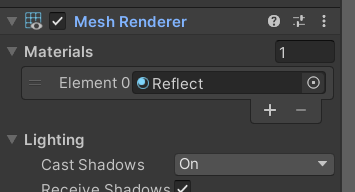
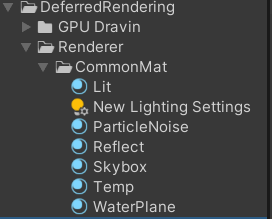
**文件结构说明**

DefferedRendering：

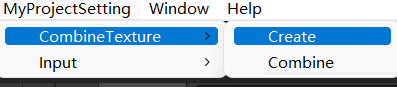
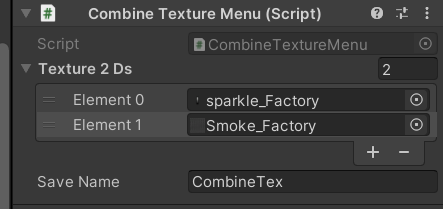
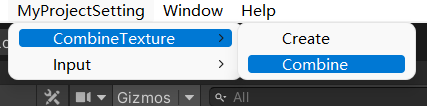
* 这个文件夹存放的是渲染管线，里面的内容不要修改，有兴趣可以看看。
* 因为这个渲染管线不再支持默认创建物体时赋予的材质，因此创建物体后找到该文件夹，选择**Lit或Reflect**替换默认材质即可。



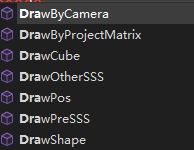
Scenes：

* 这个文件夹是用来存放所有的场景文件的，之后的所有的场景都需要放在该文件夹下。

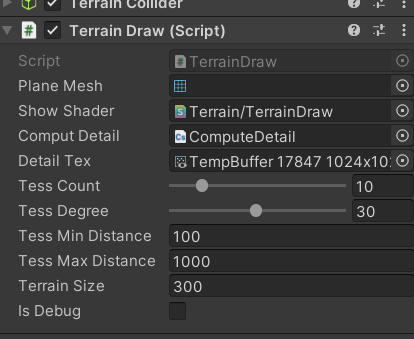
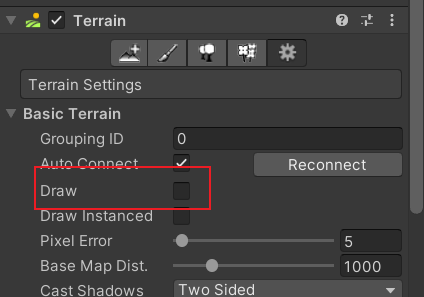
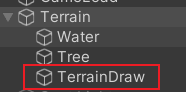
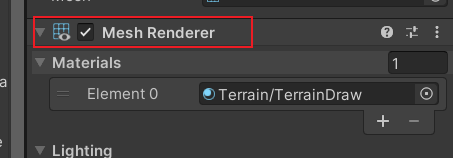
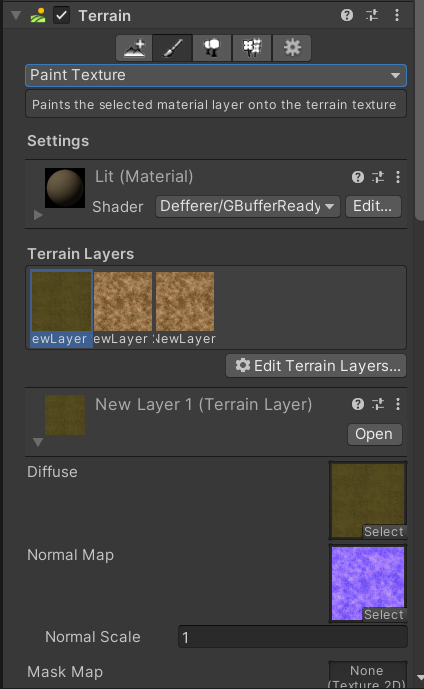
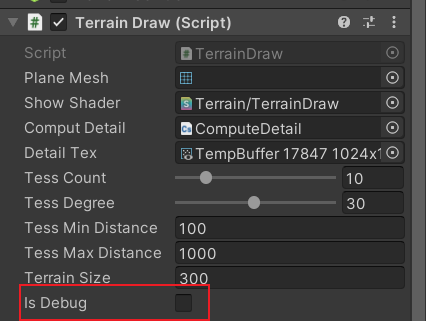
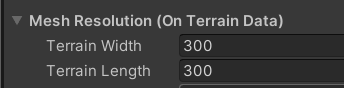
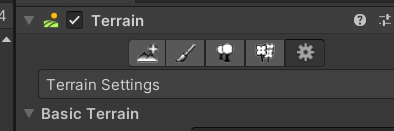
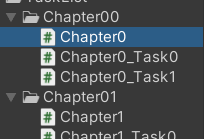
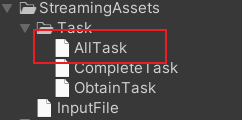
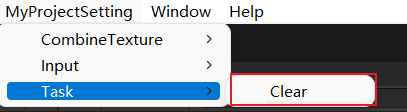
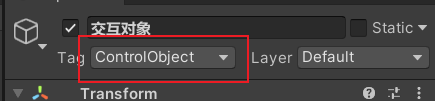
Scripts：

* 这个文件夹存储了全部的脚本文件，也是目前的框架，之后根据需求情况进行拓展。
* Common：
  + 这个文件夹存储了一些封装好的方法，用来对一些常用方法进行封装。
  + AsyncLoad是一个多线程类，通过委托将需要进行多线程处理的方法插入，依次处理。一般委托不会这么用，但是为了方便管理使用了链表的方式存储。
  + SceneObjectMap是一个场景加载类，通过标签将需要进行控制的物体用名称控制起来，之后有更好的对象控制方法可以直接换掉。
  + SustainCoroutine是协程类，用法和多线程一样，但是这个比较常用，因为对于GameObject只能在主线程中修改，协程是运行在主线程中的。这个类会在每一帧运行一次所有入栈的方法，如果这个方法放回true，就会移除这个类。这个类使用了池化技术，会比多线程快一点。
  + HandlerList：定义了一些委托类型，方便统一数据类型。
  + CommonFunction：定义了一些可能用到的方法，这个类是感觉有了SceneObjectMap就没必要了，需要查找时直接使用map映射更快。
  + FileReadAndWrite：文件读取写入类，读取字符串和写只服从到文件中
* Control： 控制文件夹，之后需要对主角、场景等进行控制的脚本都放在这个文件夹中
  + PlayControl：主角控制，目前只添加了简单的输入输出，控制主角移动等。
  + SceneChangeControl：场景切换类，先放在这里，以后用得上，但是这个类目前是直接切换，理论上是要进入到中间场景再切换是最合理的。
  + SceneObjectControl：场景物体控制类，用于存放一些节点的控制器，比如敌人、子弹、终点…
  + GameLoad：场景加载，之后的一些单例、工厂、设置等类需要优先加载时调用的类，之后可能需要定义一些加载模式，进行不同场景不同的初始化。
* Info： //存储角色数据的类
  + CharacterInfo：主角所有的根据数据都会在这个类中，以后敌人等信息也会存储在这里，以后估计会在其继承的子类上进行文件加载，毕竟要保存进度，而且不同的敌人需要不同的数据。
* Move： //运动脚本存储位置，这部分没有完善，目前就是能跑而已，可以的话最好搞成多态，让控制器只需要知道一个移动方法就能控制不同类型的物体移动，不过先这样先。
  + OribitCamera：第三人称的摄像机跟随主角脚本
  + FirstPersonCameraControl：第一人称的主角摄像机脚本
  + RigiBodyMotor：主角移动脚本
* ProjectSet：这是拓展的项目设置，以后可能进行编译器拓展时的脚本存放位置
  + MyInput：一个封装的Input类，本质上就是Unity提供的输入输出，之前想要在游戏运行时修改输入设置，没找到就写了这个类，其用法和Input类是一样的，不过将更新时间换为了FixUpdate，之后有需要拓展输入方式时可以更新该类。
* StreamingAssets：这个是一个存放文件的文件夹
  + 这个文件夹不会在Unity发表项目时打包、加密，所以可以保证在游戏发布后都能正常读取，因此以后的数值，剧情等数据都可以放在这里。
* ParticleNoiseFactory类
  + 这个类是粒子工厂类，根据传入的参数以及设置的图片可以渲染在指定位置渲染出粒子。
  + 现在提供了三种方式，一种是直接在初始位置释放，一种是在初始位置周围的一个矩形中释放，矩形的大小是cubeOffset，也就是一个Vector3代表xyz的偏移大小。球形的需要参数是radius和radian，也就是根据参数方程在一个球面上生成粒子。
  + 这个类可以指定普通的图片，也就是这个图片不需要变化，这种时候需要在我的Resources->ParticleFactory下的文件出指定一下其的行与列数，指定为1和1。之所以要这样是因为我需要确定这张图片是否本身就是一个图集，默认设置的第2张图片，这个图片本身就是一个图集，因此需要指定行数以及列数为8和8，表示总共有64个图片，渲染的粒子要一张张切换。
  + 图集图片的生成方式很简单，首先要保证图片都是一致的大小，不一致可以扔到PS中导出一张一致的图片，目前设的是256\*256，粒子没必要很精细。在点击MyProjectSetting->CombineTexture->Create：  
    会出现一个物体(或者给该物体加上组件，物体名称：CombineTextureMenu)，之后可以在其的设置面板上添加图片数组，注意图片的编号，之后传入参数的图片编号要和这个数组的编号一致，然后指定下名称就行了：
  + 继续之前的流程，点击下面的Combine，就会打包出一个图集：
* 粒子生成

在函数内引用ParticleNoiseFactory.Instance实例，内置多种渲染方法



其中参数的内容看ParticleDrawData结构体

* 地形系统
  + 首先，在地形创建后加入脚本TerrainDraw，设置为以下参数：
  + 
  + 其中的DetailTex不用管，这个是我用来Debug用的，实时生成。
  + 在渲染时需要将Draw给关掉，不用Unity来绘制高度。
  + 不过由于我渲染时的数据都是实时生成的，因此在绘制高度时可以关闭我的渲染，在Terrain的子物体处：处的，关闭后就不会渲染了，然后开启Terrain的Draw，在这个模式下绘制高度会方便一点。
  + 绘制纹理时就需要关闭Terrain的Draw模式，改用我的渲染模型。
  + 在Terrain的Paint Texture设置完纹理后就可以使用了，但是需要额外设置以下值，我将这个值定位了我纹理的主颜色。
  + 在TerrainDraw上开启is Debug后，会实时生成笔刷上绘制的数据，但是这个损耗很高，绘制时用一下，不要在游戏发布时开启。
  + 具体贴图绘制就没什么区别了，也是设置金属度和光滑程度即可。
  + Terrain Size是地形大小，和Unity的  
      
    相对应，不过这里我强制要求为正方形而已。
* 任务系统：
  + 任务系统实现的不是很好，而且估计在不同场景中切换会导致bug，不过先这样先，有bug了之后再改，至少单场景没有什么大问题。  
    同时用到了不少语言特性，理解困难。
  + 首先说明任务系统的结构，目前的设计是双层结构，也就是一层是章节，一个章节管理则一系列子章节，因此我们创建每一个章节任务时也需要按照这个结构。
  + 对于每一个章节，我们需要继承自AsynChapterBase，同时赋值一系列数据。  
    其中**chapterName**是章节名称，之后需要显示任务UI时会用到，  
    chapterTitle是章节标题，之前打算是用来做章节介绍的。  
    taskPartCount表示子章节的数量，用来判断该章节是否完成，而且方便子章节切换。  
    chapterID是这个章节的编号，需要用编号来唯一指定一个章节，方便任务管理以及查找，要判断其他章节是否完成传入一个章节编号即可。  
    chapterSavePath是章节文本保存位置，因为我没有在这里搭建对话UI，这个可以先不管先。  
    runtimeScene是运行的场景名称，如果在游戏开始时会判断该任务是否运行在该场景，是才加载该任务。  
    targetPart是默认是任务系统命名空间的前缀，在初始化时需要加上子章节的名称，得到一个包含前缀以及章节名称的字符串，之后根据目前运行的章节编号会进行子章节的对象获取、运行，因此可以发现我们子章节名称都是按照编号来设置的。  
    以上看不懂的话可以先了解一下反射的内容，这个是Java和C#这类面向对象语音都有的知识，而且很重要。
  + 创建完章节后需要进行注册，也就是拷贝章节名称到文件夹下，用文本打开就行。剩下两个文件一个用来存储完成的任务，一个用来存储运行中的任务，这些文本文件由任务系统管理以及加载。  
    由于清除数据过于麻烦，于是设了一个快捷键清空所有的游戏任务数据，不会清空注册任务用的文件，清空其他两个文件的内容而已，也就是初始化。
  + 继承文件后需要实现三个方法，首先是**CheckAndLoadChapter**，当该任务注册了，但是没有完成，也没有获取到时、或者被某个任务调用任务管理类的**ReadyChapter**方法时会调用该方法。  
    在我的构思中，只有该任务是一个支线任务时，才会某个章节结束后会调用一下，用来判断一下是否要准备一下该任务，需要的话加一些触发内容，触发后这个任务就可以加入到任务管理器中了。
  + **CompleteChapter**方法是用来当这个任务完成时，在重新加载场景时会执行的方法，这个方法打算用来**当某些场景在任务结束了就看不到了**，**看不到就直接运行该方法关闭，避免重开游戏后就没有关闭**的情况，但是实际上用这种方式关闭很傻，有更好的方式就换了吧，因此这个方法不实现也行。
  + **ExitChapter**方法在章节结束之后会调用，也就是所有子章节都完成了，就会调用该方法，用来清除一些数据之类的。
  + 子章节：
    - 子章节才是真正任务的执行方法，用来执行游戏中的行为，因为任务系统在加载时是运行在多线程中的，Unity不允许多线程访问继承MonoBehaviour的数据，因此经常需要配合我的协程类一同使用。
    - **EnterTaskEvent**方法是当该子章节开始执行时运行的方法，我设置了isLoaded参数，用来判断该子章节是否是第一次加载，如果是第一次可以加载一下对话，印象中很多游戏都是这样的，于是也这么设计了，这个参数也是不管也行。
    - **IsCompleteTask**方法是发生交互事件，也就是某个类对任务管理器发送了任务信息，并且这个任务的编号与本章节的章节编号一致时执行的方法，用来判断该章节是否完成，如果返回true，就是本章节完成，之后会自动调用**ExitTaskEvent**然后切换到下一个子章节或者结束该章节。
    - **ExitTaskEvent**就是子章节结束时执行的方法了。
  + 案例描述：任务系统设计了不少其他类，不描述一下内容我自己都会忘记，太多了
    - 对于Chapter0的内容不多，前面都写过了，就是普通的加载章节以及切换章节而已。
    - **Chapter0\_Task0**类才是重点，在进入任务时我们存储了所属章节，然后将将一个方法插入到**协程栈**中。
    - **协程栈**是对Unity自带的协程的一个整合，实际上很多开发工具都带有协程，本质上是一个附加在主线程中的程序，具体可以百度一下。  
      由于协程需要物体挂载MonoBehaviour才可以使用，但是任务没有必要继承该类，因为任务类只是管理场景中的组件用的，其本身不生成物体，因此为了方便协程的使用我建立了这个协程栈**。**这个协程栈在外部只需要使用方法**AddCoroutine**，插入一个**委托**，也就是一个方法(函数指针)，这是C#定义的一种数据类型，在C++中叫做函数指针，不过这个是一个封装过的函数指针，**类似一个函数指针类型的数组**，具体百度吧。  
      这个方法要求是无参且bool作为返回值，用来判断这个方法是否执行完成，如果**执行完成就返回true**，协程栈会自动移除该方法。如果**返回false**，则**下一帧会继续执行该方法**，因此建议一次循环内执行的内容少一点，有很多内容就分多次执行。
    - 使用协程栈是因为协程栈是运行在主线程上的，所有的数据访问都是允许的，为了方便，建议在进入方法时执行使用。  
      而且任务系统中不支持Update，需要逐帧执行就要用到**协程栈**。
    - 在插入的方法**FindObject**中，我们首先查找了游戏中的一个物体，用游戏对象映射表来查找，比使用GameObject的find方法查找快，不过需要首先在场景中创建该物体，然后设置其tag为ControlObject，这样就能够找到该物体了。  
      本质上就是该类会在游戏开始时将所有tags为ControlObject的物体都加入到映射表中，之后查找时直接映射就出对象了。
    - 接着我在该物体上挂在了一个交互对象，在**主角靠近到该物体时按E**就会执行**InteractionBase**中的**InteractionBehavior**方法，也就是执行了该交互。  
      在这里我用了一个骚操作，也就是**定义了一个交互类**，这个**类的交互方法只有一个委托**，然后**在这个委托中定义我要执行的方法**，这样就不用创建脚本然后再挂在对象上了，方便游戏中经常需要的交互内容的切换，不需要傻傻的手动拖拽。
    - 需要注意的是这个交互类为了避免多重交互，使用了交互锁定，因此需要再交互结束后调用**StopInteraction**方法，不然这个交互就会卡在这里了。
    - 另一个子任务的内容和这个差不多，就不复述了。
  + 目前任务系统并没有完成，等我实现好AI后会完成其中的UI部分，加上对话的UI等，而且估计有bug，只是暂时没发现而已。
  + 对于保存数据ObtainTask文件中的格式如：<m n>，m是章节编号，n是目前做到的小章节。CompleteTask保存的格式是一行一个章节id，表示该章节已经完成。了解一下存储格式方便判断错误。

敌人的状态机

脚本放在了Asset/Scripts/Control/Enemy/States

IStates：状态的接口，包含OnEnter，OnUpdate，OnExit三个函数

States：存放所有状态类的文件，包括Idle、Chase、Attack、Die，所有状态都实现IState接口

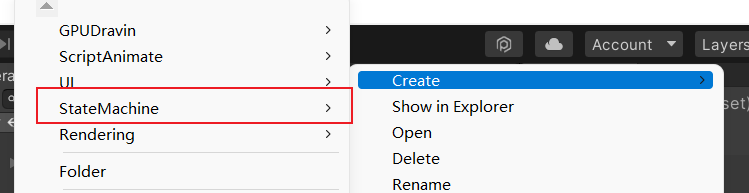
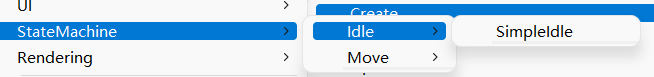
FSM：状态机，挂载到需要的敌人上，由于使用了Unity自带的寻路系统，需要再挂载Nav Mesh Agent，同时需要把场景静态物件设置为静态的，并在Windows-AI-Navigator里将场景烘焙。目前脚本内的target锁定为tag为Player的物体

添加状态的方法：

1. 在States脚本里添加新的状态类，实现IState接口的内容。
2. 在FSM里添加EnemyStateType的敌人状态类型枚举
3. 在Start函数里注册新的状态states.Add(EnemyStateType.xxx, new xxxStates(this))
4. 在别的状态类里添加进入新状态的方法TransitionState(EnemyStateType.xxx)

(本來一开始想直接利用动画器里的动作树作状态机，后来发现要不断传参太麻烦了)

状态机系统：文件夹在：Scripts/StateMachine下

* 首先控制类：StateMachineManage
  + 我打算用该控制类代替角色控制类，也就是状态机本身就是控制类，因此在状态机中有属性技能管理，运动引擎，以及动画管理。  
    目前技能系统是完善的不需要后续改进，运动引擎可能后面需要优化，变为抽象类，具体运动实现由实际上的实现类来控制。  
    **而动画管理类是一个随便写的类，可以直接修改，换为自己习惯的实现模式**。
  + 控制类在一开始会进入默认状态，然后获取上面的全部控制类。
  + 在每一个FixedUpdate中进行状态调用，首先会调用该状态的**OnFixedUpdate**方法，进行逐帧调用，然后会调用**CheckState**方法，用来判断是否要进入下一个状态，判断的方式是是否有返回下一个状态，如果有就执行退出当前状态的方法，然后执行进入下一个状态的方法，进行状态切换。
* 状态机基类：StateMachineBase
  + 该类是所有状态机的基类，所有状态机实现都要继承该类。
  + 该类本身定义了4个方法，看看注释就知道了，同时该类还继承了ScriptableObject类，让该类至此创建预制件。
* 实现的几个类，在List中，分为了三种，也就是**待机**，**走向主角**(用来执行敌人听到了主角声音，但没有看到主角，于是走来侦察，或者实现一个智能的寻路，用来查找主角)，  
  **攻击**。
  + 目前攻击状态没写完，不用管，但是可以看一下其他两个类。  
    这两个类都在类的开头写上了一行   
     **[CreateAssetMenu(menuName = "StateMachine/。。。/。。。")]**表示这个类可以创建一个代码预制件文件。当一个类其继承了ScriptableObject，并且有写上上面这行代码，就可以在Create中显示出这个条目  
    
  + 使用时只用在敌人那里挂上StateMachineManage，然后将创建出来的默认状态放到开始状态的位置就行了，不用挂道NowState上，这里是用来debug用的。
  + 注意点：ScriptableObject物体实际上就是一个类实例，不过这个类实例存储在了外存中，因此使用时不要直接获取Component，这样创建多个敌人时就只能存储一个敌人的数据了，如果有需要就将全部数据都在状态机控制类上获取，在类中只进行简单的运算。  
    不过如果这样不方便就多创建几个类实例也行。

# 需求汇总：

1. 机制-终点会根据玩家得分提供不同表现（是否达成通关条件）
2. 美术-到达终点时播放粒子特效
3. 机制-角色的下蹲、闪现、突进
4. 美术-更多的场景…
5. UI-显示角色的生命值与得分
6. UI-设置界面，调整玩家的fov，视角高度，鼠标灵敏度