**ТЕМЫ ДЛЯ ОБСУЖДЕНИЯ НА КОЛЛОКВИУМЕ**

**< 1-10 >**

1. **ПРОКОММЕНТИРОВАТЬ КОД ВРАЩЕНИЯ КАМЕРЫ ВОКРУГ УСТАНОВКИ**

**public class Scroll : MonoBehaviour**

**{**

**[SerializeField]**

**Transform targetPos;**

**int sensivity = 3;**

**int maxdistance = 20;**

**int mindistance = 1;**

**void Update()**

**{**

**if (Input.GetMouseButton(1))**

**{**

**transform.RotateAround(targetPos.position, Vector3.up, Input.GetAxis("Mouse X")\*sensivity);**

**}**

1**. `public class Scroll : MonoBehaviour`**: Эта строка объявляет открытый класс с именем "Scroll", который наследуется от MonoBehaviour. Это означает, что этот скрипт может быть присоединен к объектам в Unity и управлять ими.

2. **`[SerializeField] Transform targetPos;**`: Это объявляет переменную `**targetPos**` типа `**Transform**`. Она помечена атрибутом **`[SerializeField]`,** что позволяет редактировать ее значение в инспекторе Unity. Это будет объект, вокруг которого будет вращаться текущий объект.

3. `**int sensivity = 3;**`: Эта строка объявляет целочисленную переменную `**sensivity**` и устанавливает ей значение 3. Это будет чувствительность (скорость вращения) объекта при вращении вокруг `**targetPos**`.

4. `**int maxdistance = 20**;`: Эта строка объявляет переменную `**maxdistance**` и устанавливает ей значение 20. Это максимальное расстояние, на которое объект может удалиться от `**targetPos**`.

5. `**int mindistance = 1**;`: Эта строка объявляет переменную `**mindistance**` и устанавливает ей значение 1. Это минимальное расстояние, на которое объект может приблизиться к `**targetPos**`.

6. `**void Update()**`: Это объявление метода `**Update()**`, который вызывается каждый кадр в Unity.

7. `**if (Input.GetMouseButton(1))**`: Это условие проверяет, нажата ли правая кнопка мыши. Если это условие истинное, код ниже будет выполняться.

8. `**transform.RotateAround(targetPos.position, Vector3.up, Input.GetAxis("Mouse X") \* sensivity);**`: Эта строка вращает текущий объект (определенный как `transform`) вокруг `**targetPos**` вокруг оси `**Vector3.up` (вертикальная ось)** с учетом значения `**Input.GetAxis("Mouse X") \* sensivity`. `Input.GetAxis("Mouse X")**` возвращает изменение положения мыши по горизонтали, а `**sensivity**` управляет скоростью вращения.

1. **ПРОКОММЕНТИРОВАТЬ КОД ДВИЖЕНИЯ КАМЕРЫ В ГРАНИЦАХ ПОМЕЩЕНИЯ**

**…**

**bool ControlDistance (float distance)**

**{**

**if (distance > mindistance && distance < maxdistance) return true;**

**return false;**

**}**

**float x = Input.GetAxis("Horizontal");**

**float y = Input.GetAxis("Vertical");**

**if (x != 0 || y != 0)**

**{**

**Vector3 newpos = transform.position + (transform.TransformDirection(new Vector3(x, 0, 0)) + Vector3.up \* y) / sensivity;**

**if (ControlDistance(Vector3.Distance(newpos, targetPos.position))) transform.position = newpos;**

**}**

1. **bool ControlDistance(float distance)**: Это объявление функции **ControlDistance**. Она принимает аргумент **distance**, который предположительно представляет расстояние между текущим объектом и **targetPos**.
2. **{**: Начало блока кода функции **ControlDistance**.
3. **if (distance > mindistance && distance < maxdistance) return true;**: Это условие проверяет, находится ли значение **distance** между **mindistance** и **maxdistance**. Если это верно, функция возвращает **true**. В противном случае, она перейдет к следующей строке.
4. **return false;**: Если условие в пункте 3 ложное (расстояние находится за пределами диапазона), функция вернет **false**.
5. **}**: Конец блока кода функции **ControlDistance**.
6. **float x = Input.GetAxis("Horizontal");**: Эта строка получает ввод по горизонтали от пользователя (обычно клавиши A и D или стрелки) и сохраняет его в переменной **x**.
7. **float y = Input.GetAxis("Vertical");**: Эта строка получает ввод по вертикали от пользователя (обычно клавиши W и S или стрелки) и сохраняет его в переменной **y**.
8. **if (x != 0 || y != 0)**: Это условие проверяет, происходит ли какое-либо движение (какое-либо значение **x** или **y** не равно нулю).
9. **Vector3 newpos = transform.position + (transform.TransformDirection(new Vector3(x, 0, 0)) + Vector3.up \* y) / sensivity;**: Эта строка вычисляет новую позицию объекта, добавляя смещение в горизонтальной плоскости (**x**) и вертикальной плоскости (**y**) к текущей позиции объекта. Это смещение затем делится на **sensivity**.
10. **if (ControlDistance(Vector3.Distance(newpos, targetPos.position))) transform.position = newpos;**: Это условие вызывает функцию **ControlDistance** с расстоянием между **newpos** и **targetPos.position**. Если **ControlDistance** возвращает **true**, то новая позиция объекта **transform.position** обновляется на **newpos**, и объект перемещается в новую позицию.

В итоге, этот код обрабатывает ввод пользователя, проверяет расстояние между объектом и **targetPos**, и перемещает объект только в том случае, если расстояние находится в указанном диапазоне.

1. **ПРОКОММЕНТИРОВАТЬ КОД ПРИБЛИЖЕНИЯ И УДАЛЕНИЯ КАМЕРЫ ОТ УСТАНОВКИ**

**…**

**if (Input.GetAxis("Mouse ScrollWheel") != 0)**

**{**

**Vector3 newpos = transform.position + transform.TransformDirection(Vector3.forward \* Input.GetAxis("Mouse ScrollWheel") \* scrollSpeed);**

**if (ControlDistance(Vector3.Distance(newpos, targetPos.position))) transform.position = newpos;**

**}**

1. **if (Input.GetAxis("Mouse ScrollWheel") != 0)**: Это условие проверяет, был ли выполнен скроллинг колеса мыши. Функция **Input.GetAxis("Mouse ScrollWheel")** возвращает значение скролла колеса мыши, и это условие выполняется, если значение не равно нулю, то есть колесо мыши было прокручено.
2. **Vector3 newpos = transform.position + transform.TransformDirection(Vector3.forward \* Input.GetAxis("Mouse ScrollWheel") \* scrollSpeed);**: Здесь вычисляется новая позиция объекта. Сначала берется текущая позиция объекта (**transform.position**), затем к ней добавляется смещение в направлении **Vector3.forward** (вперед) с учетом значения скролла колеса мыши (**Input.GetAxis("Mouse ScrollWheel")**) и коэффициента **scrollSpeed**.
3. **if (ControlDistance(Vector3.Distance(newpos, targetPos.position))) transform.position = newpos;**: Это условие вызывает функцию **ControlDistance** с расстоянием между новой позицией **newpos** и **targetPos.position**. Если **ControlDistance** возвращает **true**, то новая позиция объекта **transform.position** обновляется на **newpos**, и объект перемещается в новую позицию.

Этот код позволяет изменять позицию объекта в зависимости от вращения колеса мыши вперед или назад. При этом позиция обновляется только в том случае, если расстояние между объектом и **targetPos** находится в заданном диапазоне, что контролируется функцией **ControlDistance**.

1. **ПРОЦЕДУРА ПРОГРАММИРОВАНИЯ ИНТЕРАКТИВНОГО UI С ИСПОЛЬЗОВАНИЕМ КНОПОК BUTTON**
2. **…**
3. **…**
4. **…**
5. **…**
6. **…**
7. **… (использование других событий для кнопки?)**

Программирование интерактивного взаимодействия пользователя с элементами **UI** сводится к стандартной процедуре, общей для всех элементов:

1. В сцене на основе **CANVAS** создается рабочий **UI**-объект и кнопка **Button**.
2. К созданному **UI**-объекту добавляется сценарий, который будет вызываться при обращении к нему при совершении событий, доступных для кнопок.
3. В сценарий необходимо добавить директиву **using UnityEngine.UI;** для работы с кодом **UI.**
4. Для кнопки выбирается доступное для ее обработки событие и затем создается ее связь с **UI**-объектом, к которому присоединен разработанный предварительно соответствующий **сценарий**.
5. Для кнопки выбирается в качестве функции имя сценария, добавленного для **UI**-объекта, и в раскрывающемся списке функций указывается та функция в сценарии, которая должна выполнить действие при совершении события с кнопкой.

По умолчанию у кнопки доступно событие щелчок мышью - **Click()**, для использования других событий, например, надвижение курсора мыши **PointerEnter()**, необходимо к кнопке добавить компоненту **Event/Event Trigger** и затем выбрать из предложенного списка нужное событие.

По умолчанию кнопка реагирует только на событие **OnClick**, в то время как UI-элементы на CANVAS могут отвечать на разные варианты взаимодействий. Событие **OnClick** кнопки отвечает только на полноценный щелчок - кнопка мыши нажимается, а затем отпускается.

Для программирования взаимодействий, отличных от **OnClick**, используется компонент **EventTrigger** - добавим **Add Component** к кнопке новый компонент **Event,** а в меню этого компонента выберем вариант **EventTrigger**. Щелкнув на кнопке **Add New Event Type** раскроем список из 17 различных событий для выбора и добавления их к компоненту **EventTrigger.**

Например, для программирования реакции на надвижение курсора мыши выберем вариант **PointerEnter**. Появится пустое поле для этого события, полностью аналогичное полю для события OnClick. Щелкните на кнопке со значком + (плюс), чтобы добавить элемент, и перетащите на этот элемент объект со скриптом. Последовательность действий будет той же, что и для события **OnClick**, просто на этот раз мы укажем реакцию на другое событие – **PointerEnter**.

*Для использования переменных, связываемых с элементами* ***Canvas*** *необходимо в список директив скрипта добавить модуль* ***using UnityEngine.UI;***

Естественно, для выполнения соответствующих действий при наведении курсора мыши в скрипт обрабатываемого объекта необходимо добавить новый метод (функцию), например,с именем **OnPointer()**:

**using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
using UnityEngine.UI;                 //  ИМПОРТ МОДУЛЯ ДЛЯ РАБОТЫ С КОДОМ UI  
  
public class qwest1 : MonoBehaviour  
{**

**[SerializeField]   
   private Text message;  // объявить переменную и связать ее с на текстовым окном**

**public void OnPointer () {**

**message.text = "Установить начальные значения для всех элементов симулятора установки";**

**}**

**}**

Наконец, в списке функций, доступных из скрипта обрабатываемого объекта, необходимо выбрать функцию (метод) **OnPointer ().**

1. **ПРОКОММЕНТИРОВАТЬ КОД ВЫВОДА СООБЩЕНИЯ В ТЕКСТОВОЕ ПОЛЕ ПОСЛЕ ЩЕЛЧКА ПО КНОПКЕ**

**using System.Collections;**

**using UnityEngine;**

**using UnityEngine.UI;**

**public class InText : MonoBehaviour {   
[SerializeField]**

**Text message;**

**public void OnPointer() {**

**message.text = "Установить начальные значения";**

**}**

**}**

Этот код создает скрипт с именем "InText" в Unity, который связывается с объектом, содержащим компонент Text, и изменяет текст, отображаемый в компоненте Text, при вызове метода **OnPointer()**. Давайте разберем его построчно:

1. **using System.Collections;**: Это директива using, которая добавляет пространство имен System.Collections, позволяя использовать классы и объекты из этого пространства имен в коде.
2. **using UnityEngine;**: Это директива using, которая добавляет пространство имен UnityEngine, позволяя использовать классы и объекты из этого пространства имен в коде.
3. **using UnityEngine.UI;**: Это директива using, которая добавляет пространство имен UnityEngine.UI, позволяя использовать классы и объекты из этого пространства имен в коде. В данном случае, это связано с использованием компонента Text.
4. **public class InText : MonoBehaviour {**: Это объявление класса с именем "InText", который наследуется от MonoBehaviour. Этот класс будет присоединен к объекту в Unity и управлять его поведением.
5. **[SerializeField] Text message;**: Это объявление переменной **message** типа Text и помеченной атрибутом **[SerializeField]**. Это означает, что вы сможете настроить значение этой переменной в инспекторе Unity, присоединив объект с компонентом Text к скрипту.
6. **public void OnPointer() {**: Это объявление метода **OnPointer()**. Этот метод будет вызываться при определенных событиях, например, когда пользователь взаимодействует с объектом, к которому присоединен этот скрипт.
7. **message.text = "Установить начальные значения";**: Внутри метода **OnPointer()** устанавливается текст компонента Text **message** на "Установить начальные значения". Это изменяет текст, отображаемый в объекте с компонентом Text, когда метод **OnPointer()** вызывается.

Этот скрипт, следовательно, используется для обновления текста в компоненте Text объекта при взаимодействии с этим объектом.

1. **ПРОЦЕДУРА СОЗДАНИЯ «ВСПЛЫВАЮЩЕГО» ТЕКСТОВОГО ОКНА ПРИ НАВЕДЕНИИ КУРСОРА НА КНОПКУ**

**1. …**

**2. …**

**3. …**

**4. …**

1. Создать UI-объект – окно, которое должно всплывать и расположить его на CANVAS.
2. Задать следующий скрипт для всплывающего окна.

**using UnityEngine;**

**using System.Collections;**

**public class SettingsPopup : MonoBehaviour {**

**void Start() {    
        Close();**//  ЗАКРЫТЬ ВСПЛЫВАЮЩЕЕ ОКНО ПРИ ЗАПУСКЕ ПРОГРАММЫ**}   
  
    public void Open() {  
        gameObject.SetActive(true);**//   АКТИВИРОВАТЬ ОБЪЕКТ, ЧТОБЫ ОТКРЫТЬ ОКНО.**}    
     public void Close() {  
        gameObject.SetActive(false);**// ДЕАКТИВИРОВАТЬ ОБЪЕКТ, ЧТОБЫ ЗАКРЫТЬ ОКНО.**}**

**}**

1. Создать на CANVAS две кнопки, соответственно для открытия и закрытия окна при щелчке по кнопкам мышью.
2. Назначить кнопкам общий объект обработки – всплывающее окно и соответственно выбрать в связанном с окном скрипте **SettingsPopup** в списке доступных в скрипте функций функцию **Open()**для кнопки открытия окна и **Close()**для кнопки закрытия окна.
3. **ПРОКОММЕНТИРОВАТЬ КОД РАБОТЫ «ВСПЛЫВАЮЩЕГО» ОКНА**

**using UnityEngine;**

**using System.Collections;**

**public class SettingsPopup : MonoBehaviour {**

**void Start() {**    
**Close();    
    }   
  
    public void OnOpenSettings() {    
        Open();    
    }  
  
    public void Open() {  
        gameObject.SetActive(true);    
    }    
    public void Close() {  
        gameObject.SetActive(false);**

**}**

**}**

1. **using UnityEngine;**: Эта директива using добавляет пространство имен UnityEngine, что позволяет использовать классы и объекты из этого пространства имен в коде.
2. **using System.Collections;**: Эта директива using добавляет пространство имен System.Collections, что позволяет использовать классы и объекты для работы с коллекциями в коде.
3. **public class SettingsPopup : MonoBehaviour {**: Это объявление класса "SettingsPopup", который наследуется от MonoBehaviour. Этот класс будет присоединен к объекту в Unity и управлять его поведением.
4. **void Start() {**: Это метод "Start()", который автоматически вызывается при запуске объекта, к которому прикреплен этот скрипт.
5. **Close();**: В методе "Start()" вызывается метод "Close()", который скрывает попап (устанавливает **gameObject** в неактивное состояние).
6. **public void OnOpenSettings() {**: Это публичный метод "OnOpenSettings()", который будет вызван при взаимодействии с объектом для открытия настроек.
7. **Open();**: В методе "OnOpenSettings()" вызывается метод "Open()", который открывает попап (устанавливает **gameObject** в активное состояние).
8. **public void Open() {**: Это публичный метод "Open()", который активирует (показывает) попап, устанавливая **gameObject** в активное состояние.
9. **gameObject.SetActive(true);**: В методе "Open()" используется **gameObject.SetActive(true);** для установки объекта в активное состояние, что делает его видимым.
10. **public void Close() {**: Это публичный метод "Close()", который скрывает попап, устанавливая **gameObject** в неактивное состояние.
11. **gameObject.SetActive(false);**: В методе "Close()" используется **gameObject.SetActive(false);** для установки объекта в неактивное состояние, что делает его невидимым.

Этот скрипт позволяет открывать и закрывать попап с настройками при вызове соответствующих методов. Обычно, такой скрипт используется, чтобы управлять видимостью попапа на основе действий пользователя или других событий в игре.

1. **ПРОКОММЕНТИРОВАТЬ КОД ПОДСВЕТКИ ЭЛЕМЕНТА УСТАНОВКИ ПРИ НАВЕДЕНИИ КУРСОРА МЫШИ НА КНОПКУ С НАЗВАНИЕМ ЭЛЕМЕНТА В СПИСКЕ**

**using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class ChangeColor : MonoBehaviour  
{  
      
    public void ChangCol()  
    {  
        GetComponent<Renderer> ().material.color = new Color (1, 0, 0);     
    }  
  
    public void ChangCol1()  
    {  
        GetComponent<Renderer> ().material.color = new Color (1, 1, 1);     
    }  
}**

1. **using System.Collections;**: Это директива using добавляет пространство имен System.Collections, позволяя использовать классы и объекты для работы с коллекциями в коде.
2. **using System.Collections.Generic;**: Эта директива using добавляет пространство имен System.Collections.Generic, что позволяет использовать обобщенные коллекции в коде.
3. **using UnityEngine;**: Эта директива using добавляет пространство имен UnityEngine, что позволяет использовать классы и объекты из этого пространства имен в коде.
4. **public class ChangeColor : MonoBehaviour**: Это объявление класса "ChangeColor", который наследуется от MonoBehaviour. Этот класс будет присоединен к объекту в Unity и управлять его поведением.
5. **public void ChangCol()**: Это публичный метод "ChangCol()". Этот метод будет вызываться, чтобы изменить цвет объекта на определенный цвет.
6. **GetComponent<Renderer>().material.color = new Color(1, 0, 0);**: В методе "ChangCol()" используется **GetComponent<Renderer>()** для получения компонента Renderer, присоединенного к этому объекту, а затем устанавливается цвет материала (material) объекта с помощью **new Color(1, 0, 0)**. В данном случае, устанавливается красный цвет (RGB: 1, 0, 0).
7. **public void ChangCol1()**: Это публичный метод "ChangCol1()". Этот метод будет вызываться, чтобы изменить цвет объекта на другой определенный цвет.
8. **GetComponent<Renderer>().material.color = new Color(1, 1, 1);**: В методе "ChangCol1()" также используется **GetComponent<Renderer>()** для получения компонента Renderer и устанавливается цвет материала объекта. В данном случае, устанавливается белый цвет (RGB: 1, 1, 1).

Этот скрипт позволяет изменять цвет объекта, к которому он прикреплен, вызывая соответствующие методы "ChangCol()" и "ChangCol1()".

Начало формы

1. **АЛГОРИТМ ВЫБОРА УДОБНОГО РАКУРСА ДЛЯ КАМЕРЫ НА СЦЕНЕ**
2. **…**
3. **…**
4. **…**
5. **…**

Для получения положения камеры в нужном ракурсе для просмотра соответсвующего элемента установки можно поступать следующим образом:

1. Запомнить значения позиции **Position** и поворота **Rotation** камеры в исходном положении в окне редактора среды **Inspector**.
2. Рассчитать (рационально это сделать вручную на отдельном чертеже-схеме) желаемую позицию и поворот камеры для удобного ракурса просмотра соответсвующего элемента установки – сдвиг рассчитывается как разность значений позиций XYZ между установкой и камерой, а поворот камеры на нужный угол определяется в конструкции кватерниона.

Например, для просмотра элемента на обратной стороне установки требуемый угол поворота будет равен:

**needRotaton = Quaternion.AngleAxis(-180, new Vector3(0,1,0));**

а для просмотра сверху:

**needRotaton = Quaternion.AngleAxis(0, new Vector3(0,1,0));**

1. В скрипте для перемещения камеры в нужные ракурсы задать для каждой кнопки отдельные функции.

Например, для просмотра обратной стороны установки задать функцию **Move()**:

**public void Move()**

**move = true;**

**startPosition = transform.position;**

**startRotation = transform.rotation;**

**needPosition = new Vector3(277.0f, 251, 6);**

**needRotaton = Quaternion.AngleAxis(-180, new Vector3(0,1,0));}**

Здесь булевая переменная **move**используется для запуска движения, а стартовый позиция и угол поворота задаются каждый раз для исходного положения камеры перед ее перемещением.

1. Обеспечить движение камеры от исходного положения к точке с нужным ракурсом просмотра элемента установки с использованием метода **Update()**

**ПРОКОММЕНТИРОВАТЬ КОД ОБРАБОТКИ ЩЕЛЧКОВ МЫШЬЮ ПО 3D-ОБЪЕКТАМ СЦЕНЫ**

**using UnityEngine;**

**using System.Collections;**

**using UnityEngine.EventSystems;**

**public class Script11 : MonoBehaviour, IPointerClickHandler {**

**public void OnPointerClick(PointerEventData eventData) {**

**gameObject.GetComponent<Renderer>().material.color = new Color(1,0,0);**

**}**

**}**

Данный скрипт написан на языке программирования C# и предназначен для использования в среде разработки игр Unity. Давайте рассмотрим его построчно:

**using UnityEngine;**

**using System.Collections;**

**using UnityEngine.EventSystems;**

Эти строки содержат директивы using, которые определяют пространства имен, необходимые для работы с Unity и событиями ввода.

**public class Script11 : MonoBehaviour, IPointerClickHandler {**

Эта строка определяет класс с именем "Script11", который наследует функциональность **MonoBehaviour** (базовый класс для всех скриптов в Unity) и реализует интерфейс **IPointerClickHandler**. Этот интерфейс используется для обработки событий щелчка мыши.

**public void OnPointerClick(PointerEventData eventData) {**

Здесь начинается реализация метода, объявленного в интерфейсе **IPointerClickHandler**. Этот метод вызывается при щелчке мышью на объекте, к которому привязан этот скрипт.

**gameObject.GetComponent<Renderer>().material.color = new Color(1, 0, 0);**

Эта строка изменяет цвет материала объекта на новый цвет (красный в данном случае), когда происходит щелчок мышью.

Итак, данный скрипт реагирует на событие щелчка мышью на объекте, к которому привязан скрипт, и изменяет цвет его материала на красный.

1. **ПРОКОММЕНТИРОВАТЬ КОД ПЕРЕДВИЖЕНИЯ КАМЕРЫ В ЗАДАННУЮ ТОЧКУ**

**public class Replacer : MonoBehaviour**

**{ bool move = true;**

**Vector3 startPosition;**

**Vector3 needPosition;**

**float speed = 0.01f;**

**float offset = 0;**

**Quaternion startRotation;**

**Quaternion needRotaton;**

**public void Move1()   
        {  move = true;  
            startPosition = transform.position;  
            startRotation = transform.rotation;  
            needPosition = new Vector3(277.0f, 251, 6);  
            needRotaton = Quaternion.AngleAxis(-180, new Vector3(0,1,0));        }**

**void Update()**

**{**

**if(move)**

**{**

**offset+=speed;**

**transform.position = Vector3.Lerp(startPosition, needPosition, offset);**

**transform.rotation = Quaternion.Slerp(startRotation, needRotaton, offset);**

**if (offset >= 1)**

**{**

**move = false;**

**offset = 0;**

**}**

**}**

**}**

**}**

1. **public class Replacer : MonoBehaviour**: Это объявление класса "Replacer", который наследуется от MonoBehaviour. Этот класс будет присоединен к объекту в Unity и управлять его поведением.
2. **bool move = true;**: Это булевская переменная "move", которая используется для управления движением объекта. Изначально установлена в **true**, что означает, что объект будет двигаться.
3. **Vector3 startPosition;**: Это переменная "startPosition", которая хранит начальную позицию объекта.
4. **Vector3 needPosition;**: Это переменная "needPosition", которая хранит конечную позицию, к которой объект будет двигаться.
5. **float speed = 0.01f;**: Это переменная "speed", которая устанавливает скорость движения объекта.
6. **float offset = 0;**: Это переменная "offset", которая используется для отслеживания прогресса движения объекта.
7. **Quaternion startRotation;**: Это переменная "startRotation", которая хранит начальное вращение объекта.
8. **Quaternion needRotation;**: Это переменная "needRotation", которая хранит конечное вращение объекта.
9. **public void Move1()**: Это публичный метод "Move1()", который инициирует перемещение объекта. Он устанавливает начальную позицию и вращение объекта, а также конечную позицию и вращение.
10. **void Update()**: Это метод "Update()", который вызывается на каждом кадре в Unity.
11. **if (move)**: Это условие проверяет, разрешено ли движение объекта.
12. **offset += speed;**: При каждом кадре переменная "offset" увеличивается на значение "speed".
13. **transform.position = Vector3.Lerp(startPosition, needPosition, offset);**: Здесь используется функция **Vector3.Lerp()**, чтобы плавно перемещать объект между начальной и конечной позицией в зависимости от "offset".
14. **transform.rotation = Quaternion.Slerp(startRotation, needRotation, offset);**: Здесь используется функция **Quaternion.Slerp()**, чтобы плавно вращать объект между начальным и конечным вращением в зависимости от "offset".
15. **if (offset >= 1)**: Это условие проверяет, достиг ли "offset" значения 1, что означает, что перемещение и вращение объекта завершено.
16. **move = false;**: Если "offset" достиг 1, то движение объекта завершено, и переменная "move" устанавливается в **false**, чтобы предотвратить дополнное движение.
17. **offset = 0;**: Переменная "offset" сбрасывается обратно в ноль, чтобы подготовить объект к возможному будущему перемещению.

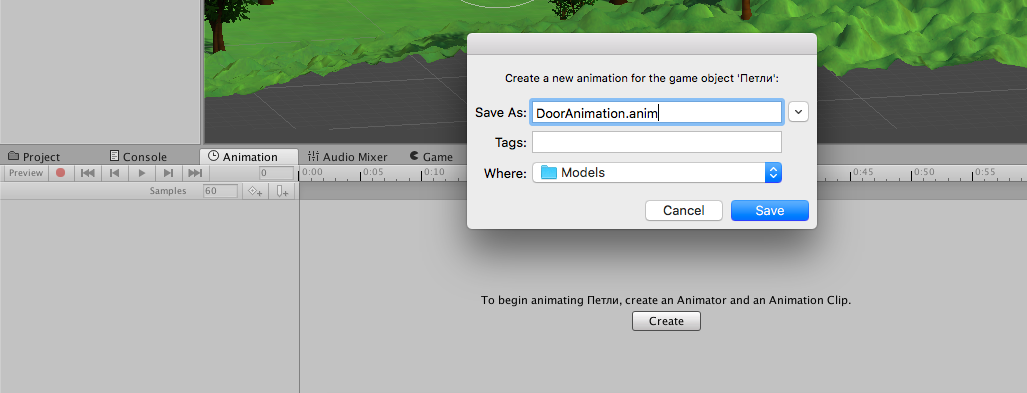
Этот скрипт создает плавное перемещение и вращение объекта между начальной и конечной позицией/вращением при вызове метода "Move1()".

**ВОПРОСЫ ПО АНИМАЦИИ В UNITY**

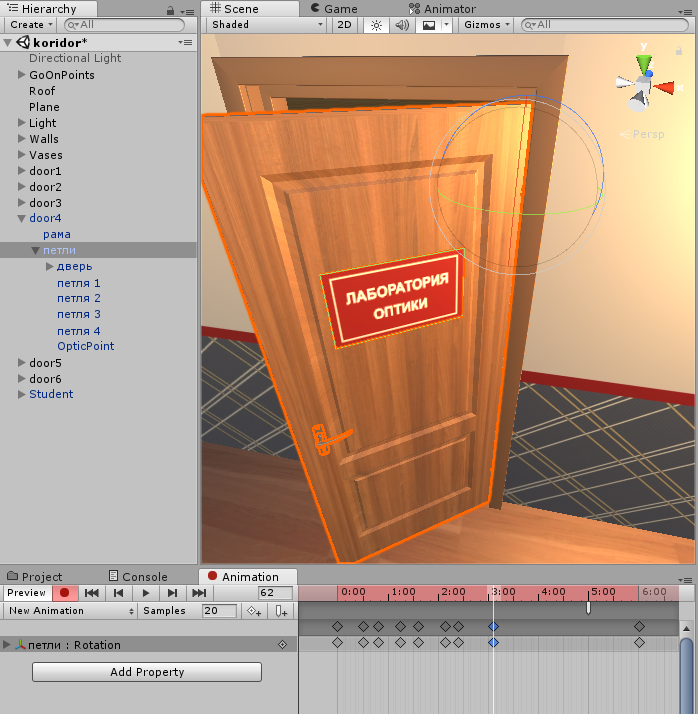
1. **Как создать файл анимации в Unity**

Для создания анимации в редакторе **Unity** нужно выбрать объект, который будет анимироваться, открыть для него вкладку **Window/Animation/Animation** и создать, прежде всего, отдельный файл анимации в открывающемся окне, нажав кнопку **Create**.

После сохранения файла анимации с типом **\*.anim** в папке проекта **Assets** создается в дальнейшем содержание анимации объекта.



На временной шкале в нижней части окна редактора **Unity** нажать кнопку с красным кружком (на рисунке) и в соответствующих ключевых кадрах записать требуемую анимацию для объекта.

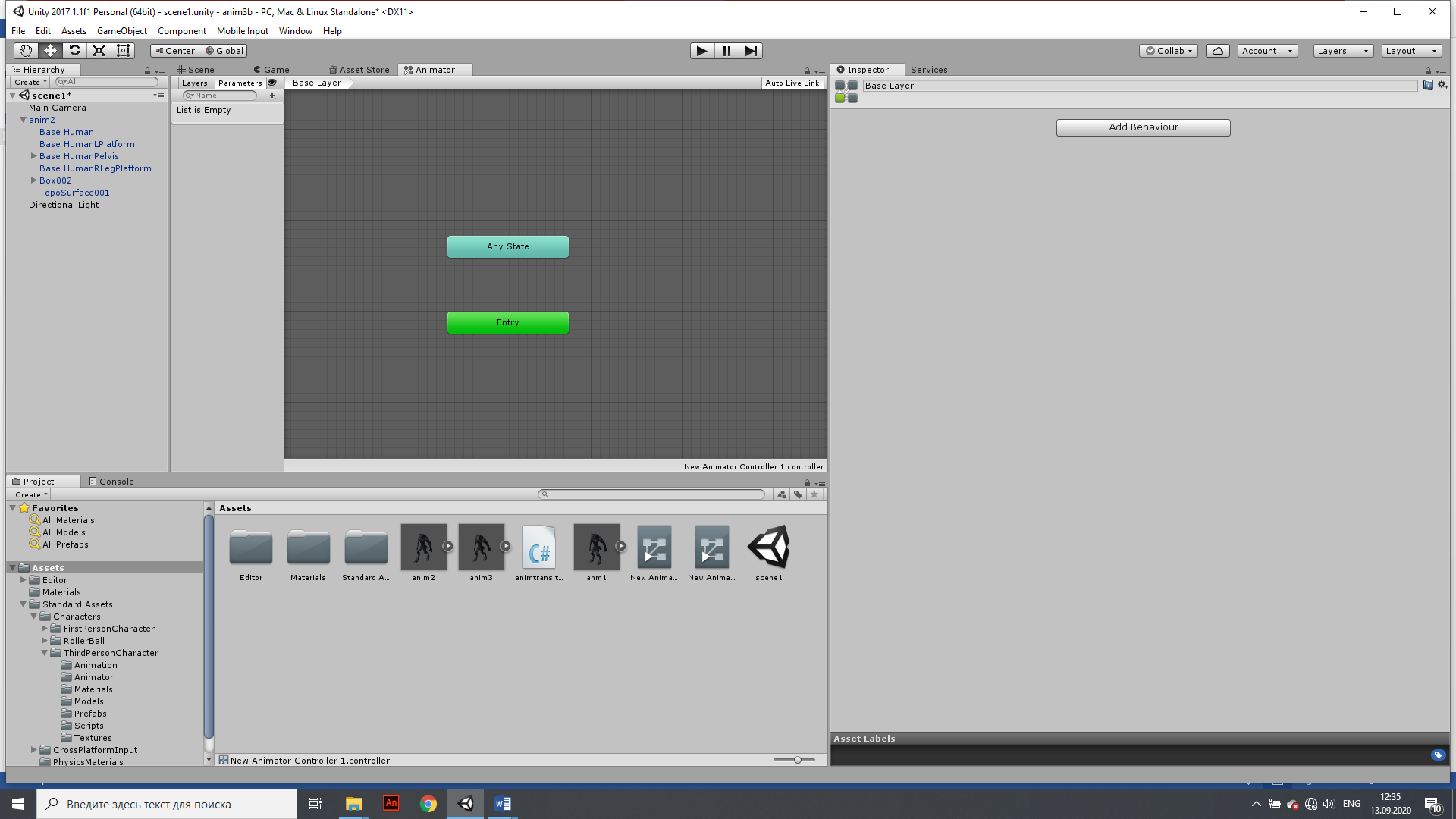


1. **Как добавить к объекту контроллер анимации**

Затем нужно добавить к объекту, контроллер анимации **Animator** – кликнуть правой кнопкой мыши по свободной области в папке **Assets**, выполнить команду **Create->Animator Controller** и дать имя контроллеру. Дважды кликнув на созданный контроллер, перейти в окно редактора.

1. **Как создать и настроить переход к анимации**

В начале есть два блока, это **Entry**, с которого будут начинать проигрываться анимации и **Any State** – особое состояние, которое всегда существует и позволяет перейти от любого состояния к конкретному.



Создать **Empty State** как состояние по умолчанию, щелкнув правой кнопкой мыши в свободной части окна аниматора и выполнив команду **Create State/Empty,** в качестве состояния покоя у двери – это состояние добавится в окно **Animator** и будет по умолчанию оранжевого цвета.

Затем нужно перетащить в окно **Animator** из папки **Assets** созданную анимацию, которая будет представлена серым прямоугольником с именем анимации как отдельное состояние.

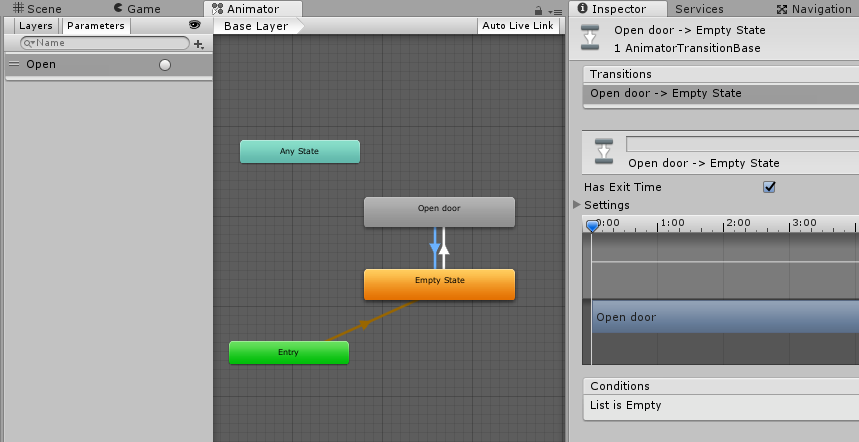
Переходы к состоянию анимации открытия двери и обратно должны происходить по булевой переменной **Bool**, причем обратно - по условию установки режима **Has Exit Time**. При этом предварительно для созданной анимации двери необходимо *снять зацикливание* **Loop Time** в контроллере анимации, после того, как она будет выбрана в папке **Assets**.

1. **Как задать скорость анимации**

Для регулировки скорости воспроизведения анимации необходимо для состояния анимации в окне **Animator** необходимо задать ее величину в окне режима **Speed**, используя движение по нему курсором мыши

1. **Как настроить файл анимации**

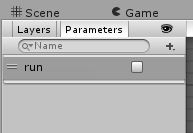
Для создания в окне **Animator** перехода от одного состояния к другому (отображается на экране стрелкой перехода) необходимо щелкнуть правой кнопкой мыши по начальному состоянию и выбрать в контекстном меню команду **Make Transition,** затем навести кусор мыши на прямоугольник другого состояния и щелкнуть по нему мышью, аналогичным образом можно создать обратный переход.



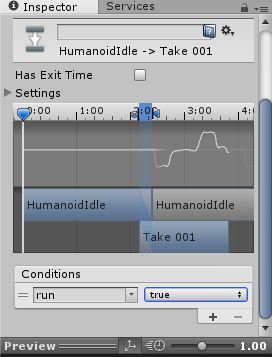
1. **Как создать и использовать переменные для управления анимацией**

Для программного управления анимацией естественно необходимо будет создать программный код на языке C# в скрипте, который должен быть довавлен к анимируемому объекту.

Однако, прежде всего, в окне аниматора необходимо **создать переменную** для перехода к анимации, открыввкладку **Parameters,** затем нажав плюс (добавить), затем выбрав тип создаваемой переменной, например, **Bool** (булевского типа) и, наконец, задав имя переменной, например **run**.



После этого выбрать в окне редактора щелчком мыши стрелку перехода от исходного состояния покоя к состоянию анимации движения. Затем в **Inspector** для этого перехода снять галочку для режима **Has Exit Time** и в списке **Conditions** и добавить в него переменную **run** со значением **true**, нажав кнопку **+**.



Чтобы создать обратный переход к состоянию покоя от анимации необходимо выбрать щелчком мыши стрелку обратного перехода, но теперь установить галочку **Has exit time**, а затем в списке **Conditions** добавить (нажав значок +) переменную **run**, но уже со значением **false**.

1. **Скрипт для переключения между состояниями анимируемого объекта**

В окне инспектора **Inspector** добавить **Add Component** для объекта новый скрипт **New Script** и в редакторе кода создатьследующий код:

**Animator anim;** //переменная типа **Animator** для ссылки на анимацию

**void Start(){**

**anim = GetComponent<Animator>();** //инициализация контроллера анимации

**}**

**void Update()**

**{**

**If(Input.GetKeyDown(KeyCode.Q))** //если нажата клавиша **Q**

**{**

**anim.SetBool(“run”, true);** // переменная, отвечающая за переход имеет значение **true**

**}**

**If(Input.GetKeyDown(KeyCode.E))** //если нажата клавиша **E**

**{**

**anim.SetBool(“run”, false);** // переменная, отвечающая за переход имеет значение **false**

**}**

**}**

1. **Как проиграть анимацию из любого состояния**

Чтобы проиграть какую-либо анимацию можно было из любого состояния, необходимо:

1. создать переход от **Any State** к этой анимации, например с именем **Hit**. Для возвращения в состояние ожидания необходимо создать переход от новой анимации **Hit** к анимации по умолчанию.
2. создать еще одну переменную типа **Trigger** (например **hitten**), которая будет отвечать за щелчок мышью для запуска анимации. Добавим к переходу от **Any State** эту переменную, а для обратного перехода оставим галочку **Has Exit Time**, чтобы переход совершался автоматически (без использования переменной) один раз после проигрывания анимации.
3. добавить в метод **Update()** код проигрывания новой пользовательской анимации по клику мыши:

**if (Input.GetMouseButtonDown(0))  {  anim.SetTrigger("hitten"); }**

**Input.GetMouseButtonDown(0):** Это метод, который возвращает true, если левая кнопка мыши была нажата в текущем кадре. Аргумент 0 указывает на левую кнопку мыши. В Unity левая кнопка мыши имеет индекс 0, правая - 1, средняя (если поддерживается) - 2.

**anim.SetTrigger("hitten"):** Если уsсловие верно (левая кнопка мыши была нажата), то вызывается метод SetTrigger объекта anim. Предполагается, что anim - это объект анимации (Animator). Метод SetTrigger используется для установки триггера анимации, что запускает определенное состояние или переход в анимационном контроллере. В данном случае, триггер с именем "hitten" активируется, что, вероятно, запускает какую-то анимацию связанную с событием удара или попадания.

Начало формы