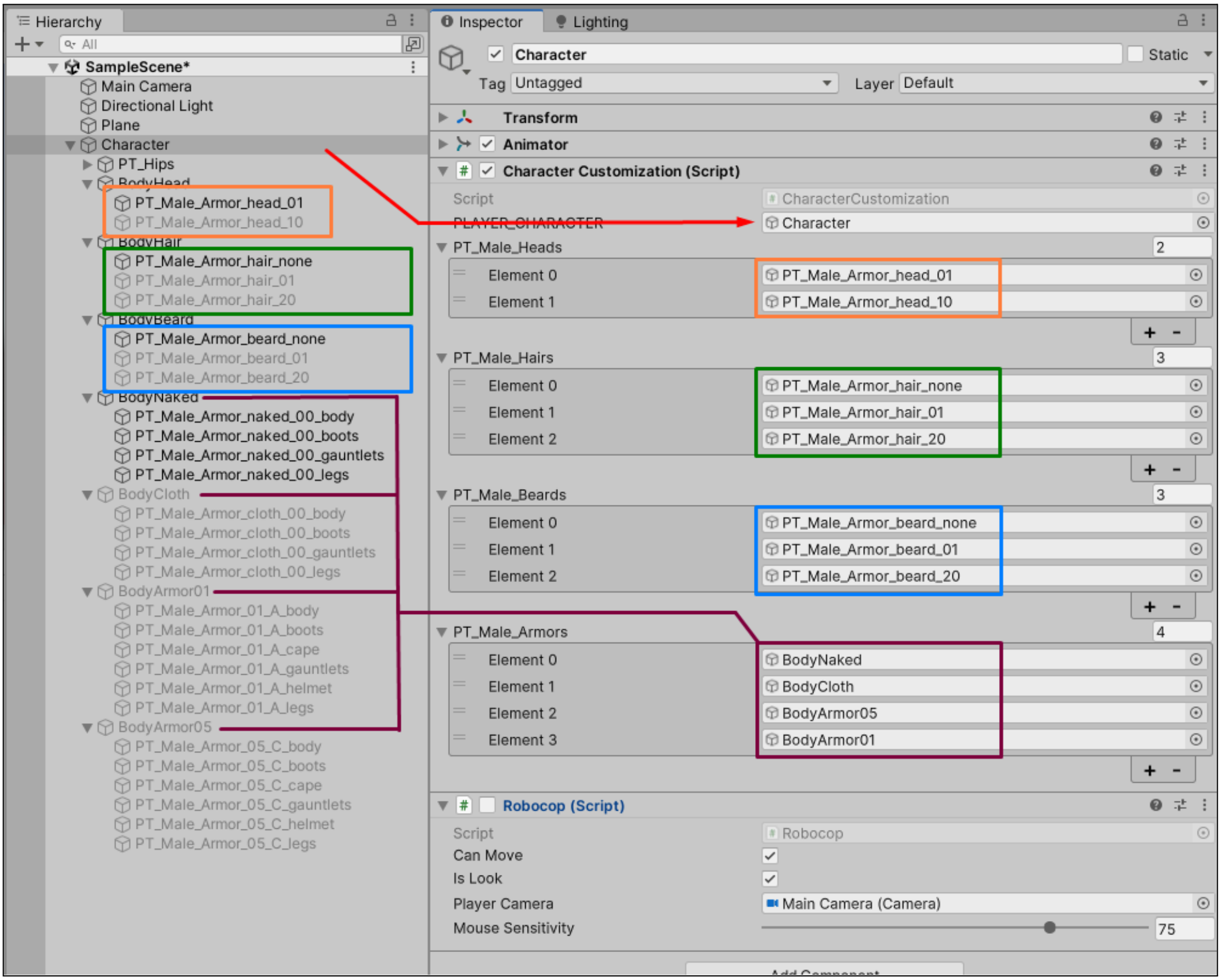
Группирующим объектам **BodyCloth, BodyArmor01, BodyArmor05** отключена активность:




В результате остается персонаж, у которого активно тело, голова, и дублированные волосы и усы без мешей.

Модели персонажа был добавлен созданный ранее анимационный контроллер, чтобы при запуске сцены не стоять в стандартной Т-позе.

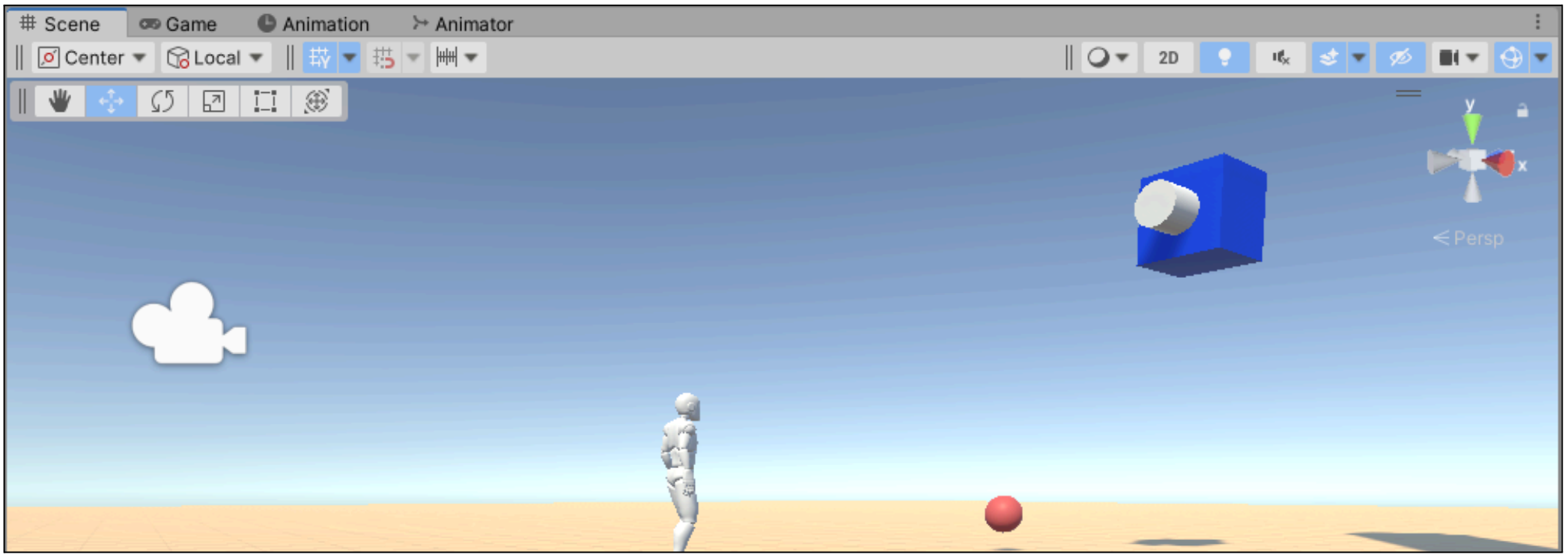
Скрипт, добавленный модели персонажа (в примере моделька на сцене переименована в **character**), создает список полей, в которые надо добавить ссылки на объекты модели. Для групп тела надо добавлять ссылку на групповой объект, а в разделах под голову, волосы, бороду надо добавлять ссылки на отдельные объекты.




При запуске сцены будут отрисованы кнопки, потыкав которые можно покрутить персонажа и изменить отображение частей модели:

## Преследование цели

На сцену с управляемой моделью персонажа добавить два объекта. Сферу (Follower), которая будет преследовать свою цель (персонажа). Куб (Observer), который будет поворачиваться в сторону наблюдаемой цели:




Создать скрипт для каждого объекта. Скрипт для сферы - **Follower**:

```
using UnityEngine;

public class Follower : MonoBehaviour
{
    [SerializeField] private Transform target;
    [SerializeField] private float speed = 1.5f;
    [SerializeField] private float followDistance = 1.5f;
    [SerializeField] private float heightOffset = 1f;
    [SerializeField] private float chaseDistance = 2f;
    [SerializeField] private float runSpeed = 4;

    private void Update()
    {
        if (target == null)
            return;

        float distance = Vector3.Distance(transform.position, target.position);

        if (distance < chaseDistance)
            return;

        speed = 1.5f;

        if (distance > chaseDistance * 2)
            speed = runSpeed;

        Vector3 endpoint = target.position + (target.forward * -followDistance) + (Vector3.up * heightOffset);
        Vector3 direction = (endpoint - transform.position).normalized;

        transform.Translate(direction * Time.deltaTime * speed);
    }
}
```

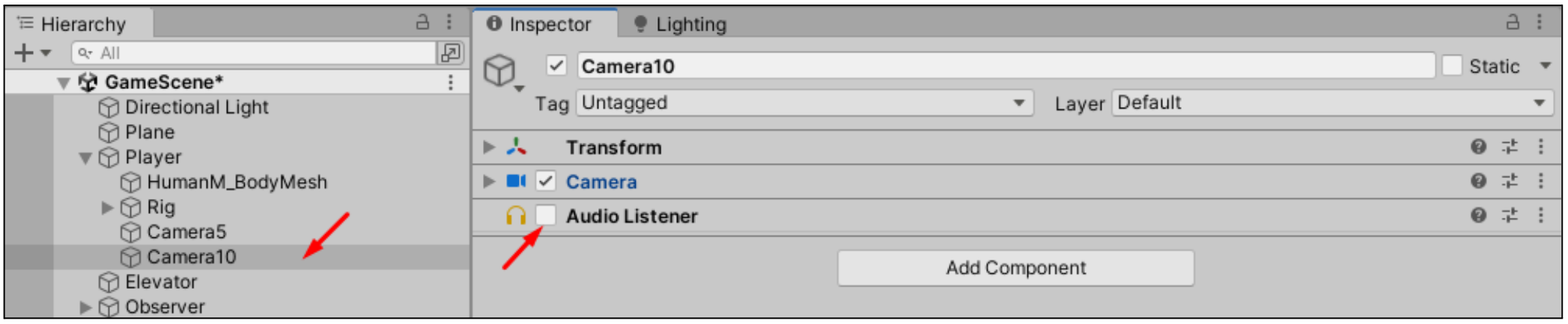
Скрипт для куба - **Observer**:

```
using UnityEngine;

public class Observer : MonoBehaviour
{
    [SerializeField] private Transform target;

    private void Update()
    {
        Vector3 direction = (target.position - transform.position).normalized;
        transform.forward = direction;

        transform.LookAt(target);
    }
}
```

Перед тестом каждому объекту указать объект-цель - сфере для преследования управляемого персонажа, кубу - персонажа или сферу.

## Переключение камеры

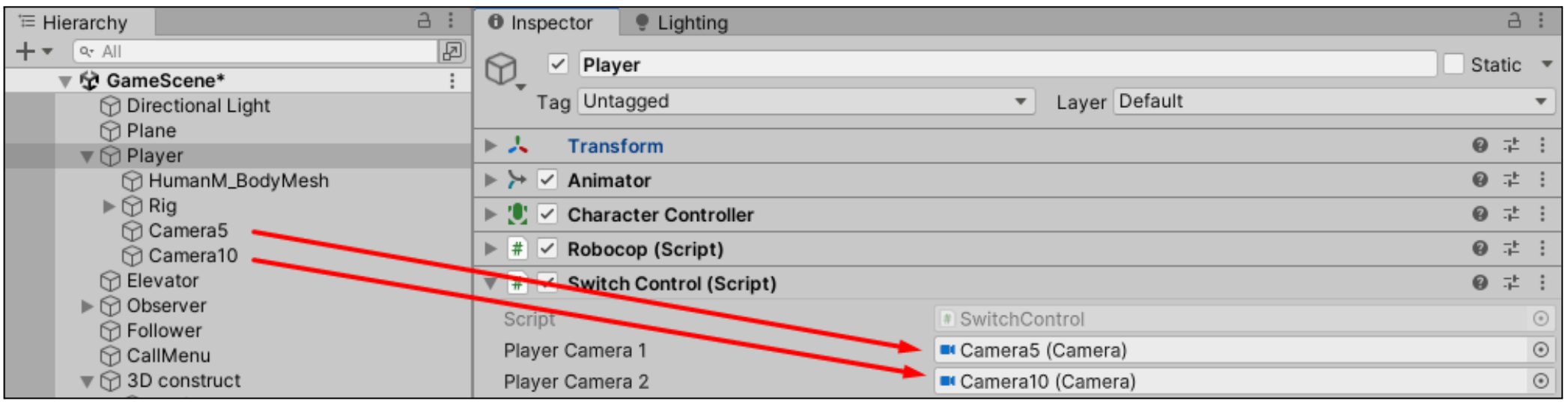
Добавить управляемому персонажу вторую камеру, которая будет находиться на более удаленной дистанции, чем первая.
На новой камере отключить компонент для прослушивания звука - [**Audio Listener**]:




Создать новый скрипт **SwitchControl** с кодом, и добавить его персонажу:

```
using UnityEngine;

public class SwitchControl : MonoBehaviour
{
    private Animator animator;
    public Camera playerCamera1;
    public Camera playerCamera2;

    private void Start()
    {
        animator = GetComponent<Animator>();
        playerCamera1.enabled = true;
        playerCamera2.enabled = false;
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Tab)) // переключать камеры нажатием табуляции
        {
            animator.applyRootMotion = !animator.applyRootMotion; // аниматор больше не блокирует перемещение персонажа
            playerCamera1.enabled = !playerCamera1.enabled;
            playerCamera2.enabled = !playerCamera2.enabled;
            ToggleCursorLock(); // Переключаем состояние курсора
        }
    }

    private void ToggleCursorLock()
    {
        if (Cursor.lockState == CursorLockMode.Locked)
        {
            Cursor.lockState = CursorLockMode.None;
            Cursor.visible = true; // Показываем курсор при разблокировке
        }
        else
        {
            Cursor.lockState = CursorLockMode.Locked;
            Cursor.visible = false; // Скрываем курсор при блокировке
        }
    }
}
```

Обе камеры надо добавить в параметры скрипта на персонаже:




Теперь при нажатии кнопки **Tab** будет переключаться активная камера персонажа.
Текущий скрипт переключения камеры нужен для удобного обзора при использовании навигатора в следующем разделе.

## ※ Умная камера

Один из вариантов камеры для наблюдения от третьего лица. Реализовано через новый скрипт **SmartCamera**:

```
using UnityEngine;

public class SmartCamera : MonoBehaviour
{
    [SerializeField] private Transform player;
    [SerializeField] private Vector3 offset = new Vector3(0, -1, 1);
    [SerializeField] private float pitch = 2f;
    [SerializeField] private float zoomSpeed = 3f;
    [SerializeField] private float minZoom = 5f;
    [SerializeField] private float maxZoom = 10f;
    [SerializeField] private float rotateSpeed = 120f;
    private float currentZoom = 8f;
    private float currentRotate = 0f;

    private void Update()
    {
        currentZoom -= Input.GetAxis("Mouse ScrollWheel") * zoomSpeed;
        currentZoom = Mathf.Clamp(currentZoom, minZoom, maxZoom);
        currentRotate -= Input.GetAxis("Horizontal") * rotateSpeed * Time.deltaTime;
    }

    private void LateUpdate()
    {
        transform.position = player.position - offset * currentZoom;
        transform.LookAt(player.position + Vector3.up * pitch);
        transform.RotateAround(player.position, Vector3.up, currentRotate);
    }
}
```

В этом варианте надо добавить скрипт на камеру, указать ей ссылку на управляемого персонажа и отрегулировать параметры.

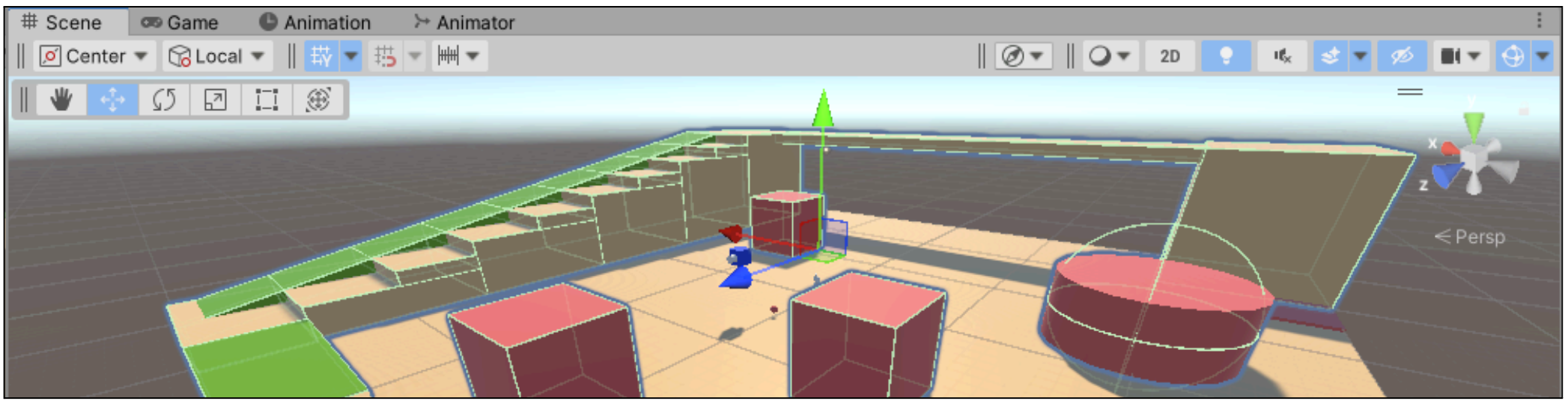
Камера будет следить за персонажем и крутиться вокруг него по орбите при нажатии кнопок [**AD**].

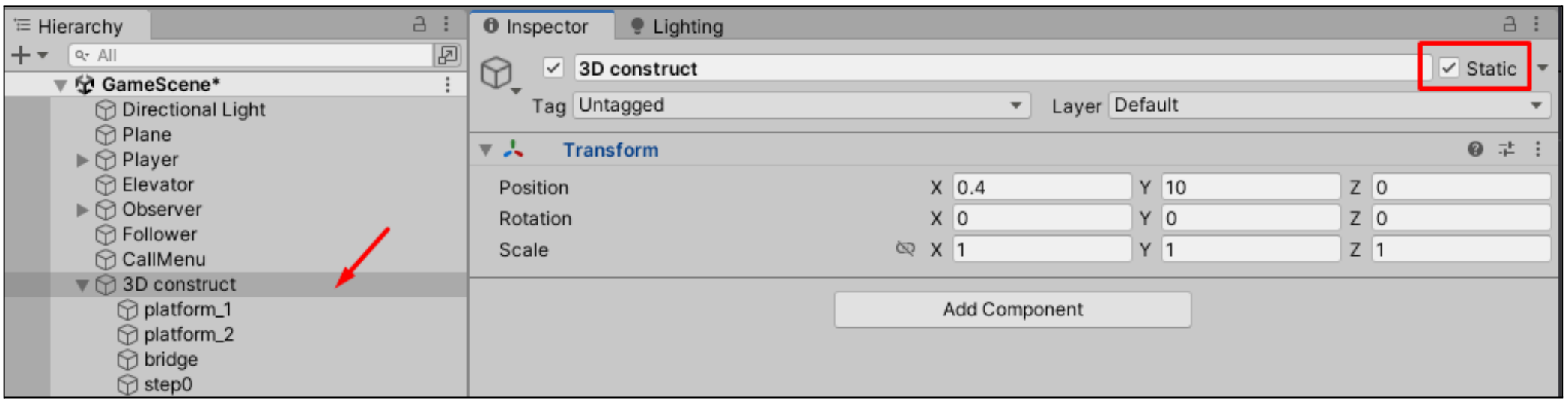
При прокручивании колеса мыши приближаться/ отдаляться.

В текущем виде с использованием кнопок [**AD**] есть нестыковка с клавишами управления на кукле, но для ознакомления с альтернативной системой управления передвижением подойдет (или как исходник для сборки другой системы управления).

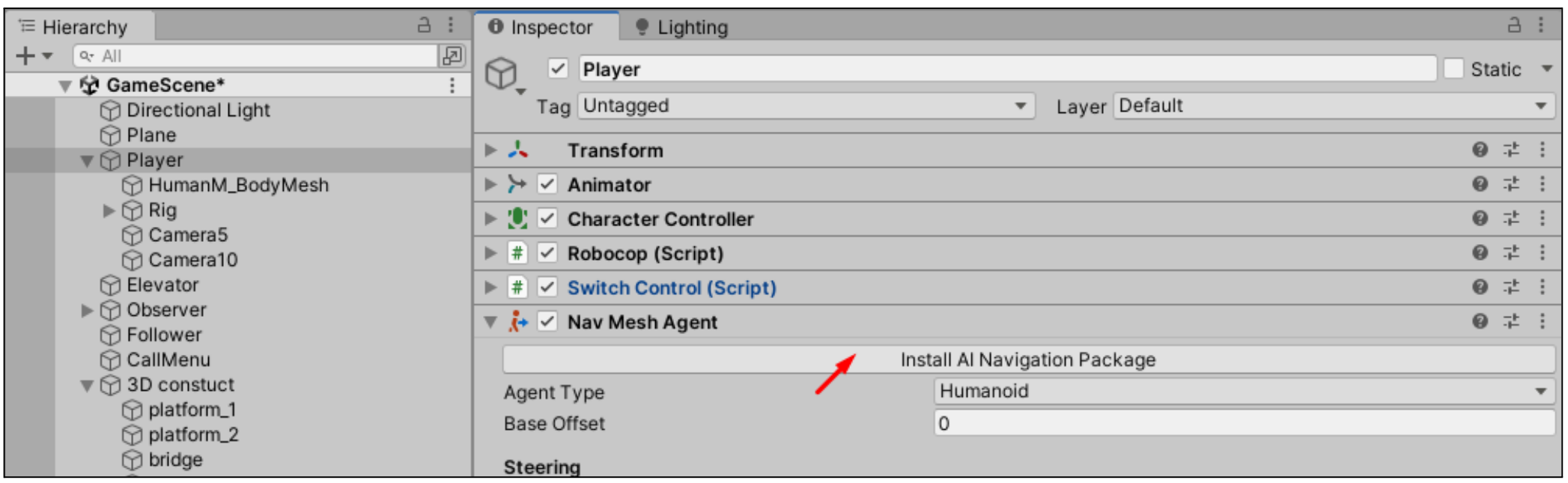
## Навигатор AI

Разместить на сцене с управляемым персонажем несколько объектов, часть из них являются препятствиями. Объекты сделать статичными:



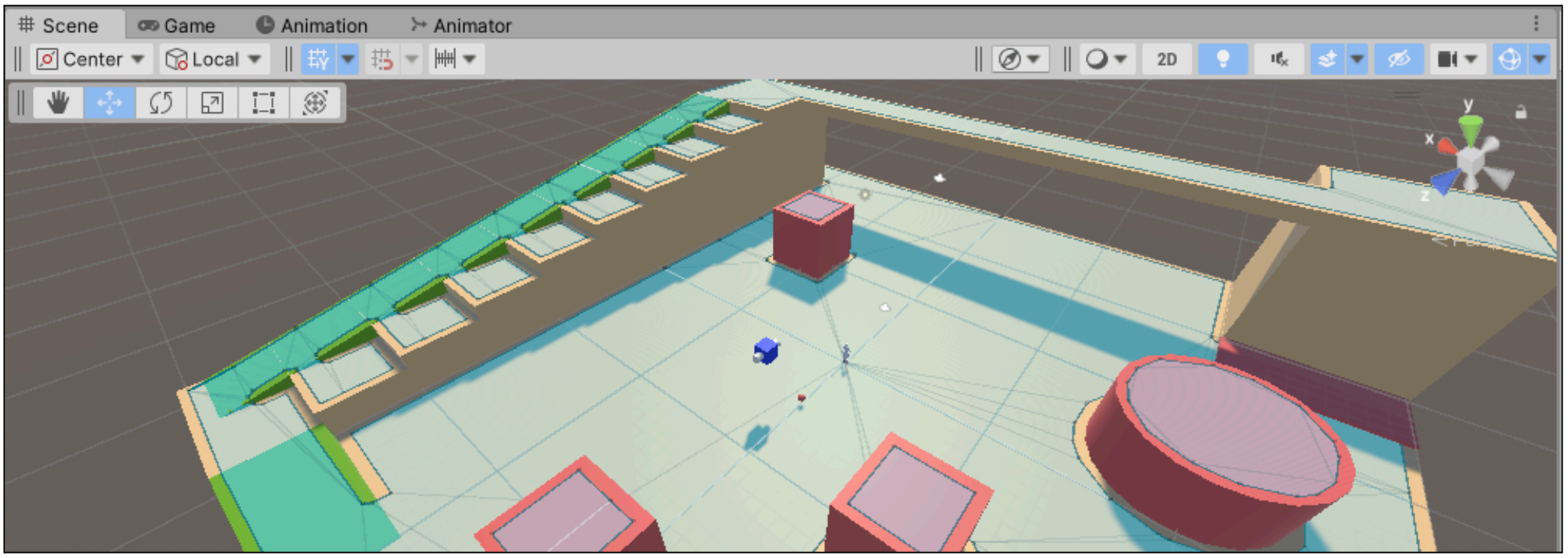




Добавить персонажу компонент [**Nav Mesh Agent**], и установить пакет для AI:




Открыть **окно** в меню редактора **Window / AI / Navigation (Obsolete)**. В окне [**Navigation (Obsolete)**] перейти на вкладку **Bake.**
Вкладка **Bake** содержит настройки запекания карты проходимости для навигации с помощью AI, на результат существенно влияют:

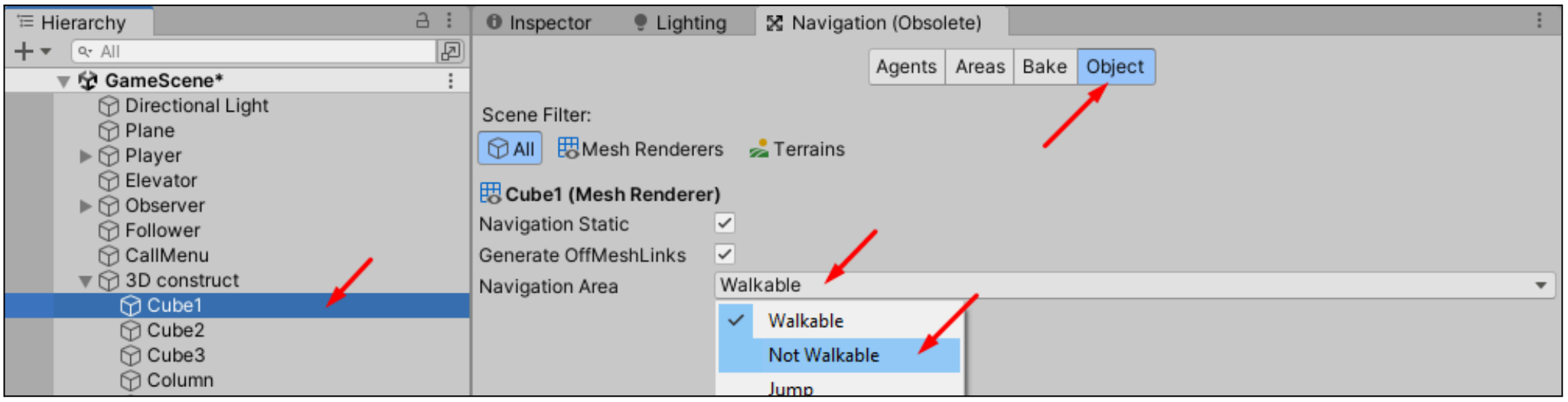
**Max Slope** - угол наклона плоскости, по которой можно подняться.
**Step Height** - высота ступенек, по которым можно подниматься.
Запекание карты проходимости учитывает только объекты на сцене в состоянии **Static**. Сформировать карту - кнопка [**Bake**].




Вкладка **Agents** позволяет регулировать настройки в режиме реального времени - во время теста.

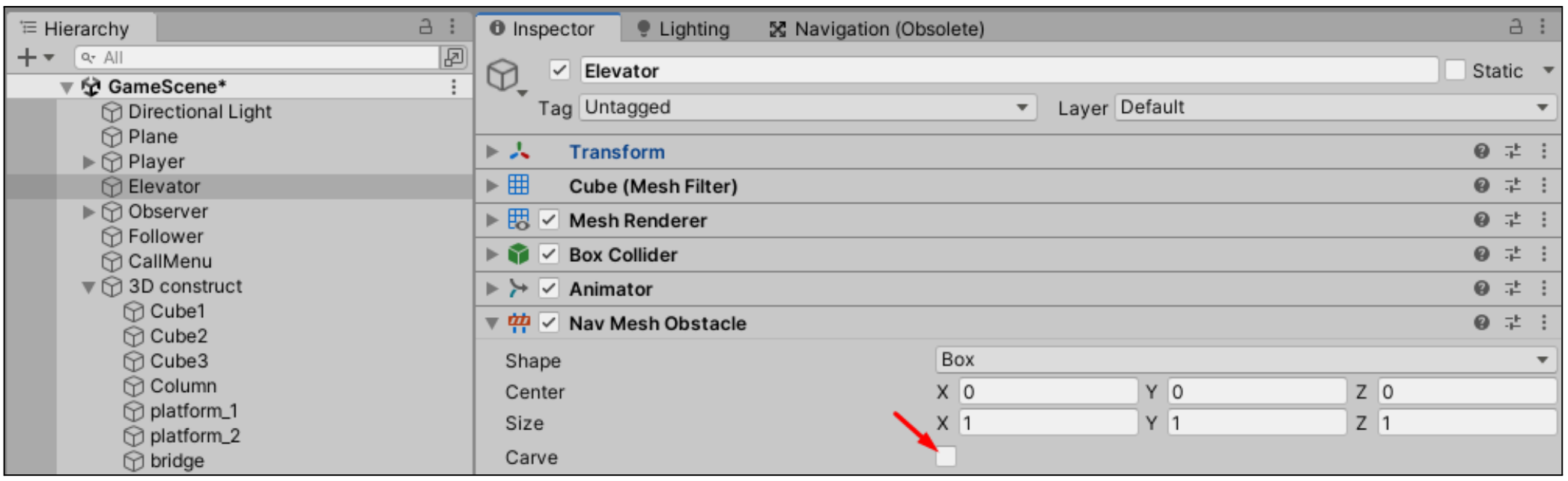
Теперь персонаж сможет передвигаться по сцене с помощью AI, но некоторые зоны ошибочно были оценены как доступные для посещения.

Если какой-либо объект не должен быть доступен для посещения, то надо его выбрать, переключится на вкладку **Object** и для свойства **Navigation Area** выбрать значение **Not Walkable**. После корректировки карту надо будет повторно запечь - [**Bake**].

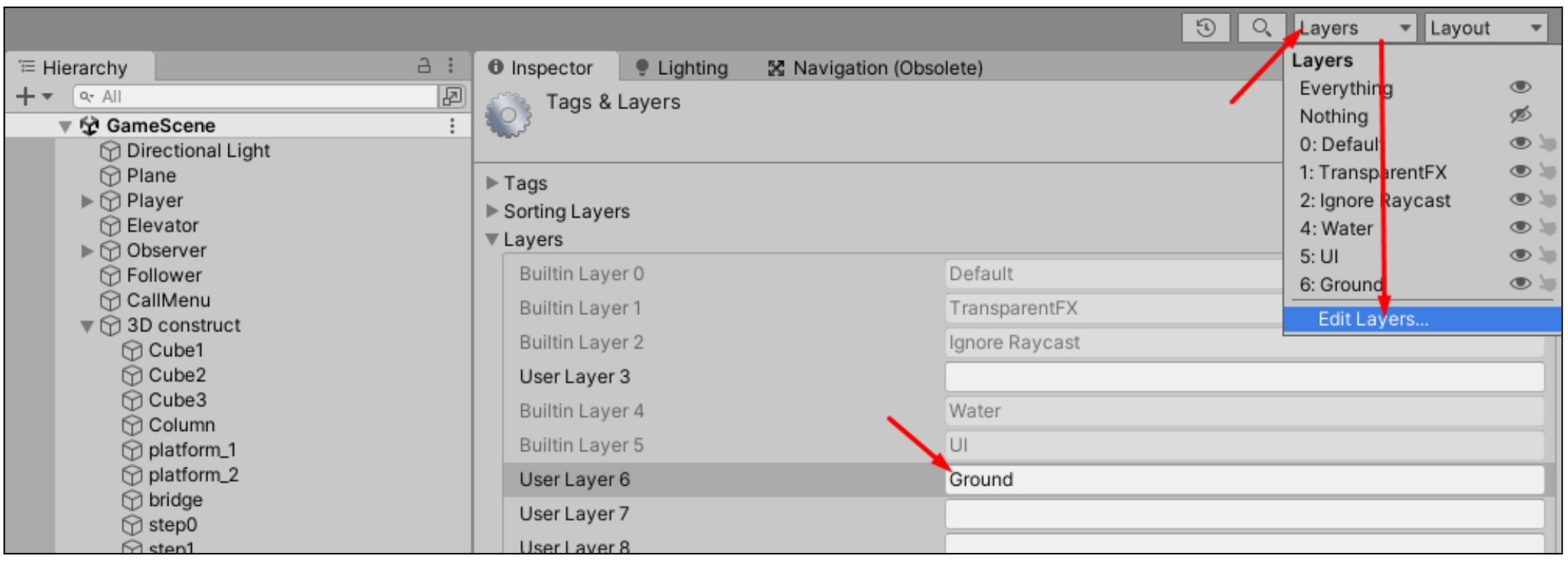



Для не статичных (движущихся) объектов на сцене надо добавлять компонент [**Nav Mesh Obstacle**] и включать у них флаг **Carve**.

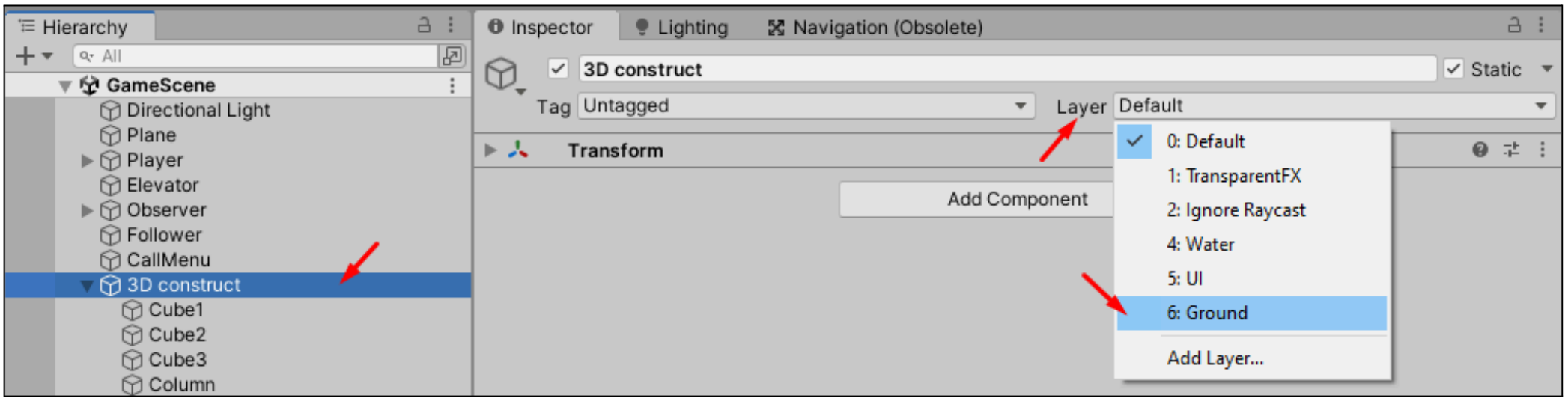
**Shape** - форма, которая вместе с размерами отрегулируют границы объекта для системы навигации (аналог коллайдера).




Создать новый слой - **Ground**.



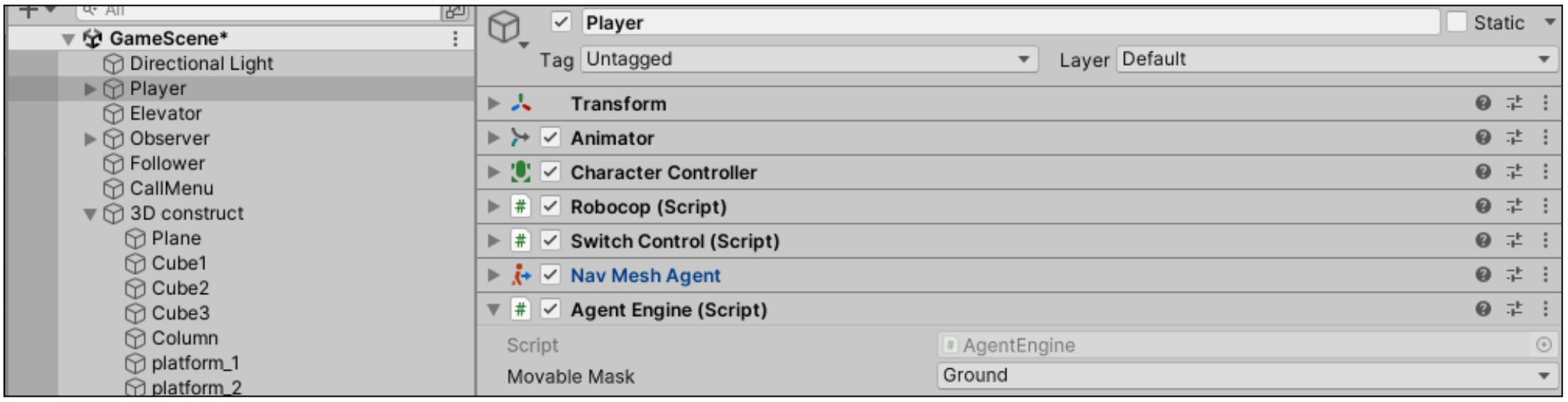

И все объекты-места куда должен добираться персонаж перенести в слой **Ground.**




Создать скрипт **AgentEngine**, который будет управлять движением персонажа через систему навигации:

```
using UnityEngine;
using UnityEngine.AI;

[RequireComponent(typeof(NavMeshAgent))]

public class AgentEngine : MonoBehaviour
{
    [SerializeField] private LayerMask movableMask;
    private NavMeshAgent agent;

    private void Start()
    {
        agent = GetComponent<NavMeshAgent>();
    }

    public void MoveToPoint(Vector3 point)
    {
        agent.SetDestination(point);
    }
}
```

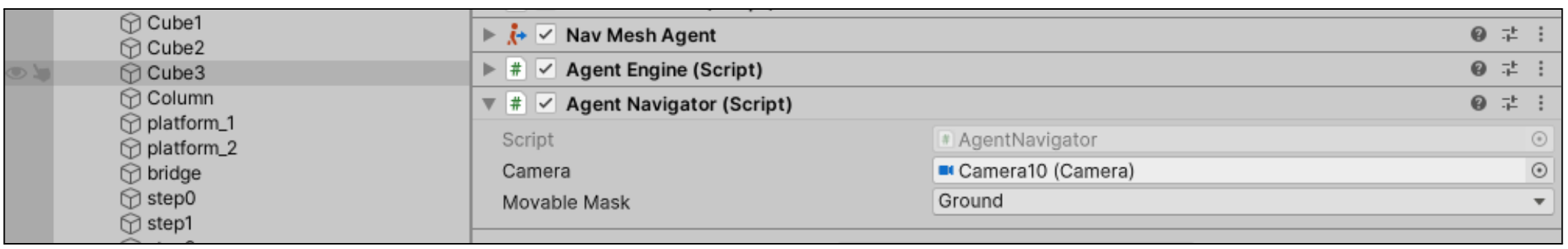
Добавить скрипт на модель и в опциях указать слой **Ground**:




Создать скрипт **AgentNavigator**, который будет определять целевую точку для перемещения:

```
using UnityEngine;

[RequireComponent(typeof(AgentEngine))]

public class AgentNavigator: MonoBehaviour
{
    private AgentEngine agent;
    [SerializeField] private Camera camera;
    [SerializeField] private LayerMask movableMask;

    private void Start()
    {
        agent = GetComponent<PlayerEngine>();
    }

    private void Update()
    {
        if (Input.GetMouseButtonDown(0)) // 0 левая кнопка мыши
        {
            Ray ray = camera.ScreenPointToRay(Input.mousePosition); // луч к точек клика
            RaycastHit hit; // для расчета столкновения луча с препятствиями
            // обработать только если луч попал в объект нужного слоя
            if (Physics.Raycast(ray, out hit, 100, movableMask)) // 100 ограничение дальности для построения луча
                agent.MoveToPoint(hit.point);
        }
    }
}
```

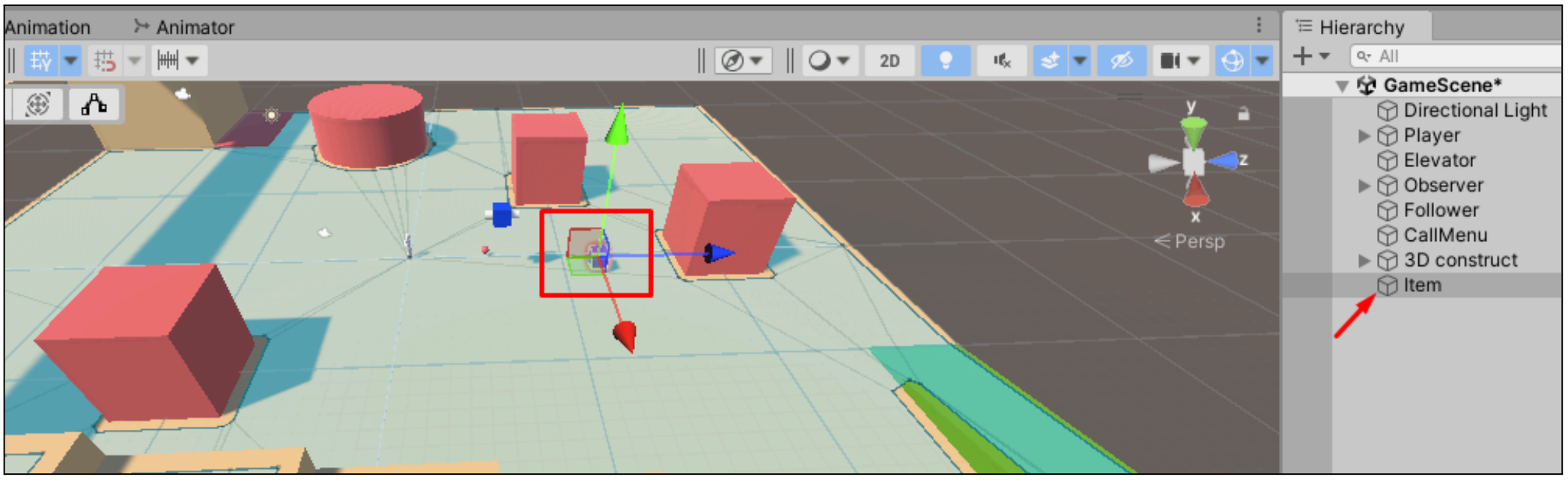
Добавить скрипт персонажу, в опциях скрипта указать ссылку на **вторую камеру** и слой **Ground**:




Теперь персонаж на сцене будет бегать под управлением агента навигации в точку, полученную от камеры при клике левой кнопкой мыши.

## ※ Интерактивное взаимодействие

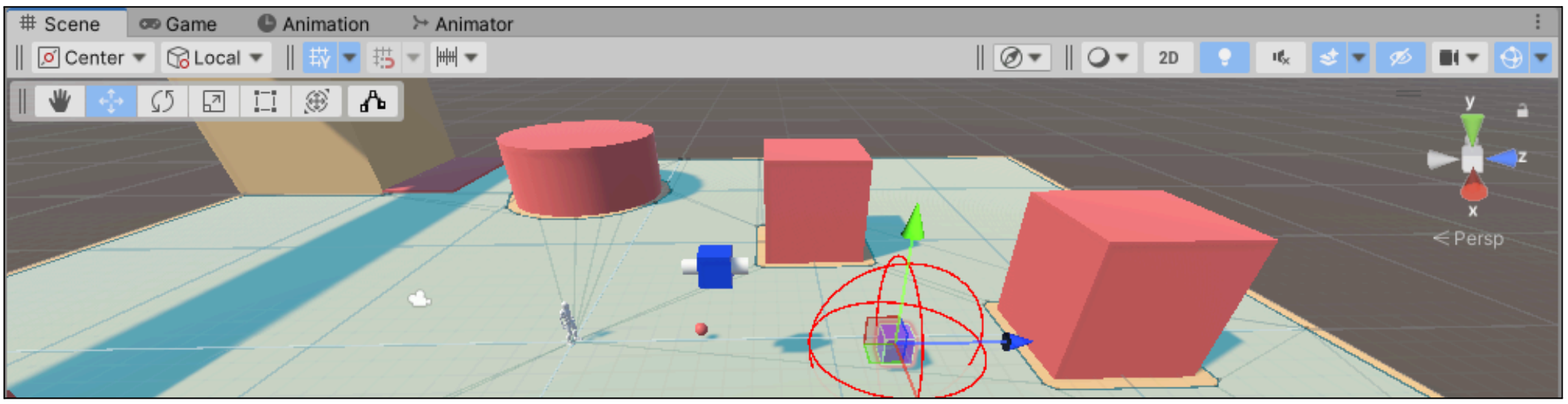
На сцену добавлен объект - куб (переименованный в Item), с которым будет взаимодействовать персонаж на сцене:




Для него и всех объектов, с которыми будет возможно взаимодействие создан скрипт **Interactable**:

```
using UnityEngine;

public class Interactable : MonoBehaviour
{
    public float interactRadius = 3f;

    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, interactRadius); // отобразить сферу gizmos вокруг объекта при просмотре сцены
    }
}
```

При добавлении скрипта объекту радиус gizmos удобно отображается на сцене - это будет зоной активации взаимодействия с объектом:




Для обеспечения взаимодействия надо внести изменения в созданные ранее скрипты, добавив немного логики и геометрии.
Скрипт **AgentEngine**:

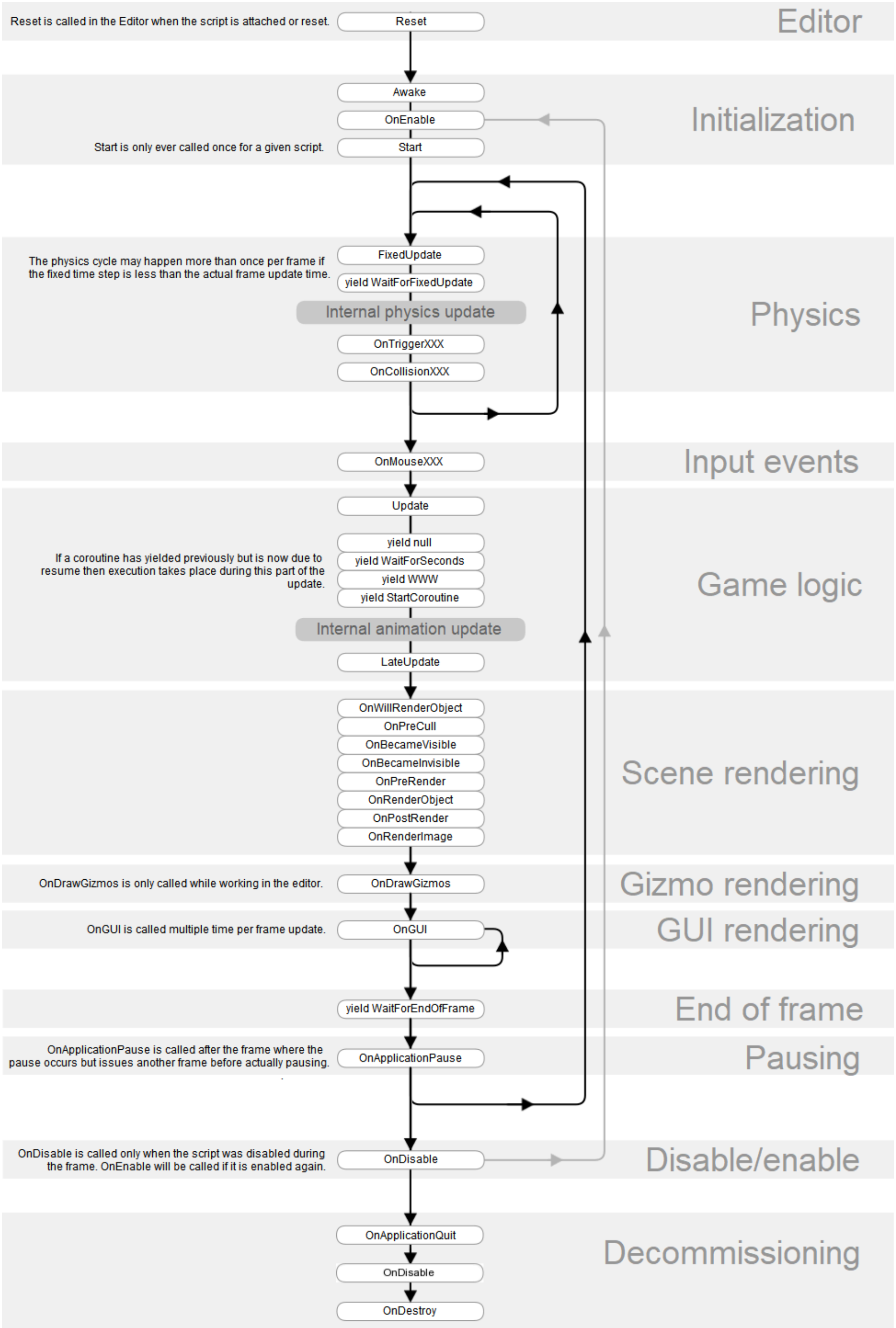
```
using UnityEngine;
using UnityEngine.AI;

[RequireComponent(typeof(NavMeshAgent))]

public class AgentEngine : MonoBehaviour
{
    [SerializeField] private LayerMask movableMask;
    private NavMeshAgent agent;
    private Transform target;

    private void Start()
    {
        agent = GetComponent<NavMeshAgent>();
    }

    private void Update() // правильнее делать через корутину - чтобы реже запускать
    {
        if (target != null)
        {
            agent.SetDestination(target.position); // двигаться к цели
            LookAtTarget(); // смотреть на цель
        }
    }

    public void MoveToPoint(Vector3 point)
```

```
    {
        agent.SetDestination(point); // двигаться в точку
    }

    public void FollowTarget(Interactable newTarget) // двигаться к цели
    {
        agent.stoppingDistance = newTarget.interactRadius;
        agent.updateRotation = false;
        target = newTarget.transform;
    }

    public void StopFollowingTarget() // прекратить движение к цели
    {
        agent.stoppingDistance = 0;
        agent.updateRotation = true;
        target = null;
    }

    private void LookAtTarget() // стандартный способ поворачивать объект в сторону цели
    {
        Vector3 direction = (target.position - transform.position).normalized; // normalized ограничивает скорость
        Quaternion lookRotation = Quaternion.LookRotation(new Vector3 (direction.x, 0, direction.z));
        transform.rotation = Quaternion.Slerp(transform.rotation, lookRotation, Time.deltaTime * 3f); // плавный поворот
    }
}
```

Скрипт **AgentNavigator**:

```
using UnityEngine;
[RequireComponent(typeof(AgentEngine))]

public class AgentNavigator : MonoBehaviour
{
    private AgentEngine agent;
    [SerializeField] private Camera camera;
    [SerializeField] private LayerMask movableMask;
    [SerializeField] private Interactable focus;

    private void Start()
    {
        agent = GetComponent<AgentEngine>();
    }

    private void Update()
    {
        if (Input.GetMouseButtonDown(0)) // 0 левая кнопка мыши
        {
            Ray ray = camera.ScreenPointToRay(Input.mousePosition); // точка к лучу
            RaycastHit hit; // для расчета столкновения луча с препятствиями
            // движение будет работать только если луч направлен в объект слоя movableMask (= GROUND)
            if (Physics.Raycast(ray, out hit, 100, movableMask)) // 100 ограничение дальности для построения луча
            {
                agent.MoveToPoint(hit.point);
                RemoveFocus(); // удаление фокуса
            }
        }
        else if (Input.GetMouseButtonDown(1)) // 1 правая кнопка мыши
        {
            Ray ray = camera.ScreenPointToRay(Input.mousePosition); // точка к лучу
            RaycastHit hit; // для расчета столкновения луча с препятствиями
            // если клик правой кнопкой на интерактивном объекте, то начнется движение к объекту (объект будет взят в фокус)
            if (Physics.Raycast(ray, out hit, 100)) // 100 ограничение дальности для построения луча
            {
                var interactable = hit.collider.GetComponent<Interactable>();

                if (interactable != null)
                    SetFocus(interactable); // фокус на интерактивном объекте
            }
        }
    }

    private void SetFocus(Interactable newFocus) // фокус на интерактивном объекте
    {
        focus = newFocus;
        agent.FollowTarget(newFocus);
    }

    private void RemoveFocus() // удаление фокуса
    {
        focus = null;
        agent.StopFollowingTarget();
    }
}
```

## Блок-схема жизненного цикла скрипта в Unity



## Порядок выполнения функций событий

В скриптинге Unity есть некоторое количество функций события, которые исполняются в заранее заданном порядке по мере исполнения скрипта. Этот порядок исполнения описан ниже:

**※ Редактор**

- **Reset():** Reset (сброс) вызывается для инициализации свойств скрипта, когда он только присоединяется к объекту и тогда, когда используется команда *Reset*. Используется редко.

**※ Первая загрузка сцены**

Функции вызываются при запуске сцены (один раз для каждого объекта на сцене).

- **Awake():** Всегда вызывается до любых функций **Start** и после того, как префаб был вызван в сцену (если **GameObject** не активен на момент старта, **Awake** не будет вызван, пока **GameObject** не будет активирован, или функция в каком-нибудь прикрепленном скрипте не вызовет **Awake**).
- **OnEnable():** Вызывается только если объект активен - сразу после включения объекта (при загрузке уровня).
- **OnLevelWasLoaded():** Вызывается, чтобы проинформировать, что был загружен новый уровень игры.

Для объектов, добавленных в сцену сразу, функции **Awake** и **OnEnable** для всех скриптов будут вызваны до вызова **Start**, **Update** и т.д. Естественно, для объектов вызванных во время игрового процесса такого не будет.

**※ Перед первым обновлением кадра**

- **Start():** Вызывается до обновления первого кадра (first frame) только если скрипт включен.

Для объектов добавленных на сцену, функция **Start** будет вызываться во всех скриптах до функции Update. Естественно, это не может быть обеспечено при создании объекта непосредственно во время игры.

**※ Между кадрами**

- **OnApplicationPause():** Вызывается в конце кадра, во время которого вызывается пауза, что эффективно между обычными обновлениями кадров. Один дополнительный кадр будет выдан после вызова **OnApplicationPause**, чтобы позволить игре отобразить графику, которая указывает на состояние паузы.

**※ Порядок обновления**

Когда вы отслеживаете игровую логику и взаимодействия, анимации, позиции камеры и т.д. есть несколько разных событий, которые вы можете использовать. По общему шаблону, большая часть задач выполняется внутри функции Update, но есть также еще другие функции, которые вы можете использовать.

- **FixedUpdate():** Зачастую случается, что **FixedUpdate** вызывается чаще чем **Update**. **FixedUpdate** может быть вызван несколько раз за кадр, если FPS низок и функция может быть и вовсе не вызвана между кадрами, если FPS высок. Все физические вычисления и обновления происходят сразу после **FixedUpdate**. При применении расчетов передвижения внутри **FixedUpdate**, вам не нужно умножать ваши значения на **Time.deltaTime**, потому что **FixedUpdate** вызывается в соответствии с надёжным таймером, не зависящим от частоты кадров.
- **Update():** **Update** вызывается раз за кадр. Это основная используемая функция, большая часть задач выполняется в ней.
- **LateUpdate():** **LateUpdate** вызывается раз в кадр, после завершения **Update**. Любые вычисления произведенные в **Update** будут уже выполнены на момент начала **LateUpdate**. Часто **LateUpdate** используют для преследующей камеры от третьего лица. Если вы перемещаете и поворачиваете персонажа в **Update**, вы можете выполнить все вычисления перемещения и вращения камеры в **LateUpdate**. Это обеспечит то, что персонаж будет двигаться до того, как камера отследит его позицию.

**※ Рендеринг**

- **OnPreCull():** Вызывается до того, как камера отсекает сцену. Отсечение определяет, какие объекты будут видны в камере. **OnPreCull** вызывается прямо перед тем, как начинается отсечение.
- **OnBecameVisible() / OnBecameInvisible():** вызывается когда объект становится видимым/невидимым любой камере.
- **OnWillRenderObject():** Вызывается **один раз** для каждой камеры, если объект в поле зрения.
- **OnPreRender():** Вызывается перед тем, как камера начнёт рендерить сцену.
- **OnRenderObject():** Вызывается, после того, как все обычные рендеры сцены завершатся. Можно использовать класс GL или Graphics.DrawMeshNow, чтобы рисовать пользовательскую геометрию в данной точке.
- **OnPostRender():** Вызывается после того, как камера завершит рендер сцены.
- **OnRenderImage() (только в Pro версии):** Вызывается после завершения рендера сцены, для возможности пост-обработки изображения экрана.
- **OnGUI():** Вызывается несколько раз за кадр и отвечает за элементы интерфейса (GUI). Сначала обрабатываются события макета и раскраски, после чего идут события клавиатуры/мышки для каждого события. Старая система, сейчас почти не используется.
- **OnDrawGizmos():** Используется для отрисовки гизмо в окне Scene View (в редакторе) в целях визуализации для удобства разработчиков.

**※ Сопрограммы (Coroutines)**

Нормальные обновления сопрограмм запускаются после завершения из функции **Update**. Сопрограмма это функция, которая приостанавливает свое выполнение (yield), пока данные YieldInstruction не завершатся. Разные способы использования сопрограмм:

- **yield** Сопрограмма продолжит выполнение, после того, как все **Update** функции были вызваны в следующем кадре.
- **yield WaitForSeconds** Продолжает выполнение после заданной временной задержки, и после всех **Update** функций, вызванных в итоговом кадре.
- **yield WaitForFixedUpdate** Продолжает выполнение после того, как все функции **FixedUpdate** были вызваны во всех скриптах
- **yield WWW** продолжает выполнение после завершения WWW-загрузки.
- **yield StartCoroutine** сцепляет сопрограмму, и будет ждать, пока не завершится сопрограмма MyFunc.

**※ При уничтожении объекта**

- **OnDestroy():** Вызывается после всех обновлений кадра в последнем кадре объекта, пока он ещё существует (объект может быть уничтожен при помощи Object.Destroy / Destroy(Object) или при закрытии сцены).

**※ При выходе**

Эти функции вызываются во всех активных объектах в вашей сцене:

- **OnApplicationQuit():** Вызывается для всех игровых объектов перед тем, как приложение закрывается. В редакторе вызывается тогда, когда игрок останавливает игровой режим. В веб-плеере вызывается по закрытия веб окна.
- **OnDisable():** Эта функция вызывается, когда объект отключается или становится неактивным.