获取所有相同类型组件：

using System.Linq;

获取所有button并监听

FindObjectsOfType<Button>().ToList().ForEach(btn => {

btn.onClick.AddListener(() => {

addToShowText(btn.transform.GetComponentInChildren<Text>().text);

### **1 控制位置**

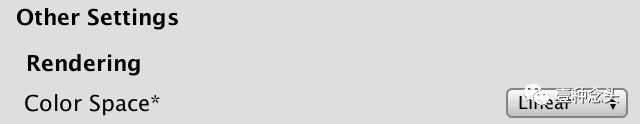
许多游戏的玩法都是基于一个角色需要四处走动从而达成目标的。玩家的任务是引导角色。动作游戏通常通过按键或转动操作杆来操纵角色，从而为你提供直接控制。点击游戏可让你指示目标位置，角色会自动移动到该位置。编程游戏可让你编写角色执行的指令。等等。

在本教程系列中，我们将重点介绍如何在3D动作游戏中控制角色。我们从简单的开始，在一个小的平面矩形上滑动一个球体。一旦我们熟练掌握了这点，将来我们就可以使其变得更加复杂。

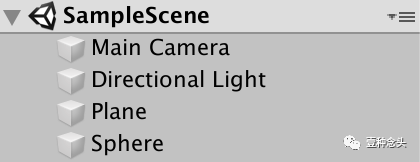
### **1.1 设置场景**

从一个新的默认3D项目开始。尽管你可以使用自己选择的渲染管线，但此时我们并不需要Package Manager。

我始终使用线性颜色空间，但你可以在项目设置中通过“Edit Project Settings Player / Other Settings”进行配置。

线性颜色空间

默认的SampleScene场景具有一个主摄像头和一个定向光，我们将保留它们。创建一个代表地面的平面以及一个球体，两者均位于原点。默认球体的半径为0.5，因此将其Y坐标设置为0.5，使其看起来像位于地平面的上面。

场景层次

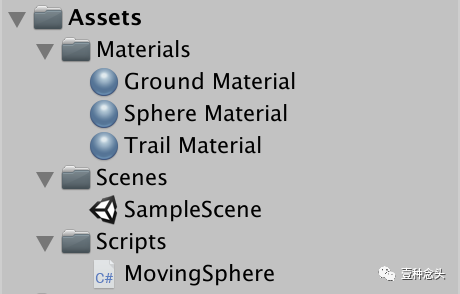
我们将自己限制为在地面上进行2D运动，因此我们把摄像机向下放置在平面上方，以便在游戏窗口中清晰地看到游戏区域。再将其投影模式设置为正交。脱离了透视之后，使我们能够看到2D运动而不会变形。

正交相机 视野朝下

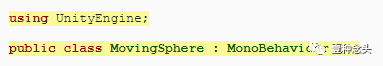
唯一还有影响的是球体的阴影。通过将灯光的“阴影类型”设置为“None”来摆脱它。

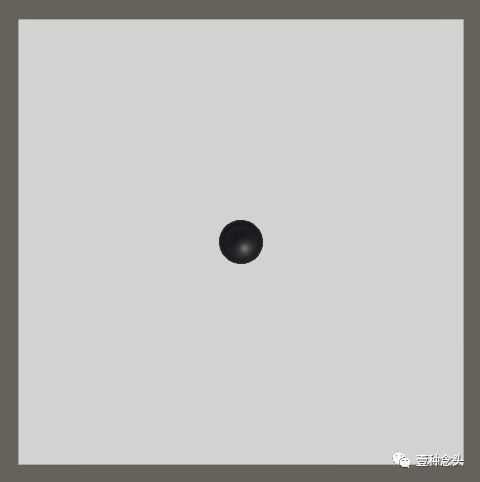
IMG_259光不使用阴影

为地面和球体创建材质，并根据需要进行配置。我将球体设为黑色，将地面变暗为浅灰色。还将通过Trail来可视化运动，因此也要为此创建材质。这里将使用一种淡红色的材质。最后，我们需要一个MovingSphere脚本来实现移动。

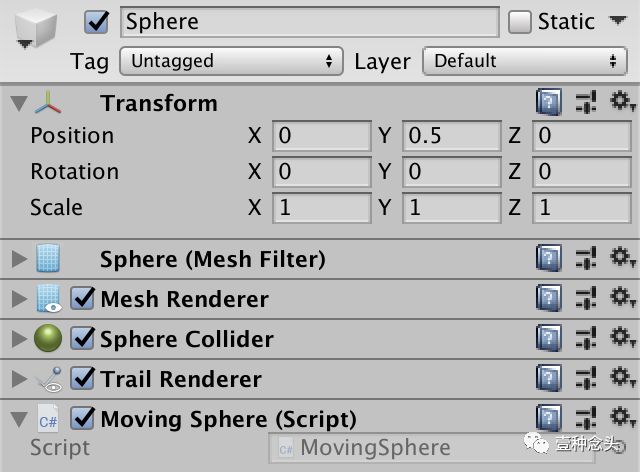
项目的Asset

该脚本可以从MonoBehaviour的空继承开始。

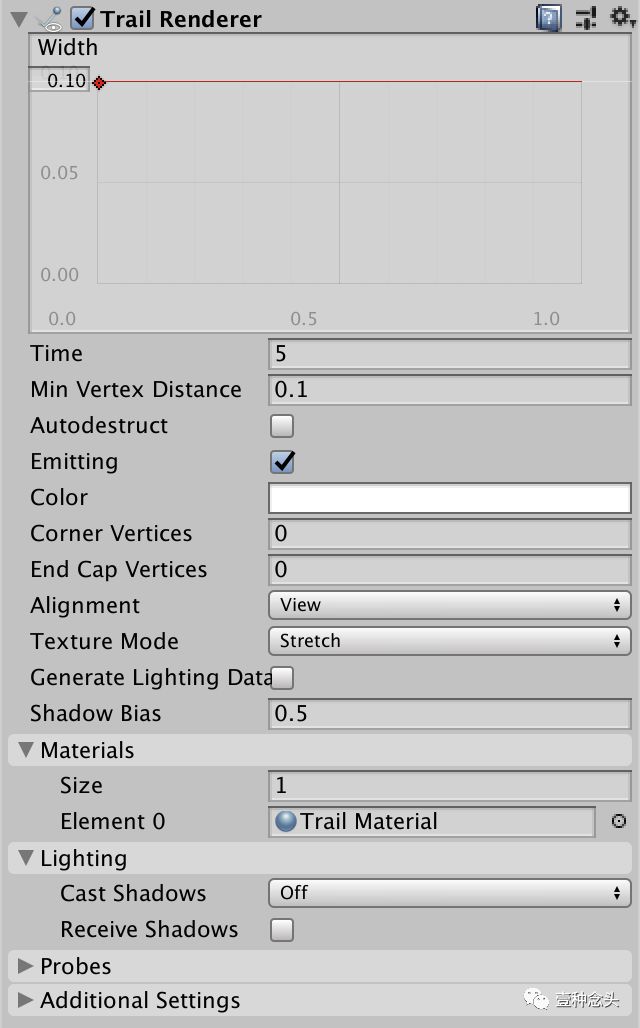


Game 视图

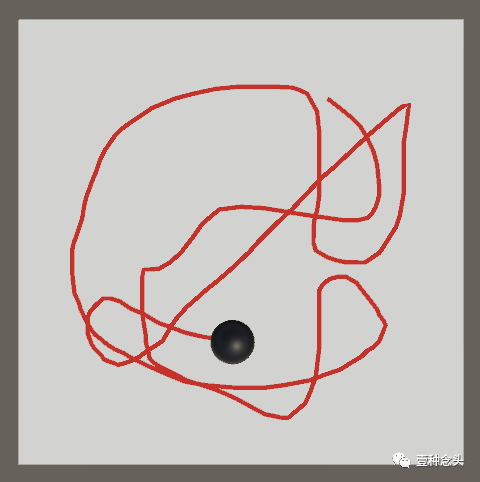
将TrailRenderer和我们的MovingSphere组件都添加到球体中。保持其他一切不变。

球体 绑定Component

将trail 材质 分配给TrailRenderer组件的Materials数组的第一个也是唯一的元素。它并不需要cast shadows，虽然并不是必需要禁用，但是我们还是把它顺手关掉吧。除此之外，将“Width ”从1.0减小到更合理的值（例如0.1），这将生成一条细线。

Trail Renderer

尽管我们现在尚未编码任何运动，但可以通过进入播放模式并在场景窗口中移动球体来预览其外观了。

移动的拖尾效果

### **1.2 读取玩家输入**

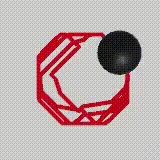
要移动球体，我们需要能够读取玩家的输入命令。可以在MovingSphere的Update方法中执行此操作。玩家输入为2D，因此我们可以将其存储在Vector2变量中。最初，我们将其X和Y分量都设置为零，然后使用它们将球体定位在XZ平面中。因此，输入的Y分量成为位置的Z分量。Y位置保持零。



从玩家的输入中检索方向性最简单方法是调用方法Input.GetAxis。默认情况下，Unity定义了水平和垂直输入轴，你可以在项目设置的“Input”部分中进行检查。我们将水平值用于X，将垂直值用于Y。

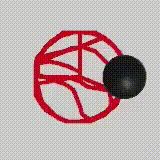
IMG_267

默认设置将这些轴链接到箭头和WASD键。输入值也经过调整，因此按键的行为有点像操纵杆。你可以根据需要调整这些设置，但我保留默认设置。



使用箭头或WASD键

两个轴都有第二个定义，将它们链接到操纵杆或左控制杆的输入。这样可以使输入更加流畅，但是除了下面这个展示之外，其他的我都会使用按键的模式。



使用控制杆

为什么不使用Input System包？  
你可以使用，但是原理是相同的。我们所需要做的就是检索两个轴值。另外，在撰写本文时，该软件包仍处于预览状态，因此尚未正式发布并受支持。

### **1.3 归一化输入向量**

轴在静止时返回零，而在极限时返回-1或1。当我们使用输入来设置球体的位置时，它被约束为具有相同范围的矩形。至少，按键输入就是这种情况，因为每次按键是独立的。如果是摇杆，则是连续的，通常我们在任何方向上都被限制为距原点的最大距离为1，从而将位置限制在一个圆内。

控制器输入的优点是，无论方向如何，输入向量的最大长度始终为1。因此，各个方向的移动速度都可以一样快。按键不是这种情况，单个按键的最大值为1，而同时按下两个按键的最大值为√2，这意味着对角线移动最快。

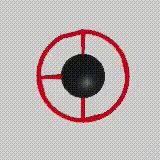
由于Pythagoream（毕达哥拉斯）定理，键的最大值为√2。轴值定义直角三角形两侧的长度，组合矢量为假设。因此，输入向量的大小为

IMG_270

。

通过将输入矢量除以其大小，可以确保矢量的长度永远不会超过1。结果始终为单位长度矢量，除非其初始长度为零，但在这种情况下，结果是未知的。此过程称为归一化向量。我们可以通过对向量调用Normalize来实现此目的，该向量会自行缩放并在结果不确定时变为零向量。

IMG_271



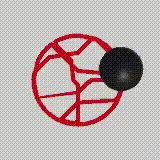
归一化按键输入

### **1.4 约束输入向量**

始终对输入向量进行归一化会将位置限制为始终位于圆上，除非输入中断了，而这时，小球最终会回到原点。原点和圆之间的线代表一个框架，球从中心跳到圆上或从圆上回退到原点。

全有或全无的输入可能是偏理想化的，但如果我们想让圆内的所有位置也有效的话， 可以仅通过调整输入矢量（如果其大小超过1）来做到这一点。一种方便的方法是调用静态Vector2.ClampMagnitude方法而不是Normalize，并使用向量（最大值为1）作为参数。结果是一个相同或缩小到所提供最大值的向量。

IMG_273



约束按键输入

## **2 控制速度**

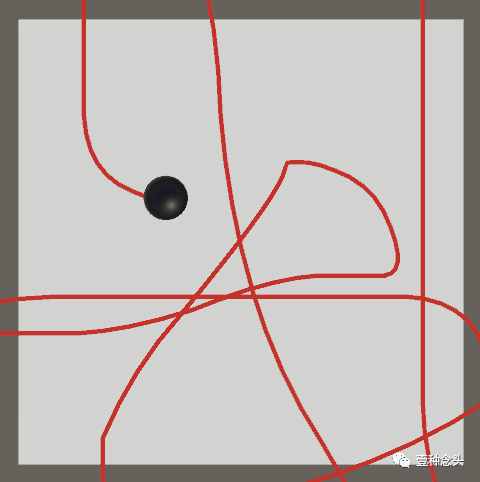
到目前为止，我们一直在直接使用输入来设置球体的位置。这意味着，当输入的向量i 改变时，球体的位置p 立即变为相同的值。因此， p = i。这不是适当的运动，它应该是隐形传送。一种更自然的控制球体的方法是，通过将位移矢量d 与其旧位置p 0相加来确定其下一个位置p 1，因此p1 = p0 + d。

### **2.1 相对运动**

通过使用d = i而不是p = i，我们可以让输入和位置之间的关联变的不那么直接。这样就消除了位置上的约束，因为它现在是相对于自身而不是固定的原点。因此，该位置由一个无限的迭代序列

IMG_275

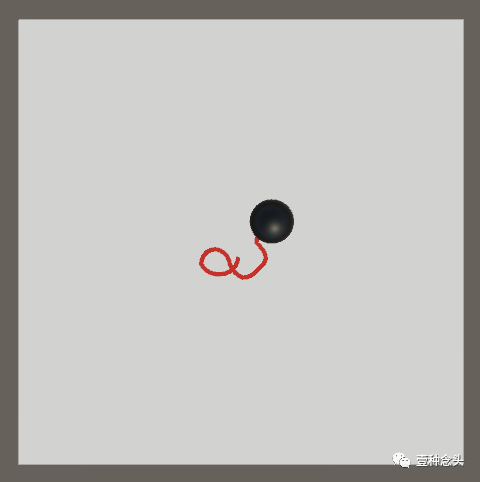
描述，其中 p0被定义为起始位置。

IMG_276相对运动

### **2.2 速率**

我们的球体确实可以移动到任何地方，但是它是如此之快以至于难以控制。这是因为因为每个 update 都添加输入向量的结果。帧速率越高，速度就越快。为了获得一致的结果，我们不希望帧率影响我们的输入。如果我们使用恒定的输入，则我们希望得到恒定的位移，而不用考虑潜在波动的帧速率。

出于我们的目的，一个帧代表一个持续时间：从上一帧的开始到当前帧之间经过了多少时间 t，可以通过Time.deltaTime访问。因此，我们的位移实际上是 d = it，但这里，我们错误地假设t 是常数。

IMG_278控制速度与帧速率无关

### **2.3 速度**

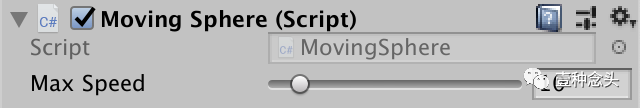
我们的最大输入向量的大小为1，代表每秒一米的速度，等于每小时3.6公里，大约每小时2.24英里。那不是很快。

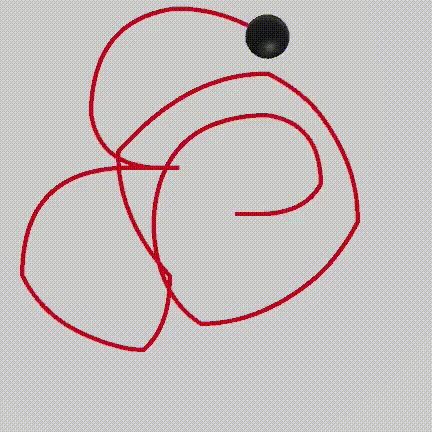
可以通过缩放输入向量来提高最大速度。比例因子表示最大速度，即没有方向的速度。为它添加一个带有SerializeField属性的maxSpeed字段（默认值为10），并为它提供Range属性（例如1–100）。

IMG_280

SerializeField做什么？  
它告诉Unity对字段进行序列化，这意味着它已保存并在Unity编辑器中公开，因此可以通过检查器进行调整。我们也可以公开该字段，但是通过这种方式，该字段仍然不受MovingSphere类之外的代码的影响。

将输入向量和最大速度相乘以找到所需的速度。

IMG_281



最大速度设置为10

### **2.4 加速度**

由于我们可以直接控制速度，因此可以立即进行更改。仅输入系统应用的过滤会稍微减慢更改的速度。实际上，速度不能立即改变。变更工作需要一定的精力和时间，就像变更职位一样。速度的变化率称为加速度a，它导致

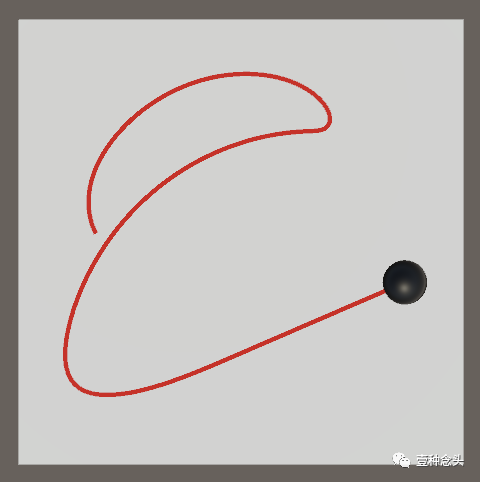
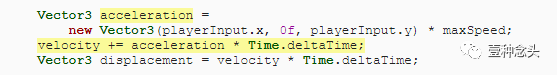
IMG_284

t，其中v0为零向量。减速只是与当前速度相反的加速度，因此不需要特殊处理。

看看如果使用输入矢量直接控制加速度而不是控制速度会发生什么。这需要我们跟踪当前速度，因此将其存储在一个字段中。

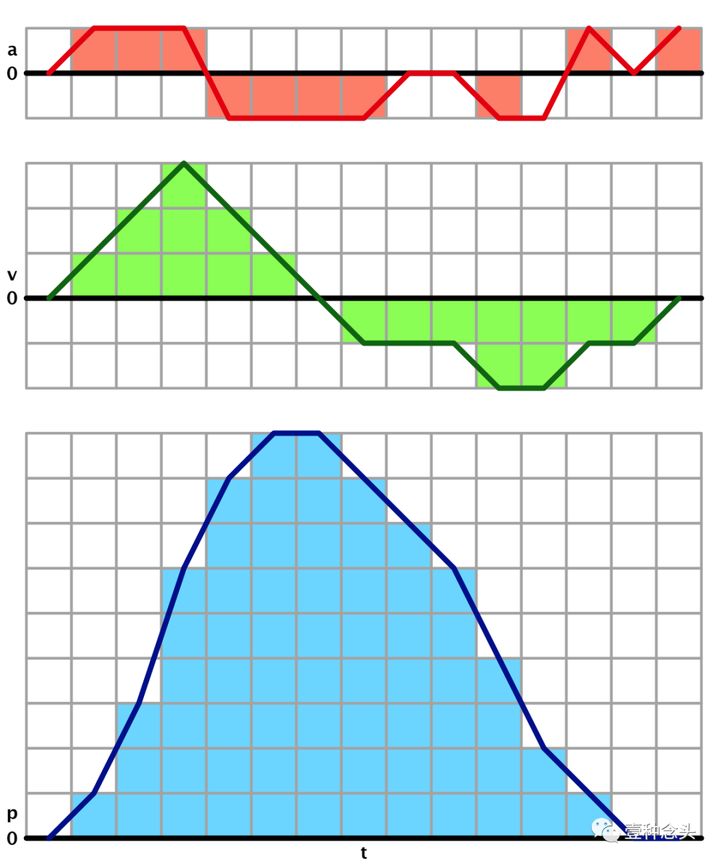
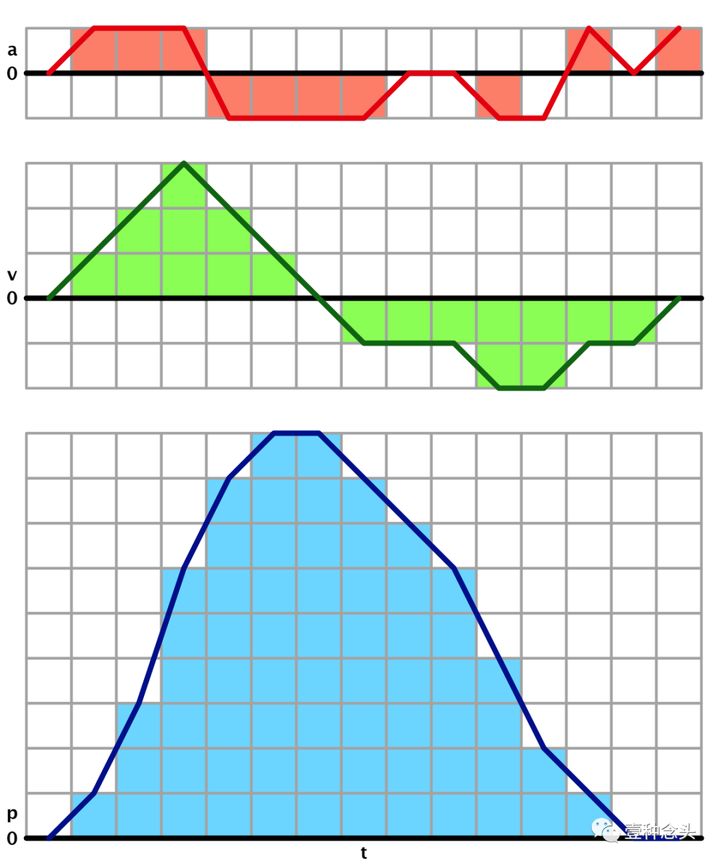
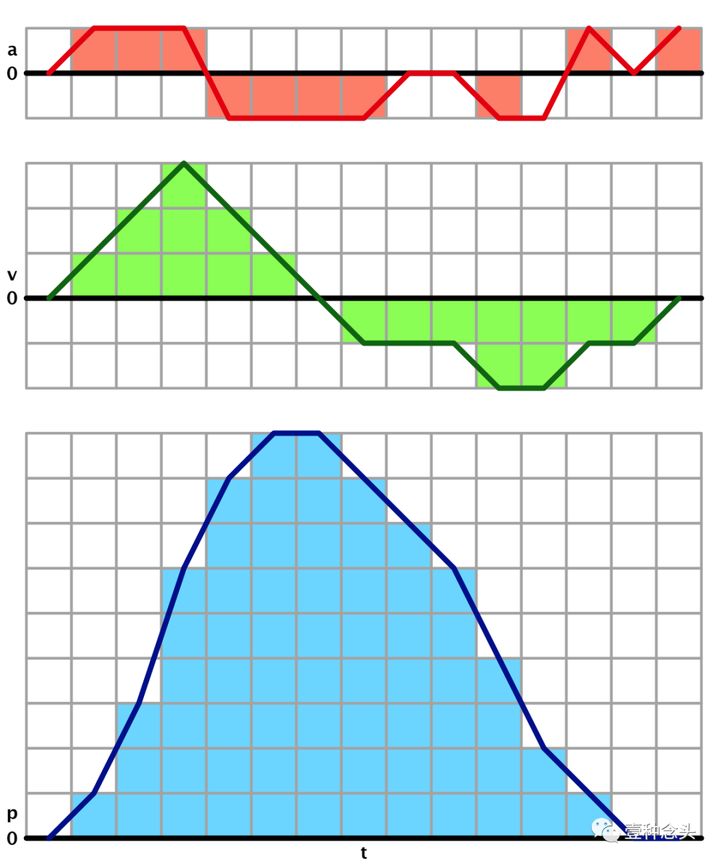
IMG_285

现在，输入向量在Update中定义了加速度，但现在让我们继续将其与maxSpeed相乘，暂时将其重新解释为最大加速度。然后将其加到速度上，然后计算位移。

平滑的速度变化

### **2.5 所需速度**

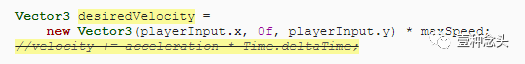
控制加速度而不是速度会产生更平滑的运动，但同时也会削弱我们对球体的控制。就像我们开车和步行的情况一样。在大多数游戏中，需要对速度进行更直接的控制，因此让我们回到这种方法。但是，施加加速度确实会产生更平滑的运动。

加速度改变速度，从而改变位置

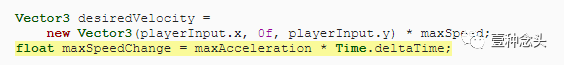
我们可以通过直接控制目标速度并将加速度应用于实际速度，直到它与期望的速度相匹配，来结合这两种方法。然后，我们可以通过调整球的最大加速度来调整球的响应速度。为此添加一个可序列化的字段。

IMG_291

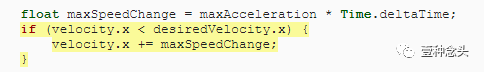
在Update中，我们现在使用输入矢量来定义所需的速度，而不再使用旧方法来调整速度。



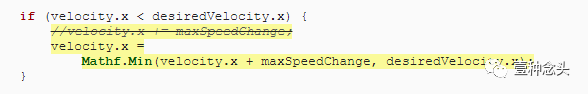
相反，我们首先将最大加速度乘以 t。这就是我们能够更改此更新速度的程度。



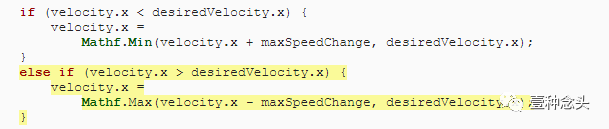
首先，我们仅考虑速度的X分量。如果小于期望值，则添加最大更改。



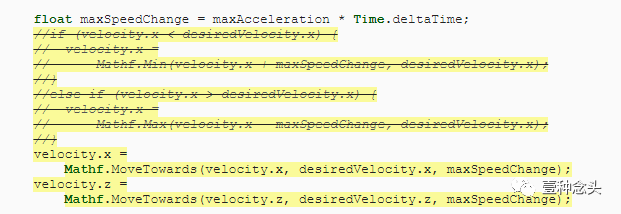
这可能会导致过冲，我们可以通过选择增加和期望的最小值来防止这种过冲。我们可以在此处使用Mathf.Min方法。

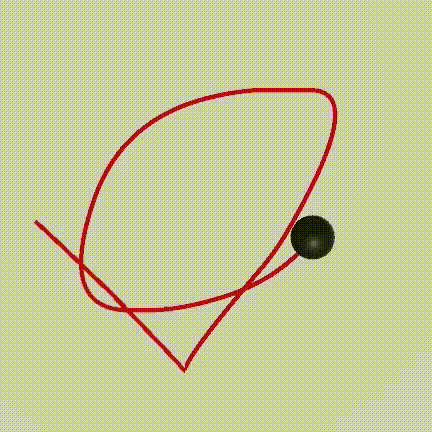


或者，速度可能大于所需速度。在这种情况下，我们减去最大变化，并通过Mathf.Max取最大变化和所需值。



我们还可以通过方便的Mathf.MoveTowards方法执行所有操作，将当前值和所需值以及最大允许的变化传递给它。分别对X和Z组件执行此操作。





最大速度和加速度都设置为10

现在，我们可以调整最大加速度，以在平滑运动和响应能力之间达成所需的权衡。