Unity3DBlog

Xavier

Unity3DBlog

Xavier

這本書的網址是 http://leanpub.com/unity3dblog

此版本發布於 2018-08-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Description

The process I learn Unity3D.
Some keypoint and note about everything.
In summary, everything I'd like to record.

Project on Github[1] Author:liarchgh[2]

```
 □□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□
 □□□□□□□□      □□□□□□□□
```

[1] https://github.com/liarchgh/Unity3DBlog
[2] https://github.com/liarchgh

Timeline

Playable Track

OnPlayableCreate

Timeline 开始时调用

OnGraphStart

Timeline 开始时调用

OnGraphStart

OnBehaviourPause

Timeline 开始时调用，如果未到此 **clip** 则调用

OnGraphStart

Timeline 开始时

OnBehaviourPause

OnGraphStop

Timeline 结束且设置为 **None**

OnGraphStop

Timeline 结束且设置为 **None**

OnBehaviourPlay

此 **clip** 开始时调用

OnBehaviourPause

DynamicBone

Root

The root of the transform hierarchy to apply physics.
根物体，动作的起点

Update Rate

Internal physics simulation rate.
更新频率，简单来说就是所控制的柔性物体的运动更新频率，测试了一下，30才算是看着比较流畅，但是帧数过高之后因为柔性物体运动太快，开不出来柔性物体形变

Update Mode

How much the bones slowed down.

- Normal
- Animate Physics
  用代码控制骨骼并且不受现有动画的影响
- Unscaled Time

Damping

How much the force applied to return each bone to original orientation.

降低柔体移动速度，可用 AnimationCurve 进行曲线定义
与 Damping 进行乘算

Elasticity(弹性)

How much bone's original orientation are preserved.
柔性物体原位置保留的比例，也可理解为恢复原来形状的速度，可类比弹性形变，越接近就越慢
与 Damping 进行乘算

Stiffness

How much character's position change is ignored in physics simulation.
使柔性物体不易形变
与 Damping 进行乘算

Inert

Each bone can be a sphere to collide with colliders. Radius describe sphere's size.
每个节点的球体碰撞器 (此碰撞器只与指定 Dynamic Bone Collider 物体碰撞，具体碰撞的物体之后由参数 Colliders 确定，只会被这些 Dynamic Bone Collider 影响，不能影响这些 Dynamic Bone Collider) 的半径
与 Damping 进行乘算

Colliders

指定具体会影响此柔性物体的 Dynamic Bone Collider

Spine

Spine.Unity.SkeletonAnimation

实现 Spine 动画的脚本，导入 Spine 时自动添加

state

类型是 Spine.AnimationState，在 Awake 中初始化

可以使用它的方法来播放动画

```
// 在 Trank0中播放“stand”动画。
skeletonAnimation.state.SetAnimation(0, “stand”, false);

// 在 Track0中添加一个“run”动画，当 Track0的最后一个动画播放结束后
skeletonAnimation.state.AddAnimation(0, “run”, true, 0f);
```

loop

动作是否循环

AnimationName

要播放的动画的名字，直接进行赋值即可控制播放什么动画，只有在赋值与正在播放动画不一致的名字（里面的字不一样，不管是不是同一字符串）才会重新播放动作

置为空串 `skeletonAnimation.AnimationName = ""` 时，会重新播放当前动画，此属性值依然是当前动画

skeleton.FlipX

控制物体水平朝向，bool 类型，取反即物体水平朝向反过来

timeScale

时间缩放，动作的实际速度是动作的初始速度呈上此参数

Track

可以设置不同 Track 的动画来同时播放多个动画，高层级的通道会覆盖低层级的 Track: 越大的通道数字就拥有越高的优先级

```
// 跑步动画运行在 Track 0，而射击动画运行在 Track 1
skeletonAnimation.state.SetAnimation(0, "run", true);
skeletonAnimation.state.SetAnimation(1, "shoot", false);
```

TrackEntry

获得有效的 TrackEntry 对象:

```
var trackEntry = skeletonAnimation.state.GetCurrent(myTra\
ckNumber); // 返回 null，就表示没有动画在运行
```

TrackEntry.Time,TrackEntry.lastTime

指定帧开始播放
Spine 的摄影表每秒30帧

```
// 从第10帧开始播放"dance" 动画
var trackEntry = skeletonAnimation.state.SetAnimation(0, \
"dance", false);
trackEntry.Time = 10f/30f;

// 你可以像这样更方便得设置 Time，而不需要使用另一个变量去储存 Trac
skeletonAnimation.state.SetAnimation(0, "dance", false).T\
ime = 10f/30f;

// 如果你这么做事为了动画事件，请确保 lastTime 和.Time 设置了一样的
在 Time0和.Time 之间的所有事件都将被捕获，并在下一个 Update 中增加/
```

TrackEntry.animationEnd

动画的结束时间点，具体是指动画未缩放时的时间

TrackEntry.TimeScale

播放速度，SkeletonAnimation.timeScale 是最后修改的播放速度
你可以将 timeScale 设置为0来暂停播放。要知道，即使你将 timeScale = 0来暂停骨骼的运动，但是每一帧的骨骼动画仍然存在，同时你对骨骼得任何更改都将会覆盖更新。

Spine.Animation

每个 Spine.Animation 是 Timeline 对象的集合

Spine.AnimationState

Start

动画开始时调用 (循环：第一次动画开始时)

End

动画结束时调用 (循环：最后一次动画结束时)

Complete

Dispost

Interrupt

Event

材质

Spine-Unity 也使用材质存储信息，包括纹理、着色器和必要的材质属性。该材质通过 `AtlasAsset` 分配。

`MeshRenderer` 的材质数组是由 `SkeletonRenderer` 管理的，每一帧都取决于 `AtlasAssets` 需要用到什么。直接修改该数组并不是 Unity 典型的设置方法。

更多的材质

你可能注意到在你的 `MeshRenderer` 中有很多材质，特别是，比你实际设置的 `AtlasAsset` 还要多。

如果你有一个以上的图集页是不正常的。渲染器的材质数组不能体现 AtlasAsset 中每一项的顺序和编号。SkeletonRenderer 根据那些需要被渲染的 Spine 附件材质规划了一个材质数组。

如果所有附件共享一个材质，SkeletonRenderer 只会把这个材质放进 MeshRenderer。

如果一些附件用到了材质 A 和一些材质 B，材质数组会根据材质的需要去排列顺序。这是基于附件绘制的顺序以及哪些附件可以在哪些材质纹理中。

更多的材质，就表示会有更多的 draw calls。

为每个实例设置材质属性

同样的，改变 MeshRenderer.material 的值是没用的。

`Renderer.material` 属性只是渲染器生成的副本，但是它会立即被 SkeletonRenderer 的渲染代码给覆盖。

另一方面，`Renderer.sharedMaterial` 会修改原始材质。如果你使用这个材质生成更多的 Spine 游戏对象，对于它的修改应用会对所有的实例进行修改。

在这个例子中，Unity 的Renderer.SetPropertyBlock[3]是有用的方法。记住，`SkeletonRenderer` 和 `SkeletonAnimation` 都使用 `MeshRenderer`。设置 MeshRenderer 的 MaterialPropertyBlock 允许你改变渲染器的属性值。**MaterialPropertyBlock mpb = new MaterialPropertyBlock();**
mpb.SetColor("_FillColor", Color.red); // **"_FillColor"** 是假设的着色器名字。
GetComponent<MeshRenderer>().SetPropertyBlock(mpb);

优化的注意事项

[3] http://docs.unity3d.com/ScriptReference/Renderer.SetPropertyBlock.html

- 使用 Renderer.SetPropertyBlock 允许具有相同材质的渲染器去处理那些由不同的 MaterialPropertyBlocks 改变的材质属性。
- 当你在 MaterialPropertyBlock 中增加或改变一个属性值的时候,你需要调用 `SetPropertyBlock`。但是你可以把 MaterialPropertyBlock 作为类的一部分，所以每当你想改变属性时，不必总是实例化一个新的
- 如果你需要频繁设置一个属性，你可以使用静态方法:`Shader.PropertyToID(string)` 去缓存一个整数 ID，这个 ID 可以代替 String，使 MaterialPropertyBlock 的 Setter 可以使用该 ID 去设置属性。

Spine 的模型、mesh、贴图结构

一个 Spine 是一个 mesh，此 mesh 中每个物件是一个互相连通的多边形（三角形组成），有一些简单物件是矩形，通过 UV 在贴图上取颜色、透明度。贴图中，矩形物体的部分一般只有中间是物体的部分是不透明的，旁边都是透明的。这样，再设置好各个物件在 mesh 中的顺序，即可实现各个物件叠加起来形成整个 Spine 模型的渲染，各个物件的遮挡关系也接着 mesh 中的三角形顺序正确实现。

分层和排序

Sorting Layer 和 **Sorting Order** 属性其实是在 `SkeletonRenderer/SkeletonAnimat` 的 Inspector 中，实际上它只是修改了 `MeshRenderer` 的sorting layer[4] 和 sorting order[5] 属性.

[4] http://docs.unity3d.com/ScriptReference/Renderer-sortingLayerID.html
[5] http://docs.unity3d.com/ScriptReference/Renderer-sortingOrder.html

尽管被隐藏在 MeshRenderer 的 Inspector 中，这些属性实际上是 `MeshRenderer`serialized/stored 的一部分，而不是 SkeletonRenderer。

Sorting Layer 中越下方的层渲染时在越上面，会遮盖掉 **Sorting Layer** 上面的层的物体，同一层物体靠距离摄像机距离判定

Spine 模型的大小

Spine 使用 1像素:1单位。缩放默认设置为0.01。

水平翻转 or 垂直翻转

访问 Skeleton.FlipX 和 Skeleton.FlipY 可以允许骨架的水平翻转和垂直翻转。

不推荐直接旋转和负数缩放。

- 不均匀的缩放会导致一个网格绕过 Unity 的配料系统。这意味着每个实例都会有它自己的 draw calls。所以对于你的主要角色这没什么问题。如果你的不均匀缩放骨架的规模为数十个，它就是有害的。
- 旋转会导致法线随着网格旋转。对于2D 精灵的光照，这意味着它们会指向错误的方位。
- 旋转 X 或者 Y 也可能会导致 Unity 的2D 碰撞发生不可预知的结果。
- 负比例的缩放会导致附加的物理碰撞和一些脚本逻辑发生不可预料的结果。

作为 UI 时注意导入应使用 SkeletonGraphic，否则添加到 UI 层不正常

Spine.Unity.SkeletonGraphic

可以用于 UI 的 Spine ### SkeletonData #### Animations 所有动画的集合

Refs

- 官方中文 Unity 文档[6](API 的名字都不对，怕是落后了好几个版本)
- 官方 API 英文文档[7]

[6] http://zh.esotericsoftware.com/spine-unity
[7] http://zh.esotericsoftware.com/spine-api-reference

脚本生命周期

Unity 一生命周期

转载自:Unity 一生命周期[8]

一、下面我们来学习下脚本生命周期常用的10个脚本函数:

（1）Reset() 组件重设为默认值时（只用于编辑状态）
（2）Awake() 脚本组件载入时（调用一次）
（3）OnEnable() 是在游戏对象可以调用时调用
（4）Start() 第一个 Update 发生之前（调用一次）
（5）FixedUpdate() 固定时间调用，常用于物理相关的计算，比如对 Rigidbody 的操作
（6）Update() 大部分游戏行为代码被执行的地方，除了物理代码
（7）LateUpdate() 每帧 Update 调用之后
（8）OnGUI() 绘制 GUI 时调用
（9）OnDisable() 当对象设置为不可用时
（10）OnDestroy() 组件销毁时调用

二、下面看在程序运行后的结果

1、代码：{#1、代码：}

[8] https://www.jianshu.com/p/8c353abb42e4

```
void
Awake
()
{
        Debug.Log(
"Awake----1"
);
    }

void
Start
()
{
        Debug.Log(
"Start----3"
);
    }

void
OnEnable
()
{
        Debug.Log(
"OnEnable----2"
);
    }


void
LateUpdate
()
{
        Debug.Log(
"LateUpdate----6"
);
```

```
36      }
37
38  void
39  FixedUpdate
40  ()
41  {
42          Debug.Log(
43  "FixedUpdate----4"
44  );
45      }
46
47  void
48  Update
49  ()
50  {
51          Debug.Log(
52  "Update----5"
53  );
54      }
55
56
57  void
58  OnGUI
59  ()
60  {
61          Debug.Log(
62  "OnGUI----7"
63  );
64      }
65
66  void
67  OnDisable
68  ()
69  {
70          Debug.Log(
```

```
"OnDisable----8"
);
    }

void
OnDestroy
()
{
        Debug.Log(
"OnDestroy----9"
);
    }
```

2、从下图得出，不管代码顺序如何的写都是按照以下顺序执行的：{#2、从下图得出，不管代码顺序如何的写都是按照以下顺序执行的：}

Awake ->OnEable-> Start -> -> FixedUpdate-> Update -> LateUpdate ->OnGUI -> OnDisable ->OnDestroy

运行结果一：

这是在暂停的情况下

运行结果二：

退出运行时所调用的

这时在运行时，去掉在 Inspector 场景中 Cube 前面的勾后，还会调用 OnDisable.

Paste_Image.png

三、生命周期的作用：

1.Awake:
用于在游戏开始之前初始化变量或游戏状态。在脚本整个生命周期内它仅被调用一次.Awake 在所有对象被初始化之后调用，所以你可以安全的与其他对象对话或用诸如 GameObject.FindWithTag() 这样的函数搜索它们。每个游戏物体上的 Awake 以随机的顺序被调用。因此，你应该用 Awake 来设置脚本间的引用，并用 Start 来传递信息 Awake 总是在 Start 之前被调用。它不能用来执行协同程序。
2.Start:
仅在 Update 函数第一次被调用前调用。Start 在 behaviour 的生命周期中只被调用一次。它和 Awake 的不同是 Start 只在脚本实例被启用时调用。你可以按需调整延迟初始化代码。Awake 总是在 Start 之前执行。这允许你协调初始化顺序。在所有脚本实例中，Start 函数总是在 Awake 函数之后调用。
3.FixedUpdate:
固定帧更新，在 Unity 导航菜单栏中，点击“Edit”—>“Project Setting”—>“Time”菜单项后，右侧的 Inspector 视图将弹出时间管理器，其中“Fixed Timestep”选项用于设置 FixedUpdate() 的更新频率，更新频率默认为0.02s。
4.Update:
正常帧更新，用于更新逻辑。每一帧都执行，处理 Rigidbody 时，需要用 FixedUpdate 代替 Update。例如: 给刚体加一个作用力时，你必须应用作用力在 FixedUpdate 里的固定帧，而不是 Update 中的帧。(两者帧长不同)FixedUpdate，每固定帧绘制时执行一次，和 update 不同的是 FixedUpdate 是渲染帧执行，如果你的渲染效率低下的时候 FixedUpdate 调用次数就会跟着下降。FixedUpdate 比较适用于物理引擎的计算，因为是

跟每帧渲染有关。Update 就比较适合做控制。

5.LateUpdate:

在所有 Update 函数调用后被调用，和 fixedupdate 一样都是每一帧都被调用执行，这可用于调整脚本执行顺序。例如: 当物体在 Update 里移动时，跟随物体的相机可以在 LateUpdate 里实现。LateUpdate, 在每帧 Update 执行完毕调用，他是在所有 update 结束后才调用，比较适合用于命令脚本的执行。官网上例子是摄像机的跟随，都是在所有 update 操作完才跟进摄像机，不然就有可能出现摄像机已经推进了，但是视角里还未有角色的空帧出现。

6.OnGUI:

在渲染和处理 GUI 事件时调用。比如：你画一个 button 或 label 时常常用到它。这意味着 OnGUI 也是每帧执行一次。

7.Reset:

在用户点击检视面板的 Reset 按钮或者首次添加该组件时被调用。此函数只在编辑模式下被调用。Reset 最常用于在检视面板中给定一个默认值。

8.OnDisable:

当物体被销毁时 OnDisable 将被调用，并且可用于任意清理代码。脚本被卸载时，OnDisable 将被调用，OnEnable 在脚本被载入后调用。注意：OnDisable 不能用于协同程序。

9.OnDestroy:

当 MonoBehaviour 将被销毁时，这个函数被调用。OnDestroy 只会在预先已经被激活的游戏物体上被调用。注意: OnDestroy 也不能用于协同程序。

如若不理解，可参考一下：{# 如若不理解，可参考一下：}

http://blog.csdn.net/akof1314/article/details/39323081[9]
http://blog.csdn.net/zhaoguanghui2012/article/details/49121103[10]
https://docs.unity3d.com/Manual/ExecutionOrder.html[11]

[9] https://link.jianshu.com?t=http://blog.csdn.net/akof1314/article/details/39323081
[10] http://blog.csdn.net/zhaoguanghui2012/article/details/49121103
[11] https://docs.unity3d.com/Manual/ExecutionOrder.html

Execution Order of Event Functions

Execution Order of Event Functions

In Unity scripting, there are a number of event functions that get executed in a predetermined order as a script executes. This execution order is described below:

First Scene Load

These functions get called when a **scene**
starts (once for each object in the scene).

- **Awake**: This function is always called before any Start functions and also just after a **prefab** is instantiated. (If a GameObject is inactive during start up Awake is not called until it is made active.)
- **OnEnable**: (only called if the Object is active): This function is called just after the object is enabled. This happens when a MonoBehaviour instance is created, such as when a level is loaded or a **GameObject** with the script **component** is instantiated.
- **OnLevelWasLoaded**: This function is executed to inform the game that a new level has been loaded.

Note that for objects added to the scene, the Awake and OnEnable functions for *all* **scripts**
will be called before Start, Update, **etc**

are called for any of them. Naturally, this cannot be enforced when an object is instantiated during gameplay.

Editor

- **Reset**: Reset is called to initialize the script' s properties when it is first attached to the object and also when the *Reset* command is used.

Before the first frame update

- **Start**: Start is called before the first frame update only if the script instance is enabled.

For objects added to the scene, the Start function will be called on all scripts before Update, etc are called for any of them. Naturally, this cannot be enforced when an object is instantiated during gameplay.

In between frames

- **OnApplicationPause**: This is called at the end of the frame where the pause is detected, effectively between the normal frame updates. One extra frame will be issued after **OnApplicationPause** is called to allow the game to show graphics that indicate the paused state.

Update Order

When you' re keeping track of game logic and interactions, animations, **camera**
positions, etc., there are a few different events you can use. The common pattern is to perform most tasks inside the **Update** function, but there are also other functions you can use.

- **FixedUpdate**: **FixedUpdate** is often called more frequently than **Update**. It can be called multiple times per frame, if the frame rate is low and it may not be called between frames at all if the frame rate is high. All physics calculations and updates occur immediately after **FixedUpdate**. When applying movement calculations inside **FixedUpdate**, you do not need to multiply your values by **Time.deltaTime**. This is because **FixedUpdate** is called on a reliable timer, independent of the frame rate.
- **Update**: **Update** is called once per frame. It is the main workhorse function for frame updates.
- **LateUpdate**: **LateUpdate** is called once per frame, after **Update** has finished. Any calculations that are performed in **Update** will have completed when **LateUpdate** begins. A common use for **LateUpdate** would be a following third-person camera. If you make your character move and turn inside **Update**, you can perform all camera movement and rotation calculations in **LateUpdate**. This will ensure that the character has moved completely before the camera tracks its position.

Rendering

- **OnPreCull**: Called before the camera culls the scene. Culling determines which objects are visible to the camera. OnPreCull is called just before culling takes place.
- **OnBecameVisible/OnBecameInvisible**: Called when an object becomes visible/invisible to any camera.
- **OnWillRenderObject**: Called **once** for each camera if the object is visible.
- **OnPreRender**: Called before the camera starts **rendering** the scene.
- **OnRenderObject**: Called after all regular scene rendering is

done. You can use GL[12] class or Graphics.DrawMeshNow[13] to draw custom geometry at this point.

- **OnPostRender**: Called after a camera finishes rendering the scene.
- **OnRenderImage**: Called after scene rendering is complete to allow post-processing of the image, see Post-processing Effects[14].
- **OnGUI**: Called multiple times per frame in response to GUI events. The Layout and Repaint events are processed first, followed by a Layout and keyboard/mouse event for each input event.
- **OnDrawGizmos** Used for drawing **Gizmos** in the **scene view** for visualisation purposes.

Coroutines

Normal coroutine updates are run after the Update function returns. A coroutine is a function that can suspend its execution (yield) until the given YieldInstruction finishes. Different uses of Coroutines:

- **yield** The coroutine will continue after all Update functions have been called on the next frame.
- **yield WaitForSeconds** Continue after a specified time delay, after all Update functions have been called for the frame
- **yield WaitForFixedUpdate** Continue after all FixedUpdate has been called on all scripts
- **yield WWW** Continue after a WWW download has completed.
- **yield StartCoroutine** Chains the coroutine, and will wait for the MyFunc coroutine to complete first.

[12] https://docs.unity3d.com/ScriptReference/GL.html
[13] https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshNow.html
[14] https://docs.unity3d.com/Manual/PostProcessingOverview.html

When the Object is Destroyed

- **OnDestroy**: This function is called after all frame updates for the last frame of the object's existence (the object might be destroyed in response to Object.Destroy or at the closure of a scene).

When Quitting

These functions get called on all the active objects in your scene:

- **OnApplicationQuit**: This function is called on all game objects before the application is quit. In the editor it is called when the user stops playmode.
- **OnDisable**: This function is called when the behaviour becomes disabled or inactive.

Script Lifecycle Flowchart

The following diagram summarises the ordering and repetition of event functions during a script's lifetime.

C# 委托

使用委托（C# 编程指南）

委托[15]是安全封装方法的类型，类似于 C 和 C++ 中的函数指针。与 C 函数指针不同的是，委托是面向对象的、类型安全的和可靠的。委托的类型由委托的名称确定。以下示例声明名为 `Del` 的委托，该委托可以封装采用字符串[16]作为参数并返回 void[17] 的方法：C#

```
public delegate void Del(string message);
```

委托对象通常通过提供委托将封装的方法的名称或使用匿名方法[18]构造。对委托进行实例化后，委托会将对其进行的方法调用传递到该方法。调用方传递到委托的参数将传递到该方法，并且委托会将方法的返回值（如果有）返回到调用方。这被称为调用委托。实例化的委托可以按封装的方法本身进行调用。例如:C#

[15] https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/delegate

[16] https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/string

[17] https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/void

[18] https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-methods

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

与循环结合

注意!!! 循环变量请不要直接在委托里面使用！因为等委托真实执行的时候才会去访问循环变量，而此时循环已经结束，然后访问到的就是结束后的循环变量，吃了大亏！

错误的使用方式:

错误的使用方式:

参考:

- MSDN 委托（C# 编程指南）[19]

[19] https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/delegates/

XLua

加载 lua

- 运行 **lua** 语句

  `luaenv.DoString("print('success')");`
- 使用 **require** 运行文件中 **lua** 语句

  `luaenv.DoString("require 'LuaCode'");` 注意：默认只能读取名称为”*.lua.txt” 的文件
- 自定义 **loader**

```
luaenv.AddLoader(delegate(ref string filepath)
{
  string code = Resources.Load(filepath).ToString();
  return System.Text.Encoding.UTF8.GetBytes(code);
});
// 加载 Resources 文件夹中的 LuaCode.lua.txt 或者 LuaCode.lua.xml
luaenv.DoString("require('LuaCode.lua')");
```

C# 访问 Lua

- 通过 **`LuaEnv.Global`** 获取一个全局基本数据类型

  `string a = luaenv.Global.Get<string>("a");`

SUIFW

生命周期

```
#region  窗体生命周期
//窗口从隐藏到显示时调用
public override void Display()
{
    base.Display();
}

//从 Freeze 状态到重新显示时调用
public override void Redisplay()
{
    base.Redisplay();
}

//窗体显示在其他窗体下面
public override void Freeze()
{
    base.Freeze();
}

//页面隐藏（不在“栈”集合中）
public override void Hiding()
{
    base.Hiding();
}
#endregion
```

```
CloseUIForm();
```

关闭当前窗口（调用此方法的窗口）

```
RigisterFullPathObjectEvent("Btn", p =>
ClickBtn());
```

Btn 按钮点击后调用 ClickBtn() 方法

```
CurrentUIType.UIForms_Type =
UIFormType.Normal;
```

设置当前窗口类型

```
1   //UI 窗体（位置）类型
2   public enum UIFormType
3   {
4       //普通窗体
5       Normal,
6       //固定窗体
7       Fixed,
8       //弹出窗体
9       PopUp
10  }
```

CurrentUIType.UIForms_ShowMode =
UIFormShowMode.HideOther;

设置当前窗口显示方式

```
//UI 窗体的显示类型
public enum UIFormShowMode
{
    //普通
    Normal,
    //反向切换
    ReverseChange,
    //隐藏其他
    HideOther
}
```

窗体透明度

```
//UI 窗体透明度类型
public enum UIFormLucenyType
{
    //完全透明，不能穿透
    Lucency,
    //半透明，不能穿透
    Translucence,
    //低透明度，不能穿透
    ImPenetrable,
    //可以穿透
    Pentrate
}
```

`UIManager.GetInstance().ShowUIForms(UIProConst.Form`

加载指定 Form 窗口

与脚本生命周期关系

Display

窗体开始:`Awake()` → `OnEnable()` → `Start()`

参考:

- Unity UI 框架 SUIFW——LiuGuozhu[20]
- 【Unity 插件】Event System —Dispatcher 事件系统插件官方文档翻译[21], pdf[22]

[20] http://www.cnblogs.com/LiuGuozhu/tag/UnityUI%E6%A1%86%E6%9E%B6/
[21] http://blog.xudawang.fun/2018/05/19/event-system-dispatcher-cn/
[22] http://blog.xudawang.fun/wp-content/uploads/2018/05/2018051901053669.pdf

RectTransform

Anchors

规定基本显示区域，即Anchor Presets 的边框参数乘以父物体的Anchor Presets 规定出的最终长宽

Pivot

中心，这一点的位置处于Anchor Presets 规定的位置，如果规定了的话参数乘以父物体的Anchor Presets 规定出的最终长宽

Anchor Presets

根据Anchors 规定的基本区域以不同方式规定出最终的显示区域

GPU Instancing

要利用 GPU Instancing，则必须使用相同的材质，并传递额外的参数到着色器，如颜色，浮点数等。

C# 协变和逆变

注意

只有引用类型才支持使用泛型接口中的变体。值类型不支持变体。

参考

- 协变和逆变 (C#)[23]

[23] https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/concepts/covariance-contravariance/

Mask

Image

Shader

Transparent

Shader

时间 _Time float4 Time (t/20, t, t*2, t*3), use to animate things inside the shaders.

_SinTime float4 Sine of time: (t/8, t/4, t/2, t).

_CosTime float4 Cosine of time: (t/8, t/4, t/2, t).

unity_DeltaTime float4 Delta time: (dt, 1/dt, smoothDt, 1/smoothDt).
`_Time.x = time / 20 _Time.y = time _Time.z = time * 2 _Time.w = time * 3`

属性面板

前缀 [Toggle] 会使属性面板上显示为类似布尔型的选框
它的值要么是0要么是1 `[Toggle]_Start("if Start", int) = 0`

疑惑

企图使用_Start 置为真的时候记录开始位置失败，这一行根本不执行，前后语句都会执行，很奇怪。 `_TimeStart += (_Time.y - _TimeStart) * (1-_Start);`

Miscellaneous

协程

一个返回值为 **IEnumerator** 的函数

```
IEnumerator Cro()
{
    while (True)
    {
        yield return new WaitForSeconds(1);
    }
}
```

将其作为协程启动

```
StartCoroutine(Cro());
```

停止指定名称的协程

```
StopCoroutine("Cro");
```

只能停止没有参数的协程

停止此脚本启动的全部协程

```
StopAllCoroutines();
```

协程中使用协程

```
IEnumerator Cro0()
{
    while (True)
    {
        // Cro0等待 Cro1执行完再继续执行
        yield return StartCoroutine(Cro1());
    }
}
```

会产生垃圾回收

简单一个移动物体的脚本产生了125K 的 GC，需要注意一下
使用 `yield return null;` 会好一些，因为没有返回值了

自定义类型转换

隐式转换

节约了几个方法重载的代码量，懒得挨个写了

```
public static implicit operator 目标类型（被转化类型 变量参数）
{
   return 目标类型结果;
}
```

显式转换

```
    public static explicit operator 目标类型（被转化类型 变量参数）
    {
       return 目标类型结果;
    }
```

运算符重载

节约了几个方法重载的代码量，懒得挨个写了

```
    public static OperatorTest operator + (OperatorTest o\
1, OperatorTest o2)
    {
        OperatorTest o = new OperatorTest();
        o.Value = o1.Value + o2.Value;
        return o;
    }
```

Lambda 表达式

可以和委托结合起来

```
    delegate int del(int i);
    static void Main(string[] args)
    {
        del myDelegate = x => x * x;
        int j = myDelegate(5); //j = 25
    }
```

基本形式

```
    (input-parameters) => expression
```

零个参数

```
() => SomeMethod()
```

一个参数

```
    (x) => x * x
```

只有一个输入参数时，括号是可选的

```
    x => x * x
```

多个参数

```
    (x, y) => x == y
```

有时，编译器难以或无法推断输入类型，可以显式指定类型

```
    (int x, string s) => s.Length > x
```

引用类型作为参数时不能使用 **ref** 修饰

修改层级关系

UI 同一层越靠下越后渲 “‘ go.SetSiblingIndex(index);

```
## Serializable 脚本
使用```[System.Serializable]``而不是直接继承 ScriptableObject 的脚本放
amespace 里有问题 (2018.1.0f2)

## Mask 顺序
Mask 的渲染顺序就是材质设定的渲染顺序

## 随机数

### 一个指定范围的随机整数
返回一个 u 到 x 范围内的随机整数，范围包含 u 不包含 x
```Random.Range(u, x)```

### 对 List<> 随机排序
```

```
new System.Random().Shuffle(aList);
```

```
## [使用集合初始值设定项初始化字典](https://docs.microsoft.com/zh-cn/d
net/csharp/programming-guide/classes-and-structs/how-to-i\
nitialize-a-dictionary-with-a-collection-initializer)
```

```
Dictionary<int, StudentName> students = new Dictionary<in\
t, string>()
{
    { 111, "1"},
    { 112, "2"},
};
```
```

```
## 按钮事件

### Raycast Target
Image 或者 Text 等组件上有```Raycast Target```选项，选中的话当前物体会
向自己的下方传递。

### EventTrigger 和 ScrollRect 冲突
现象：使用 EventTrigger 给 ScrollRect 中的 Content 下的元素添加点击事件
ect 无法滚动，好像是需要点击非子物体的位置才行（没有验证）。
解决方法：自己新建了一个 ButtonWithLongPress 类，继承自 Button，同时还
调用 IsPressed() 获取是否时按下状态，自己实现长按事件，点击直接使用
评价：比较勉强的一个解决方法，并不完美，只是刚好能用，复用性较差。

## 自动布局组件
### Horizontal Layout Group（水平布局）
### Vertical Layout Group（垂直布局）
### Grid Layout Group （网格布局）

## 动态变化
使用 DOTween 插件，最开始我使用的协程自己实现，很麻烦，这个插件简单
```DOTween.To()```

## 序列化
- 将私有变量显示在监视面板上
- 将对象存储在本地，即持久化存储
注意私有成员变量的私有成员变量也需要添加`[SerializeField]`标签，自定
ss`前添加`[Serializable]`

{bump-link-number}

{leanpub-filename="cmiscellaneous.md"}

# C#Miscellaneous

## List
```
```

```
### 传值复制

- ToArray()
```

List<string> t = new List<string>(); //original List<string> t2 = new List<string>(t.ToArray()); // copy of t “‘ - ForEach

```
- ConvertAll()
```

namespace Delegates { class X { public int Id { get; set; } public string Name { get; set; } }

```
class Y
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        List<X> x = new List<X>();
        for (int i = 0; i < 100; i++)
            x.Add(new X { Id = i, Name = string.Format("x\
_{0}", i.ToString()) });
        // copy x to y
        List<Y> y = new List<Y>(x.ConvertAll<Y>(e => { re\
turn new Y { Id = e.Id, Name = e.Name }; }));
        Debug.Assert(x.Count == y.Count);
    }

}
```

} “‘

变为只读的

使用 List 的 AsReadOnly 方法，只是类型也会变

```
    List<int> a = new List<int> {1, 2, 3, 4, 5};
    IList<int> b = a.AsReadOnly();
```

自定义排序

```
    list.Sort((info0, info1) =>
    {
        return info0.CompareTo(info1);
    });
```

文件系统组织的路径字符串中分隔符

```
    Console.WriteLine("Path.AltDirectorySeparatorChar={0}\
",
        Path.AltDirectorySeparatorChar);
```

UGUI Rich Text

b

效果：加粗
例子：

```
<b>Hello World</b> Hello World
```

i

效果：斜体
例子：

```
<i>Hello World</i> Hello World
```

size

效果：大小
例子：

```
<size=40>Hello World</size> Hello World
```

color

效果：颜色
名称对应的颜色：

例子：

- 使用单词指定颜色 `<color=purple>Hello World</color>`

  Hello World 
- 使用 RGB 指定颜色 `<color=#ff0000>Hello World</color>`

  Hello World 
- 使用 RGBA 指定颜色和透明度 `<color=#ff000055>Hello World</color>` Hello World 

material

效果:材质,适用于 text mesh 中例子:`<material=1>TestTest</material>`

复杂一些的应用

```
<b><i><size=600><color=#ff000088>Hello World</color>
</size></i></b>
Hello World
```

Refs

UGUI 使用富文本 Rich Text - 简书[24]
Unity - Manual: Rich Text[25]
【转】(八) unity4.6Ugui 中文教程文档——-概要-UGUI Rich Text - 博客园[26]

[24] https://www.jianshu.com/p/e9efe7e6e02b
[25] https://docs.unity3d.com/Manual/StyledText.html
[26] https://www.cnblogs.com/slysky/p/4301676.html