1. 2D to 3D Effect
   1. 问题综述：本项目为一款2D游戏，但部分交互需要展现3D效果，如借助透视原理产生的近大远小的效果，以加强交互视觉体验，这就需要根据具体需求实现2D视角向3D视角的转换。以下分别以打地鼠、羌笛音游两个部分举例：
      1. HitMole: sortingLayer, draw sequence
         1. 需求概述：实现人物从地鼠洞中间“穿出”的透视效果。
         2. 方案阐述：通过改变鼠洞与人物的渲染层级(Sorting Layer)关系，实现打地鼠游戏的透视效果。
         3. 应用示例：



图(1): 地鼠洞.sortingLayer = 0，角色.sortingLayer = 1

(角色层次高，显示在地鼠洞之前)



图(2): 在图(1)基础上，新增一个sortingLayer == 2的”半个地鼠洞”图片



图(3): 将其拼接至适当位置，即可实现”地鼠穿出”的效果

* + - 1. Sorting Layer详解：

Unity中的渲染顺序自上而下大致分为三层。 最高层为Camera层，可以在Camera的depth那里设置，设置之后，图形的渲染顺序就是先绘制depth低的相机下的物体，再绘制depth高的相机下的物体，也就是说，depth高的相机会覆盖depth低的相机（具体的覆盖关系有don’t clear， solid color等等几种）

比Camera层稍低一层的是sorting layer层， 随便找一个可以设置sorting layer的地方，选择sorting layer，点添加按钮，就可以看到当前所有的sorting layer，并且可以更改sorting layer的顺序，排位靠后的sorting layer会覆盖排位靠前的sorting layer。 设置好sorting layer的相互关系之后，就可以给任何一个继承于Renderer类，或者有renderer的子类作为field的对象设置sorting layer了。 注意这些sorting layer的遮挡关系是在同一个camera的层级下的。 不同camera下的renderer渲染顺序以camera的depth为准。 有的component的sorting layer可以直接在unity editor里面设置，比如Sprite Renderer。 有的则需要用代码来设置，比如设置Particle system的sorting layer， 就需要在代码中取到 ParticleSystem.Renderer.SortingLayer 来进行设置。

比sorting layer再低一层的是sorting order， 这个数字指代的是在同一个sorting layer下的渲染顺序。

需要注意不要混淆的是gameobject的layer，和renderer的sorting layer。 gameObject的layer个人理解是一个逻辑上的分层，用于camera的culling mask等。 而renderer的sorting layer则用于渲染。只有继承与renderer或者有renderer作为filed的component才需要设置sorting layer。

简单总结一下，决定Unity渲染关系的层级顺序是：Camera > sorting layer > sorting order

* 1. QiangPipe: Scaling up and speeding up every elapsed time
     1. 需求概述：实现音符运动过程中的透视效果，即：近大远小、近快远慢
     2. 方案阐述：在逐帧函数中，根据一定时间间隔，等比增大音符物体的尺寸，同时加快其沿y轴的反方向的运动速度。
     3. 应用实例：

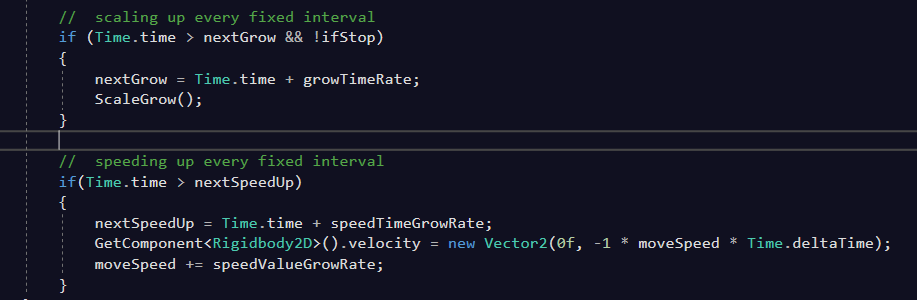


图(1): 音符位置较远时：尺寸较小、速度较慢



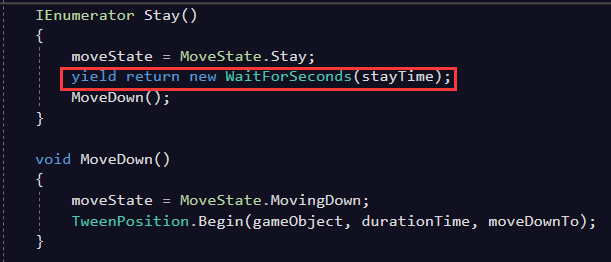
图(2): 音符位置较近时：尺寸增大、速度加快

* + 1. 方案详解：

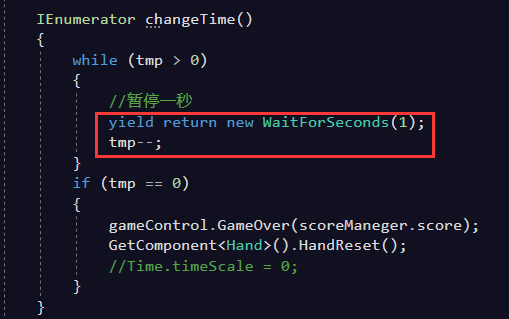


如上图代码示例，每隔一定时间间隔(growTimeRate, speedTimeGrowRate)，分别对尺寸和速度(沿-y轴方向)进行一定比例的增大，由于音符物体始终在朝-y方向运动，随着时间的推移，位置由远至近，与此同时，尺寸由小至大、速度由慢至快，实现近大远小、近快远慢的效果。

1. Coroutine:
   1. 需求概述：游戏中常遇到需要延迟调用某方法、某功能在执行前或执行过程中需要暂缓若干帧或若干秒再继续执行的情况，如倒计时、角色运动模式的切换等。
   2. 方案阐述：上述需求，可总结为“分部执行，遇到条件挂起，直到条件满足才会被唤醒继续向后执行”，可统一使用Coroutine(协程)来进行处理。
   3. 应用示例：



图(1): 打地鼠游戏中，角色“钻出后”停留一段时间(stayTime)，再“钻回”



图(2): 打地鼠与音乐游戏中的倒计时功能

* 1. Coroutine 深入：

1. 什么是协调程序：unity协程是一个能暂停执行，暂停后立即返回，直到中断指令完成后继续执行的函数。它类似一个子线程单独出来处理一些问题，性能开销较小，但是他在一个MonoBehaviour提供的主线程里只能有一个处于运行状态的协程。
2. 协同程序的特点：
   1. 协程在中断指令(Yield Instruction)产生时暂停执行；
   2. 协程一暂停执行便立即返回（中断协程后返回主函数，暂停结束后继续执行协程剩余的函数）；
   3. 中断指令完成后从中断指令的下一行继续执行；
   4. 同一时刻、一个脚本实例中可以有多个暂停的协程，但只有一个运行着的协程；
   5. 函数体全部执行完后，协程结束；
   6. 协程可以很好的控制跨越一定帧数后执行的行为；
   7. 协程在性能上、相比于一般函数几乎没有更多的开销
3. 创建一个协程函数

IEnumerator methodName(Object parameter1,Object parameter2,...){

// to do something

yield return YieldInstruction/other/null;

// to do something else

}

IEnumerator methodName(Object parameter1,Object parameter2,...){

// to do something

yield return YieldInstruction/other/null;

// to do something else

}

注意：协同函数的返回值的类型必须是Coroutine，Coroutine继承与Yieldinstruction。

所以协同程序的返回类型就只能是null,等待的时间，等待的帧数。由此可见WWW 也是实现了Coroutine的。

1. 开始一个协同程序
2. 通过MonoBehaviour提供的StartCoroutine方法来实现启动协同程序：StartCoroutine(IEnumerator routine);
   1. 优点：灵活，性能开销小。
   2. 缺点：无法单独的停止这个协程，如果需要停止这个协程只能等待协同程序运行完毕或则使用StopAllCoroutine();方法。
3. StartCoroutine (methodName:string, value : object = null);
   1. 优点：可以直接通过传入协同程序的方法名来停止这个协程：StopCoroutine(string methodName);
   2. 缺点：性能的开销较大，只能传递一个参数。
4. 停止协同程序
   1. StopCoroutine(string methodName);
   2. StopAllCoroutine();
   3. 设置gameobject的active为false时可以终止协同程序，但是再次设置为true后协程不会再启动。

6.协同程序的执行顺序：开始协同程序 -> 执行协同程序 -> 中断协同程序（中断指令）-> 返回上层继续执行->中断指令结束后继续执行协同程序剩下的内容