Игровые объекты и компоненты

Сцены в Unity состоят из *игровых объектов* (Game Objects) Сами по себе эти объекты невидимы и не обладают ничем, кроме имени. Их поведение определяется компонентами.

Каждый игровой объект включает по меньшей мере один компонент, который нельзя удалить. Это компонент `Transform` (или `RectTransform` для объектов UI). Этот компонент отвечает за настройки позиционирования, вращения и масштабирования объекта. Даже пустой игровой объект, создаваемый командой GameObject | Create Empty, содержит компонент `Transform`. Этот компонент необходимо для определения положения объекта на сцене.

Для добавления к игровому объекту новых компонентов можно использовать меню Component; можно также перетаскивать на игровой объект сценарии (скрипты), каждый из которых тоже определяет компонент.

Игровые объекты могут быть дочерними по отношению к другим объектам (родителя можно определить с помощью свойства `transform.parent`). При этом характеристики компонента `Transform` дочерних объектов комбинируются с соответствующими характеристиками их родительских объектов (нулевые характеристики позиции дочернего объекта фактически преобразуются в соответствующие характеристики позиции его родителя). Если же объект не имеет родителя, то нулевые характеристики соответствуют нулевым характеристикам координат сцены.

Чтобы создать сценарий, нужно:

1. Открыть меню Assets (Ресурсы) и выбрать пункт Create | Script | C# Script (кнопка Create с теми же пунктами меню есть и на панели проекта Project).
2. Присвоить имя сценарию. В папке, выбранной на панели проекта, появится новый сценарий с указанным именем (это же имя будет присвоено классу, определенному в сценарии).
3. В панели проекта дважды щелкнуть на сценарии. Сценарий откроется в редакторе сценариев, роль которого по умолчанию играет MonoDevelop (изменить среду для редактирования сценариев можно с помощью команды меню Edit | Preferences | External Tools | External Script Editor). Большинство сценариев будут выглядеть примерно так:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class AlienSpaceship : MonoBehaviour {
  // Используйте этот метод для инициализации
  void Start () {
  }
  // Метод Update вызывается один раз перед отображением каждого кадра
  void Update () {
  }
}
```

В Unity не нужно разрабатывать конструкторы для классов, наследующих `MonoBehaviour`, потому что конструирование объектов осуществляется самим движком Unity.

Сценарии в Unity фактически ничего не делают сами по себе — их код не запускается, пока они не будут подключены к игровому объекту. Существует два основных способа подключения сценария к игровому объекту:

1. Перетащить мышью сценарий с панели проекта на игровой объект на панели иерархии или (для выбранного игрового объекта) на панель инспектора.
2. Используя кнопку Add Component в инспекторе.

Поскольку сценарии, подключенные к игровым объектам, редактор Unity отображает в виде компонентов, можно определять в сценариях свойства, доступные для редактирования в инспекторе. Для этого достаточно определить в своем сценарии открытые переменные.

```
public class AlienSpaceship : MonoBehaviour {
  public string shipName;
  // В инспекторе появится поле "Ship Name",
  // доступное для редактирования
}
```

После подключения сценария к игровому объекту он появится в инспекторе, если выбрать этот объект. Unity автоматически отобразит все открытые переменные в порядке их объявления в коде.

Также в инспекторе появятся закрытые переменные, имеющие атрибут `[SerializeField]`. Это удобно, когда требуется, чтобы поле было доступно в инспекторе, но недоступно другим сценариям.

Отображая имена переменных в инспекторе, редактор Unity заменяет первую букву в имени буквой верхнего регистра, а перед каждой последующей буквой верхнего регистра вставляет пробел. Например, переменная `shipName` будет отображаться в редакторе как «Ship Name».

Получение компонентов

Сценарии имеют доступ к разным компонентам, присутствующим в игровом объекте. Для этого используется метод `GetComponent`.

```
// получить компонент Animator этого объекта, если имеется (иначе вернуть null)
var animator = GetComponent<Animator>();
```

Также можно вызывать метод `GetComponent` других игровых объектов, чтобы получить компоненты, подключенные к ним.

Аналогично можно получать компоненты, подключенные к родительским или дочерним объектам, используя методы `GetComponentInParent` и `GetComponentInChildren`.

Метод `GetComponents` позволяет получить массив всех компонентов, связанных с некоторым игровым объектом (или всех компонентов, имеющих требуемый тип):

```
using UnityEngine;
using System.Collections;

public class MyCustomComponent : MonoBehaviour
{
  private Component[] allComponents = null;

  void Start()
  {
    allComponents = GetComponents<Component>();
    foreach(Component C in allComponents)
    {
      Debug.Log(C.ToString());
    }
  }
}
```

Имеются варианты этого метода для получения всех компонентов *всех* дочерних объектов и всех компонентов *всех* родительских объектов: `GetComponentsInChildren` и `GetComponentsInParent`.

Кэширование ссылок на компоненты

Распространенной ошибкой в сценариях Unity является чрезмерное использование метода `GetComponent`. Для примера рассмотрим следующий фрагмент, проверяющий состояние здоровья (значение `health`) персонажа и выполняющий отключение ряда компонентов, если здоровье падает ниже нулевой отметки (0), чтобы подготовиться к анимации «death»:

```
void TakeDamage()
{
  if (GetComponent<HealthComponent>().health < 0)
  {
    GetComponent<Rigidbody>().enabled = false;
    GetComponent<Collider>().enabled = false;
    GetComponent<AIControllerComponent>().enabled = false;
    GetComponent<Animator>().SetTrigger("death");
  }
}
```

При каждом вызове этот метод будет запрашивать ссылки на пять разных компонентов Component. Даже если эта функция вызывается не из метода Update, ее вызов все равно может совпасть по времени с другими важными событиями.

Кэширование этих ссылок для дальнейшего использования потребует очень небольшого объема памяти, поэтому имеет смысл получить все ссылки во время инициализации и хранить их, пока они необходимы:

```
private HealthComponent healthComponent;
private Rigidbody rigidbody;
private Collider collider;
private AIControllerComponent aiController;
private Animator animator;
void Awake()
{
  healthComponent = GetComponent<HealthComponent>();
  rigidbody = GetComponent<Rigidbody>();
  collider = GetComponent<Collider>();
  aiController = GetComponent<AIControllerComponent>();
  animator = GetComponent<Animator>();
}
void TakeDamage()
{
  if (healthComponent.health < 0)
  {
    rigidbody.detectCollisions = false;
    collider.enabled = false;
    aiController.enabled = false;
    animator.SetTrigger("death");
  }
}
```

Самый быстрый метод получения ссылок на компоненты

Имеются три перегруженные версии метода GetComponent: GetComponent(string), GetComponent<T>() и GetComponent(typeof(T)).

В Unity 4 быстрее всех работает метод GetComponent(typeof(T)). В Unity 5 производительность методов GetComponent<T>() и GetComponent(typeof(T)) примерно одинакова, а метод GetComponent(string) работает почти в 30 раз медленнее (интересно, что он стал даже медленнее, чем в Unity 4).

В Unity 4 можно пользоваться различными свойствами класса GameObject, такими как collider, rigidbody, camera и т. д. Эти свойства подобны предварительно кэшированным ссылкам на компоненты и действуют значительно быстрее, чем любые традиционные методы GetComponent:

Однако для уменьшения числа зависимостей и улучшения модульности кода в серверной части движка в версии Unity 5 было решено отказаться от поддержки этих переменных. Сохранено только свойство transform.

Сохранение ссылок на существующие объекты

Альтернативой программному кэшированию является явное заполнение полей в инспекторе Unity.

Рассмотрим класс, который экспортирует в инспектор три закрытые переменные:

```
public class EnemySpawnerComponent : MonoBehaviour
{
  [SerializeField] private int numEnemies;
  [SerializeField] private GameObject enemyPrefab;
  [SerializeField] private EnemyManagerComponent enemyManager;
  void Start()
  {
    SpawnEnemies(_numEnemies);
```

```
  }
  void SpawnEnemies(int numEnemies)
  {
    for(int i = 0; i < _numEnemies; ++i)
    {
      GameObject enemy = (GameObject)GameObject.Instantiate(enemyPrefab, Vector3.zero,
        Quaternion.identity);
      enemyManager.AddEnemy(enemy);
    }
  }
}
```

В поле Enemy Prefab инспектора можно перетащить мышью ссылку на шаблон объекта из окна проектов или даже ссылку на другой игровой объект, присутствующий в сцене. Тем самым мы определим значение закрытого поля `enemyPrefab` в автоматически создаваемом экземпляре класса `EnemySpawnerComponent`.

Поле Enemy Manager хранит ссылку на компонент, а не на игровой объект. Если перетащить в это поле игровой объект, его значением станет ссылка на компонент данного объекта, а не на сам игровой объект. Если данный объект не содержит нужного компонента, присваивания не произойдет.

Поиск игровых объектов

Чтобы получить ссылку на игровой объект из какого-либо *его* компонента, достаточно использовать свойство компонента `gameObject`.

Найти игровой объект можно с помощью статических методов `Find` и `FindGameObjectWithTag` класса `GameObject`. Метод `GameObject.Find(name)` возвращает *первый* игровой объект с указанным именем name (регистр учитывается; имена объектов указываются в панели иерархии):

```
ObjPlayer = GameObject.Find("Player");
```

Поиск по тегу с помощью метода `FindGameObjectWithTag` является более эффективным. По умолчанию строковое свойство `tag` игрового объекта равно `"Untagged"`. Изменить тег объекта можно не только программным способом, но и с помощью инспектора, развернув выпадающий список Tag и выбрав требуемый тег. Команда Add Tag позволяет определить новый тег. Примерами стандартных тегов являются Player, MainCamera, GameController.

Метод `FindGameObjectWithTag` возвращает первый активный игровой объект, имеющий указанный тег (или `null`, если ни одного объекта не найдено). Метод `FindGameObjectsWithTag` возвращает массив всех активных игровых объектов с указанным тегом (или пустой массив, если требуемые объекты отсутствуют):

```
using UnityEngine;
using System.Collections;

public class ObjectFinder : MonoBehaviour
{
  public string TagName = "Enemy";
  public GameObject[] FoundObjects;

  void Start ()
  {
    FoundObjects = GameObject.FindGameObjectsWithTag(TagName);
  }
}
```

Одному объекту нельзя присвоить несколько тегов. Однако можно добавить к объекту набор пустых дочерних объектов и присвоить им различные теги. В классе `GameObject` имеется также статический метод `FindWithTag`, который, судя по всему, является синонимом метода `FindGameObjectWithTag` (интересно, что в справке Unity методы `FindWithTag` и `FindGameObjectsWithTag` упоминаются, а метод `FindGameObjectWithTag` — нет).

Сравнение тегов

Для сравнения тега игрового объекта с конкретным значением следует использовать экземплярную функцию CompareTag(tag) класса GameObject (этот же метод есть и в классе Component):

```
bool bMatch = gameObject.CompareTag("Player");
```

Примечание. Сравнение тегов с помощью метода CompareTag более эффективно, чем явное сравнение строк вида gameObject.tag == "Player", поскольку свойство gameObject.tag на каждое обращение создает новую строку, чего метод для сравнения тегов не делает. Впрочем, в справочной системе Unity имеются примеры явного сравнения тегов.

Чтобы проверить, что две ссылки указывают на один и тот же объект, можно использовать метод GetInstanceID, возвращающий уникальное значение типа int. Следующий фрагмент кода получает все объекты с указанным тегом, а при их обработке пропускает текущий объект:

```
FoundObjects = GameObject.FindGameObjectsWithTag(TagName);
foreach(GameObject o in FoundObjects)
{
  if(o.GetInstanceID() == gameObject.GetInstanceID())
    continue;
  ...
}
```

Впрочем, аналогичного результата можно добиться, используя операцию сравнения ==:

```
  if (o == gameObject)
```

Классы, связанные с игровыми объектами и компонентами

Класс UnityEngine.Object

Использованный выше метод GetInstanceID и операция сравнения == (как и операция сравнения !=) определены в классе Object из пространства имен UnityEngine. У этого класса имеется также операция неявного приведения к типу bool, которая возвращает true, если объект существует. Таким образом, вместо if (GetComponent<Rigidbody>() != null) можно использовать if (GetComponent<Rigidbody>()).

Этот класс является базовым как для игровых объектов, так и для компонентов. Он имеет свойство name, определяющее имя игрового объекта (имена компонентов совпадают с именем того игрового объекта, к которому они присоединены).

Среди его статических методов можно отметить следующие:

void Destroy(Object obj, float time = 0.0f) — уничтожает объект через указанный промежуток времени (в секундах);

void DontDestroyOnLoad(Object obj) — запрещает удаление объекта obj при загрузке новой сцены;

T FindObjectOfType<T>() — возвращает первый активный объект указанного типа (активность игрового объекта, всех его компонентов и дочерних игровых объектов можно настраивать с помощью метода SetActive(bool value) класса GameObject);

T[] FindObjectsOfType<T>() — возвращает все активные объекты указанного типа;

Object Instantiate(Object original) — возвращает копию существующего объекта (данный метод имеет перегруженные варианты, позволяющие указывать родительский объект для созданной копии, а также изменить ее позицию и ориентацию.

Класс Component

Класс Component является прямым потомком класса Object. Он содержит следующие дополнительные свойства:

gameObject — игровой объект, к которому присоединен компонент;

tag — тег игрового объекта, к которому присоединен данный компонент;

transform — компонент Transform игрового объекта, к которому присоединен данный компонент.

В этом классе реализованы следующие методы (многие из которых были описаны ранее, а другие будут описаны далее):

BroadcastMessage, CompareTag, GetComponent, GetComponentInChildren, GetComponentInParent, GetComponents, GetComponentsInChildren, GetComponentsInParent, SendMessage, SendMessageUpwards

Класс GameObject

Класс GameObject тоже является прямым потомком класса Object. Он содержит следующие дополнительные свойства:

- activeInHierarchy — является ли игровой объект активным на сцене;
- active — является ли игровой объект активным (объект будет активным на сцене, если все его родительские объекты являются тоже активными);
- layer — слой, к которому относится игровой объект;
- scene — сцена, на которой находится игровой объект;
- tag — тег игрового объекта;
- transform — компонент Transform игрового объекта.

Для класса GameObject предусмотрены три вида конструкторов:

```
public GameObject();
public GameObject(string name);
public GameObject(string name, params Type[] components);
```

В любом случае при создании игрового объекта к нему присоединяется компонент Transform.

В этом классе реализованы все методы, ранее приведенные для класса Component, а также два дополнительных метода:

AddComponent, SetActive

Статические методы класса GameObject:

CreatePrimitive, Find, FindGameObjectsWithTag, FindWithTag

Метод CreatePrimitive имеет параметр типа перечисления PrimitiveType со значениями Sphere, Capsule, Cylinder, Cube, Plane, Quad.

Иерархия игровых объектов

Графическое представление иерархии игровых объектов приводится в панели «Иерархия». Ниже приведены примеры программной настройки и анализа иерархии игровых объектов:

```
using UnityEngine;
using System.Collections;

public class Parenter : MonoBehaviour
{
  private GameObject Child;
  private GameObject Parent;

  void Start ()
  {
    Child = GameObject.Find("Child");
    Parent = GameObject.Find("Parent");
    Child.transform.parent = Parent.transform;

    for(int i = 0; i < Parent.transform.childCount; i++)
      Debug.Log(Parent.transform.GetChild(i).name);
  }
}
```

Классы Behaviour и MonoBehaviour

Класс Behaviour является прямым потомком класса Component. В отличие от класса Component класс Behaviour может находиться в доступном и недоступном состоянии (для недоступных компонентов не вызываются методы Start, Update, FixedUpdate, LateUpdate). В этот класс добавлены всего два новых свойства: enabled и isActiveAndEnabled.

Класс MonoBehaviour является прямым потомком класса Behaviour и базовым классом для всех сценариев.

Он содержит следующие экземплярные методы (которые описываются далее):

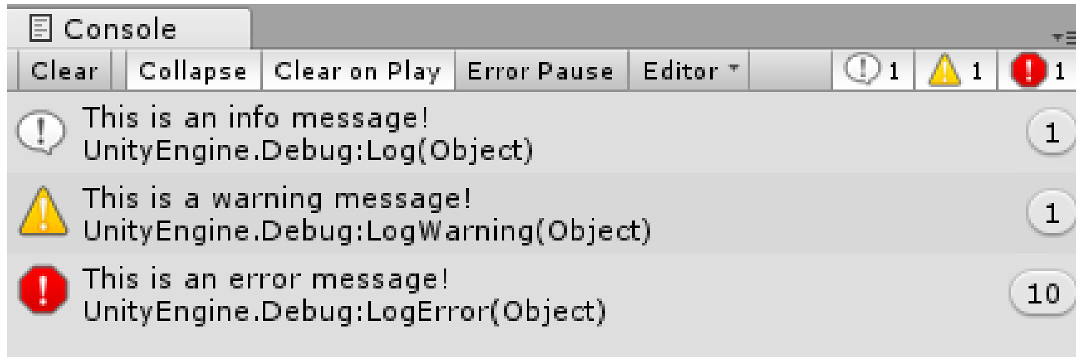
CancelInvoke, Invoke, InvokeRepeating, IsInvoking, StartCoroutine, StopAllCoroutines, StopCoroutine

В нем определен статический метод print, являющийся синонимом метода Debug.Log.

Дополнительные возможности вывода в консоль

Кроме функции Debug.Log (и ее синонима print) можно использовать ее варианты LogWarning и LogError:

```
Debug.Log("This is an info message!");
Debug.LogWarning("This is a warning message!");
for (int i = 0; i < 10; i++)
  Debug.LogError("This is an error warning message!");
```



При включенном режиме Collapse одинаковые рядом стоящие сообщения объединяются, и рядом с ними указывается их количество. При включенном режиме Error Pause игра переходит в режим паузы при выводе сообщения об ошибке (LogError).

Все указанные выше методы имеют вариант с суффиксом Format (например, Debug.LogFormat), который позволяет выводить форматированную строку с набором данных.

В отладочные сообщения можно включать теги форматирования <size> (размер), <b> (полужирный), <i> (наклонный) и <color> (цвет):

```
Debug.Log("<color=red>[ERROR]</color>This is a <i>very</i>
  <size=14><b>specific</b></size> kind of log message");
```

Создание объектов в процессе игры

Существует два способа создания объектов в течение игры:

- создать копию имеющегося объекта;
- создать пустой игровой объект (или трехмерный примитив) и подключить к нему компоненты программным способом.

Первый способ особенно удобен, когда копируемый объект является *шаблонным* (prefab). Шаблонные объекты — это объекты, подготовленные заранее и хранящиеся в виде ресурсов. Это означает, что можно создать один шаблон объекта, а затем создать из него множество копий в разных сценах.

Создать копию объекта можно с помощью метода Instantiate:

```
public GameObject myPrefab; // перед запуском игры нужный шаблон перетаскивается
                            // на поле My Prefab в инспекторе
void Start()
{
  // Создать новую копию объекта myPrefab
  // и поместить его в ту же позицию, где находится объект this
  var newObject = (GameObject)Instantiate(myPrefab);
  newObject.transform.position = this.transform.position;
}
```

Метод Instantiate возвращает значение типа Object, а не GameObject, поэтому чтобы использовать его как экземпляр GameObject, требуется выполнить приведение типа.

Для создания игрового объекта «с нуля» прежде всего нужно вызвать конструктор класса GameObject, а затем добавить в полученному экземпляру компоненты, используя метод AddComponent.

```
// Создать новый игровой объект; он появится в иерархии под именем "My New GameObject"
var newObject = new GameObject("My New GameObject");
// Добавить в него новый компонент SpriteRenderer
var renderer = newObject.AddComponent<SpriteRenderer>();
// Определить спрайт, который должен отображать новый SpriteRenderer
renderer.sprite = mySprite;
```

Метод `AddComponent` принимает обобщенный параметр — тип компонента (любой потомок класса `Component`, в том числе потомок класса `MonoBehavior`).

Объекты, создаваемые в режиме проигрывания, исчезают после остановки игры, однако их можно сохранить, выполнив следующие действия:

- в режиме проигрывания выбрать в окне иерархии нужные объекты;
- скопировать их командой Edit | Copy (Ctrl+C);
- выйти из режима проигрывания;
- вставить скопированные объекты в окно иерархии командой Edit | Paste (Ctrl+V).

Уничтожение объектов

Метод `Destroy` обеспечивает удаление как игровых объектов, так и компонентов:

```
// Удалить игровой объект, к которому подключен данный сценарий
Destroy(gameObject);
```

Если в вызов `Destroy` передать только ссылку `this`, он удалит не игровой объект, а компонент с данным сценарием, подключенный к игровому объекту. Игровой объект останется в сцене, но уже без данного сценария.

Основные события класса MonoBehaviour

Класс `MonoBehaviour` имеет большое число функций-событий, предназначенных для переопределения и вызываемых в определенные моменты выполнения программы.

Awake, OnEnable, OnDisable, OnDestroy

Метод `Awake` вызывается сразу после создания экземпляра объекта в сцене, и это первая возможность выполнить код в своем сценарии. `Awake` вызывается только один раз на протяжении всего времени жизни экземпляра.

Метод `OnDestroy` вызывается при уничтожении компонента (которое автоматически происходит при завершении сцены или всей игры).

Методы `OnEnable` и `OnDisable` вызываются всякий раз, когда экземпляр переходит в доступное или недоступное состояние. Первый раз метод `OnEnable` вызывается после метода `Awake` (если после создания объект является доступным), последний раз метод `OnDisable` вызывается при разрушении доступного объекта (перед методом `OnDestroy`).

Start

Метод `Start` вызывается один раз непосредственно перед первым вызовом метода `Update` объекта, при условии, что объект является доступным, а связанный с ним игровой объект является активным. Методы `Start` всех объектов вызываются только после того, как будут завершены вызовы всех методов `Awake`.

Это удобно, например, когда в объекте A имеется поле, используемое объектом B:

ObjectA.cs

```
class ObjectA : MonoBehaviour
{
  public Animator animator;
  void Awake()
  {
    animator = GetComponent<Animator>();
  }
}
```

ObjectB.cs

```
class ObjectB : MonoBehaviour
```

```
{
  public ObjectA someObject;

  void Start()
  {
    Debug.Log("Start: " + someObject.animator != null); // true
  }
}
```

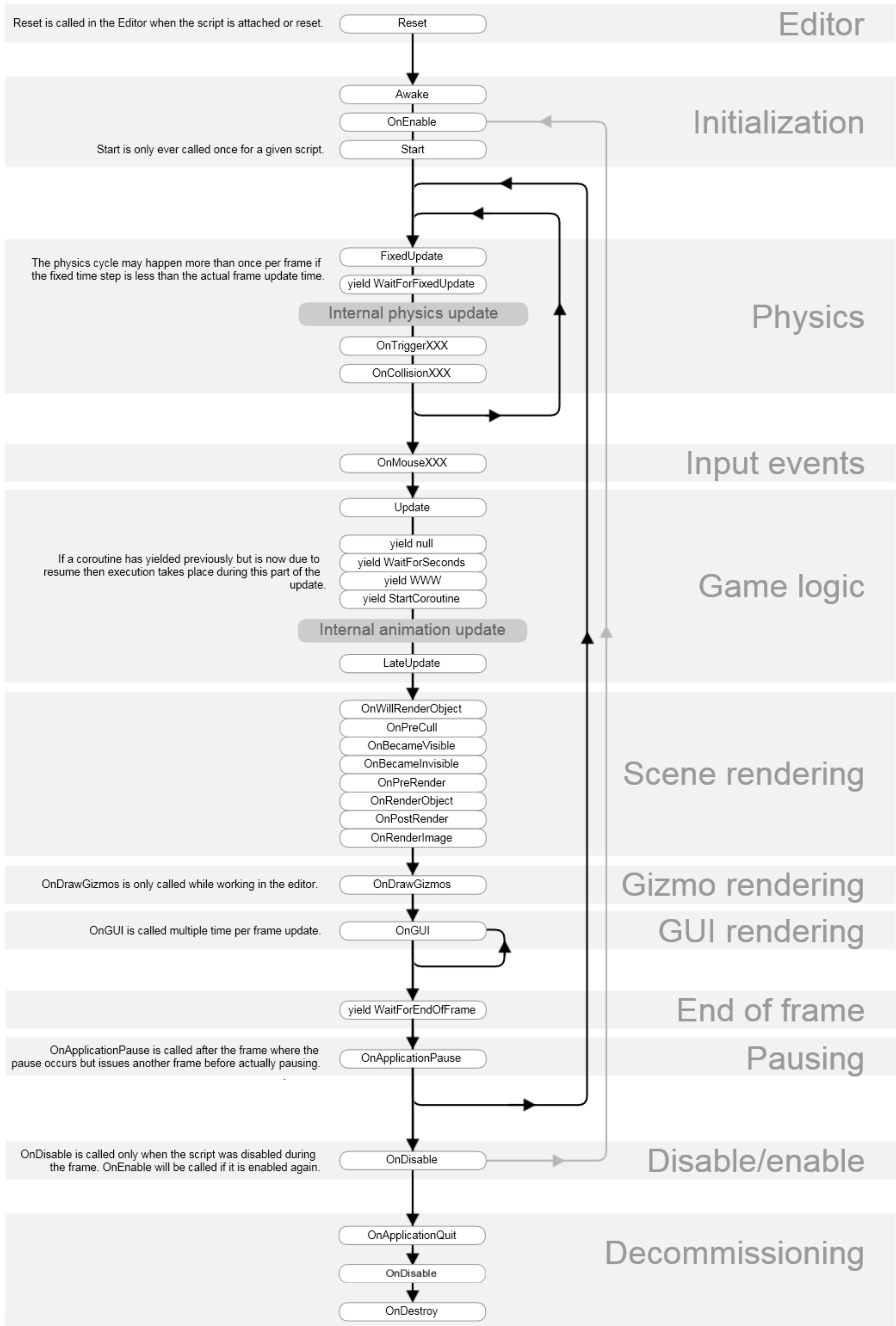
Update, LateUpdate и FixedUpdate

Метод `Update` вызывается для каждого кадра, пока компонент доступен, а игровой объект, к которому подключен компонент, является активным.

Методы `Update` должны выполнять свою работу максимально быстро, потому что вызываются для каждого кадра. Для выполнения продолжительных операций можно использовать *сопрограммы* (coroutine), которые описываются далее.

Unity вызывает методы `Update` во всех сценариях, имеющих эти методы. Затем вызываются методы `LateUpdate` во всех сценариях, имеющих эти методы. Никакой из методов `LateUpdate` не будет вызван, пока не выполнятся все методы `Update`.

Кроме метода `Update` можно также использовать метод `FixedUpdate`. В отличие от `Update`, который вызывается один раз для каждого кадра, метод `FixedUpdate` вызывается *фиксированное число раз в секунду*. Этот метод применяется для реализации физических явлений, когда некоторые операции должны выполняться через регулярные интервалы времени.



Editor
Reset is called in the Editor when the script is attached or reset.
Reset
Initialization
Awake
OnEnable
Start is only ever called once for a given script.
Start
Physics
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
yield WaitForFixedUpdate
Internal physics update
OnTriggerXXX
OnCollisionXXX
Input events
OnMouseXXX
Game logic
Update
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
Internal animation update
LateUpdate
Scene rendering
OnWillRenderObject
OnPreCull
OnBecameVisible
OnBecameInvisible
OnPreRender
OnRenderObject
OnPostRender
OnRenderImage
Gizmo rendering
OnDrawGizmos is only called while working in the editor.
OnDrawGizmos
GUI rendering
OnGUI is called multiple time per frame update.
OnGUI
End of frame
yield WaitForEndOfFrame
Pausing
OnApplicationPause is called after the frame where the pause occurs but issues another frame before actually pausing.
OnApplicationPause
Disable/enable
OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.
OnDisable
Decommissioning
OnApplicationQuit
OnDisable
OnDestroy

Время в сценариях

Для получения информации о текущем времени в играх используется класс `Time`. В этом классе доступно несколько статических свойств, но наиболее важным и часто используемым является `deltaTime`. Во всех свойствах время измеряется в секундах и имеет тип `float`.

`deltaTime` — время, прошедшее между вызовами предыдущего и текущего кадра (read-only);

`fixedDeltaTime` — время между вызовами метода `FixedUpdate`;

`time` — время от начала игры до начала текущего кадра (read-only);

`fixedTime` — время от начала игры до начала последнего вызова `FixedUpdate` (read-only);

`timeSinceLevelLoad` — время после загрузки последней сцены (read-only).

`timeScale` — масштаб времени (по умолчанию равен 1), если равен 0.5, то время замедляется в два раза. Все указанные выше характеристики зависят от значения `timeScale` (за исключением `fixedDeltaTime`). Свойство `timeScale` можно использовать для реализации эффекта замедления времени. При `timeScale`, равном 0, функции `Update`, `LastUpdate` и `FixedUpdate` не вызываются.

Переменную `Time.deltaTime` можно использовать для выполнения действий, которые должны продолжаться на протяжении нескольких кадров, но в течение определенного отрезка времени.

Если требуется реализовать равномерное движение объекта, то не следует перемещать его на фиксированное расстояние в каждом кадре, поскольку частота кадров в секунду может существенно изменяться. Например, если камера направлена на относительно простой участок сцены, частота кадров может быть очень высокой, а при отображении визуально более сложных сцен частота может уменьшаться.

Так как невозможно гарантировать постоянную частоту кадров, следует включить в расчеты переменную `Time.deltaTime`:

```
void Update ()
{
  transform.localPosition += transform.forward * Speed * Time.deltaTime;
}
```

Впрочем, имеется и другой способ: организовать перемещение объекта в методе `FixedUpdate`, который вызывается с фиксированной частотой.

Удаление пустых объявлений обратных вызовов

При создании нового файла сценария с реализацией MonoBehaviour в Unity 4 или 5 редактор Unity автоматически создаст заготовки двух методов:

```
// Используется для инициализации
void Start () {
}
// Вызывается один раз в каждом кадре
void Update () {
}
```

Движок Unity определяет присутствие этих методов на этапе инициализации и добавляет их в список методов обратного вызова. Но если оставить пустые объявления в коде, это приведет к появлению небольших накладных расходов на их вызов движком.

Сопрограммы

Сопрограмма (coroutine) — это функция, которая выполняется на протяжении нескольких кадров. Чтобы создать сопрограмму, нужно объявить метод, возвращающий значение типа `IEnumerator` и использовать в нем конструкцию `yield return` для временной приостановки, чтобы дать возможность выполниться остальным фрагментам игры. Например, так можно реализовать уменьшение прозрачности игрового объекта вплоть до его полного исчезновения в течение 2 секунд:

```
IEnumerator Fade()
{
    for (float f = 1f; f >= 0; f -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = f;
```

```
            renderer.material.color = c;
            yield return new WaitForSeconds(0.2f);
        }
    }
```

Конструкция `yield return` временно приостанавливает выполнение функции. Unity автоматически возобновит ее работу позднее, но когда это произойдет, зависит от значения, возвращаемого `yield return`.

Некоторые конструкции `yield return`, предусмотренные для сопрограмм:

- приостановка выполнения до следующего кадра:

```
yield return null;
```

- приостановка выполнения до следующего вызова метода `FixedUpdate`:

```
yield return WaitForFixedUpdate();
```

- приостановка выполнения на три секунды (учитывается текущий масштаб времени `timeScale`; имеется также метод `WaitForSecondsRealtime`, который использует реальное время, без учета его масштабирования):

```
yield return new WaitForSeconds(3f);
```

- приостановка выполнения до момента, когда переменная `someBoolVar` получит значение `true` (имеется аналогичный метод `WaitWhile`, приостанавливающий выполнение, пока указанное в нем лямбда-выражение возвращает значение `true`).

```
yield return new WaitUntil(() => someBoolVar)
```

Остановить сопрограмму можно конструкцией `yield break`. Кроме того, сопрограммы автоматически останавливаются, когда выполнение достигает конца метода.

Для запуска сопрограммы надо использовать специальный метод `StartCoroutine`, определенный в классе `MonoBehaviour`:

```
StartCoroutine(Fade());
```

В качестве параметра обычно указывается *вызов* требуемой сопрограммы (таким образом, параметр имеет тип `IEnumerable`). Имеется также менее эффективный вариант, принимающий *имя* сопрограммы и, возможно, единственный ее параметр (для этого варианта нельзя использовать сопрограммы с более чем одним параметром). Метод `StartCoroutine` возвращает значение типа `Coroutine`, которое может использоваться для досрочного завершения сопрограммы методом `StopCoroutine`:

```
Coroutine myCoroutine = StartCoroutine(MyCoroutine());
// ...
StopCoroutine(myCoroutine);
```

Метод `StopCoroutine` может также принимать строковый параметр — имя сопрограммы (неэффективный вариант, который останавливает *первую* сопрограмму, для запуска которой было использовано это имя) или параметр типа `IEnumerable` (в этом случае он должен совпадать со значением, возвращенным той сопрограммой, которую требуется остановить).

Имеется также метод `StopAllCoroutines` (без параметров), позволяющий остановить все сопрограммы, запущенные данным компонентом.

Кроме обычных значений в конструкции `yield return` можно также запустить другую сопрограмму (тем же методом `StartCoroutine`). В этом случае вызывающая сопрограмма будет ждать, пока завершится вызванная ею сопрограмма, после чего продолжит свое выполнение.

Хотя сопрограммы можно настроить так, чтобы они выполняли определенные действия в каждом кадре (используя конструкцию `yield return null`), наиболее эффективным является их применение в случае, если требуется выполнять периодические действия с достаточно большим интервалом.

Метод Update, сопрограммы и метод InvokeRepeating

Метод `Update` вызывается в каждом кадре. В нем можно реализовать логику, позволяющую выполнять определенные действия через указанные промежутки времени, но этот подход является очень неэффективным:

```
void Update()
{
  _timer += Time.deltaTime;
  if (_timer > updateFrequency)
```

```
    {
      DoSomething();
      _timer -= updateFrequency;
    }
  }
```
Эффективным вариантом является использование сопрограммы:
```
void Start()
{
  StartCoroutine(DoInCoroutine());
}

IEnumerator DoInCoroutine()
{
  while (true)
  {
    yield return new WaitForSeconds(updateFrequency);
    DoSomething();
  }
}
```
Альтернативным вариантом является использование метода InvokeRepeating, также определенного в класса MonoBehaviour:
```
void Start()
{
  InvokeRepeating("DoSomething", updateFrequency, updateFrequency);
}
```
Первый параметр этого метода содержит строку с именем вызываемого метода, второй — временную задержку перед первым вызовом метода, третий — промежуток времени между последующими вызовами метода. Метод InvokeRepeating более прост в применении по сравнении с сопрограммой, но при этом обладает меньшими возможностями. Этот метод не имеет перегруженных вариантов.

Имеется также метод Invoke с двумя параметрами, который выполняет *однократный* запуск требуемого метода через указанное время. С его помощью тоже можно реализовать многократный запуск метода, причем этот вариант позволит прервать цепочку запусков при выполнении некоторого условия:
```
void Start()
{
  Invoke("DoByInvoke", updateFrequency);
}
void DoByInvoke()
{
  DoSomething();
  Invoke("DoByInvoke", updateFrequency);
}
```
Для того чтобы прервать *все* вызовы Invoke и InvokeRepeating, выполненные в данном компоненте, предназначен метод CancelInvoke без параметров. Имеется также метод IsInvoking(methodName), возвращающий true, если в данный момент методом Invoke или InvokeRepeating запущен метод с именем methodName.

Следует отметить, что даже при недоступном компоненте (когда метод Update для него не вызывается) функции InvokeRepeating и сопрограммы будут продолжать выполняться. Однако функции InvokeRepeating и сопрограммы с вариантом yield return new WaitForSeconds не будут выполняться, если значение timeScale равно 0. Сопрограммы перестают выполняться, если игровой объект, связанный с запустившим их компонентом, становится неактивным или уничтожается («In Unity 2018.1, coroutines returned from a MonoBehaviour while its GameObject is being disabled or destroyed are no longer started»).

SendMessage and BroadcastMessage

Методы BroadcastMessage, SendMessage и SendMessageUpwards определены как с классе Component, так и в классе GameObject. Несмотря на то что эти методы не рекомендуется использовать по причине их низкой эффективности (о чем будет подробнее сказано далее), всё-таки приведем их описание.

Все эти методы могут принимать четыре варианта параметров:

```
(string methodName)
(string methodName, object value)
(string methodName, object value, SendMessageOptions options)
(string methodName, SendMessageOptions options)
```

Параметр methodName определяет имя метода, который должен быть вызван, необязательный параметр value определяет параметр этого метода, а метод options определяет действия при отсутствии требуемого метода. Если параметр options отсутствует или равен RequireReceiver, то в случае отсутствия нужного метода в консоль выводится сообщение об ошибке, если же параметр равен DontRequireReceiver, то сообщение об ошибке не выводится.

Методы BroadcastMessage, SendMessage и SendMessageUpwards различаются областью поиска нужного метода. Метод SendMessage выполняет поиск во всех компонентах того игрового объекта, в котором он вызван, метод BroadcastMessage делает то же самое и дополнительно просматривает все дочерние игровые объекты, а метод SendMessageUpwards делает то же, что и SendMessage, и дополнительно просматривает все родительские игровые объекты в восходящем порядке (хотя в документации об этом сказано весьма туманно: «Calls the method named methodName on every MonoBehaviour in this game object and on every ancestor of the behaviour»).

Если игровой компонент является неактивным или компонент недоступным, то в них поиск не проводится.

Методы семейства SendMessage (и методы Find) являются крайне дорогостоящими, и их следует избегать при любой возможности. Метод SendMessage выполняется примерно в 2000 раз медленнее вызова простой функции, а медлительность метода Find особенно ярко проявляется в сложных сценах, поскольку он последовательно перебирает все игровые объекты в сцене. Метод Find допускается вызывать во время инициализации сцены, например из методов Awake и Start, для поиска объектов, которые уже существуют в сцене. Однако использование любого из этих методов для организации взаимодействий объектов во время выполнения приведет к очень заметным накладным расходам.

Приведем пример плохой реализации, связанной с использованием методов Find и SendMessage. Рассматриваемый метод создает заданное количество экземпляров врагов из шаблона объекта, а затем уведомляет объект EnemyManager об их существовании:

```
public void SpawnEnemies(int numEnemies)
{
  for(int i = 0; i < numEnemies; ++i)
  {
    GameObject enemy = (GameObject)GameObject.Instantiate(enemyPrefab,
      Vector3.zero, Quaternion.identity);
    GameObject enemyManagerObj = GameObject.Find("EnemyManager");
    enemyManagerObj.SendMessage("AddEnemy", enemy, SendMessageOptions.DontRequireReceiver);
  }
}
```

После оптимизации получаем следующий код:

```
public void SpawnEnemies(int numEnemies)
{
  GameObject enemyManagerObj = GameObject.Find("EnemyManager");
  EnemyManagerComponent enemyMgr = enemyManagerObj.GetComponent<EnemyManagerComponent>();
  for(int i = 0; i < numEnemies; ++i)
  {
    GameObject enemy = (GameObject)GameObject.Instantiate(enemyPrefab,
      Vector3.zero, Quaternion.identity);
```

```
    enemyMgr.AddEnemy(enemy);
  }
}
```

Еще лучше (если данный метод предполагается использовать в метода Update) предварительно кэшировать ссылку на объект EnemyManagerComponent.

Глобальные классы и игровые объекты в Unity

Рассмотренный в предыдущем разделе объект EnemyManager представляет собой глобальный объект, существующий в программе в единственном экземпляре. Для реализации таких объектов можно использовать несколько подходов, в том числе статические классы и объекты-одиночки.

Статические классы

Это решение основано на создании класса, который всегда доступен из любого места в коде. Объект этого класса существует от момента запуска приложения до его завершения. Статический класс, который функционирует во многом так же, как EnemyManagerComponent в предыдущем примере, можно определить следующим образом:

```
using System.Collections.Generic;
public static class EnemyManager
{
  static List<GameObject> enemies;
  public static void AddEnemy(GameObject enemy)
  {
    enemies.Add(enemy);
  }
  ...
}
```

Статические классы могут иметь статический конструктор для инициализации значений свойств, который гарантированно будет вызываться перед первым обращением к классу:

```
static EnemyManager()
{
  enemies = new List<GameObject>();
}
```

Впрочем, в данном случае достаточно использовать инициализатор поля enemies.

Компоненты-одиночки

Недостаток решения на основе статического класса заключается в необходимости наследования самого низкоуровневого класса — System.Object. Это означает, что статические классы не могут наследовать класс MonoBehaviour и, соответственно, использовать его инструменты, имеющие отношение к Unity, включая обработку событий и сопрограммы. Кроме того, из-за отсутствия объекта для выбора теряется возможность увидеть данные в инспекторе во время выполнения.

Однако все эти инструменты доступны в глобальных классах-одиночках, порождающих игровой объект и предоставляющих статические методы для глобального доступа:

```
using UnityEngine;
using System.Collections;

public class EnemyManager : MonoBehaviour
{
  public static EnemyManager Instance
  {
    get { return instance; }
  }

  static EnemyManager instance = null;
  static List<GameObject> enemies;
```

```
  void Awake()
  {
    if(instance)
    {
      DestroyImmediate(gameObject);
      return;
    }
    enemies = new List<GameObject>();
    instance = this;
    DontDestroyOnLoad(gameObject);
  }
  public static void AddEnemy(GameObject enemy)
  {
    enemies.Add(enemy);
  }
  ...
}
```

Для доступа к данному компоненту надо использовать статическое свойство `EnemyManager.Instance`, доступное для чтения из любого компонента игры.

Объект, связанный с данным компонентом (как и ранее рассмотренный статический класс) будет сохраняться даже при смене сцены, благодаря вызову метода `DontDestroyOnLoad` класса `UnityEngine.Object`.

Заметим, что для работы со сценами предназначен класс `SceneManager`, содержащий, в частности, метод `LoadScene(sceneName)`.