

6. Скриптинг.

6.1. Создание скриптов.

Поведение игровых объектов контролируется с помощью компонентов (Components), которые присоединяются к ним. Несмотря на то, что встроенные компоненты Unity могут быть очень разносторонними, вскоре вы обнаружите, что вам нужно выйти за пределы их возможностей, чтобы реализовать ваши собственные особенности геймплея. Unity позволяет вам создавать свои компоненты, используя скрипты. Они позволяют активировать игровые события, изменять параметры компонентов, и отвечать на ввод пользователя каким вам угодно способом. Unity изначально поддерживает два языка программирования:

- **C#**, стандартный в отрасли язык подобный Java или C++;
- **UnityScript**, язык, разработанный специально для использования в Unity по образцу JavaScript;

В отличие от других ассетов, скрипты обычно создаются непосредственно в Unity. Вы можете создать скрипт используя меню Create в левом верхнем углу панели Project или выбрав Assets > Create > C# Script (или JavaScript/Boo скрипт) в главном меню. Новый скрипт будет создан в папке, которую вы выбрали в панели Project. Имя нового скрипта будет выделено, предлагая вам ввести новое имя.

После двойного щелчка на скрипте в Unity, он будет открыт в текстовом редакторе. По умолчанию Unity будет использовать MonoDevelop, но вы можете выбрать любой редактор из панели External Tools в настройках Unity.

```

using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }

}

```

Рисунок 32 – Структура «пустого» скрипта

Скрипт взаимодействует с внутренними механизмами Unity за счет создания класса, наследованного от встроенного класса, называемого `MonoBehaviour`. Вы можете думать о классе как о своего рода плане для создания нового типа компонента, который может быть прикреплен к игровому объекту. Каждый раз, когда вы присоединяете скриптовый компонент к игровому объекту, создается новый экземпляр объекта, определенный планом. Имя класса берется из имени, которое вы указали при создании файла. Имя класса и имя файла должны быть одинаковыми, для того, чтобы скриптовый компонент мог быть присоединен к игровому объекту.

Основные вещи, достойные внимания, это две функции, определенные внутри класса. Функция `Update` - это место для размещения кода, который будет обрабатывать обновление кадра для игрового объекта. Это может быть движение, срабатывание действий и ответная реакция на ввод пользователя, в основном всё, что должно быть обработано с течением времени во игровом процессе. Чтобы позволить функции `Update` выполнять свою работу, часто бывает полезно инициализировать переменные, считать свойства и осуществить связь с другими игровыми объектами до того, как будут совершены какие-либо действия. Функция `Start` будет вызвана Unity до начала игрового процесса (т.е. до первого вызова функции `Update`), и это идеальное место для выполнения инициализации.

6.2. Управление `GameObject`.

В редакторе Unity вы изменяете свойства Компонента используя окно Inspector. Так, например, изменения позиции компонента Transform приведет к изменению позиции игрового объекта. Аналогично, вы можете изменить цвет материала компонента Renderer или массу твёрдого тела (Rigidbody) с соответствующим влиянием на отображение или поведение игрового объекта. По большей части скрипты также изменяют свойства компонентов для управления игровыми объектами. Разница, однако, в том, что скрипт может изменять значение свойства постепенно со временем или по получению ввода от пользователя. За счет изменения, создания и уничтожения объектов в заданное время может быть реализован любой игровой процесс.

Наиболее простым и распространенным является случай, когда скрипту необходимо обратиться к другим компонентам, присоединенных к тому же GameObject. Компонент на самом деле является экземпляром класса, так что первым шагом будет получение ссылки на экземпляр компонента, с которым вы хотите работать. Это делается с помощью функции GetComponent. Типично, объект компонента сохраняют в переменную, это делается в C# посредством следующего синтаксиса:

```
void Start () {  
    Rigidbody rb = GetComponent<Rigidbody>();  
}
```

Рисунок 33 – Получение компонента в скрипте

Как только у вас есть ссылка на экземпляр компонента, вы можете устанавливать значения его свойств, тех же, которые вы можете изменить в окне Inspector:

```
void Start () {  
    Rigidbody rb = GetComponent<Rigidbody>();  
  
    // Change the mass of the object's Rigidbody.  
    rb.mass = 10f;  
}
```

Рисунок 34 – Установка значения компонента из скрипта

Имейте ввиду, что нет причины, по которой вы не можете иметь больше одного пользовательского скрипта, присоединенного к одному и тому же объекту. Если вам нужно обратиться к одному скрипту из другого, вы можете использовать, как обычно, GetComponent, используя при этом имя класса скрипта (или имя файла), чтобы указать какой тип Компонента вам нужен.

Пусть иногда они и существуют изолированно, все же, обычно, скрипты отслеживают другие объекты. Например, преследующий враг должен знать позицию игрока. Unity предоставляет несколько путей получения других объектов, каждый подходит для конкретной ситуации. Самый простой способ найти нужный игровой объект - добавить в скрипт переменную типа `GameObject` с уровнем доступа `public`:

```
public class Enemy : MonoBehaviour {
    public GameObject player;

    // Other variables and functions...
}
```

Рисунок 35 – `GameObject` с публичным уровнем доступа

Переменная будет видна в окне `Inspector`, как и любые другие. Теперь вы можете перетащить объект со сцены или из панели `Hierarchy` в эту переменную, чтобы назначить его. Функция `GetComponent` и доступ к переменным компонента доступны как для этого объекта, так и для других, то есть вы можете использовать следующий код:

```
public class Enemy : MonoBehaviour {
    public GameObject player;

    void Start() {
        // Start the enemy ten units behind the player character.
        transform.position = player.transform.position - Vector3.forward * 10f;
    }
}
```

Рисунок 36 – Использование публичной переменной в скрипте

Иногда игровая сцена может использовать несколько объектов одного типа, таких как враги, путевые точки и препятствия. Может возникнуть необходимость отслеживания их в определенном скрипте, который управляет или реагирует на них (например, все путевые точки могут потребоваться для скрипта поиска пути). Можно использовать переменные для связывания этих объектов, но это сделает процесс проектирования утомительным, если каждую новую путевую точку нужно будет перетащить в переменную в скрипте. Аналогично, при удалении путевой точки придется удалять ссылку на отсутствующий объект. В случаях, наподобие этого, чаще всего удобно управлять набором объектов, сделав их дочерними одного родительского объекта. Дочерние объекты могут быть получены, используя компонент `Transform` родителя (так как все игровые объекты неявно содержат `Transform`):

```

using UnityEngine;

public class WaypointManager : MonoBehaviour {
    public Transform[] waypoints;

    void Start() {
        waypoints = new Transform[transform.childCount];
        int i = 0;

        foreach (Transform t in transform) {
            waypoints[i++] = t;
        }
    }
}

```

Рисунок 37 – Трансформирование дочернего объекта

Нахождение игровых объектов в любом месте иерархии доступно всегда, когда у вас есть некоторая информация, по которой их можно идентифицировать. Отдельные объекты могут быть получены по имени, используя функцию `GameObject.Find`:

```

GameObject player;

void Start() {
    player = GameObject.Find("MainHeroCharacter");
}

```

Рисунок 38 – Нахождение объекта по имени

Объект или коллекция объектов могут быть также найдены по их тегу, используя функции `GameObject.FindWithTag` и `GameObject.FindGameObjectsWithTag`:

```

GameObject player;
GameObject[] enemies;

void Start() {
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}

```

Рисунок 39 – Нахождение объекта по тегу

6.3. Функции событий.

Скрипт в Unity не похож на традиционную идею программы, где код работает постоянно в цикле, пока не завершит свою задачу. Вместо этого, Unity периодически передаёт

управление скрипту при вызове определённых объявленных в нём функций. Как только функция завершает исполнение, управление возвращается обратно к Unity. Эти функции известны как функции событий, т.к. их активирует Unity в ответ на события, которые происходят в процессе игры. Unity использует схему именования, чтобы определить, какую функцию вызвать для определённого события. Например, вы уже видели функцию Update (вызывается перед сменой кадра) и функцию Start (вызывается прямо перед первым кадром объекта).

Игра - это что-то вроде анимации, в которой кадры генерируются на ходу. Ключевой концепт в программировании игр заключается в изменении позиции, состояния и поведения объектов в игре прямо перед отрисовкой кадра. Такой код в Unity обычно размещают в функции Update. Update вызывается перед отрисовкой кадра и перед расчётом анимаций.

```
void Update() {  
    float distance = speed * Time.deltaTime * Input.GetAxis("Horizontal");  
    transform.Translate(Vector3.right * distance);  
}
```

Рисунок 39 – Функция Update

Физический движок также обновляется фиксированными по времени шагами, аналогично тому как работает отрисовка кадра. Отдельная функция события FixedUpdate вызывается прямо перед каждым обновлением физических данных. Т.к. обновление физики и кадра происходит не с одинаковой частотой, то вы получите более точные результаты от кода физики, если поместите его в функцию FixedUpdate, а не в Update.

```
void FixedUpdate() {  
    Vector3 force = transform.forward * driveForce * Input.GetAxis("Vertical");  
    rigidbody.AddForce(force);  
}
```

Рисунок 40 – Функция FixedUpdate

Также иногда полезно иметь возможность внести дополнительные изменения в момент, когда у всех объектов в сцене отработали функции Update и FixedUpdate и рассчитались все анимации. В качестве примера, камера должна оставаться привязанной к целевому объекту; подстройка ориентации камеры должна производиться после того, как целевой объект сместился. Другим примером является ситуация, когда код скрипта должен переопределить эффект анимации (допустим, заставить голову персонажа повернуться к

целевому объекту в сцене). В ситуациях такого рода можно использовать функцию LateUpdate.

```
void LateUpdate() {  
    Camera.main.transform.LookAt(target.transform);  
}
```

Рисунок 41 – Функция LateUpdate