# 一、Game Play

## 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：

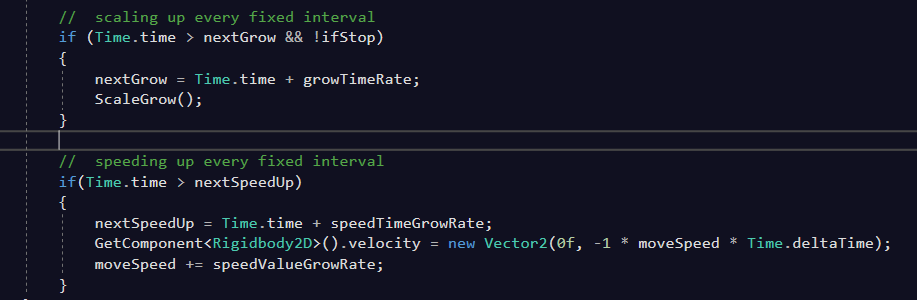


图(1): 音符位置较远时：尺寸较小、速度较慢



图(2): 音符位置较近时：尺寸增大、速度加快

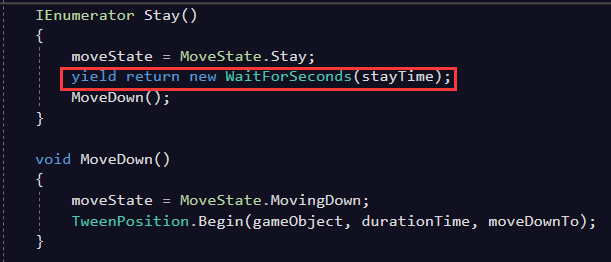
* + 1. 方案详解：



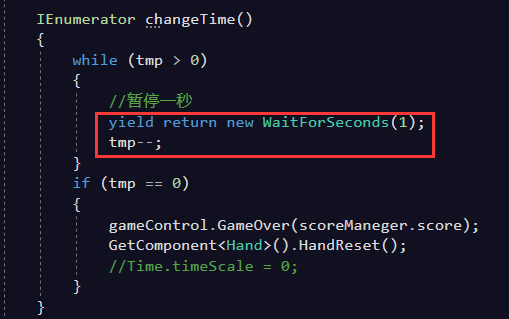
如上图代码示例，每隔一定时间间隔(growTimeRate, speedTimeGrowRate)，分别对尺寸和速度(沿-y轴方向)进行一定比例的增大，由于音符物体始终在朝-y方向运动，随着时间的推移，位置由远至近，与此同时，尺寸由小至大、速度由慢至快，实现近大远小、近快远慢的效果。

## 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
   1. 创建协程函数：

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

}

* 1. 开始协同程序
     1. 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。
     2. 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。

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. 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。
    2. 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

## Surface Shader

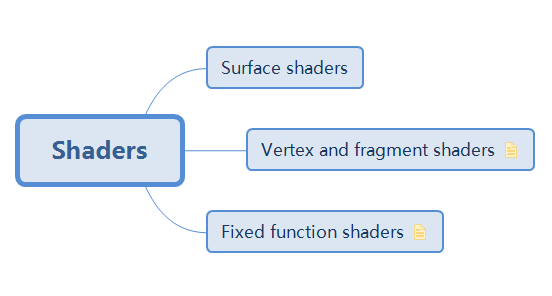
1. 需求概述：迷宫游戏中，需要表现出在山洞中具有实时探照灯的光影效果。
2. 方案阐述：主要思路应从以下两方面着手：
   1. 光源类型：根据需求与参考图示，应选取Point Light(点光源)作为光照类型；
   2. 着色器：编写shader脚本使游戏物体能够与光进行交互，从而产生光影效果。
3. 应用实例：







1. Shader 深入：
   1. 概述：



图(1) Shader主要结构

Shader（着色器）实际上就是一小段程序，它负责将输入的Mesh（网格）以指定的方式和输入的贴图或者颜色等组合作用，然后输出。绘图单元可以依据这个输出来将图像绘制到屏幕上。输入的贴图或者颜色等，加上对应的Shader，以及对Shader的特定的参数设置，将这些内容（Shader及输入参数）打包存储在一起，得到的就是一个Material（材质）。之后，我们便可以将材质赋予合适的renderer（渲染器）来进行渲染（输出）了。

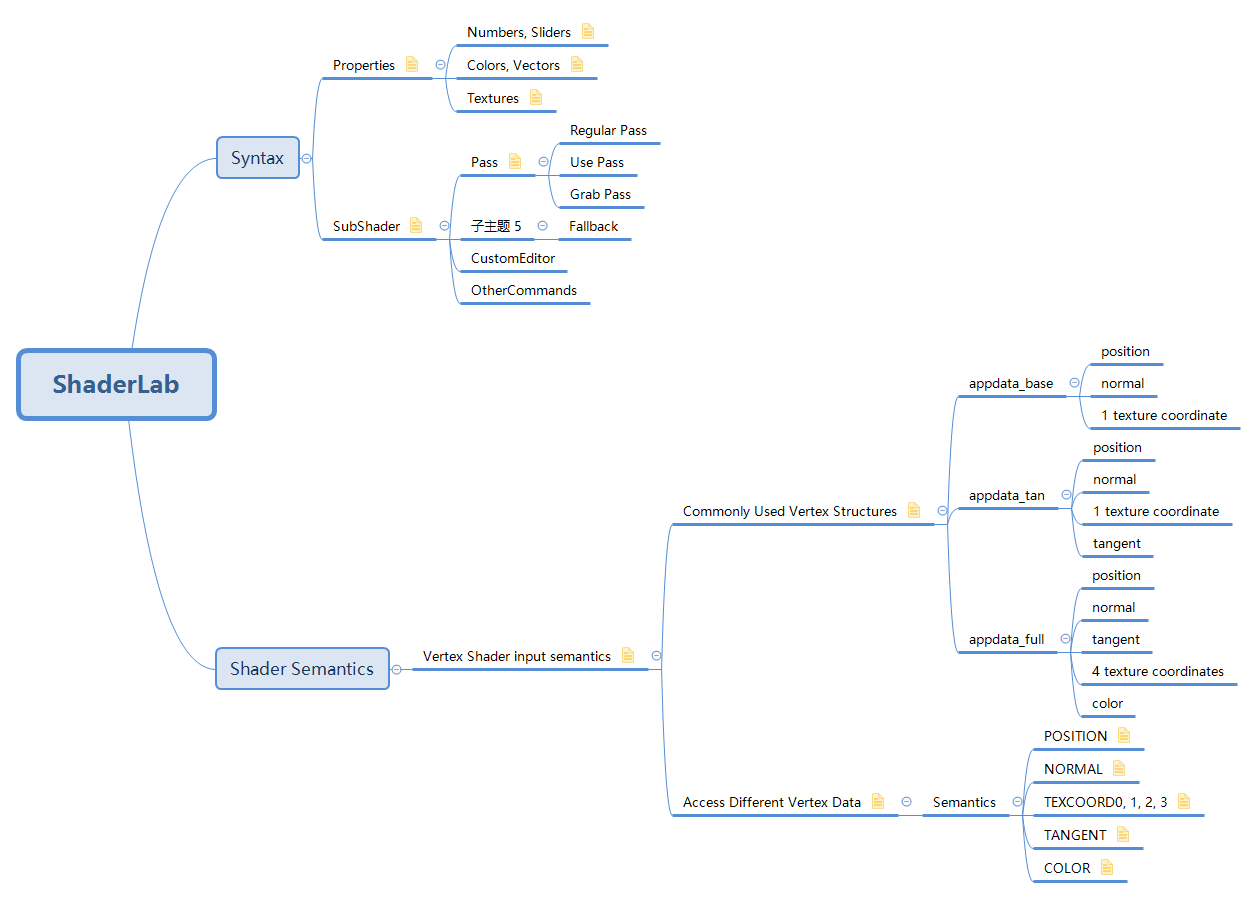
如上图所示，Unity中Shader主要分为三类：Surface Shader(表面着色器)、Vertex and Fragment Shader(顶点与片段着色器)以及Fixed Function Shader(固定方法着色器)。

Surface Shader: Unity中的Surface Shaders是一种代码生成方法，与使用低级别vertex/fragment shader(顶点/片源着色器)相比，可以更轻松地编写光照着色器。 请注意，表面着色器中没有自定义语言，它只产生所有必须手写的重复代码。 开发者仍然可在HLSL中编写着色器代码。Surface Shader实际上是封装了CG语言，隐藏了很多光照处理的细节，它的设计初衷是为了让用户仅仅使用一些指令（#pragma）就可以完成很多事情，并且封装了很多常用的光照模型和函数，例如Lambert、Blinn-Phong等。

Vertex and Fragment Shader:

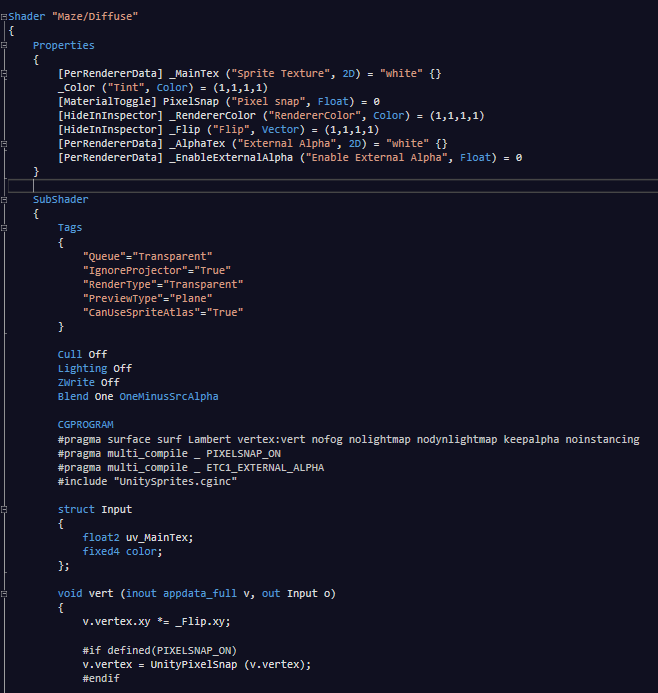
Vertex Shader: 顶点着色器是一段在3D模型的每个顶点上运行的程序。

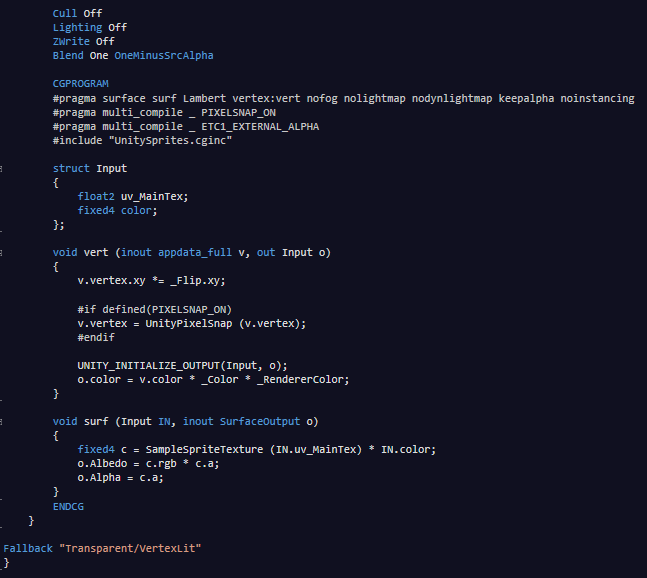
Fragment Shader: 片段着色器是这样一段程序：它运行在对象占用屏幕上的每个像素上，并且通常用于计算和输出每个像素的颜色。



图(2) ShaderLab语法结构

* 1. 漫反射着色器的编写：





完成着色器的编写后，创建材质并添加该着色器，再将新建的材质附着到目标物体上，添加点光源，即完成物体与光源的交互。但该着色器仍存在一些缺陷：光源作用方向始终面向正前方，因此光影效果只能在相机与物体之间的范围内才能得到表现。

关于光源的选取，还需具备对Unity中光源类型及其应用的相关知识：

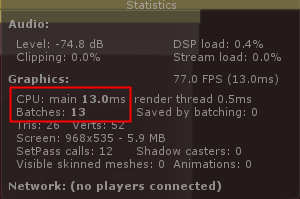
1. 光源类型：
   1. Point Lights(点光源)：点光源位于空间的一个点上，并且均匀地向各个方向发出光线。 光线撞击表面的方向是从接触点回到光线物体中心的线。 强度随着距光线的距离而减小，在指定范围内达到零。 光强度与距光源距离的平方成反比。 这被称为“反平方律”，与光在现实世界中的表现类似。
   2. Spot Lights(聚光灯)：像点光一样，聚光灯具有特定的位置和范围，光在该位置和范围内脱落。 然而，聚光灯被限制在一个角度，导致锥形的照明区域。 锥体的中心指向灯光对象的前（Z）方向。 灯光的锥体边缘光线也会减弱。 加宽角度会增加锥体的宽度，并增加该渐变的大小，即“半影”。
   3. Directional Lights(平行光/方向光)：类型太阳光，平行光可以被认为是无限远处存在的远距离光源。 它不具有任何可识别的光源位置，因此灯光对象可以放置在场景中的任何位置。 场景中的所有物体都被照亮，就好像光始终来自相同的方向一样。 光线与目标物体的距离没有规定，所以光线不会减弱。
   4. Area Lights(区域光)：区域光由空间中的矩形定义。 光在所有方向上均匀地在它们的表面区域上发射，但仅从矩形的一侧发射。 区域光的范围没有手动控制，但强度在距离光源远处的反平方时会减小。 由于照明计算的处理器密集程度相当高，因此区域光在运行时不可用，并且只能映射到光照贴图中。
2. 常见用途：
   1. Point Lights: 点光源可用于模拟场景中的灯和其他本地光源。常用于作为可照亮周围环境的荧光或爆炸光效。
   2. Spot Lights: 聚光灯通常用于人造光源，如手电筒，汽车大灯和探照灯。
   3. Directional Lights: 平行光可以用来模拟太阳或月亮。 在虚拟世界中，平行光可用来在不精确指定光线来自哪里的情况下为对象添加逼真的阴影。
   4. Area Lights: 区域光可用于创造一个逼真的路灯或靠近玩家的一盏灯。 小面积的光可以模拟较小的光源（例如室内照明），但具有比点光更真实的效果。

综上所述，根据项目实际需求，应选用Point Light(点光源)，创建一片可照亮周围一小部分环境的荧光效果。

## Pack Sprites

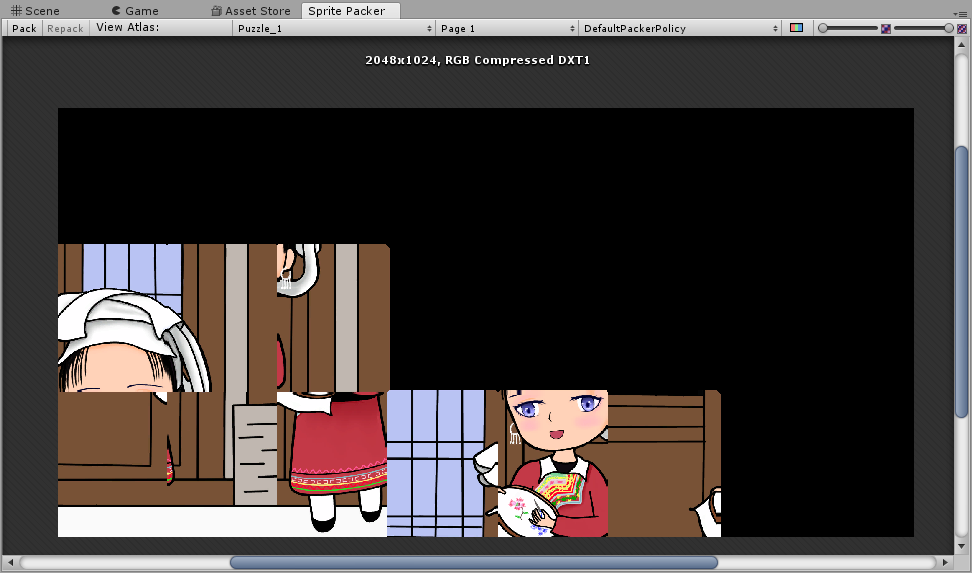
* 1. 需求概述：对精灵图片进行批处理，降低缓存损耗，获得更佳性能。
  2. 方案阐述：使用 Sprite Packer 插件，将多个精灵图片打包封装为一个Atlas(图集)。
  3. 应用实例：
     1. 打包前：



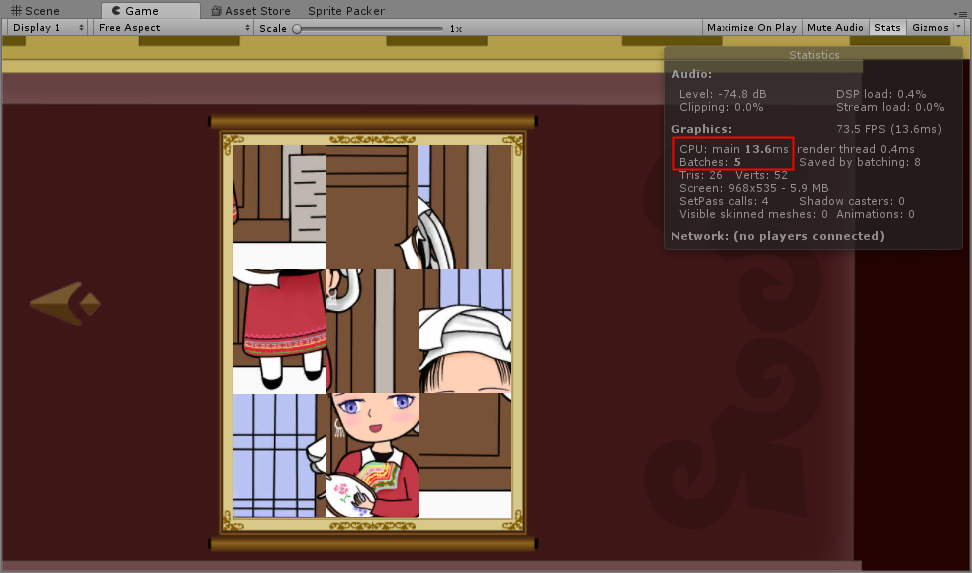


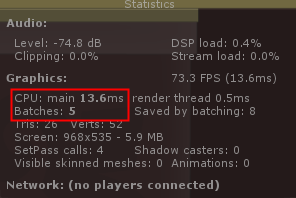
**Batches(Draw Call): 13**

* + 1. 使用 Sprite Packer 对指定图片进行打包：



* + 1. 打包后：





**Batches(Draw Call): 5**

* 1. 深入：
     1. 对比结论：由上图对比分析可得出结论：将多张单独的精灵图片经过Sprite Packer 打包后，Batches 数量明显减少，有效地降低了性能损耗。（但由于本项目规模较小，Draw Call 数量上限较低，即便数量明显降低了，对CPU耗时的影响也十分微弱。不过性能优化是必要需求，即使在性能优化部分，本项目并非典型示例，也应着重指出、深入理解。）
     2. Draw Call: Unity生成一帧的画面的处理过程大致过程是：引擎经过可见性测试，确定摄像机可以看到的物体，然后把这些物体的顶点（包括顶点位置、法线、uv 等),索引（如何组成三角形）,变换（物体的位置 旋转 缩放 以及摄像机的位置）相关光源，纹理，渲染方式（由材质/Shader决定）等数据准备好 然后通知图形API（或者就简单地看作是通知GPU）开始绘制，GPU基于这些数据经过一些列的运算，在屏幕上画出组成图形的三角形，构成一幅画。在unity 中每次引擎装备数据并通知GPU的过程为一次Draw Call。
     3. Batch: Batch 其实是Draw Call的另外一种称呼。每一次的Draw Call 可视为是CPU把一个Batch里的数据（物体的顶点数据）传输给GPU，由GPU绘制到屏幕上。
     4. 优化原理：
        1. 减少空白区域：在进行2D贴图时，比较直接和简单的方法是分别对每个需要贴图的游戏物体都赋予一张单独的精灵图片。但对于单张精灵图片，其内部常存在一些无用的空隙，这些空隙将造成运行时空间浪费。为了获得更佳性能，最好能够将多个精灵图片紧紧包装在一起，形成一个Atlas(图集)，可有效减少无用的空白区域。
        2. 减少Draw Call：在形成图集之后，对图集中的任意一张精灵图片，都只需要统一从该图集中获取。若打包前的精灵图片共有n张，打包后生成一张Atlas图集，则打包前后可降低n-1次Draw Call。

# Git版本控制