Бесплатная электронная книга

учусь unity3d

Free unaffiliated eBook created from
Stack Overflow contributors.

#unity3d

1
1: unity3d 2
2
2
Examples 5
5
5
6
6
6
6
Linux 7
7
8
8
10
2: API CullingGroup 13
13
Examples 13
13
15
16
16
3: Prefabs 18
18
Examples 18
18
18
19
20
20

21
21
4: Raycast 26
26
Examples 26
Raycast 26
2D Raycast2D 27
Raycast 27
28
5: ScriptableObject 29
29
ScriptableObjects AssetBundles 29
Examples 29
29
ScriptableObject 29
ScriptableObject 30
ScriptableObjects PlayMode 30
ScriptableObject 31
6: Unity Animation 33
Examples 33
33
34
2D- Sprite 36
38
7: Unity Profiler 41
41
Profiler 41
Android 41
IOS 42
Examples 42
42

Profiler 42
8: Vector3 44
44
44
Examples 44
44
Vector3.zero Vector3.one 44
45
47
Vector3 47
47
Vector2 Vector4 48
49
Lerp LerpUnclamped 49
MoveTowards 50
SmoothDamp 52
9: 54
54
54
SerializeField 54
Examples 55
55
57
58
59
61
10: 65
65
Examples 65
- 65
11: (VR) 66

Examples....66
VR....66
SDKs:....66
:....66
VR....66
....67
12: 69
Examples....69
GetKey, GetKeyDown GetKeyUp....69
()....70
()....71
()....71
(, ,)....72
13: MonoBehaviour....75
Examples....75
....75
14: (IMGUI)....76
....76
Examples....76
GUILayout....76
15: () 77
....77
....77
Examples....77
....77
....78
16: 82
....82
....82
Examples....82
Unity C #....82
Unity JavaScript....83

17: Git Unity 84
Examples 84
Git (LFS) Unity 84
..... 84
Git & Git-LFS 84
1: Git GUI 84
2: Git & Git-LFS 85
Git 85
Git Unity 85
Unity Ignore Folders 85
Unity 86
..... 86
..... 87
18: 89
Examples 89
..... 89
.unitypackage 89
19: 91
..... 91
Examples 91
Quaternion vs Euler 91
..... 91
20: 93
Examples 93
..... 93
..... 93
..... 93
..... 94
..... 94
..... 94
..... 95

95
96
96
96
96
97
97
97
98
99
99
102
102
-102
102
21: 103
Examples 103
103
103
104
104
105
22: 106
106
Examples 106
106
TouchPhase 106
23: 108
Examples 108
108
108

24: **110**

Examples 110

..... 110

..... 113

..... 114

25: **116**

..... 116

Examples 116

..... 116

/ **116**

..... **116**

..... **116**

Coroutine 117

..... **117**

..... **117**

..... **117**

..... **118**

..... 118

..... **118**

..... 119

Debug 119

..... **120**

..... 121

, ... 122

..... 122

26: **123**

Examples 123

..... 123

..... **123**

..... **123**

124
125
126
127
27: Android 101 - 129
129
129
Android- 129
129
130
Examples 130
UnityAndroidPlugin.cs 130
UnityAndroidNative.java 130
UnityAndroidPluginGUI.cs 131
28: GameObjects 132
132
132
132
132
Examples 133
GameObject 133
GameObject 133
133
GameObjects MonoBehaviour 133
GameObjects 134
29: (UI) 135
Examples 135
135
135
30: 137
137

137

Examples 137

137

140

143

Gizmos 147

1 147

149

150

150

151

152

? 152

WindowWindow 152

152

154

156

SceneView 156

31: 161

Examples 161

161

101 161

161

162

163

32: 164

Examples 164

164

( ) 164

( ) 165

Zip- 165

166

33: 169

..... 169

Unity 169

Examples 170

, ... 170

, ... 170

..... 170

..... 172

34: 174

..... 174

..... 174

Examples 175

RuntimeInitializeOnLoadMethodAttribute 175

Singleton MonoBehaviour Unity C # 176

Singleton 176

Singleton 179

Singleton Entity-Componity Unitys 180

MonoBehaviour & ScriptableObject Singleton Class 181

35: 186

Examples 186

..... 186

LayerMask 186

36: 188

..... 188

..... 188

..... 188

YieldInstructions 188

Examples 188

..... 189

..... 190

.....

191

MonoBehaviour, Coroutines 192

192

194

37: **197**

197

Examples 197

197

197

198

198

GameObjects : 199

GameObject 199

GameObject 200

200

38: **202**

202

Examples 202

202

203

39: **205**

Examples 205

Rigidbodies 205

205

Rigidbody **205**

Rigidbody **205**

205

, **206**

isKinematic **206**

206

207

207

40: 209

Examples 209

(MVC) 209

213

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: unity3d

It is an unofficial and free unity3d ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official unity3d.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с unity3d

замечания

Unity обеспечивает среду разработки кросс-платформенной игры для разработчиков. Разработчики могут использовать язык C # и / или JavaScript, основанный на синтаксисе UnityScript для программирования игры. Целевые платформы развертывания могут легко переключаться в редакторе. Все основные игровые коды остаются такими же, за исключением некоторых зависимых от платформы функций. Список всех версий и соответствующих загрузок и примечаний к выпуску можно найти здесь: https://unity3d.com/get-unity/download/archive .

Версии

Версия	Дата выхода
Unity 2017.1.0	2017-07-10
5.6.2	2017-06-21
5.6.1	2017-05-11
5.6.0	2017-03-31
5.5.3	2017-03-31
5.5.2	2017-02-24
5.5.1	2017-01-24
5,5	2016-11-30
5.4.3	2016-11-17
5.4.2	2016-10-21
5.4.1	2016-09-08
5.4.0	2016-07-28
5.3.6	2016-07-20
5.3.5	2016-05-20
5.3.4	2016-03-15

Версия	Дата выхода
5.3.3	2016-02-23
5.3.2	2016-01-28
5.3.1	2015-12-18
5.3.0	2015-12-08
5.2.5	2016-06-01
5.2.4	2015-12-16
5.2.3	2015-11-19
5.2.2	2015-10-21
5.2.1	2015-09-22
5.2.0	2015-09-08
5.1.5	2015-06-07
5.1.4	2015-10-06
5.1.3	2015-08-24
5.1.2	2015-07-16
5.1.1	2015-06-18
5.1.0	2015-06-09
5.0.4	2015-07-06
5.0.3	2015-06-09
5.0.2	2015-05-13
5.0.1	2015-04-01
5.0.0	2015-03-03
4.7.2	2016-05-31
4.7.1	2016-02-25
4.7.0	2015-12-17
4.6.9	2015-10-15

Версия	Дата выхода
4.6.8	2015-08-26
4.6.7	2015-07-01
4.6.6	2015-06-08
4.6.5	2015-04-30
4.6.4	2015-03-26
4.6.3	2015-02-19
4.6.2	2015-01-29
4.6.1	2014-12-09
4.6.0	2014-11-25
4.5.5	2014-10-13
4.5.4	2014-09-11
4.5.3	2014-08-12
4.5.2	2014-07-10
4.5.1	2014-06-12
4.5.0	2014-05-27
4.3.4	2014-01-29
4.3.3	2014-01-13
4.3.2	2013-12-18
4.3.1	2013-11-28
4.3.0	2013-11-12
4.2.2	2013-10-10
4.2.1	2013-09-05
4.2.0	2013-07-22
4.1.5	2013-06-08
4.1.4	2013-06-06

Версия	Дата выхода
4.1.3	2013-05-23
4.1.2	2013-03-26
4.1.0	2013-03-13
4.0.1	2013-01-12
4.0.0	2012-11-13
3.5.7	2012-12-14
3.5.6	2012-09-27
3.5.5	2012-08-08
3.5.4	2012-07-20
3.5.3	2012-06-30
3.5.2	2012-05-15
3.5.1	2012-04-12
3.5.0	2012-02-14
3.4.2	2011-10-26
3.4.1	2011-09-20
3.4.0	2011-07-26

Examples

Установка или настройка

обзор

Unity работает на Windows и Mac. Существует также версия альфа-версии Linux .

Существует 4 разных плана платежей для Unity:

1. **Персональный** - бесплатно *(см. Ниже)*
2. **Плюс** - $ 35 USD в месяц за место *(см. Ниже)*
3. **Pro** - 125 долларов США в месяц за место - после подписания плана Pro в течение 24

месяцев подряд у вас есть возможность прекратить подписку и сохранить версию, которую вы имеете.
4. **Предприятие** - контакт Unity для получения дополнительной информации

*Согласно EULA: Компании или объединенные организации, оборот которых превысил 100 000 долларов США в последний финансовый год, должны использовать **Unity Plus** (или более высокую лицензию); на сумму свыше 200 000 долларов США они должны использовать **Unity Pro** (или Enterprise).*

Установка

1. Загрузите помощник загрузки Unity .
2. Запустите помощник и выберите модули, которые вы хотите загрузить и установить, например, редактор Unity, среду разработки MonoDevelop, документацию и необходимые модули для сборки платформы.

Если у вас установлена более старая версия, вы можете обновить ее до последней стабильной версии .

Если вы хотите установить Unity без помощника загрузки Unity, вы можете получить **установщиков компонентов** из примечаний к выпуску Unity 5.5.1 .

Установка нескольких версий единства

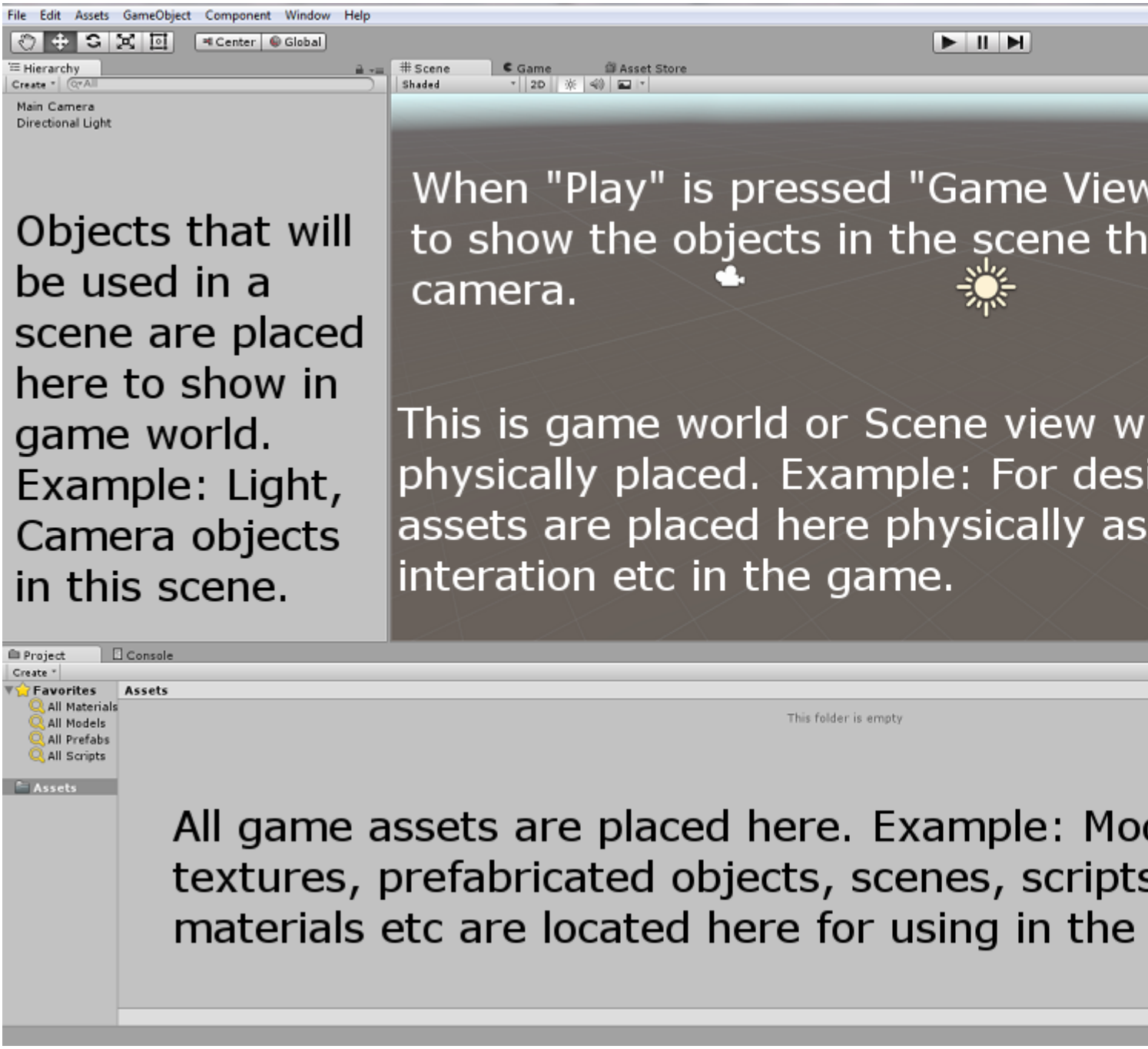
Часто бывает необходимо установить несколько версий Unity одновременно. Для этого:

- В Windows измените каталог установки по умолчанию на пустую папку, которую вы создали ранее, например `Unity 5.3.1f1` .
- На Mac установщик всегда будет устанавливать в `/Applications/Unity` . Переименуйте эту папку для существующей установки (например, в `/Applications/Unity5.3.1f1`) перед запуском установщика для другой версии.
- Вы можете удерживать `Alt` при запуске Unity, чтобы заставить его выбрать проект для открытия. В противном случае последний загруженный проект будет пытаться загрузить (если он доступен), и может предложить вам обновить проект, который вы не хотите обновлять.

Основной редактор и код

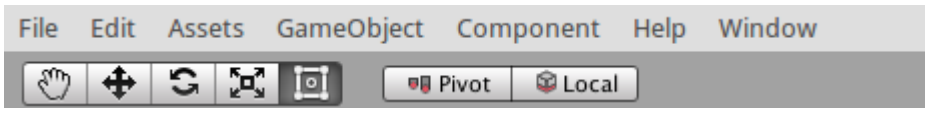
раскладка

Основной редактор Unity будет выглядеть ниже. Основные функции некоторых окон / вкладок по умолчанию описаны на изображении.



Макет Linux

Существует небольшая разница в компоновке меню версии Linux, например, снимок экрана ниже,



Основное использование

Создайте пустой объект `GameObject` , щелкнув правой кнопкой мыши в окне иерархии и

выберите « `Create Empty` . Создайте новый скрипт, щелкнув правой кнопкой мыши в окне Project и выберите `Create` > `C# Script` . Переименуйте его по мере необходимости.

Когда в окне «Иерархия» выбран пустой `GameObject` , перетащите вновь созданный скрипт в окно «Инспектор». Теперь скрипт прикреплен к объекту в окне иерархии. Откройте скрипт с установленной по умолчанию MonoDevelop IDE или вашим предпочтением.

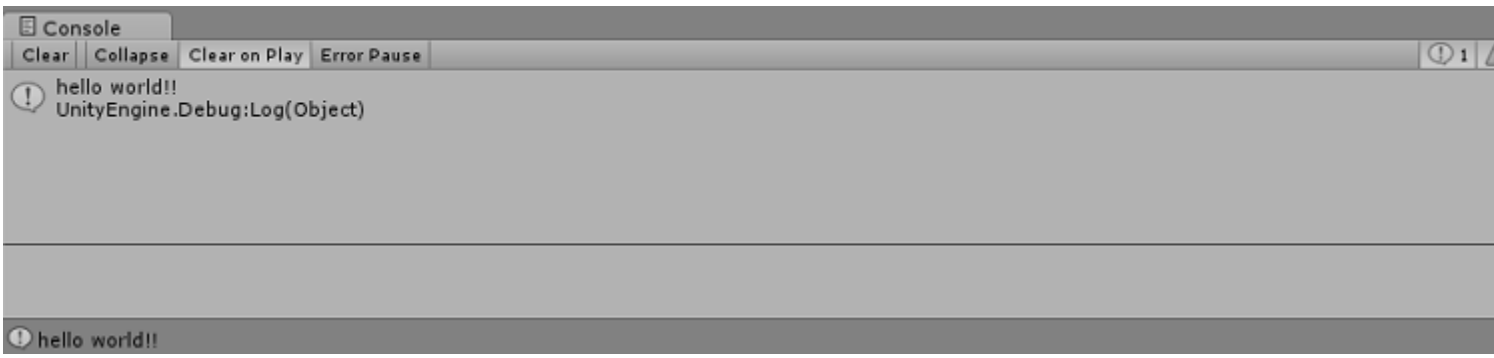
Основные скрипты

Базовый код будет выглядеть ниже, кроме строки `Debug.Log("hello world!!");` ,

```
using UnityEngine;
using System.Collections;

public class BasicCode : MonoBehaviour {

    // Use this for initialization
    void Start () {
        Debug.Log("hello world!!");
    }

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

    }
}
```

Добавьте строку `Debug.Log("hello world!!");` в методе `void Start()` . Сохраните сценарий и вернитесь в редактор. Запустите его, нажав « **Воспроизвести**» в верхней части редактора.

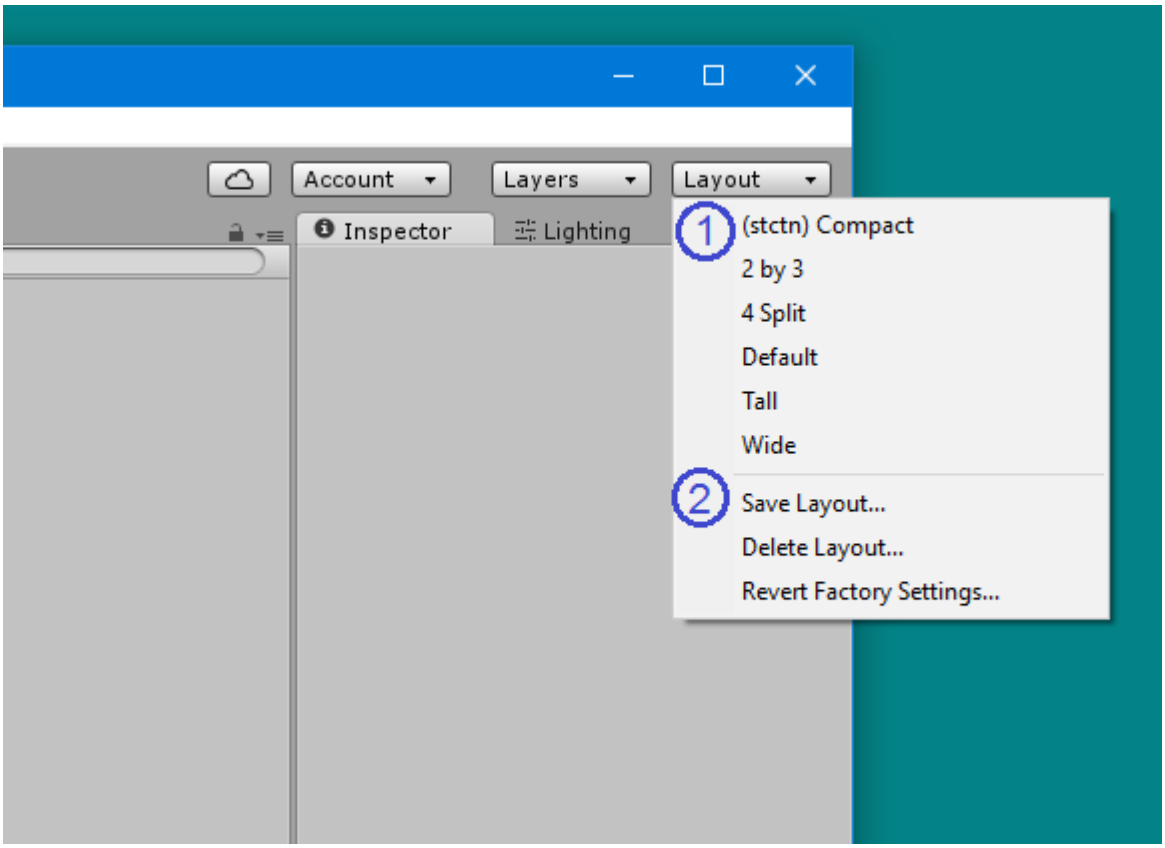
Результат должен выглядеть следующим образом в окне консоли:



Макеты редакторов

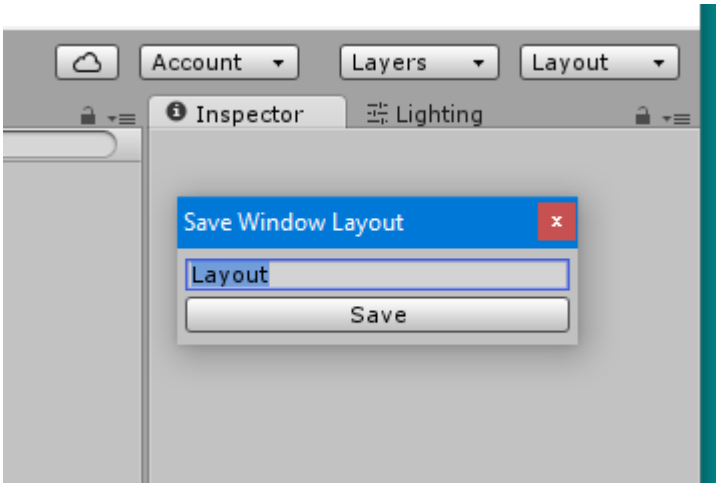
Вы можете сохранить расположение ваших вкладок и окон, чтобы стандартизировать свою рабочую среду.

Меню макетов можно найти в правом верхнем углу редактора Unity:

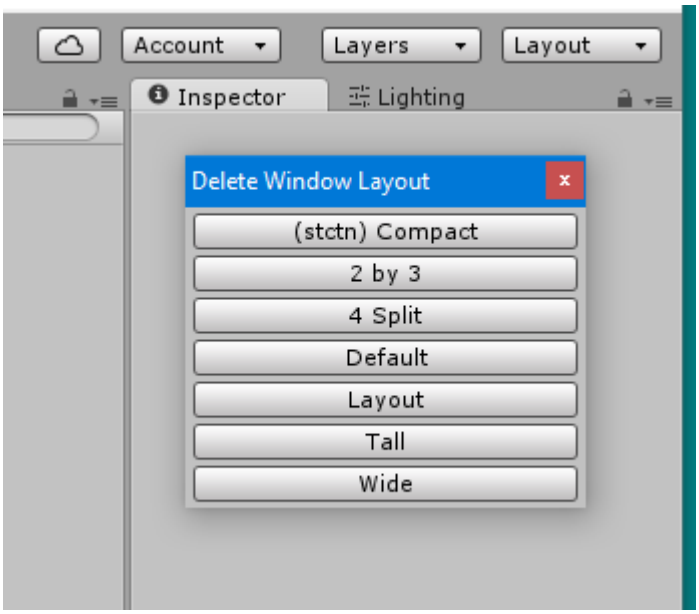


Unity поставляется с 5 стандартными макетами (2 на 3, 4 сплита, по умолчанию, высокая, широкая) *(отмечена 1)* . На картинке выше, помимо макетов по умолчанию, в верхней части также есть собственный макет.

Вы можете добавить свои собственные макеты, нажав кнопку **«Сохранить макет ...»** в меню *(с пометкой 2)* :



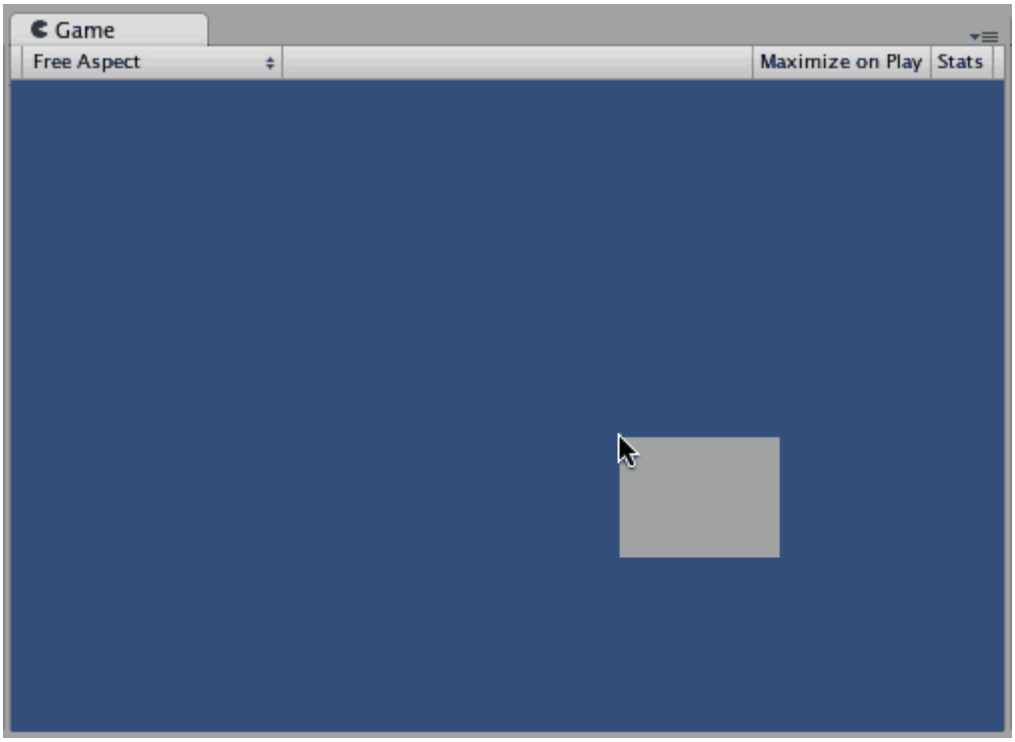
Вы также можете удалить любой макет, нажав кнопку **«Удалить макет ...»** в меню *(с пометкой 2)* :



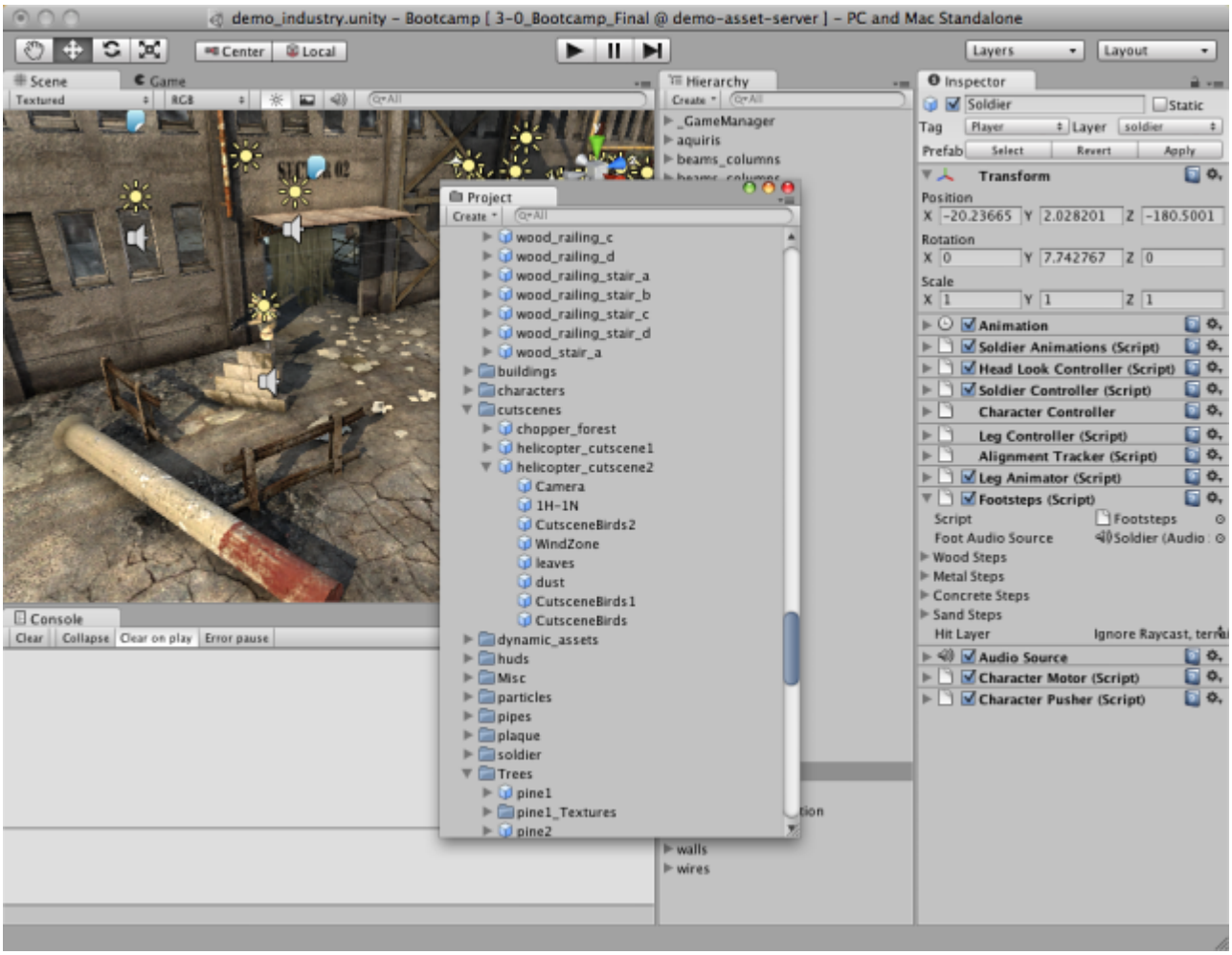
Кнопка **«** Восстановить **заводские настройки ...»** удаляет все пользовательские макеты и восстанавливает макеты по умолчанию *(помечены знаком 2)* .

Настройка рабочей области

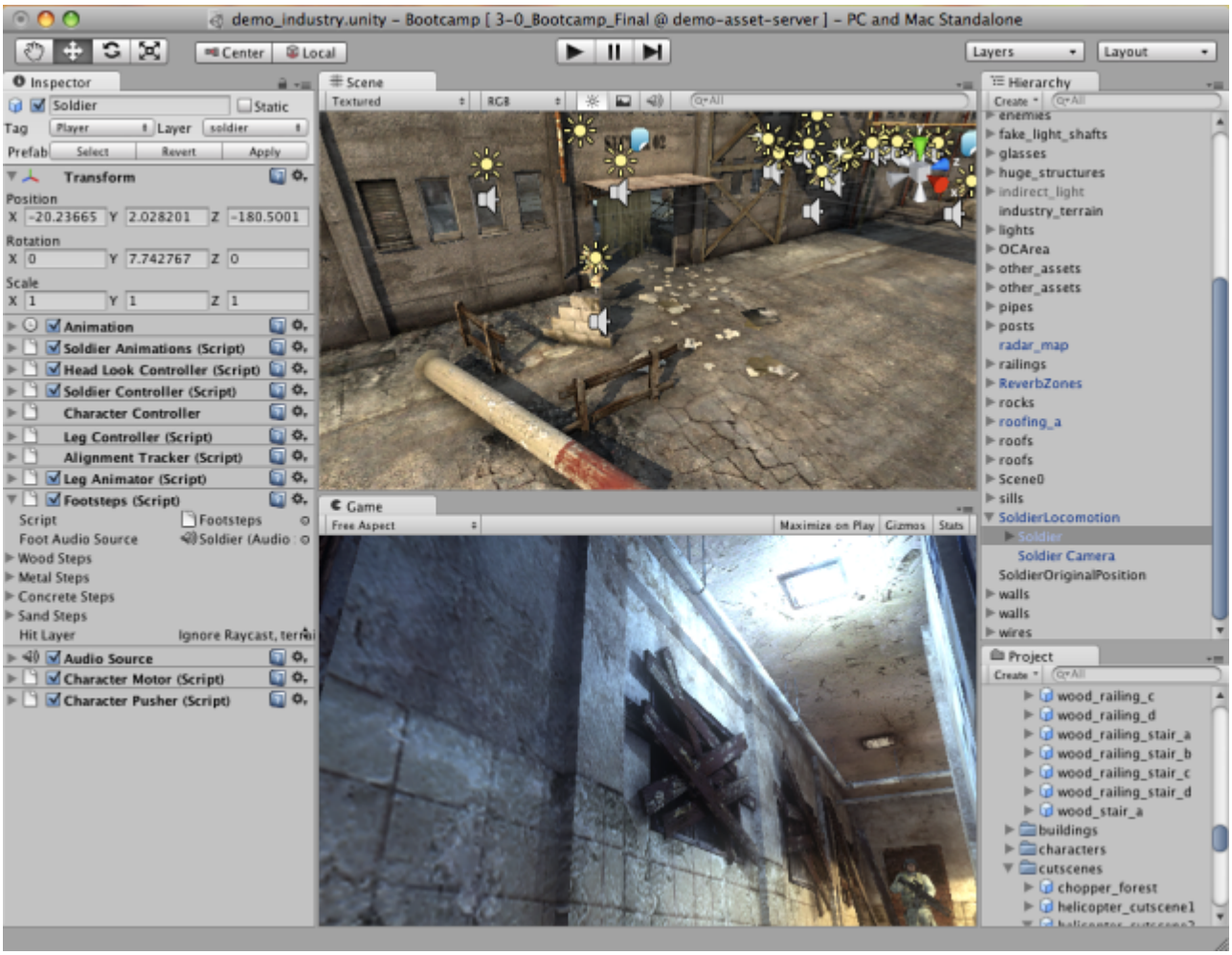
Вы можете настроить макет представлений, перетаскивая вкладку любого вида в одно из нескольких мест. Отбрасывание вкладки в области вкладок существующего окна добавит вкладку рядом с любыми существующими вкладками. В качестве альтернативы, отбрасывание вкладки в любой зоне док-станции добавит представление в новое окно.



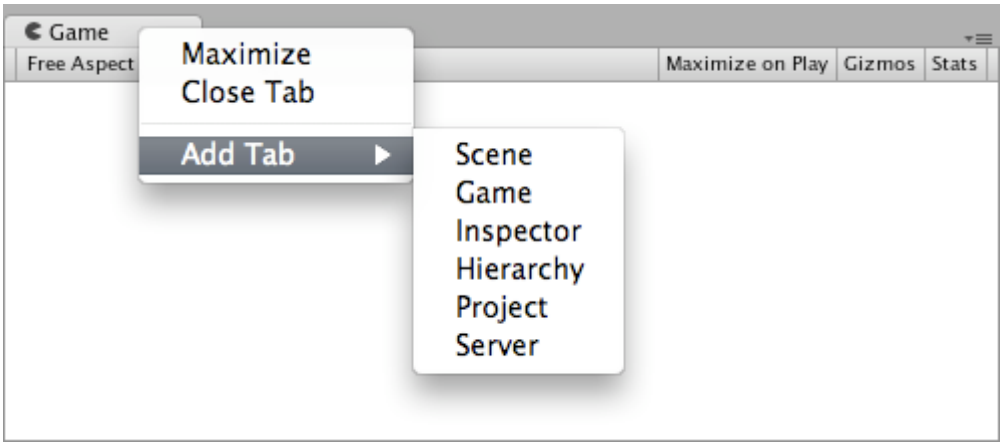
Вкладки также могут быть отделены от главного окна редактора и помещены в их собственный плавающий редактор Windows. Плавающие окна могут содержать расположение представлений и вкладок, как и окно главного редактора.



Когда вы создали макет редактора, вы можете сохранить макет и восстановить его в любое время. См. Этот пример для макетов редакторов .



В любое время вы можете щелкнуть правой кнопкой мыши вкладку любого вида, чтобы просмотреть дополнительные параметры, такие как «Максимизировать» или добавить новую вкладку в одно и то же окно.



Прочитайте Начало работы с unity3d онлайн: https://riptutorial.com/ru/unity3d/topic/846/начало-работы-с-unity3d

глава 2: API CullingGroup

замечания

Поскольку использование CullingGroups не всегда очень просто, может оказаться полезным инкапсулировать основную часть логики класса менеджера.

Ниже приведен пример того, как может работать такой менеджер.

```
using UnityEngine;
using System;
public interface ICullingGroupManager
{
    int ReserveSphere();
    void ReleaseSphere(int sphereIndex);
    void SetPosition(int sphereIndex, Vector3 position);
    void SetRadius(int sphereIndex, float radius);
    void SetCullingEvent(int sphereIndex, Action<CullingGroupEvent> sphere);
}
```

Суть в том, что вы резервируете сферу отбраковки от менеджера, который возвращает индекс зарезервированной сферы. Затем вы используете данный индекс для управления вашей зарезервированной сферой.

Examples

Отбор объектов

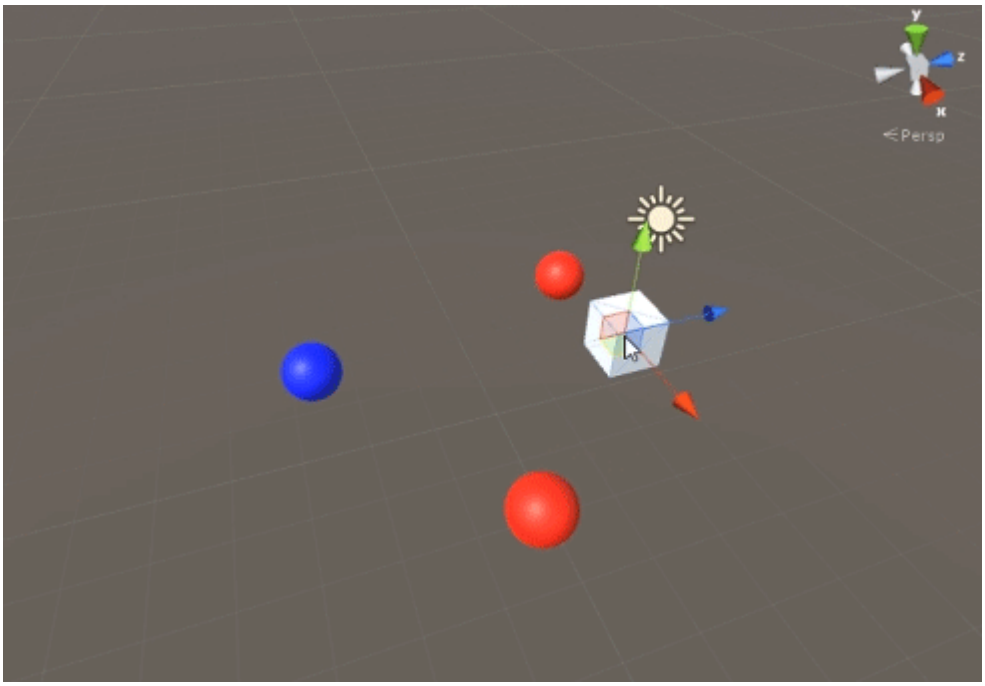
В следующем примере показано, как использовать CullingGroups для получения уведомлений в соответствии с контрольной точкой расстояния.

> Этот сценарий был упрощен для краткости и использует несколько высокопроизводительных методов.

```
using UnityEngine;
using System.Linq;

public class CullingGroupBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;
    Transform[] meshTransforms;
    BoundingSphere[] cullingPoints;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();
```

```
        meshRenderers = FindObjectsOfType<MeshRenderer>()
                .Where((MeshRenderer m) => m.gameObject != this.gameObject)
                .ToArray();

        cullingPoints = new BoundingSphere[meshRenderers.Length];
        meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.SetBoundingDistances(new float[] { 0f, 5f });
        localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
        localCullingGroup.targetCamera = Camera.main;
    }

    void FixedUpdate()
    {
        localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
        for (var i = 0; i < meshTransforms.Length; i++)
        {
            cullingPoints[i].position = meshTransforms[i].position;
        }
    }

    void CullingEvent(CullingGroupEvent sphere)
    {
        Color newColor = Color.red;
        if (sphere.currentDistance == 1) newColor = Color.blue;
        if (sphere.currentDistance == 2) newColor = Color.white;
        meshRenderers[sphere.index].material.color = newColor;
    }

    void OnDisable()
    {
        localCullingGroup.Dispose();
    }
}
```

Добавьте скрипт в GameObject (в данном случае куб) и нажмите «Воспроизвести». Каждый другой объект GameObject в сцене меняет цвет в соответствии с их расстоянием до контрольной точки.



Очистка видимости объекта

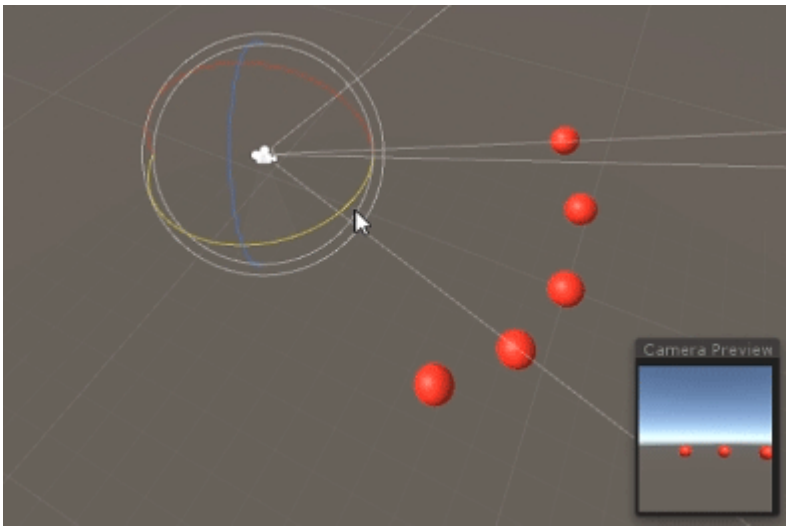
Следующий сценарий иллюстрирует, как получать события в соответствии с видимостью для установленной камеры.

> Этот сценарий для краткости использует несколько высокопроизводительных методов.

```
using UnityEngine;
using System.Linq;

public class CullingGroupCameraBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
            .Where((MeshRenderer m) => m.gameObject != this.gameObject)
            .ToArray();

        BoundingSphere[] cullingPoints = new BoundingSphere[meshRenderers.Length];
        Transform[] meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
```

```
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.targetCamera = Camera.main;
    }

    void CullingEvent(CullingGroupEvent sphere)
    {
        meshRenderers[sphere.index].material.color = sphere.isVisible ? Color.red :
Color.white;
    }

    void OnDisable()
    {
        localCullingGroup.Dispose();
    }
}
```

Добавьте сценарий в сцену и нажмите «Играть». Вся геометрия в сцене изменит цвет, основываясь на их видимости.



> Подобный эффект может быть достигнут с использованием `MonoBehaviour.OnBecameVisible()` если объект имеет компонент `MeshRenderer` . Используйте CulingGroups, когда вам нужно отбросить пустые координаты GameObjects, `Vector3` или когда вам нужен централизованный метод отслеживания видимости объектов.

Ограничительные расстояния

Вы можете добавить ограничивающие расстояния поверх радиуса точки высева. Они находятся в режиме дополнительных условий срабатывания вне основного радиуса точки выбраковки, например, «близко», «далеко» или «очень далеко».

```
cullingGroup.SetBoundingDistances(new float[] { 0f, 10f, 100f});
```

> Ограничения расстояний влияют только при использовании с дистанционной контрольной точкой. Они не действуют во время отбраковки камеры.

Визуализация граничных расстояний

То, что может изначально вызвать путаницу, заключается в том, что ограничивающие расстояния добавляются поверх радиусов сферы.

Во-первых, группа выбраковки вычисляет *площадь* как ограничивающей сферы, так и граничного расстояния. Эти две области объединены вместе, и результатом является область триггера для диапазона расстояния. Радиус этой области может быть использован для визуализации поля действия ограничивающего расстояния.

```
float cullingPointArea = Mathf.PI * (cullingPointRadius * cullingPointRadius);
float boundingArea = Mathf.PI * (boundingDistance * boundingDistance);
float combinedRadius = Mathf.Sqrt((cullingPointArea + boundingArea) / Mathf.PI);
```

Прочитайте API CullingGroup онлайн: https://riptutorial.com/ru/unity3d/topic/4574/api-cullinggroup

глава 3: Prefabs

Синтаксис

- public static Object PrefabUtility.InstantiatePrefab (Object target);
- public static Object AssetDatabase.LoadAssetAtPath (string assetPath, тип типа);
- public static Object Object.Instantiate (Object original);
- public static Object Resources.Load (строка пути);

Examples

Вступление

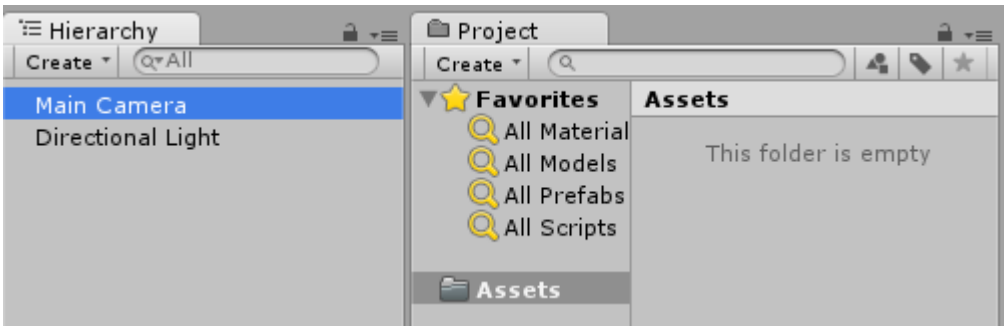
Prefabs - это тип актива, который позволяет хранить полный GameObject с его компонентами, свойствами, прикрепленными компонентами и сериализованными значениями свойств. Существует много сценариев, где это полезно, в том числе:

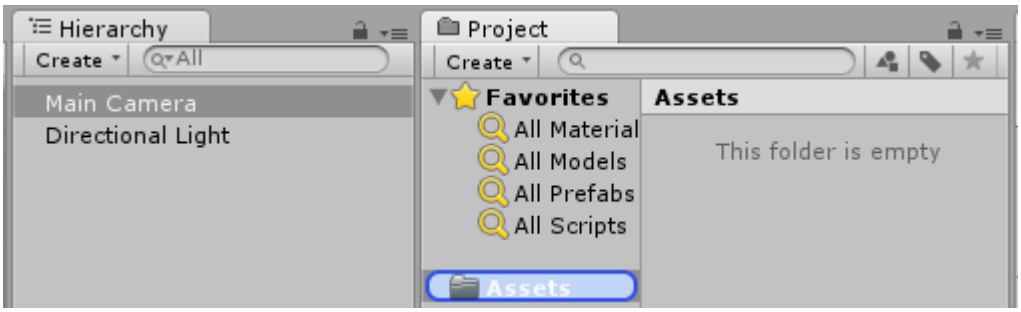
- Дублирование объектов в сцене
- Совместное использование общего объекта в нескольких сценах
- Возможность модифицировать сборку один раз и иметь изменения применительно к нескольким объектам / сценам
- Создание дублирующих объектов с незначительными изменениями, при этом общие элементы можно редактировать из одного базового сборника
- Инициирование GameObjects во время выполнения

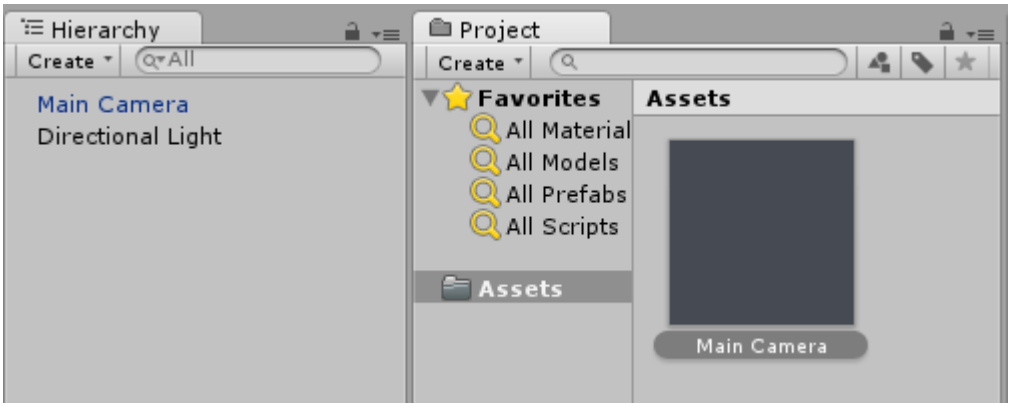
В Unity есть эмпирическое правило, в котором говорится, что «все должно быть Prefabs». Хотя это, вероятно, преувеличение, оно поощряет повторное использование кода и создание GameObjects многоразовым способом, который одновременно эффективен как с точки зрения памяти, так и с хорошим дизайном.

Создание сборных

Чтобы создать сборку, перетащите игровой объект из иерархии сцен в папку « **Активы** » или вложенную папку:

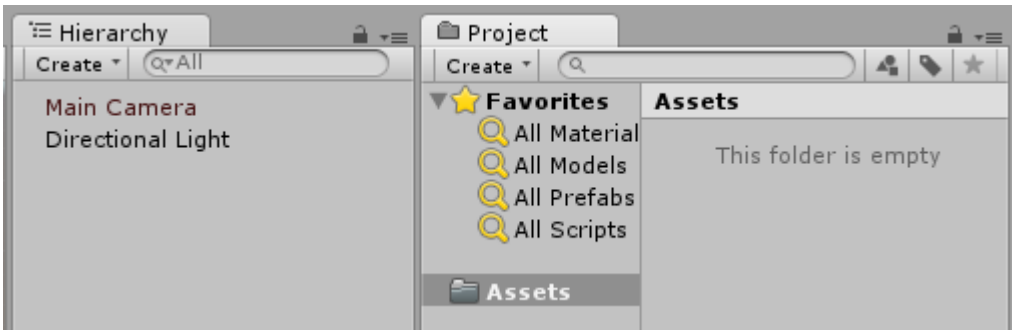






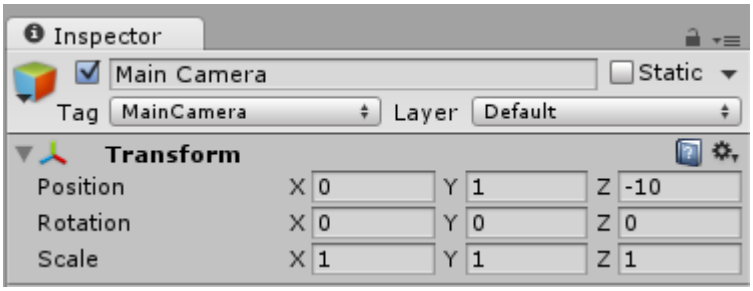
Имя игрового объекта становится синим, указывая, что оно **связано с сборкой** .
Теперь этот объект является **экземпляром prefab** , как экземпляр объекта класса.

После создания может быть удалена сборная сборка. В этом случае имя ранее связанного с ним игрового объекта становится красным:

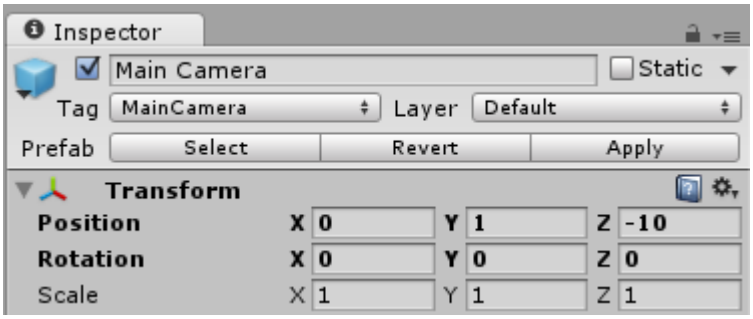


Инспектор по сборке

Если вы выберете сборку в иерархическом представлении, вы заметите, что ее инспектор немного отличается от обычного игрового объекта:



против



Жирные свойства означают, что их значения отличаются от значений prefab. Вы можете изменить любое свойство созданного экземпляра, не влияя на исходные значения prefab. Когда значение изменяется в экземпляре prefab, оно становится полужирным, и любые последующие изменения одного и того же значения в prefab не будут отображаться в измененном экземпляре.

Вы можете вернуться к исходным значениям prefab, нажав кнопку **Revert** , которая также будет иметь изменения значения, отраженные в экземпляре. Кроме того, чтобы вернуть индивидуальное значение, вы можете щелкнуть его правой кнопкой мыши и нажать « **Восстановить значение» до «Prefab»** . Чтобы вернуть компонент, щелкните его правой кнопкой мыши и нажмите « **Вернуть в Prefab»** .

Нажатие кнопки « **Применить»** перезаписывает значения свойств prefab с текущими значениями свойств игрового объекта. Нет кнопки «Отменить» или подтверждения, поэтому обращайтесь с этой кнопкой осторожно.

Кнопка **выбора** подчеркивает связанную сборку в структуре папок проекта.

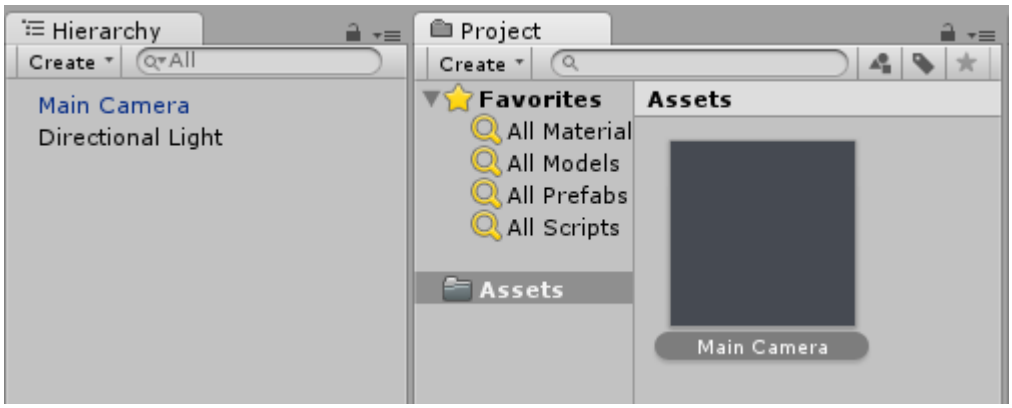
Создание экземпляров

Существует два способа создания экземпляров: во время **разработки** или **времени выполнения** .

Время создания проекта

Создание экземпляров prefabs во время разработки полезно для визуального размещения нескольких экземпляров одного и того же объекта (например, *размещение деревьев при разработке уровня вашей игры*).

- Чтобы визуально создать экземпляр prefab, перетащите его из представления проекта в иерархию сцен.



- Если вы пишете расширение редактора , вы также можете создать экземпляр prefab, программируя вызов `PrefabUtility.InstantiatePrefab()` :

```
GameObject gameObject =
(GameObject)PrefabUtility.InstantiatePrefab(AssetDatabase.LoadAssetAtPath("Assets/MainCamera.pref
typeof(GameObject)));
```

Исполнение времени выполнения

Создание экземпляров prefabs во время выполнения полезно для создания экземпляров объекта в соответствии с некоторой логикой (например, *размножение противника каждые 5 секунд*).

Чтобы создать экземпляр сборника, вам нужна ссылка на объект prefab. Это можно сделать, если в вашем `MonoBehaviour` (и установить его значение с помощью инспектора в редакторе Unity) можно `MonoBehaviour public GameObject` поле `public GameObject` :

```
public class SomeScript : MonoBehaviour {
   public GameObject prefab;
}
```

Или, поместив prefab в папку Resource и используя `Resources.Load` :

```
GameObject prefab = Resources.Load("Assets/Resources/MainCamera");
```

Когда у вас есть ссылка на объект prefab, вы можете создать его экземпляр, используя функцию `Instantiate` любом месте вашего кода (например, *внутри цикла для создания нескольких объектов*):

```
GameObject gameObject = Instantiate<GameObject>(prefab, new Vector3(0,0,0),
Quaternion.identity);
```

Примечание. В режиме исполнения термин *Prefab* не существует.

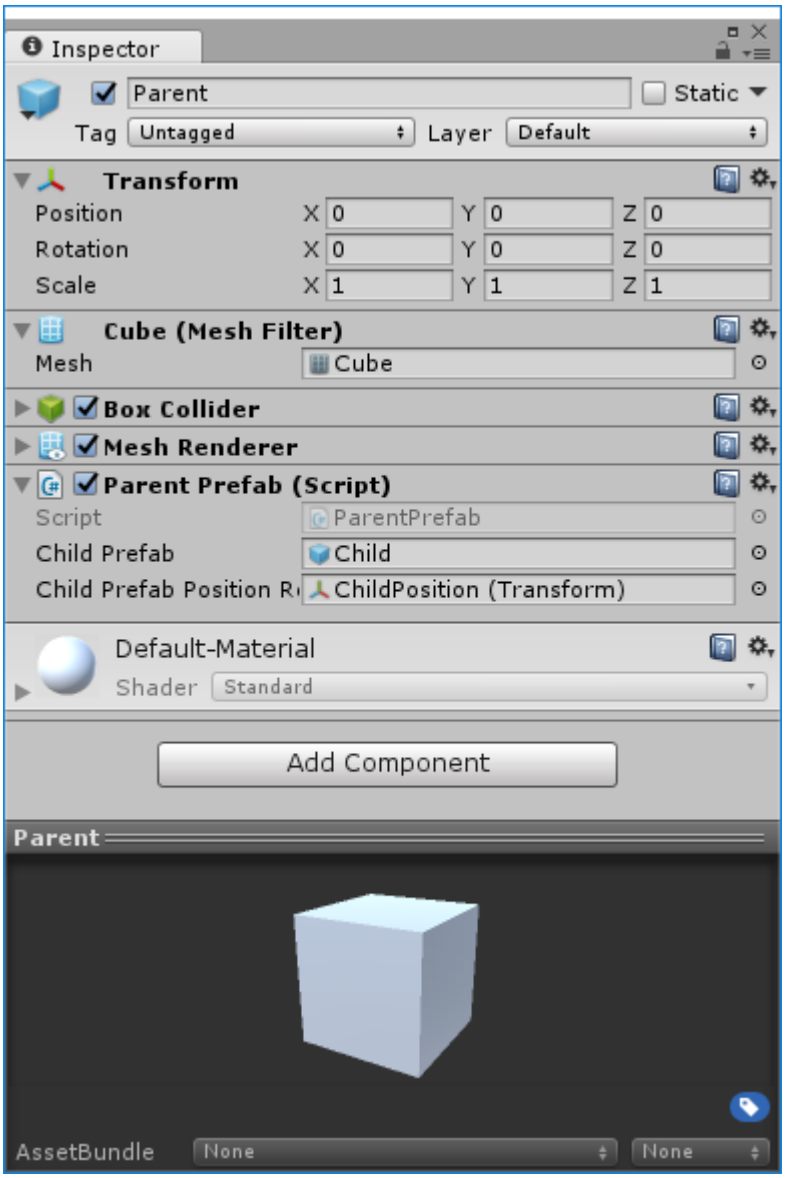
Вложенные сборные

В настоящее время в Unity недоступны вложенные сборники. Вы можете перетащить один prefab в другой и применить это, но любые изменения в дочернем сборнике не будут применены к вложенному.

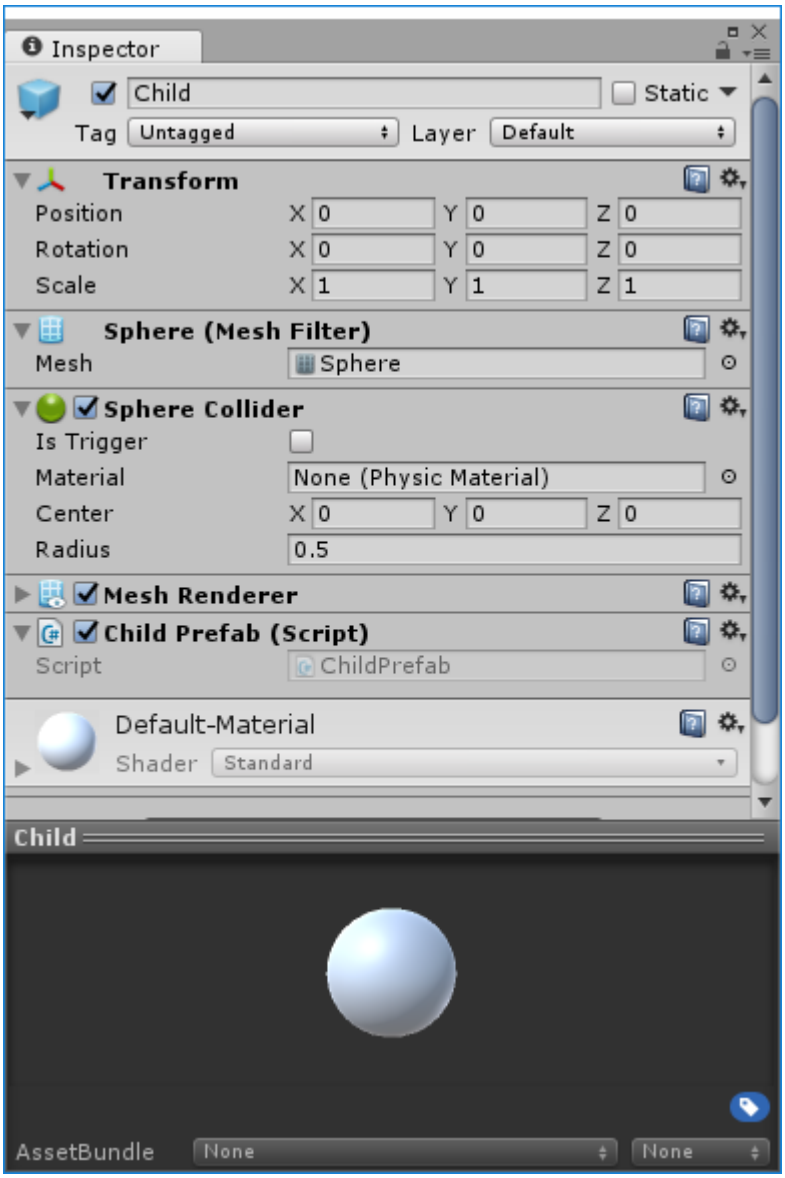
Но есть простой способ - **вы должны добавить в родительский сборник простой скрипт, который будет создавать экземпляр дочернего.**

```
using UnityEngine;

public class ParentPrefab : MonoBehaviour {

    [SerializeField] GameObject childPrefab;
    [SerializeField] Transform childPrefabPositionReference;

    // Use this for initialization
    void Start () {
        print("Hello, I'm a parent prefab!");
        Instantiate(
            childPrefab,
            childPrefabPositionReference.position,
            childPrefabPositionReference.rotation,
            gameObject.transform
        );
    }
}
```

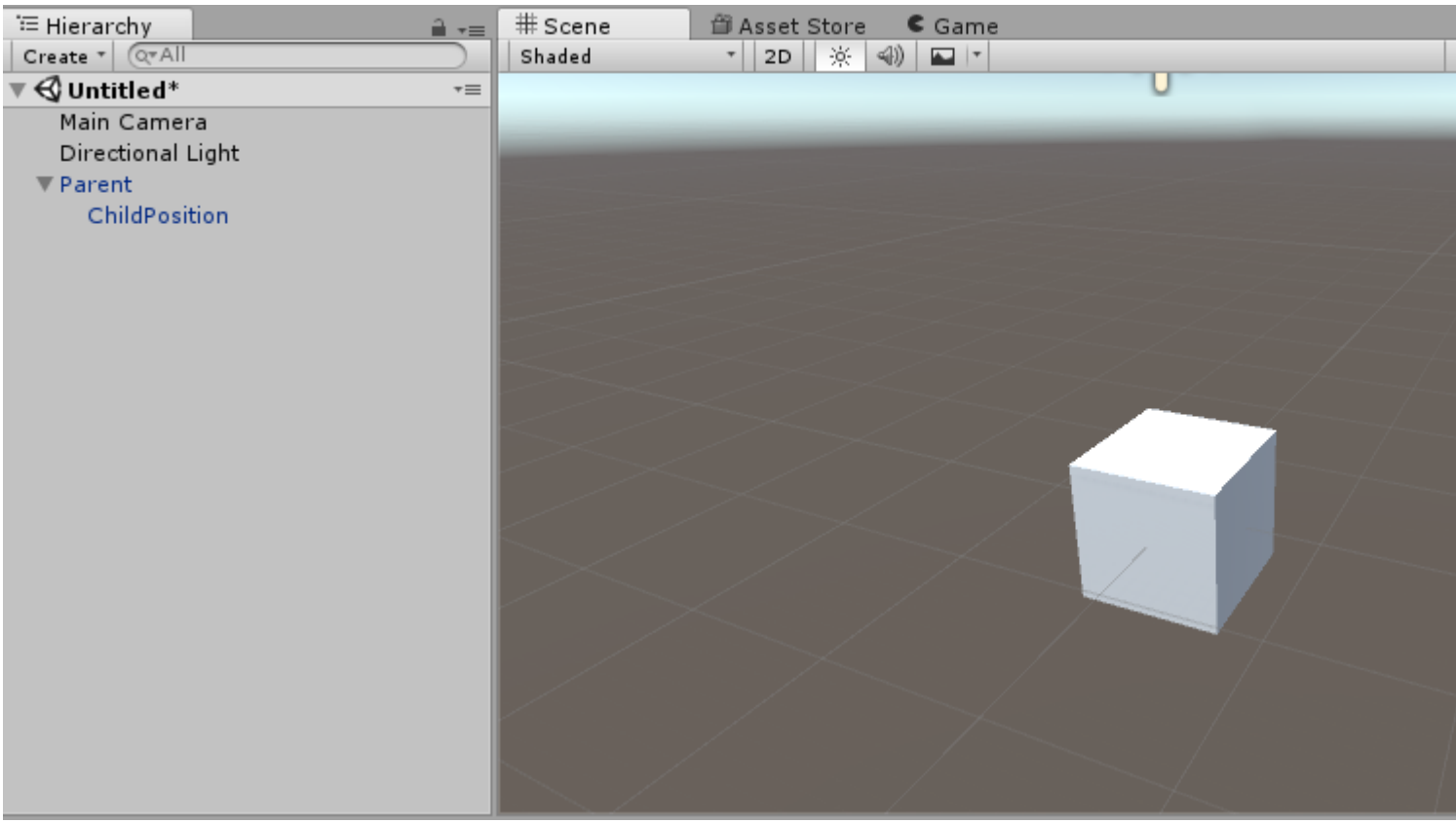
Родительский сборник:

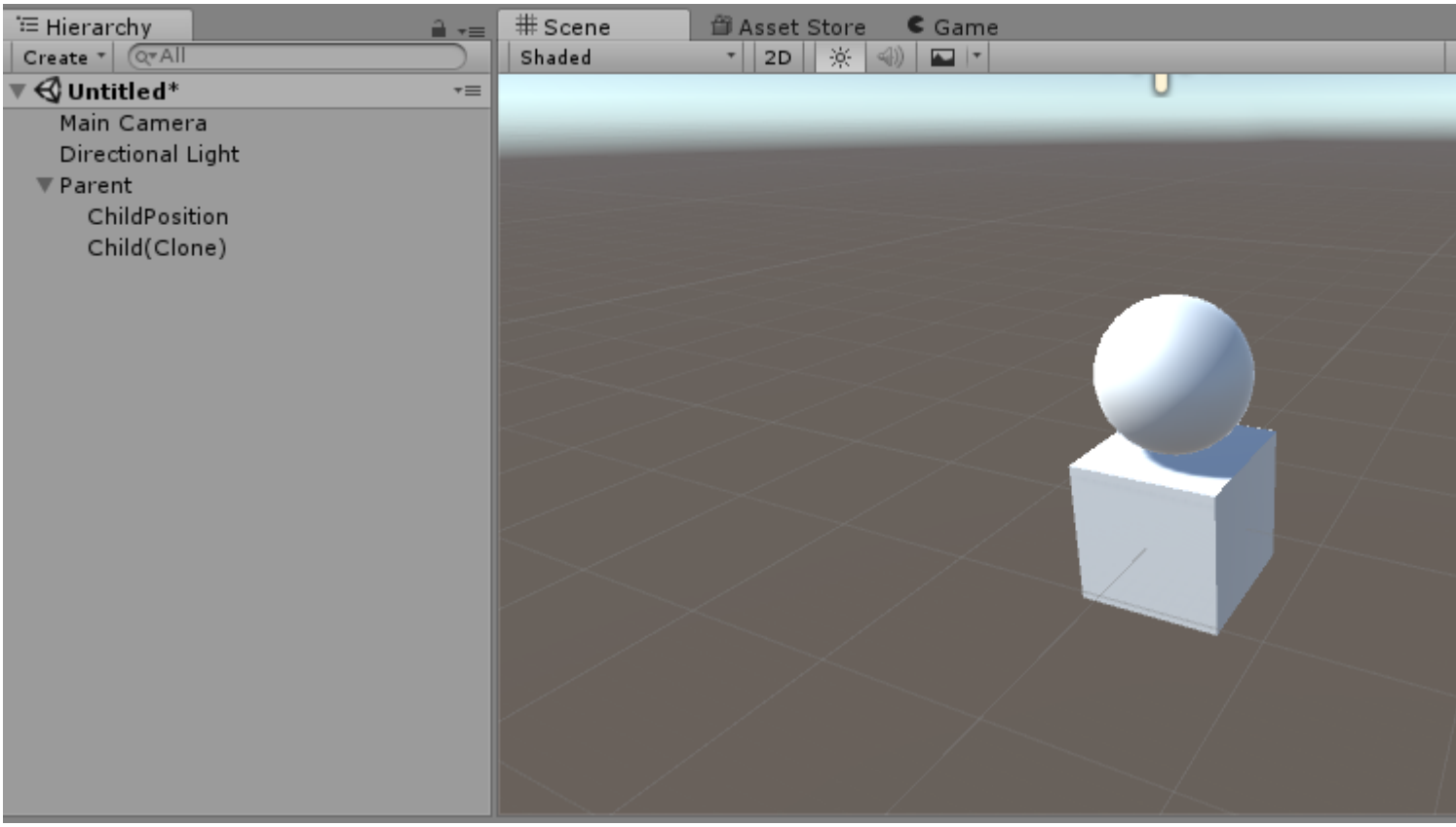


Детский сборник:



Сцена до и после старта:





Прочитайте Prefabs онлайн: https://riptutorial.com/ru/unity3d/topic/2133/prefabs

глава 4: Raycast

параметры

параметр	подробности
происхождения	Начальная точка луча в мировых координатах
направление	Направление луча
maxDistance	Максимальное расстояние, которое луч должен проверять на наличие столкновений
слой маски	Маска слоя, которая используется для выборочного игнорирования коллайдеров при бросании луча.
queryTriggerInteraction	Указывает, где этот запрос должен ударить триггеры.

Examples

Физика Raycast

Эта функция передает луч из `origin` в направлении `direction` длины `maxDistance` против всех коллайдеров в сцене.

Функция принимает в `origin` `direction` `maxDistance` и вычислить , если есть коллайдер перед GameObject.

```
Physics.Raycast(origin, direction, maxDistance);
```

Например, эта функция будет печатать `Hello World` на консоли, если к ней подключено что-то в пределах 10 единиц `GameObject` :

```
using UnityEngine;

public class TestPhysicsRaycast: MonoBehaviour
{
    void FixedUpdate()
    {
        Vector3 fwd = transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, fwd, 10))
            print("Hello World");
    }
}
```

Физика2D Raycast2D

Вы можете использовать raycasts, чтобы проверить, может ли ai ходить, не падая с края уровня.

```
using UnityEngine;

public class Physics2dRaycast: MonoBehaviour
    {
        public LayerMask LineOfSightMask;
        void FixedUpdate()
        {
            RaycastHit2D hit = Physics2D.Raycast(raycastRightPart, Vector2.down, 0.6f *
heightCharacter, LineOfSightMask);
            if(hit.collider != null)
            {
                //code when the ai can walk
            }
            else
            {
                //code when the ai cannot walk
            }
    }
}
```

В этом примере направление правильное. Переменная raycastRightPart является правой частью персонажа, поэтому raycast будет происходить в правой части персонажа. Расстояние в 0.6f раз превышает высоту персонажа, поэтому raycast не даст удар, когда он ударит по земле, которая ниже, чем земля, на которой он сейчас стоит. Убедитесь, что Layermask установлен только на землю, иначе он также обнаружит другие объекты.

> RaycastHit2D сам по себе является структурой, а не классом, поэтому удар не может быть нулевым; это означает, что вам нужно проверить коллайдер переменной RaycastHit2D.

Инкапсуляция вызовов Raycast

Если ваши сценарии называют `Raycast` напрямую, это может привести к проблемам, если вам нужно изменить матрицы столкновений в будущем, так как вам придется отслеживать каждое поле `LayerMask` для внесения изменений. В зависимости от размера вашего проекта это может стать огромным начинанием.

Инкапсуляция вызовов `Raycast` может облегчить вам жизнь.

Рассматривая это по принципу SoC , игровой объект действительно не должен знать или заботиться о LayerMasks. Ему нужен только способ сканирования своего окружения. Получает ли результат raycast то или это не имеет значения для игрового объекта. Он должен действовать только на информацию, которую он получает, и не делать каких-либо предположений об окружающей среде, в которой он существует.

Одним из способов решения этой проблемы является , чтобы переместить значение LayerMask в ScriptableObject экземпляров и использовать их в качестве формы raycast услуг , которые вы инъекционная в скрипты.

```
// RaycastService.cs
using UnityEngine;

[CreateAssetMenu(menuName = "StackOverflow")]
public class RaycastService : ScriptableObject
{
    [SerializeField]
    LayerMask layerMask;

    public RaycastHit2D Raycast2D(Vector2 origin, Vector2 direction, float distance)
    {
        return Physics2D.Raycast(origin, direction, distance, layerMask.value);
    }

    // Add more methods as needed

}
```

```
// MyScript.cs
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [SerializeField]
    RaycastService raycastService;

    void FixedUpdate()
    {
        RaycastHit2D hit = raycastService.Raycast2D(Vector2.zero, Vector2.down, 1f);
    }
}
```

Это позволяет вам сделать несколько сервисов raycast, все с различными комбинациями LayerMask для разных ситуаций. У вас может быть тот, который ударяет только с коллайдерами на землю, а другой - с земными коллайдерами и односторонними платформами.

Если вам когда-либо понадобится внести радикальные изменения в настройки LayerMask, вам нужно только обновить эти ресурсы RaycastService.

дальнейшее чтение

- Инверсия контроля
- Внедрение зависимости

Прочитайте Raycast онлайн: https://riptutorial.com/ru/unity3d/topic/2826/raycast

глава 5: ScriptableObject

замечания

ScriptableObjects с AssetBundles

Обратите внимание на добавление сборных данных в AssetBundles, если они содержат ссылки на ScriptableObjects. Поскольку ScriptableObjects по существу являются активами, Unity создает их дубликаты перед добавлением их в AssetBundles, что может привести к нежелательному поведению во время выполнения.

Когда вы загружаете такой GameObject из AssetBundle, может потребоваться повторно загрузить атрибуты ScriptableObject в загруженные сценарии, заменив связанные объекты. См. Injection Dependency Injection

Examples

Вступление

ScriptableObjects - это сериализованные объекты, которые не привязаны к сценам или игровым объектам как MonoBehaviours. Иными словами, это данные и методы, связанные с файлами активов внутри вашего проекта. Эти объекты ScriptableObject могут быть переданы MonoBehaviours или другим ScriptableObjects, к которым можно получить доступ к их общедоступным методам.

Из-за своей природы как сериализованных активов они делают отличные классы менеджеров и источники данных.

Создание объектов ScriptableObject

Ниже приведена простая реализация ScriptableObject.

```
using UnityEngine;

[CreateAssetMenu(menuName = "StackOverflow/Examples/MyScriptableObject")]
public class MyScriptableObject : ScriptableObject
{
    [SerializeField]
    int mySerializedNumber;

    int helloWorldCount = 0;

    public void HelloWorld()
```

```
    {
        helloWorldCount++;
        Debug.LogFormat("Hello! My number is {0}.", mySerializedNumber);
        Debug.LogFormat("I have been called {0} times.", helloWorldCount);
    }
}
```

Добавив атрибут `CreateAssetMenu` к классу, Unity перечислит его в подменю « **Активы** / **Создать»** . В этом случае это находится в разделе **Assets / Create / StackOverflow / Examples** .

После создания экземпляры ScriptableObject могут быть переданы другим скриптам и ScriptableObjects через Инспектор.

```
using UnityEngine;

public class SampleScript : MonoBehaviour {

    [SerializeField]
    MyScriptableObject myScriptableObject;

    void OnEnable()
    {
        myScriptableObject.HelloWorld();
    }
}
```

Создание экземпляров ScriptableObject с помощью кода

Вы создаете новые экземпляры ScriptableObject через `ScriptableObject.CreateInstance<T>()`

```
T obj = ScriptableObject.CreateInstance<T>();
```

Где `T` расширяет `ScriptableObject` .

> Не создавайте ScriptableObjects, вызывая их конструкторы, т.е. `new ScriptableObject()` .

Создание ScriptableObjects по коду во время выполнения редко вызвано, потому что их основным использованием является сериализация данных. На этом этапе вы также можете использовать стандартные классы. Это чаще встречается при написании расширений редактора сценариев.

ScriptableObjects сериализуются в редакторе даже в PlayMode

Следует проявлять особую осторожность при доступе к сериализованным полям экземпляра ScriptableObject.

Если поле отмечено `public` или сериализованным через `SerializeField` , изменение его значения является постоянным. Они не сбрасываются при выходе из режима

воспроизведения, такого как MonoBehaviours. Иногда это может быть полезно, но это также может вызвать беспорядок.

Из-за этого лучше всего делать сериализованные поля только для чтения и вообще избегать публичных полей.

```
public class MyScriptableObject : ScriptableObject
{
    [SerializeField]
    int mySerializedValue;

    public int MySerializedValue
    {
        get { return mySerializedValue; }
    }
}
```

Если вы хотите сохранить общедоступные значения в объекте ScriptableObject, которые были сброшены между сеансами воспроизведения, рассмотрите возможность использования следующего шаблона.

```
public class MyScriptableObject : ScriptableObject
{
    // Private fields are not serialized and will reset to default on reset
    private int mySerializedValue;

    public int MySerializedValue
    {
        get { return mySerializedValue; }
        set { mySerializedValue = value; }
    }
}
```

Найти существующие объекты ScriptableObject во время выполнения

Чтобы найти *активные* объекты ScriptableObject во время выполнения, вы можете использовать `Resources.FindObjectsOfTypeAll()` .

```
T[] instances = Resources.FindObjectsOfTypeAll<T>();
```

Где `T` - тип экземпляра ScriptableObject, который вы ищете. *Active* означает, что он ранее был загружен в память в какой-либо форме.

Этот метод очень медленный, поэтому не забудьте кэшировать возвращаемое значение и не часто его вызывать. Предпочтительным вариантом является ссылка на ScriptableObjects непосредственно в ваших сценариях.

> *Совет.* Вы можете поддерживать собственные коллекции экземпляров для более быстрого поиска. Попросите свои ScriptableObjects зарегистрироваться в общую коллекцию во время `OnEnable()` .

Прочитайте ScriptableObject онлайн: https://riptutorial.com/ru/unity3d/topic/3434/scriptableobject

глава 6: Unity Animation

Examples

Основная анимация для бега

Этот код показывает простой пример анимации в Unity.

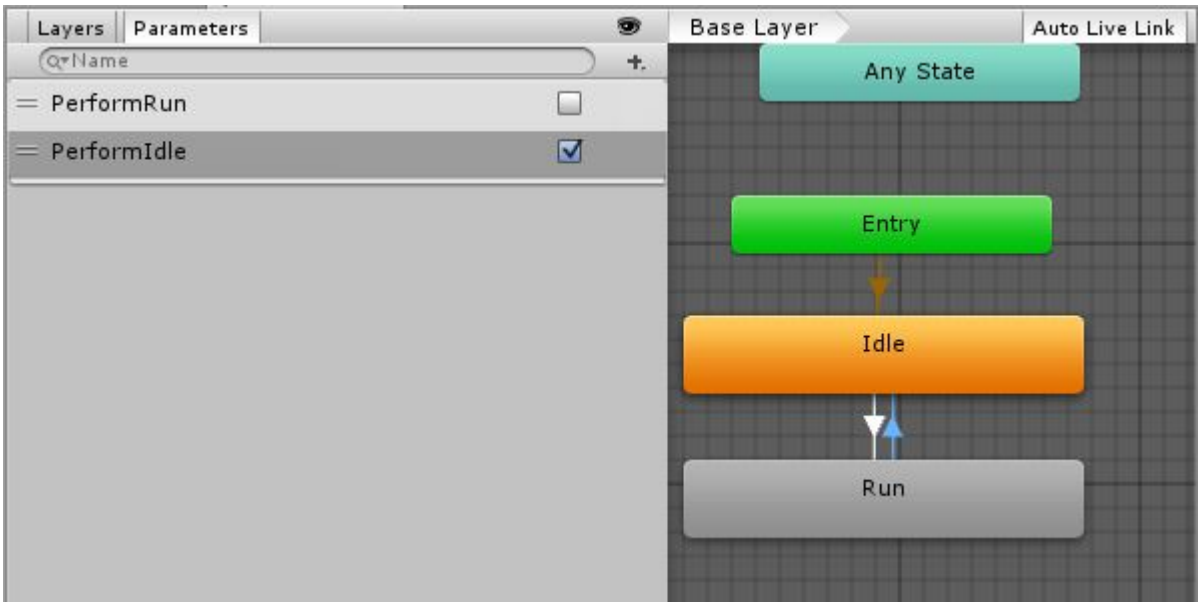
Для этого примера у вас должно быть 2 анимационных клипа; Run и Idle. Эти анимации должны быть движками Stand-In-Place. После выбора клипов анимации создайте Animator Controller. Добавьте этот контроллер к игроку или игровому объекту, который хотите оживить.

Откройте окно Animator из окна Windows. Перетащите 2 анимационных клипа в окно Animator и создайте 2 состояния. После создания используйте вкладку левых параметров, чтобы добавить 2 параметра, оба из них как bool. Назовите его «PerformRun» и другие как «PerformIdle». Установите для параметра «PerformIdle» значение true.

Сделайте переход из состояния ожидания для запуска и запуска в режим ожидания (см. Изображение). Перейдите в режим Idle-> Run и в окне Inspector отмените выбор HasExit. Сделайте то же самое для другого перехода. Для перехода в режим Idle-> Run добавьте условие: PerformIdle. Для Run-> Idle добавьте условие: PerformRun. Добавьте скрипт C #, указанный ниже, в игровой объект. Он должен запускаться с анимацией с помощью кнопки «Вверх» и поворот с помощью кнопок «Влево» и «Вправо».

```
using UnityEngine;
using System.Collections;

public class RootMotion : MonoBehaviour {

//Public Variables
[Header("Transform Variables")]
public float RunSpeed = 0.1f;
public float TurnSpeed = 6.0f;


Animator animator;

void Start()
{
    /**
    * Initialize the animator that is attached on the current game object i.e. on which you
will attach this script.
    */
    animator = GetComponent<Animator>();
}

void Update()
{
    /**
```

```
    * The Update() function will get the bool parameters from the animator state machine and
set the values provided by the user.
    * Here, I have only added animation for Run and Idle. When the Up key is pressed, Run
animation is played. When we let go, Idle is played.
    */

    if (Input.GetKey (KeyCode.UpArrow)) {
            animator.SetBool ("PerformRun", true);
            animator.SetBool ("PerformIdle", false);
        } else {
            animator.SetBool ("PerformRun", false);
            animator.SetBool ("PerformIdle", true);
        }
}

void OnAnimatorMove()
    {
        /**
         * OnAnimatorMove() function will shadow the "Apply Root Motion" on the animator. Your
game objects psoition will now be determined
         * using this fucntion.
         */
        if (Input.GetKey (KeyCode.UpArrow)){
            transform.Translate (Vector3.forward * RunSpeed);
            if (Input.GetKey (KeyCode.RightArrow)) {
                transform.Rotate (Vector3.up * Time.deltaTime * TurnSpeed);
            }
            else if (Input.GetKey (KeyCode.LeftArrow)) {
                transform.Rotate (-Vector3.up * Time.deltaTime * TurnSpeed);
            }
        }

    }



}
```

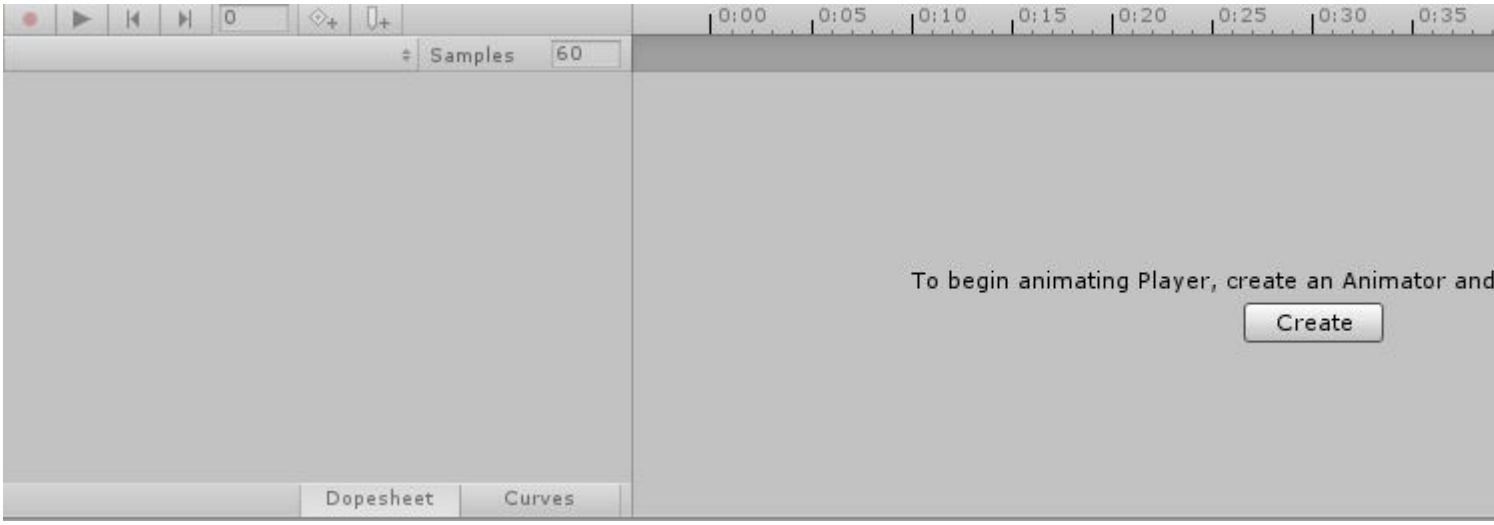


Создание и использование анимационных клипов

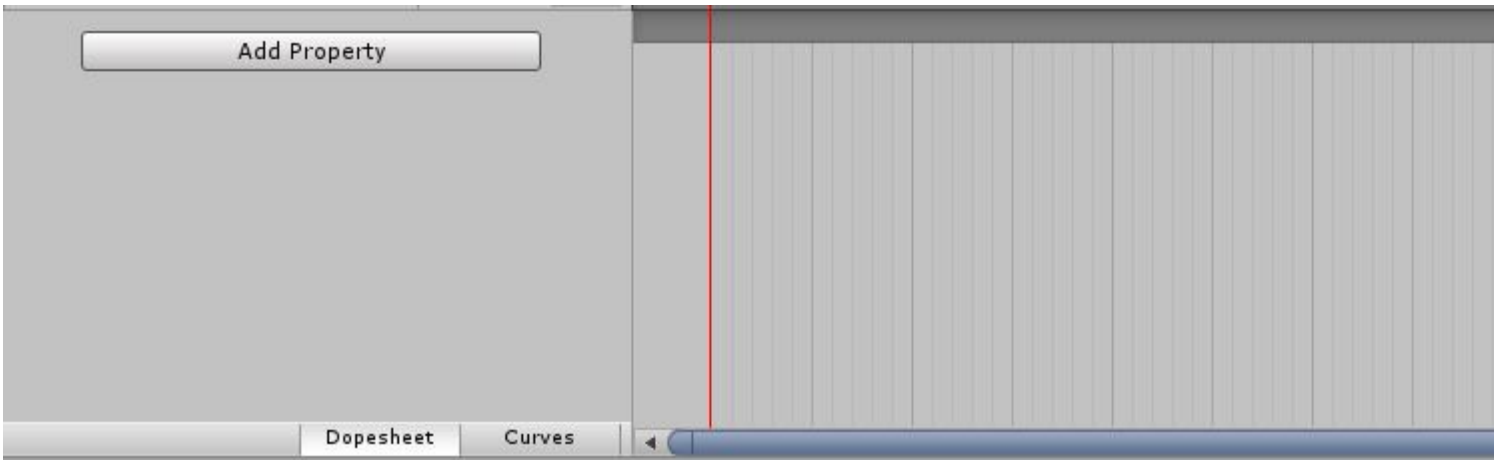
В этом примере будет показано, как создавать и использовать клипы анимации для игровых объектов или игроков.

Обратите внимание: модели, используемые в этом примере, загружаются из Unity Asset Store. Игрок был загружен со следующей ссылки: https://www.assetstore.unity3d.com/ru/#!/content/21874 .

Чтобы создать анимацию, сначала откройте окно анимации. Вы можете открыть его, нажав «Окно» и «Выбрать анимацию» или нажмите Ctrl + 6. Выберите игровой объект, к которому вы хотите применить клип анимации, из окна «Иерархия», а затем нажмите кнопку «Создать» в окне анимации.

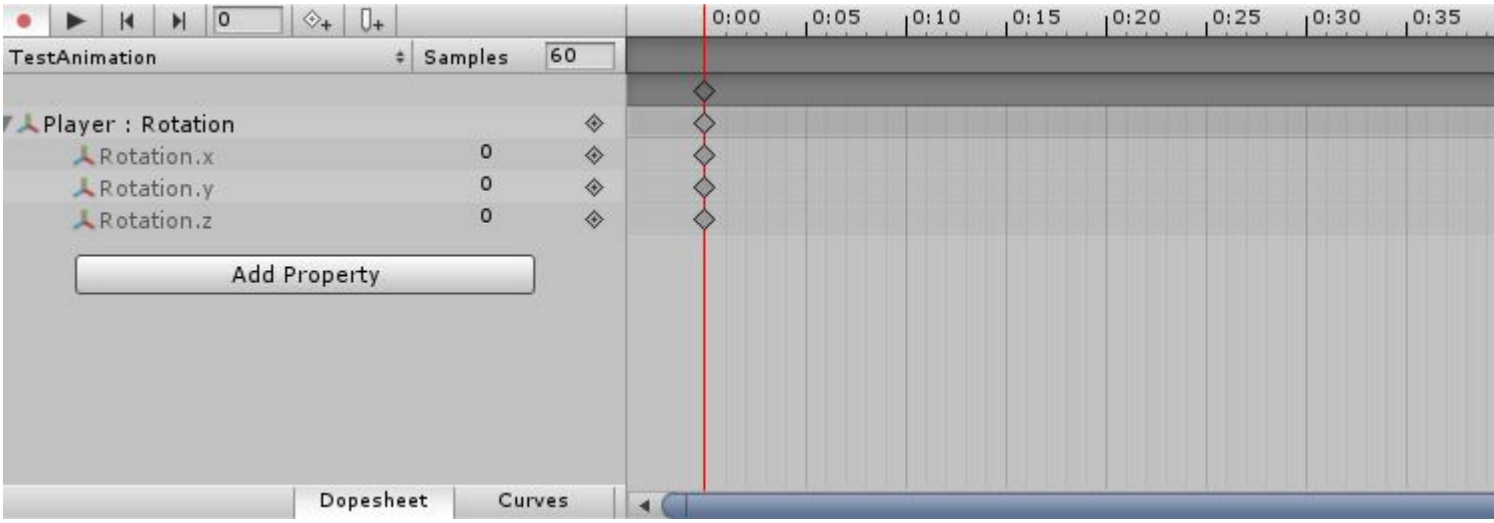


Назовите свою анимацию (например, IdlePlayer, SprintPlayer, DyingPlayer и т. Д.) И сохраните ее. Теперь из окна анимации нажмите кнопку «Добавить свойство». Это позволит вам изменить свойство игрового объекта или игрока по времени. Это может включать в себя Tranforms, такие как вращение, положение и масштаб, а также любое другое свойство, которое привязано к игровому объекту, например, коллайдер, Mesh Renderer и т. Д.



Чтобы создать запущенную анимацию для игрового объекта, вам понадобится гуманоидальная 3D-модель. Вы можете загрузить модель из приведенной выше ссылки. Следуйте приведенным выше шагам, чтобы создать новую анимацию. Добавьте свойство

Transform и выберите Rotation для одной из ног персонажа.



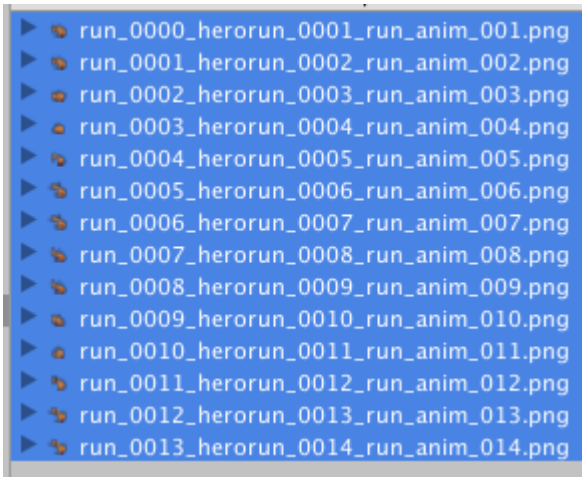
В этот момент ваши кнопки Play и Rotation в свойстве игрового объекта стали бы красными. Нажмите стрелку вниз, чтобы увидеть значения вращения X, Y и Z. Время анимации по умолчанию равно 1 секунде. Анимации используют ключевые кадры для интерполяции между значениями. Чтобы оживить, добавьте ключи в разные моменты времени и измените значения вращения из окна «Инспектор». Например, значение вращения в момент 0.0 может быть 0.0. В момент 0.5s значение может быть 20.0 для X. В момент 1.0s значение может быть 0.0. Мы можем завершить анимацию в 1.0s.

Длина вашей анимации зависит от последних добавленных к анимации клавиш. Вы можете добавить больше клавиш, чтобы сделать интерполяцию более плавной.

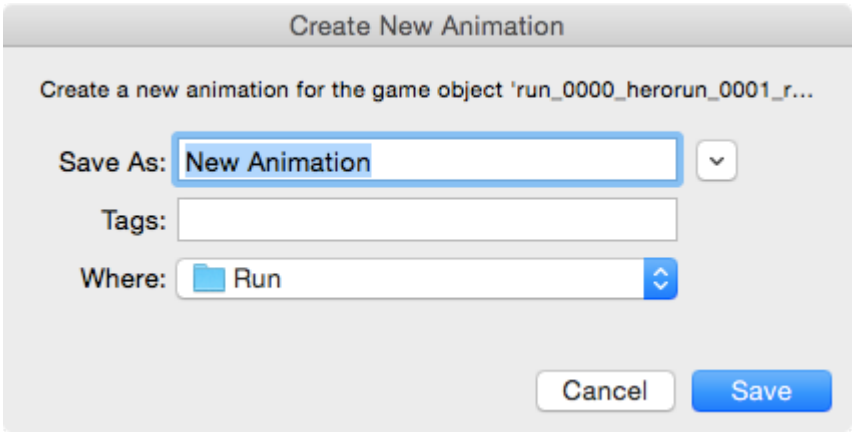
2D-анимация Sprite

Анимация Sprite состоит в показе существующей последовательности изображений или фреймов.

Сначала импортируйте последовательность изображений в папку ресурса. Либо создайте некоторые изображения с нуля, либо загрузите их из хранилища активов. (В этом примере используется этот бесплатный актив .)

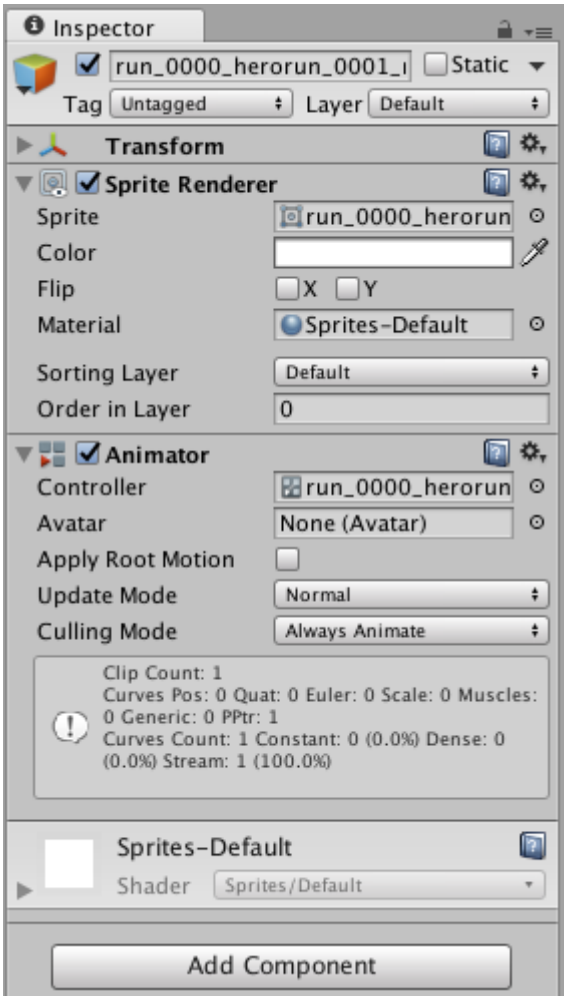


Перетащите каждое отдельное изображение одной анимации из папки ресурсов в представление сцены. Unity покажет диалог для именования нового клипа анимации.



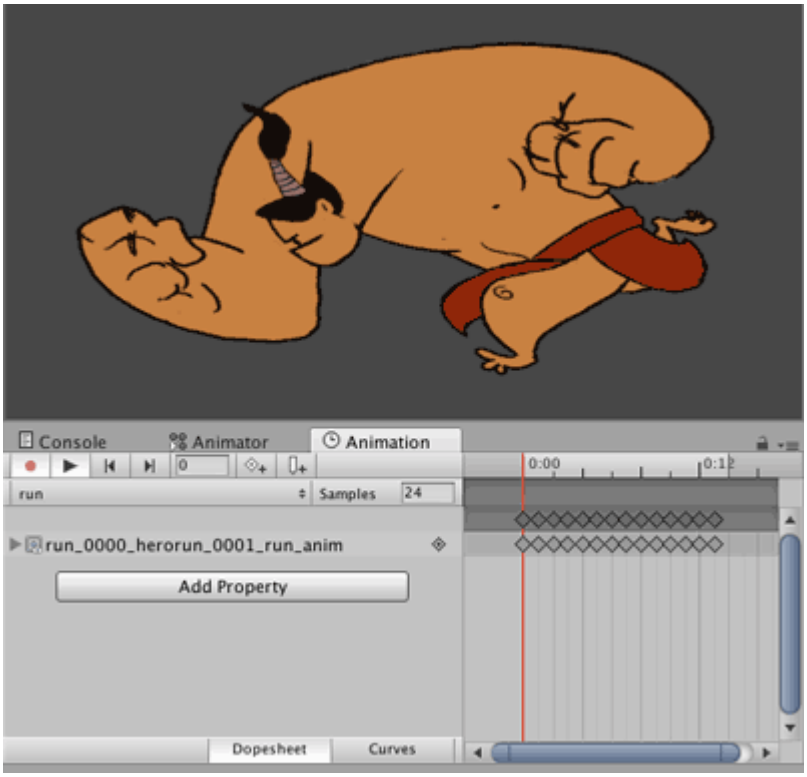
Это полезный ярлык для:

- создание новых игровых объектов
- назначая два компонента (Sprite Renderer и Animator)
- создание анимационных контроллеров (и привязка нового компонента Animator к ним)
- создание анимационных клипов с выбранными кадрами



Предварительный просмотр воспроизведения на вкладке анимации, нажав кнопку

«Воспроизвести»:



Этот же метод можно использовать для создания новых анимаций для одного и того же игрового объекта, а затем для удаления нового игрового объекта и контроллера анимации. Добавьте новый анимационный клип в контроллер анимации этого объекта так же, как и с 3D-анимацией.

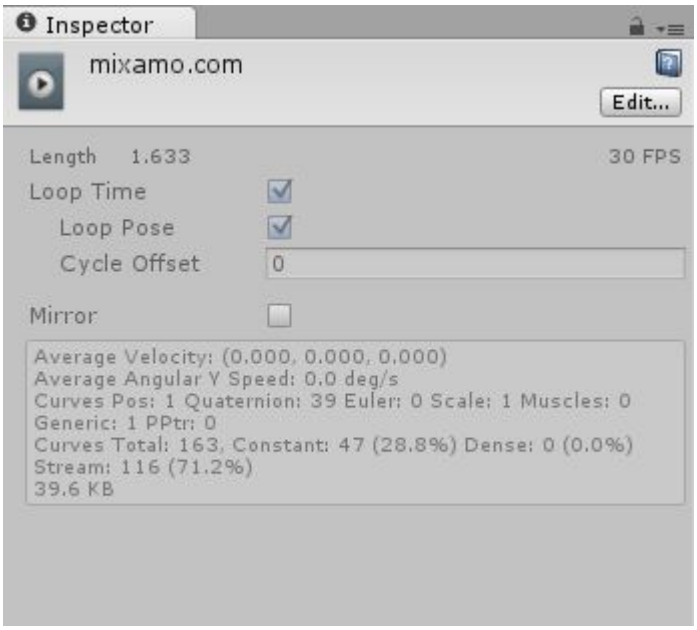
Анимационные кривые

Анимационные кривые позволяют вам изменять параметр float при воспроизведении анимации. Например, если есть анимация длительностью 60 секунд и вам нужно значение float / параметр, вызовите его X, чтобы измениться по анимации (например, во время анимации = 0.0, X = 0.0, во время анимации = 30,0 с; X = 1,0, во время анимации = 60,0 с, X = 0,0).

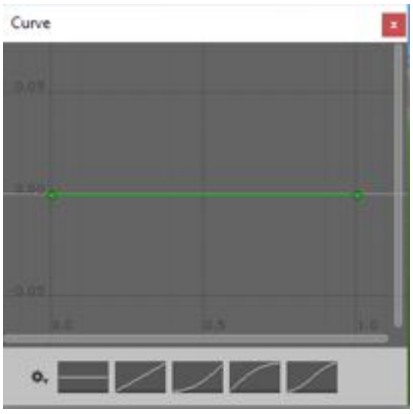
После того как вы получили значение float, вы можете использовать его для перевода, поворота, масштабирования или использования его любым другим способом.

Для моего примера я покажу игровой игровой объект. Когда анимация для запуска воспроизводится, скорость перевода игрока должна возрастать по мере продолжения анимации. Когда анимация достигает своего конца, скорость перевода должна уменьшаться.

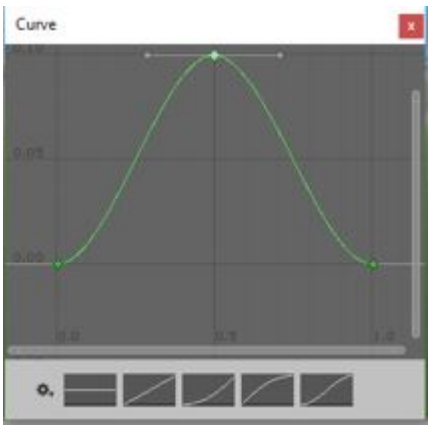
У меня создан запущенный анимационный клип. Выберите клип, а затем в окне инспектора нажмите «Изменить».



После этого прокрутите вниз до Curves. Нажмите на знак +, чтобы добавить кривую. Назовите кривую, например, ForwardRunCurve. Нажмите на миниатюрную кривую справа. Он откроет небольшое окно с кривой по умолчанию.



Нам нужна параболическая кривая, где она поднимается, а затем падает. По умолчанию на линии есть 2 точки. Вы можете добавить больше очков, дважды щелкнув по кривой. Перетащите точки, чтобы создать форму, похожую на следующую.



В окне Animator добавьте текущий клип. Также добавьте параметр float с тем же именем, что и кривая, т.е. ForwardRunCurve.

Когда анимация воспроизводится, значение поплавка изменяется в соответствии с кривой. Следующий код покажет, как использовать значение float:

```
using UnityEngine;
using System.Collections;

public class RunAnimation : MonoBehaviour {

Animator animator;
float curveValue;

void Start()
{
    animator = GetComponent<Animator>();
}

void Update()
{
    curveValue = animator.GetFloat("ForwardRunCurve");

    transform.Translate (Vector3.forward * curveValue);
}

}
```

Переменная curveValue удерживает значение кривой (ForwardRunCruve) в любой момент времени. Мы используем это значение для изменения скорости перевода. Вы можете прикрепить этот скрипт к игровому объекту игрока.

Прочитайте Unity Animation онлайн: https://riptutorial.com/ru/unity3d/topic/5448/unity-animation

глава 7: Unity Profiler

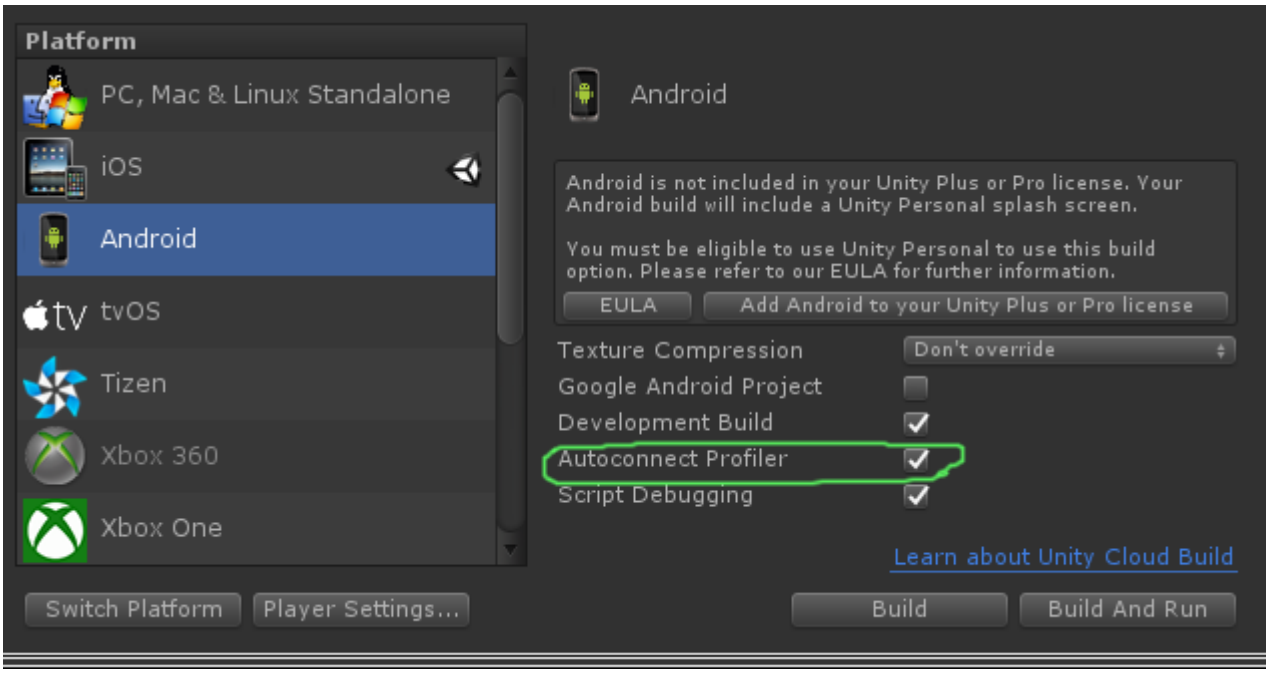
замечания

Использование Profiler на другом устройстве

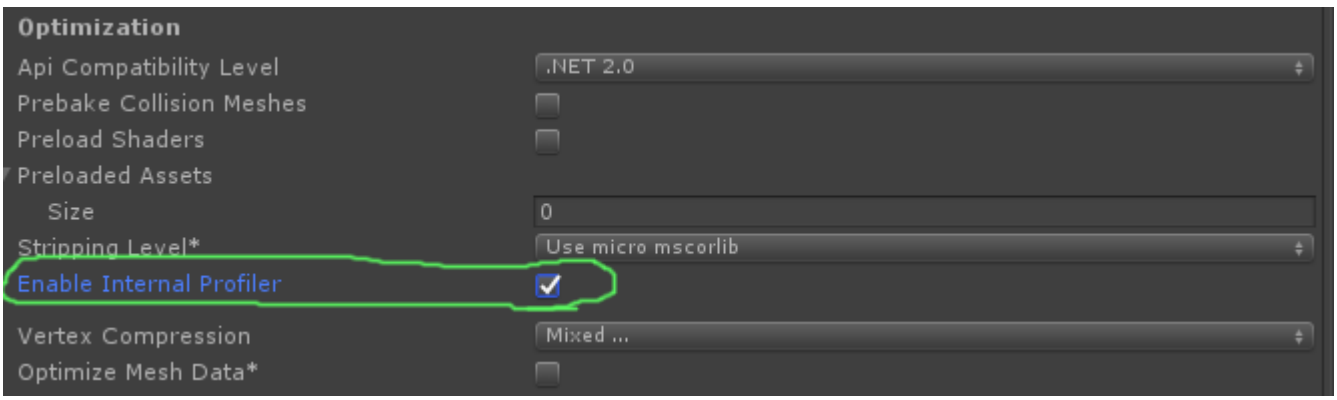
Есть несколько вещей, которые нужно знать, чтобы правильно подключить Profiler на разных платформах.

Android

Для правильного прикрепления профиля необходимо использовать кнопку «Построить и запустить» в окне «Параметры сборки» с параметром **Autoconnect Profiler** .



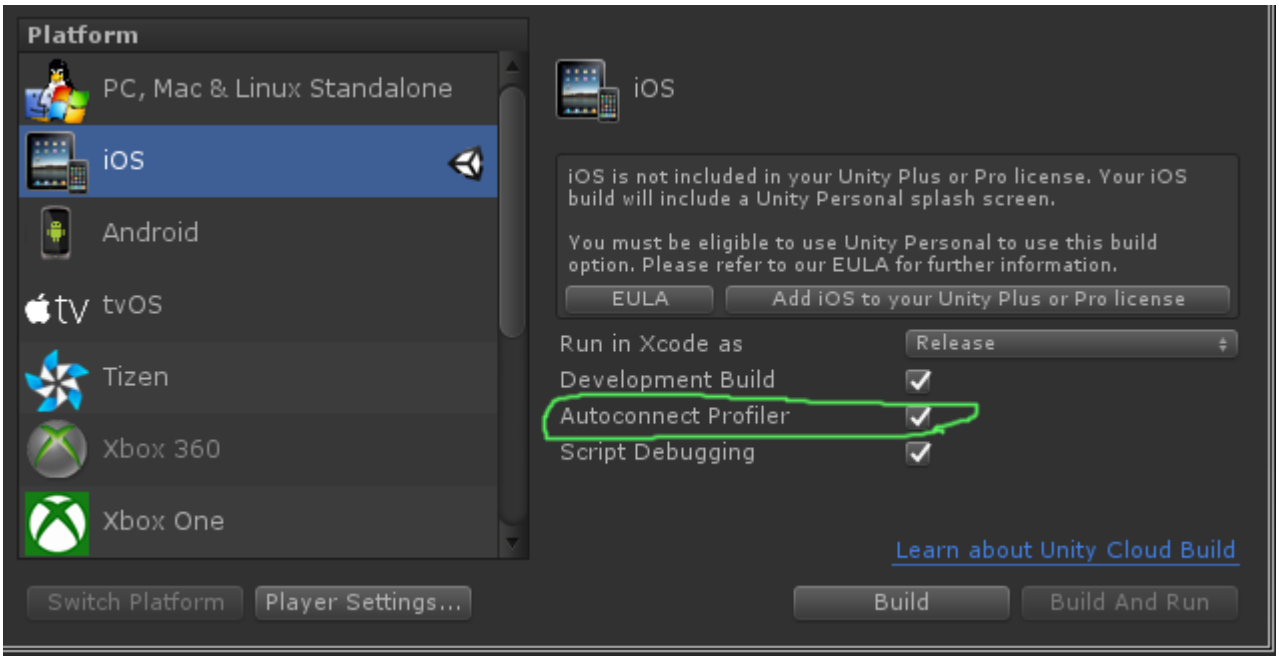
Еще один обязательный параметр, в инспекторе настроек Android Player на вкладке « Другие настройки», есть флажок « **Включить внутренний профайлер»,** который необходимо проверить, чтобы LogCat выдавал информацию о профилировщике.



Использование только «Build» не позволит профилировщику подключаться к устройству Android, потому что «Build and Run» использует определенные аргументы командной строки, чтобы запустить его с помощью LogCat.

IOS

Чтобы правильно прикрепить профиль, кнопка «Создать и запустить» в окне «Параметры сборки» с опцией « **Autoconnect Profiler»** должна быть проверена при первом запуске.



В iOS нет параметров в настройках проигрывателя, которые должны быть настроены для включения Profiler. Он должен работать из коробки.

Examples

Разметка профилировщика

Использование класса Profiler

Одна очень хорошая практика - использовать Profiler.BeginSample и Profiler.EndSample, потому что у него будет свой собственный вход в окне Profiler.

Кроме того, те теги будут удалены из сборки без разработки с использованием ConditionalAttribute, поэтому вам не нужно удалять их из вашего кода.

```
public class SomeClass : MonoBehaviour
{
    void SomeFunction()
    {
        Profiler.BeginSample("SomeClass.SomeFunction");
```

```
        // Various call made here
        Profiler.EndSample();
    }
}
```

Это создаст запись «SomeClass.SomeFunction» в окне Profiler, которая позволит упростить отладку и идентификацию бутылочной шейки.

Прочитайте Unity Profiler онлайн: https://riptutorial.com/ru/unity3d/topic/6974/unity-profiler

глава 8: Vector3

Вступление

Структура `Vector3` представляет собой трехмерную координату и является одной из `UnityEngine` библиотеки `UnityEngine` . Структура `Vector3` чаще всего встречается в компоненте `Transform` большинства игровых объектов, где используется для сохранения *положения* и *масштаба* . `Vector3` обеспечивает хорошую функциональность для выполнения общих векторных операций. Вы можете больше `Vector3` структуре `Vector3` в Unity API.

Синтаксис

- public Vector3 ();
- public Vector3 (float x, float y);
- public Vector3 (float x, float y, float z);
- Vector3.Lerp (Vector3 startPosition, Vector3 targetPosition, float movementFraction);
- Vector3.LerpUnclamped (Vector3 startPosition, Vector3 targetPosition, float movementFraction);
- Vector3.MoveTowards (Vector3 startPosition, Vector3 targetPosition, расстояние по плаванию);

Examples

Статические значения

Структура `Vector3` содержит некоторые статические переменные, которые предоставляют обычно используемые значения `Vector3` . Большинство из них представляют собой *направление* , но они все еще могут быть использованы творчески для обеспечения дополнительной функциональности.

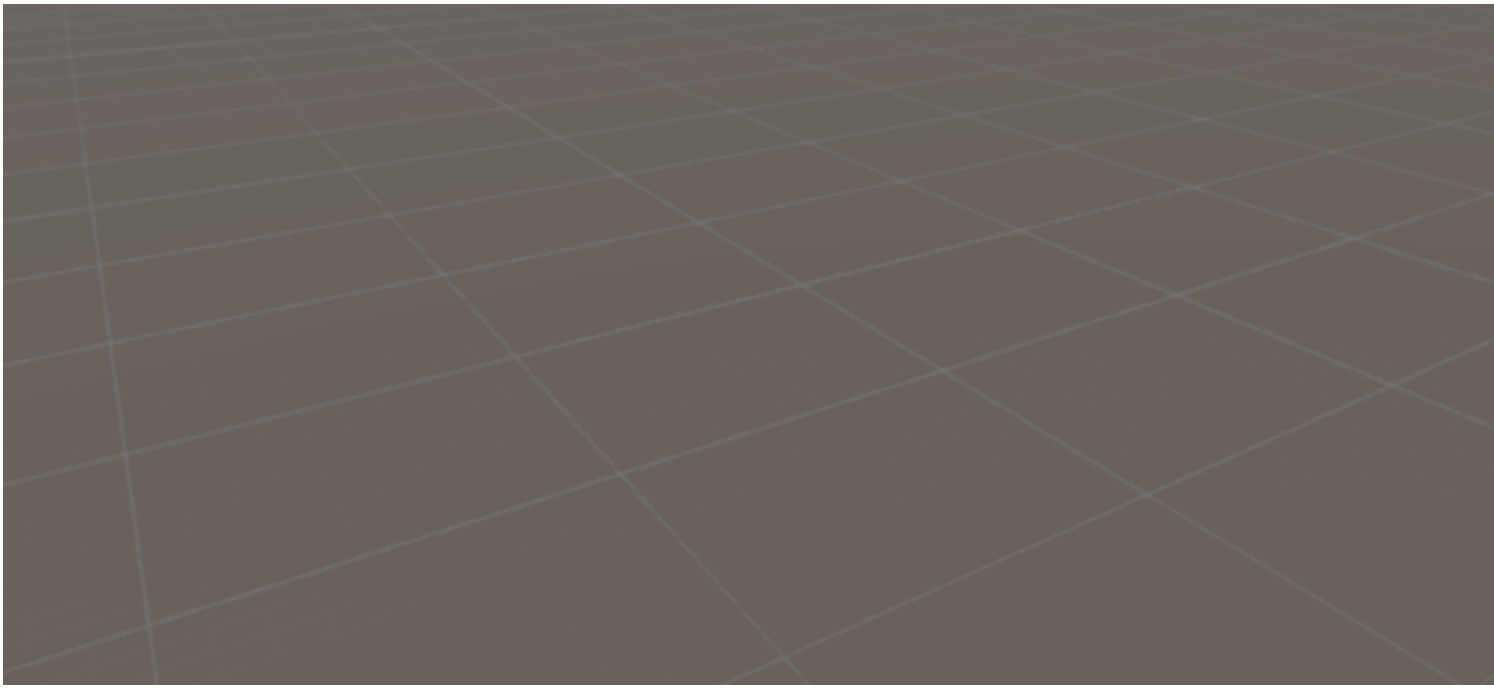
`Vector3.zero` И `Vector3.one`

`Vector3.zero` и `Vector3.one` обычно используются в связи с *нормализованным* `Vector3` ; то есть `Vector3` где значения `x` , `y` и `z` имеют величину 1. Таким образом, `Vector3.zero` представляет наименьшее значение, в то время как `Vector3.one` представляет наибольшее значение.

`Vector3.zero` также обычно используется для установки позиции по умолчанию для объектных преобразований.

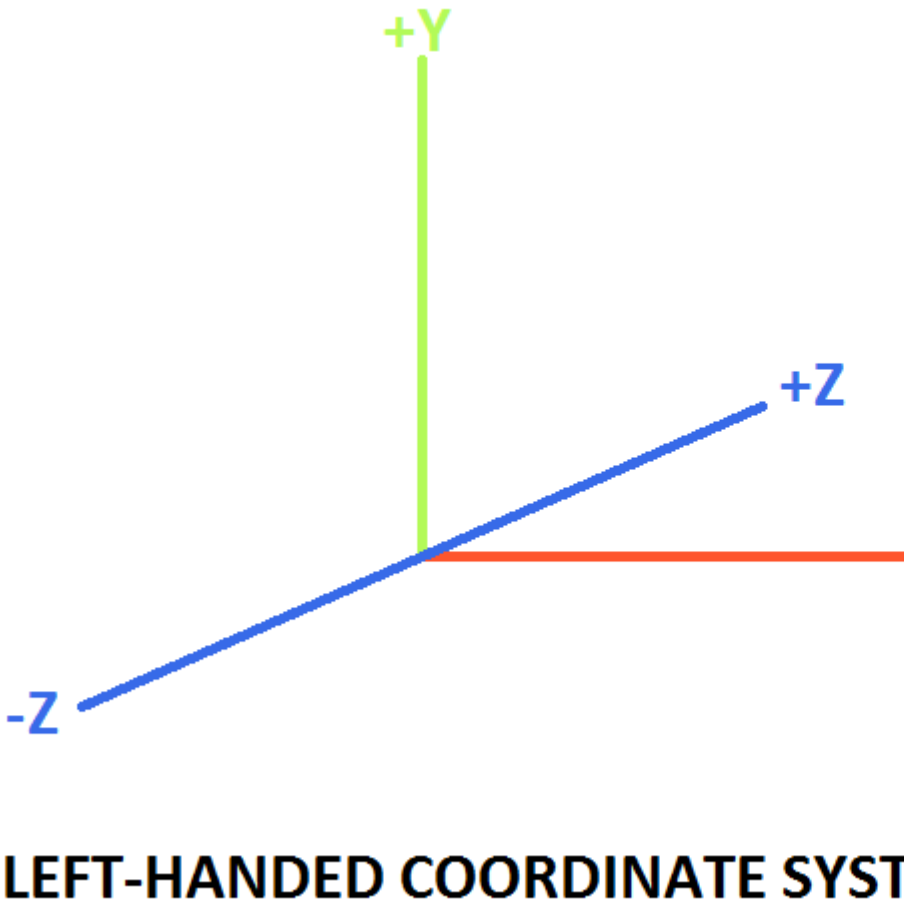
Следующий класс использует `Vector3.zero` и `Vector3.one` для раздувания и `Vector3.one` сферы.

```
using UnityEngine;

public class Inflater : MonoBehaviour
{
    <summary>A sphere set up to inflate and deflate between two values.</summary>
    public ScaleBetween sphere;

    ///<summary>On start, set the sphere GameObject up to inflate
    /// and deflate to the corresponding values.</summary>
    void Start()
    {
        // Vector3.zero = Vector3(0, 0, 0); Vector3.one = Vector3(1, 1, 1);
        sphere.SetScale(Vector3.zero, Vector3.one);
    }
}
```



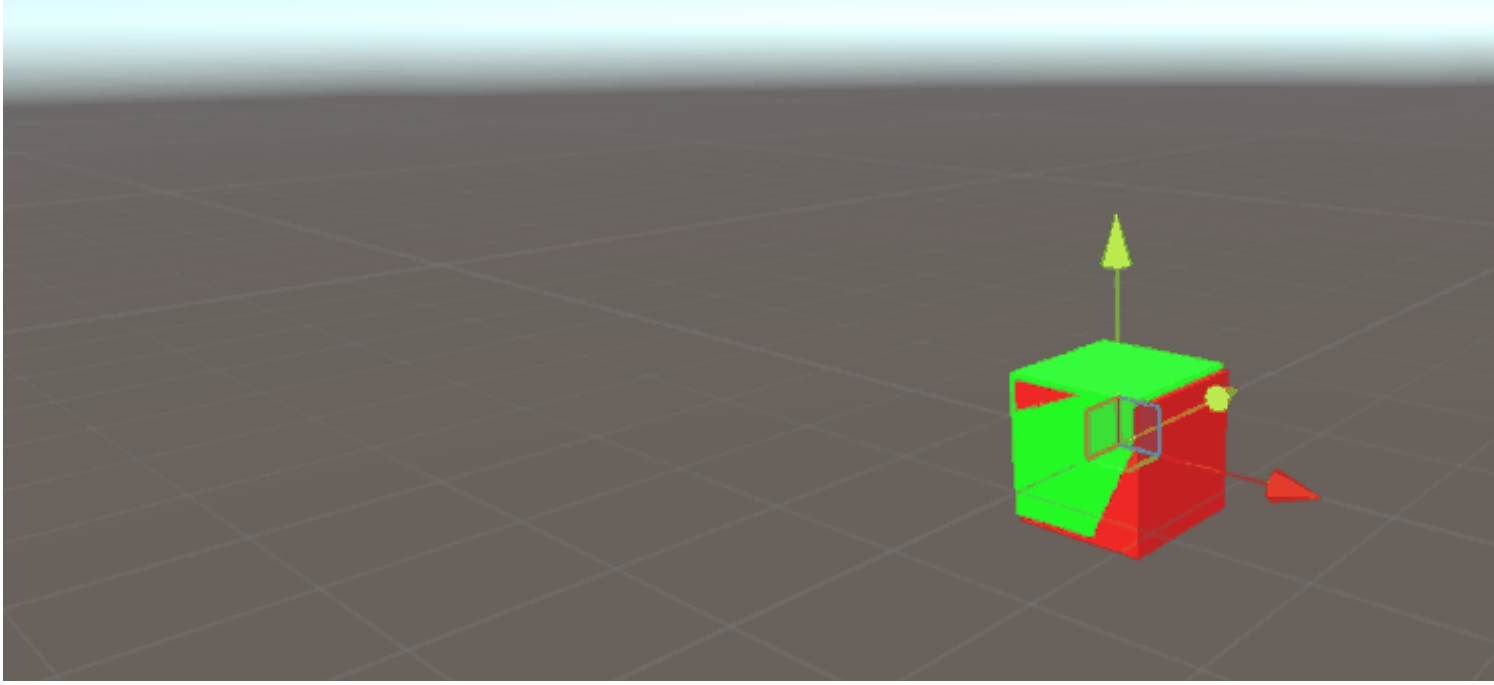
Статические направления

Статические направления могут быть полезны в ряде приложений с направлением вдоль положительной и отрицательной всех трех осей. Важно отметить, что Unity использует левую систему координат, которая влияет на направление.



Следующий класс использует статические направления `Vector3` для перемещения объектов вдоль трех осей.

```
using UnityEngine;

public class StaticMover : MonoBehaviour
{
    <summary>GameObjects set up to move back and forth between two directions.</summary>
    public MoveBetween xMovement, yMovement, zMovement;

    ///<summary>On start, set each MoveBetween GameObject up to move
    /// in the corresponding direction(s).</summary>
    void Start()
    {
        // Vector3.left = Vector3(-1, 0, 0); Vector3.right = Vector3(1, 0, 0);
        xMovement.SetDirections(Vector3.left, Vector3.right);

        // Vector3.down = Vector3(0, -1, 0); Vector3.up = Vector3(0, 0, 1);
        yMovement.SetDirections(Vector3.down, Vector3.up);

        // Vector3.back = Vector3(0, 0, -1); Vector3.forward = Vector3(0, 0, 1);
        zMovement.SetDirections(Vector3.back, Vector3.forward);
    }
}
```



Индекс

Значение	Икс	Y	Z	Эквивалентный `new Vector3()`
`Vector3.zero`	0	0	0	`new Vector3(0, 0, 0)`
`Vector3.one`	1	1	1	`new Vector3(1, 1, 1)`
`Vector3.left`	-1	0	0	`new Vector3(-1, 0, 0)`
`Vector3.right`	1	0	0	`new Vector3(1, 0, 0)`
`Vector3.down`	0	-1	0	`new Vector3(0, -1, 0)`
`Vector3.up`	0	1	0	`new Vector3(0, 1, 0)`
`Vector3.back`	0	0	-1	`new Vector3(0, 0, -1)`
`Vector3.forward`	0	0	1	`new Vector3(0, 0, 1)`

Создание Vector3

Структура `Vector3` может быть создана несколькими способами. `Vector3` является структурой, и как таковой, как правило, необходимо создать экземпляр перед использованием.

Конструкторы

Существует три встроенных конструктора для создания экземпляра `Vector3` .

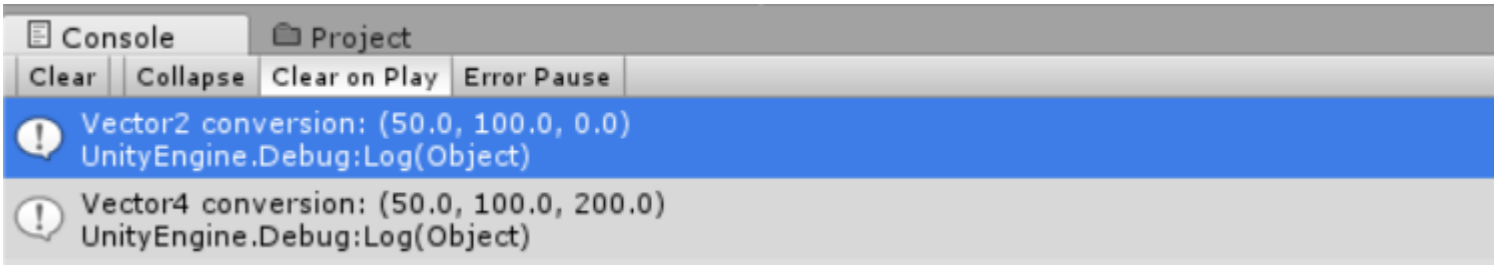
Конструктор	Результат
`new Vector3()`	Создает структуру `Vector3` с координатами (0, 0, 0).
`new Vector3(float x, float y)`	Создает структуру `Vector3` с заданными координатами `x` и `y` . `z` будет установлено в 0.
`new Vector3(float x, float y, float z)`	Создает структуру `Vector3` с заданными координатами `x` , `y` и `z` .

Преобразование из `Vector2` или `Vector4`

В редких случаях вы можете столкнуться с ситуациями, когда вам нужно будет обрабатывать координаты структуры `Vector2` или `Vector4` как `Vector3` . В таких случаях вы можете просто передать `Vector2` или `Vector4` непосредственно в `Vector3` , не `Vector3` ранее. Как следует полагать, структура `Vector2` будет передавать только значения `x` и `y` , тогда как класс `Vector4` будет опускать свой `w` .

Мы можем видеть прямое преобразование в приведенном ниже скрипте.

```
void VectorConversionTest()
{
    Vector2 vector2 = new Vector2(50, 100);
    Vector4 vector4 = new Vector4(50, 100, 200, 400);

    Vector3 fromVector2 = vector2;
    Vector3 fromVector4 = vector4;

    Debug.Log("Vector2 conversion: " + fromVector2);
    Debug.Log("Vector4 conversion: " + fromVector4);
}
```



Применение движения

Структура `Vector3` содержит некоторые статические функции, которые могут обеспечить полезность, если мы хотим применить движение к `Vector3` .

`Lerp` и `LerpUnclamped`

Функции lerp обеспечивают перемещение между двумя координатами, основанными на заданной фракции. Если `Lerp` разрешает движение между двумя координатами, `LerpUnclamped` позволяет фракциям, которые перемещаются за пределы границ между двумя координатами.

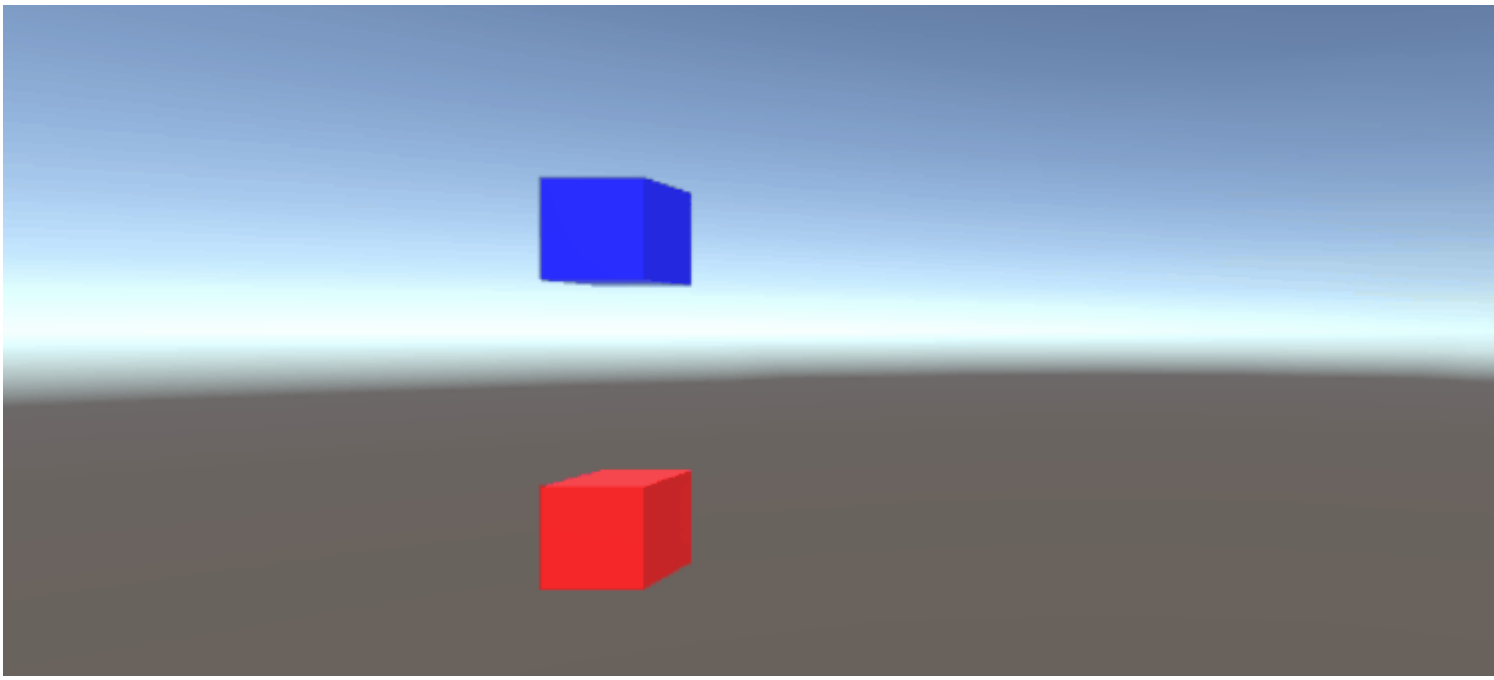
Мы предоставляем долю движения в виде `float` . При значении `0.5` мы находим середину между двумя `Vector3` . Значение `0` или `1` вернет первый или второй `Vector3` , respectivley, так как эти значения либо коррелируют с отсутствием движения (таким образом, возвращая первый `Vector3`), либо завершенным движением (это возвращает второй `Vector3`). Важно отметить, что ни одна из функций не будет учитываться при изменении доли движения. Это то, что нам нужно для учета вручную.

С `Lerp` все значения зажаты между `0` и `1` . Это полезно, когда мы хотим обеспечить движение к направлению и не хотим перерегулировать пункт назначения. `LerpUnclamped` может принимать любое значение, и может быть использован для обеспечения движения от пункта назначения, или *мимо* цели.

Следующий сценарий использует `Lerp` и `LerpUnclamped` для `LerpUnclamped` перемещения объекта.

```
using UnityEngine;

public class Lerping : MonoBehaviour
{
    /// <summary>The red box will use Lerp to move. We will link
    /// this object in via the inspector.</summary>
    public GameObject lerpObject;
    /// <summary>The starting position for our red box.</summary>
    public Vector3 lerpStart = new Vector3(0, 0, 0);
    /// <summary>The end position for our red box.</summary>
    public Vector3 lerpTarget = new Vector3(5, 0, 0);

    /// <summary>The blue box will use LerpUnclamped to move. We will
    /// link this object in via the inspector.</summary>
    public GameObject lerpUnclampedObject;
    /// <summary>The starting position for our blue box.</summary>
    public Vector3 lerpUnclampedStart = new Vector3(0, 3, 0);
    /// <summary>The end position for our blue box.</summary>
    public Vector3 lerpUnclampedTarget = new Vector3(5, 3, 0);

    /// <summary>The current fraction to increment our lerp functions by.</summary>
    public float lerpFraction = 0;
```

```
    private void Update()
    {
        // First, I increment the lerp fraction.
        // delaTime * 0.25 should give me a value of +1 every second.
        lerpFraction += (Time.deltaTime * 0.25f);

        // Next, we apply the new lerp values to the target transform position.
        lerpObject.transform.position
            = Vector3.Lerp(lerpStart, lerpTarget, lerpFraction);
        lerpUnclampedObject.transform.position
            = Vector3.LerpUnclamped(lerpUnclampedStart, lerpUnclampedTarget, lerpFraction);
    }
}
```



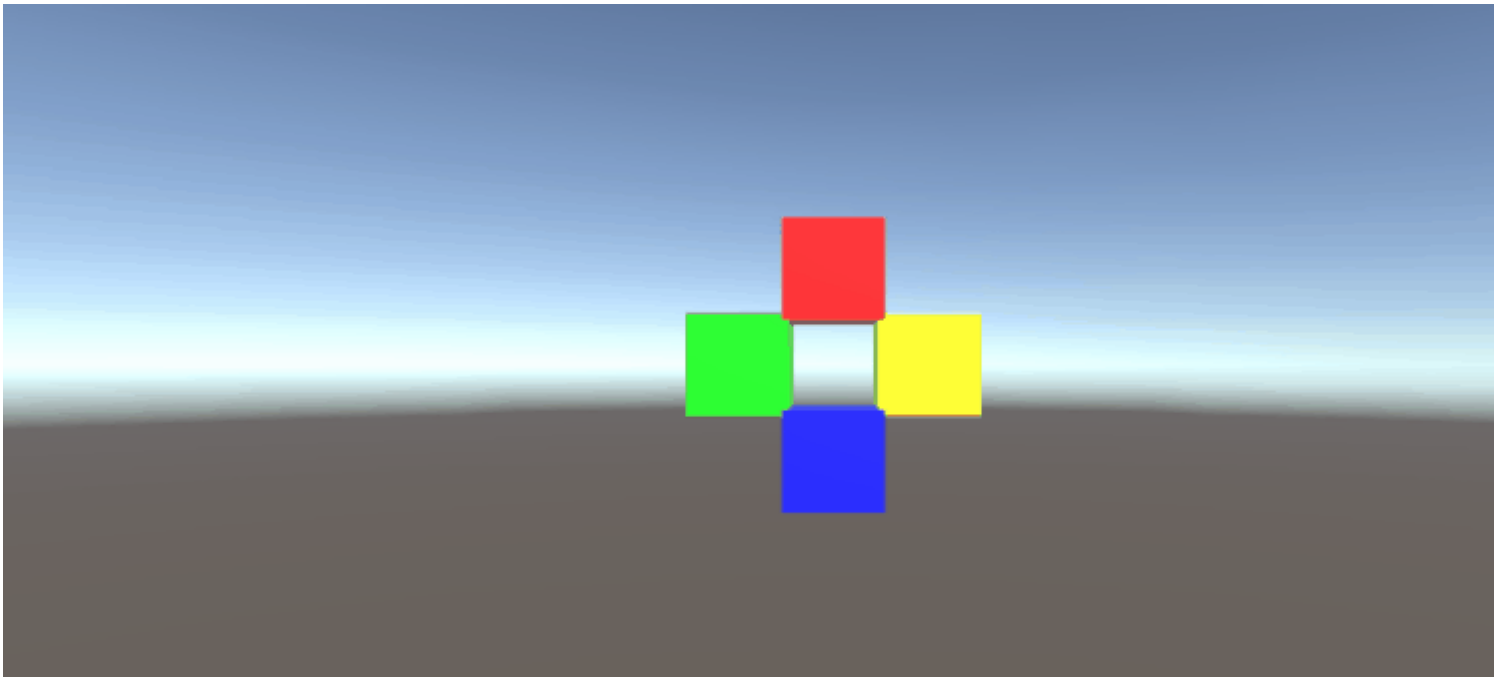
MoveTowards

`MoveTowards` ведет себя *очень похоже* на `Lerp` ; основное отличие состоит в том, что мы предоставляем фактическое *расстояние* для перемещения вместо *доли* между двумя точками. Важно отметить, что `MoveTowards` не будет проходить мимо целевого `Vector3` .

Так же, как с `LerpUnclamped` , мы можем обеспечить *отрицательное* значение расстояния , чтобы *отойти* от цели `Vector3` . В таких случаях мы никогда не двигаемся мимо целевого `Vector3` , и, следовательно, движение неопределено. В этих случаях мы можем рассматривать целевой `Vector3` как «противоположное направление»; пока `Vector3` указывает в том же направлении, относительно начала `Vector3` , отрицательное движение должно вести себя как обычно.

Следующий скрипт использует `MoveTowards` для перемещения группы объектов по отношению к набору позиций с использованием сглаженного расстояния.

```
using UnityEngine;

public class MoveTowardsExample : MonoBehaviour
{
    /// <summary>The red cube will move up, the blue cube will move down,
    /// the green cube will move left and the yellow cube will move right.
    /// These objects will be linked via the inspector.</summary>
    public GameObject upCube, downCube, leftCube, rightCube;
    /// <summary>The cubes should move at 1 unit per second.</summary>
    float speed = 1f;

    void Update()
    {
        // We determine our distance by applying a deltaTime scale to our speed.
        float distance = speed * Time.deltaTime;

        // The up cube will move upwards, until it reaches the
        //position of (Vector3.up * 2), or (0, 2, 0).
        upCube.transform.position
            = Vector3.MoveTowards(upCube.transform.position, (Vector3.up * 2f), distance);

        // The down cube will move downwards, as it enforces a negative distance..
        downCube.transform.position
            = Vector3.MoveTowards(downCube.transform.position, Vector3.up * 2f, -distance);

        // The right cube will move to the right, indefinetly, as it is constantly updating
        // its target position with a direction based off the current position.
        rightCube.transform.position = Vector3.MoveTowards(rightCube.transform.position,
            rightCube.transform.position + Vector3.right, distance);

        // The left cube does not need to account for updating its target position,
        // as it is moving away from the target position, and will never reach it.
        leftCube.transform.position
            = Vector3.MoveTowards(leftCube.transform.position, Vector3.right, -distance);
    }
}
```



SmoothDamp

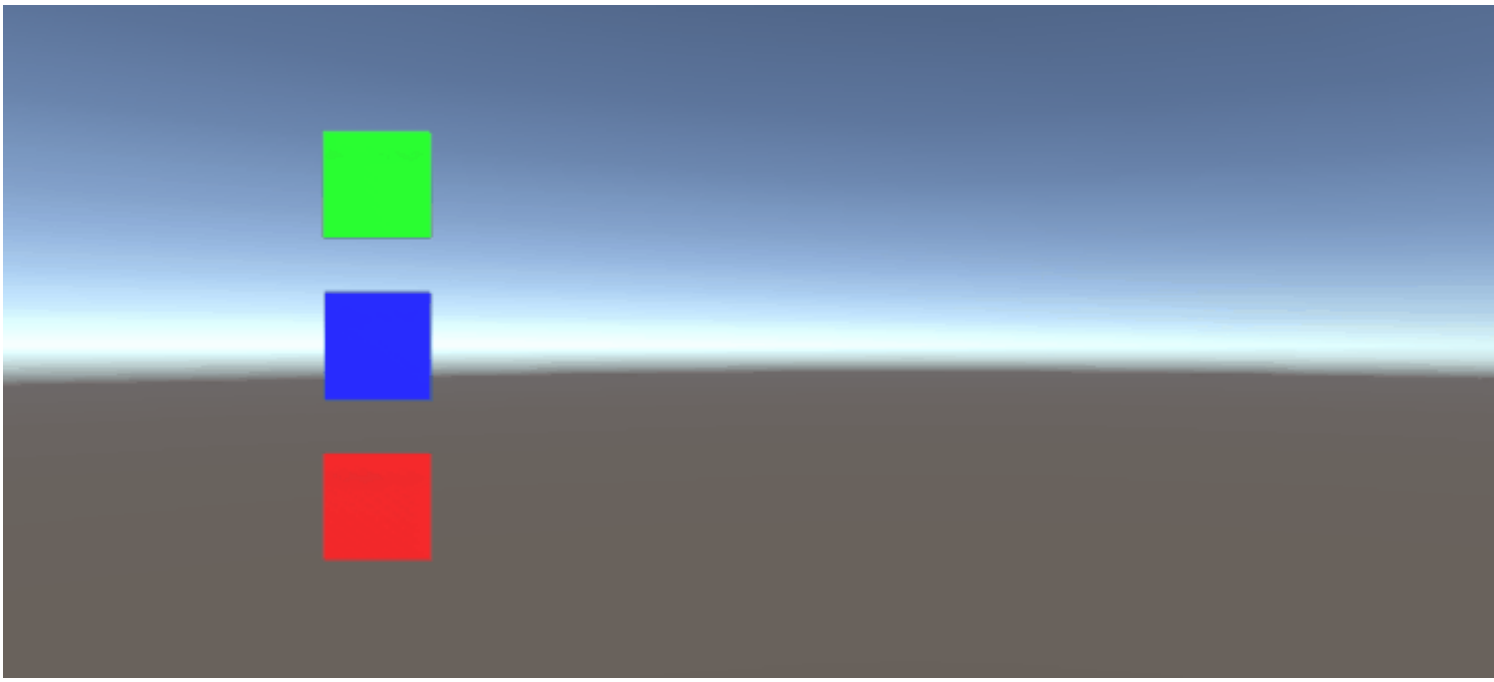
Подумайте о `SmoothDamp` как о варианте `MoveTowards` со встроенным сглаживанием. Согласно официальной документации, эта функция чаще всего используется для обеспечения гладкой камеры.

Наряду с начальными и целевыми координатами `Vector3` мы также должны предоставить `Vector3` для представления скорости, а `float` - *приблизительное* время, необходимое для завершения движения. В отличие от предыдущих примеров, мы предоставляем скорость в качестве *эталона* , чтобы быть увеличенным внутри. Важно отметить это, поскольку изменение скорости вне функции, пока мы все еще выполняем функцию, может иметь нежелательные результаты.

В дополнение к *требуемым* переменным мы также можем предоставить `float` для представления максимальной скорости нашего объекта и `float` для представления временного промежутка с момента предыдущего вызова `SmoothDamp` . Нам не *нужно* предоставлять эти значения; по умолчанию не будет максимальной скорости, а временной интервал будет интерпретироваться как `Time.deltaTime` . Что еще более важно, если вы вызываете функцию по одному объекту внутри функции `MonoBehaviour.Update()` , вам *не* нужно `MonoBehaviour.Update()` промежуток времени.

```
using UnityEngine;

public class SmoothDampMovement : MonoBehaviour
{
    /// <summary>The red cube will imitate the default SmoothDamp function.
    /// The blue cube will move faster by manipulating the "time gap", while
    /// the green cube will have an enforced maximum speed. Note that these
    /// objects have been linked via the inspector.</summary>
    public GameObject smoothObject, fastSmoothObject, cappedSmoothObject;

    /// <summary>We must instantiate the velocities, externally, so they may
    /// be manipulated from within the function. Note that by making these
    /// vectors public, they will be automatically instantiated as Vector3.Zero
    /// through the inspector. This also allows us to view the velocities,
    /// from the inspector, to observe how they change.</summary>
    public Vector3 regularVelocity, fastVelocity, cappedVelocity;

    /// <summary>Each object should move 10 units along the X-axis.</summary>
    Vector3 regularTarget = new Vector3(10f, 0f);
    Vector3 fastTarget = new Vector3(10f, 1.5f);
    Vector3 cappedTarget = new Vector3(10f, 3f);

    /// <summary>We will give a target time of 5 seconds.</summary>
    float targetTime = 5f;

    void Update()
    {
        // The default SmoothDamp function will give us a general smooth movement.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime);
```

```
        // Note that a "maxSpeed" outside of reasonable limitations should not have any
        // effect, while providing a "deltaTime" of 0 tells the function that no time has
        // passed since the last SmoothDamp call, resulting in no movement, the second time.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime, 10f, 0f);

        // Note that "deltaTime" defaults to Time.deltaTime due to an assumption that this
        // function will be called once per update function. We can call the function
        // multiple times during an update function, but the function will assume that enough
        // time has passed to continue the same approximate movement. As a result,
        // this object should reach the target, quicker.
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);

        // Lastly, note that a "maxSpeed" becomes irrelevant, if the object does not
        // realistically reach such speeds. Linear speed can be determined as
        // (Distance / Time), but given the simple fact that we start and end slow, we can
        // infer that speed will actually be higher, during the middle. As such, we can
        // infer that a value of (Distance / Time) or (10/5) will affect the
        // function. We will half the "maxSpeed", again, to make it more noticeable.
        cappedSmoothObject.transform.position = Vector3.SmoothDamp(
            cappedSmoothObject.transform.position,
            cappedTarget, ref cappedVelocity, targetTime, 1f);
    }
}
```



Прочитайте Vector3 онлайн: https://riptutorial.com/ru/unity3d/topic/7827/vector3

глава 9: Атрибуты

Синтаксис

- [AddComponentMenu (string menuName)]
- [AddComponentMenu (string menuName, int order)]
- [CanEditMultipleObjects]
- [ContextMenuItem (имя строки, строковая функция)]
- [ContextMenu (имя строки)]
- [CustomEditor (Тип inspectedType)]
- [CustomEditor (Тип inspectedType, bool editorForChildClasses)]
- [CustomPropertyDrawer (Тип типа)]
- [CustomPropertyDrawer (Тип типа, bool useForChildren)]
- [DisallowMultipleComponent]
- [DrawGizmo (GizmoType gizmo)]
- [DrawGizmo (GizmoType gizmo, Тип drawGizmoType)]
- [ExecuteInEditMode]
- [Заголовок (заголовок строки)]
- [HideInInspector]
- [InitializeOnLoad]
- [InitializeOnLoadMethod]
- [MenuItem (string itemName)]
- [MenuItem (string itemName, bool isValidateFunction)]
- [MenuItem (string itemName, bool isValidateFunction, int priority)]
- [Многострочные (int lines)]
- [PreferenceItem (имя строки)]
- [Диапазон (поплавок мин., Макс. Поплавок)]
- [RequireComponent (Тип типа)]
- [RuntimeInitializeOnLoadMethod]
- [RuntimeInitializeOnLoadMethod (RuntimeInitializeLoadType loadType)]
- [SerializeField]
- [Пробел (высота поплавка)]
- [TextArea (int minLines, int maxLines)]
- [Всплывающая подсказка (строка подсказки)]

замечания

SerializeField

Система сериализации Unity может использоваться для выполнения следующих действий:

- **Может** сериализовать общепринятые нестатические поля (сериализуемых типов)
- **Может** сериализовать непубличные нестатические поля, отмеченные атрибутом [SerializeField]
- **Невозможно** сериализовать статические поля
- **Невозможно** сериализовать статические свойства

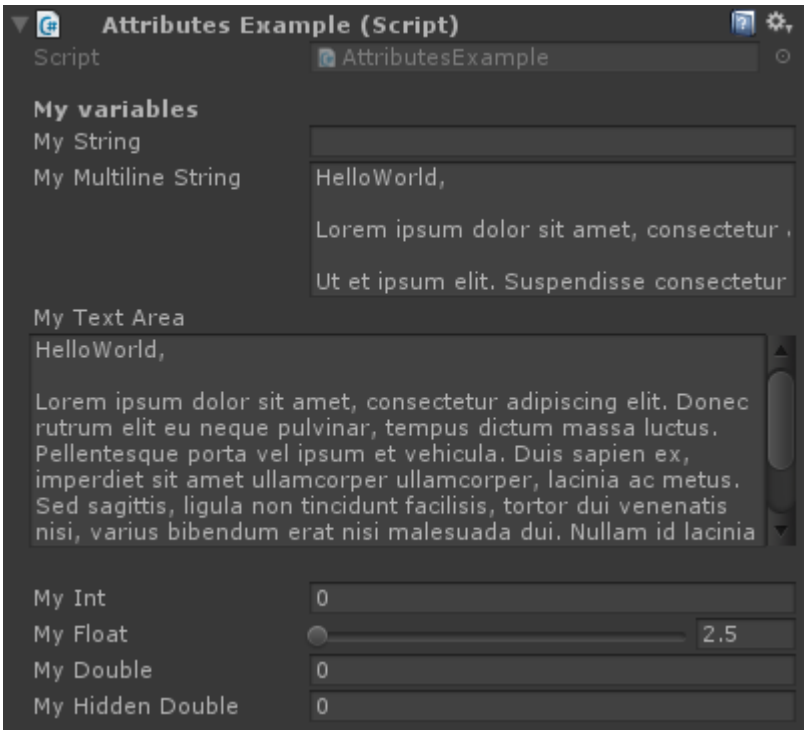
Ваше поле, даже если оно отмечено с помощью атрибута SerializeField, будет отнесено только в том случае, если оно относится к типу, который Unity может сериализовать, а именно:

- Все классы, наследующие от UnityEngine.Object (например, GameObject, Component, MonoBehaviour, Texture2D)
- Все основные типы данных, такие как int, string, float, bool
- Некоторые встроенные типы, такие как Vector2 / 3/4, Quaternion, Matrix4x4, Color, Rect, LayerMask
- Массивы сериализуемого типа
- Список сериализуемого типа
- Перечисления
- Структуры

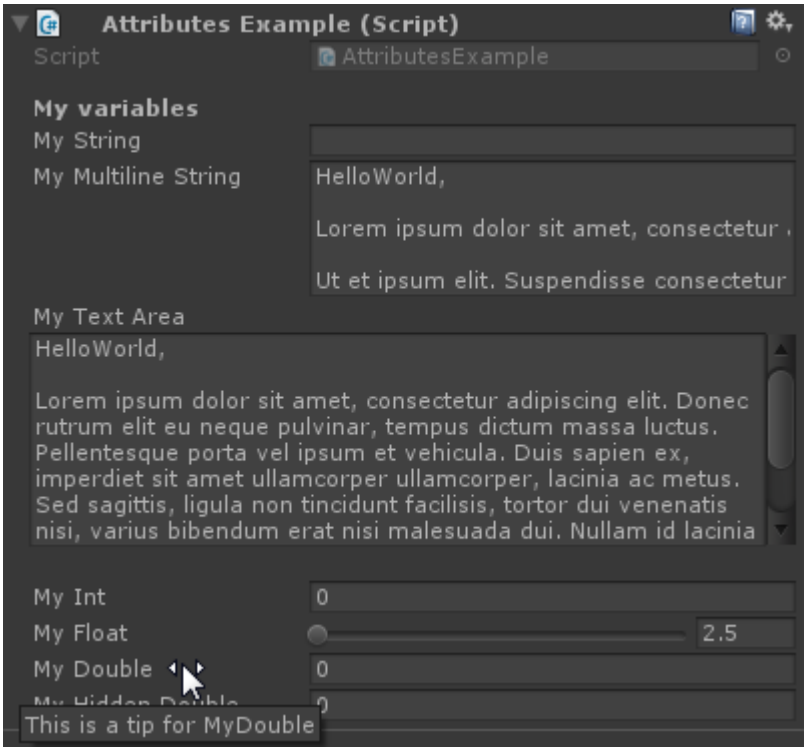
Examples

Общие атрибуты инспектора

```
[Header( "My variables" )]
public string MyString;

[HideInInspector]
public string MyHiddenString;

[Multiline( 5 )]
public string MyMultilineString;

[TextArea( 2, 8 )]
public string MyTextArea;

[Space( 15 )]
public int MyInt;

[Range( 2.5f, 12.5f )]
public float MyFloat;

[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;

[SerializeField]
private double myHiddenDouble;
```



При наведении указателя на поле:



```
[Header( "My variables" )]
public string MyString;
```

Заголовок помещает жирный ярлык, содержащий текст над приписанным полем. Это часто используется для маркировки групп, чтобы сделать их отличными от других меток.

```
[HideInInspector]
public string MyHiddenString;
```

HideInInspector не позволяет публичным полям отображаться в инспекторе. Это полезно для доступа к полям из других частей кода, где они иначе не видны или не изменяются.

```
[Multiline( 5 )]
public string MyMultilineString;
```

Многострочный создает текстовое поле с указанным количеством строк. Превышение этой суммы не приведет ни к расширению окна, ни к переносу текста.

```
[TextArea( 2, 8 )]
public string MyTextArea;
```

TextArea позволяет использовать текст в многострочном стиле с автоматическим оберткой и полосами прокрутки, если текст превышает выделенную область.

```
[Space( 15 )]
public int MyInt;
```

Космос заставляет инспектора добавлять дополнительное пространство между предыдущими и текущими объектами - полезными для разграничения и разделения групп.

```
[Range( 2.5f, 12.5f )]
public float MyFloat;
```

Диапазон задает числовое значение между минимумом и максимумом. Этот атрибут также работает с целыми числами и удвоениями, хотя min и max указаны как float.

```
[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;
```

Подсказка показывает дополнительное описание всякий раз, когда надпись поля зависнет.

```
[SerializeField]
private double myHiddenDouble;
```

SerializeField заставляет Unity сериализовать поле - полезно для частных полей.

Атрибуты компонентов

```
[DisallowMultipleComponent]
[RequireComponent( typeof( Rigidbody ) )]
public class AttributesExample : MonoBehaviour
{
    [...]
}
```

```
[DisallowMultipleComponent]
```

Атрибут DisallowMultipleComponent запрещает пользователям добавлять несколько экземпляров этого компонента в один GameObject.

```
[RequireComponent( typeof( Rigidbody ) )]
```

Атрибут RequireComponent позволяет указать другой компонент (или более) как требования для того, когда этот компонент добавлен в GameObject. Когда вы добавляете этот компонент в GameObject, необходимые компоненты будут автоматически добавлены (если они еще не присутствуют), и эти компоненты не могут быть удалены до тех пор, пока тот, который их требует, будет удален.

Атрибуты времени выполнения

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
{

    [RuntimeInitializeOnLoadMethod]
    private static void FooBar()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
    private static void Foo()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
    private static void Bar()
    {
        [...]
    }

    void Update()
    {
        if ( Application.isEditor )
        {
            [...]
        }
        else
        {
            [...]
        }
    }
}
```

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
```

Атрибут ExecuteInEditMode заставляет Unity выполнять магические методы этого скрипта, даже если игра не воспроизводится.

> Функции не вызываются постоянно, как в режиме воспроизведения
>
> - Обновление вызывается только тогда, когда что-то в сцене изменилось.
> - OnGUI вызывается, когда Game View получает событие.
> - OnRenderObject и другие функции обратного вызова рендеринга вызываются при каждой перерисовке вида сцены или представления игры.

```
[RuntimeInitializeOnLoadMethod]
private static void FooBar()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
private static void Foo()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
private static void Bar()
```

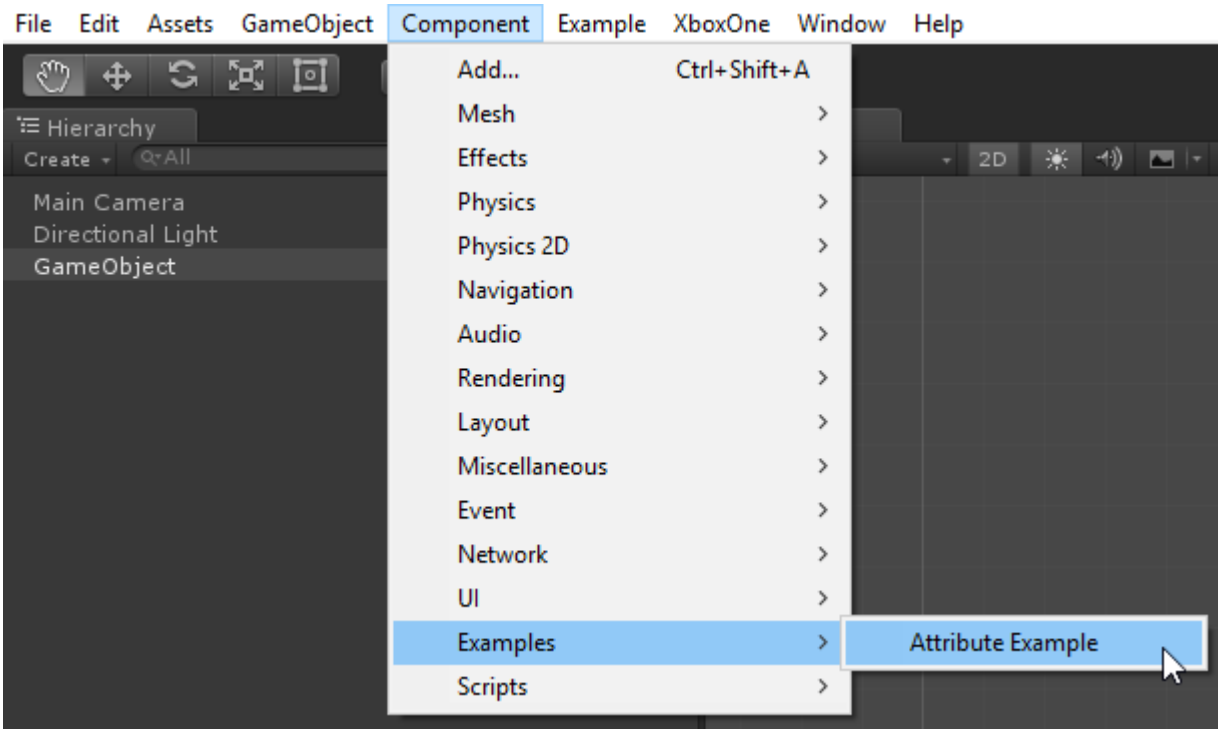
Атрибут RuntimeInitializeOnLoadMethod позволяет вызывать метод класса времени выполнения, когда игра загружает среду выполнения без какого-либо взаимодействия с пользователем.

Вы можете указать, хотите ли вы, чтобы метод вызывался до или после загрузки сцены (по умолчанию - по умолчанию). Порядок выполнения не гарантируется для методов, использующих этот атрибут.

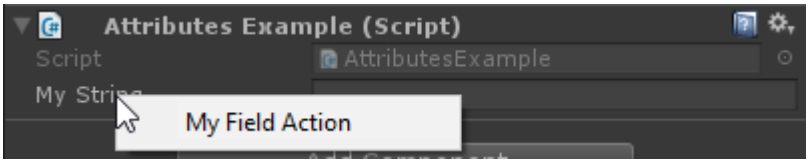
Атрибуты меню

```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
{
    [ContextMenuItem( "My Field Action", "MyFieldContextAction" )]
    public string MyString;

    private void MyFieldContextAction()
    {
        [...]
    }

    [ContextMenu( "My Action" )]
    private void MyContextMenuAction()
    {
        [...]
    }
}
```

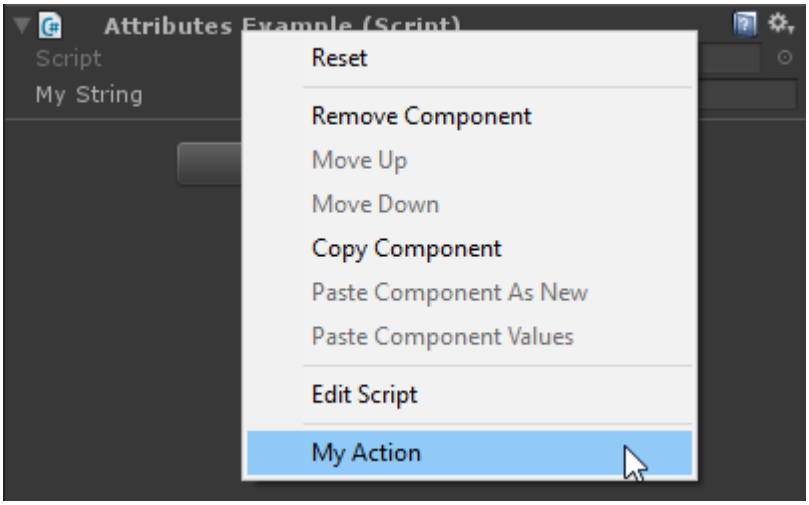
Результат атрибута [AddComponentMenu]



Результат атрибута [ContextMenuItem]



Результат атрибута [ContextMenu]



```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
```

Атрибут AddComponentMenu позволяет разместить компонент в любом месте в меню «Компонент», а не в меню «Компонент-> Сценарии».

```
[ContextMenuItem( "My Field Action", "MyFieldContextAction" )]
```

```
public string MyString;

private void MyFieldContextAction()
{
    [...]
}
```

Атрибут ContextMenuItem позволяет вам определять функции, которые можно добавить в контекстное меню поля. Эти функции будут выполняться после выбора.

```
[ContextMenu( "My Action" )]
private void MyContextMenuAction()
{
    [...]
}
```

Атрибут ContextMenu позволяет вам определять функции, которые можно добавить в контекстное меню компонента.

Атрибуты редактора

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{

    static AttributesExample()
    {
        [...]
    }

    [InitializeOnLoadMethod]
    private static void Foo()
    {
        [...]
    }
}
```

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{

    static AttributesExample()
    {
        [...]
    }
```

Атрибут InitializeOnLoad позволяет пользователю инициализировать класс без какого-либо взаимодействия с пользователем. Это происходит всякий раз, когда редактор запускается или перекомпилируется. Статический конструктор гарантирует, что это будет вызываться перед любыми другими статическими функциями.

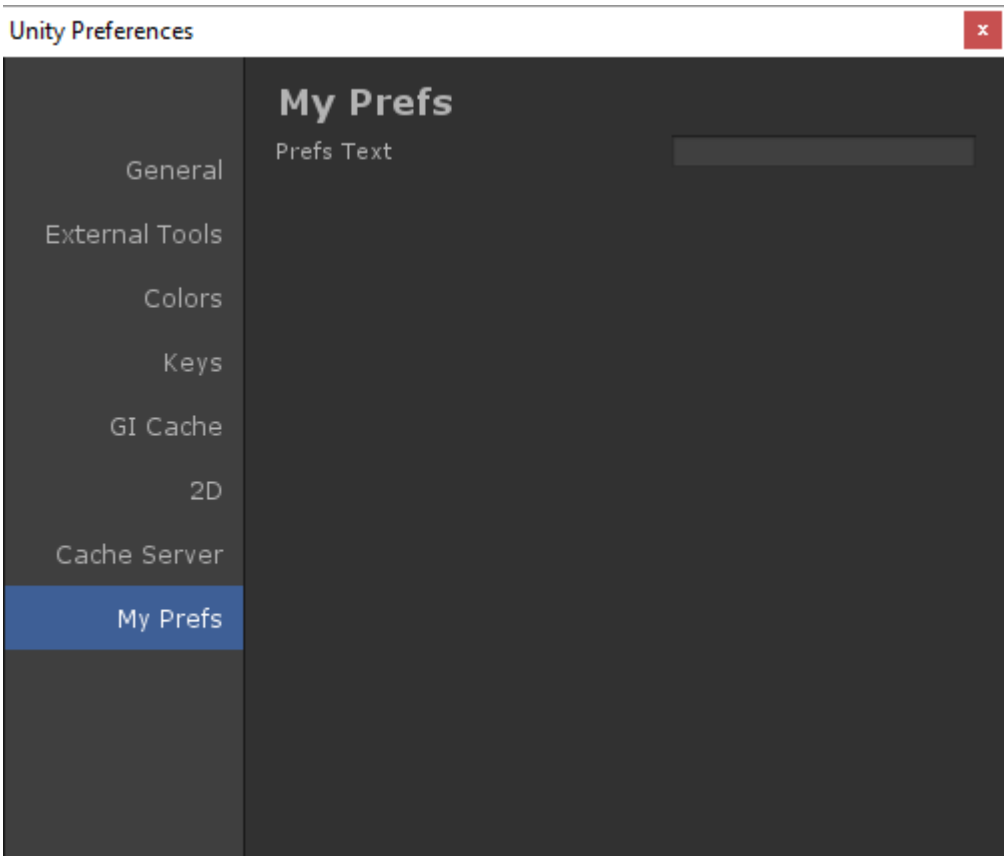
```
[InitializeOnLoadMethod]
```

```
private static void Foo()
{
    [...]
}
```

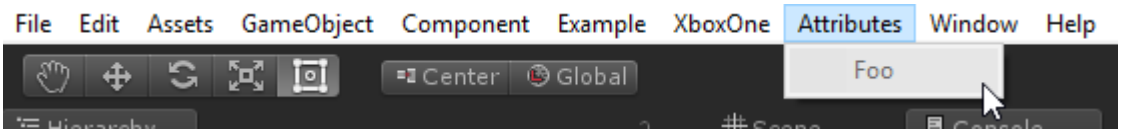
Атрибут InitializeOnLoad позволяет пользователю инициализировать класс без какого-либо взаимодействия с пользователем. Это происходит всякий раз, когда редактор запускается или перекомпилируется. Порядок выполнения не гарантируется для методов, использующих этот атрибут.

```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
{

    public int MyInt;

    private static string prefsText = "";

    [PreferenceItem( "My Prefs" )]
    public static void PreferencesGUI()
    {
        prefsText = EditorGUILayout.TextField( "Prefs Text", prefsText );
    }

    [MenuItem( "Attributes/Foo" )]
    private static void Foo()
    {
        [...]
    }

    [MenuItem( "Attributes/Foo", true )]
    private static bool FooValidate()
    {
        return false;
    }
}
```

Результат атрибута [PreferenceItem]



Результат атрибута [MenuItem]



```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
```

Атрибут CanEditMultipleObjects позволяет редактировать значения из вашего компонента через несколько GameObjects. Без этого компонента вы не увидите, что ваш компонент выглядит как обычно при выборе нескольких GameObjects, но вместо этого вы увидите сообщение «Редактирование нескольких объектов не поддерживается»,

> Этот атрибут предназначен для пользовательских редакторов для поддержки многократного редактирования. Нестандартные редакторы автоматически поддерживают многократное редактирование.

```
[PreferenceItem( "My Prefs" )]
public static void PreferencesGUI()
```

Атрибут PreferenceItem позволяет создать дополнительный элемент в меню настроек Unity. Метод приема должен быть статичным для использования.

```
[MenuItem( "Attributes/Foo" )]
```

```
private static void Foo()
{
    [...]
}

[MenuItem( "Attributes/Foo", true )]
private static bool FooValidate()
{
    return false;
}
```

Атрибут MenuItem позволяет создавать пользовательские элементы меню для выполнения функций. В этом примере также используется функция валидатора (которая всегда возвращает false), чтобы предотвратить выполнение функции.

```
[CustomEditor( typeof( MyComponent ) )]
public class AttributesExample : Editor
{
    [...]
}
```

Атрибут CustomEditor позволяет создавать собственные редакторы для ваших компонентов. Эти редакторы будут использоваться для рисования вашего компонента в инспекторе и должны вытекать из класса Editor.

```
[CustomPropertyDrawer( typeof( MyClass ) )]
public class AttributesExample : PropertyDrawer
{
    [...]
}
```

Атрибут CustomPropertyDrawer позволяет создать пользовательский ящик свойств для инспектора. Вы можете использовать эти ящики для своих пользовательских типов данных, чтобы их можно было увидеть в инспекторе.

```
[DrawGizmo( GizmoType.Selected )]
private static void DoGizmo( AttributesExample obj, GizmoType type )
{
    [...]
}
```

Атрибут DrawGizmo позволяет создавать пользовательские вещицы для ваших компонентов. Эти вещицы будут нарисованы в режиме просмотра сцены. Вы можете решить, когда рисовать gizmo, используя параметр GizmoType в атрибуте DrawGizmo.

> Метод приема требует двух параметров: первый - это компонент для рисования объекта gizmo, а второй - состояние, в котором объект, которому нужен рисованный гизмо, находится.

Прочитайте Атрибуты онлайн: https://riptutorial.com/ru/unity3d/topic/5535/атрибуты

глава 10: Аудио система

Вступление

Это документация о воспроизведении звука в Unity3D.

Examples

Аудиокласс - Воспроизведение аудио

```
using UnityEngine;

public class Audio : MonoBehaviour {
    AudioSource audioSource;
    AudioClip audioClip;

    void Start() {
        audioClip = (AudioClip)Resources.Load("Audio/Soundtrack");
        audioSource.clip = audioClip;
        if (!audioSource.isPlaying) audioSource.Play();
    }
```

Прочитайте Аудио система онлайн: https://riptutorial.com/ru/unity3d/topic/8064/аудио-система

глава 11: Виртуальная реальность (VR)

Examples

Платформы VR

В VR есть две основные платформы: одна - мобильная платформа, такая как **Google Cardboard** , **Samsung GearVR** , другая - платформа для ПК, такая как **HTC Vive, Oculus, PS VR** ...

Unity официально поддерживает **Oculus Rift** , **Google Carboard** , **Steam VR** , **Playstation VR** , **Gear VR** и **Microsoft Hololens** .

У большинства платформ есть своя поддержка и sdk. Обычно вам нужно загрузить sdk как расширение, прежде всего, для единства.

SDKs:

- Картон Google
- Платформа Daydream
- Samsung GearVR (интегрированный с Unity 5.3)
- Oculus Rift
- HTC Vive / Open VR
- Microsoft Hololens

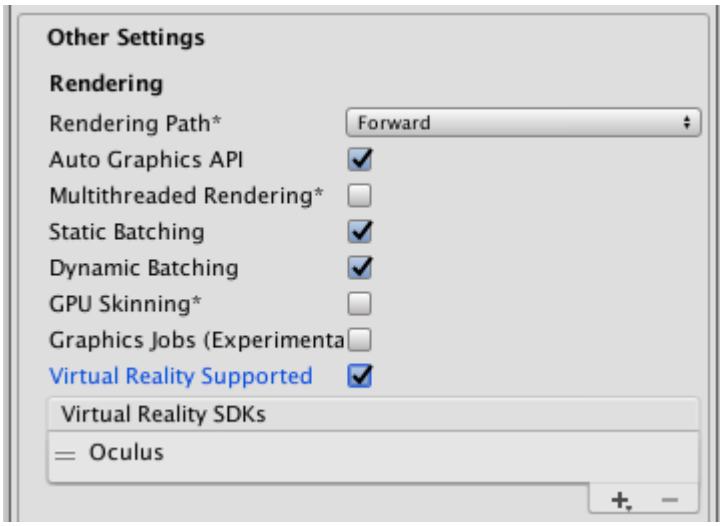
Документация:

- Google Cardboard / Daydream
- Samsung GearVR
- Oculus Rift
- HTC Vive
- Microsoft Hololens

Включение поддержки VR

В редакторе Unity откройте **Настройки проигрывателя** (Edit> Project Settings> Player).

В разделе « **Другие настройки**» проверьте поддержку *виртуальной реальности* .



Добавьте или удалите VR-устройства для каждой цели сборки в списке *SDK Virtual Reality* под этим флажком.

аппаратные средства

Существует необходимая аппаратная зависимость для приложения VR, которое обычно зависит от платформы, для которой вы строите. Существуют 2 широкие категории аппаратных устройств, основанные на их возможностях движения:

1. 3 DOF (степени свободы)
2. 6 DOF (степени свободы)

3 DOF означает, что движение дисплея с головкой (HMD) ограничено для работы в трех измерениях, которые вращаются вокруг трех ортогональных осей с центром в центре тяжести HMD - продольной, вертикальной и горизонтальной осей. Движение вокруг продольной оси называется рулоном, движение вокруг боковой оси называется шагом, а движение вокруг перпендикулярной оси называется рысканием, аналогичные принципы, которые управляют движением любого движущегося объекта, такого как самолет или автомобиль, что означает, что, хотя вы будете способный видеть во всех направлениях X, Y, Z движение вашего HMD в виртуальной среде, но вы не сможете ничего перемещать или трогать (движение с помощью дополнительного контроллера bluetooth не совпадает).

Тем не менее, 6 DOF допускают опыт работы в комнате, в котором вы также можете перемещаться вокруг оси X, Y и Z отдельно от движений рулона, высоты тона и рыскания относительно его центра тяжести, следовательно, 6 степеней свободы.

В настоящее время VR с объемным пространством, способствующим 6 DOF, требует высокой вычислительной производительности с высококачественной графической картой и оперативной памятью, которую вы, вероятно, не получите от стандартных ноутбуков, и потребует настольный компьютер с оптимальной производительностью, а также не менее 6 футов × 6 футов в то время как опыт 3 DOF может быть достигнут только стандартным смартфоном с встроенным гироскопом (который встроен в большинство современных

смартфонов, стоимость которых составляет около 200 долларов США или более).

Некоторые распространенные устройства, доступные на рынке сегодня:

- Oculus Rift (6 DOF)
- HTC Vive (6 DOF)
- Мечта (3 DOF)
- Gear VR Powered by Oculus (3 DOF)
- Картон Google (3 DOF)

Прочитайте Виртуальная реальность (VR) онлайн: https://riptutorial.com/ru/unity3d/topic/5787/виртуальная-реальность--vr-

глава 12: Входная система

Examples

Чтение ключа Пресса и различие между GetKey, GetKeyDown и GetKeyUp

Вход должен быть прочитан из функции «Обновить».

Ссылка для всех доступных перечислений Keycode .

1. Чтение нажатия клавиши с `Input.GetKey` :

`Input.GetKey` будет **повторно** возвращать `true` пока пользователь удерживает указанный ключ. Это можно использовать для **многократного** запуска оружия, удерживая указанный ключ нажатым. Ниже приведен пример автоматического огня пули, когда удерживается клавиша Space. Игрок не должен нажимать и отпускать клавишу снова и снова.

```
public GameObject bulletPrefab;
public float shootForce = 50f;

void Update()
{
    if (Input.GetKey(KeyCode.Space))
    {
        Debug.Log("Shooting a bullet while SpaceBar is held down");

        //Instantiate bullet
        GameObject bullet = Instantiate(bulletPrefab, transform.position, transform.rotation)
as GameObject;

        //Get the Rigidbody from the bullet then add a force to the bullet
        bullet.GetComponent<Rigidbody>().AddForce(bullet.transform.forward * shootForce);
    }
}
```

2. Нажмите клавишу Enter с `Input.GetKeyDown` :

`Input.GetKeyDown` будет `Input.GetKeyDown` только **один раз** при нажатии указанной клавиши. Это ключевое различие между `Input.GetKey` и `Input.GetKeyDown` . Одним из примеров использования его использования является включение пользовательского интерфейса или фонарика или элемента вкл. / Выкл.

```
public Light flashLight;
bool enableFlashLight = false;

void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        //Toggle Light
```

```
        enableFlashLight = !enableFlashLight;
        if (enableFlashLight)
        {
            flashLight.enabled = true;
            Debug.Log("Light Enabled!");
        }
        else
        {
            flashLight.enabled = false;
            Debug.Log("Light Disabled!");
        }
    }
}
```

3. Нажмите клавишу Enter с `Input.GetKeyUp` :

Это полная противоположность `Input.GetKeyDown` . Он используется для обнаружения при отпускании / снятии нажатия клавиши. Как и `Input.GetKeyDown` , он возвращает `true` только **один раз** . Например, вы можете `enable` свет при удерживании клавиши с помощью `Input.GetKeyDown` затем отключите свет, когда клавиша будет выпущена с помощью `Input.GetKeyUp` .

```
public Light flashLight;
void Update()
{
    //Disable Light when Space Key is pressed
    if (Input.GetKeyDown(KeyCode.Space))
    {
        flashLight.enabled = true;
        Debug.Log("Light Enabled!");
    }

    //Disable Light when Space Key is released
    if (Input.GetKeyUp(KeyCode.Space))
    {
        flashLight.enabled = false;
        Debug.Log("Light Disabled!");
    }
}
```

Прочитать датчик акселерометра (базовый)

`Input.acceleration` используется для считывания датчика акселерометра. Он возвращает `Vector3` в результате, который содержит значения оси `x` , `y` и `z` в трехмерном пространстве.

```
void Update()
{
    Vector3 acclerometerValue = rawAccelValue();
    Debug.Log("X: " + acclerometerValue.x + "  Y: " + acclerometerValue.y + "  Z: " +
acclerometerValue.z);
}

Vector3 rawAccelValue()
{
    return Input.acceleration;
```

```
}
```

Прочитать датчик акселерометра (предварительный)

Использование исходных значений непосредственно с датчика акселерометра для перемещения или поворота GameObject может вызвать такие проблемы, как отрывистые движения или вибрации. Перед использованием рекомендуется сгладить значения. Фактически, значения от датчика акселерометра всегда должны быть сглажены перед использованием. Это может быть выполнено с помощью фильтра нижних частот, и именно здесь `Vector3.Lerp` встает на свои места.

```
//The lower this value, the less smooth the value is and faster Accel is updated. 30 seems
fine for this
const float updateSpeed = 30.0f;

float AccelerometerUpdateInterval = 1.0f / updateSpeed;
float LowPassKernelWidthInSeconds = 1.0f;
float LowPassFilterFactor = 0;
Vector3 lowPassValue = Vector3.zero;

void Start()
{
    //Filter Accelerometer
    LowPassFilterFactor = AccelerometerUpdateInterval / LowPassKernelWidthInSeconds;
    lowPassValue = Input.acceleration;
}

void Update()
{

    //Get Raw Accelerometer values (pass in false to get raw Accelerometer values)
    Vector3 rawAccelValue = filterAccelValue(false);
    Debug.Log("RAW X: " + rawAccelValue.x + "  Y: " + rawAccelValue.y + "  Z: " +
rawAccelValue.z);

    //Get smoothed Accelerometer values (pass in true to get Filtered Accelerometer values)
    Vector3 filteredAccelValue = filterAccelValue(true);
    Debug.Log("FILTERED X: " + filteredAccelValue.x + "  Y: " + filteredAccelValue.y + "  Z: "
+ filteredAccelValue.z);
}

//Filter Accelerometer
Vector3 filterAccelValue(bool smooth)
{
    if (smooth)
        lowPassValue = Vector3.Lerp(lowPassValue, Input.acceleration, LowPassFilterFactor);
    else
        lowPassValue = Input.acceleration;

    return lowPassValue;
}
```

Прочитать датчик акселерометра (точность)

Считывайте датчик акселерометра с точностью.

В этом примере выделяется память:

```
void Update()
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + "  Y: " + accelValue.y + "  Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    foreach (AccelerationEvent tempAccelEvent in Input.accelerationEvents)
    {
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}
```

В этом примере **не** выделяется память:

```
void Update()
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + "  Y: " + accelValue.y + "  Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    for (int i = 0; i < Input.accelerationEventCount; ++i)
    {
        AccelerationEvent tempAccelEvent = Input.GetAccelerationEvent(i);
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}
```

Обратите внимание, что это не фильтруется. Пожалуйста , смотрите здесь для того, как сгладить значение акселерометра для удаления шума.

Прочитать кнопку мыши (левый, средний, правый)

Эти функции используются для проверки щелчков мыши.

- `Input.GetMouseButton(int button);`
- `Input.GetMouseButtonDown(int button);`
- `Input.GetMouseButtonUp(int button);`

Все они принимают один и тот же параметр.

- 0 = щелчок левой кнопкой мыши.

- 1 = Щелкните правой кнопкой мыши.
- 2 = Средний щелчок мыши.

`GetMouseButton` используется для обнаружения, когда кнопка мыши *постоянно удерживается* . Он возвращает `true` когда указанная кнопка мыши удерживается нажатой.

```
void Update()
{
    if (Input.GetMouseButton(0))
    {
        Debug.Log("Left Mouse Button Down");
    }

    if (Input.GetMouseButton(1))
    {
        Debug.Log("Right Mouse Button Down");
    }

    if (Input.GetMouseButton(2))
    {
        Debug.Log("Middle Mouse Button Down");
    }
}
```

`GetMouseButtonDown` используется для обнаружения при щелчке мышью. Он возвращает `true` если он нажат **один раз** . Он не вернется `true` , пока кнопка мыши не будет отпущена и нажата снова.

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Debug.Log("Left Mouse Button Clicked");
    }

    if (Input.GetMouseButtonDown(1))
    {
        Debug.Log("Right Mouse Button Clicked");
    }

    if (Input.GetMouseButtonDown(2))
    {
        Debug.Log("Middle Mouse Button Clicked");
    }
}
```

`GetMouseButtonUp` используется для обнаружения, когда выделенная кнопка мыши отпущена. Это будет возвращать только `true` , как только указанная кнопка мыши отпущена. Чтобы снова вернуть true, его нужно снова нажать и отпустить.

```
void Update()
{
    if (Input.GetMouseButtonUp(0))
    {
        Debug.Log("Left Mouse Button Released");
```

```
    }

    if (Input.GetMouseButtonUp(1))
    {
        Debug.Log("Right Mouse Button Released");
    }

    if (Input.GetMouseButtonUp(2))
    {
        Debug.Log("Middle Mouse Button Released");
    }
}
```

Прочитайте Входная система онлайн: https://riptutorial.com/ru/unity3d/topic/3413/входная-система

глава 13: Выполнение класса MonoBehaviour

Examples

Нет переопределенных методов

Причина, по которой вам не нужно переопределять `Awake` , `Start` , `Update` и другой метод, заключается в том, что они не являются виртуальными методами, определенными в базовом классе.

При первом обращении к сценарию сценарий выполнения скриптов просматривает скрипт, чтобы определить, определены ли некоторые методы. Если они есть, эта информация кэшируется и методы добавляются в их соответствующий список. Эти списки затем просто зацикливаются в разное время.

Причина, по которой эти методы не являются виртуальными, связана с производительностью. Если бы все сценарии имели бы `Awake` , `Start` , `OnEnable` , `OnDisable` , `Update` , `LateUpdate` и `FixedUpdate` , тогда все они были бы добавлены в их списки, что означало бы, что все эти методы будут выполнены. Обычно это не будет большой проблемой, однако все эти вызовы методов относятся к исходной стороне (C ++) к управляемой стороне (C #), которая поставляется с производительностью.

Теперь представьте себе, что все эти методы находятся в их списках, а некоторые / большинство из них могут даже не иметь фактического тела метода. Это означало бы, что огромное количество производительности тратится на вызов методов, которые даже не делают ничего. Чтобы предотвратить это, Unity отказалась от использования виртуальных методов и сделала систему обмена сообщениями, которая гарантирует, что эти методы будут вызваны только тогда, когда они будут определены на самом деле, сохраняя ненужные вызовы методов.

Вы можете прочитать больше об этом в блоге Unity здесь: 10000 Update () Вызовы и многое другое на IL2CPP здесь: Введение в IL2CPP Internals

Прочитайте Выполнение класса MonoBehaviour онлайн:
https://riptutorial.com/ru/unity3d/topic/2304/выполнение-класса-monobehaviour

глава 14: Графическая система пользовательского интерфейса немедленного режима (IMGUI)

Синтаксис

- public static void GUILayout.Label (строковый текст, параметры GUILayoutOption [])
- public static bool GUILayout.Button (строковый текст, параметры GUILayoutOption [])
- public static string GUILayout.TextArea (текст строки, параметры GUILayoutOption [])

Examples

GUILayout

Старый инструмент пользовательского интерфейса, теперь используемый для быстрого и простого прототипирования или отладки в игре.

```
void OnGUI ()
{
    GUILayout.Label ("I'm a simple label text displayed in game.");

    if ( GUILayout.Button("CLICK ME") )
    {
        GUILayout.TextArea ("This is a \n
                             multiline comment.")
    }
}
```

Функция **GUILayout** работает внутри функции **OnGUI** .

Прочитайте Графическая система пользовательского интерфейса немедленного режима (IMGUI) онлайн: https://riptutorial.com/ru/unity3d/topic/6947/графическая-система-пользовательского-интерфейса-немедленного-режима--imgui-

глава 15: Импортеры и (пост) процессоры

Синтаксис

- AssetPostprocessor.OnPreprocessTexture ()

замечания

Используйте `String.Contains()` для обработки только активов, которые имеют заданную строку в своих путях активов.

```
if (assetPath.Contains("ProcessThisFolder"))
{
    // Process asset
}
```

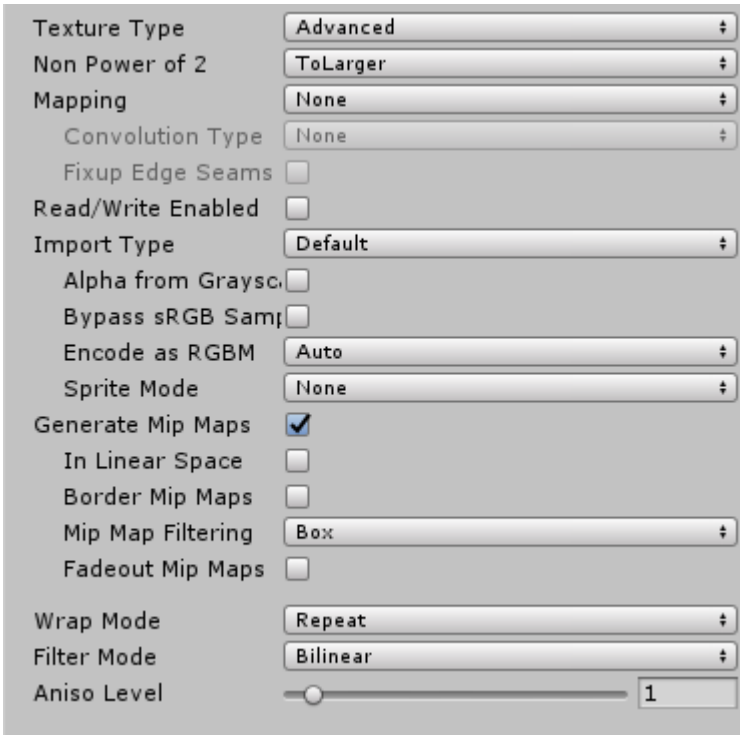
Examples

Постпроцессор текстуры

Создайте файл `TexturePostProcessor.cs` любом месте папки « **Активы** »:

```
using UnityEngine;
using UnityEditor;

public class TexturePostProcessor : AssetPostprocessor
{
    void OnPostprocessTexture(Texture2D texture)
    {
        TextureImporter importer = assetImporter as TextureImporter;
        importer.anisoLevel = 1;
        importer.filterMode = FilterMode.Bilinear;
        importer.mipmapEnabled = true;
        importer.npotScale = TextureImporterNPOTScale.ToLarger;
        importer.textureType = TextureImporterType.Advanced;
    }
}
```

Теперь, каждый раз, когда Unity импортирует текстуру, он будет иметь следующие параметры:



Если вы используете постпроцессор, вы не можете изменять параметры текстуры, управляя **настройками импорта** в редакторе.
Когда вы нажмете кнопку « **Применить»** , текстура будет переименована, а постпроцессорный код снова запустится.

Основной импортер

Предположим, у вас есть собственный файл, для которого вы хотите создать импортер. Это может быть файл .xls или что-то еще. В этом случае мы собираемся использовать файл JSON, потому что это легко, но мы собираемся выбрать пользовательское расширение, чтобы было легко определить, какие файлы являются нашими?

Предположим, что формат JSON-файла

```
{
  "someValue": 123,
  "someOtherValue": 456.297,
  "someBoolValue": true,
  "someStringValue": "this is a string",
}
```

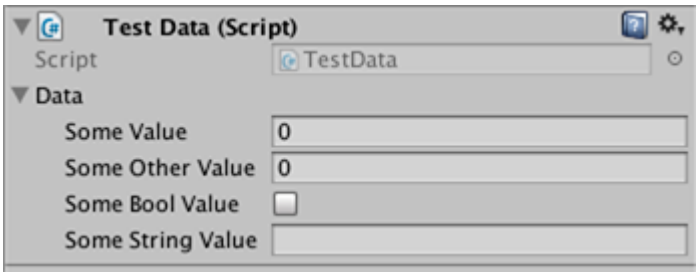
Давайте сохраним это как `Example.test` где-то *за пределами* активов.

Затем создайте `MonoBehaviour` с пользовательским классом только для данных. Пользовательский класс предназначен исключительно для облегчения десериализации JSON. Вы не должны использовать пользовательский класс, но это делает этот пример короче. Мы сохраним это в `TestData.cs`

```
using UnityEngine;
```

```
using System.Collections;

public class TestData : MonoBehaviour {

    [System.Serializable]
    public class Data {
        public int someValue = 0;
        public float someOtherValue = 0.0f;
        public bool someBoolValue = false;
        public string someStringValue = "";
    }

    public Data data = new Data();
}
```

Если вы должны вручную добавить этот скрипт в GameObject, вы увидите что-то вроде



Затем создайте папку `Editor` где-нибудь в разделе « `Assets` . Я могу быть на любом уровне. Внутри папки Editor `TestDataAssetPostprocessor.cs` файл `TestDataAssetPostprocessor.cs` и поместите его в него.

```
using UnityEditor;
using UnityEngine;
using System.Collections;

public class TestDataAssetPostprocessor : AssetPostprocessor
{
    const string s_extension = ".test";

    // NOTE: Paths start with "Assets/"
    static bool IsFileWeCareAbout(string path)
    {
        return System.IO.Path.GetExtension(path).Equals(
           s_extension,
           System.StringComparison.Ordinal);
    }

    static void HandleAddedOrChangedFile(string path)
    {
        string text = System.IO.File.ReadAllText(path);
        // should we check for error if the file can't be parsed?
        TestData.Data newData = JsonUtility.FromJson<TestData.Data>(text);

        string prefabPath = path + ".prefab";
        // Get the existing prefab
        GameObject existingPrefab =
            AssetDatabase.LoadAssetAtPath(prefabPath, typeof(Object)) as GameObject;
        if (!existingPrefab)
        {
```

```
            // If no prefab exists make one
            GameObject newGameObject = new GameObject();
            newGameObject.AddComponent<TestData>();
            PrefabUtility.CreatePrefab(prefabPath,
                                       newGameObject,
                                       ReplacePrefabOptions.Default);
            GameObject.DestroyImmediate(newGameObject);
            existingPrefab =
                AssetDatabase.LoadAssetAtPath(prefabPath, typeof(Object)) as GameObject;
        }

        TestData testData = existingPrefab.GetComponent<TestData>();
        if (testData != null)
        {
            testData.data = newData;
            EditorUtility.SetDirty(existingPrefab);
        }
    }

    static void HandleRemovedFile(string path)
    {
        // Decide what you want to do here. If the source file is removed
        // do you want to delete the prefab? Maybe ask if you'd like to
        // remove the prefab?
        // NOTE: Because you might get many calls (like you deleted a
        // subfolder full of .test files you might want to get all the
        // filenames and ask all at once ("delete all these prefabs?").
    }

    static void OnPostprocessAllAssets (string[] importedAssets, string[] deletedAssets,
string[] movedAssets, string[] movedFromAssetPaths)
    {
        foreach (var path in importedAssets)
        {
            if (IsFileWeCareAbout(path))
            {
                HandleAddedOrChangedFile(path);
            }
        }

        foreach (var path in deletedAssets)
        {
            if (IsFileWeCareAbout(path))
            {
                HandleRemovedFile(path);
            }
        }

        for (var ii = 0; ii < movedAssets.Length; ++ii)
        {
            string srcStr = movedFromAssetPaths[ii];
            string dstStr = movedAssets[ii];

            // the source was moved, let's move the corresponding prefab
            // NOTE: We don't handle the case if there already being
            // a prefab of the same name at the destination
            string srcPrefabPath = srcStr + ".prefab";
            string dstPrefabPath = dstStr + ".prefab";

            AssetDatabase.MoveAsset(srcPrefabPath, dstPrefabPath);
        }
```

```
    }
}
```

С сохранением вы сможете перетащить файл `Example.test` который мы создали выше, в вашу папку Unity Assets, и вы должны увидеть соответствующий сборник. Если вы отредактируете `Example.test` вы увидите, что данные в prefab будут немедленно обновлены. Если вы перетащите сборку в иерархию сцен, вы увидите ее обновление, а также изменения `Example.test` . Если вы переместите `Example.test` в другую папку, соответствующий сборник будет перемещаться вместе с ним. Если вы измените поле в экземпляре, то измените файл `Example.test` вы увидите только обновленные поля, которые вы не изменяли.

Усовершенствования. В приведенном выше примере, после того, как вы перетащили `Example.test` в свою папку « `Assets` », вы увидите, что есть пример `Example.test` и `Example.test.prefab` . Было бы здорово , чтобы знать , чтобы сделать его работу более как модельные импортеры работают мы вы бы волшебно видеть только `Example.test` и что это `AssetBundle` или некоторые такие вещи. Если вы знаете, как, пожалуйста, укажите этот пример

Прочитайте Импортеры и (пост) процессоры онлайн:
https://riptutorial.com/ru/unity3d/topic/5279/импортеры-и--пост--процессоры

глава 16: Интеграция рекламы

Вступление

Эта тема посвящена интеграции сторонних рекламных сервисов, таких как Unity Ads или Google AdMob, в проект Unity.

замечания

Это относится к Unity Ads .

> **Убедитесь, что режим тестирования для объявлений Unity включен во время разработки**

Вы, как разработчик, не можете создавать показы или установки, нажимая на объявления в своей собственной игре. Это нарушает соглашение об Условиях использования Unity , и вам будет запрещено использовать сеть Unity Ads для попыток мошенничества.

Для получения дополнительной информации ознакомьтесь с Соглашением об Условиях использования Unity .

Examples

Основы рекламы Unity в C

```
using UnityEngine;
using UnityEngine.Advertisements;

public class Example : MonoBehaviour
{
    #if !UNITY_ADS // If the Ads service is not enabled
    public string gameId; // Set this value from the inspector
    public bool enableTestMode = true; // Enable this during development
    #endif

    void InitializeAds () // Example of how to initialize the Unity Ads service
    {
        #if !UNITY_ADS // If the Ads service is not enabled
        if (Advertisement.isSupported) { // If runtime platform is supported
            Advertisement.Initialize(gameId, enableTestMode); // Initialize
        }
        #endif
    }

    void ShowAd () // Example of how to show an ad
    {
        if (Advertisement.isInitialized || Advertisement.IsReady()) { // If the ads are ready
```

```
to be shown
            Advertisement.Show(); // Show the default ad placement
        }
    }
}
```

Основы рекламы Unity в JavaScript

```
#pragma strict
import UnityEngine.Advertisements;

#if !UNITY_ADS // If the Ads service is not enabled
public var gameId : String; // Set this value from the inspector
public var enableTestMode : boolean = true; // Enable this during development
#endif

function InitializeAds () // Example of how to initialize the Unity Ads service
{
    #if !UNITY_ADS // If the Ads service is not enabled
    if (Advertisement.isSupported) { // If runtime platform is supported
        Advertisement.Initialize(gameId, enableTestMode); // Initialize
    }
    #endif
}

function ShowAd () // Example of how to show an ad
{
    if (Advertisement.isInitialized && Advertisement.IsReady()) { // If the ads are ready to
be shown
        Advertisement.Show(); // Show the default ad placement
    }
}
```

Прочитайте Интеграция рекламы онлайн: https://riptutorial.com/ru/unity3d/topic/9796/интеграция-рекламы

глава 17: Использование контроля источника Git с Unity

Examples

Использование Git для хранения больших файлов (LFS) с Unity

предисловие

Git может работать с разработкой видеоигр. Однако основное предостережение заключается в том, что файлы с большим объемом файлов (более 5 МБ) в версии с большим количеством файлов могут быть проблемой в долгосрочной перспективе, так как ваши истории взлома истории - Git просто не был создан для двоичных файлов версий.

Отличная новость заключается в том, что с середины 2015 года GitHub выпустил плагин для Git Git LFS, который напрямую занимается этой проблемой. Теперь вы можете легко и эффективно создавать большие двоичные файлы!

Наконец, эта документация ориентирована на конкретные требования и информацию, необходимые для обеспечения хорошей жизни Git с развитием видеоигр. В этом руководстве не будет рассказано, как использовать Git.

Установка Git & Git-LFS

У вас есть ряд опций, доступных вам как разработчик, и первым выбором является установка базовой командной строки Git или включение в нее одного из популярных приложений Git GUI.

Вариант 1: использовать приложение Git GUI

Это действительно личное предпочтение здесь, поскольку существует множество вариантов с точки зрения графического интерфейса Git или вообще использовать графический интерфейс. У вас есть несколько приложений на выбор, вот 3 из наиболее популярных:

- Sourcetree (бесплатно)
- Github Desktop (бесплатно)
- SmartGit (Commerical)

После того, как вы установили свое приложение по своему выбору, пожалуйста, выполните Google и следуйте инструкциям о том, как обеспечить его настройку для Git-LFS. Мы пропустим этот шаг в этом руководстве, так как это приложение специфично.

Вариант 2: установите Git & Git-LFS

Это довольно просто - установите Git . Затем. Установите Git LFS .

Настройка большого хранилища файлов Git в вашем проекте

Если вы используете плагин Git LFS для лучшей поддержки двоичных файлов, вам необходимо установить некоторые типы файлов, которые будут управляться Git LFS. Добавьте ниже в свой файл `.gitattributes` в корень вашего репозитория для поддержки общих двоичных файлов, используемых в проектах Unity:

```
# Image formats:
*.tga filter=lfs diff=lfs merge=lfs -text
*.png filter=lfs diff=lfs merge=lfs -text
*.tif filter=lfs diff=lfs merge=lfs -text
*.jpg filter=lfs diff=lfs merge=lfs -text
*.gif filter=lfs diff=lfs merge=lfs -text
*.psd filter=lfs diff=lfs merge=lfs -text

# Audio formats:
*.mp3 filter=lfs diff=lfs merge=lfs -text
*.wav filter=lfs diff=lfs merge=lfs -text
*.aiff filter=lfs diff=lfs merge=lfs -text

# 3D model formats:
*.fbx filter=lfs diff=lfs merge=lfs -text
*.obj filter=lfs diff=lfs merge=lfs -text

# Unity formats:
*.sbsar filter=lfs diff=lfs merge=lfs -text
*.unity filter=lfs diff=lfs merge=lfs -text

# Other binary formats
*.dll filter=lfs diff=lfs merge=lfs -text
```

Настройка хранилища Git для Unity

При инициализации репозитория Git для развития Unity необходимо выполнить несколько вещей.

Unity Ignore Folders

Не все должно быть версией в репозитории. Вы можете добавить шаблон ниже в свой `.gitignore` файл в корень вашего репозитория. Или, альтернативно, вы можете проверить Unity .gitignore с открытым исходным кодом на GitHub и, альтернативно, генерировать один, используя gitignore.io для единства.

```
# Unity Generated
[Tt]emp/
[Ll]ibrary/
[Oo]bj/

# Unity3D Generated File On Crash Reports
sysinfo.txt

# Visual Studio / MonoDevelop Generated
ExportedObj/
obj/
*.csproj
*.unityproj
*.sln
*.suo
*.tmp
*.user
*.userprefs
*.pidb
*.booproj
*.svd

# OS Generated
desktop.ini
.DS_Store
.DS_Store?
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
```

Подробнее о том, как настроить файл .gitignore, читайте здесь .

Настройки проекта Unity

По умолчанию проекты Unity не настроены на правильное управление версиями.

1. (Пропустите этот шаг в версии 4.5 и выше) Включите параметр `External` в `Unity → Preferences → Packages → Repository` .
2. Переключитесь на `Visible Meta Files` в `Edit → Project Settings → Editor → Version Control Mode` .
3. Переключитесь в « `Force Text` в « `Edit → Project Settings → Editor → Asset Serialization Mode` .
4. Сохраните сцену и проект из меню « `File` .

Дополнительная конфигурация

Одним из немногих серьезных неприятностей, с которыми можно столкнуться при использовании проектов Git с Unity, является то, что Git не заботится о каталогах и с радостью покидает пустые каталоги после удаления из них файлов. Unity сделает файлы `*.meta` для этих каталогов и может вызвать битву между членами команды, когда Git обязуется продолжать добавлять и удалять эти метафайлы.

Добавьте этот крюк Git post-merge в папку `/.git/hooks/` для репозиториев с проектами Unity в них. После любого Git pull / merge, он будет смотреть, какие файлы были удалены, проверьте, пуст ли каталог, в котором он существовал, и если он его удаляет.

Сцены и сборки сборных

Общей проблемой при работе с Unity является то, что 2 или более разработчиков модифицируют сцену Unity или prefab (файлы * .unity). Git не знает, как правильно слить их из коробки. К счастью, команда Unity развернула инструмент SmartMerge, который упрощает автоматическое слияние. Первое, что нужно сделать, это добавить следующие строки в ваш файл `.git` или `.gitconfig` : (Windows: `%USERPROFILE%\.gitconfig` , Linux / Mac OS X: `~/.gitconfig`)

```
[merge]
tool = unityyamlmerge

[mergetool "unityyamlmerge"]
trustExitCode = false
cmd = '<path to UnityYAMLMerge>' merge -p "$BASE" "$REMOTE" "$LOCAL" "$MERGED"
```

В **Windows** путь к UnityYAMLMerge:

```
C:\Program Files\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

или же

```
C:\Program Files (x86)\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

и на **MacOSX** :

```
/Applications/Unity/Unity.app/Contents/Tools/UnityYAMLMerge
```

Как только это будет сделано, mergetool будет доступен, если возникают конфликты во время слияния / переадресации. Не забудьте запустить `git mergetool` вручную, чтобы запустить UnityYAMLMerge.

Прочитайте Использование контроля источника Git с Unity онлайн:

https://riptutorial.com/ru/unity3d/topic/2195/использование-контроля-источника-git-c-unity

глава 18: Как использовать пакеты активов

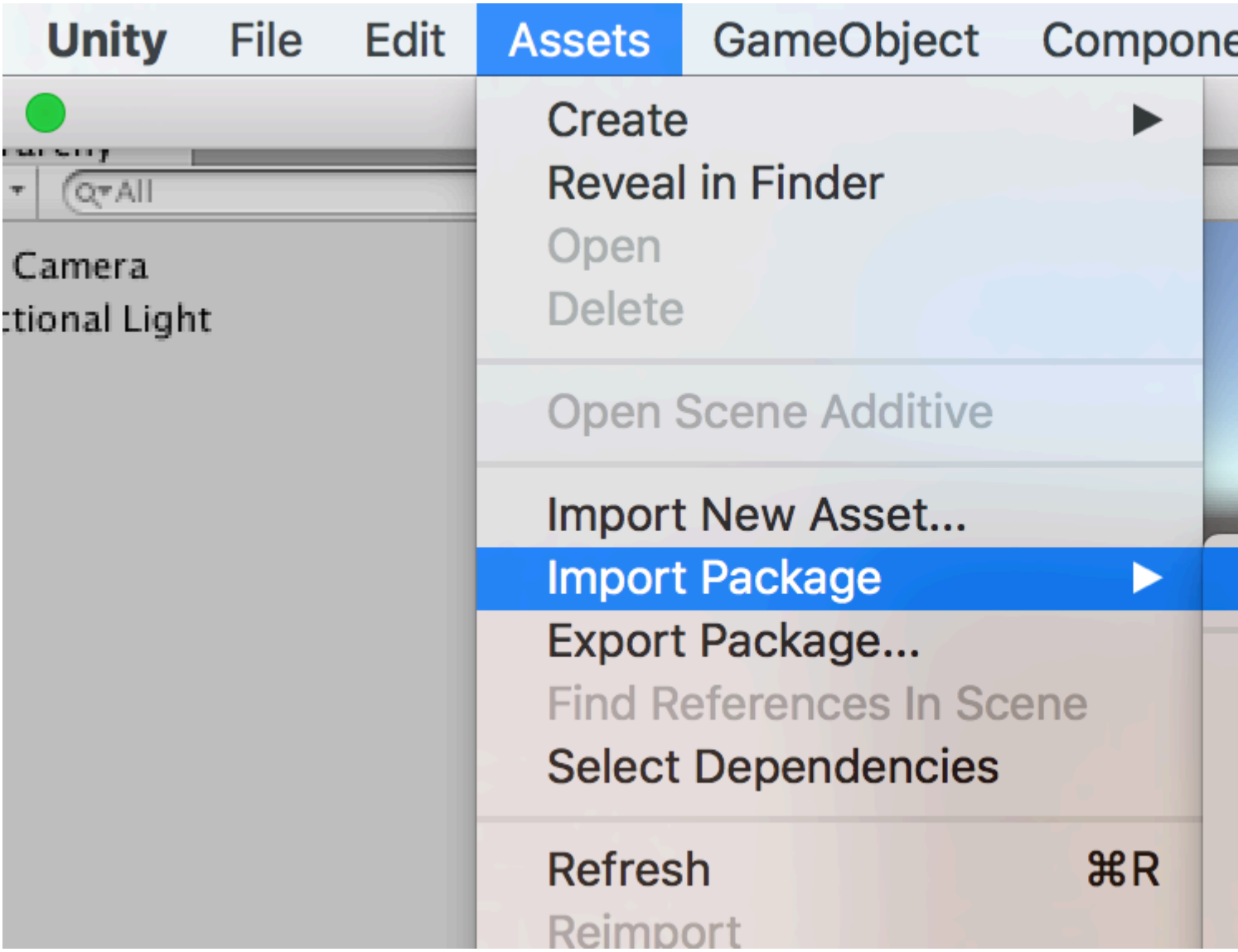
Examples

Пакеты активов

Пакеты активов (с файловым форматом `.unitypackage`) являются обычно используемым способом распространения проектов Unity другим пользователям. При работе с периферийными устройствами, имеющими собственные SDK (например, Oculus), вас могут попросить загрузить и импортировать один из этих пакетов.

Импорт пакета .unitypackage

Чтобы импортировать пакет, перейдите в панель меню Unity и выберите « `Assets > Import Package > Custom Package...` , затем перейдите к файлу `.unitypackage` в `.unitypackage` браузере файлов.



Прочитайте Как использовать пакеты активов онлайн:
https://riptutorial.com/ru/unity3d/topic/4491/как-использовать-пакеты-активов

глава 19: Кватернионы

Синтаксис

- Quaternion.LookRotation (Vector3 forward [, Vector3 вверх]);
- Quaternion.AngleAxis (поплавковые углы, ось VectorOfRotation);
- float angleBetween = Quaternion.Angle (Quaternion rotation1, Quaternion rotation2);

Examples

Вступление к Quaternion vs Euler

Угол Эйлера - это «углы градуса», такие как 90, 180, 45, 30 градусов. Кватернионы отличаются от углов Эйлера тем, что они представляют точку на единичной сфере (радиус равен 1 единице). Вы можете представить эту сферу как трехмерную версию окружности Единицы, которую вы изучаете в тригонометрии. Кватернионы отличаются от углов Эйлера тем, что они используют мнимые числа для определения 3D-вращения.

Хотя это может показаться сложным (и, возможно, это так), Unity имеет большие встроенные функции, которые позволяют переключаться между углами Эйлера и кватерами, а также функции для изменения кватернионов, не зная ни одной вещи о математике позади них.

Преобразование между Эйлером и Кватернионом

```
// Create a quaternion that represents 30 degrees about X, 10 degrees about Y
Quaternion rotation = Quaternion.Euler(30, 10, 0);

// Using a Vector
Vector3 EulerRotation = new Vector3(30, 10, 0);
Quaternion rotation = Quaternion.Euler(EulerRotation);

// Convert a transfroms Quaternion angles to Euler angles
Quaternion quaternionAngles = transform.rotation;
Vector3 eulerAngles = quaternionAngles.eulerAngles;
```

Зачем использовать кватернион?

Кватернионы решают проблему, известную как фиксация карданного вала. Это происходит, когда первичная ось вращения становится коллинеарной с третичной осью вращения. Вот наглядный пример @ 2:09

Ротация вращения кватерниона

`Quaternion.LookRotation(Vector3 forward [, Vector3 up])` создаст поворот кватерниона, который

смотрит вперед «вниз» вперед и имеет ось Y, выровненную с вектором «вверх». Если вектор вверх не указан, будет использоваться Vector3.up.

Поверните этот игровой объект, чтобы посмотреть на целевой объект игры

```
// Find a game object in the scene named Target
public Transform target = GameObject.Find("Target").GetComponent<Transform>();

// We subtract our position from the target position to create a
// Vector that points from our position to the target position
// If we reverse the order, our rotation would be 180 degrees off.
Vector3 lookVector = target.position - transform.position;
Quaternion rotation = Quaternion.LookRotation(lookVector);
transform.rotation = rotation;
```

Прочитайте Кватернионы онлайн: https://riptutorial.com/ru/unity3d/topic/1782/кватернионы

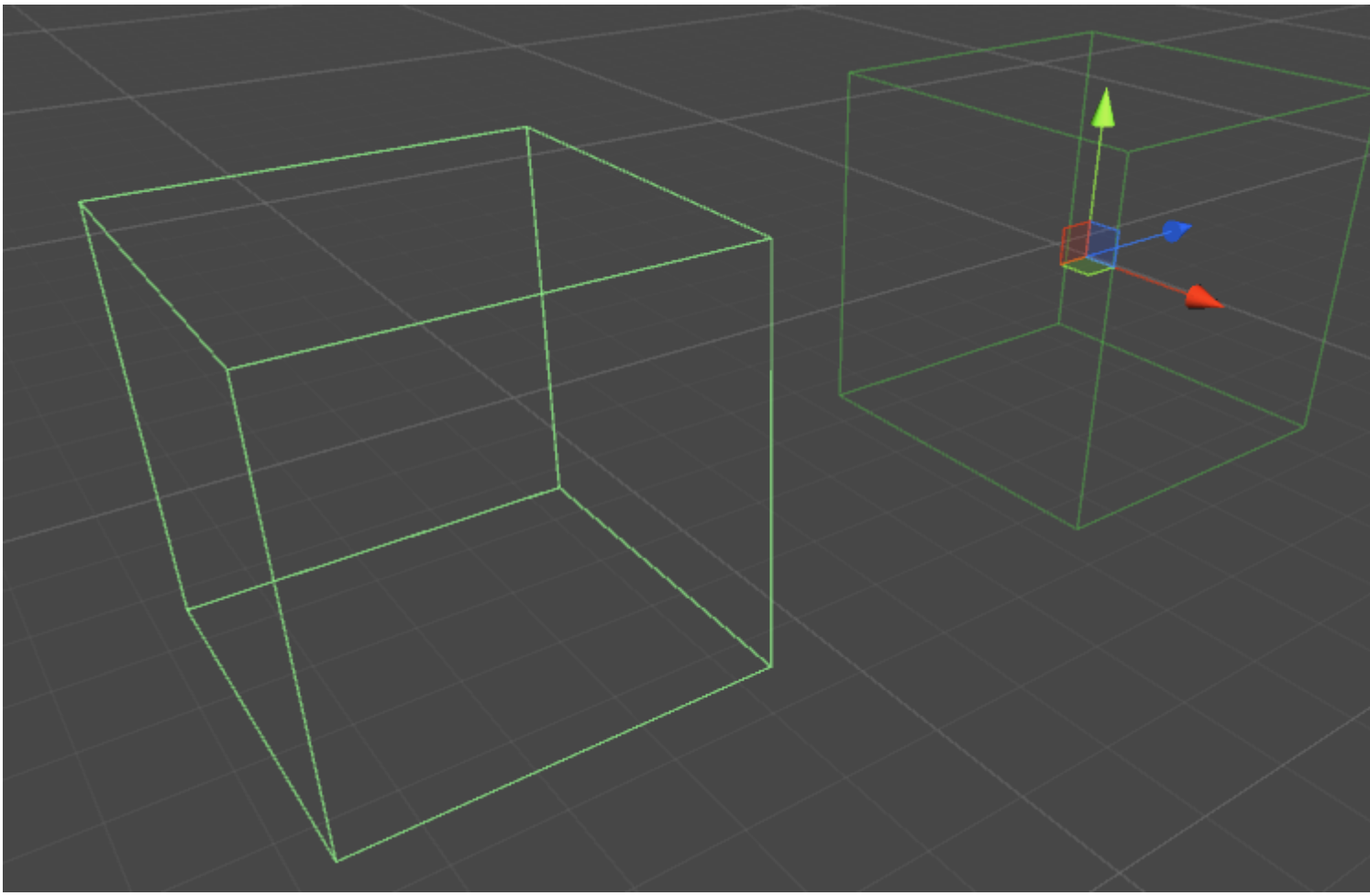
глава 20: коллизия

Examples

Коллайдеры

Коробчик с коробкой

Первобытный коллайдер в форме кубика.



свойства

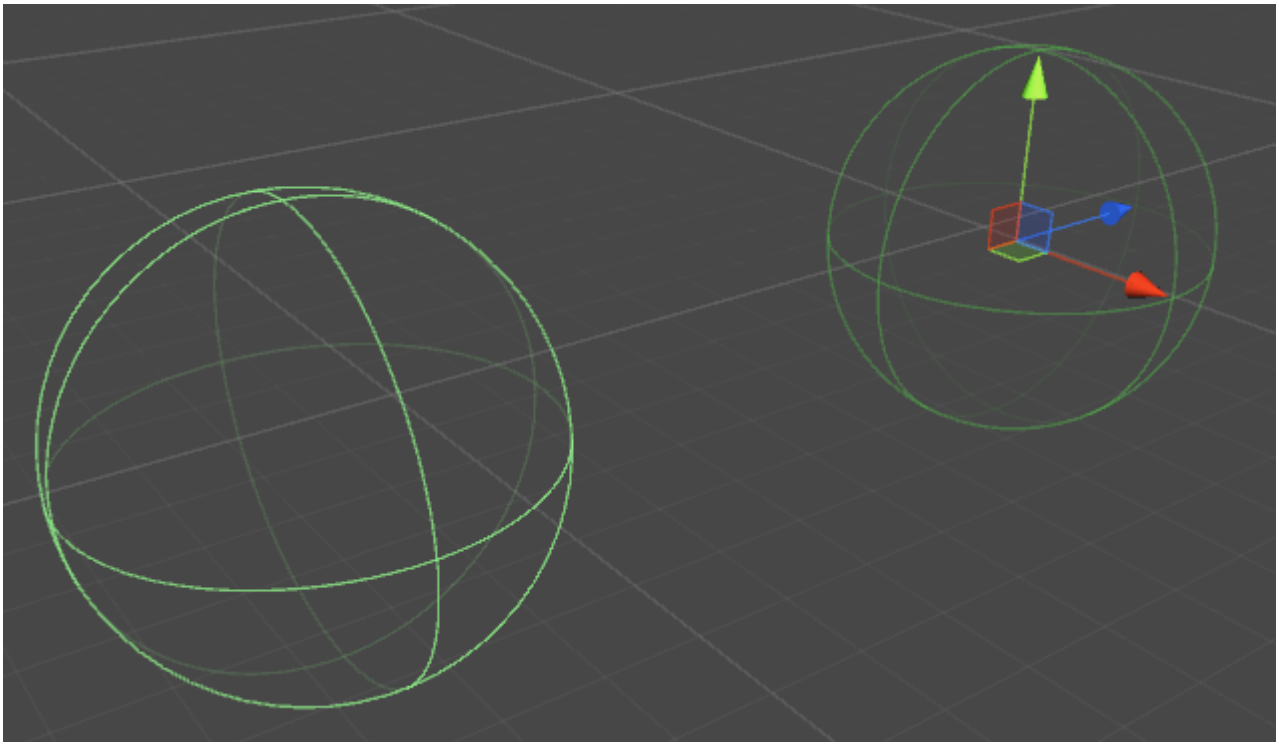
- **Является триггером** - если отмечено, Коллайдер Box игнорирует физику и становится триггерным коллайдером
- **Материал** - ссылка, если она указана, физическому материалу Box Collider
- **Center** - Центральное положение Box Collider в местном пространстве
- **Размер** - размер коллайдера Box, измеренный в локальном пространстве

пример

```
// Add a Box Collider to the current GameObject.
BoxCollider myBC = BoxCollider)myGameObject.gameObject.AddComponent(typeof(BoxCollider));

// Make the Box Collider into a Trigger Collider.
myBC.isTrigger= true;

// Set the center of the Box Collider to the center of the GameObject.
myBC.center = Vector3.zero;

// Make the Box Collider twice as large.
myBC.size = 2;
```

Сферный коллайдер

Первобытный коллайдер в форме шара.



свойства

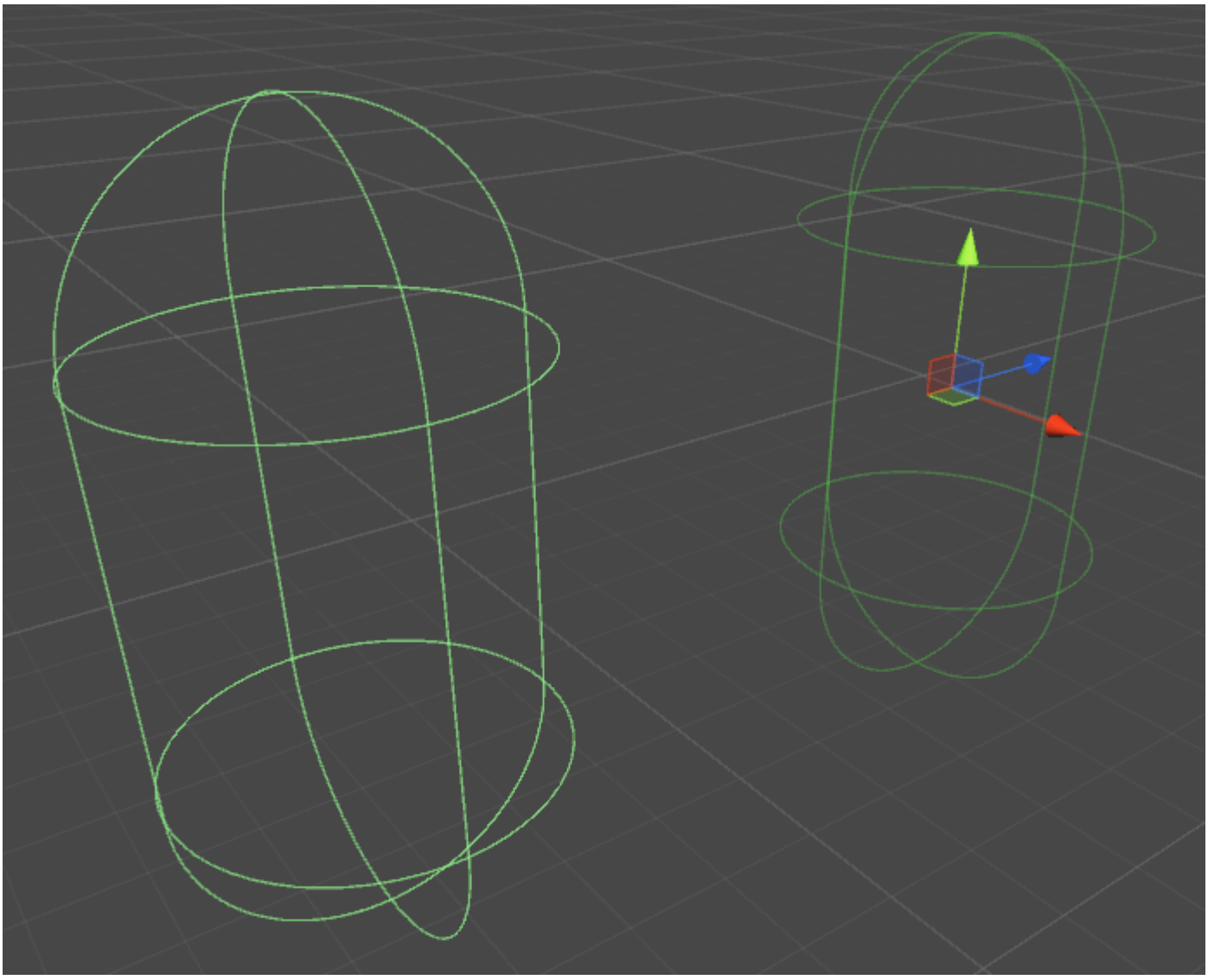
- **Является триггером** - если галочкой, Sphere Collider будет игнорировать физику и стать триггерным коллайдером
- **Материал** - ссылка, если указано, физическому материалу Сферного коллайдера
- **Center** - Центральное положение Sphere Collider в локальном пространстве
- **Радиус** - радиус коллайдера

пример

```
// Add a Sphere Collider to the current GameObject.
SphereCollider mySC =
SphereCollider)myGameObject.gameObject.AddComponent(typeof(SphereCollider));

// Make the Sphere Collider into a Trigger Collider.
mySC.isTrigger= true;

// Set the center of the Sphere Collider to the center of the GameObject.
mySC.center = Vector3.zero;

// Make the Sphere Collider twice as large.
mySC.radius = 2;
```

Капсульный коллайдер

Две полусферы, соединенные цилиндром.



свойства

- **Является триггером** - если галочкой, Capsule Collider будет игнорировать физику и стать триггерным коллайдером
- **Материал** - ссылка, если указано, физическому материалу Коллайдера капсулы
- **Центр** - центральное положение капсульного коллайдера в местном пространстве
- **Радиус** - радиус в локальном пространстве
- **Высота** - общая высота коллайдера
- **Направление** - ось ориентации в локальном пространстве

пример

```
// Add a Capsule Collider to the current GameObject.
CapsuleCollider myCC =
CapsuleCollider)myGameObject.gameObject.AddComponent(typeof(CapsuleCollider));

// Make the Capsule Collider into a Trigger Collider.
myCC.isTrigger= true;

// Set the center of the Capsule Collider to the center of the GameObject.
myCC.center = Vector3.zero;

// Make the Sphere Collider twice as tall.
myCC.height= 2;

// Make the Sphere Collider twice as wide.
myCC.radius= 2;

// Set the axis of lengthwise orientation to the X axis.
myCC.direction = 0;

// Set the axis of lengthwise orientation to the Y axis.
myCC.direction = 1;

// Set the axis of lengthwise orientation to the Y axis.
myCC.direction = 2;
```

Коллайдер колес

свойства

- **Масса** - масса колесного коллайдера
- **Радиус** - радиус в локальном пространстве

- **Коэффициент** демпфирования **колес** - значение демпфирования для коллайдера колес
- **Расстояние подвески** - максимальное удлинение вдоль оси Y в локальном пространстве
- **Расстояние до точки приложения** - точка, в которой будут применены силы,
- **Центр** - Центр коллайдера колес в местном пространстве

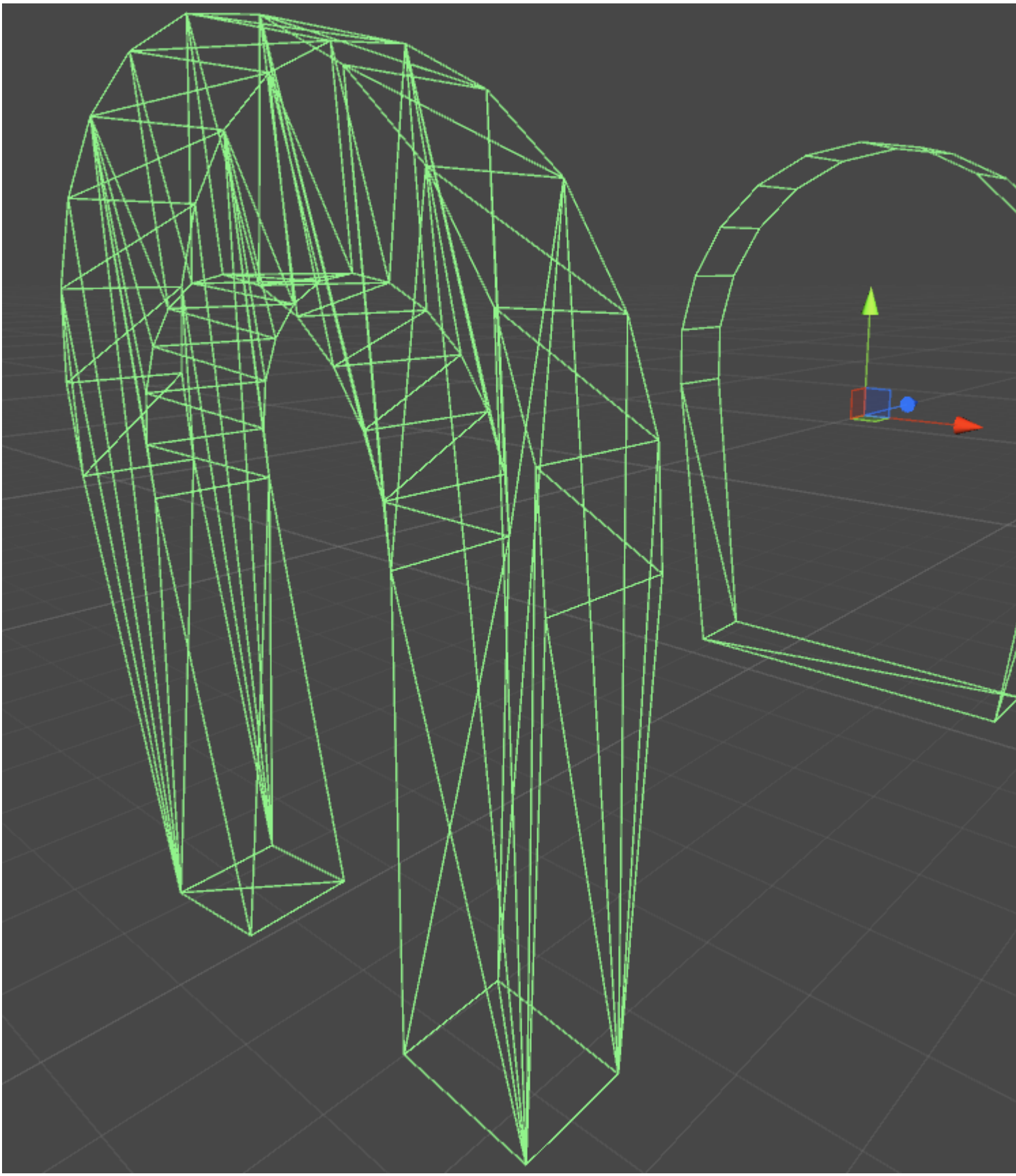
Подвеска Весна

- **Весна** - скорость, с которой колесо пытается вернуться в целевое положение
- **Damper** - большее значение уменьшает скорость больше, а подвеска движется медленнее
- **Целевая позиция** - по умолчанию - 0,5, при 0 - нижняя нижняя часть подвески, при 1 - при полном растяжении
- **Прямое** / **боковое трение** - как работает шина при движении вперед или вбок

пример

Сетчатый коллайдер

Коллайдер на основе Mesh Asset.



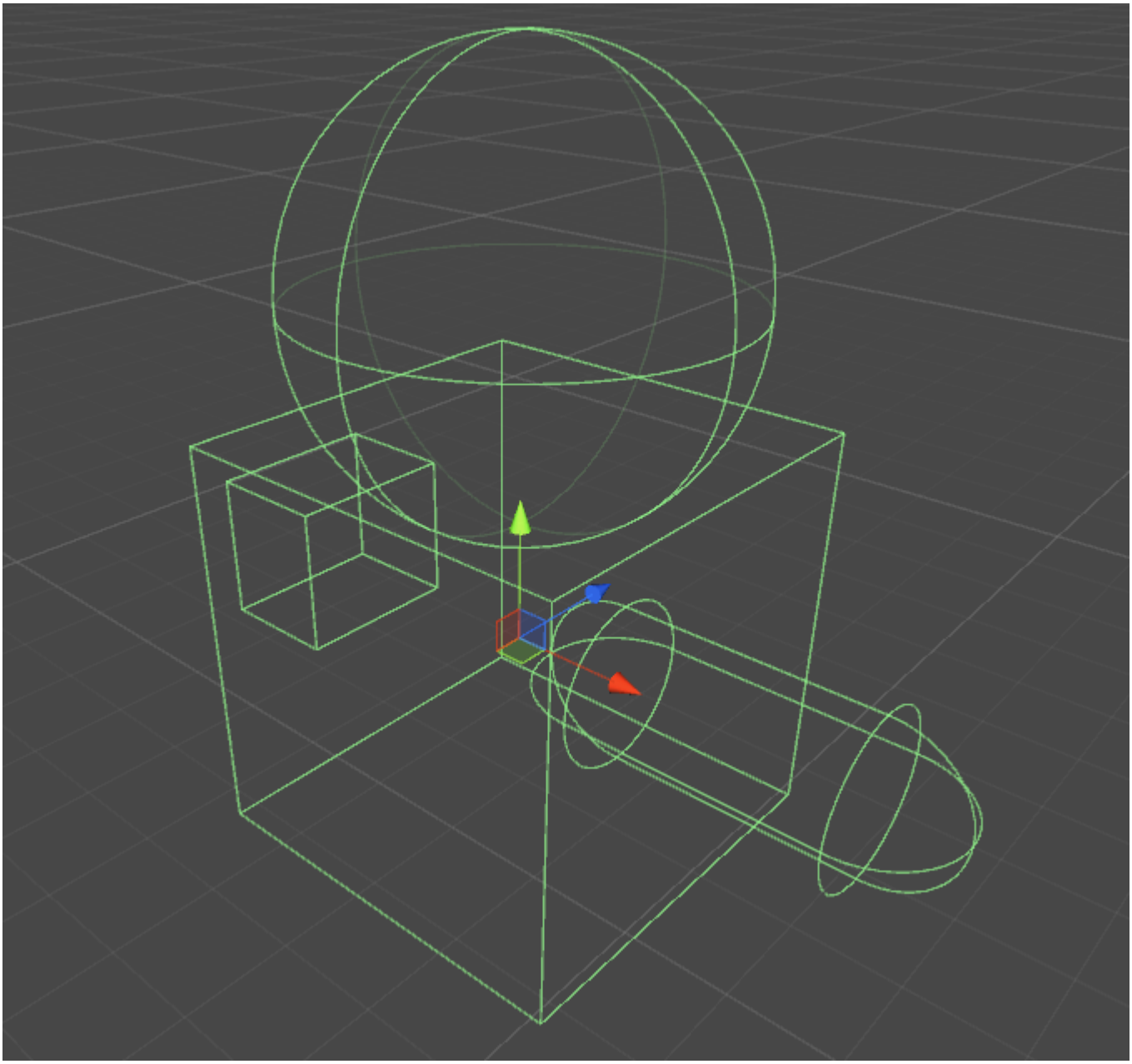
свойства

- **Является триггером** - если отмечено, Коллайдер Box игнорирует физику и становится триггерным коллайдером

- **Материал** - ссылка, если она указана, физическому материалу Box Collider
- **Mesh** - ссылка на сетку, на которой работает коллайдер, основана на
- **Convex** - Convex Mesh-коллайдеры ограничены 255 полигонами - если этот параметр включен, этот коллайдер может столкнуться с другими коллайдерами сетки

пример

Если вы применяете несколько коллайдеров к GameObject, мы называем это сложным коллайдером.



Коллайдер колес

Коллайдер колеса внутри единства построен на коллайдере колеса PhysX от Nvidia и,

следовательно, обладает многими аналогичными свойствами. Технически единство - это «бесплотная» программа, но для того, чтобы все имело смысл, требуются некоторые стандартные единицы.

Основные свойства

- Масса - вес колеса в килограммах, это используется для импульса колеса и момента взаимодействия при вращении.
- Радиус - в метрах, радиус коллайдера.
- Колесная скорость демпфирования - регулирует, как «реагировать» на крутящий момент.
- Расстояние подвески - общее расстояние в метрах, которое колесо может перемещаться
- Force App Point Distance - где сила от подвески применяется к материнской жесткости
- Центр - центральное положение колеса

Настройки подвески

- Весна - это постоянная пружины, K, в Ньютонах / метр в уравнении:

  Force = Spring Constant * Расстояние

Хорошей отправной точкой для этого значения должна быть общая масса вашего автомобиля, деленная на количество колес, умноженная на число от 50 до 100. Например, если у вас есть автомобиль объемом 2 000 кг с 4 колесами, тогда каждое колесо должно поддержка 500 кг. Умножьте это на 75, и ваша постоянная пружины должна составлять 37 500 ньютонов / метр.

- Демпфер - эквивалент амортизатора в автомобиле. Более высокие показатели делают ожидание «более жестким» и более низкие ставки делают его «более мягким» и более склонным к колебаниям.
  Я не знаю единиц или уравнений для этого, я думаю, что это имеет отношение к частотным уравнениям в физике.

Боковые настройки трения

Кривая трения в единице имеет значение скольжения, определяемое тем, сколь скользит колесо (в м / с) от желаемого положения относительно фактического положения.

- Extremum Slip - это максимальное количество (в м / с), когда колесо может скользить, прежде чем оно потеряет сцепление

- Экстремальное значение - это максимальное количество трения, которое должно применяться к колесу.

Значения для Exrtremum Slip должны быть от 0,2 до 2 м / с для большинства реалистичных

автомобилей. 2 м / с составляет около 6 футов в секунду или 5 миль в час, что является большим количеством промахов. Если вы считаете, что ваше транспортное средство должно иметь значение выше 2 м / с для скольжения, вы должны учитывать увеличение максимального трения (значение экстремума).

Max Fraction (Extremum Value) - коэффициент трения в уравнении:

> Сила трения (в ньютонах) = Коэффициент трения * Сила нисходящего движения (в ньютонах)

Это означает, что с коэффициентом 1, вы применяете всю силу автомобиля + подвеску, противоположную направлению скольжения. В реальных приложениях значения, превышающие 1, редки, но не невозможны. Для шины на сухом асфальте значения между .7 и .9 реалистичны, поэтому предпочтительнее значение по умолчанию 1,0.

Это значение не должно реалистично не превышать 2,5, так как произойдет странное поведение. Например, вы начинаете поворачивать направо, но поскольку это значение настолько велико, большая сила применяется против вашего направления, и вы начинаете скользить в поворот, а не прочь.

Если у вас есть maxed оба значения, вы должны начать поднимать асимптотическое скольжение и значение. Асимптотический сдвиг должен быть между 0,5 и 2 м / с и определяет коэффициент трения для любого значения скольжения за пропуском Asymptote. Если вы обнаружите, что ваши транспортные средства ведут себя хорошо, пока не сломают тягу, в этот момент он действует так, как будто на льду, вы должны повысить значение Asymptote. Если вы обнаружите, что ваш автомобиль не может дрейфовать, вы должны снизить его значение.

Прямое трение

Прямое трение идентично боковому трению, за исключением того, что это определяет, насколько тяга колесо имеет направление движения. Если значения слишком низкие, ваши автомобили будут делать выгорания и просто вращать шины, прежде чем двигаться вперед, медленно. Если он слишком высок, ваше транспортное средство может иметь тенденцию пытаться делать или, что еще хуже, переворачиваться.

Дополнительные примечания

Не ожидайте, что сможете создать клон GTA или другой гоночный клон, просто изменив эти значения. В большинстве игр вождения эти значения постоянно меняются в сценарии для разных скоростей, ландшафтов и поворотных значений. Кроме того, если вы просто применяете постоянный крутящий момент к коллайдерам колес при нажатии клавиши, ваша игра не будет вести себя реалистично. В реальном мире автомобили имеют кривые крутящего момента и трансмиссии для изменения крутящего момента, приложенного к колесам.

Для достижения наилучших результатов вы должны настроить эти значения до тех пор, пока автомобиль не будет достаточно хорошо реагировать, а затем внесите изменения в крутящий момент колеса, максимальный угол поворота и значения трения в сценарии.

Более подробную информацию о коллайдерах колес можно найти в документации Nvidia: http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Vehicles.html

Триггерные коллайдеры

методы

- `OnTriggerEnter()`
- `OnTriggerStay()`
- `OnTriggerExit()`

Вы можете сделать коллайдер в **триггер** , чтобы использовать `OnTriggerEnter()` , `OnTriggerStay()` и `OnTriggerExit()` . Триггер-коллайдер физически не реагирует на столкновения, другие GameObjects просто проходят через него. Они полезны для обнаружения, когда другой GameObject находится в определенной области или нет, например, при сборе элемента, мы можем захотеть просто запустить его, но обнаружить, когда это произойдет.

Скрипты триггера-коллайдера

пример

Ниже приведен пример триггерного прослушивателя, который обнаруживает, когда другой коллайдер входит в коллайдер GameObject (например, игрок). Триггерные методы могут быть добавлены в любой скрипт, назначенный GameObject.

```
void OnTriggerEnter(Collider other)
{
    //Check collider for specific properties (Such as tag=item or has component=item)
}
```

Прочитайте коллизия онлайн: https://riptutorial.com/ru/unity3d/topic/4405/коллизия

глава 21: Магазин активов

Examples

Доступ к хранилищу активов

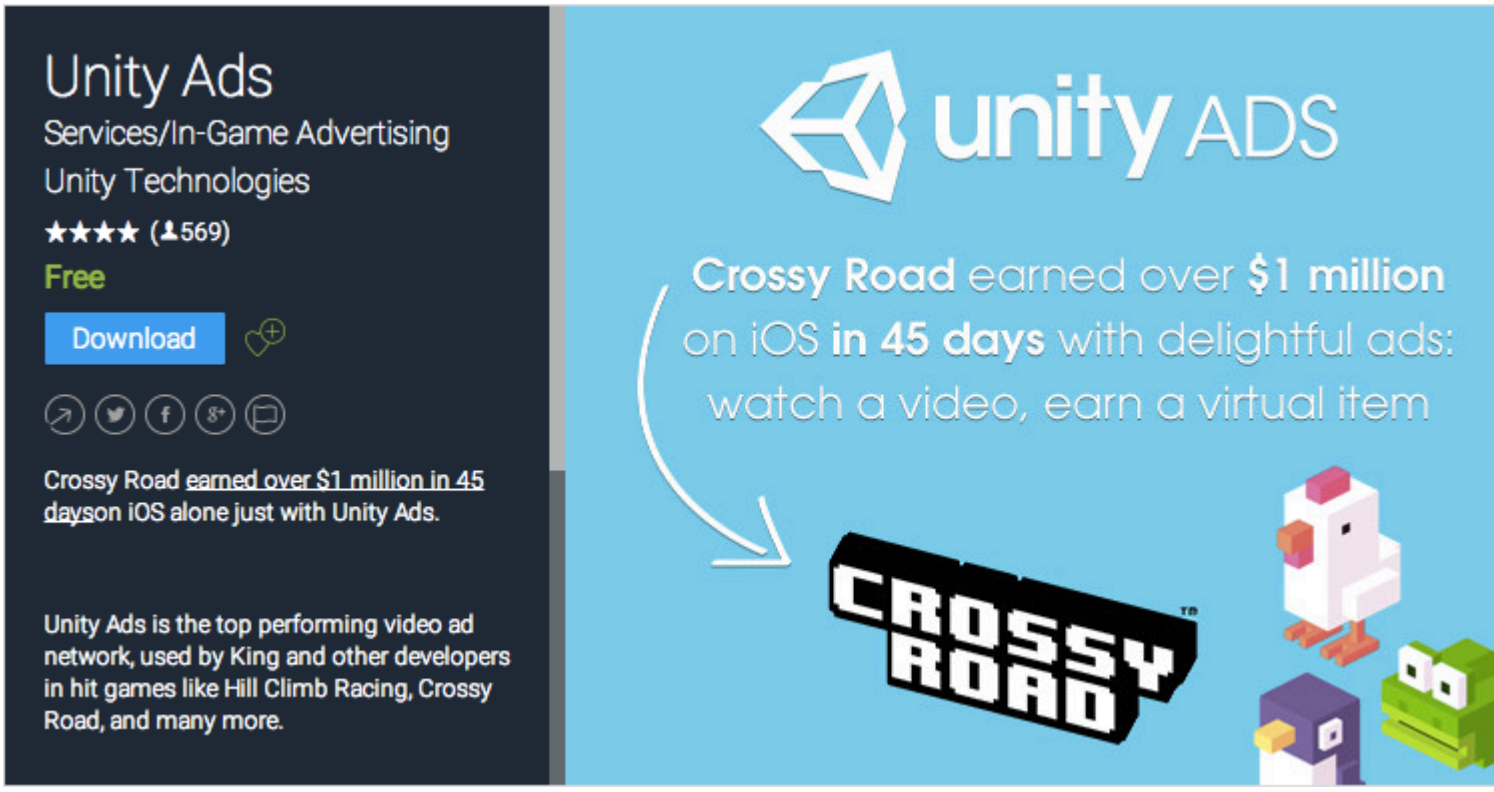
Существует три способа доступа к хранилищу активов Unity:

- Откройте окно «Магазин активов», выбрав «Окно» → «Хранилище активов» в главном меню «Единство».
- Используйте клавишу быстрого доступа (Ctrl + 9 в Windows / 9 в Mac OS)
- Просмотрите веб-интерфейс: https://www.assetstore.unity3d.com/

Возможно, вам будет предложено создать бесплатную учетную запись пользователя или войти в систему, если вы впервые получаете доступ к хранилищу активов Unity.

Покупка активов

После доступа к хранилищу активов и просмотра актива, который вы хотите загрузить, просто нажмите кнопку « **Загрузить»** . Текст кнопки также может быть « **Покупать»,** если у объекта есть соответствующая стоимость.



Если вы просматриваете Unity Asset Store через веб-интерфейс, текст кнопки « **Загрузить»** может отображаться как « **Открыть в Unity»** . Выбор этой кнопки приведет к запуску экземпляра Unity и отображению актива в окне *Asset Store* .

Возможно, вам будет предложено создать бесплатную учетную запись пользователя или войти в систему, если это ваша первая покупка в магазине Unity Asset Store.

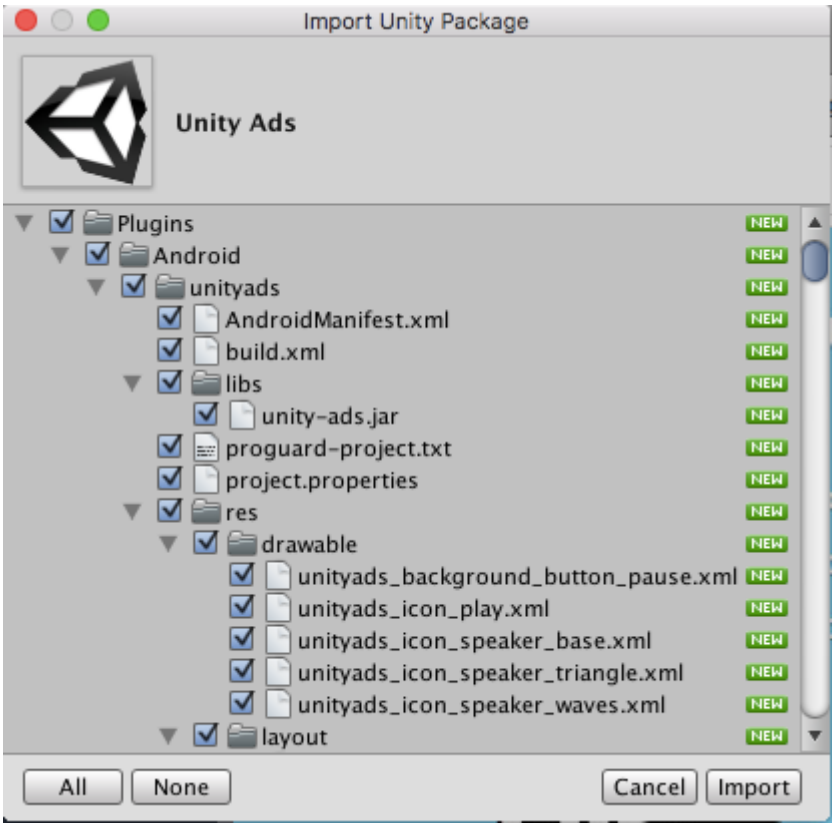
Unity затем приступит к принятию вашего платежа, если это применимо.

Импорт активов

После того как ресурс был загружен в Unity, кнопка « **Загрузить»** или « **Купить сейчас»** изменится на « **Импорт»** .

Выбор этого параметра подскажет пользователю окно *Import Unity Package* , в котором пользователь может выбрать файлы активов, которые они хотели бы импортировать в рамках своего проекта.

Выберите « **Импорт»,** чтобы подтвердить процесс, поместив выбранные файлы активов в папку «Активы», показанные в *окне «Просмотр проекта»* .



Издательские активы

1. сделать учетную запись издателя
2. добавить актив в учетной записи издателя
3. загрузить инструменты хранилища активов (из хранилища активов)
4. перейдите в раздел «Инструменты хранилища активов»> «Загрузка пакета»
5. выберите правильный пакет и папку проекта в окне инструментов хранилища активов
6. загрузить клик
7. представить свой актив в Интернете

TODO - добавьте картинки, подробнее

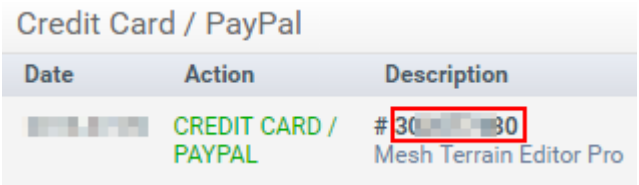
Подтвердите номер счета одной покупки

Номер счета-фактуры используется для проверки продажи для издателей. Многие издатели платного актива или плагина запрашивают номер счета-фактуры по запросу поддержки. Номер счета также используется в качестве лицензионного ключа для активации какого-либо актива или плагина.

Номер счета-фактуры можно найти в двух местах:

1. После того, как вы купили актива, вам будет отправлено электронное письмо, предметом которого является «Подтверждение покупки в магазине Unity Asset Store ...». Номер счета-фактуры находится в прикрепленном PDF-сообщении этого электронного письма.

2. Откройте https://www.assetstore.unity3d.com/#!/account/transactions , затем вы можете найти номер счета в столбце *Описание* .



Прочитайте Магазин активов онлайн: https://riptutorial.com/ru/unity3d/topic/5705/магазин-активов

глава 22: Мобильные платформы

Синтаксис

- public static int Input.touchCount
- public static Touch Input.GetTouch (индекс int)

Examples

Обнаружение касания

Чтобы обнаружить прикосновение в Unity, довольно просто, нам просто нужно использовать `Input.GetTouch()` и передать ему индекс.

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Began)
        {
            //Do Stuff
        }
    }
}
```

или же

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        for(int i = 0; i < Input.touchCount; i++)
        {
            if (Input.GetTouch(i).phase == TouchPhase.Began)
            {
                //Do Stuff
            }
        }
    }
}
```

Эти примеры приходят в контакт с последней игровой рамкой.

TouchPhase

Внутри перечисления TouchPhase есть 5 различных типов TouchPhase

- Начался - пальцем коснулся экран
- Перемещено - палец перемещается по экрану
- Стационарный - палец отображается на экране, но не перемещается
- Закончено - палец был снят с экрана
- Отменено - система отменяет отслеживание касания

Например, чтобы переместить объект, этот скрипт привязан к экрану на основе касания.

```
public class TouchMoveExample : MonoBehaviour
{
    public float speed = 0.1f;

    void Update () {
        if(Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Moved)
        {
            Vector2 touchDeltaPosition = Input.GetTouch(0).deltaPosition;
            transform.Translate(-touchDeltaPosition.x * speed, -touchDeltaPosition.y * speed,
0);
        }
    }
}
```

Прочитайте Мобильные платформы онлайн: https://riptutorial.com/ru/unity3d/topic/6285/мобильные-платформы

глава 23: Мультиплатформенная разработка

Examples

Определения компилятора

Определения компилятора используют код конкретной платформы. Используя их, вы можете делать небольшие различия между различными платформами.

- Достижения Trigger Game Center на устройствах Apple и достижения Google Play на устройствах Android.
- Измените значки в меню (логотип Windows в Windows, Linux penguin в Linux).
- Возможно, в зависимости от платформы возможно наличие специфичной для платформы механики.
- И многое другое...

```
void Update(){

#if UNITY_IPHONE
    //code here is only called when running on iPhone
#endif

#if UNITY_STANDALONE_WIN && !UNITY_EDITOR
    //code here is only ran in a unity game running on windows outside of the editor
#endif

//other code that will be ran regardless of platform

}
```

Полный список определений компилятора Unity можно найти здесь

Организация специальных методов платформы для частичных классов

Частичные классы обеспечивают чистый способ отделить основную логику ваших скриптов от конкретных методов платформы.

Частичные классы и методы отмечены ключевым словом `partial` . Это сигнализирует компилятору оставить класс «открытым» и посмотреть в других файлах для остальной части реализации.

```
// ExampleClass.cs
using UnityEngine;

public partial class ExampleClass : MonoBehaviour
```

```
{
    partial void PlatformSpecificMethod();

    void OnEnable()
    {
        PlatformSpecificMethod();
    }
}
```

Теперь мы можем создавать файлы для наших скриптов, специфичных для платформы, которые реализуют частичный метод. Частичные методы могут иметь параметры (также `ref`), но должны возвращать `void` .

```
// ExampleClass.Iphone.cs

#if UNITY_IPHONE
using UnityEngine;

public partial class ExampleClass
{
    partial void PlatformSpecificMethod()
    {
        Debug.Log("I am an iPhone");
    }
}
#endif
```

```
// ExampleClass.Android.cs

#if UNITY_ANDROID
using UnityEngine;

public partial class ExampleClass
{
    partial void PlatformSpecificMethod()
    {
        Debug.Log("I am an Android");
    }
}
#endif
```

Если частичный метод не реализован, компилятор опустит вызов.

> Совет. Этот шаблон полезен при создании специальных методов редактора.

Прочитайте Мультиплатформенная разработка онлайн:
https://riptutorial.com/ru/unity3d/topic/4816/мультиплатформенная-разработка

глава 24: Объединение объектов

Examples

Пул объектов

Иногда, когда вы делаете игру, вам нужно создавать и уничтожать много объектов одного и того же типа снова и снова. Вы можете просто сделать это, сделав сборку и инстанцируйте / уничтожьте это, когда вам нужно, однако, это неэффективно и может замедлить игру.

Один из способов обойти эту проблему - объединение объектов. В основном это означает, что у вас есть пул (с или без ограничения количества) объектов, которые вы собираетесь использовать повторно, когда это возможно, чтобы предотвратить ненужное создание или уничтожение.

Ниже приведен пример простого пула объектов

```
public class ObjectPool : MonoBehaviour
{
    public GameObject prefab;
    public int amount = 0;
    public bool populateOnStart = true;
    public bool growOverAmount = true;

    private List<GameObject> pool = new List<GameObject>();

    void Start()
    {
        if (populateOnStart && prefab != null && amount > 0)
        {
            for (int i = 0; i < amount; i++)
            {
                var instance = Instantiate(Prefab);
                instance.SetActive(false);
                pool.Add(instance);
            }
        }
    }

    public GameObject Instantiate (Vector3 position, Quaternion rotation)
    {
        foreach (var item in pool)
        {
            if (!item.activeInHierarchy)
            {
                item.transform.position = position;
                item.transform.rotation = rotation;
                item.SetActive( true );
                return item;
            }
        }
```

```
        if (growOverAmount)
        {
            var instance = (GameObject)Instantiate(prefab, position, rotation);
            pool.Add(instance);
            return instance;
        }

        return null;
    }
}
```

Перейдем сначала к переменным

```
public GameObject prefab;
public int amount = 0;
public bool populateOnStart = true;
public bool growOverAmount = true;

private List<GameObject> pool = new List<GameObject>();
```

- `GameObject prefab` : это сборник, который пул объектов будет использовать для создания новых объектов в пуле.
- `int amount` : Это максимальное количество предметов, которые могут быть в пуле. Если вы хотите создать экземпляр другого элемента, и пул уже достиг своего предела, будет использоваться другой элемент из пула.
- `bool populateOnStart` : вы можете выбрать заполнение пула при запуске или нет. Это приведет к заполнению пула экземплярами сборника, так что при первом вызове `Instantiate` вы получите уже существующий объект
- `bool growOverAmount` : установка этого значения в true позволяет пулу расти, всякий раз, когда запрашивается сумма в определенный промежуток времени. Вы не всегда можете точно предсказать количество предметов, которые нужно положить в ваш пул, чтобы при необходимости добавить в свой бассейн больше.
- `List<GameObject> pool` : это пул, место, где хранятся все ваши экземпляры / уничтоженные объекты.

Теперь давайте посмотрим на функцию `Start`

```
void Start()
{
    if (populateOnStart && prefab != null && amount > 0)
    {
        for (int i = 0; i < amount; i++)
        {
            var instance = Instantiate(Prefab);
            instance.SetActive(false);
            pool.Add(instance);
        }
    }
}
```

В функции запуска мы проверяем, нужно ли заполнять список при запуске и делать это,

если `prefab` была установлена, а количество больше, чем 0 (иначе мы будем создавать бесконечно).

Это просто простой цикл, создающий новые объекты и помещая их в пул. Следует обратить внимание на то, что мы установили все экземпляры в неактивные. Таким образом, они еще не видны в игре.

Далее, есть функция `Instantiate` , в которой происходит большая часть магии

```
public GameObject Instantiate (Vector3 position, Quaternion rotation)
{
    foreach (var item in pool)
    {
        if (!item.activeInHierarchy)
        {
            item.transform.position = position;
            item.transform.rotation = rotation;
            item.SetActive(true);
            return item;
        }
    }

    if (growOverAmount)
    {
        var instance = (GameObject)Instantiate(prefab, position, rotation);
        pool.Add(instance);
        return instance;
    }

    return null;
}
```

`Instantiate` функция выглядит так же , как и собственное единство по `Instantiate` функция, кроме сборной уже указана выше в качестве члена класса.

Первый шаг функции `Instantiate` проверяет, есть ли в пуле неактивный объект прямо сейчас. Это означает, что мы можем повторно использовать этот объект и вернуть его запрашивающему. Если в пуле есть неактивный объект, мы устанавливаем позицию и поворот, устанавливаем его активным (иначе его можно было бы повторно использовать случайно, если вы забыли его активировать) и вернуть его запрашивающему.

Второй шаг возможен только в том случае, если в пуле нет неактивных элементов, и пул может расти по сравнению с начальной суммой. Все просто: добавляется другой экземпляр сборника и добавляется в пул. Разрешение роста пула помогает вам иметь нужное количество объектов в пуле.

Третий «шаг» происходит только в том случае, если в пуле нет неактивных элементов, и пул *не* может расти. Когда это произойдет, запросчик получит нулевой объект GameObject, что означает, что ничего не было доступно и должно быть обработано должным образом, чтобы предотвратить `NullReferenceExceptions` .

Важный!

Чтобы ваши предметы возвращались в пул, вы **не** должны уничтожать игровые объекты. Единственное, что вам нужно сделать, это установить их в неактивные и сделать их доступными для повторного использования через пул.

Простой пул объектов

Ниже приведен пример пула объектов, который позволяет арендовать и возвращать заданный тип объекта. Чтобы создать пул объектов, необходим Func для функции create и Action для уничтожения объекта, чтобы предоставить пользователю гибкость. При запросе объекта, когда пул пуст, будет создан новый объект и при запросе, когда пул имеет объекты, тогда объекты удаляются из пула и возвращаются.

Пул объектов

```
public class ResourcePool<T> where T : class
{
    private readonly List<T> objectPool = new List<T>();
    private readonly Action<T> cleanUpAction;
    private readonly Func<T> createAction;

    public ResourcePool(Action<T> cleanUpAction, Func<T> createAction)
    {
        this.cleanUpAction = cleanUpAction;
        this.createAction = createAction;
    }

    public void Return(T resource)
    {
        this.objectPool.Add(resource);
    }

    private void PurgeSingleResource()
    {
        var resource = this.Rent();
        this.cleanUpAction(resource);
    }

    public void TrimResourcesBy(int count)
    {
        count = Math.Min(count, this.objectPool.Count);
        for (int i = 0; i < count; i++)
        {
            this.PurgeSingleResource();
        }
    }

    public T Rent()
    {
        int count = this.objectPool.Count;
        if (count == 0)
        {
            Debug.Log("Creating new object.");
            return this.createAction();
        }
```

```
        else
        {
            Debug.Log("Retrieving existing object.");
            T resource = this.objectPool[count-1];
            this.objectPool.RemoveAt(count-1);
            return resource;
        }
    }
}
```

Пример использования

```
public class Test : MonoBehaviour
{
    private ResourcePool<GameObject> objectPool;

    [SerializeField]
    private GameObject enemyPrefab;

    void Start()
    {
        this.objectPool = new ResourcePool<GameObject>(Destroy,() =>
Instantiate(this.enemyPrefab) );
    }

    void Update()
    {
        // To get existing object or create new from pool
        var newEnemy = this.objectPool.Rent();
        // To return object to pool
        this.objectPool.Return(newEnemy);
        // In this example the message 'Creating new object' should only be seen on the frame
call
        // after that the same object in the pool will be returned.
    }
}
```

Еще один простой пул объектов

Другой пример: Оружие, которое стреляет из Пули.

Оружие действует как пул объектов для создаваемых им Пули.

```
public class Weapon : MonoBehaviour {

    // The Bullet prefab that the Weapon will create
    public Bullet bulletPrefab;

    // This List is our object pool, which starts out empty
    private List<Bullet> availableBullets = new List<Bullet>();

    // The Transform that will act as the Bullet starting position
    public Transform bulletInstantiationPoint;

    // To spawn a new Bullet, this method either grabs an available Bullet from the pool,
    // otherwise Instantiates a new Bullet
    public Bullet CreateBullet () {
```

```
        Bullet newBullet = null;

        // If a Bullet is available in the pool, take the first one and make it active
        if (availableBullets.Count > 0) {
            newBullet = availableBullets[availableBullets.Count - 1];

            // Remove the Bullet from the pool
            availableBullets.RemoveAt(availableBullets.Count - 1);

            // Set the Bullet's position and make its GameObject active
            newBullet.transform.position = bulletInstantiationPoint.position;
            newBullet.gameObject.SetActive(true);
        }
        // If no Bullets are available in the pool, Instantiate a new Bullet
        else {
            newBullet newObject = Instantiate(bulletPrefab, bulletInstantiationPoint.position,
Quaternion.identity);

            // Set the Bullet's Weapon so we know which pool to return to later on
            newBullet.weapon = this;
        }

        return newBullet;
    }

}

public class Bullet : MonoBehaviour {

    public Weapon weapon;

    // When Bullet collides with something, rather than Destroying it, we return it to the
pool
    public void ReturnToPool () {
        // Add Bullet to the pool
        weapon.availableBullets.Add(this);

        // Disable the Bullet's GameObject so it's hidden from view
        gameObject.SetActive(false);
    }

}
```

Прочитайте Объединение объектов онлайн: https://riptutorial.com/ru/unity3d/topic/2276/объединение-объектов

глава 25: оптимизация

замечания

1. Если возможно, отключите скрипты на объектах, когда они не нужны. Например, если у вас есть скрипт на объекте противника, который ищет и стреляет в игрока, подумайте об отключении этого скрипта, когда противник слишком далеко, например, от игрока.

Examples

Быстрые и эффективные проверки

Избегайте ненужных операций и вызовов методов, где бы вы ни находились, особенно в методе, который вызывается много раз в секунду, например « `Update` .

Проверка расстояния / диапазона

При сравнении расстояний используйте `sqrMagnitude` вместо `magnitude` . Это позволяет избежать ненужных операций `sqrt` . Обратите внимание, что при использовании `sqrMagnitude` правая часть также должна быть в квадрате.

```
if ((target.position - transform.position).sqrMagnitude < minDistance * minDistance))
```

Проверка границ

Пересечения объектов можно грубо проверить, проверив, пересекаются ли их границы `Collider` / `Renderer` . Структура `Bounds` также имеет удобный метод `Intersects` который помогает определить, пересекаются ли две границы.

`Bounds` также помогают нам рассчитать приблизительное приблизительное *фактическое* (от поверхности к поверхности) расстояние между объектами (см. `Bounds.SqrDistance`).

Предостережения

Проверка границ работает очень хорошо для выпуклых объектов, но проверки границ вогнутых объектов могут приводить к гораздо более высоким неточностям в зависимости от формы объекта.

Использование `Mesh.bounds` не рекомендуется, поскольку оно возвращает границы локального пространства. `MeshRenderer.bounds` этого используйте `MeshRenderer.bounds` .

Мощность Coroutine

использование

Если у вас длительная работа, основанная на небезопасном Unity API, используйте Coroutines, чтобы разделить ее на несколько фреймов и откликнуться на ваше приложение.

Coroutines также помогает выполнять дорогостоящие действия для каждого n-го кадра вместо того, чтобы запускать это действие в каждом кадре.

Разделение длинных очередей на несколько кадров

Coroutines помогают распределять длительные операции над несколькими кадрами, чтобы поддерживать скорость работы вашего приложения.

Процедуры, которые рисуют или генерируют рельеф в процедуре или создают шум, являются примерами, которые могут потребовать обработки Coroutine.

```
for (int y = 0; y < heightmap.Height; y++)
{
    for (int x = 0; x < heightmap.Width; x++)
    {
        // Generate pixel at (x, y)
        // Assign pixel at (x, y)

        // Process only 32768 pixels each frame
        if ((y * heightmap.Height + x) % 32 * 1024) == 0)
            yield return null; // Wait for next frame
    }
}
```

> Приведенный выше код является простым для понимания примером. В производственном коде лучше избегать чек на пиксель , который проверяет , когда `yield return` (возможно делать это через каждые 2-3 ряда) и предварительно рассчитать `for` длине петли заранее.

Выполнение дорогостоящих действий

менее часто

Coroutines помогает вам выполнять дорогостоящие действия реже, так что это не так сильно, как это было бы при каждом кадре.

Приведя следующий пример непосредственно из Руководства :

```
private void ProximityCheck()
{
    for (int i = 0; i < enemies.Length; i++)
    {
        if (Vector3.Distance(transform.position, enemies[i].transform.position) <
dangerDistance)
                return true;
    }
    return false;
}

private IEnumerator ProximityCheckCoroutine()
{
    while(true)
    {
        ProximityCheck();
        yield return new WaitForSeconds(.1f);
    }
}
```

Тесты близости можно еще более оптимизировать, используя API CullingGroup .

Общие проблемы

Разработчики общей ошибки делают доступ к результатам или побочным эффектам сопрограммы *вне* сопрограммы. Coroutines возвращает управление вызывающему абоненту, как только встретится оператор `yield return` и результат или побочный эффект еще не могут быть выполнены. Чтобы обойти проблемы, когда вы *должны* использовать результат / побочный эффект вне сопрограммы, проверьте этот ответ .

Струны

Можно утверждать, что в Unity больше богов ресурсов, чем скромная строка, но это один из самых простых аспектов для исправления на ранней стадии.

Строковые операции создают мусор

Большинство операций с строкой создают крошечные суммы мусора, но если эти операции вызывают несколько раз в течение одного обновления, он складывается. Со временем это

вызовет автоматическую сборку мусора, которая может привести к видимому всплеску процессора.

Кэш ваших строковых операций

Рассмотрим следующий пример.

```
string[] StringKeys = new string[] {
    "Key0",
    "Key1",
    "Key2"
};

void Update()
{
    for (var i = 0; i < 3; i++)
    {
        // Cached, no garbage generated
        Debug.Log(StringKeys[i]);
    }

    for (var i = 0; i < 3; i++)
    {
        // Not cached, garbage every cycle
        Debug.Log("Key" + i);
    }

    // The most memory-efficient way is to not create a cache at all and use literals or
constants.
    // However, it is not necessarily the most readable or beautiful way.
    Debug.Log("Key0");
    Debug.Log("Key1");
    Debug.Log("Key2");
}
```

Это может выглядеть глупо и избыточно, но если вы работаете с шейдерами, вы можете столкнуться с такими ситуациями. Кэширование клавиш будет иметь значение.

Обратите внимание, что строковые *литералы* и *константы* не генерируют никакого мусора, так как они вставляются статически в пространство стека программ. Если вы *генерируете* строки во время выполнения и *гарантированно* генерируете **одни и** те **же** строки каждый раз, как в приведенном выше примере, кеширование определенно поможет.

В других случаях, когда генерируемая строка не является одинаковой каждый раз, нет другой альтернативы для генерации этих строк. Таким образом, всплеск памяти с ручным генерированием строк каждый раз обычно ничтожно, если только десятки тысяч строк не генерируются за раз.

Большинство строковых операций являются сообщениями Debug

Выполнение строковых операций для сообщений Debug, т.е. `Debug.Log("Object Name: " + obj.name)` в порядке и не может быть предотвращено во время разработки. Тем не менее, важно обеспечить, чтобы нерелевантные отладочные сообщения не попадали в выпущенный продукт.

Один из способов - использовать атрибут Conditional в ваших отладочных вызовах. Это не только удаляет вызовы метода, но и все операции с строкой, входящие в него.

```
using UnityEngine;
using System.Collections;

public class ConditionalDebugExample: MonoBehaviour
{
    IEnumerator Start()
    {
        while(true)
        {
            // This message will pop up in Editor but not in builds
            Log("Elapsed: " + Time.timeSinceLevelLoad);
            yield return new WaitForSeconds(1f);
        }
    }

    [System.Diagnostics.Conditional("UNITY_EDITOR")]
    void Log(string Message)
    {
        Debug.Log(Message);
    }
}
```

> Это упрощенный пример. Возможно, вы захотите потратить некоторое время на разработку более полноценной рутинной процедуры регистрации.

Сравнение строк

Это небольшая оптимизация, но стоит упомянуть. Сравнение строк немного больше, чем можно было бы подумать. По умолчанию система будет пытаться учитывать культурные различия. Вместо этого вы можете использовать простое двоичное сравнение, которое выполняется быстрее.

```
// Faster string comparison
if (strA.Equals(strB, System.StringComparison.Ordinal)) {...}
// Compared to
if (strA == strB) {...}

// Less overhead
if (!string.IsNullOrEmpty(strA)) {...}
// Compared to
if (strA == "") {...}

// Faster lookups
Dictionary<string, int> myDic = new Dictionary<string, int>(System.StringComparer.Ordinal);
```

```
// Compared to
Dictionary<string, int> myDictionary = new Dictionary<string, int>();
```

Ссылки на кеш

Ссылки на кеш, чтобы избежать дорогостоящих вызовов, особенно в функции обновления. Это можно сделать путем кэширования этих ссылок при запуске, если они доступны или когда они доступны, и проверки нулевого / bool flat, чтобы избежать повторной ссылки.

Примеры:

Ссылки на компоненты кэша

менять

```
void Update()
{
    var renderer = GetComponent<Renderer>();
    renderer.material.SetColor("_Color", Color.green);
}
```

в

```
private Renderer myRenderer;
void Start()
{
    myRenderer = GetComponent<Renderer>();
}

void Update()
{
    myRenderer.material.SetColor("_Color", Color.green);
}
```

Ссылки на объекты кэша

менять

```
void Update()
{
    var enemy = GameObject.Find("enemy");
    enemy.transform.LookAt(new Vector3(0,0,0));
}
```

в

```
private Transform enemy;

void Start()
{
    this.enemy = GameObject.Find("enemy").transform;
}
```

```
void Update()
{
    enemy.LookAt(new Vector3(0, 0, 0));
}
```

Кроме того, кешируйте дорогие вызовы, например, звонки в Mathf, где это возможно.

Избегайте вызывать методы, используя строки

Избегайте вызывать методы, используя строки, которые могут принимать методы. Этот подход будет использовать отражение, которое может замедлить вашу игру, особенно при использовании в функции обновления.

Примеры:

```
    //Avoid StartCoroutine with method name
    this.StartCoroutine("SampleCoroutine");

    //Instead use the method directly
    this.StartCoroutine(this.SampleCoroutine());

    //Avoid send message
    var enemy = GameObject.Find("enemy");
    enemy.SendMessage("Die");

    //Instead make direct call
    var enemy = GameObject.Find("enemy") as Enemy;
    enemy.Die();
```

Избегайте пустых методов единства

Избегайте пустых методов единства. Помимо плохого стиля программирования, во время исполнения сценариев очень мало накладных расходов. Во многих случаях это может создать и повлиять на производительность.

```
void Update
{
}

void FixedUpdate
{
}
```

Прочитайте оптимизация онлайн: https://riptutorial.com/ru/unity3d/topic/3433/оптимизация

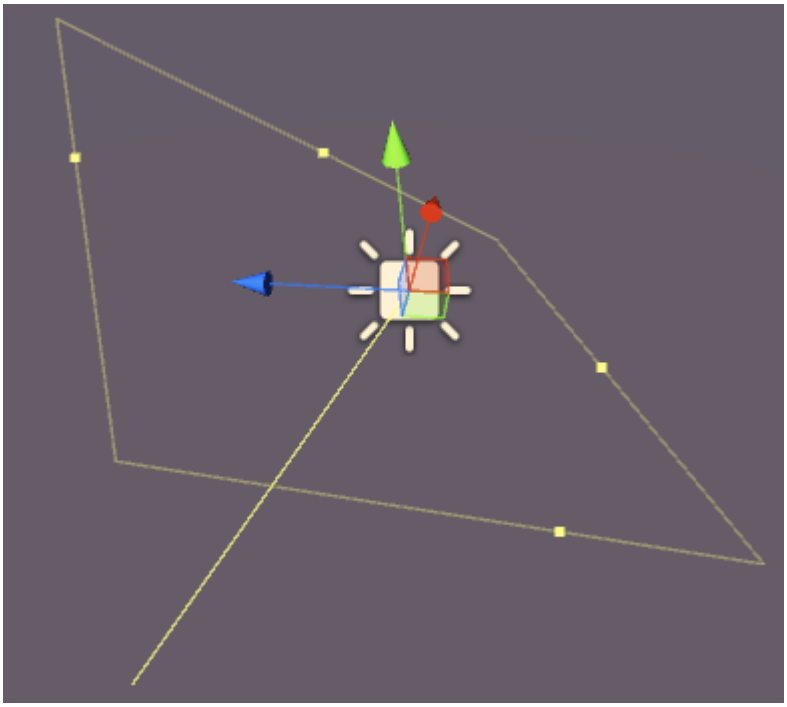
глава 26: Освещение единства

Examples

Типы света

Зональный свет

Свет излучается по поверхности прямоугольной области. Они испечены только, что означает, что вы не сможете увидеть эффект, пока не испепете сцену.

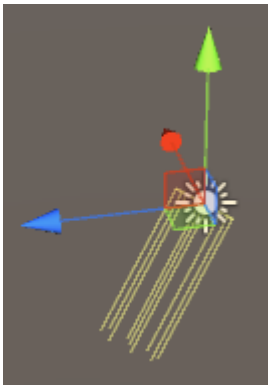


Облачные огни обладают следующими свойствами:

- **Ширина** - ширина области света.
- **Высота** - высота зоны света.
- **Цвет.** Назначьте цвет света.
- **Интенсивность** - насколько силен свет от 0 до 8.
- **Интенсивность отскока** - насколько мощный *косвенный* свет от 0 до 8.
- **Нарисуйте Halo** - Нарисуйте ореол вокруг света.
- **Flare** - Позволяет назначить световой эффект вспышки.
- **Режим рендеринга** - Авто, Важно, Не важно.
- **Culling Mask** - Позволяет выборочно освещать части сцены.

Направленный свет

Направленные огни испускают свет в одном направлении (подобно солнцу). Не имеет значения, где на сцене находится фактический объект GameObject, поскольку свет «повсюду». Интенсивность света не уменьшается, как и другие типы света.

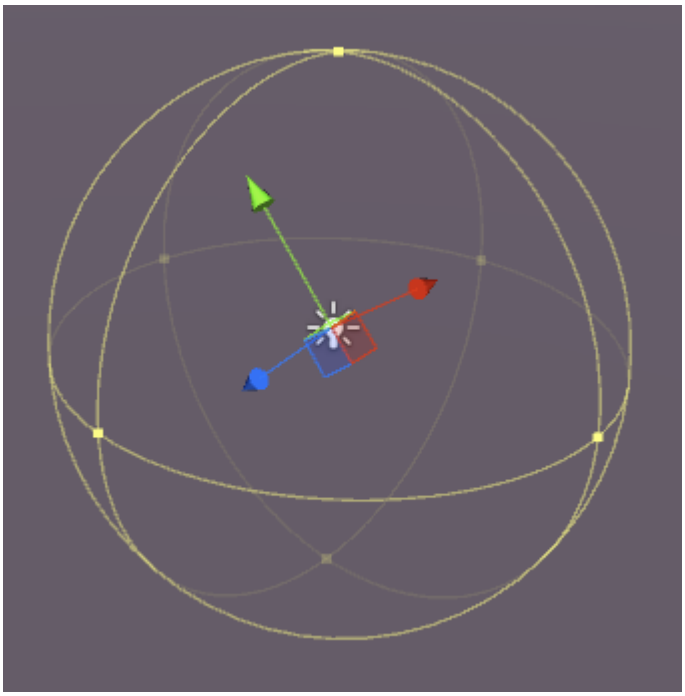


Направленный свет обладает следующими свойствами:

- **Выпечка** - в реальном времени, запеченная или смешанная.
- **Цвет.** Назначьте цвет света.
- **Интенсивность** - насколько силен свет от 0 до 8.
- **Интенсивность отскока** - насколько мощный *косвенный* свет от 0 до 8.
- **Тип тени** - без теней, жестких теней или мягких теней.
- **Cookie** - Позволяет назначать куки-файл для освещения.
- **Размер** файла cookie - размер назначенного файла cookie.
- **Нарисуйте Halo** - Нарисуйте ореол вокруг света.
- **Flare** - Позволяет назначить световой эффект вспышки.
- **Режим рендеринга** - Авто, Важно, Не важно.
- **Culling Mask** - Позволяет выборочно освещать части сцены.

Точечный свет

Точечный свет излучает свет из точки в пространстве во всех направлениях. Чем дальше от точки происхождения, тем интенсивнее свет.

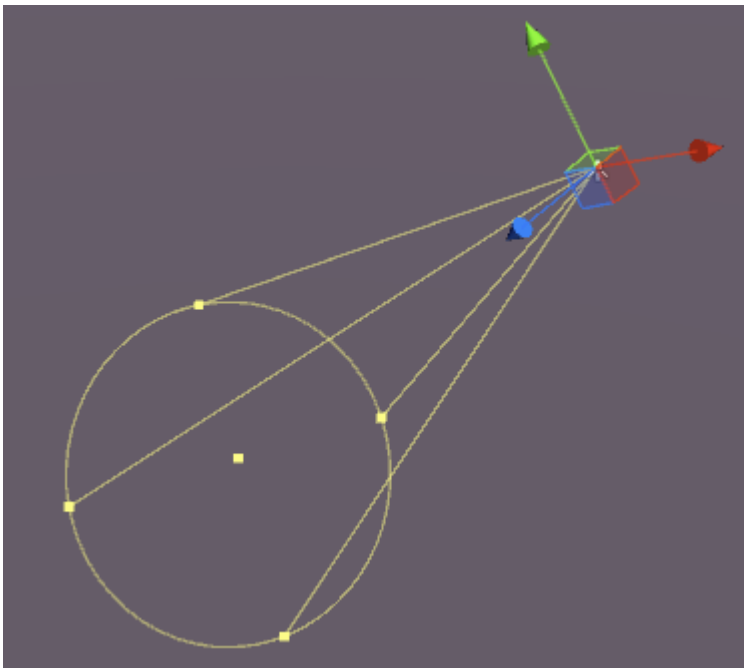


Точечные светильники обладают следующими свойствами:

- **Выпечка** - в реальном времени, запеченная или смешанная.
- **Диапазон** - расстояние от точки, где свет больше не достигает.
- **Цвет.** Назначьте цвет света.
- **Интенсивность** - насколько силен свет от 0 до 8.
- **Интенсивность отскока** - насколько мощный *косвенный* свет от 0 до 8.
- **Тип тени** - без теней, жестких теней или мягких теней.
- **Cookie** - Позволяет назначать куки-файл для освещения.
- **Нарисуйте Halo** - Нарисуйте ореол вокруг света.
- **Flare** - Позволяет назначить световой эффект вспышки.
- **Режим рендеринга** - Авто, Важно, Не важно.
- **Culling Mask** - Позволяет выборочно освещать части сцены.

Прожектор

Точечный свет очень похож на точечный свет, но излучение ограничено углом. Результатом является «конус» света, полезный для фар автомобилей или прожекторов.



Spot Lights обладают следующими свойствами:

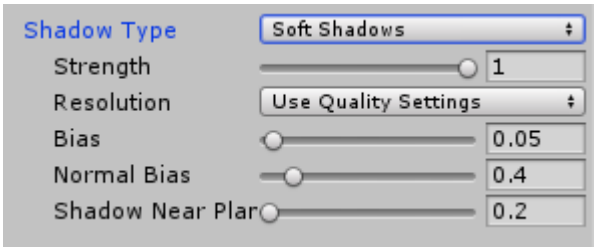
- **Выпечка** - в реальном времени, запеченная или смешанная.
- **Диапазон** - расстояние от точки, где свет больше не достигает.
- **Угол пятна** - угол излучения света.
- **Цвет.** Назначьте цвет света.
- **Интенсивность** - насколько силен свет от 0 до 8.
- **Интенсивность отскока** - насколько мощный *косвенный* свет от 0 до 8.
- **Тип тени** - без теней, жестких теней или мягких теней.
- **Cookie** - Позволяет назначать куки-файл для освещения.
- **Нарисуйте Halo** - Нарисуйте ореол вокруг света.
- **Flare** - Позволяет назначить световой эффект вспышки.
- **Режим рендеринга** - Авто, Важно, Не важно.
- **Culling Mask** - Позволяет выборочно освещать части сцены.

Примечание о тенях

Если вы выбираете Hard или Soft Shadows, в инспекторе становятся доступны следующие опции:

- **Сила** - насколько темны тени от 0 до 1.
- **Резолюция** - Как подробные тени.
- **Предвзятость** - степень, в которой теневые поверхности литья отталкиваются от света.
- **Нормальное смещение** . Степень, в которой поверхности теневого литья вставляются внутрь вдоль их нормалей.

- **Тень около плоскости** - 0,1 - 10.



излучение

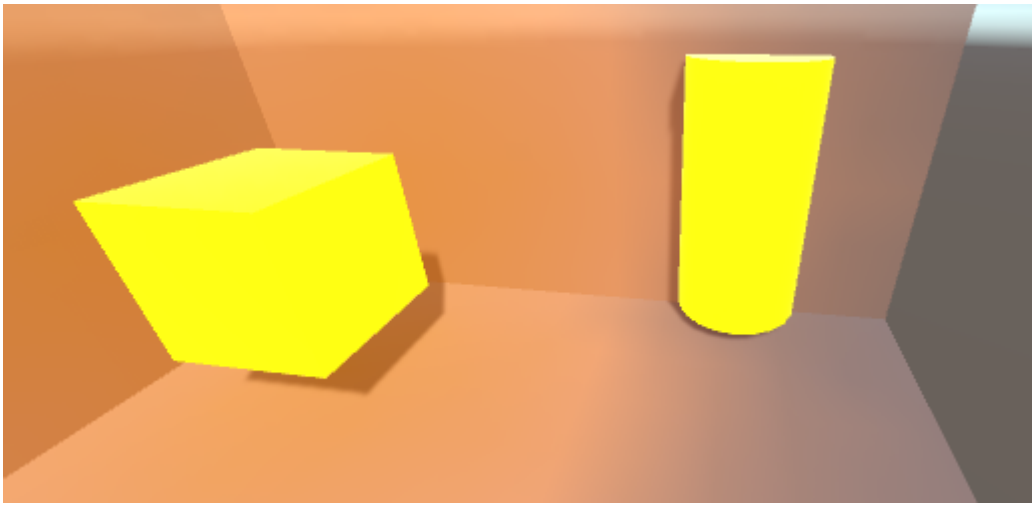
Выброс - это когда поверхность (или, скорее, материал) излучает свет. В панели инспектора для материала на статическом объекте с использованием стандартного шейдера существует свойство эмиссии:

Если вы измените это свойство на значение, превышающее значение по умолчанию 0, вы можете установить цвет излучения или присвоить **карту излучения** материалу. Любая текстура, назначенная этому слоту, позволит эмиссии использовать свои собственные цвета.

Существует также опция Global Illumination, которая позволяет вам установить, испускается ли выброс на соседние статические объекты или нет:

- **Запеченный** - Эмиссия будет запекаться в сцене
- **Realtime** - Эмиссия будет влиять на динамические объекты
- **Нет** . Эмиссия не влияет на соседние объекты

Если объект *не* настроен на статический, эффект все равно заставит объект «светиться», но свет не будет излучаться. Куб здесь статический, цилиндр не является:



Вы можете установить цвет излучения в коде следующим образом:

```
Renderer renderer = GetComponent<Renderer>();
Material mat = renderer.material;
mat.SetColor("_EmissionColor", Color.yellow);
```

Излучаемый свет упадет с квадратичной скоростью и будет демонстрироваться только против статических материалов в сцене.

Прочитайте Освещение единства онлайн: https://riptutorial.com/ru/unity3d/topic/7884/освещение-единства

глава 27: Плагины для Android 101 - Введение

Вступление

Этот раздел является первой частью серии о том, как создавать Android-модули для Unity. Начните здесь, если у вас мало опыта создания плагинов и / или ОС Android.

замечания

В этой серии я широко использую внешние ссылки, которые я рекомендую вам прочитать. В то время как перефразируемые версии соответствующего контента будут включены здесь, могут быть случаи, когда дополнительное чтение поможет.

Начиная с Android-плагинов

В настоящее время Unity предоставляет два способа вызова собственного кода Android.

1. Запишите собственный Android-код в Java и вызовите эти функции Java, используя C #
2. Напишите код C # для прямого вызова функций, входящих в ОС Android

Для взаимодействия с собственным кодом Unity предоставляет некоторые классы и функции.

- AndroidJavaObject - это базовый класс, который Unity обеспечивает взаимодействие с собственным кодом. Почти любой объект, возвращаемый из собственного кода, может быть сохранен как и AndroidJavaObject
- AndroidJavaClass - Наследует от AndroidJavaObject. Это используется для сравнения классов в вашем собственном коде
- Получить / Установить значения экземпляра собственного объекта и статические версии GetStatic / SetStatic
- Call / CallStatic для вызова собственных нестатических и статических функций

Схема создания плагина и терминологии

1. Записать собственный Java-код в Android Studio
2. Экспортируйте код в файл JAR / AAR (шаги здесь для файлов JAR и файлов AAR)

3. Скопируйте файл JAR / AAR в проект Unity в **Assets / Plugins / Android**
4. Напишите код в Unity (C # всегда был для этого), чтобы вызвать функции в плагине

Обратите внимание, что первые три шага применяются ТОЛЬКО, если вы хотите иметь собственный плагин!

Отсюда я буду ссылаться на файл JAR / AAR как на **родной плагин** , а на C # - на **C #**

Выбор между методами создания плагинов

Сразу видно, что первый способ создания плагинов длинный, поэтому выбор маршрута кажется спорным. Однако метод 1 является единственным способом вызова пользовательского кода. Итак, как выбрать?

Проще говоря, ваш плагин

1. Привлечение пользовательского кода - Выберите метод 1
2. Вызывать только собственные функции Android? - Выберите метод 2

Пожалуйста, **не** пытайтесь «смешивать» (т. Е. Часть плагина, используя метод 1, а другой с использованием метода 2) два метода! Хотя это вполне возможно, часто непрактично и болезненно управлять.

Examples

UnityAndroidPlugin.cs

Создайте новый скрипт C # в Unity и замените его содержимым следующим

```
using UnityEngine;
using System.Collections;

public static class UnityAndroidPlugin {

}
```

UnityAndroidNative.java

Создайте новый класс Java в Android Studio и замените его следующим

```
package com.axs.unityandroidplugin;
import android.util.Log;
import android.widget.Toast;
import android.app.ActivityManager;
import android.content.Context;
```

```
public class UnityAndroidNative {


}
```

UnityAndroidPluginGUI.cs

Создайте новый скрипт C # в Unity и вставьте это содержимое

```
using UnityEngine;
using System.Collections;

public class UnityAndroidPluginGUI : MonoBehaviour {

    void OnGUI () {

    }

}
```

Прочитайте Плагины для Android 101 - Введение онлайн:
https://riptutorial.com/ru/unity3d/topic/10032/плагины-для-android-101---введение

глава 28: Поиск и сбор GameObjects

Синтаксис

- public static GameObject Find (имя строки);
- public static GameObject FindGameObjectWithTag (строковый тег);
- public static GameObject [] FindGameObjectsWithTag (строковый тег);
- public static Object FindObjectOfType (тип типа);
- public static Object [] FindObjectsOfType (Тип типа);

замечания

Какой метод использовать

Будьте осторожны при поиске GameObjects во время выполнения, поскольку это может быть ресурсоемким. В частности: не запускайте FindObjectOfType или найдите в Update, FixedUpdate или, в более общем плане, в методе, называемом одним или несколькими моментами на кадр.

- Вызовите методы выполнения `FindObjectOfType` и `Find` только в случае необходимости
- `FindGameObjectWithTag` имеет очень хорошую производительность по сравнению с другими методами на основе строк. Unity хранит отдельные вкладки на помеченных объектах и запрашивает их вместо всей сцены.
- Для «статических» GameObjects (таких как элементы пользовательского интерфейса и сборные файлы), созданных в редакторе, используйте сериализуемую ссылку GameObject в редакторе
- Храните свои списки GameObjects в списке или массивах, которыми вы управляете сами
- В общем случае, если вы создаете экземпляр множества GameObjects того же типа, взгляните на Object Pooling
- Кэш ваших результатов поиска, чтобы избежать дорогостоящих методов поиска снова и снова.

Идти глубже

Помимо методов, которые приходят с Unity, относительно легко разработать собственные методы поиска и сбора.

- В случае `FindObjectsOfType()` вас могут быть ваши скрипты, которые хранят список в `static` коллекции. Гораздо быстрее перебирать готовый список объектов, чем искать и проверять объекты со сцены.

- Или создайте скрипт, который хранит их экземпляры в `Dictionary` основе строк, и у вас есть простая система тегов, которую вы можете расширить.

Examples

Поиск по названию GameObject

```
var go = GameObject.Find("NameOfTheObject");
```

Pros	Cons
Легко использовать	Производительность ухудшается по количеству игровых объектов в сцене
	Строки - это *слабые ссылки* и подозрения на ошибки пользователя

Поиск по тегам GameObject

```
var go = GameObject.FindGameObjectWithTag("Player");
```

Pros	Cons
Возможность поиска как отдельных объектов, так и целых групп	Строки - это слабые ссылки и подозрения на ошибки пользователя.
Относительно быстрый и эффективный	Код не переносится, поскольку теги жестко закодированы в сценариях.

Вставка в сценарии в режиме редактирования

```
[SerializeField]
GameObject[] gameObjects;
```

Pros	Cons
Отличное выступление	Коллекция объектов статическая
Портативный код	Может ссылаться только на GameObjects с той же сцены

Поиск GameObjects с помощью скриптов MonoBehaviour

```
ExampleScript script = GameObject.FindObjectOfType<ExampleScript>();
```

```
GameObject go = script.gameObject;
```

`FindObjectOfType()` возвращает `null` если ни один не найден.

Pros	Cons
Сильно напечатан	Производительность ухудшается по количеству игровых объектов, необходимых для оценки
Возможность поиска как отдельных объектов, так и целых групп	

Найти GameObjects по имени из дочерних объектов

```
Transform tr = GetComponent<Transform>().Find("NameOfTheObject");
GameObject go = tr.gameObject;
```

`Find` возвращает `null` если ни один не найден

Pros	Cons
Ограниченная, четко определенная область поиска	Строки - слабые ссылки

Прочитайте Поиск и сбор GameObjects онлайн: https://riptutorial.com/ru/unity3d/topic/3793/поиск-и-сбор-gameobjects

глава 29: Пользовательский интерфейс (UI)

Examples

Подписка на событие в коде

По умолчанию следует подписываться на событие с помощью инспектора, но иногда лучше делать это в коде. В этом примере мы подписываемся на событие щелчка кнопки для его обработки.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Button))]
public class AutomaticClickHandler : MonoBehaviour
{
    private void Awake()
    {
        var button = this.GetComponent<Button>();
        button.onClick.AddListener(HandleClick);
    }

    private void HandleClick()
    {
        Debug.Log("AutomaticClickHandler.HandleClick()", this);
    }
}
```

Компоненты пользовательского интерфейса обычно предоставляют своим основным слушателям легко:

- Кнопка: onClick
- Выпадающее меню : onValueChanged
- InputField: onEndEdit , onValidateInput , onValueChanged
- Полоса прокрутки: onValueChanged
- ScrollRect: onValueChanged
- Ползунок: onValueChanged
- Переключить: onValueChanged

Добавление прослушивателей мыши

Иногда вы хотите добавить слушателей к конкретным событиям, которые не предусмотрены компонентами, в частности событиями мыши. Для этого вам придется добавить их самостоятельно, используя компонент `EventTrigger` :

```
using UnityEngine;
using UnityEngine.EventSystems;
```

```
[RequireComponent(typeof( EventTrigger ))]
public class CustomListenersExample : MonoBehaviour
{
    void Start( )
    {
        EventTrigger eventTrigger = GetComponent<EventTrigger>( );
        EventTrigger.Entry entry = new EventTrigger.Entry( );
        entry.eventID = EventTriggerType.PointerDown;
        entry.callback.AddListener( ( data ) => { OnPointerDownDelegate(
(PointerEventData)data ); } );
        eventTrigger.triggers.Add( entry );
    }

    public void OnPointerDownDelegate( PointerEventData data )
    {
        Debug.Log( "OnPointerDownDelegate called." );
    }
}
```

Возможны различные eventID:

- PointerEnter
- PointerExit
- PointerDown
- PointerUp
- PointerClick
- Тащить, тянуть
- Капля
- манускрипт
- UpdateSelected
- Выбрать
- Отмените
- Переехать
- InitializePotentialDrag
- BeginDrag
- EndDrag
- Отправить
- отменить

Прочитайте Пользовательский интерфейс (UI) онлайн:
https://riptutorial.com/ru/unity3d/topic/2296/пользовательский-интерфейс--ui-

глава 30: Расширение редактора

Синтаксис

- [MenuItem (string itemName)]
- [MenuItem (string itemName, bool isValidateFunction)]
- [MenuItem (string itemName, bool isValidateFunction, int priority)]
- [ContextMenu (имя строки)]
- [ContextMenuItem (имя строки, строковая функция)]
- [DrawGizmo (GizmoType gizmo)]
- [DrawGizmo (GizmoType gizmo, Тип drawGizmoType)]

параметры

параметр	подробности
MenuCommand	MenuCommand используется для извлечения контекста для MenuItem
MenuCommand.context	Объект, являющийся целью команды меню
MenuCommand.userData	Int для передачи пользовательской информации в пункт меню

Examples

Пользовательский инспектор

Использование пользовательского инспектора позволяет изменить способ рисования сценария в инспекторе. Иногда вы хотите добавить дополнительную информацию в инспекторе для своего скрипта, что невозможно сделать с помощью специализированного ящика свойств.

Ниже приведен простой пример пользовательского объекта, который с помощью пользовательского инспектора может отображать более полезную информацию.

```
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

public class InspectorExample : MonoBehaviour {

    public int Level;
```

```
    public float BaseDamage;

    public float DamageBonus {
        get {
            return Level / 100f * 50;
        }
    }

    public float ActualDamage {
        get {
            return BaseDamage + DamageBonus;
        }
    }
}

#if UNITY_EDITOR
[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {

    public override void OnInspectorGUI() {
        base.OnInspectorGUI();

        var ie = (InspectorExample)target;

        EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
        EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
    }
}
#endif
```

Сначала мы определяем наше поведение с некоторыми полями

```
public class InspectorExample : MonoBehaviour {
    public int Level;
    public float BaseDamage;
}
```

Поля, показанные выше, автоматически рисуются (без специального инспектора), когда вы просматриваете скрипт в окне инспектора.

```
public float DamageBonus {
    get {
        return Level / 100f * 50;
    }
}

public float ActualDamage {
    get {
        return BaseDamage + DamageBonus;
    }
}
```

Эти свойства автоматически не создаются Unity. Чтобы показать эти свойства в представлении «Инспектор», мы должны использовать наш пользовательский инспектор.

Сначала мы должны определить наш пользовательский инспектор, как это

```
[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {
```

Пользовательский инспектор должен получить от *редактора* и нуждается в *атрибуте CustomEditor* . Параметр атрибута - это тип объекта, для которого должен использоваться пользовательский инспектор.

Далее следует метод OnInspectorGUI. Этот метод вызывается, когда скрипт отображается в окне инспектора.

```
public override void OnInspectorGUI() {
    base.OnInspectorGUI();
}
```

Мы вызываем *base.OnInspectorGUI (),* чтобы позволить Unity обрабатывать другие поля, которые находятся в скрипте. Если бы мы не назвали это, нам пришлось бы делать больше работы самостоятельно.

Ниже приведены наши пользовательские свойства, которые мы хотим показать

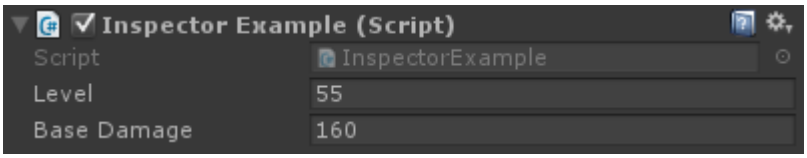
```
var ie = (InspectorExample)target;

EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
```

Мы должны создать временную переменную, которая содержит цель, выбранную для нашего пользовательского типа (цель доступна, потому что мы получаем редактор).

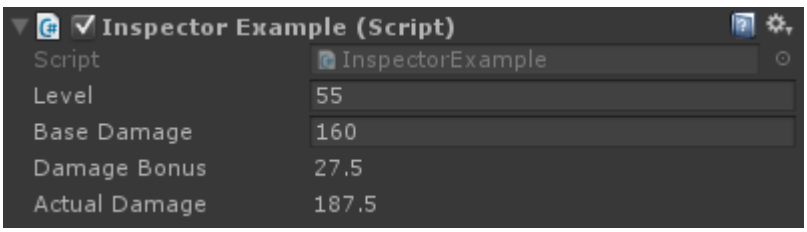
Затем мы можем решить, как нарисовать наши свойства, в этом случае достаточно двух лабораторных окон, так как мы просто хотим показать значения и не в состоянии их редактировать.

Результат

До



После



Пользовательский ящик свойств

Иногда у вас есть пользовательские объекты, которые содержат данные, но не выводятся из MonoBehaviour. Добавление этих объектов в качестве поля в классе, который является MonoBehaviour, не будет иметь визуального эффекта, если вы не напишите свой собственный пользовательский ящик свойств для типа объекта.

Ниже приведен простой пример пользовательского объекта, добавленного в MonoBehaviour, и пользовательский ящик свойств для настраиваемого объекта.

```
public enum Gender {
    Male,
    Female,
    Other
}

// Needs the Serializable attribute otherwise the CustomPropertyDrawer wont be used
[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

// The class that you can attach to a GameObject
public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UInfo;
}

[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {

    public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
        // The 6 comes from extra spacing between the fields (2px each)
        return EditorGUIUtility.singleLineHeight * 4 + 6;
    }

    public override void OnGUI( Rect position, SerializedProperty property, GUIContent label )
{
        EditorGUI.BeginProperty( position, label, property );

        EditorGUI.LabelField( position, label );

        var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
        var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
        var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );

        EditorGUI.indentLevel++;

        EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
        EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
        EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

        EditorGUI.indentLevel--;

        EditorGUI.EndProperty();
    }
```

```
}
```

Сначала мы определяем пользовательский объект со всеми его требованиями. Просто простой класс, описывающий пользователя. Этот класс используется в нашем классе PropertyDrawerExample, который мы можем добавить в GameObject.

```
public enum Gender {
    Male,
    Female,
    Other
}

[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UInfo;
}
```

> Пользовательский класс нуждается в атрибуте Serializable, иначе CustomPropertyDrawer не будет использоваться

Далее следует CustomPropertyDrawer

Сначала мы должны определить класс, который происходит из PropertyDrawer. Для определения класса также требуется атрибут CustomPropertyDrawer. Прошедший параметр - это тип объекта, для которого вы хотите использовать этот ящик.

```
[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {
```

Затем мы переопределим функцию GetPropertyHeight. Это позволяет нам определить пользовательскую высоту для нашего свойства. В этом случае мы знаем, что у нашего имущества будет четыре части: ярлык, имя, возраст и пол. Поэтому мы используем *EditorGUIUtility.singleLineHeight * 4* , добавляем еще 6 пикселей, потому что мы хотим разместить каждое поле с двумя пикселями между ними.

```
public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
    return EditorGUIUtility.singleLineHeight * 4 + 6;
}
```

Далее - это метод OnGUI. Мы начинаем с *EditorGUI.BeginProperty ([...])* и заканчиваем функцию *EditorGUI.EndProperty ()* . Мы делаем это так, чтобы, если бы это свойство было частью сборника, фактическая предикатная логика prefab будет работать для всего между этими двумя методами.

```
public override void OnGUI( Rect position, SerializedProperty property, GUIContent label ) {
    EditorGUI.BeginProperty( position, label, property );
```

После этого мы покажем метку, содержащую имя поля, и мы уже определили прямоугольники для наших полей.

```
EditorGUI.LabelField( position, label );

var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );
```

> Каждое поле разнесено на 16 + 2 пикселя, а высота равна 16 (это то же самое, что и EditorGUIUtility.singleLineHeight)

Затем мы отступаем от пользовательского интерфейса с одной вкладкой для более красивого макета, отображаем свойства, отступаем от графического интерфейса и заканчиваем *EditorGUI.EndProperty* .

```
EditorGUI.indentLevel++;

EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

EditorGUI.indentLevel--;

EditorGUI.EndProperty();
```

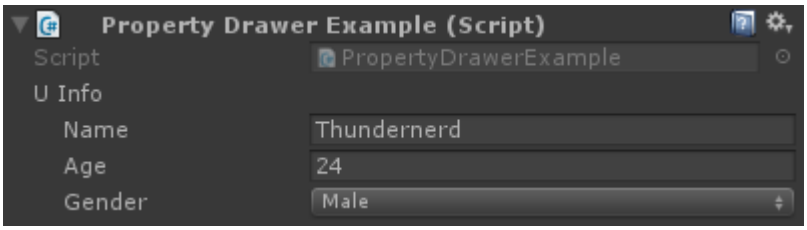
Мы показываем поля с помощью *EditorGUI.PropertyField, для* которого требуется прямоугольник для позиции и SerializedProperty для отображаемого свойства. Мы приобретаем свойство, вызывая *FindPropertyRelative ("...")* для свойства, переданного в функции *OnGUI* . Обратите внимание, что это чувствительные к регистру и не публичные свойства не могут быть найдены!

> В этом примере я не сохраняю свойства return from property.FindPropertyRelative ("..."). Вы должны сохранить их в закрытых полях в классе, чтобы предотвратить ненужные вызовы

Результат

До

После



Пункты меню

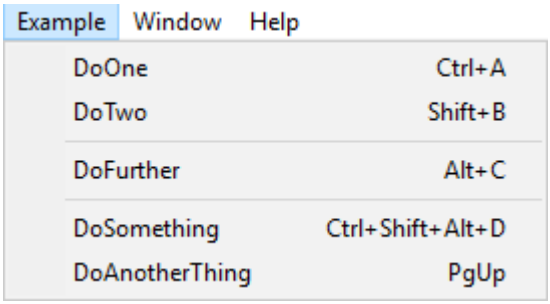
Элементы меню - отличный способ добавить пользовательские действия в редактор. Вы можете добавлять элементы меню в панель меню, использовать их как контекстные клики для определенных компонентов или даже в контекстных кликах по полям ваших сценариев.

Ниже приведен пример того, как вы можете применять пункты меню.

```
public class MenuItemsExample : MonoBehaviour {

    [MenuItem( "Example/DoSomething %#&d" )]
    private static void DoSomething() {
        // Execute some code
    }

    [MenuItem( "Example/DoAnotherThing", true )]
    private static bool DoAnotherThingValidator() {
        return Selection.gameObjects.Length > 0;
    }

    [MenuItem( "Example/DoAnotherThing _PGUP", false )]
    private static void DoAnotherThing() {
        // Execute some code
    }

    [MenuItem( "Example/DoOne %a", false, 1 )]
    private static void DoOne() {
        // Execute some code
    }

    [MenuItem( "Example/DoTwo #b", false, 2 )]
    private static void DoTwo() {
        // Execute some code
    }

    [MenuItem( "Example/DoFurther &c", false, 13 )]
    private static void DoFurther() {
        // Execute some code
    }

    [MenuItem( "CONTEXT/Camera/DoCameraThing" )]
    private static void DoCameraThing( MenuCommand cmd ) {
        // Execute some code
    }

    [ContextMenu( "ContextSomething" )]
    private void ContentSomething() {
        // Execute some code
    }
```

```
    [ContextMenuItem( "Reset", "ResetDate" )]
    [ContextMenuItem( "Set to Now", "SetDateToNow" )]
    public string Date = "";

    public void ResetDate() {
        Date = "";
    }

    public void SetDateToNow() {
        Date = DateTime.Now.ToString();
    }
}
```

Что выглядит так



Перейдем к основному пункту меню. Как вы можете видеть ниже, вам нужно определить статическую функцию с атрибутом *MenuItem* , который вы передаете в качестве заголовка для элемента меню. Вы можете поместить свой пункт меню на несколько уровней, добавив a / в имя.

```
[MenuItem( "Example/DoSomething %#&d" )]
private static void DoSomething() {
    // Execute some code
}
```

> У вас не может быть пункта меню на верхнем уровне. Ваши пункты меню должны находиться в подменю!

Специальные символы в конце имени MenuItem предназначены для сочетаний клавиш, это не является обязательным требованием.

Существуют специальные символы, которые вы можете использовать для ваших сочетаний клавиш:

- % - Ctrl на Windows, Cmd на OS X
- # - Сдвиг
- & - Alt

Это означает, что ярлык *% # & d* означает ctrl + shift + alt + D в Windows и cmd + shift + alt + D в OS X.

Если вы хотите использовать ярлык без каких-либо специальных клавиш, так, например,

только клавишу «D», вы можете добавить символ _ (подчеркивание) к клавише быстрого доступа, которую вы хотите использовать.

Существуют и другие специальные клавиши, которые поддерживаются:

- LEFT, RIGHT, UP, DOWN - для клавиш со стрелками
- F1..F12 - для функциональных клавиш
- HOME, END, PGUP, PGDN - для клавиш навигации

  Клавиши быстрого вызова должны быть отделены от любого другого текста пространством

Далее перечислены пункты меню валидатора. Элементы меню «Валидатор» позволяют отключать пункты меню (недоступно, не кликается), когда условие не выполняется. Примером этого может быть то, что ваш пункт меню действует на текущий выбор GameObjects, который вы можете проверить в элементе меню проверки.

```
[MenuItem( "Example/DoAnotherThing", true )]
private static bool DoAnotherThingValidator() {
    return Selection.gameObjects.Length > 0;
}

[MenuItem( "Example/DoAnotherThing _PGUP", false )]
private static void DoAnotherThing() {
    // Execute some code
}
```

Для работы элемента валидатора вам необходимо создать две статические функции, как с атрибутом MenuItem, так и с одним и тем же именем (клавиша быстрого доступа не имеет значения). Разница между ними заключается в том, что вы отмечаете их как функцию валидатора или нет, передавая логический параметр.

Вы также можете определить порядок пунктов меню, добавив приоритет. Приоритет определяется целым числом, которое вы передаете в качестве третьего параметра. Чем меньше число, тем выше число в списке, тем больше число ниже в списке. Вы можете добавить разделитель между двумя пунктами меню, убедившись, что между приоритетом пунктов меню находится не менее 10 цифр.

```
[MenuItem( "Example/DoOne %a", false, 1 )]
private static void DoOne() {
    // Execute some code
}

[MenuItem( "Example/DoTwo #b", false, 2 )]
private static void DoTwo() {
    // Execute some code
}

[MenuItem( "Example/DoFurther &c", false, 13 )]
private static void DoFurther() {
    // Execute some code
```

```
}
```

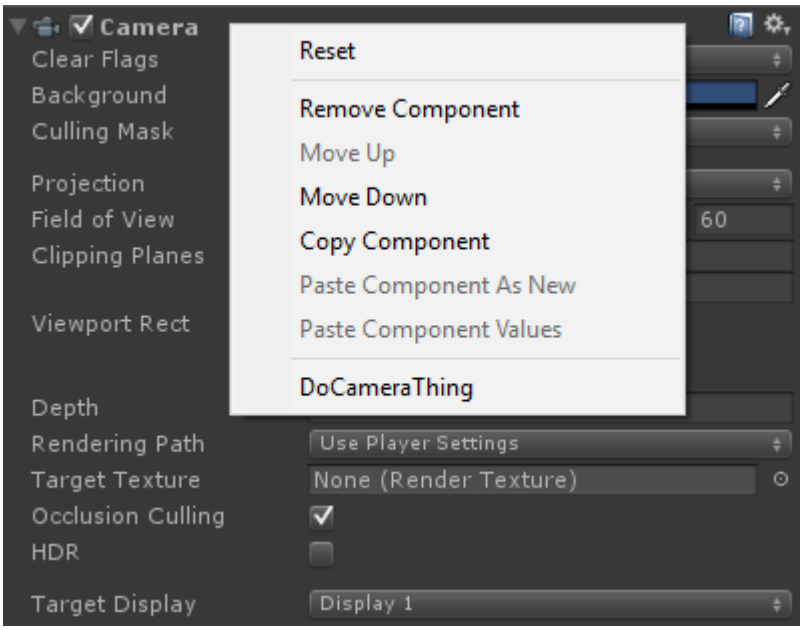
> Если у вас есть список меню, в котором есть комбинация приоритетных и неприоритетных позиций, то приоритет не будет выделен из приоритетных позиций.

Далее добавляется пункт меню в контекстное меню уже существующего компонента. Вы должны запустить имя MenuItem с помощью CONTEXT (с учетом регистра), и ваша функция принимает параметр MenuCommand.

Следующий фрагмент добавит элемент контекстного меню в компонент «Камера».

```
[MenuItem( "CONTEXT/Camera/DoCameraThing" )]
private static void DoCameraThing( MenuCommand cmd ) {
    // Execute some code
}
```

Что выглядит так

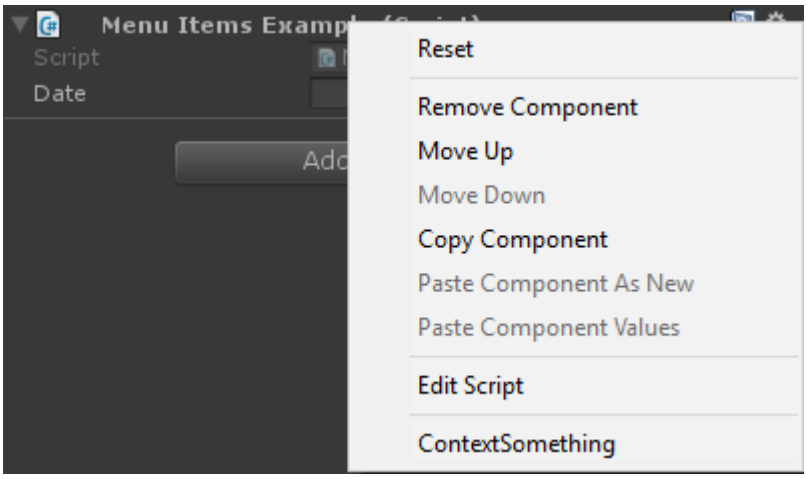


Параметр MenuCommand предоставляет вам доступ к значению компонента и любым пользовательским данным, которые отправляются с ним.

Вы также можете добавить элемент контекстного меню к своим собственным компонентам, используя атрибут ContextMenu. Этот атрибут принимает только имя, валидность или приоритет и должен быть частью нестатического метода.

```
[ContextMenu( "ContextSomething" )]
private void ContentSomething() {
    // Execute some code
}
```

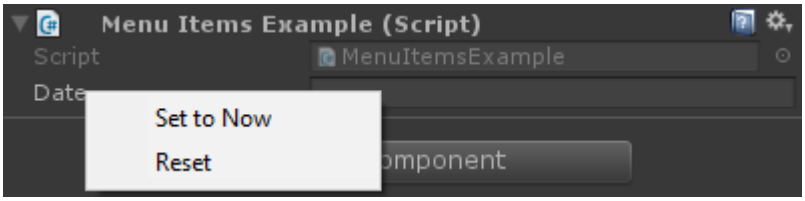
Что выглядит так



Вы также можете добавлять элементы контекстного меню в поля вашего собственного компонента. Эти пункты меню появятся при контекстном щелчке по полю, к которому они принадлежат, и могут выполнять методы, которые вы определили в этом компоненте. Таким образом вы можете добавить, например, значения по умолчанию или текущую дату, как показано ниже.

```
[ContextMenuItem( "Reset", "ResetDate" )]
[ContextMenuItem( "Set to Now", "SetDateToNow" )]
public string Date = "";

public void ResetDate() {
    Date = "";
}

public void SetDateToNow() {
    Date = DateTime.Now.ToString();
}
```

Что выглядит так



Gizmos

Gizmos используются для рисования фигур в сцене. Вы можете использовать эти фигуры для получения дополнительной информации о ваших GameObjects, например, усеченной косой или диапазона обнаружения.

Ниже приведены два примера, как это сделать.

Пример 1

В этом примере используются *методы OnDrawGizmos* и *OnDrawGizmosSelected* (магия).

```
public class GizmoExample : MonoBehaviour {

    public float GetDetectionRadius() {
        return 12.5f;
    }

    public float GetFOV() {
        return 25f;
    }

    public float GetMaxRange() {
        return 6.5f;
    }

    public float GetMinRange() {
        return 0;
    }

    public float GetAspect() {
        return 2.5f;
    }

    public void OnDrawGizmos() {
        var gizmoMatrix = Gizmos.matrix;
        var gizmoColor = Gizmos.color;

        Gizmos.matrix = Matrix4x4.TRS( transform.position, transform.rotation,
transform.lossyScale );
        Gizmos.color = Color.red;
        Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect()
);

        Gizmos.matrix = gizmoMatrix;
        Gizmos.color = gizmoColor;
    }

    public void OnDrawGizmosSelected() {
        Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
    }
}
```

В этом примере у нас есть два метода рисования гизмосов: тот, который рисует, когда объект активен (OnDrawGizmos), и один, когда объект выбран в иерархии (OnDrawGizmosSelected).

```
public void OnDrawGizmos() {
    var gizmoMatrix = Gizmos.matrix;
    var gizmoColor = Gizmos.color;

    Gizmos.matrix = Matrix4x4.TRS( transform.position, transform.rotation,
transform.lossyScale );
    Gizmos.color = Color.red;
    Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect() );

    Gizmos.matrix = gizmoMatrix;
    Gizmos.color = gizmoColor;
```

```
}
```

Сначала мы сохраняем матрицу и цвет gizmo, потому что мы собираемся изменить ее и хотим вернуть ее обратно, когда мы закончим, чтобы не повлиять на какой-либо другой рисунок гизмо.

Затем мы хотим нарисовать усечку, что наш объект, однако, нам нужно изменить матрицу Gizmos, чтобы она соответствовала положению, вращению и масштабу. Мы также устанавливаем цвет Gizmos на красный, чтобы подчеркнуть усечение. Когда это будет сделано, мы можем вызвать *Gizmos.DrawFrustum,* чтобы нарисовать *усечку* в сцене.

Когда мы закончим рисовать то, что хотим рисовать, мы возвращаем матрицу и цвет Gizmos.

```
public void OnDrawGizmosSelected() {
    Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
}
```

Мы также хотим нарисовать диапазон обнаружения, когда мы выберем наш GameObject. Это делается с помощью класса *Handles* , так как класс *вещицы* не имеет каких - либо методов для дисков.

Использование этой формы рисования gizmos приводит к результату, показанному ниже.

Пример два

В этом примере используется атрибут *DrawGizmo* .

```
public class GizmoDrawerExample {

    [DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]
    public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
        var gizmoMatrix = Gizmos.matrix;
        var gizmoColor = Gizmos.color;

        Gizmos.matrix = Matrix4x4.TRS( obj.transform.position, obj.transform.rotation,
obj.transform.lossyScale );
        Gizmos.color = Color.red;
        Gizmos.DrawFrustum( Vector3.zero, obj.GetFOV(), obj.GetMaxRange(), obj.GetMinRange(),
obj.GetAspect() );

        Gizmos.matrix = gizmoMatrix;
        Gizmos.color = gizmoColor;

        if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {
            Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius()
);
        }
    }
}
```

Этот способ позволяет отделить вызовы gizmo от вашего скрипта. Большинство из них использует тот же код, что и другой пример, за исключением двух вещей.

```
[DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]
public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
```

Вам нужно использовать атрибут DrawGizmo, который переводит перечисление GizmoType в качестве первого параметра и Type как второй параметр. Тип должен быть типом, который вы хотите использовать для рисования гизмо.

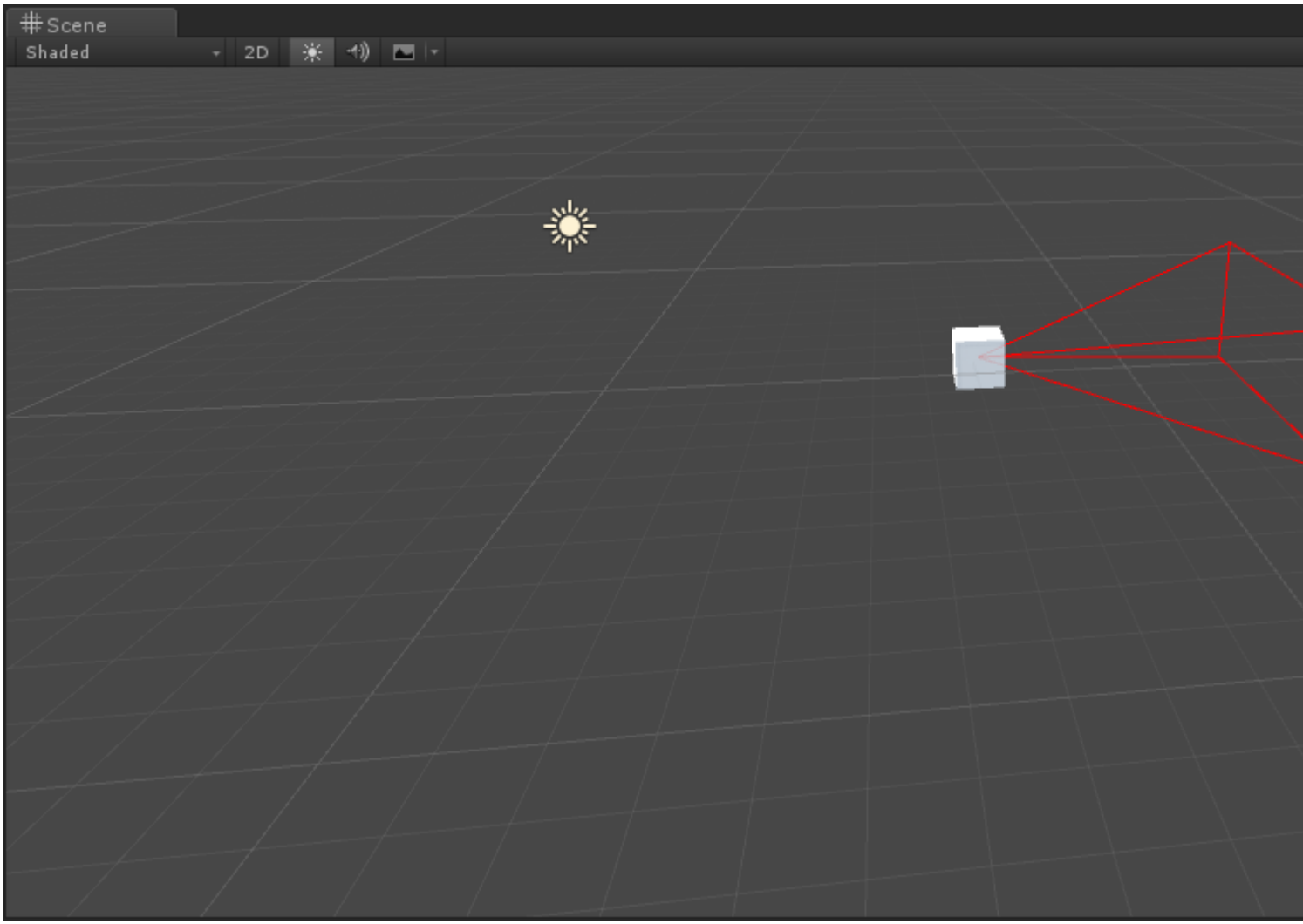
Метод рисования gizmo должен быть статичным, общедоступным или непубличным, и его можно назвать любым, что вы хотите. Первым параметром является тип, который должен соответствовать типу, переданному в качестве второго параметра в атрибуте, а вторым параметром является перечисление GizmoType, которое описывает текущее состояние вашего объекта.

```
if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {
    Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius() );
}
```

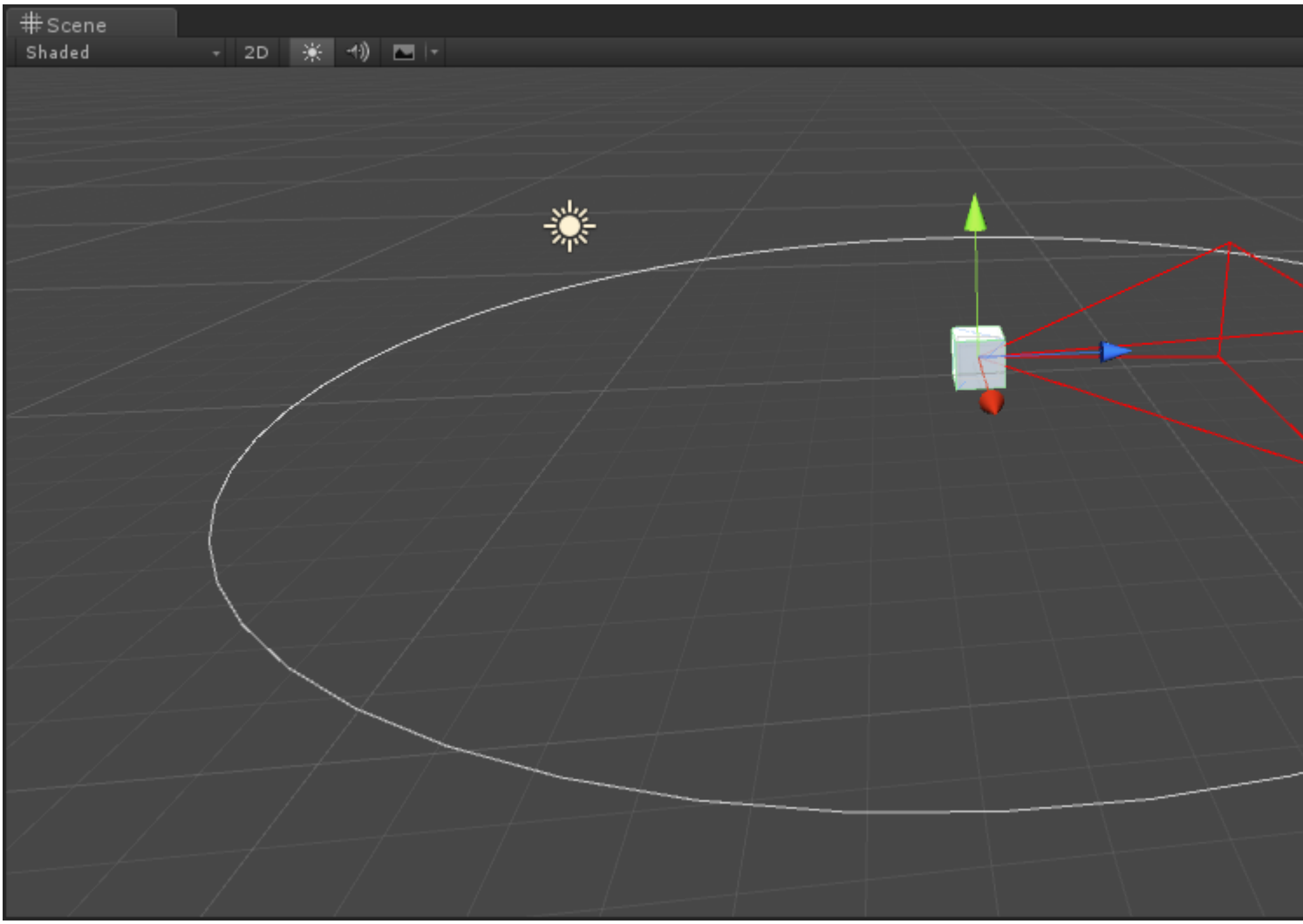
Другое отличие состоит в том, что для проверки того, что такое объект GizmoType, вам нужно выполнить AND и проверить параметр и тип, который вы хотите.

Результат

Не выбран



выбранный



Окно редактора

Почему окно редактора?

Как вы могли видеть, вы можете делать много вещей в обычном инспекторе (если вы не знаете, что такое пользовательский инспектор, посмотрите пример здесь: http://www.riptutorial.com/unity3d/topic/2506 / extension-the-editor . Но в какой-то момент вы можете захотеть реализовать панель конфигурации или настраиваемую палитру свойств . В этих случаях вы будете использовать редактор WindowWindow . Пользовательский интерфейс Unity состоит из редактора Windows, вы можете открыть их (обычно через верхний бар), вставлять их и т. д.

Создайте базовый редактор WindowWindow

Простой пример

Создание пользовательского окна редактора довольно просто. Все, что вам нужно сделать, это расширить класс EditorWindow и использовать методы Init () и OnGUI (). Вот

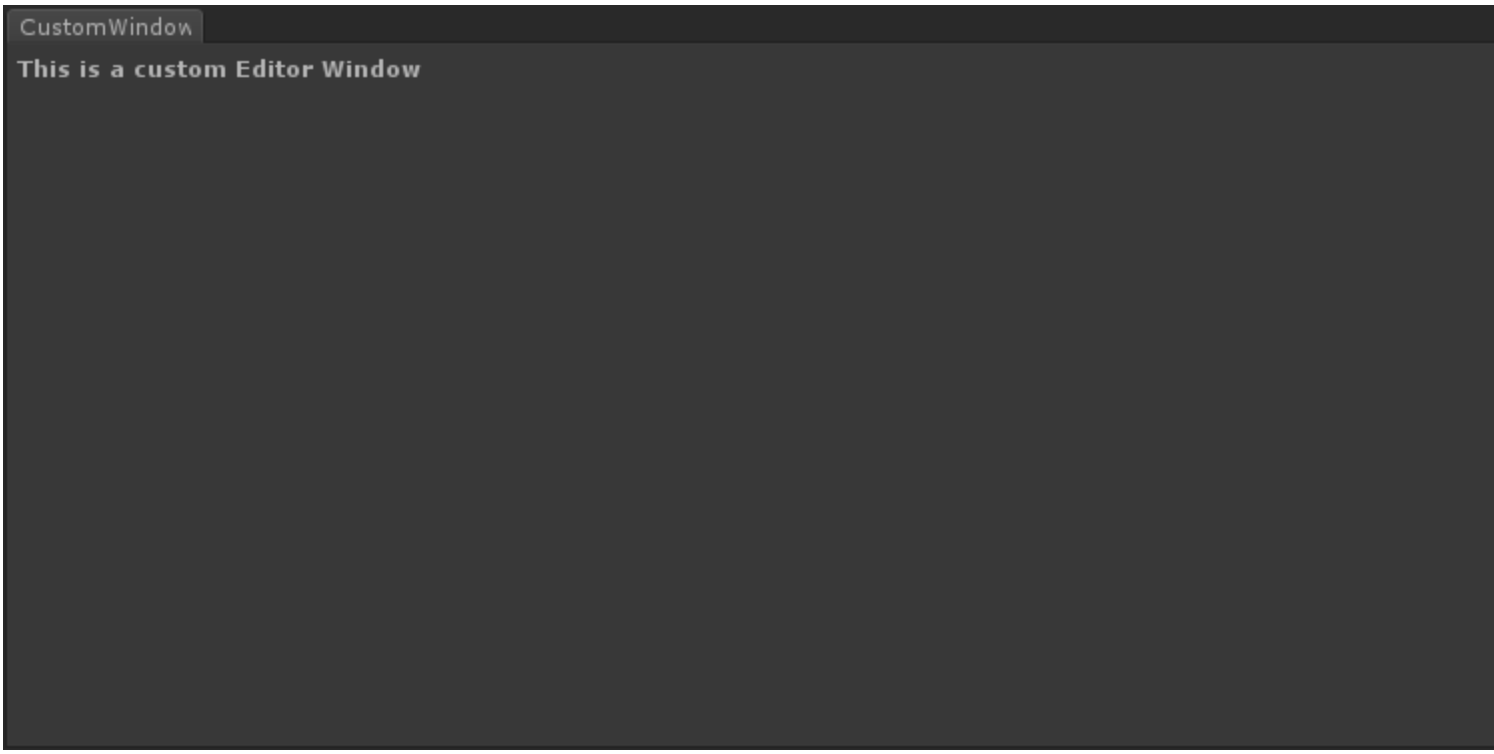
простой пример:

```
using UnityEngine;
using UnityEditor;

public class CustomWindow : EditorWindow
{
    // Add menu named "Custom Window" to the Window menu
    [MenuItem("Window/Custom Window")]
    static void Init()
    {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
        window.Show();
    }

    void OnGUI()
    {
        GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);
    }
}
```

3 важных момента:

1. Не забудьте расширить EditorWindow
2. Используйте Init (), как показано в примере. EditorWindow.GetWindow проверяет, уже создан ли CustomWindow. Если нет, он создаст новый экземпляр. Используя это, вы убедитесь, что у вас нет нескольких экземпляров вашего окна одновременно
3. Использовать OnGUI (), как обычно, для отображения информации в вашем окне

Окончательный результат будет выглядеть так:



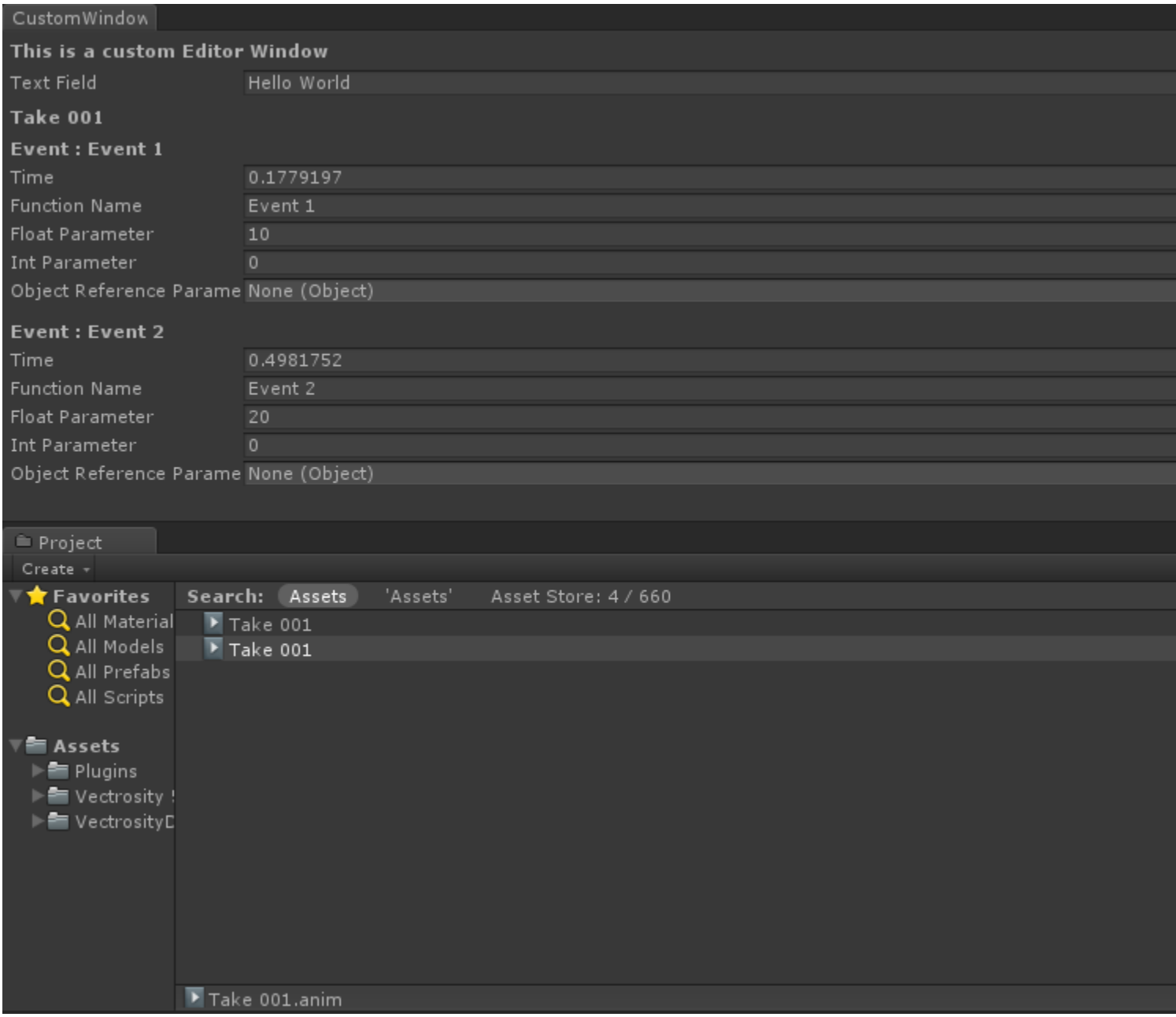
Идти глубже

Конечно, вы, вероятно, захотите управлять или модифицировать некоторые активы, используя этот редактор. Ниже приведен пример использования класса Selection (для получения активного выбора) и изменения выбранных свойств актива через SerializedObject и SerializedProperty .

```
    using System.Linq;
    using UnityEngine;
    using UnityEditor;

    public class CustomWindow : EditorWindow
    {
        private AnimationClip _animationClip;
        private SerializedObject _serializedClip;
        private SerializedProperty _events;

        private string _text = "Hello World";

        // Add menu named "Custom Window" to the Window menu
        [MenuItem("Window/Custom Window")]
        static void Init()
        {
            // Get existing open window or if none, make a new one:
            CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
            window.Show();
        }

        void OnGUI()
        {
            GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);

            // You can use EditorGUI, EditorGUILayout and GUILayout classes to display
anything you want
            // A TextField example
            _text = EditorGUILayout.TextField("Text Field", _text);

            // Note that you can modify an asset or a gameobject using an EditorWindow. Here
is a quick example with an AnimationClip asset
            // The _animationClip, _serializedClip and _events are set in OnSelectionChange()

            if (_animationClip == null || _serializedClip == null || _events == null) return;

            // We can modify our serializedClip like we would do in a Custom Inspector. For
example we can grab its events and display their information

            GUILayout.Label(_animationClip.name, EditorStyles.boldLabel);

            for (var i = 0; i < _events.arraySize; i++)
            {
                EditorGUILayout.BeginVertical();

                EditorGUILayout.LabelField(
                    "Event : " +
_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName").stringValue,
                    EditorStyles.boldLabel);

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("time"),
```

```
true,
                    GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName"),

                    true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("floatParameter"),

                    true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("intParameter"),

                    true, GUILayout.ExpandWidth(true));
                EditorGUILayout.PropertyField(

_events.GetArrayElementAtIndex(i).FindPropertyRelative("objectReferenceParameter"), true,
                    GUILayout.ExpandWidth(true));

                EditorGUILayout.Separator();
                EditorGUILayout.EndVertical();
            }

            // Of course we need to Apply the modified properties. We don't our changes won't
be saved
            _serializedClip.ApplyModifiedProperties();
        }

        /// This Message is triggered when the user selection in the editor changes. That's
when we should tell our Window to Repaint() if the user selected another AnimationClip
        private void OnSelectionChange()
        {
            _animationClip =
                Selection.GetFiltered(typeof(AnimationClip),
SelectionMode.Assets).FirstOrDefault() as AnimationClip;
            if (_animationClip == null) return;

            _serializedClip = new SerializedObject(_animationClip);
            _events = _serializedClip.FindProperty("m_Events");
            Repaint();
        }
    }
```

Вот результат:



Расширенные темы

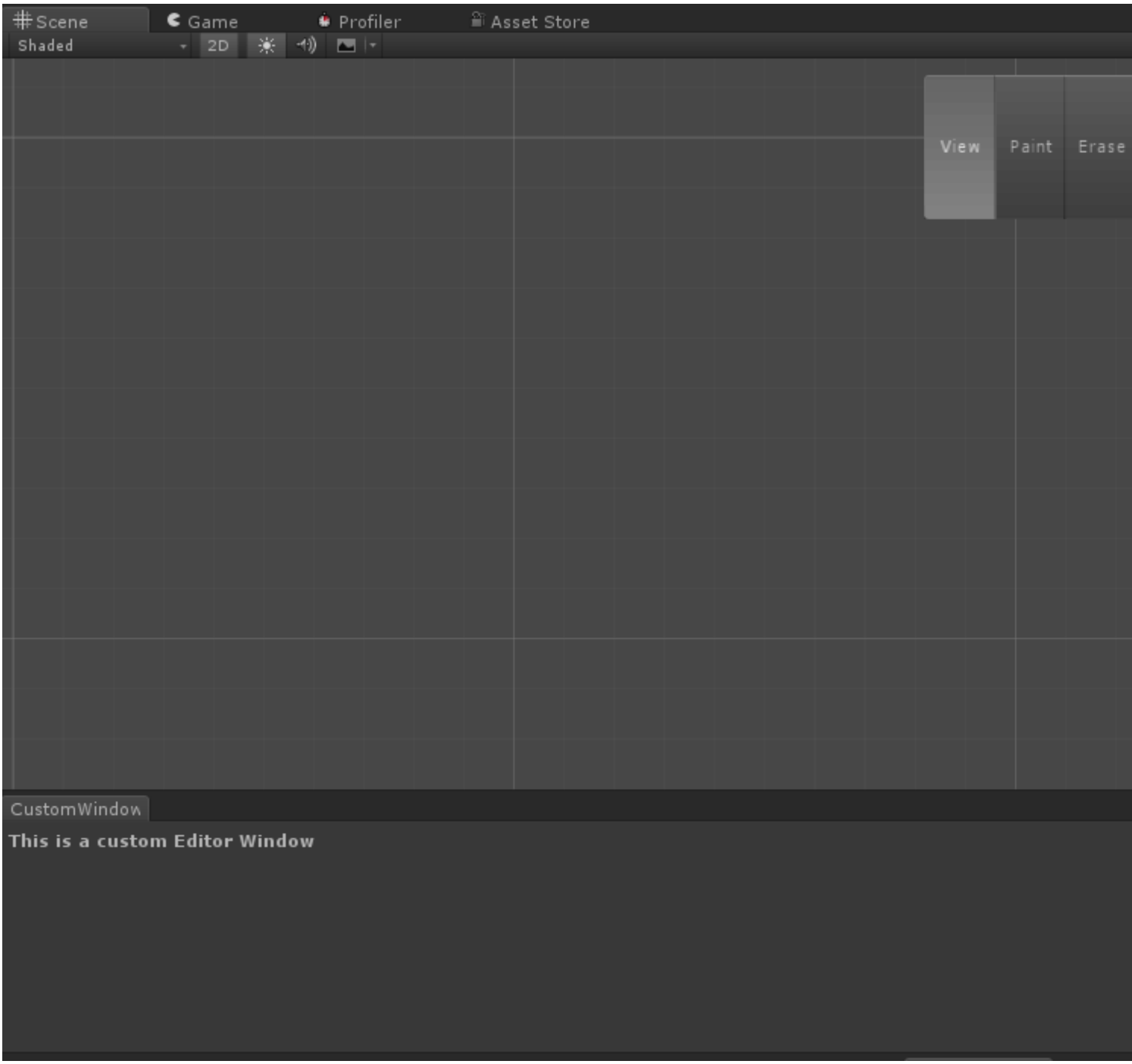
Вы можете сделать некоторые действительно продвинутые вещи в редакторе, а класс EditorWindow идеально подходит для отображения большого объема информации. Большинство продвинутых ресурсов в Unity Asset Store (например, NodeCanvas или PlayMaker) используют EditorWindow для отображения пользовательских представлений.

Рисование в SceneView

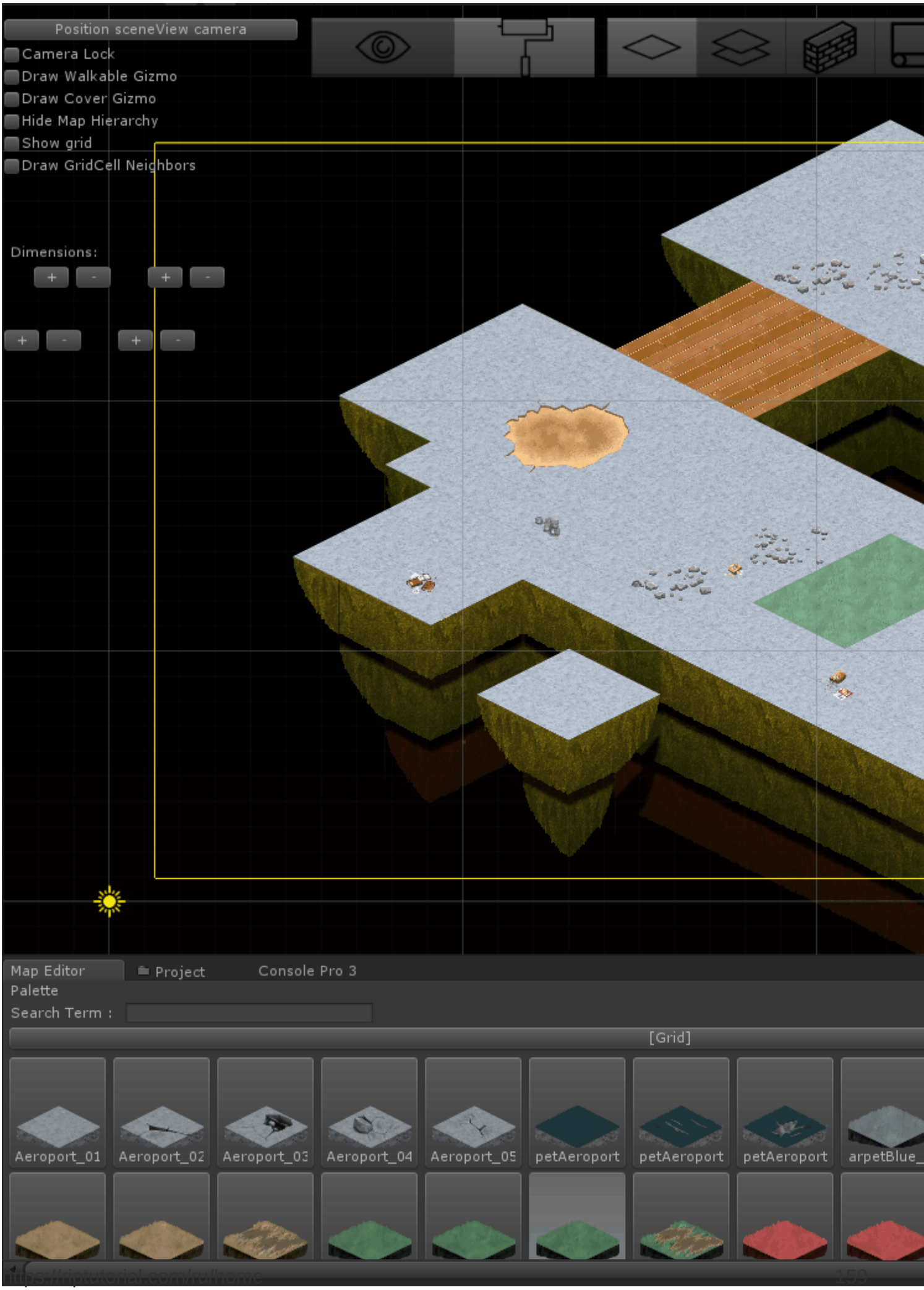
Одна интересная вещь, связанная с EditorWindow, заключается в отображении информации непосредственно в вашем SceneView. Таким образом, вы можете создать полностью настроенный редактор карт / мира, например, используя свой собственный EditorWindow в качестве палитры свойств и прослушивание кликов в SceneView для создания новых объектов. Вот пример:

```
using UnityEngine;
using System;
using UnityEditor;

public class CustomWindow : EditorWindow {

    private enum Mode {
        View = 0,
        Paint = 1,
        Erase = 2
    }

    private Mode CurrentMode = Mode.View;

    [MenuItem ("Window/Custom Window")]
    static void Init () {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow)EditorWindow.GetWindow (typeof (CustomWindow));
        window.Show();
    }

    void OnGUI () {
        GUILayout.Label ("This is a custom Editor Window", EditorStyles.boldLabel);
    }

    void OnEnable() {
        SceneView.onSceneGUIDelegate = SceneViewGUI;
        if (SceneView.lastActiveSceneView) SceneView.lastActiveSceneView.Repaint();
    }

    void SceneViewGUI(SceneView sceneView) {
        Handles.BeginGUI();
        // We define the toolbars' rects here
        var ToolBarRect = new Rect((SceneView.lastActiveSceneView.camera.pixelRect.width / 6),
10, (SceneView.lastActiveSceneView.camera.pixelRect.width * 4 / 6) ,
SceneView.lastActiveSceneView.camera.pixelRect.height / 5);
        GUILayout.BeginArea(ToolBarRect);
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
         CurrentMode = (Mode) GUILayout.Toolbar(
            (int) CurrentMode,
            Enum.GetNames(typeof(Mode)),
            GUILayout.Height(ToolBarRect.height));
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();
        GUILayout.EndArea();
        Handles.EndGUI();
    }
}
```

Это отобразит панель инструментов непосредственно в вашем SceneView



Вот краткий обзор того, как далеко вы можете пойти:


Position sceneView camera
Camera Lock
Draw Walkable Gizmo
Draw Cover Gizmo
Hide Map Hierarchy
Show grid
Draw GridCell Neighbors
Dimensions:
Map Editor
Project
Console Pro 3
Palette
Search Term :
[Grid]
Aeroport_01
Aeroport_02
Aeroport_03
Aeroport_04
Aeroport_05
petAeroport
petAeroport
petAeroport
arpetBlue_

расширение-редактора

глава 31: Ресурсы

Examples

Вступление

С классом ресурсов можно динамически загружать активы, которые не являются частью сцены. Это очень полезно, когда вам нужно использовать активы по требованию, например, локализовать многоязычные аудио, тексты и т. Д.

Активы должны быть помещены в папку « **Ресурсы**» . Возможно, что в иерархии проекта имеется несколько папок ресурсов. Класс `Resources` проверяет все доступные вам папки с ресурсами.

Каждый актив, размещенный в Ресурсах, будет включен в сборку, даже если в вашем коде не указано. Таким образом, не вставляйте активы в Ресурсы без разбора.

```
//Example of how to load language specific audio from Resources

[RequireComponent(typeof(AudioSource))]
public class loadIntroAudio : MonoBehaviour {
    void Start () {
        string language = Application.systemLanguage.ToString();
        AudioClip ac = Resources.Load(language + "/intro") as AudioClip; //loading intro.mp3
specific for user's language (note the file file extension should not be used)
        if (ac==null)
        {
            ac = Resources.Load("English/intro") as AudioClip; //fallback to the english
version for any unsupported language
        }
        transform.GetComponent<AudioSource>().clip = ac;
        transform.GetComponent<AudioSource>().Play();
    }
}
```

Ресурсы 101

Вступление

Unity имеет несколько «специально названных» папок, которые позволяют использовать различные виды использования. Одна из этих папок называется «Ресурсы»,

Папка «Ресурсы» - один из двух способов загрузки активов во время выполнения в Unity (другой - AssetBundles (Unity Docs)

Папка «Ресурсы» может находиться где угодно внутри вашей папки «Активы», и вы

можете иметь несколько папок с именем «Ресурсы». Содержимое всех папок «Ресурсы» объединяется во время компиляции.

Основным способом загрузки актива из папки «Ресурсы» является использование функции « Ресурсы» . Эта функция принимает строковый параметр, который позволяет указать путь к файлу **относительно** папки «Ресурсы». Обратите внимание, что вам не нужно указывать расширения файлов при загрузке актива

```
public class ResourcesSample : MonoBehaviour {

    void Start () {
        //The following line will load a TextAsset named 'foobar' which was previously place
under 'Assets/Resources/Stackoverflow/foobar.txt'
        //Note the absence of the '.txt' extension! This is important!

        var text = Resources.Load<TextAsset>("Stackoverflow/foobar").text;
        Debug.Log(string.Format("The text file had this in it :: {0}", text));
    }
}
```

Объекты, которые состоят из нескольких объектов, также могут быть загружены из ресурсов. Примерами являются такие объекты, как 3D-модели с текстурами, испеченными или несколькими спрайтами.

```
//This example will load a multiple sprite texture from Resources named "A_Multiple_Sprite"
var sprites = Resources.LoadAll("A_Multiple_Sprite") as Sprite[];
```

Объединяя все это

Вот один из моих вспомогательных классов, которые я использую для загрузки всех звуков для любой игры. Вы можете прикрепить это к любому GameObject в сцене, и он загрузит указанные аудиофайлы из папки «Ресурсы / Звуки»

```
public class SoundManager : MonoBehaviour {

    void Start () {

        //An array of all sounds you want to load
        var filesToLoad = new string[] { "Foo", "Bar" };

        //Loop over the array, attach an Audio source for each sound clip and assign the
        //clip property.
        foreach(var file in filesToLoad) {
            var soundClip = Resources.Load<AudioClip>("Sounds/" + file);
            var audioSource = gameObject.AddComponent<AudioSource>();
            audioSource.clip = soundClip;
        }
    }
```

```
}
```

Итоговые заметки

1. Unity является разумным, когда дело доходит до включения активов в вашу сборку. Любой объект, который не сериализуется (т.е. используется в сцене, включенной в сборку), исключается из сборки. ОДНАКО, что это НЕ применяется к любому активу в папке Ресурсы. Поэтому, не переходите за борт при добавлении активов в эту папку

2. Активы, которые загружаются с использованием ресурсов. Load или Resources.LoadAll могут быть выгружены в будущем с помощью ресурсов. Ресурсы или ресурсы Ресурсы.UnloadAsset

Прочитайте Ресурсы онлайн: https://riptutorial.com/ru/unity3d/topic/4070/ресурсы

глава 32: Связь с сервером

Examples

Получить

Get получает данные с веб-сервера. и `new WWW("https://urlexample.com");` с url, но без второго параметра делает **Get** .

т.е.

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    public string url = "http://google.com";

    IEnumerator Start()
    {
        WWW www = new WWW(url); // One get.
        yield return www;
        Debug.Log(www.text); // The data of the url.
    }
}
```

Простая почта (почтовые поля)

Каждый экземпляр **WWW** со вторым параметром - это *сообщение* .

Ниже приведен пример отправки *идентификатора пользователя* и *пароля* на сервер.

```
void Login(string id, string pwd)
{
    WWWForm dataParameters = new WWWForm();    // Create a new form.
    dataParameters.AddField("username", id);
    dataParameters.AddField("password", pwd);   // Add fields.
    WWW www = new WWW(url+"/account/login",dataParameters);
    StartCoroutine("PostdataEnumerator", www);
}

IEnumerator PostdataEnumerator(WWW www)
{
    yield return www;
    if (!string.IsNullOrEmpty(www.error))
    {
        Debug.Log(www.error);
    }
    else
    {
        Debug.Log("Data Submitted");
    }
```

```
}
```

Сообщение (Загрузить файл)

Загрузка файла на сервер - это также сообщение. Вы можете легко загрузить файл через **WWW** , как **показано** ниже:

Загрузка Zip-файла на сервер

```
string mainUrl = "http://server/upload/";
string saveLocation;

void Start()
{
    saveLocation = "ftp:///home/xxx/x.zip"; // The file path.
    StartCoroutine(PrepareFile());
}

// Prepare The File.
IEnumerator PrepareFile()
{
    Debug.Log("saveLoacation = " + saveLocation);

    // Read the zip file.
    WWW loadTheZip = new WWW(saveLocation);

    yield return loadTheZip;

    PrepareStepTwo(loadTheZip);
}

void PrepareStepTwo(WWW post)
{
    StartCoroutine(UploadTheZip(post));
}

// Upload.
IEnumerator UploadTheZip(WWW post)
{
    // Create a form.
    WWWForm form = new WWWForm();

    // Add the file.
    form.AddBinaryData("myTestFile.zip",post.bytes,"myFile.zip","application/zip");

    // Send POST request.
    string url = mainUrl;
    WWW POSTZIP = new WWW(url,form);

    Debug.Log("Sending zip...");
    yield return POSTZIP;
    Debug.Log("Zip sent!");
}
```

В этом примере он использует **сопрограмму** для подготовки и загрузки файла, если вы хотите узнать больше о сопрограмме Unity, посетите Coroutines .

Отправка запроса на сервер

Существует множество способов общения с серверами, использующими Unity в качестве клиента (некоторые методологии лучше других, в зависимости от вашей цели). Во-первых, необходимо определить необходимость того, чтобы сервер мог эффективно отправлять операции на сервер и с него. В этом примере мы отправим несколько данных на наш сервер для проверки.

Скорее всего, программист установит на своем сервере своего рода обработчик, чтобы получать события и отвечать на запросы соответственно, однако это выходит за рамки этого примера.

C #:

```
using System.Net;
using System.Text;

public class TestCommunicationWithServer
{
    public string SendDataToServer(string url, string username, string password)
    {
        WebClient client = new WebClient();

        // This specialized key-value pair will store the form data we're sending to the
server
        var loginData = new System.Collections.Specialized.NameValueCollection();
        loginData.Add("Username", username);
        loginData.Add("Password", password);

        // Upload client data and receive a response
        byte[] opBytes = client.UploadValues(ServerIpAddress, "POST", loginData);

        // Encode the response bytes into a proper string
        string opResponse = Encoding.UTF8.GetString(opBytes);

        return opResponse;
    }
```

Первое, что нужно сделать - это бросить в свои операторы, которые позволяют нам использовать классы WebClient и NameValueCollection.

В этом примере функция SendDataToServer принимает 3 (необязательных) строковых параметра:

1. Url сервера, с которым мы общаемся
2. Первая часть данных
3. Вторая часть данных, которые мы отправляем на сервер

Имя пользователя и пароль - это дополнительные данные, которые я отправляю на сервер. В этом примере мы используем его для дальнейшей проверки из базы данных или любого другого внешнего хранилища.

Теперь, когда мы настроили нашу структуру, мы создадим новый WebClient, который будет использоваться для фактической отправки наших данных. Теперь нам нужно загрузить наши данные в наш NameValueCollection и загрузить данные на сервер.

Функция UploadValues также принимает 3 необходимых параметра:

1. IP-адрес сервера
2. Метод HTTP
3. Данные, которые вы отправляете (имя пользователя и пароль в нашем случае)

Эта функция возвращает массив байтов ответа от сервера. Нам нужно закодировать возвращаемый массив байтов в правильную строку, чтобы фактический мог манипулировать и анализировать ответ.

Можно было бы сделать что-то вроде этого:

```
if(opResponse.Equals(ReturnMessage.Success))
{
    Debug.Log("Unity client has successfully sent and validated data on server.");
}
```

Теперь вы все еще можете смутить, поэтому, я думаю, я дам краткое объяснение того, как обращаться с сервером ответов на стороне.

В этом примере я буду использовать PHP для обработки ответа от клиента. Я бы рекомендовал использовать PHP в качестве исходного кода для сценариев, потому что он супер универсален, прост в использовании и, самое главное, быстро. Определенно есть другие способы обработки ответа на сервере, но, на мой взгляд, PHP - самая простая и простая реализация в Unity.

PHP:

```
// Check to see if the unity client send the form data
if(!isset($_REQUEST['Username']) || !isset($_REQUEST['Password']))
{
    echo "Empty";
}
else
{
    // Unity sent us the data - its here so do whatever you want

    echo "Success";

}
```

Так что это самая важная часть - «эхо». Когда наш клиент загружает данные на сервер, клиент сохраняет ответ (или ресурс) в этот массив байтов. Как только клиент получит ответ, вы знаете, что данные были проверены, и вы можете перейти на клиент после того, как это событие произошло. Вам также нужно подумать о том, какой тип данных вы

отправляете (в некоторой степени) и как минимизировать сумму, которую вы отправляете.

Таким образом, это только один способ отправки / получения данных от Unity - есть другие способы, которые могут быть более эффективными для вас в зависимости от вашего проекта.

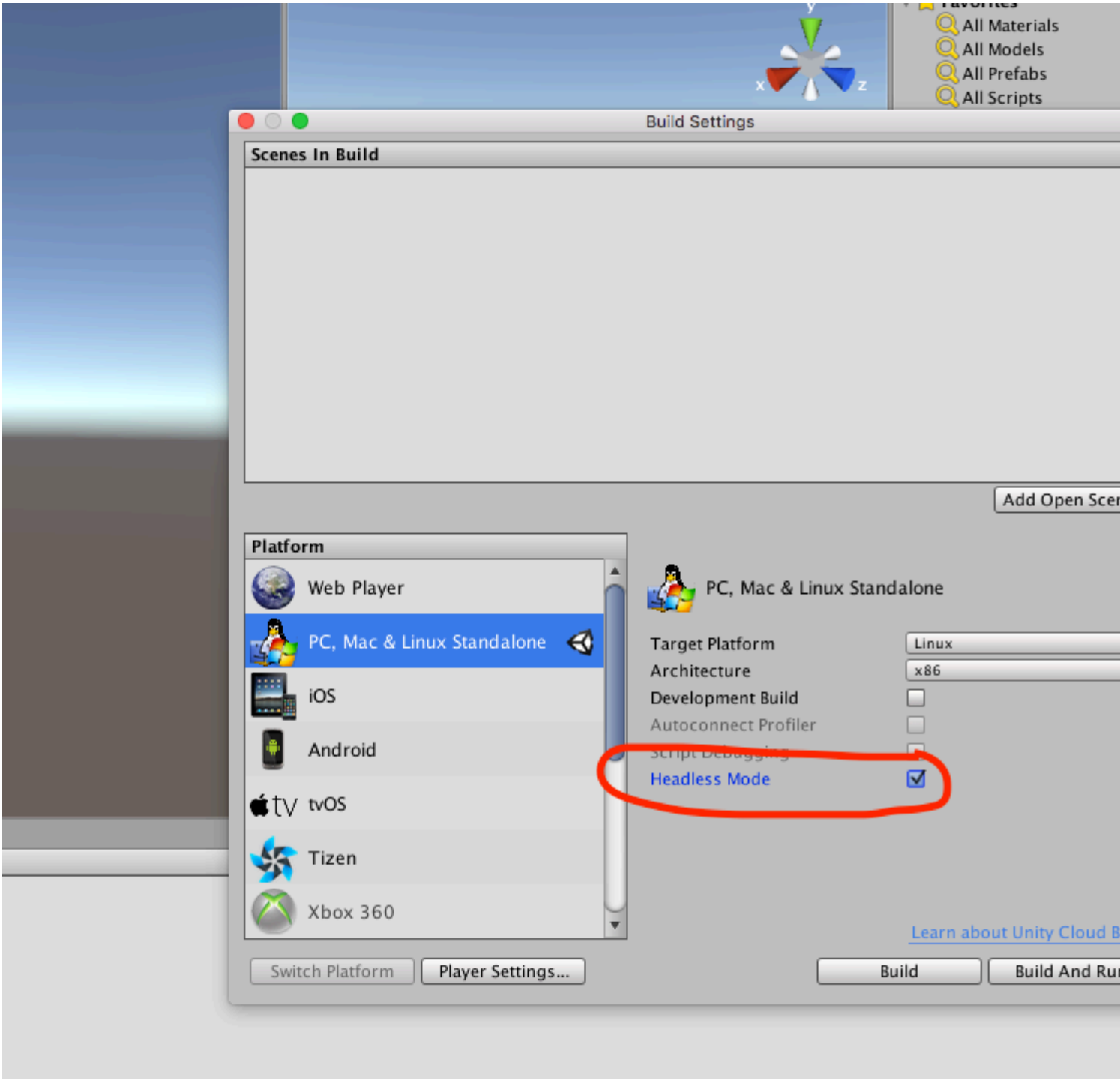
Прочитайте Связь с сервером онлайн: https://riptutorial.com/ru/unity3d/topic/5578/связь-с-сервером

глава 33: сетей

замечания

Безголовый режим в Unity

Если вы создаете сервер для развертывания в Linux, настройки Build имеют опцию «Безголовый режим». Прикладная сборка с этой опцией ничего не отображает и не читает ввод пользователя, что обычно требуется для сервера.



Examples

Создание сервера, клиента и отправка сообщения.

Unity networking предоставляет API высокого уровня (HLA) для обработки абстрагирования сетевых коммуникаций с реализаций низкого уровня.

В этом примере мы увидим, как создать сервер, который может связываться с одним или несколькими клиентами.

HLA позволяет нам легко сериализовать класс и отправлять объекты этого класса по сети.

Класс, который мы используем для сериализации

Этот класс должен наследовать от MessageBase, в этом примере мы просто отправим строку внутри этого класса.

```
using System;
using UnityEngine.Networking;

public class MyNetworkMessage : MessageBase
{
    public string message;
}
```

Создание сервера

Мы создаем сервер, который прослушивает порт 9999, допускает максимум 10 подключений и считывает объекты из сети нашего пользовательского класса.

HLA связывает различные типы сообщений с идентификатором. Существует тип сообщений по умолчанию, определенный в классе MsgType из Unity Networking. Например, тип подключения имеет идентификатор 32, и он вызывается на сервере, когда клиент подключается к нему или клиент, когда он подключается к серверу. Вы можете регистрировать обработчики для управления различными типами сообщений.

Когда вы отправляете собственный класс, например наш случай, мы определяем обработчики с новым идентификатором, связанным с классом, который мы отправляем по сети.

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class Server : MonoBehaviour {
```

```
int port = 9999;
int maxConnections = 10;

// The id we use to identify our messages and register the handler
short messageID = 1000;

// Use this for initialization
void Start () {
    // Usually the server doesn't need to draw anything on the screen
    Application.runInBackground = true;
    CreateServer();
}

void CreateServer() {
    // Register handlers for the types of messages we can receive
    RegisterHandlers ();

    var config = new ConnectionConfig ();
    // There are different types of channels you can use, check the official documentation
    config.AddChannel (QosType.ReliableFragmented);
    config.AddChannel (QosType.UnreliableFragmented);

    var ht = new HostTopology (config, maxConnections);

    if (!NetworkServer.Configure (ht)) {
        Debug.Log ("No server created, error on the configuration definition");
        return;
    } else {
        // Start listening on the defined port
        if(NetworkServer.Listen (port))
            Debug.Log ("Server created, listening on port: " + port);
        else
            Debug.Log ("No server created, could not listen to the port: " + port);
    }
}

void OnApplicationQuit() {
    NetworkServer.Shutdown ();
}

private void RegisterHandlers () {
    // Unity have different Messages types defined in MsgType
    NetworkServer.RegisterHandler (MsgType.Connect, OnClientConnected);
    NetworkServer.RegisterHandler (MsgType.Disconnect, OnClientDisconnected);

    // Our message use his own message type.
    NetworkServer.RegisterHandler (messageID, OnMessageReceived);
}

private void RegisterHandler(short t, NetworkMessageDelegate handler) {
    NetworkServer.RegisterHandler (t, handler);
}

void OnClientConnected(NetworkMessage netMessage)
{
    // Do stuff when a client connects to this server

    // Send a thank you message to the client that just connected
    MyNetworkMessage messageContainer = new MyNetworkMessage();
    messageContainer.message = "Thanks for joining!";
```

```
        // This sends a message to a specific client, using the connectionId
        NetworkServer.SendToClient(netMessage.conn.connectionId,messageID,messageContainer);

        // Send a message to all the clients connected
        messageContainer = new MyNetworkMessage();
        messageContainer.message = "A new player has conencted to the server";

        // Broadcast a message a to everyone connected
        NetworkServer.SendToAll(messageID,messageContainer);
    }

    void OnClientDisconnected(NetworkMessage netMessage)
    {
        // Do stuff when a client dissconnects
    }

    void OnMessageReceived(NetworkMessage netMessage)
    {
        // You can send any object that inherence from MessageBase
        // The client and server can be on different projects, as long as the MyNetworkMessage
or the class you are using have the same implementation on both projects
        // The first thing we do is deserialize the message to our custom type
        var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();
        Debug.Log("Message received: " + objectMessage.message);

    }
}
```

Клиент

Теперь мы создаем Клиент

```
using System;
using UnityEngine;
using UnityEngine.Networking;


public class Client : MonoBehaviour
{
    int port = 9999;
    string ip = "localhost";

    // The id we use to identify our messages and register the handler
    short messageID = 1000;

    // The network client
    NetworkClient client;

    public Client ()
    {
        CreateClient();
    }

    void CreateClient()
    {
        var config = new ConnectionConfig ();
```

```
        // Config the Channels we will use
        config.AddChannel (QosType.ReliableFragmented);
        config.AddChannel (QosType.UnreliableFragmented);

        // Create the client ant attach the configuration
        client = new NetworkClient ();
        client.Configure (config,1);

        // Register the handlers for the different network messages
        RegisterHandlers();

        // Connect to the server
        client.Connect (ip, port);
    }

    // Register the handlers for the different message types
    void RegisterHandlers () {

        // Unity have different Messages types defined in MsgType
        client.RegisterHandler (messageID, OnMessageReceived);
        client.RegisterHandler(MsgType.Connect, OnConnected);
        client.RegisterHandler(MsgType.Disconnect, OnDisconnected);
    }

    void OnConnected(NetworkMessage message) {
        // Do stuff when connected to the server

        MyNetworkMessage messageContainer = new MyNetworkMessage();
        messageContainer.message = "Hello server!";

        // Say hi to the server when connected
        client.Send(messageID,messageContainer);
    }

    void OnDisconnected(NetworkMessage message) {
        // Do stuff when disconnected to the server
    }

    // Message received from the server
    void OnMessageReceived(NetworkMessage netMessage)
    {
        // You can send any object that inherence from MessageBase
        // The client and server can be on different projects, as long as the MyNetworkMessage
or the class you are using have the same implementation on both projects
        // The first thing we do is deserialize the message to our custom type
        var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();

        Debug.Log("Message received: " + objectMessage.message);
    }
}
```

Прочитайте сетей онлайн: https://riptutorial.com/ru/unity3d/topic/5671/сетей

глава 34: Синглтоны в единстве

замечания

Несмотря на то, что есть мысли, которые приводят к аргументам, почему безусловное использование Singletons - плохая идея, например Singleton on gameprogrammingpatterns.com , бывают случаи, когда вы захотите сохранить GameObject в Unity в нескольких сценах (например, для бесшовной фоновой музыки) обеспечивая при этом существование не более одного экземпляра; идеальный вариант использования для Singleton.

Добавив этот скрипт в GameObject, как только он будет создан (например, включив его где угодно в сцену), он останется активным во всех сценариях, и только один экземпляр будет когда-либо существовать.

ScriptableObject (UnityDoc) экземпляры обеспечивают действительную альтернативу Одиночки для некоторых случаев применения. Хотя они не подразумевают принудительное применение правила единственного экземпляра, они сохраняют свое состояние между сценами и прекрасно играют с процессом сериализации Unity. Они также способствуют инверсии контроля, поскольку зависимости вводятся через редактор .

```
// MyAudioManager.cs
using UnityEngine;

[CreateAssetMenu] // Remember to create the instance in editor
public class MyAudioManager : ScriptableObject {
    public void PlaySound() {}
}
```

```
// MyGameObject.cs
using UnityEngine;

public class MyGameObject : MonoBehaviour
{
    [SerializeField]
    MyAudioManager audioManager; //Insert through Inspector

    void OnEnable()
    {
        audioManager.PlaySound();
    }
}
```

дальнейшее чтение

- Реализация Singleton в C #

Examples

Выполнение с использованием RuntimeInitializeOnLoadMethodAttribute

Начиная с **Unity 5.2.5** , можно использовать RuntimeInitializeOnLoadMethodAttribute для выполнения логики инициализации в обход порядка исполнения MonoBehaviour . Он обеспечивает возможность создания более чистой и надежной реализации:

```
using UnityEngine;

sealed class GameDirector : MonoBehaviour
{
    // Because of using RuntimeInitializeOnLoadMethod attribute to find/create and
    // initialize the instance, this property is accessible and
    // usable even in Awake() methods.
    public static GameDirector Instance
    {
        get; private set;
    }

    // Thanks to the attribute, this method is executed before any other MonoBehaviour
    // logic in the game.
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    static void OnRuntimeMethodLoad()
    {
        var instance = FindObjectOfType<GameDirector>();

        if (instance == null)
            instance = new GameObject("Game Director").AddComponent<GameDirector>();

        DontDestroyOnLoad(instance);

        Instance = instance;
    }

    // This Awake() will be called immediately after AddComponent() execution
    // in the OnRuntimeMethodLoad(). In other words, before any other MonoBehaviour's
    // in the scene will begin to initialize.
    private void Awake()
    {
        // Initialize non-MonoBehaviour logic, etc.
        Debug.Log("GameDirector.Awake()", this);
    }
}
```

Результирующий порядок выполнения:

1. `GameDirector.OnRuntimeMethodLoad()` запущен ...
2. `GameDirector.Awake()`
3. `GameDirector.OnRuntimeMethodLoad()` завершено.
4. `OtherMonoBehaviour1.Awake()`
5. `OtherMonoBehaviour2.Awake()` и т. Д.

Простой Singleton MonoBehaviour в Unity C

В этом примере частный статический экземпляр класса объявляется в начале.

Значение статического поля распределяется между экземплярами, поэтому , если новый экземпляр этого класса получает создан , `if` будет найти ссылку на первый объект Singleton, разрушив новый экземпляр (или его игровой объект).

```
using UnityEngine;

public class SingletonExample : MonoBehaviour {

    private static SingletonExample _instance;

    void Awake(){

        if (_instance == null){

            _instance = this;
            DontDestroyOnLoad(this.gameObject);

            //Rest of your Awake code

        } else {
            Destroy(this);
        }
    }

    //Rest of your class code

}
```

Продвинутое единство Singleton

В этом примере сочетаются несколько вариантов однопользовательских моноблоков MonoBehaviour, найденных в Интернете, в одном и позволяют изменять его поведение в зависимости от глобальных статических полей.

Этот пример был протестирован с использованием Unity 5. Чтобы использовать этот синглтон, все, что вам нужно сделать, это расширить его следующим образом: `public class MySingleton : Singleton<MySingleton> {}` . Вам также может потребоваться переопределить `AwakeSingleton` чтобы использовать его вместо обычного `Awake` . Для дальнейшей настройки измените значения по умолчанию для статических полей, как описано ниже.

---

1. Эта реализация использует атрибут DisallowMultipleComponent для хранения одного экземпляра на GameObject.
2. Этот класс является абстрактным и может быть расширен только. Он также содержит один виртуальный метод `AwakeSingleton` который необходимо переопределить, вместо того чтобы реализовать обычный `Awake` .
3. Эта реализация является потокобезопасной.

4. Этот синглтон оптимизирован. Используя `instantiated` флажка вместо экземпляра null check, мы избегаем накладных расходов, связанных с реализацией Unity оператора `==` . (Читать дальше)
5. Эта реализация не позволяет вызвать вызовы одиночного экземпляра, когда он собирается уничтожить Unity.
6. Этот синглтон поставляется со следующими параметрами:

- `FindInactive` : искать другие экземпляры компонентов того же типа, подключенные к неактивному GameObject.
- `Persist` : сохранять ли живые объекты между сценами.
- `DestroyOthers` : уничтожить ли какие-либо другие компоненты одного типа и сохранить только один.
- `Lazy` : установить одиночный экземпляр «на лету» (в `Awake`) или только «по требованию» (когда вызывается геттер).

```
using UnityEngine;

[DisallowMultipleComponent]
public abstract class Singleton<T> : MonoBehaviour where T : Singleton<T>
{
    private static volatile T instance;
    // thread safety
    private static object _lock = new object();
    public static bool FindInactive = true;
    // Whether or not this object should persist when loading new scenes. Should be set in
Init().
    public static bool Persist;
    // Whether or not destory other singleton instances if any. Should be set in Init().
    public static bool DestroyOthers = true;
    // instead of heavy comparision (instance != null)
    // http://blogs.unity3d.com/2014/05/16/custom-operator-should-we-keep-it/
    private static bool instantiated;

    private static bool applicationIsQuitting;

    public static bool Lazy;

    public static T Instance
    {
        get
        {
            if (applicationIsQuitting)
            {
                Debug.LogWarningFormat("[Singleton] Instance '{0}' already destroyed on
application quit. Won't create again - returning null.", typeof(T));
                return null;
            }
            lock (_lock)
            {
                if (!instantiated)
                {
                    Object[] objects;
                    if (FindInactive) { objects = Resources.FindObjectsOfTypeAll(typeof(T)); }
                    else { objects = FindObjectsOfType(typeof(T)); }
                    if (objects == null || objects.Length < 1)
```

```
                    {
                        GameObject singleton = new GameObject();
                        singleton.name = string.Format("{0} [Singleton]", typeof(T));
                        Instance = singleton.AddComponent<T>();
                        Debug.LogWarningFormat("[Singleton] An Instance of '{0}' is needed in
the scene, so '{1}' was created{2}", typeof(T), singleton.name, Persist ? " with
DontDestoryOnLoad." : ".");
                    }
                    else if (objects.Length >= 1)
                    {
                        Instance = objects[0] as T;
                        if (objects.Length > 1)
                        {
                            Debug.LogWarningFormat("[Singleton] {0} instances of '{1}'!",
objects.Length, typeof(T));
                            if (DestroyOthers)
                            {
                                for (int i = 1; i < objects.Length; i++)
                                {
                                    Debug.LogWarningFormat("[Singleton] Deleting extra '{0}'
instance attached to '{1}'", typeof(T), objects[i].name);
                                    Destroy(objects[i]);
                                }
                            }
                        }
                        return instance;
                    }
                }
                return instance;
            }
        }
        protected set
        {
            instance = value;
            instantiated = true;
            instance.AwakeSingleton();
            if (Persist) { DontDestroyOnLoad(instance.gameObject); }
        }
    }

    // if Lazy = false and gameObject is active this will set instance
    // unless instance was called by another Awake method
    private void Awake()
    {
        if (Lazy) { return; }
        lock (_lock)
        {
            if (!instantiated)
            {
                Instance = this as T;
            }
            else if (DestroyOthers && Instance.GetInstanceID() != GetInstanceID())
            {
                Debug.LogWarningFormat("[Singleton] Deleting extra '{0}' instance attached to
'{1}'", typeof(T), name);
                Destroy(this);
            }
        }
    }

    // this might be called for inactive singletons before Awake if FindInactive = true
```

```
    protected virtual void AwakeSingleton() {}

    protected virtual void OnDestroy()
    {
        applicationIsQuitting = true;
        instantiated = false;
    }
}
```

Реализация Singleton через базовый класс

В проектах, которые имеют несколько одноэлементных классов (как это часто бывает), может быть чисто и удобно абстрагировать поведение singleton к базовому классу:

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

public abstract class MonoBehaviourSingleton<T> : MonoBehaviour {

    private static Dictionary<Type, object> _singletons
        = new Dictionary<Type, object>();

    public static T Instance {
        get {
            return (T)_singletons[typeof(T)];
        }
    }

    void OnEnable() {
        if (_singletons.ContainsKey(GetType())) {
            Destroy(this);
        } else {
            _singletons.Add(GetType(), this);
            DontDestroyOnLoad(this);
        }
    }
}
```

MonoBehaviour может затем реализовать одноэлементный шаблон, расширив MonoBehaviourSingleton. Такой подход позволяет использовать шаблон с минимальным размером на сам Синглтон:

```
using UnityEngine;
using System.Collections;

public class SingletonImplementation : MonoBehaviourSingleton<SingletonImplementation> {

    public string Text= "String Instance";

    // Use this for initialisation
    IEnumerator Start () {
        var demonstration = "SingletonImplementation.Start()\n" +
                            "Note that the this text logs only once and\n"
                            "only one class instance is allowed to exist.";
        Debug.Log(demonstration);
```

```
        yield return new WaitForSeconds(2f);
        var secondInstance = new GameObject();
        secondInstance.AddComponent<SingletonImplementation>();
    }

}
```

Обратите внимание, что одним из преимуществ шаблона singleton является то, что ссылка на экземпляр может быть получена статически:

```
// Logs: String Instance
Debug.Log(SingletonImplementation.Instance.Text);
```

Имейте в виду, что эту практику следует минимизировать, чтобы уменьшить сцепление. Этот подход также имеет небольшую производительность из-за использования словаря, но поскольку эта коллекция может содержать только один экземпляр каждого одноэлементного класса, компромисс с точки зрения принципа DRY («Не повторяйте себя»), читаемости и удобный маленький.

Шаблон Singleton с использованием системы Entity-Componity Unitys

Основная идея состоит в том, чтобы использовать GameObjects для представления синглетов, которые имеют несколько преимуществ:

- Сохраняет сложность до минимума, но поддерживает концепции, такие как инъекция зависимостей
- Синглтоны имеют нормальный жизненный цикл Unity как часть системы Entity-Component
- Синглтоны могут быть лениво загружены и кэшированы локально, где это необходимо (например, в циклах обновления)
- Не требуется статических полей
- Не нужно изменять существующие MonoBehaviours / Components, чтобы использовать их как Singletons
- Легко перезагрузиться (просто уничтожьте GameObject Singletons), будет снова загружен с лени при следующем использовании
- Легко вводить насмешки (просто инициализируйте его макетом перед его использованием)
- Проверка и настройка с использованием обычного редактора Unity и может произойти уже во время редактора (Снимок экрана Singleton, доступный в редакторе Unity)

Test.cs (который использует пример singleton):

```
using UnityEngine;
using UnityEngine.Assertions;

public class Test : MonoBehaviour {
```

```
    void Start() {
        ExampleSingleton singleton = ExampleSingleton.instance;
        Assert.IsNotNull(singleton); // automatic initialization on first usage
        Assert.AreEqual("abc", singleton.myVar1);
        singleton.myVar1 = "123";
        // multiple calls to instance() return the same object:
        Assert.AreEqual(singleton, ExampleSingleton.instance);
        Assert.AreEqual("123", ExampleSingleton.instance.myVar1);
    }
}
```

ExampleSingleton.cs (который содержит пример и действительный класс Singleton):

```
using UnityEngine;
using UnityEngine.Assertions;

public class ExampleSingleton : MonoBehaviour {
    public static ExampleSingleton instance { get { return Singleton.get<ExampleSingleton>();
} }
    public string myVar1 = "abc";
    public void Start() { Assert.AreEqual(this, instance, "Singleton more than once in
scene"); }
}

/// <summary> Helper that turns any MonBehaviour or other Component into a Singleton
</summary>
public static class Singleton {
    public static T get<T>() where T : Component {
        return GetOrAddGo("Singletons").GetOrAddChild("" + typeof(T)).GetOrAddComponent<T>();
    }
    private static GameObject GetOrAddGo(string goName) {
        var go = GameObject.Find(goName);
        if (go == null) { return new GameObject(goName); }
        return go;
    }
}

public static class GameObjectExtensionMethods {
    public static GameObject GetOrAddChild(this GameObject parentGo, string childName) {
        var childGo = parentGo.transform.FindChild(childName);
        if (childGo != null) { return childGo.gameObject; } // child found, return it
        var newChild = new GameObject(childName);        // no child found, create it
        newChild.transform.SetParent(parentGo.transform, false); // add it to parent
        return newChild;
    }

    public static T GetOrAddComponent<T>(this GameObject parentGo) where T : Component {
        var comp = parentGo.GetComponent<T>();
        if (comp == null) { return parentGo.AddComponent<T>(); }
        return comp;
    }
}
```

Два метода расширения для GameObject полезны и в других ситуациях, если вам не нужны они, перемещая их внутри класса Singleton и делая их закрытыми.

MonoBehaviour & ScriptableObject на основе Singleton Class

Большинство примеров Singleton используют MonoBehaviour как базовый класс. Основным недостатком является то, что этот класс Singleton работает только во время выполнения. Это имеет некоторые недостатки:

- Нет способа прямого редактирования полей singleton, кроме изменения кода.
- Никакой способ сохранить ссылку на другие активы на Singleton.
- Невозможно установить синглтон в качестве места назначения события Unity UI. Я в конечном итоге использую то, что я называю «Прокси-компоненты», что его единственное предложение состоит в том, чтобы иметь 1 линейный метод, который вызывает «GameManager.Instance.SomeGlobalMethod ()».

Как отмечалось в примечаниях, существуют реализации, которые пытаются решить это, используя ScriptableObjects в качестве базового класса, но теряют преимущества MonoBehaviour во время выполнения. Эта реализация решает эти проблемы, используя ScriptableObject как базовый класс и связанное с ним MonoBehavior во время выполнения:

- Это актив, поэтому его свойства могут быть обновлены в редакторе, как и любой другой ресурс Unity.
- Он отлично работает с процессом сериализации Unity.
- Можно назначить ссылки на singleton другим ресурсам из редактора (зависимости вводятся через редактор).
- События Unity могут напрямую вызывать методы на Singleton.
- Можно вызвать его из любой точки в базе кода, используя « SingletonClassName.Instance»
- Имеет доступ к событиям времени MonoBehaviour и таким методам, как: Обновление, Пробуждение, Старт, ФиксUpdate, StartCoroutine и т. Д.

```
/*************************************************************
 * Better Singleton by David Darias
 * Use as you like - credit where due would be appreciated :D
 * Licence: WTFPL V2, Dec 2014
 * Tested on Unity v5.6.0 (should work on earlier versions)
 * 03/02/2017 - v1.1
 * ************************************************************/

using System;
using UnityEngine;
using SingletonScriptableObjectNamespace;

public class SingletonScriptableObject<T> :
SingletonScriptableObjectNamespace.BehaviourScriptableObject where T :
SingletonScriptableObjectNamespace.BehaviourScriptableObject
{
    //Private reference to the scriptable object
    private static T _instance;
    private static bool _instantiated;
    public static T Instance
    {
        get
        {
            if (_instantiated) return _instance;
```

```
            var singletonName = typeof(T).Name;
            //Look for the singleton on the resources folder
            var assets = Resources.LoadAll<T>("");
            if (assets.Length > 1) Debug.LogError("Found multiple " + singletonName + "s on
the resources folder. It is a Singleton ScriptableObject, there should only be one.");
            if (assets.Length == 0)
            {
                _instance = CreateInstance<T>();
                Debug.LogError("Could not find a " + singletonName + " on the resources
folder. It was created at runtime, therefore it will not be visible on the assets folder and
it will not persist.");
            }
            else _instance = assets[0];
            _instantiated = true;
            //Create a new game object to use as proxy for all the MonoBehaviour methods
            var baseObject = new GameObject(singletonName);
            //Deactivate it before adding the proxy component. This avoids the execution of
the Awake method when the the proxy component is added.
            baseObject.SetActive(false);
            //Add the proxy, set the instance as the parent and move to DontDestroyOnLoad
scene
            SingletonScriptableObjectNamespace.BehaviourProxy proxy =
baseObject.AddComponent<SingletonScriptableObjectNamespace.BehaviourProxy>();
            proxy.Parent = _instance;
            Behaviour = proxy;
            DontDestroyOnLoad(Behaviour.gameObject);
            //Activate the proxy. This will trigger the MonoBehaviourAwake.
            proxy.gameObject.SetActive(true);
            return _instance;
        }
    }
    //Use this reference to call MonoBehaviour specific methods (for example StartCoroutine)
    protected static MonoBehaviour Behaviour;
    public static void BuildSingletonInstance() {
SingletonScriptableObjectNamespace.BehaviourScriptableObject i = Instance; }
    private void OnDestroy(){ _instantiated = false; }
}

// Helper classes for the SingletonScriptableObject
namespace SingletonScriptableObjectNamespace
{
    #if UNITY_EDITOR
    //Empty custom editor to have cleaner UI on the editor.
    using UnityEditor;
    [CustomEditor(typeof(BehaviourProxy))]
    public class BehaviourProxyEditor : Editor
    {
        public override void OnInspectorGUI(){}
    }

    #endif

    public class BehaviourProxy : MonoBehaviour
    {
        public IBehaviour Parent;

        public void Awake() { if (Parent != null) Parent.MonoBehaviourAwake(); }
        public void Start() { if (Parent != null) Parent.Start(); }
        public void Update() { if (Parent != null) Parent.Update(); }
        public void FixedUpdate() { if (Parent != null) Parent.FixedUpdate(); }
    }
```

```
    public interface IBehaviour
    {
        void MonoBehaviourAwake();
        void Start();
        void Update();
        void FixedUpdate();
    }

    public class BehaviourScriptableObject : ScriptableObject, IBehaviour
    {
        public void Awake() { ScriptableObjectAwake(); }
        public virtual void ScriptableObjectAwake() { }
        public virtual void MonoBehaviourAwake() { }
        public virtual void Start() { }
        public virtual void Update() { }
        public virtual void FixedUpdate() { }
    }
}
```

Здесь приведен пример singleton-класса GameManager с использованием SingletonScriptableObject (с большим количеством комментариев):

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

//this attribute is optional but recommended. It will allow the creation of the singleton via
the asset menu.
//the singleton asset should be on the Resources folder.
[CreateAssetMenu(fileName = "GameManager", menuName = "Game Manager", order = 0)]
public class GameManager : SingletonScriptableObject<GameManager> {

    //any properties as usual
    public int Lives;
    public int Points;

    //optional (but recommended)
    //this method will run before the first scene is loaded. Initializing the singleton here
    //will allow it to be ready before any other GameObjects on every scene and will
    //will prevent the "initialization on first usage".
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    public static void BeforeSceneLoad() { BuildSingletonInstance(); }

    //optional,
    //will run when the Singleton Scriptable Object is first created on the assets.
    //Usually this happens on edit mode, not runtime. (the override keyword is mandatory for
this to work)
    public override void ScriptableObjectAwake(){
        Debug.Log(GetType().Name + " created." );
    }

    //optional,
    //will run when the associated MonoBehavioir awakes. (the override keyword is mandatory
for this to work)
    public override void MonoBehaviourAwake(){
        Debug.Log(GetType().Name + " behaviour awake." );
```

```
        //A coroutine example:
        //Singleton Objects do not have coroutines.
        //if you need to use coroutines use the atached MonoBehaviour
        Behaviour.StartCoroutine(SimpleCoroutine());
    }

    //any methods as usual
    private IEnumerator SimpleCoroutine(){
        while(true){
            Debug.Log(GetType().Name + " coroutine step." );
            yield return new WaitForSeconds(3);
        }
    }

    //optional,
    //Classic runtime Update method (the override keyword is mandatory for this to work).
    public override void Update(){

    }

    //optional,
    //Classic runtime FixedUpdate method (the override keyword is mandatory for this to work).
    public override void FixedUpdate(){

    }
}

/*
*  Notes:
*  - Remember that you have to create the singleton asset on edit mode before using it. You
have to put it on the Resources folder and of course it should be only one.
*  - Like other Unity Singleton this one is accessible anywhere in your code using the
"Instance" property i.e: GameManager.Instance
*/
```

Прочитайте Синглтоны в единстве онлайн: https://riptutorial.com/ru/unity3d/topic/2137/синглтоны-в-единстве

глава 35: Слои

Examples

Использование слоев

Уровни единства похожи на теги, так как они могут использоваться для определения объектов, которые должны взаимодействовать или должны вести себя определенным образом, однако слои в основном используются с функциями класса `Physics` : Документация Unity - физика

Слои представлены целым числом и могут быть переданы таким функциям таким образом:

```
using UnityEngine;
class LayerExample {

    public int layer;

    void Start()
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, 5f, layer);
    }
}
```

Использование слоя таким образом будет включать только Colliders, у GameEbjects которых есть слой, указанный в выполненных вычислениях. Это упрощает логику, а также улучшает производительность.

Структура LayerMask

Структура `LayerMask` - это интерфейс, который функционирует почти так же, как передача целого числа в соответствующую функцию. Тем не менее, его самое большое преимущество позволяет пользователю выбрать этот слой из раскрывающегося меню в инспекторе.

```
using UnityEngine;
class LayerMaskExample{

    public LayerMask mask;
    public Vector3 direction;

    void Start()
    {
        if(Physics.Raycast(transform.position, direction, 35f, mask))
        {
            Debug.Log("Raycast hit");
        }
    {
}
```

Он также имеет несколько статических функций, которые позволяют преобразовывать имена слоев в индексы или индексы в имена слоев.

```
using UnityEngine;
class NameToLayerExample{

    void Start()
    {
        int layerindex = LayerMask.NameToLayer("Obstacle");
    {
}
```

Чтобы упростить проверку уровня, определите следующий метод расширения.

```
public static bool IsInLayerMask(this GameObject @object, LayerMask layerMask)
{
    bool result = (1 << @object.layer & layerMask) == 0;

    return result;
}
```

Этот метод позволит вам проверить, находится ли игровой объект в layermask (выбранном в редакторе) или нет.

Прочитайте Слои онлайн: https://riptutorial.com/ru/unity3d/topic/4762/слои

глава 36: Сопрограммы

Синтаксис

- общедоступный Coroutine StartCoroutine (процедура IEnumerator);
- public Coroutine StartCoroutine (string methodName, значение объекта = null);
- public void StopCoroutine (string methodName);
- public void StopCoroutine (процедура IEnumerator);
- public void StopAllCoroutines ();

замечания

Требования к производительности

Лучше всего использовать сопрограммы в умеренных количествах, так как гибкость приносит стоимость производительности.

- Coroutines в большом количестве требует от CPU большего, чем стандартные методы обновления.
- В некоторых версиях Unity существует проблема, при которой сопрограммы производят мусор каждый цикл обновления из-за того, что Unity боксирует возвращаемое значение `MoveNext` . Последнее наблюдалось в 5.4.0b13. (Отчет об ошибке)

Сокращение мусора путем кэширования YieldInstructions

Общим трюком для сокращения мусора, сгенерированного в сопрограммах, является кэширование `YieldInstruction` .

```
IEnumerator TickEverySecond()
{
    var wait = new WaitForSeconds(1f); // Cache
    while(true)
    {
        yield return wait; // Reuse
    }
}
```

Утолщение `null` дает лишнего мусора.

Examples

Сопрограммы

Во-первых, важно понять, что игровые движки (такие как Unity) работают над парадигмой, основанной на «фреймах».

Код выполняется во время каждого кадра.

Это включает собственный код Unity и ваш код.

Когда мы думаем о кадрах, важно понимать, что нет ***абсолютно*** никаких гарантий того, когда будут происходить кадры. Они ***не*** происходят в обычном ритме. Промежутки между кадрами могут быть, например, 0,02632, затем 0,021167, затем 0,029738 и т. Д. В этом примере все они «около» 1/50 секунды, но все они разные. И в любое время вы можете получить фрейм, который занимает гораздо больше времени или короче; и ваш код может быть выполнен в любой момент в пределах фрейма.

Помня об этом, вы можете спросить: как вы получаете доступ к этим фреймам в своем коде, в Unity?

Достаточно просто использовать либо вызов Update (), либо использовать сопрограмму. (В самом деле - они точно такие же: они позволяют запускать код в каждом кадре).

Цель сопрограммы состоит в том, что:

вы можете запустить некоторый код, а затем «остановить и подождать» ***до некоторого будущего фрейма*** .

Вы можете подождать до **следующего кадра** , вы можете подождать **несколько кадров** , или вы можете ждать некоторое ***приблизительное*** время в секундах в будущем.

Например, вы можете подождать «около одной секунды», то есть он будет ждать около одной секунды, а затем помещать ваш код в некоторый кадр примерно через одну секунду. (И действительно, в этом фрейме код мог быть запущен в любое время, как бы то ни было.) Повторить: это будет не ровно одна секунда. Точный момент времени не имеет смысла в игровом движке.

Внутри сопрограммы:

Чтобы подождать один кадр:

```
// do something
yield return null;  // wait until next frame
// do something
```

Подождать три кадра:

```
// do something
```

```
yield return null;  // wait until three frames from now
yield return null;
yield return null;
// do something
```

Подождать **примерно** полсекунды:

```
// do something
yield return new WaitForSeconds (0.5f); // wait for a frame in about .5 seconds
// do something
```

Делайте что-то каждый отдельный кадр:

```
while (true)
{
    // do something
    yield return null;  // wait until the next frame
}
```

Этот пример буквально идентичен простому помещению чего-то внутри вызова «Обновить» Unity: код «делать что-то» запускается каждый кадр.

пример

Присоедините тикер к `GameObject` . Пока этот игровой объект активен, галочка будет работать. Обратите внимание, что скрипт тщательно останавливает сопрограмму, когда игровой объект становится неактивным; это, как правило, важный аспект правильного использования инженерных сопроцессоров.

```
using UnityEngine;
using System.Collections;

public class Ticker:MonoBehaviour {

    void OnEnable()
    {
        StartCoroutine(TickEverySecond());
    }

    void OnDisable()
    {
        StopAllCoroutines();
    }

    IEnumerator TickEverySecond()
    {
        var wait = new WaitForSeconds(1f); // REMEMBER: IT IS ONLY APPROXIMATE
        while(true)
        {
            Debug.Log("Tick");
            yield return wait;  // wait for a frame, about 1 second from now
        }
```

```
    }
}
```

Завершение сопрограммы

Часто вы разрабатываете сопрограммы для естественного завершения, когда выполняются определенные цели.

```
IEnumerator TickFiveSeconds()
{
    var wait = new WaitForSeconds(1f);
    int counter = 1;
    while(counter < 5)
    {
        Debug.Log("Tick");
        counter++;
        yield return wait;
    }
    Debug.Log("I am done ticking");
}
```

Чтобы остановить сопрограмму «внутри» сопрограммы, вы не можете просто «вернуться», как вы хотели бы уйти от обычной функции. Вместо этого вы используете `yield break` .

```
IEnumerator ShowExplosions()
{
    ... show basic explosions
    if(player.xp < 100) yield break;
    ... show fancy explosions
}
```

Вы также можете заставить все сопрограммы, запущенные сценарием, остановиться до завершения.

```
void OnDisable()
{
    // Stops all running coroutines
    StopAllCoroutines();
}
```

Метод остановки *конкретной* сопрограммы от вызывающего зависит от того, как вы ее начали.

Если вы запустили coroutine по имени строки:

```
StartCoroutine("YourAnimation");
```

то вы можете остановить его, вызвав StopCoroutine с тем же именем строки:

```
StopCoroutine("YourAnimation");
```

Кроме того , вы можете сохранить ссылку на *либо* в `IEnumerator` , возвращаемый методом сопрограммного, *или* `Coroutine` объекта , возвращенный `StartCoroutine` и вызвать `StopCoroutine` на любом из этих:

```
public class SomeComponent : MonoBehaviour
{
    Coroutine routine;

    void Start () {
        routine = StartCoroutine(YourAnimation());
    }

    void Update () {
        // later, in response to some input...
        StopCoroutine(routine);
    }

    IEnumerator YourAnimation () { /* ... */ }
}
```

Методы MonoBehaviour, которые могут быть Coroutines

Есть три метода MonoBehaviour, которые можно сделать сопрограммами.

1. Начните()
2. OnBecameVisible ()
3. OnLevelWasLoaded ()

Это можно использовать для создания, например, скриптов, которые выполняются только тогда, когда объект видим для камеры.

```
using UnityEngine;
using System.Collections;

public class RotateObject : MonoBehaviour
{
    IEnumerator OnBecameVisible()
    {
        var tr = GetComponent<Transform>();
        while (true)
        {
            tr.Rotate(new Vector3(0, 180f * Time.deltaTime));
            yield return null;
        }
    }

    void OnBecameInvisible()
    {
        StopAllCoroutines();
    }
}
```

Цеповые сопрограммы

Корутинцы могут выжить внутри себя и ждать **других сопрограмм** .

Таким образом, вы можете цеплять последовательности - «один за другим».

Это очень просто и является основной, основной, технологией в Unity.

В играх абсолютно естественно, что некоторые вещи должны произойти «в порядке». Почти каждый «раунд» игры начинается с определенной серии событий, происходящих в течение некоторого времени в определенном порядке. Вот как вы можете начать гоночную игру:

```
IEnumerator BeginRace()
{
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
}
```

Итак, когда вы звоните в BeginRace ...

```
 StartCoroutine(BeginRace());
```

Он будет запускать рутину «подготовить гонку». (Возможно, мигание некоторых огней и запуск шума толпы, сброс баллов и т. Д.). Когда это будет завершено, оно запустит вашу последовательность обратного отсчета, где вы могли бы оживить, возможно, обратный отсчет времени в пользовательском интерфейсе. Когда это будет завершено, он запустит ваш код запуска, в котором вы, возможно, запустите звуковые эффекты, запустите некоторые AI-драйверы, поместите камеру определенным образом и так далее.

Для ясности поймите, что три вызова

```
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
```

сами должны **быть в** сопрограмме. То есть они должны быть в функции типа `IEnumerator` . Итак, в нашем примере это `IEnumerator BeginRace` . Итак, из «нормального» кода вы запускаете эту сопрограмму с вызовом `StartCoroutine` .

```
 StartCoroutine(BeginRace());
```

Чтобы еще больше понять цепочку, вот функция, которая направляет сопрограммы. Вы передаете массив сопрограмм. Функция выполняет столько сопрограмм, сколько вы проходите, по порядку, один за другим.

```
// run various routines, one after the other
IEnumerator OneAfterTheOther( params IEnumerator[] routines )
{
```

```
    foreach ( var item in routines )
    {
        while ( item.MoveNext() ) yield return item.Current;
    }

    yield break;
}
```

Вот как вы бы назвали это ... допустим, у вас есть три функции. Напомним, что все они должны быть `IEnumerator` :

```
IEnumerator PrepareRace()
{
    // codesay, crowd cheering and camera pan around the stadium
    yield break;
}

IEnumerator Countdown()
{
    // codesay, animate your countdown on UI
    yield break;
}

IEnumerator StartRace()
{
    // codesay, camera moves and light changes and launch the AIs
    yield break;
}
```

Вы бы назвали это так

```
StartCoroutine( MultipleRoutines( PrepareRace(), Countdown(), StartRace() ) );
```

или, возможно, так

```
IEnumerator[] routines = new IEnumerator[] {
     PrepareRace(),
     Countdown(),
     StartRace() };
StartCoroutine( MultipleRoutines( routines ) );
```

Повторяю, одним из самых основных требований в играх является то, что определенные вещи происходят один за другим «в последовательности» с течением времени. Вы достигаете этого в Единстве очень просто, с

```
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
```

Способы выхода

Вы можете подождать до следующего кадра.

```
yield return null; // wait until sometime in the next frame
```

Вы можете иметь несколько этих вызовов подряд, чтобы просто подождать столько кадров, сколько необходимо.

```
//wait for a few frames
yield return null;
yield return null;
```

Подождите ***примерно*** n секунд. Крайне важно понимать, что это всего лишь **приблизительное время** .

```
yield return new WaitForSeconds(n);
```

Совершенно невозможно использовать вызов «WaitForSeconds» для любой формы точного времени.

Часто вы хотите связать действия. Итак, сделайте что-нибудь, и когда это будет сделано, сделайте что-нибудь еще, и когда это будет сделано, сделайте что-нибудь еще. Для этого подожди еще одну сопрограмму:

```
yield return StartCoroutine(coroutine);
```

Поймите, что вы можете вызывать это только из сопрограммы. Так:

```
StartCoroutine(Test());
```

Вот как вы начинаете сопрограмму из «нормального» кода.

Затем внутри исполняемой сопрограммы:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

Это напечатает A, запустит длительный процесс и **сразу же напечатает B.** Он **не** будет ждать завершения долгого процесса. С другой стороны:

```
Debug.Log("A");
yield return StartCoroutine(LongProcess());
Debug.Log("B");
```

Это напечатает A, запустит длительный процесс, **дождитесь окончания** и затем напечатает B.

Всегда стоит помнить, что сопрограммы абсолютно не связаны ни с чем, ни с нитками. С помощью этого кода:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

легко думать об этом как о «как», начиная с LongProcess в другом потоке в фоновом режиме. Но это абсолютно неверно. Это всего лишь сопрограмма. Игровые движки основаны на фреймах, а «сопрограммы» в Unity просто позволяют вам получить доступ к фреймам.

Очень просто дождаться завершения веб-запроса.

```
void Start() {
    string url = "http://google.com";
    WWW www = new WWW(url);
    StartCoroutine(WaitForRequest(www));
}

IEnumerator WaitForRequest(WWW www) {
    yield return www;

    if (www.error == null) {
        //use www.data);
    }
    else {
        //use www.error);
    }
}
```

Для полноты: в очень редких случаях вы используете фиксированное обновление в Unity; существует `WaitForFixedUpdate()` который обычно никогда не будет использоваться. Существует определенный вызов (`WaitForEndOfFrame()` в текущей версии Unity), который используется в определенных ситуациях в связи с созданием захвата экрана во время разработки. (Точный механизм немного меняется по мере того, как Unity развивается, поэтому google для получения последней информации, если это необходимо.)

Прочитайте Сопрограммы онлайн: https://riptutorial.com/ru/unity3d/topic/3415/сопрограммы

глава 37: Теги

Вступление

Тег - это строка, которая может применяться для обозначения типов `GameObject` . Таким образом, упрощается идентификация определенных объектов `GameObject` помощью кода.

Тег можно применить к одному или нескольким игровым объектам, но игровой объект всегда будет иметь только один тег. По умолчанию тег *«Untagged»* используется для представления `GameObject` , который не был специально помечен.

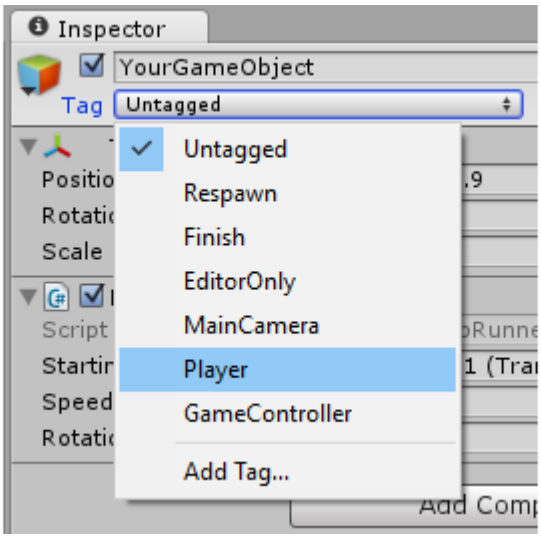
Examples

Создание и применение тегов

Теги обычно применяются через редактор; однако вы также можете применять теги через скрипт. Любой пользовательский тег должен быть создан через окно « *Теги и слои»* перед тем, как его применить к игровому объекту.

Установка тегов в редакторе

При выборе одного или нескольких игровых объектов вы можете выбрать тег из инспектора. Объекты игры всегда будут иметь один тег; по умолчанию игровые объекты будут помечены как *«Без тегов»* . Вы также можете перейти в окно « *Теги и слои»* , выбрав *«Добавить тег ...»* ; однако важно отметить, что это приведет вас только к окну *Tags & Layers* . Любой тэг создается *не* будет автоматически применяться к объекту игры.



Настройка тегов через скрипт

Вы можете напрямую изменить тег игровых объектов через код. Важно отметить, что вы *должны* предоставить тег из списка текущих тегов; если вы добавите тег, который еще не был создан, это приведет к ошибке.

Как указано в других примерах, использование серии `static string` переменных в отличие от ручной записи каждого тега может обеспечить согласованность и надежность.

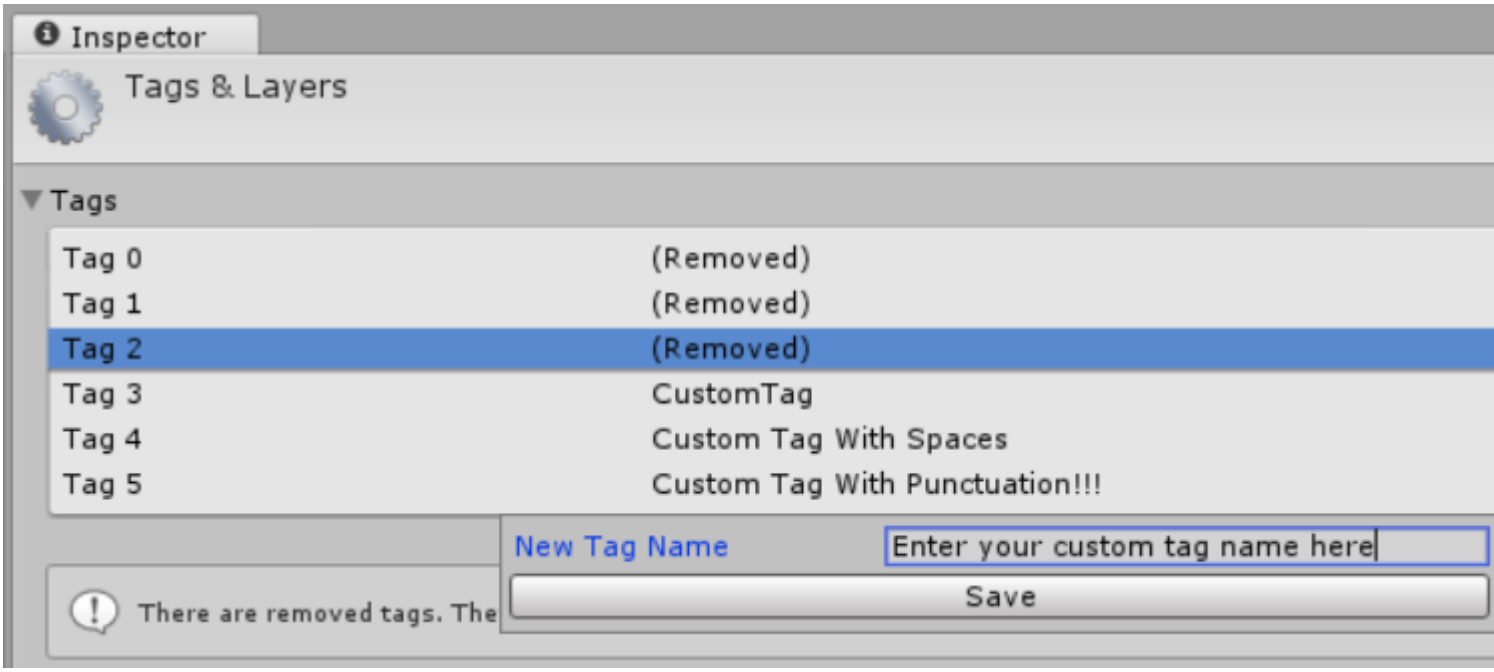
Следующий сценарий демонстрирует, как мы можем изменить ряд тегов игровых объектов, используя `static string` ссылки на `static string` для обеспечения согласованности. Обратите внимание на предположение, что каждая `static string` представляет собой тег, который уже был создан в окне « *Теги и слои»* .

```
using UnityEngine;

public class Tagging : MonoBehaviour
{
    static string tagUntagged = "Untagged";
    static string tagPlayer = "Player";
    static string tagEnemy = "Enemy";

    /// <summary>Represents the player character. This game object should
    /// be linked up via the inspector.</summary>
    public GameObject player;
    /// <summary>Represents all the enemy characters. All enemies should
    /// be added to the array via the inspector.</summary>
    public GameObject[] enemy;

    void Start ()
    {
        // We ensure that the game object this script is attached to
        // is left untagged by using the default "Untagged" tag.
        gameObject.tag = tagUntagged;

        // We ensure the player has the player tag.
        player.tag = tagUntagged;

        // We loop through the enemy array to ensure they are all tagged.
        for(int i = 0; i < enemy.Length; i++)
        {
            enemy[i].tag = tagEnemy;
        }
    }
}
```

Создание пользовательских тегов

Независимо от того, устанавливаете ли вы теги через Inspector или через скрипт, теги *должны* быть объявлены через окно *Tags & Layers* перед использованием. Вы можете

получить доступ к этому окну, выбрав *«Добавить теги ...»* в раскрывающемся меню тега игровых объектов. Кроме того, вы можете найти окно в разделе « **Редактирование»> «Настройки проекта»> «Метки и слои»** .



Просто выберите кнопку `+` , введите нужное имя и выберите « `Сохранить»`, чтобы создать тег. При выборе кнопки `-` удаляется текущий выделенный тег. Обратите внимание, что таким образом тег будет немедленно отображаться как *«(Удалено)»* и будет полностью удален, когда проект будет перезагружен.

Выбор шестерни / зубца в правом верхнем углу окна позволит вам сбросить все пользовательские параметры. Это немедленно удалит все пользовательские теги вместе с любым настраиваемым слоем, который у вас может быть в разделе *«Сортировка слоев»* и *« Слои»* .

Поиск GameObjects по тегу:

Метки делают его особенно удобным для определения определенных игровых объектов. Мы можем искать один игровой объект или искать несколько.

Поиск единого `GameObject`

Мы можем использовать статическую функцию `GameObject.FindGameObjectWithTag(string tag)` для поиска отдельных игровых объектов. Важно отметить, что таким образом игровые объекты не запрашиваются в каком-либо конкретном порядке. Если вы ищете тег, который используется в нескольких игровых объектах в сцене, эта функция не сможет гарантировать, *какой* игровой объект будет возвращен. Таким образом, более удобно, когда мы знаем, что только *один* игровой объект использует такой тег или когда нас не

беспокоит точный экземпляр `GameObject` который возвращается.

```
///<summary>We create a static string to allow us consistency.</summary>
string playerTag = "Player"

///<summary>We can now use the tag to reference our player GameObject.</summary>
GameObject player = GameObject.FindGameObjectWithTag(playerTag);
```

Поиск массива экземпляров `GameObject`

Мы можем использовать статическую функцию `GameObject.FindGameObjectsWithTag(string tag)` чтобы искать *все* игровые объекты, которые используют определенный тег. Это полезно, когда мы хотим итерации через группу определенных игровых объектов. Это также может быть полезно, если мы хотим найти *один* игровой объект, но можем иметь *несколько* игровых объектов с использованием одного и того же тега. Поскольку мы не можем гарантировать точный экземпляр , возвращаемый `GameObject.FindGameObjectWithTag(string tag)` , мы должны вместо этого получить массив всех потенциальных `GameObject` экземпляров с `GameObject.FindGameObjectsWithTag(string tag)` , и далее анализировать полученный массив , чтобы найти экземпляр мы находимся находясь в поиске.

```
///<summary>We create a static string to allow us consistency.</summary>
string enemyTag = "Enemy";

///<summary>We can now use the tag to create an array of all enemy GameObjects.</summary>
GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag );

// We can now freely iterate through our array of enemies
foreach(GameObject enemy in enemies)
{
    // Do something to each enemy (link up a reference, check for damage, etc.)
}
```

Сравнение тегов

При сравнении двух GameObjects по тегам следует отметить, что следующее приведет к сбоям в работе сборщика мусора, поскольку строка создается каждый раз:

```
if (go.Tag == "myTag")
{
    //Stuff
}
```

При выполнении этих сравнений внутри Update () и другого обратного вызова обычного или единственного Unity вы должны использовать этот метод без кучи:

```
if (go.CompareTag("myTag")
{
    //Stuff
```

```
}
```

Кроме того, ваши теги проще хранить в статическом классе.

```
public static class Tags
{
    public const string Player = "Player";
    public const string MyCustomTag = "MyCustomTag";
}
```

Тогда вы можете безопасно сравнивать

```
if (go.CompareTag(Tags.MyCustomTag)
{
    //Stuff
}
```

таким образом, ваши строки тегов генерируются во время компиляции, и вы ограничиваете последствия орфографических ошибок.

Так же, как сохранение тегов в статическом классе, можно также сохранить его в перечислении:

```
public enum Tags
{
    Player, Ennemies, MyCustomTag;
}
```

и затем вы можете сравнить его с помощью метода enum `toString()` :

```
if (go.CompareTag(Tags.MyCustomTag.toString())
{
    //Stuff
}
```

Прочитайте Теги онлайн: https://riptutorial.com/ru/unity3d/topic/5534/теги

глава 38: Трансформации

Синтаксис

- void Transform.Translate (Vector3 translation, Space relativeTo = Space.Self)
- void Transform.Translate (float x, float y, float z, Space relativeTo = Space.Self)
- void Transform.Rotate (Vector3 eulerAngles, Space relativeTo = Space.Self)
- void Transform.Rotate (float xAngle, float yAngle, float zAngle, Space relativeTo = Space.Self)
- void Transform.Rotate (ось Vector3, угол поплавка, космос relativeTo = Space.Self)
- void Transform.RotateAround (Vector3 point, Vector3 axis, float angle)
- void Transform.LookAt (Transform target, Vector3 worldUp = Vector3.up)
- void Transform.LookAt (Vector3 worldPosition, Vector3 worldUp = Vector3.up)

Examples

обзор

Преобразования содержат большинство данных об объекте в единстве, включая его родителя (ов), ребенка (ов), положение, поворот и масштаб. Он также имеет функции для изменения каждого из этих свойств. Каждый GameObject имеет Transform.

Перевод (перемещение) объекта

```
// Move an object 10 units in the positive x direction
transform.Translate(10, 0, 0);

// translating with a vector3
vector3 distanceToMove = new Vector3(5, 2, 0);
transform.Translate(distanceToMove);
```

Поворот объекта

```
// Rotate an object 45 degrees about the Y axis
transform.Rotate(0, 45, 0);

// Rotates an object about the axis passing through point (in world coordinates) by angle in
degrees
transform.RotateAround(point, axis, angle);
// Rotates on it's place, on the Y axis, with 90 degrees per second
transform.RotateAround(Vector3.zero, Vector3.up, 90 * Time.deltaTime);

// Rotates an object to make it's forward vector point towards the other object
transform.LookAt(otherTransform);
// Rotates an object to make it's forward vector point towards the given position (in world
coordinates)
transform.LookAt(new Vector3(10, 5, 0));
```

Более подробную информацию и примеры можно найти в документации Unity .

Также обратите внимание, что если игра использует жесткие тела, то преобразование не должно взаимодействовать напрямую (если только твердое тело не имеет `isKinematic == true`). В этом случае используйте AddForce или другие подобные методы, чтобы действовать прямо на жесткое тело.

Воспитание детей и детей

Unity работает с иерархиями, чтобы поддерживать ваш проект. Вы можете назначить объекты в иерархии с помощью редактора, но вы также можете сделать это с помощью кода.

Воспитание

Вы можете установить родитель объекта следующими способами

```
var other = GetOtherGameObject();
other.transform.SetParent( transform );
other.transform.SetParent( transform, worldPositionStays );
```

Всякий раз, когда вы устанавливаете родительский элемент transforms, он будет удерживать позицию объектов как мировое положение. Вы можете сделать эту позицию относительной, передав *false* для параметра *worldPositionStays* .

Вы также можете проверить, является ли объект дочерним по отношению к другому преобразованию следующим способом

```
other.transform.IsChildOf( transform );
```

Получение ребенка

Так как объекты могут быть отслежены друг от друга, вы также можете найти детей в иерархии. Самый простой способ сделать это - использовать следующий метод

```
transform.Find( "other" );
transform.FindChild( "other" );
```

Примечание: вызовы FindChild Найдите под капотом

Вы также можете искать детей по иерархии. Вы делаете это, добавляя «/», чтобы указать уровень глубже.

```
transform.Find( "other/another" );
transform.FindChild( "other/another" );
```

Другой способ извлечения ребенка - использовать GetChild

```
transform.GetChild( index );
```

> Для GetChild требуется целое число как индекс, который должен быть меньше общего числа детей

```
int count = transform.childCount;
```

Изменение индекса Sibling

Вы можете изменить порядок детей GameObject. Вы можете сделать это, чтобы определить порядок рисования для детей (при условии, что они находятся на одном уровне Z и тот же порядок сортировки).

```
other.transform.SetSiblingIndex( index );
```

Вы также можете быстро установить индекс sibling для первого или последнего, используя следующие методы

```
other.transform.SetAsFirstSibling();
other.transform.SetAsLastSibling();
```

Отсоединение всех детей

Если вы хотите освободить всех детей преобразования, вы можете сделать это:

```
foreach(Transform child in transform)
{
    child.parent = null;
}
```

Кроме того, Unity предоставляет метод для этой цели:

```
transform.DetachChildren();
```

В принципе, и looping, и `DetachChildren()` устанавливают родителям детей с глубиной до нуля, что означает, что у них не будет родителей.

(дети первой глубины: преобразования, непосредственно связанные с преобразованием)

Прочитайте Трансформации онлайн: https://riptutorial.com/ru/unity3d/topic/2190/трансформации

глава 39: физика

Examples

Rigidbodies

обзор

Компонент Rigidbody дает GameObject *физическое присутствие* на сцене в том, что он способен реагировать на силы. Вы можете применить силы непосредственно к GameObject или позволить ему реагировать на внешние силы, такие как гравитация, или другой Rigidbody, ударяющий по нему.

Добавление компонента Rigidbody

Вы можете добавить Rigidbody, нажав **Component> Physics> Rigidbody**

Перемещение объекта Rigidbody

Рекомендуется, если вы примените Rigidbody к GameObject, который использует силы или крутящий момент, чтобы переместить его, а не манипулировать его Transform. Для этого используйте `AddForce()` или `AddTorque()` :

```
// Add a force to the order of myForce in the forward direction of the Transform.
GetComponent<Rigidbody>().AddForce(transform.forward * myForce);

// Add torque about the Y axis to the order of myTurn.
GetComponent<Rigidbody>().AddTorque(transform.up * torque * myTurn);
```

масса

Вы можете изменить массу объекта GameGuide, чтобы повлиять на то, как он реагирует на другие Rigidbodies и силы. Более высокая масса означает, что GameObject будет иметь большее влияние на другие основанные на физике GameObjects и потребует большей силы для перемещения. Объекты различной массы будут падать с одинаковой скоростью, если они имеют одинаковые значения перетаскивания. Чтобы изменить массу в коде:

```
GetComponent<Rigidbody>().mass = 1000;
```

Тащить, тянуть

Чем выше значение перетаскивания, тем больше объект будет замедляться при перемещении. Подумайте об этом как о противостоянии. Чтобы изменить перетаскивание кода:

```
GetComponent<Rigidbody>().drag = 10;
```

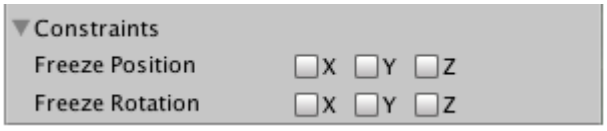
isKinematic

Если вы отметите Rigidbody как **Kinematic,** то это не может быть затронуто другими силами, но все равно может повлиять на другие GameObjects. Изменить код:

```
GetComponent<Rigidbody>().isKinematic = true;
```

Ограничения

Также возможно добавить ограничения для каждой оси, чтобы заморозить положение или вращение Rigidbody в локальном пространстве. По умолчанию используется `RigidbodyConstraints.None` как показано здесь:



Пример ограничений в коде:

```
// Freeze rotation on all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeRotation

// Freeze position on all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePosition

// Freeze rotation and motion an all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeAll
```

Вы можете использовать побитовый оператор OR `|` для объединения нескольких ограничений:

```
// Allow rotation on X and Y axes and motion on Y and Z axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePositionZ |
    RigidbodyConstraints.FreezeRotationX;
```

Столкновения

Если вы хотите, чтобы GameObject с Rigidbody на нем реагировал на столкновения, вам также нужно добавить к нему коллайдер. Типы коллайдера:

- Коллайдер коробки
- Сферный коллайдер
- Капсульный коллайдер
- Коллайдер колес
- Мешевый коллайдер

Если вы применяете несколько коллайдеров к GameObject, мы называем это сложным коллайдером.

Вы можете сделать коллайдер в **триггере** , чтобы использовать `OnTriggerEnter()` , `OnTriggerStay()` и `OnTriggerExit()` . Триггерный коллайдер физически не реагирует на столкновения, другие GameObjects просто проходят через него. Они полезны для обнаружения, когда другой GameObject находится в определенной области или нет, например, при сборе элемента, мы можем захотеть просто запустить его, но обнаружить, когда это произойдет.

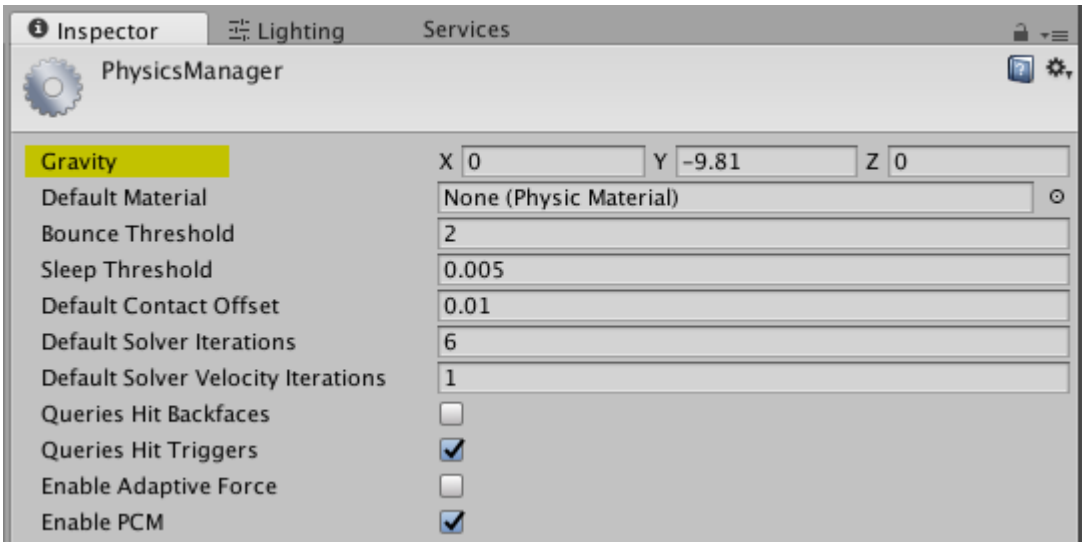
Гравитация в жестком теле

Свойство `useGravity` `RigidBody` контролирует, влияет ли на него гравитация или нет. Если установлено значение `false` `RigidBody` будет вести себя так, как если бы в космическом пространстве (без постоянной силы, применяемой к нему в некотором направлении).

```
GetComponent<RigidBody>().useGravity = false;
```

Это очень полезно в ситуациях, когда вам нужны все другие свойства `RigidBody` кроме движения, управляемого гравитацией.

Когда включено, на `RigidBody` будет влиять гравитационная сила, созданная в разделе « `Physics Settings` :



Гравитация определяется в мировых единицах в секунду в квадрате и вводится здесь как трехмерный вектор: это означает, что с настройками в изображении примера все `RigidBodies` с свойством `useGravity` установленным в `True` будут испытывать силу в 9,81 единиц в секунду *в секунду* в направлении вниз (как отрицательные значения Y в системе координат Unity вниз).

Прочитайте физика онлайн: https://riptutorial.com/ru/unity3d/topic/3680/физика

глава 40: Шаблоны проектирования

Examples

Модельный шаблон контроллера модели (MVC)

Контроллер представления модели является очень распространенным шаблоном проектирования, который существует уже довольно давно. Этот шаблон фокусируется на сокращении кода *спагетти* путем разделения классов на функциональные части. Недавно я экспериментировал с этим шаблоном проектирования в Unity и хотел бы изложить основной пример.

Конструкция MVC состоит из трех основных частей: модели, вида и контроллера.

Модель: модель представляет собой класс, представляющий часть данных вашего объекта. Это может быть игрок, инвентарь или весь уровень. Если вы правильно запрограммировали, вы сможете использовать этот скрипт и использовать его за пределами Unity.

Обратите внимание на несколько вещей о модели:

- Он не должен наследовать от Monobehaviour
- Он не должен содержать специальный код Unity для переносимости
- Поскольку мы избегаем вызовов Unity API, это может помешать вещам вроде неявных преобразователей в классе Model (обходные пути необходимы)

Player.cs

```
using System;

public class Player
{
    public delegate void PositionEvent(Vector3 position);
    public event PositionEvent OnPositionChanged;

    public Vector3 position
    {
        get
        {
            return _position;
        }
        set
        {
            if (_position != value) {
                _position = value;
                if (OnPositionChanged != null) {
                    OnPositionChanged(value);
                }
```

```
            }
        }
    }
    private Vector3 _position;
}
```

Vector3.cs

Пользовательский класс Vector3 для использования с нашей моделью данных.

```
using System;

public class Vector3
{
    public float x;
    public float y;
    public float z;

    public Vector3(float x, float y, float z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

Вид: представление представляет собой класс, представляющий часть просмотра, привязанную к модели. Это подходящий класс для выхода из Monobehaviour. Это должно содержать код, который взаимодействует напрямую с отдельными API-интерфейсами Unity, включая `OnCollisinEnter` , `Start` , `Update` и т. Д. ...

- Обычно наследуется от Monobehaviour
- Содержит специальный код Unity

PlayerView.cs

```
using UnityEngine;

public class PlayerView : Monobehaviour
{
    public void SetPosition(Vector3 position)
    {
        transform.position = position;
    }
}
```

Контроллер: контроллер - это класс, который связывает вместе и модель, и представление. Контроллеры поддерживают синхронизацию модели и представления, а также взаимодействие с приводом. Контроллер может прослушивать события от одного из партнеров и обновлять соответственно.

- Привязывает как модель, так и представление по состоянию синхронизации

- Может управлять взаимодействием между партнерами
- Контроллеры могут быть или не быть переносимыми (вам может потребоваться использовать код Unity здесь)
- Если вы решили не переносить свой контроллер портативным, подумайте над тем, чтобы сделать его Monobehaviour, чтобы помочь с проверкой редактора

PlayerController.cs

```
using System;

public class PlayerController
{
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public PlayerController(Player model, PlayerView view)
    {
        this.model = model;
        this.view = view;

        this.model.OnPositionChanged += OnPositionChanged;
    }

    private void OnPositionChanged(Vector3 position)
    {
        // Sync
        Vector3 pos = this.model.position;

        // Unity call required here! (we lost portability)
        this.view.SetPosition(new UnityEngine.Vector3(pos.x, pos.y, pos.z));
    }

    // Calling this will fire the OnPositionChanged event
    private void SetPosition(Vector3 position)
    {
        this.model.position = position;
    }
}
```

Конечное использование

Теперь, когда у нас есть все основные части, мы можем создать завод, который будет генерировать все три части.

PlayerFactory.cs

```
using System;

public class PlayerFactory
{
    public PlayerController controller { get; private set; }
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public void Load()
```

```
    {
        // Put the Player prefab inside the 'Resources' folder
        // Make sure it has the 'PlayerView' Component attached
        GameObject prefab = Resources.Load<GameObject>("Player");
        GameObject instance = GameObject.Instantiate<GameObject>(prefab);
        this.model = new Player();
        this.view = instance.GetComponent<PlayerView>();
        this.controller = new PlayerController(model, view);
    }
}
```

И, наконец, мы можем вызвать фабрику у менеджера ...

Manager.cs

```
using UnityEngine;

public class Manager : Monobehaviour
{
    [ContextMenu("Load Player")]
    private void LoadPlayer()
    {
        new PlayerFactory().Load();
    }
}
```

Прикрепите скрипт Manager к пустому GameObject в сцене, щелкните правой кнопкой мыши компонент и выберите «Загрузить проигрыватель».

Для более сложной логики вы можете ввести наследование с абстрактными базовыми классами и интерфейсами для улучшенной архитектуры.

Прочитайте Шаблоны проектирования онлайн: https://riptutorial.com/ru/unity3d/topic/10842/шаблоны-проектирования

кредиты

S. No	Главы	Contributors
1	Начало работы с unity3d	Alexey Shimansky, Chris McFarland, Community, Desutoroiya, driconmax, Ḟ̄ạḿíṅḡ ỏḿƀíé, James Radvan, josephsw, Linus Juhlin, Luís Fonseca, Maarten Bicknese, martinhodler, matiaslauriti, Mike B, Minzkraut, PlanetVaster, R.K123, S. Tarık Çetin, Skyblade, SourabhV, SP., tenpn, tim , user3071284
2	API CullingGroup	volvis
3	Prefabs	Brandon Mintern, Dávid Florek, Ḟ̄ạḿíṅḡ ỏḿƀíé, gman, Gnemlock, Guglie, James Radvan, Jean Vitor, josephsw, Lich, matiaslauriti, Skyblade, Thulani Chivandikwa, ɬolɐəz əɥɬ qoq, Woltus, yummypasta
4	Raycast	driconmax, Meinkraft, Skyblade, user3570542, volvis, wouterrobot
5	ScriptableObject	volvis
6	Unity Animation	4444, Fiery Raccoon, Guglie
7	Unity Profiler	Amitayu Chakraborty, ForceMagic, RamenChef, Skyblade
8	Vector3	driconmax, Ḟ̄ạḿíṅḡ ỏḿƀíé, Gnemlock
9	Атрибуты	4444, Thundernerd
10	Аудио система	R4mbi, ɬolɐəz əɥɬ qoq
11	Виртуальная реальность (VR)	4444, Airwarfare, Guglie, pew., Pratham Sehgal, tim
12	Входная система	Programmer, Skyblade, ɬolɐəz əɥɬ qoq
13	Выполнение класса MonoBehaviour	matiaslauriti, Skyblade, Thundernerd, user3797758
14	Графическая система пользовательского интерфейса немедленного режима	Skyblade, Soaring Code

	(IMGUI)	
15	Импортеры и (пост) процессоры	gman, Skyblade, volvis
16	Интеграция рекламы	ɬolɐəz əɥɬ qoq
17	Использование контроля источника Git с Unity	Commodore Yournero, Hacky, James Radvan, matiaslauriti, Max Yankov, Maxim Kamalov, Pierrick Bignet, Ricardo Amores, S. Tarık Çetin, S.Richmond, Skyblade, Thulani Chivandikwa, YsenGrimm, yummypasta
18	Как использовать пакеты активов	Ḟ̄ąḿíṅḡ ỏḿƀíé
19	Кватернионы	matiaslauriti, Tiziano Coroneo, Xander Luciano, yummypasta
20	коллизия	Ḟ̄ąḿíṅḡ ỏḿƀíé, jjhavokk, Xander Luciano
21	Магазин активов	JakeD, Trent, zwcloud
22	Мобильные платформы	Airwarfare, Skyblade
23	Мультиплатформенная разработка	user3797758, volvis
24	Объединение объектов	Chris McFarland, Ed Marty, lase, matiaslauriti, S. Tarık Çetin , Thulani Chivandikwa, Thundernerd, ɬolɐəz əɥɬ qoq, volvis
25	оптимизация	Ed Marty, EvilTak, Ḟ̄ąḿíṅḡ ỏḿƀíé, Grigory, JohnTube, Skyblade, Thulani Chivandikwa, volvis
26	Освещение единства	Ḟ̄ąḿíṅḡ ỏḿƀíé
27	Плагины для Android 101 - Введение	Venkat at Axiom Studios
28	Поиск и сбор GameObjects	Pierrick Bignet, S. Tarık Çetin, volvis
29	Пользовательский интерфейс (UI)	Hellium, matiaslauriti, Maxim Kamalov, Programmer, RamenChef, Skyblade, Umair M
30	Расширение редактора	Pierrick Bignet, Skyblade, Thundernerd, ɬolɐəz əɥɬ qoq, volvis

31	Ресурсы	glaubergft, MadJlzz, Skyblade, Venkat at Axiom Studios
32	Связь с сервером	David Martinez, devon t, Ḟ̄ą́ḿíṅḡ ỏḿƀíé, Maxim Kamalov, tim
33	сетей	David Martinez, driconmax, Rafiwui, RamenChef
34	Синглтоны в единстве	David Darias, Fehr, James Radvan, JohnTube, matiaslauriti, Maxim Kamalov, Simon Heinen, SP., Tiziano Coroneo, Umair M, volvis, Zze,
35	Слои	Arijoon, dreadnought, Light Drake, RamenChef, Skyblade
36	Сопрограммы	agiro, Fattie, Fehr, Giuseppe De Francesco, Problematic, Skyblade, Thulani Chivandikwa, Thundernerd, ɬolɐəz əɥɬ qoq, volvis
37	Теги	Arijoon, Augure, glaubergft, Gnemlock, MadJlzz, Skyblade, Trent
38	Трансформации	ADB, Jean Vitor, matiaslauriti, S. Tarık Çetin, Skyblade, Thundernerd, Xander Luciano
39	физика	eunoia, Ḟ̄ą́ḿíṅḡ ỏḿƀíé, jack jay
40	Шаблоны проектирования	Ian Newland