Урок 7

Урок 7

Работа с классом Editor

Работа с классом Editor.

Введение

Собственные шаблоны скриптов

Gizmos

Пользовательские редакторы

Базовые понятия

Атрибуты для кастомизации переменных в инспекторе

Реализация собственных атрибутов в .Net

Реализация собственных атрибутов в Unity

Добавление своего пункта в меню

Создание своего окна

Расширение скриптов

RayCast в редакторе

Editor в пользовательских скриптах

Практическое задание

Дополнительные материалы

Используемая литература

На этом уроке:

1. Познакомимся с расширением редактора
2. Научимся использовать атрибуты и создавать свои

Введение

На этом уроке разберем такую важную тему как пользовательский редактор.

Изменение пользовательского редактора особенно востребовано и полезно, если вы работаете в команде. Когда вы напишете удобный интерфейс для изменения компонентов или их взаимодействия с объектами, для вызова самописных функций в редакторе, будьте уверены – гейм-дизайнер или моделлер будут благодарны. Специалистам не из сферы программирования некомфортно «погружаться» в скрипт и выяснять, зачем нужны переменные, какой объект вставить и так далее. Кроме того, с помощью редакторских скриптов можно обезопасить программу от ввода некорректных данных.

Собственные шаблоны скриптов

При создании нового скрипта в окне **Project** для него можно выбрать один из существующих шаблонов. Если вы хотите, чтобы в этом скрипте уже был написан определенный код, то шаблоны можно заранее отредактировать или создать свои.

Шаблоны обычно расположены в следующих папках:

Windows: \Program Files\Unity\Editor\Data\Resources\ScriptTemplates

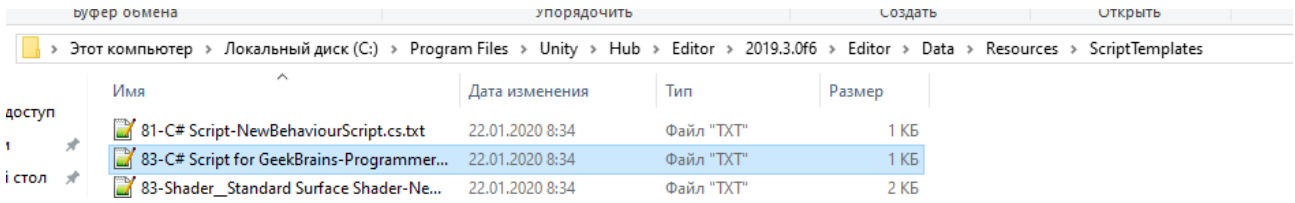
OS X: /Applications/Unity/Unity.app/Contents/Resources/ScriptTemplates

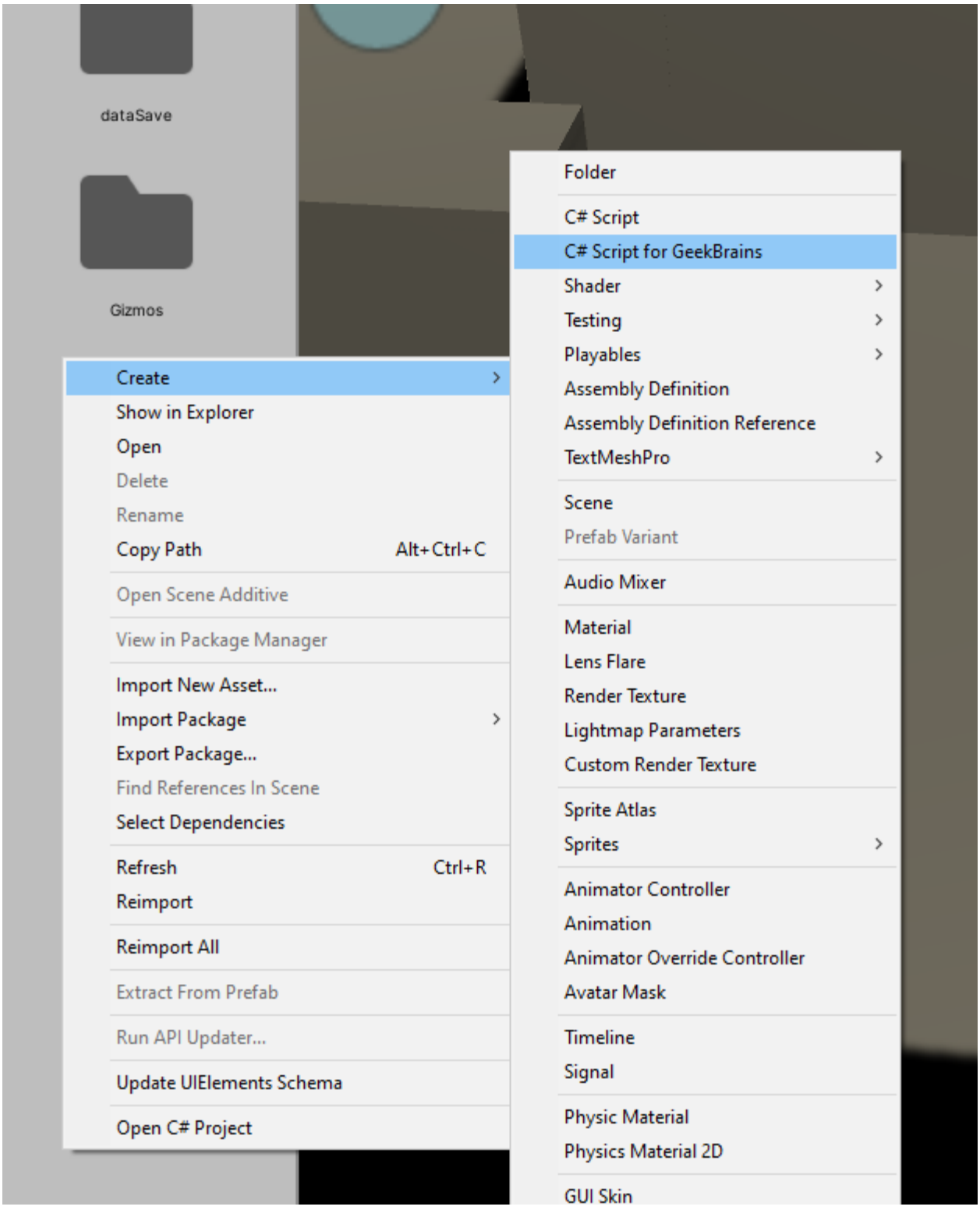
Опишем параметры в имени шаблона «83-C# Script for GeekBrains- ProgrammerIsTheBest .cs»:

«83» – порядковый номер шаблона в контекстном меню;

«C# Script for GeekBrains» – отображаемое имя в контекстном меню;

«ProgrammerIsTheBest» – дефолтное имя созданного скрипта.



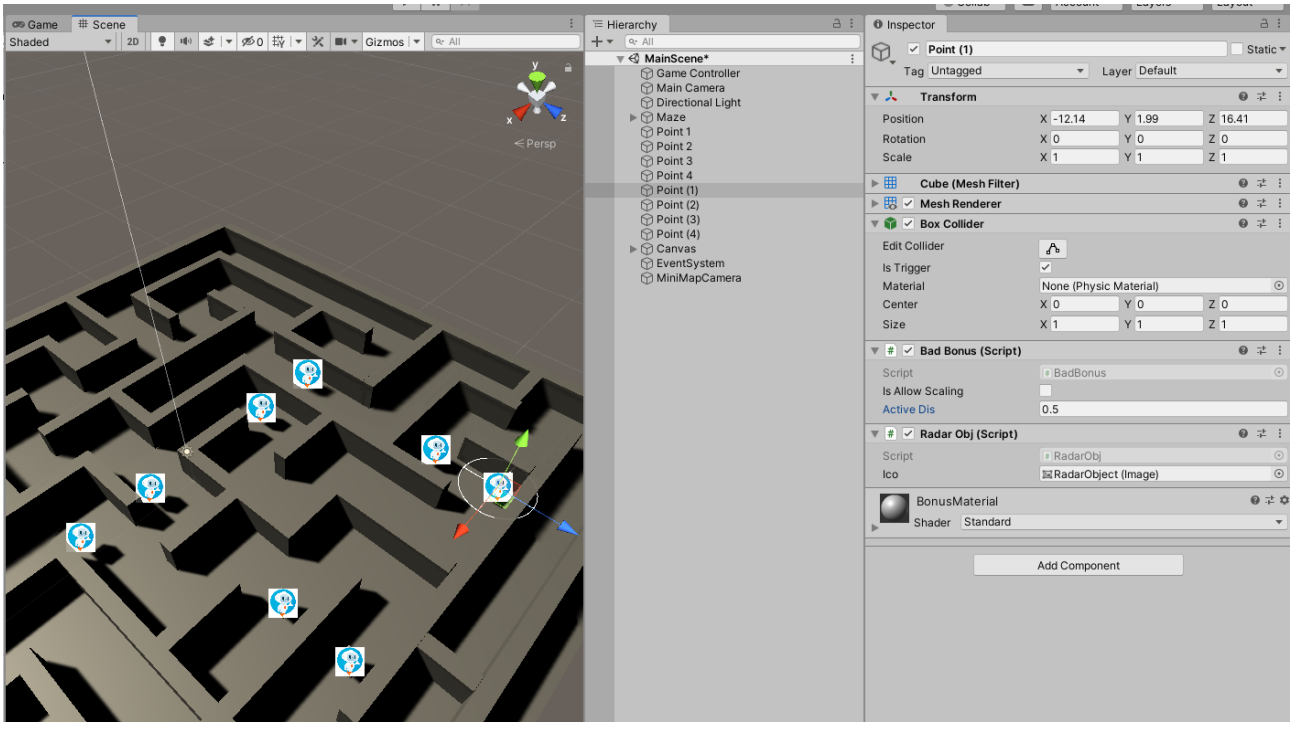


Gizmos

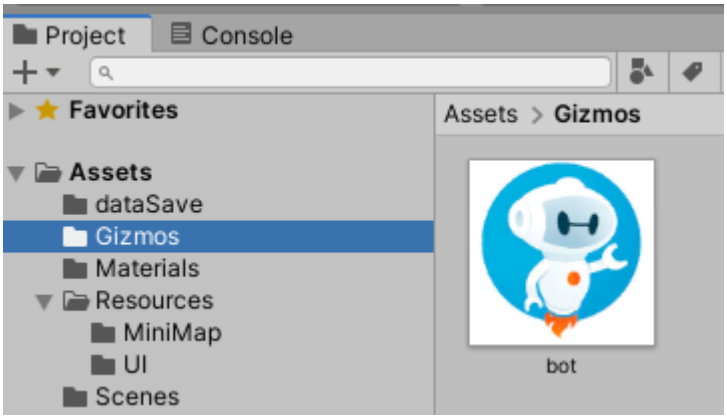
С помощью класса Gizmos можно отрисовывать вспомогательные линии, прорисовку иконки возле объекта и т.д.

```
using UnityEngine;
using static UnityEngine.Random;
```

```
namespace Geekbrains
{
    public abstract class InteractiveObject : MonoBehaviour, IExecute
    {
        [SerializeField] private bool _isAllowScaling;
        [SerializeField] private float ActiveDis;

        //…….

        private void OnDrawGizmos()
        {
            Gizmos.DrawIcon(transform.position, "bot.jpg", _isAllowScaling);
        }

        private void OnDrawGizmosSelected()
        {
            #if UNITY_EDITOR
            Transform t = transform;

            //Gizmos.matrix = Matrix4x4.TRS(t.position, t.rotation, t.localScale);
            //Gizmos.DrawWireCube(Vector3.zero, Vector3.one);

            var flat = new Vector3(ActiveDis, 0, ActiveDis);
            Gizmos.matrix = Matrix4x4.TRS(t.position, t.rotation, flat);
            Gizmos.DrawWireSphere(Vector3.zero, 5);
            #endif
        }
    }
}
```



Чтобы иконка прорисовалась, необходимо поместить изображение в специализированную папку **Assets\Gizmos.**



Пользовательские редакторы

Базовые понятия

Редакторские скрипты обязательно должны находиться в папке **Assets\Editor.** Выдержка из документации:

«Вы можете выбирать любые имена для папок проекта. Но в Unity зарезервирован ряд имен, которые указывают на то, что содержащийся в папках контент имеет специальное назначение. Некоторые из них влияют на порядок компиляции скриптов. У нее есть четыре отдельные фазы, и порядок их компилирования определяется родительской папкой.

Это имеет большое значение в случае, если скрипт должен обратиться к классам, определенным в других скриптах. Основное правило заключается в том, чтобы не было ссылок на скрипты, которые компилируются в фазе «после». Все, что компилируется в текущей или ранее выполненной фазе, должно быть полностью доступно.

Фазы компиляции:

Фаза 1: Выполняются скрипты из папок с именами Standard Assets, Pro Standard Assets и Plugins;

Фаза 2: Скрипты редактора в папках с названием Standard Assets/Editor, Pro Standard Assets/Editor и Plugins/Editor;

Фаза 3: Все прочие скрипты, не находящиеся в папке Editor;

Фаза 4: Все оставшиеся скрипты (находящиеся в папке Editor)».

Редакторские скрипты позволяют расширить базовый функционал редактора. В этом разделе мы рассмотрим:

1. Атрибуты для кастомизации переменных в инспекторе;
2. Добавление своего пункта в меню;
3. Создание своего окна;
4. Расширение скриптов.

Атрибуты для кастомизации переменных в инспекторе

Атрибуты, которые мы рассмотрим, делают работу с пользовательскими классами в инспекторе Unity более комфортной. Они не требуют написания редакторского скрипта и указываются в том классе, который необходимо расширить.

Такой код позволяет скрыть переменную в инспекторе:

```
[HideInInspector]
public int Variable = 0;
```

А **SerializeField,** наоборот, показывает переменную в инспекторе:

```
[SerializeField]
private int Variable = 0;
```

HeaderAttribute добавляет перед переменной заголовок с заданным текстом:

```
[Header("Header text")]
public int Variable = 0;
```

RangeAttribute позволяет установить диапазон, в котором может изменяться значение числовой переменной. Первый параметр – минимальное значение, второй – максимальное. Редактировать значение этой переменной в инспекторе можно будет с помощью слайдера:

```
[Range(10, 30)]
public int Variable;
```

SpaceAttribute добавляет отступ между двумя переменными. В качестве параметра необходимо указать высоту отступа в пикселях:

```
[Space(20)]
public int Variable = 0;
```

MultilineAttribute предназначен для ввода текста, состоящего из нескольких строк. Высота поля ввода задается в строках:

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

TextAreaAttribute – продвинутая альтернатива **MultilineAttribute**. Позволяет указать минимальную и максимальную высоту. Кроме того, поле ввода будет занимать всю ширину инспектора, а при превышении максимального числа строк появится скроллбар:

```
[TextArea(3, 5)]
public string Variable;
```

TooltipAttribute добавляет подсказку, появляющуюся при наведении курсора мыши:

```
[Tooltip("Tooltip text")]
public int Variable = 0;
```

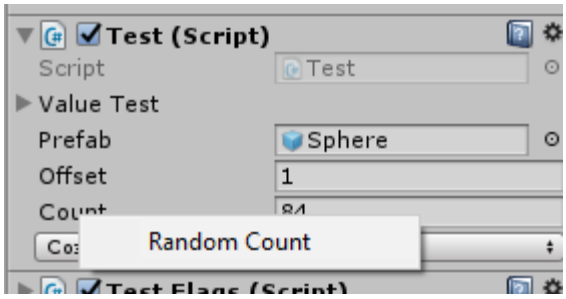
Чтобы класс отобразился в инспекторе как поле другого класса, нужно добавить к нему атрибут **[System.Serializable]**. Если вместе с компонентом нужно обязательно добавить еще один или несколько, то используем атрибут **[RequireComponent(typeof (Нужный компонент))]**.

По умолчанию **MonoBehaviours** выполняются только в режиме воспроизведения. При добавлении атрибута **ExecuteInEditMode** любой экземпляр **MonoBehaviour** будет выполнять функции обратного вызова, пока редактор не находится в режиме воспроизведения:

```
[ExecuteInEditMode]
public class Test  : MonoBehaviour
{
   private void Start()
   {
      var tempRenderer = GetComponent<Renderer>();
      if (tempRenderer != null) tempRenderer.material.color = Random.ColorHSV();
   }
}
```

Атрибут **ContextMenuItem** добавляет контекстное меню в поле, которое вызывает именованный метод:

```
[ContextMenuItem("Random Count", nameof(Random))]
[SerializeField] private int _count;
private void Random()
{
 _count = UnityEngine.Random.Range(0, 100);
}
```



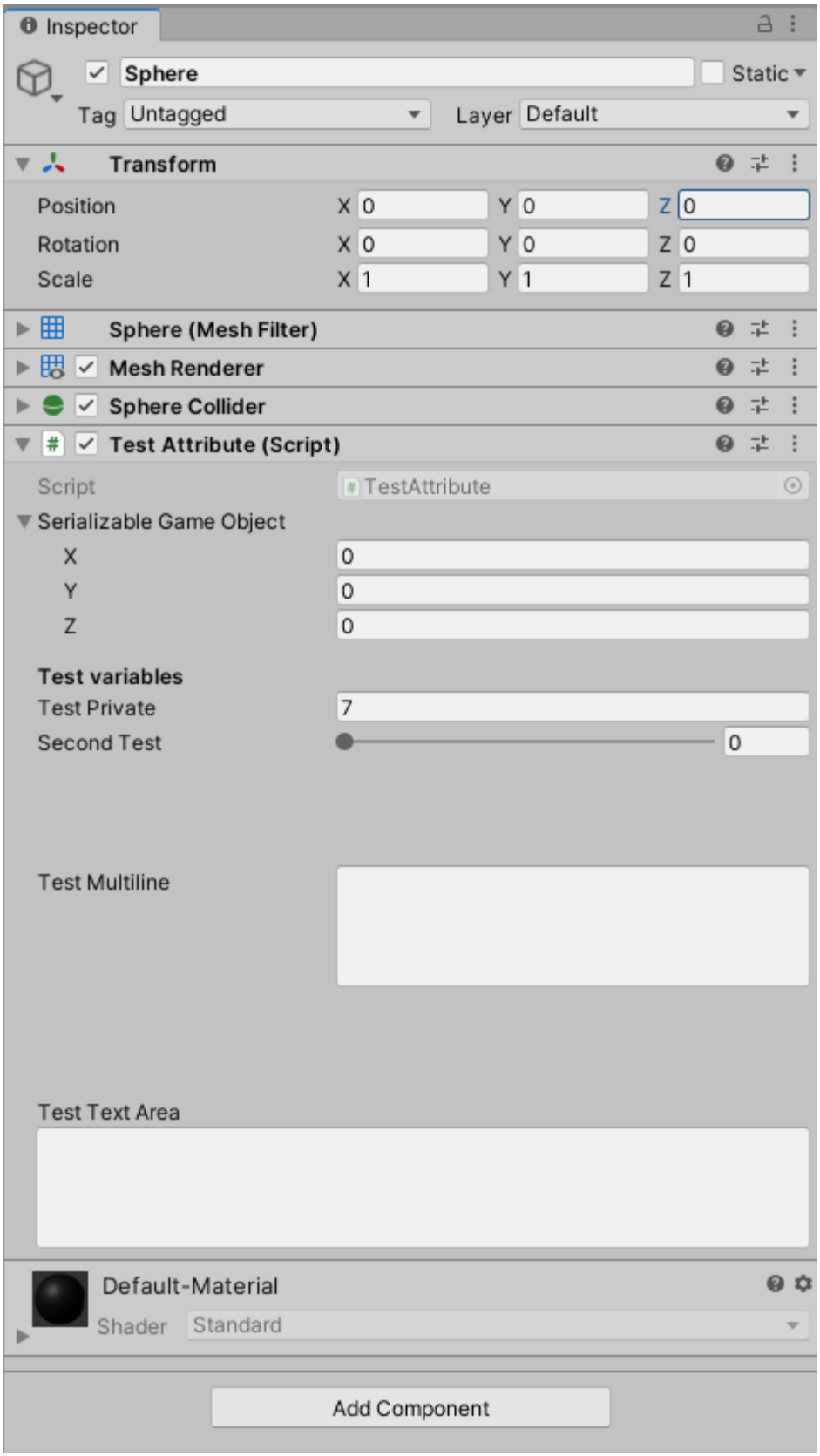
Атрибутом **Obsolete** отмечается не рекомендуемая к использованию сущность программы. Каждый случай использования сущности, отмеченной как устаревшая, будет приводить к генерированию предупреждения или ошибки – в зависимости от настроек этого атрибута:

```
[Obsolete("Устарело. Используй что-то другое")]
private void TestObsolete()
```

```
{
}
```

Демонстрация перечисленных атрибутов

```
using System;
using UnityEngine;

namespace Geekbrains
{
   [RequireComponent(typeof(Renderer)), ExecuteInEditMode]
   public sealed class TestAttribute : MonoBehaviour
   {
      [HideInInspector] public float TestPublic;

      public Vector3Serializable SerializableGameObject;
      private const int _min = 0;
      private const int _max = 100;
      [Header("Test variables")]
      [ContextMenuItem("Randomize Number", nameof(Randomize))]
      [SerializeField] private float _testPrivate = 7;

      [Range(_min, _max)]
      public int SecondTest;

      [Space(60)]
      [SerializeField, Multiline(5)] private string _testMultiline;
      [Space(60)]
      [SerializeField, TextArea(5, 5), Tooltip("Tooltip text")] private string
_testTextArea;


        private void Update()
        {
            GetComponent<Renderer>().sharedMaterial.color =
UnityEngine.Random.ColorHSV();
        }

        private void Randomize()
      {
         _testPrivate = UnityEngine.Random.Range(_min, _max);
      }

      [Obsolete("Устарело. Используй что-то другое")]
      private void TestObsolete()
      {
      }
   }
}
```



Это основные атрибуты, которые чаще всего применяются при разработке игр. Помимо использования встроенных атрибутов, доступно и создание собственных.

Реализация собственных атрибутов в .Net

Атрибуты в .NET представляют специальные инструменты, которые позволяют встраивать в сборку дополнительные метаданные. Атрибуты могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.). Основу атрибутов составляет класс System.Attribute, от которого образованы все остальные классы атрибутов. С помощью атрибута AttributeUsage можно ограничить типы, к которым будет применяться атрибут.
Ограничение задает перечисление AttributeTargets, которое может принимать еще ряд значений:

- All: используется всеми типами
- Assembly: атрибут применяется к сборке
- Constructor: атрибут применяется к конструктору
- Delegate: атрибут применяется к делегату
- Enum: применяется к перечислению
- Event: атрибут применяется к событию
- Field: применяется к полю типа
- Interface: атрибут применяется к интерфейсу
- Method: применяется к методу
- Property: применяется к свойству
- Struct: применяется к структуре

С помощью логической операции ИЛИ можно комбинировать эти значения. Например, пусть атрибут может применяться к классам и структурам: [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]

Реализация собственных атрибутов в Unity

Если вам недостаточно атрибутов приведенных выше, вы всегда можете воспользоваться API для написания собственных настраиваемых атрибутов. Реализация данного инструмента также достаточно проста и заключается в нескольких шагах.

Во-первых, нужно определить класс-наследник от стандартного класса PropertyAttribute.
Во-вторых, необходимо создать скрипт редактора, в котором будет рисоваться данный класс. Его нужно унаследовать от PropertyDrawer, а также написать к нему атрибут CustomPropertyDrawer

Добавление своего пункта в меню

Перейдем к созданию редакторских скриптов. Создаем новый класс **MenuItems**, помещаем его в пространство имен Geekbrains.Editor.

```
using UnityEditor;

namespace Geekbrains
{
   public class MenuItems
```

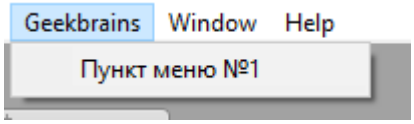
```
    {
        [MenuItem("Geekbrains/Пункт меню №1 ")]
        private static void MenuOption()
        {
        }
    }
}
```

Функция, к которой добавлен атрибут, должна быть статической!

Параметры атрибута:

1. Путь расположения пункта меню;
2. Флаг группировки пунктов, схожих по названию;
3. Приоритет пункта.

Путь расположения пункта меню может быть любым: можно добавлять свой пункт в уже существующий или создавать новый пункт или подпункт меню.



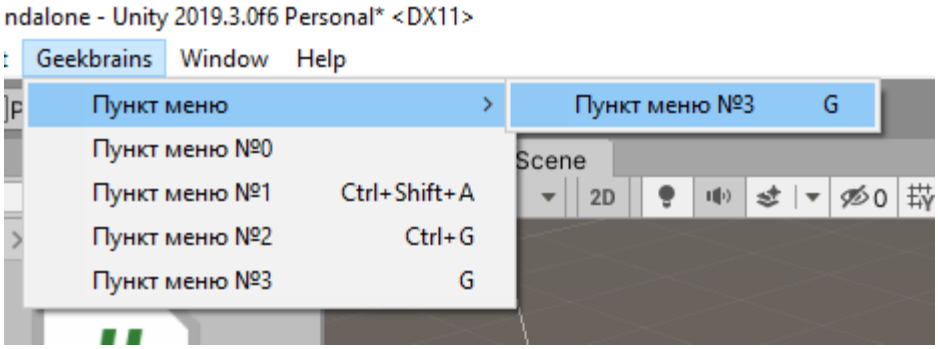
Можно добавлять горячие клавиши для вызова пункта меню:

%	CTRL on Windows / CMD on OSX
#	Shift
&	Alt
LEFT/RIGHT/UP/DOWN	Arrow keys
F1…F2	F keys

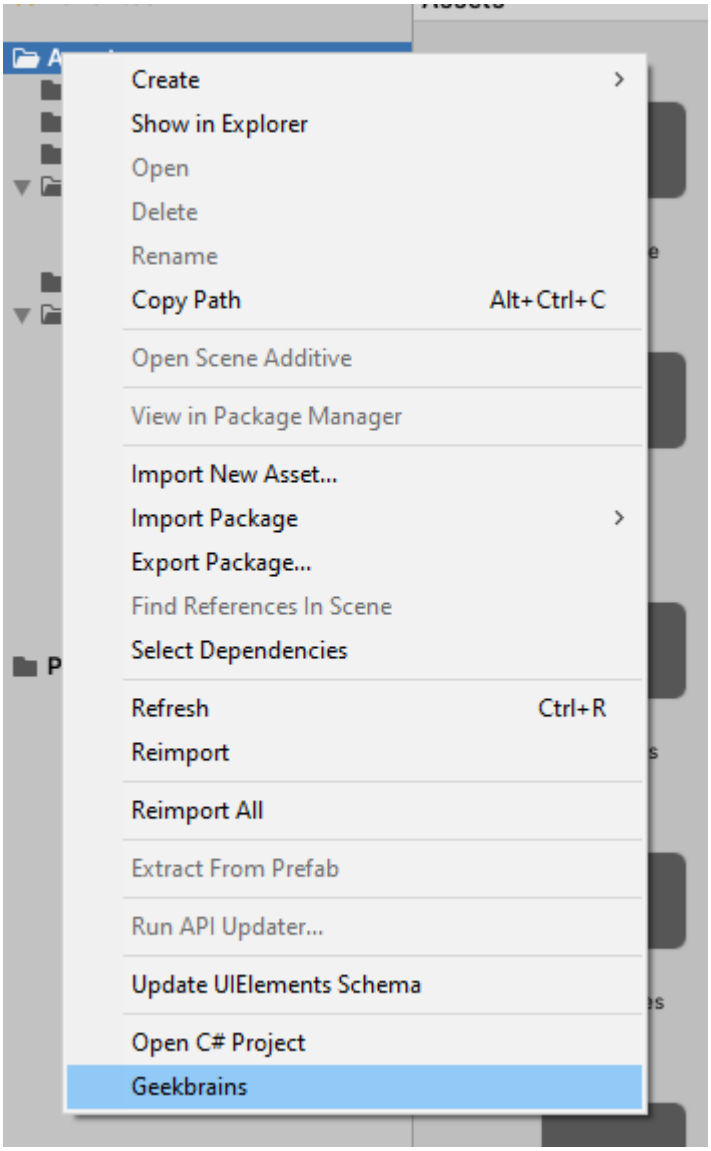
Добавим горячие клавиши к нашим пунктам меню:

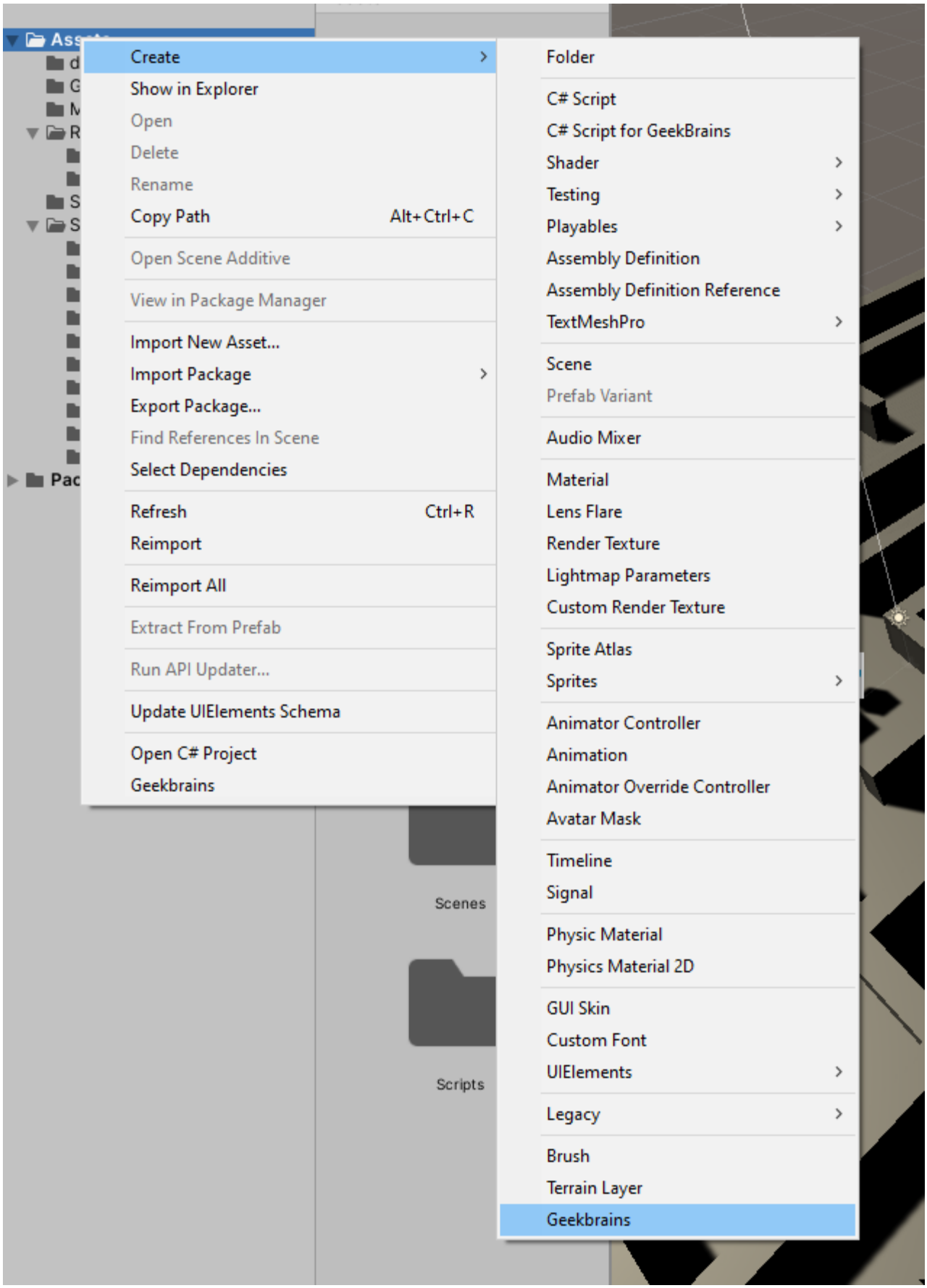
```
using UnityEditor;

namespace Geekbrains
{
    public class MenuItems
    {
        [MenuItem("Geekbrains/Пункт меню №0 ")]
        private static void MenuOption()
        {
        }

        [MenuItem("Geekbrains/Пункт меню №1 %#a")]
        private static void NewMenuOption()
        {
```

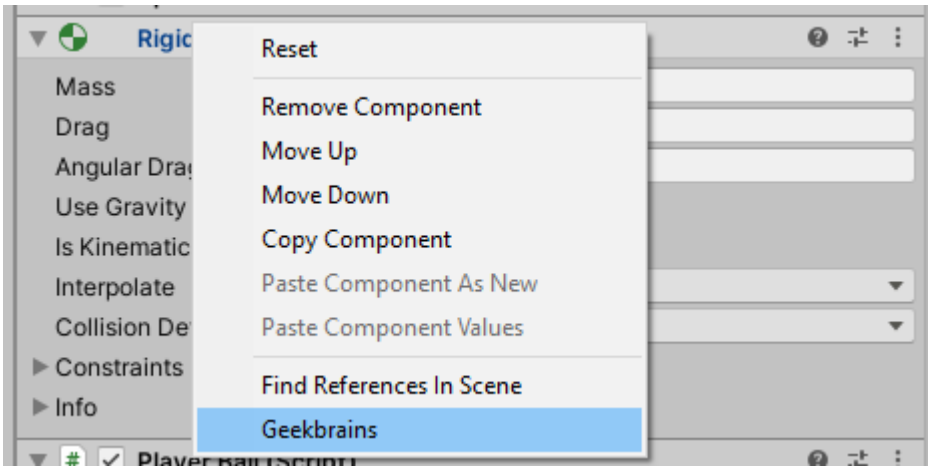
```
        }

        [MenuItem("Geekbrains/Пункт меню №2 %g")]
        private static void NewNestedOption()
        {
        }

        [MenuItem("Geekbrains/Пункт меню №3 _g")]
        private static void NewOptionWithHotkey()
        {
        }

        [MenuItem("Geekbrains/Пункт меню/Пункт меню №3 _g")]
        private static void NewOptionWithHot()
        {
        }
        [MenuItem("Assets/Geekbrains")]
        private static void LoadAdditiveScene()
        {
        }
        [MenuItem("Assets/Create/Geekbrains")]
        private static void AddConfig()
        {
        }
        [MenuItem("CONTEXT/Rigidbody/Geekbrains")]
        private static void NewOpenForRigidBody()
        {
        }
    }
}
```



Добавлять свои пункты можно не только в главную панель, но и в контекстные меню:




Create
Show in Explorer
Open
Delete
Rename
Copy Path Alt+Ctrl+C
Open Scene Additive
View in Package Manager
Import New Asset...
Import Package
Export Package...
Find References In Scene
Select Dependencies
Refresh Ctrl+R
Reimport
Reimport All
Extract From Prefab
Run API Updater...
Update UIElements Schema
Open C# Project
Geekbrains
Folder
C# Script
C# Script for GeekBrains
Shader
Testing
Playables
Assembly Definition
Assembly Definition Reference
TextMeshPro
Scene
Prefab Variant
Audio Mixer
Material
Lens Flare
Render Texture
Lightmap Parameters
Custom Render Texture
Sprite Atlas
Sprites
Animator Controller
Animation
Animator Override Controller
Avatar Mask
Timeline
Signal
Physic Material
Physics Material 2D
GUI Skin
Custom Font
UIElements
Legacy
Brush
Terrain Layer
Geekbrains
Scenes
Scripts



Создание своего окна

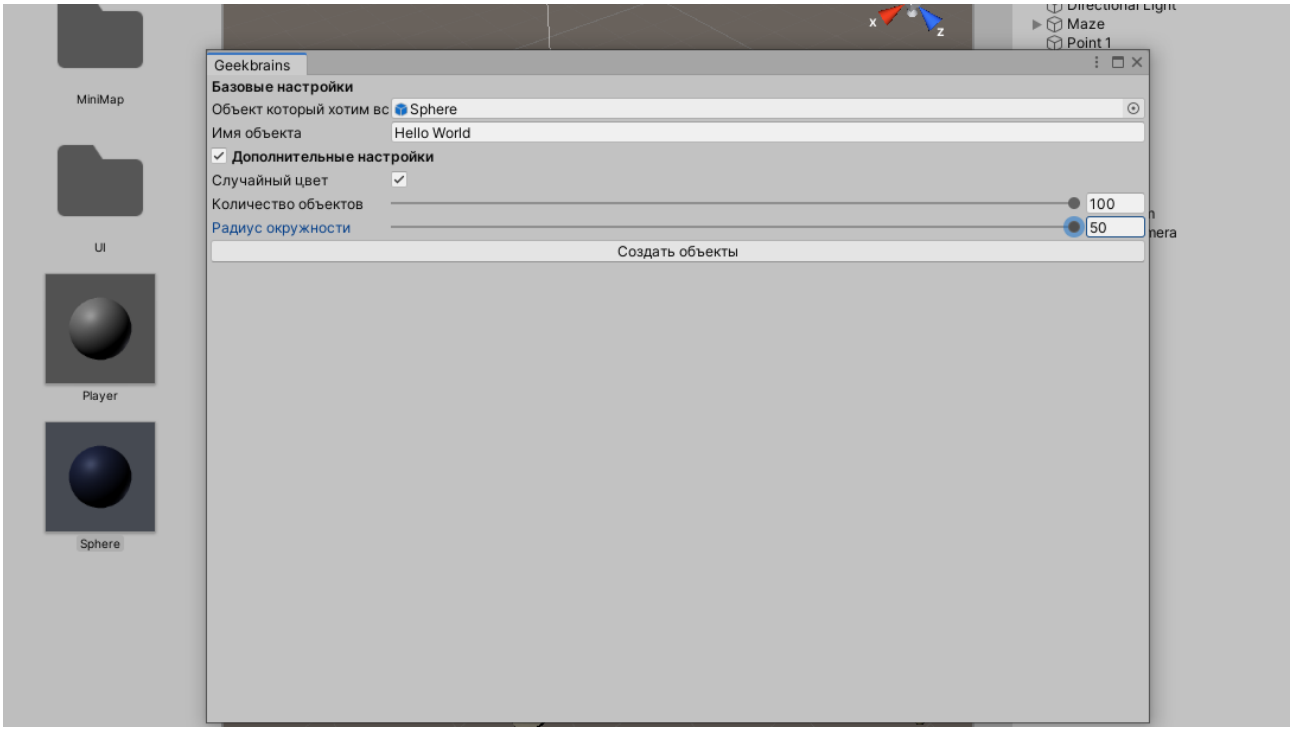
Создаем новый класс **MyWindow**, помещаем его в пространство имен Geekbrains.Editor. Наследуем его от класса **EditorWindow**:

```
using UnityEditor;
using UnityEngine;

namespace Geekbrains
{
   public class MyWindow : EditorWindow
   {
      public static GameObject ObjectInstantiate;
      public string _nameObject = "Hello World";
      public bool _groupEnabled;
      public bool _randomColor = true;
      public int _countObject = 1;
      public float _radius = 10;

      private void OnGUI()
      {
         GUILayout.Label("Базовые настройки", EditorStyles.boldLabel);
         ObjectInstantiate =
            EditorGUILayout.ObjectField("Объект который хотим вставить",
                  ObjectInstantiate, typeof(GameObject), true)
               as GameObject;
         _nameObject = EditorGUILayout.TextField("Имя объекта", _nameObject);
         _groupEnabled = EditorGUILayout.BeginToggleGroup("Дополнительные настройки",
            _groupEnabled);
         _randomColor = EditorGUILayout.Toggle("Случайный цвет", _randomColor);
         _countObject = EditorGUILayout.IntSlider("Количество объектов",
            _countObject, 1, 100);
         _radius = EditorGUILayout.Slider("Радиус окружности", _radius, 10, 50);
         EditorGUILayout.EndToggleGroup();
         var button = GUILayout.Button("Создать объекты");
         if (button)
         {
            if (ObjectInstantiate)
            {
               GameObject root = new GameObject("Root");
```

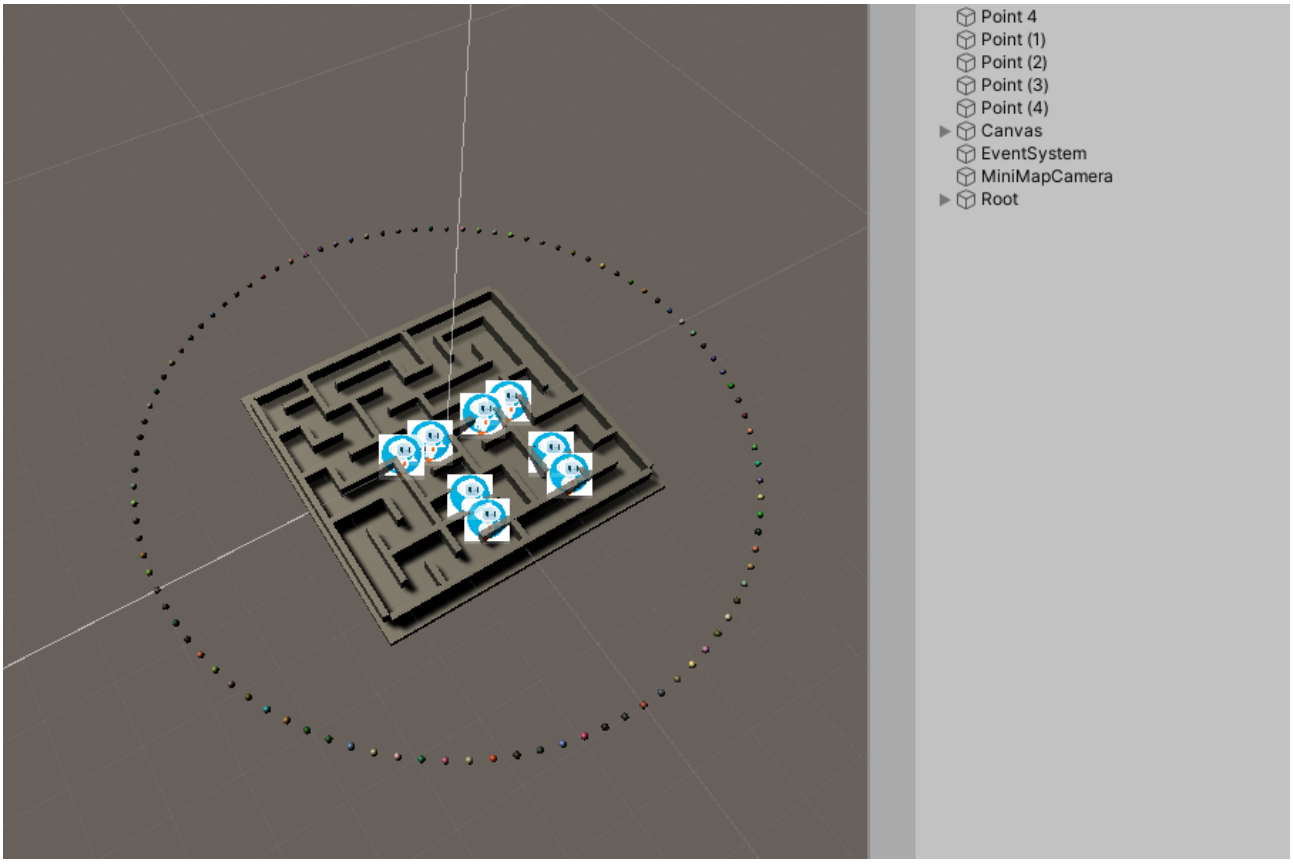
```
                for (int i = 0; i < _countObject; i++)
                {
                    float angle = i * Mathf.PI * 2 / _countObject;
                    Vector3 pos = new Vector3(Mathf.Cos(angle), 0,
                                              Mathf.Sin(angle)) * _radius;
                    GameObject temp = Instantiate(ObjectInstantiate, pos,
                        Quaternion.identity);
                    temp.name = _nameObject + "(" + i + ")";
                    temp.transform.parent = root.transform;
                    var tempRenderer = temp.GetComponent<Renderer>();
                    if (tempRenderer && _randomColor)
                    {
                        tempRenderer.material.color = Random.ColorHSV();
                    }
                }
            }
        }
    }
}
```

```
using UnityEditor;

namespace Geekbrains
{
    public class MenuItems
    {
        [MenuItem("Geekbrains/Пункт меню №0 ")]
        private static void MenuOption()
        {
            EditorWindow.GetWindow(typeof(MyWindow), false, "Geekbrains");
        }
    }
}
```

Этот алгоритм расставит клоны выбранного объекта по окружности с заданным радиусом. Так выглядит созданное нами окно:


MiniMap
UI
Player
Sphere
Geekbrains
Базовые настройки
Объект который хотим вс
Sphere
Имя объекта
Hello World
Дополнительные настройки
Случайный цвет
Количество объектов
100
Радиус окружности
50
Создать объекты
Maze
Point 1

Расставленные объекты:

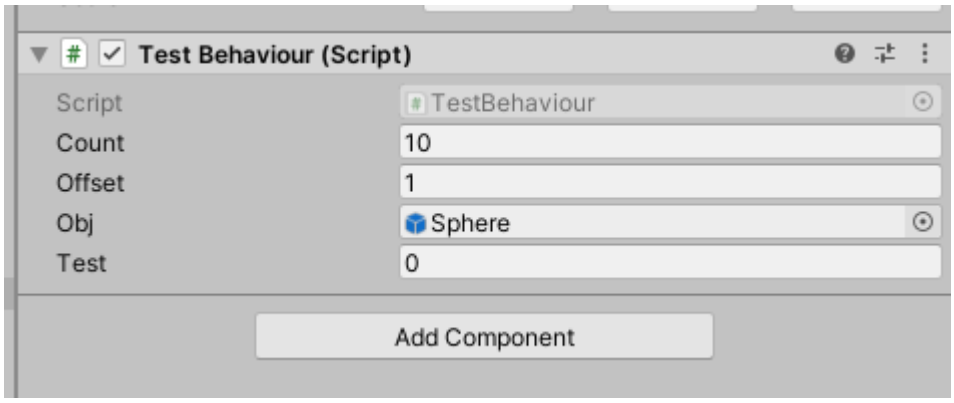


Расширение скриптов

Создаем новый класс для теста и называем его **MyScript**. Добавляем пустой объект и в качестве компонента – наш новый класс. Наделим наш класс простым функционалом: он будет создавать конкретное количество объектов на определенном расстоянии по заданной оси.

```
using UnityEngine;

namespace Geekbrains
{
   public sealed class TestBehaviour : MonoBehaviour
   {
      public int count = 10;
      public int offset = 1;
      public GameObject obj;

      public float Test;
      private Transform _root;

      private void Start()
      {
         CreateObj();
      }

      public void CreateObj()
      {
         _root = new GameObject("Root").transform;
```

```
        for (var i = 1; i <= count; i++)
        {
            Instantiate(obj, new Vector3(0, offset * i, 0),
                Quaternion.identity, _root);
        }
    }

    public void AddComponent()
    {
        gameObject.AddComponent<Rigidbody>();
        gameObject.AddComponent<MeshRenderer>();
        gameObject.AddComponent<BoxCollider>();
    }

    public void RemoveComponent()
    {
        DestroyImmediate(GetComponent<Rigidbody>());
        DestroyImmediate(GetComponent<MeshRenderer>());
        DestroyImmediate(GetComponent<BoxCollider>());
    }
  }
}
```

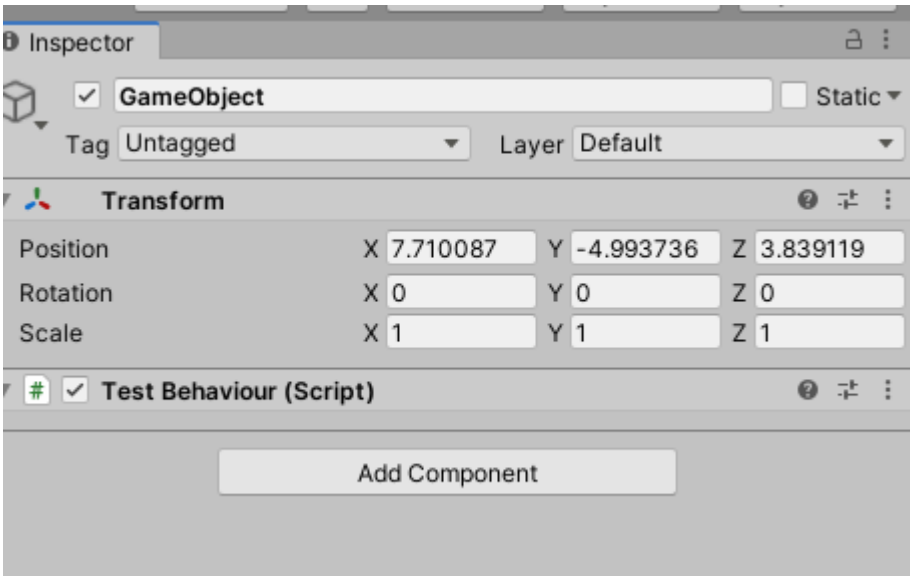


Теперь создадим класс, расширяющий возможности скрипта MyScript.

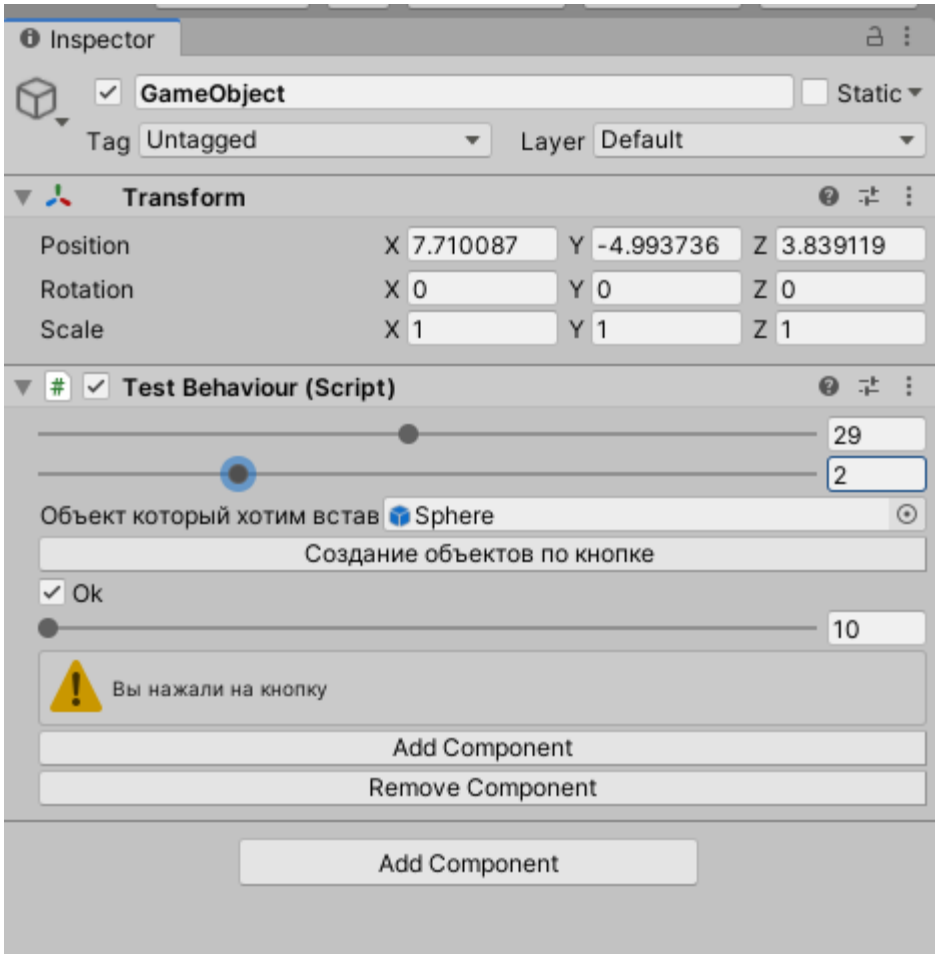
```
using UnityEditor;
using UnityEngine;

namespace Geekbrains
{
   [CustomEditor(typeof(TestBehaviour))]
   public class TestBehaviourEditor : UnityEditor.Editor
   {
      private bool _isPressButtonOk;

      public override void OnInspectorGUI()
      {
         // DrawDefaultInspector();
         TestBehaviour testTarget = (TestBehaviour)target;

         testTarget.count = EditorGUILayout.IntSlider(testTarget.count, 10, 50);
         testTarget.offset = EditorGUILayout.IntSlider(testTarget.offset, 1, 5);
```

```
        testTarget.obj =
           EditorGUILayout.ObjectField("Объект который хотим вставить",
                 testTarget.obj, typeof(GameObject), false)
              as GameObject;

        var isPressButton = GUILayout.Button("Создание объектов по кнопке",
           EditorStyles.miniButtonLeft);

        _isPressButtonOk = GUILayout.Toggle(_isPressButtonOk, "Ok");

        if (isPressButton)
        {
           testTarget.CreateObj();
           _isPressButtonOk = true;
        }

        if (_isPressButtonOk)
        {
           testTarget.Test = EditorGUILayout.Slider(testTarget.Test, 10, 50);
           EditorGUILayout.HelpBox("Вы нажали на кнопку", MessageType.Warning);

           var isPressAddButton = GUILayout.Button("Add Component",
              EditorStyles.miniButtonLeft);
           var isPressRemoveButton = GUILayout.Button("Remove Component",
              EditorStyles.miniButtonLeft);
           if (isPressAddButton)
           {
              testTarget.AddComponent();
           }
           if (isPressRemoveButton)
           {
              testTarget.RemoveComponent();
           }
        }
     }
  }

}
```

Строчка **[CustomEditor(typeof(MyScript))]** означает, что мы создали расширение для класса **MyScript**. Если в начале не вызвать функцию **DrawDefaultInspector**, то наш инспектор будет выглядеть так:



Функции прорисовки редакторских элементов сравнимы с такими функциями в старом пользовательском интерфейсе Unity3d. Выведем на экран то, что получилось:



RayCast в редакторе

```
using System;
using UnityEngine;
```

```
namespace Geekbrains
{
   public sealed class CreateWayPoint : MonoBehaviour
   {
      [SerializeField] private GameObject _prefab;
      private PathBot _rootWayPoint;

      public void InstantiateObj(Vector3 pos)
      {
         if (!_rootWayPoint)
         {
            _rootWayPoint = new GameObject("WayPoint").AddComponent<PathBot>();
         }

         if (_prefab != null)
         {
            Instantiate(_prefab, pos, Quaternion.identity, _rootWayPoint.transform);
         }
         else
         {
            throw new Exception($"Нет префаба на компоненте {typeof(CreateWayPoint)}
{gameObject.name}");
         }
      }
   }
}
```

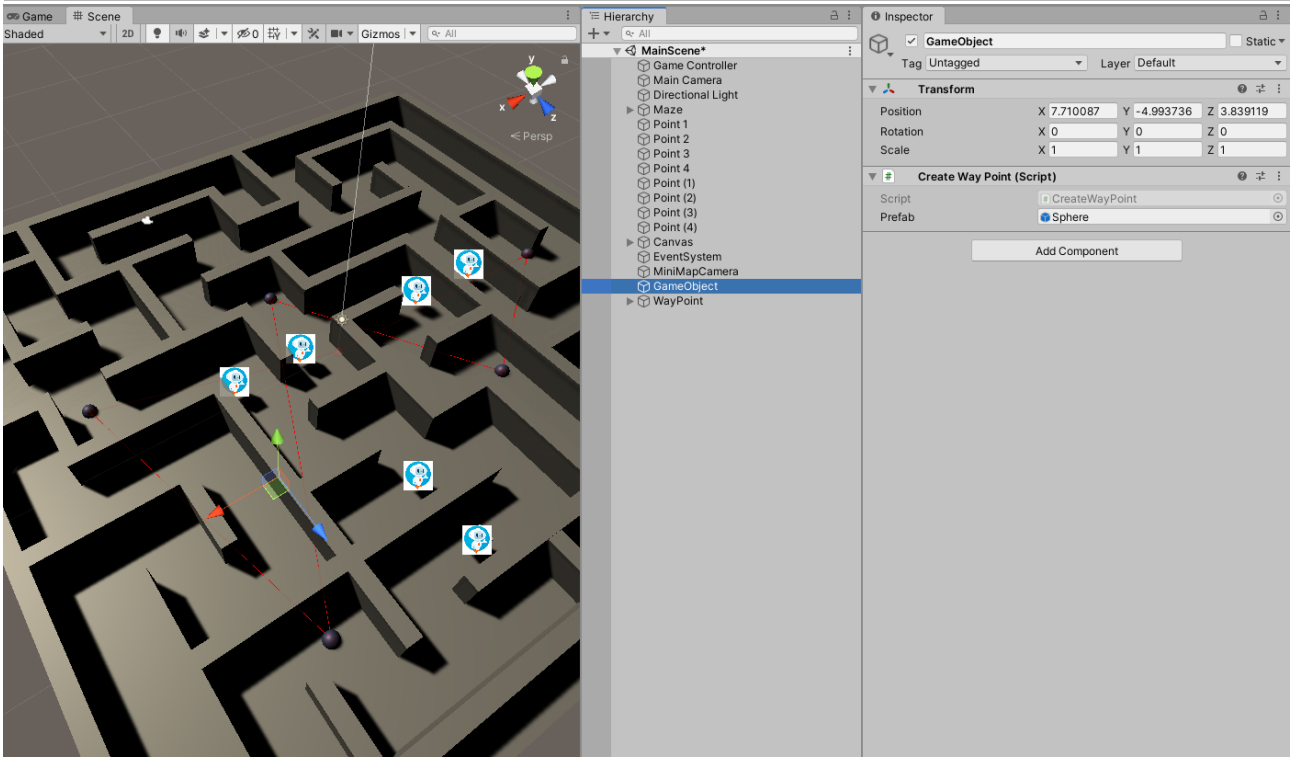
Класс для прорисовки линий

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

namespace Geekbrains
{
   public class PathBot : MonoBehaviour
   {
      [SerializeField]
      private Color _lineColor = Color.red;
      private List<Transform> _nodes = new List<Transform>();

      private void OnValidate()
      {
         _nodes = GetComponentsInChildren<Transform>().ToList();
      }

      // OnDrawGizmosSelected()
      private void OnDrawGizmos()
      {
         Gizmos.color = _lineColor;
         for (var i = 0; i < _nodes.Count; i++)
         {
            var currentNode = _nodes[i].position;
            var previousNode = Vector3.zero;
            if (i > 0)
            {
               previousNode = _nodes[i - 1].position;
            }
```

```
            else if (i == 0 && _nodes.Count > 1)
            {
               previousNode = _nodes[_nodes.Count - 1].position;
            }
            Gizmos.DrawLine(previousNode, currentNode);
            Gizmos.DrawWireSphere(currentNode, 0.3f);
         }
      }
   }
}
```

```
#if UNITY_EDITOR
using UnityEditor;
using UnityEditor.SceneManagement;
using UnityEngine;

namespace Geekbrains
{
   [CustomEditor(typeof(CreateWayPoint))]
   public class CreateWayPointEditor : UnityEditor.Editor
   {
      private CreateWayPoint _testTarget;

      private void OnEnable()
      {
         _testTarget = (CreateWayPoint)target;
      }


      private void OnSceneGUI()
      {
         if (Event.current.button == 0 && Event.current.type == EventType.MouseDown)
         {
            Ray ray = Camera.current.ScreenPointToRay(new
Vector3(Event.current.mousePosition.x,
               SceneView.currentDrawingSceneView.camera.pixelHeight -
Event.current.mousePosition.y));

            if (Physics.Raycast(ray, out var hit))
            {
               _testTarget.InstantiateObj(hit.point);
               SetObjectDirty(_testTarget.gameObject);
            }
         }

         Selection.activeGameObject = FindObjectOfType<CreateWayPoint>().gameObject;

      }

      public void SetObjectDirty(GameObject obj)
      {
         if (!Application.isPlaying)
         {
            EditorUtility.SetDirty(obj);
            EditorSceneManager.MarkSceneDirty(obj.scene);
         }
      }
```

```
    }
#endif
}
```



Editor в пользовательских скриптах

```
using System.Threading;
#if UNITY_EDITOR
using UnityEditor;
#endif
using UnityEngine;

namespace Geekbrains
{
   public sealed class TestEditorBehaviour : MonoBehaviour
   {
      public float Count = 42;
      public int Step = 2;
      private void Start()
      {
#if UNITY_EDITOR
         for (var i = 0; i < Count; i++)
         {
            EditorUtility.DisplayProgressBar("Загрузка", $" проценты {i}",
               i / Count);
            Thread.Sleep(Step * 100);
         }
         EditorUtility.ClearProgressBar();
         var isPressed = EditorUtility.DisplayDialog("Вопрос", @"А оно тебе нужно ? ",
"Ага", "Или нет");
         if (isPressed)
         {
            EditorApplication.isPaused = true;
```

```
        }
#endif
      }
   }
}
```

Практическое задание

1. Разработать свое окно редактора, добавить в него базовые составляющие интерфейса.
2. Реализовать вызов разработанного окна с помощью меню.
3. Расширить функционал скрипта в окне инспектора.
4. *Реализовать атрибут для получения пути к объекту

Дополнительные материалы

1. https://habrahabr.ru/post/163071/ – хорошие примеры по сериализации данных;
2. https://docs.unity3d.com/ScriptReference/Gizmos.html
3. https://docs.unity3d.com/ScriptReference/Handles.html
4. http://catlikecoding.com/unity/tutorials/curves-and-splines/ – хороший материал по рисованию кривых Безье.
5. https://mopsicus.ru/all/draw-lines-bezier-unity/ - Рисование кривых в Unity
6. https://docs.unity3d.com/ScriptReference/Handles.html
7. https://github.com/dbrizov/NaughtyAttributes

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://docs.unity3d.com/ScriptReference/Editor.html
2. https://docs.unity3d.com/ru/current/Manual/editor-EditorWindows.html
3. https://docs.unity3d.com/ScriptReference/Editor.CreateEditor.html
4. https://docs.unity3d.com/ru/530/Manual/editor-CustomEditors.html
5. https://docs.unity3d.com/ru/530/Manual/RunningEditorCodeOnLaunch.html
6. https://unity3d.com/ru/learn/tutorials/topics/interface-essentials/unity-editor-extensions-menu-items?playlist=17090
7. https://docs.unity3d.com/ScriptReference/Editor.OnSceneGUI.html