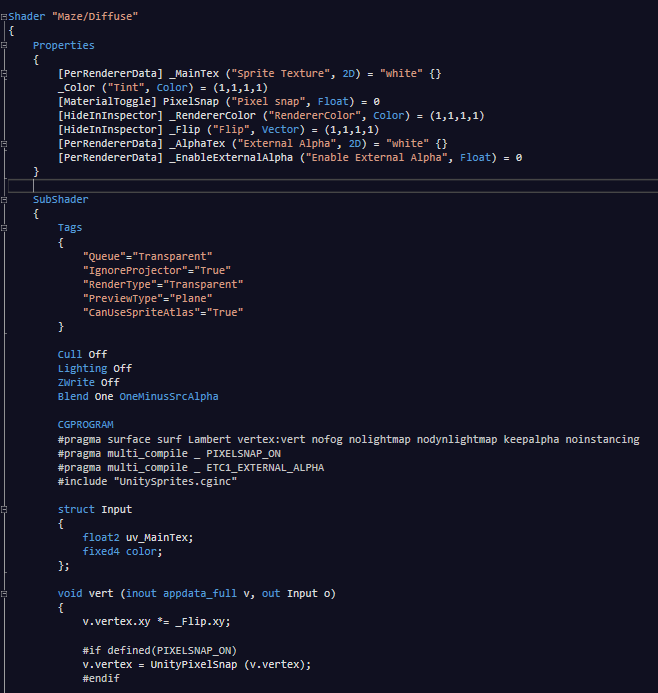
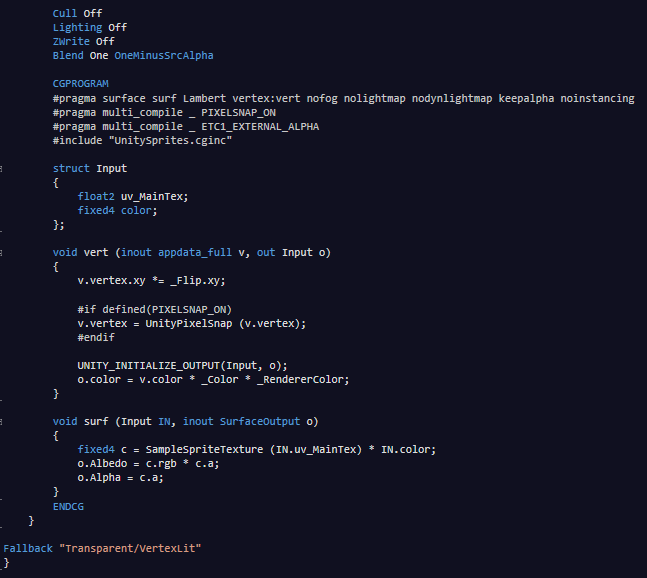
1. 光影效果的制作：
   1. 问题描述：迷宫游戏中，需要表现出在山洞中具有实时探照灯的光影效果。
   2. 解决方案：
      1. 光源类型的选择：根据需求与参考图示，选取合适的光源作为光照类型：
         1. Point Lights(点光源)：点光源位于空间的一个点上，并且均匀地向各个方向发出光线。 光线撞击表面的方向是从接触点回到光线物体中心的线。 强度随着距光线的距离而减小，在指定范围内达到零。 光强度与距光源距离的平方成反比。 这被称为“反平方律”，与光在现实世界中的表现类似。点光源可用于模拟场景中的灯和其他本地光源。常用于作为可照亮周围环境的荧光或爆炸光效。
         2. Spot Lights(聚光灯)：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”。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/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。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)：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。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。
         5. 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。
      2. 着色器：编写Surface-Shader(表面着色器)使游戏物体能够与光进行交互，从而产生光影效果。具体效果应从漫反射的属性作为切入点，参考Unity内置shader：Sprite/Diffuse进行编写。
      3. 漫反射着色器的编写：





* + 1. 注意事项：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。若发现物体漆黑一片，有可能是由于光源沿z轴的坐标小于物体的z坐标，使得物体处于光源光照范围之外。
  1. 结果：实现预期光影效果，如下图：

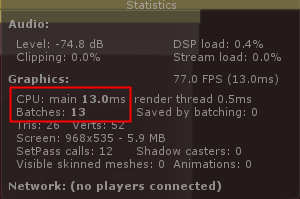






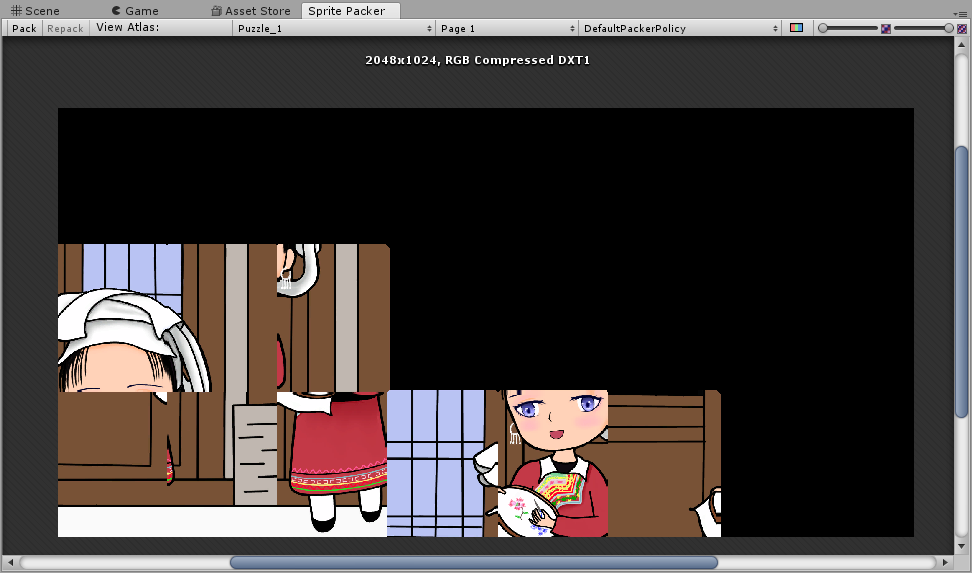
1. 减少Draw Call以优化性能：
   1. 问题描述：美术素材均为单张精灵图片，对每张精灵图片都分别作单独处理将产生不必要的空间浪费与性能损耗；
   2. 解决方案：
      * 1. 减少空白区域：在进行2D贴图时，比较直接和简单的方法是分别对每个需要贴图的游戏物体都赋予一张单独的精灵图片。但对于单张精灵图片，其内部常存在一些无用的空隙，这些空隙将造成运行时空间浪费。为了获得更佳性能，最好能够将多个精灵图片紧紧包装在一起，形成一个Atlas(图集)，可有效减少无用的空白区域。
        2. 减少Draw Call：在形成图集之后，对图集中的任意一张精灵图片，都只需要统一从该图集中获取。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。
        3. 综上所述，可采用Sprite Packer 插件对需要处理的多个精灵图片打包封装为一个Atlas(图集)。
   3. 结果：
      1. 打包前：



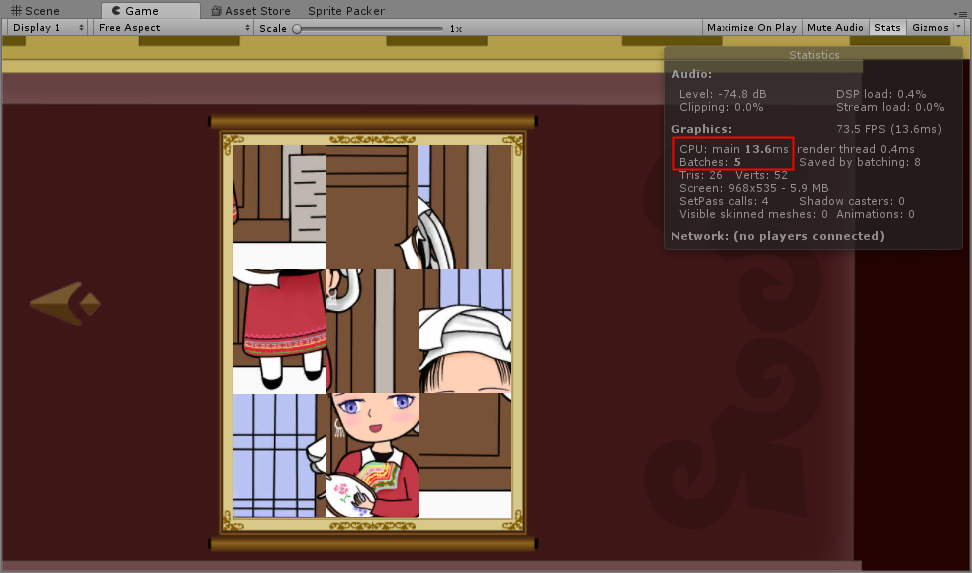


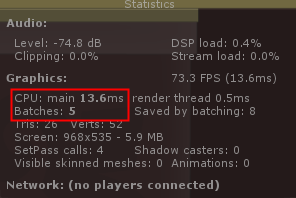
**Batches(Draw Call): 13**

* + 1. 使用 Sprite Packer 对指定图片进行打包：



* + 1. 打包后：





**Batches(Draw Call): 5**

* + 1. 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。）

1. UI屏幕适配
   1. 问题描述：UI布局在不同分辨率的终端上显示不一致
   2. 解决方案：
      1. 方案1：
         1. 调整Canvas的Render Mode属性为Screen Space - Camera， 将映射游戏内容的Camera拖入Render Camera中，下一个属性Plane Distance表示UI与Camera的在Z轴距离（其实就是变相反映了UI的Z轴位置）；
         2. 接着在Canvas Scaler属性里将Ui Scale Mode属性设置为Scale With Screen Size，表示Canvas会根据屏幕比例缩放；
         3. 下面的Screen Match Mode属性选中Match Width Or Height，表示采用宽度或高度（Unity自带适配方式）适配。然后Match调整为0或1，0表示完全宽度适配，1表示完全高度适配，其他值表示介于两者之间采用比例适配。
      2. 方案2：
         1. 调整Canvas的Render Mode属性为World Space；
         2. 将Event Camera设置为映射游戏内容的Camera；
         3. 调整Rect Transform组件中的Width和Height为设计尺寸的宽和高，同时将Scale属性的X和Y都调整为0.01（对应unity2d默认情况下像素Pixels与引擎单位Unit对应比例100）。
   3. 结果：UI元素可根据当前设备分辨率进行自适应布局。