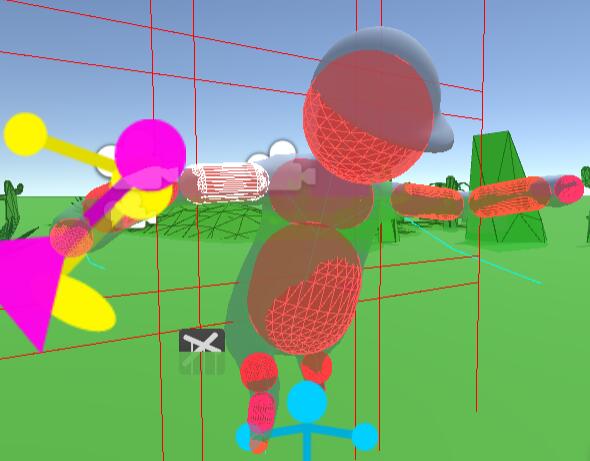
# 需要解决的问题

## 活性布娃娃系统（Active Ragdoll）

布娃娃系统即在角色的各个关节物体上添加刚体，并通过物理力来连接各个刚体关节来实现具有物理交互的人体结构

1. 不同于其它大多数的游戏，只将布娃娃系统应用于角色的尸体上，我们将布娃娃系统直接运用于角色的整个生命周期，以来为玩家提供真实的物理交互
2. 人物的设置



* 1. 从人物身上可以看到具体的 刚体配置，其中红色的代表的就是单独的刚体，刚体之间通过使用 力而非直接调整绝对位置来控制连接。
  2. 此外实际上 刚体对应的游戏物体并不带有 网格信息(即模型外观)，二者是互相独立的，我们通过使用刚体来模拟物理，而后将mesh 的位置设置到刚体的位置来实现可视的物理效果
     1. 这样做的原因是 可以让动画带来的位置/旋转信息和物理计算的位置/旋转信息共存，从而让任意动画都可以拥有物理效果

## 万向挥砍

万向挥砍功能即玩家可以通过使用鼠标的挥动来自由地挥动武器，这点不同于传统的直接使用动画的游戏。这意味着玩家拥有更自由，更细节化的战斗方式并且搭配上活性布娃娃系统，让游戏的战斗系统变地更加真实有趣



1. 问题：如上图，武器的朝向如果不做修改那么将会一直保持同一个方向，不会
2. 解决方法：  
   我们考察攻击和防御时的手部的朝向发现。**攻击时**：挥动武器一样武器的刃部一般和武器的速度方向相同。**防御时**：武器一般垂直于武器的速度方向来达到最大的防御面积，由此我们总结出如下解决方法
   1. 我们将会对 手部的旋转 根据 挥动的速度和 相对 角色身体中心的位置 以及攻击目标进行 一个加权计算来得到手部的旋转角度。
   2. 或使用 现有动画进行手部遮罩 配合IK 系统，在动画运行过程中调整 动画映射到IK的权重，从而达成对挥砍轨迹的自定义。

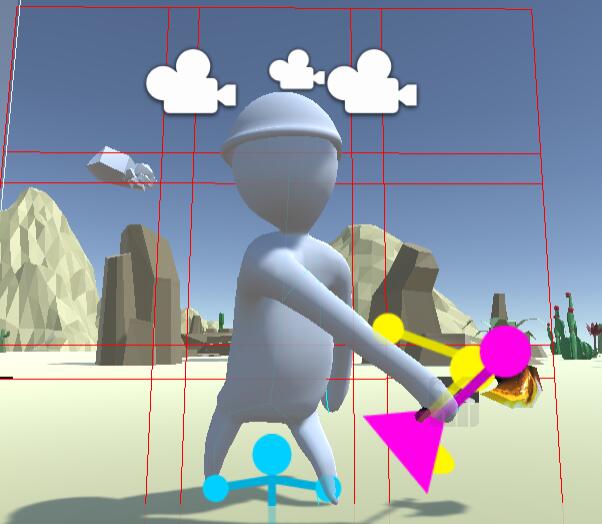
## 武器系统

不同于一般的武器系统，为了实现更加贴近真实并且更具有趣味性的武器系统。我们实现了武器的场景嵌入，即只要速度只够快，并且方向正确，武器的刃部就会嵌入到场景中。此时玩家就必须通过鼠标的位移来让角色用力以将武器拔出场景



## 剑技系统

1. 剑技的触发 ：
   1. 需求
      1. 要求玩家将 武器位置 放置到 对应的 剑技的起始位置 然后按下触发键表示 希望进行剑技的触发
   2. 实现
      1. 使用 触发盒， 为 玩家 布置 9 个 触发体 表示 9个不同的位置
         1. 好处： 易于配置 , 可以使用 不同的 几何形状，较为通用
         2. 坏处 处于接缝处【也可以设置 接缝处不进行剑技的触发】
      2. 我们使用了一个性能消耗较低的方式，即将身体重心到手部的向量映射到一个固有的平面上，并将平面划分为9宫格(如下图)。将三维空间缩放到二维，通过二维空间下这个向量的终点位置处于9宫格的对应位置来使用对应的技能

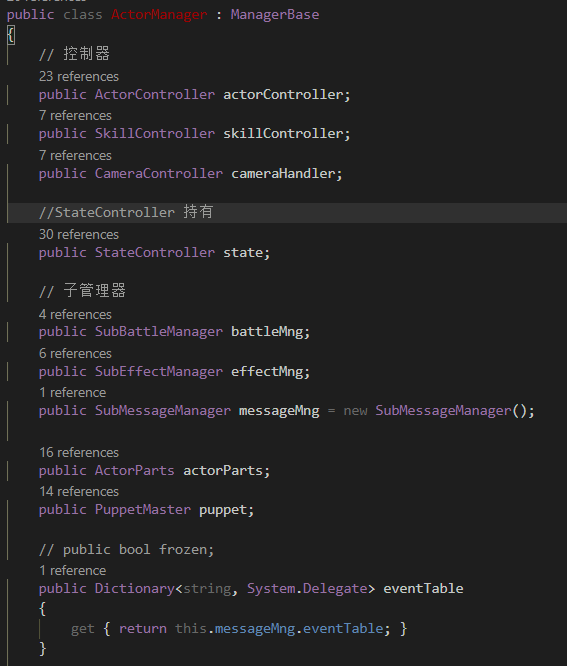


1. 剑技的效果
   1. 需求
      1. 剑技触发之后 玩家将被动画带动 使用技能
      2. 并且 要求 在 使用技能的途中 依旧可以进行 一定的ik 操作
         1. 要进行IK 操作 则要求按紧鼠标对应的键位 否则将没有 IK 作用【完全按照原动画播放】
         2. *（待判定）IK 操作的时候 若 基本没有 鼠标的移动，则 同样 应该将 IK 复原回原本位置*
         3. IK 方向和 移动方向 的相对偏移将导致 攻击力下降
      3. 注意 要减去 过渡的时间 让 剑技直接在 当前的位置进行释放 而不会进入过渡的动画 。
   2. 实现
      1. 由于 技能 数量可能很多并且各个剑技的过渡属性等可能皆有不同，不适合 直接使用animatorController来设置动画状态机。
         1. 可以 使用 animator.crossFaded 函数来指定剑技的名字来 实现 动画的转态和 过渡
      2. IK 微调，IK微调指的是玩家可以通过鼠标的移动来实现在武器沿着剑技轨迹移动的时候，做到辅助运动，如果玩家滑动的轨迹和剑技原本的轨迹基本相同的话那么将会达到更快速，威力更大的剑技效果
         1. IKWeight 设置 非常小。
         2. IK 位置要和 动画的 原位置 直接挂钩
         3. 具体做法即在IK 更新动画之前获取到动画的位置
   3. 特效
      1. 武器拖尾特效
         1. 武器的拖尾特效当前直接使用的是Unity 自带的 LineRenderer实现
      2. 武器的发光
         1. 武器的发光则是通过使用通用的PBS shader(物理模拟shader) 来修改发光参数实现

# 游戏总体设计

## ActorManager

ActorManager 用于总管角色（无论是玩家还是敌人）所需的所有组件，并作为这些组件和其它的Manager类交互的唯一接口。这些组件包括各个Controller控制器用于管理对应的内容的行为。SubManager 和Controller的区别只在于 前者不是MonoBehaviour的子类，其行为的调用取决于外部的事件(委托)。此外还有一些非Controller和SubManager 的组件成员如 ActorParts 和 puppetMaster 虽然ActorManager本身不适用这几个成员但是这些成员会被各个组件访问到所以放于 ActorManager中作为角色组件的全局变量

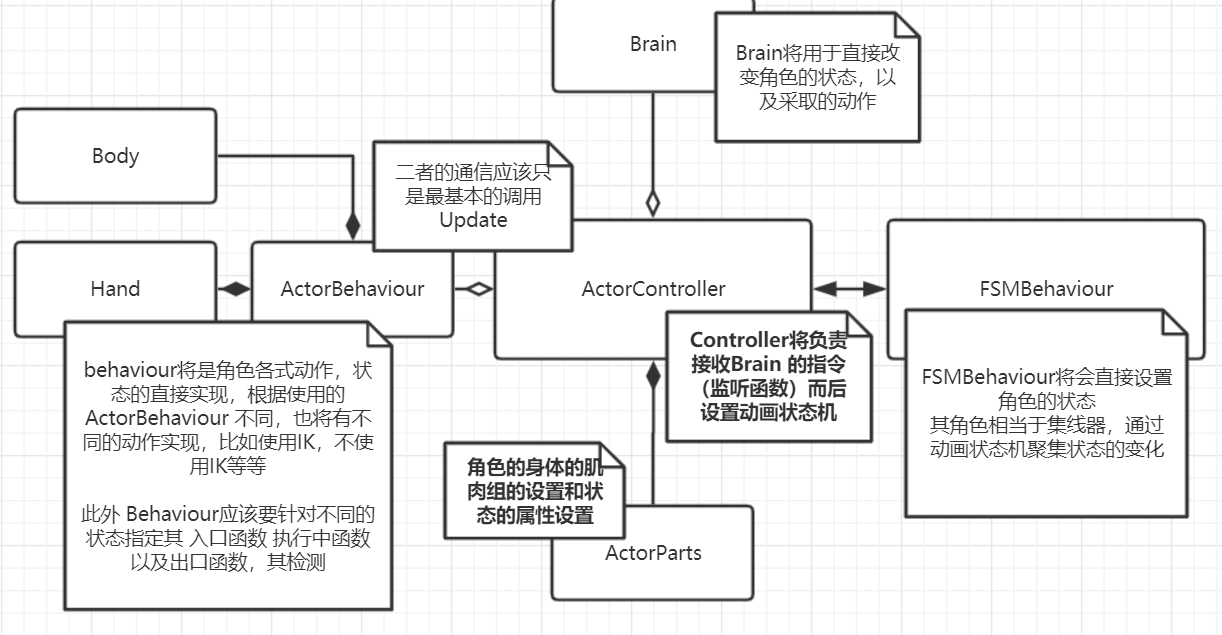


### StateController

StateController用于管理角色的状态，这将包括角色手部/身体的状态以及角色具体的数值。

1. 状态设置如下  
   鉴于角色的动作上下半身可以分离，所以我们将状态分辨为HandAction和BodyAction,这样手部的状态和身体的状态将分离，并各自独立更新  
   
2. 状态的设置位置 是在 状态动画机 进入特定的状态的时候被设置将在 ActorController 中具体讲到

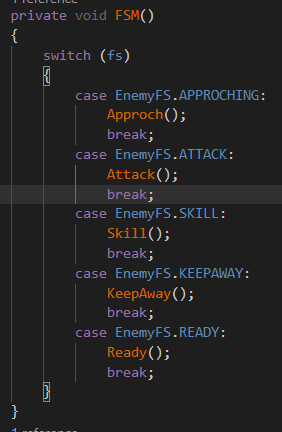
### ActorController

ActorController 及其附属的组件和说明如下图  


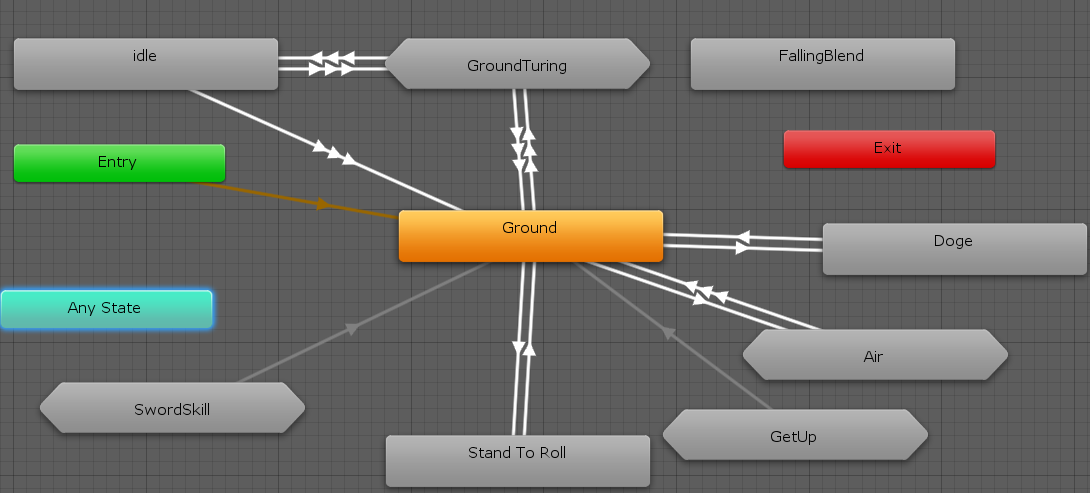
1. ActorController强调低耦合和高复用，从下图中可以看出，像是Brain,ActorBehaviour 和 ActorController 都是聚合关系而非组合关系，这意味着即便ActorContorller没有这两个的具体实现依旧可以正常运行(比如现阶段的简单AI)这体现了低耦合.此外ActorController中的 Behaviour 、Parts 、Brain 都是基类，依旧可以适用继承来具体实现其内容。
2. ActorController 是角色控制器并且是一个抽象类，只是定义必须的内容。此类将掌管角色的运作逻辑，是外部输入的入口(外部输入包括Brain类的AI状态机输入和InputManager 的键盘输入)，各种动画(动作)执行前的操作在这里被定义（如下图，着包括 Dodge，Jump，Air等等动作）：

#### Brain

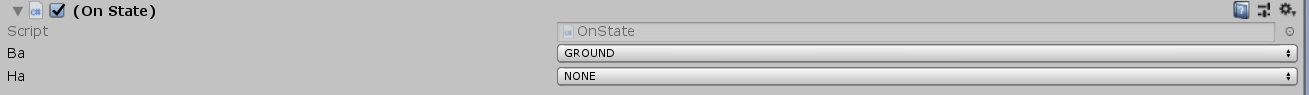
Brain 顾名思义即为角色的大脑，用于根据当前的场景情况来决定要使用 actorController 中的哪一个函数。对于玩家而言，brain即为 InputManager 即玩家的输入，而对于AI而言这个类必须提供具体的定义以来调用 ActorController 的函数。

1. 目前典型的实现是 使用 有限状态机  
   根据不同的 状态来调用不同的更新函数
   1. 一般而言 Brain 要含有NavMeshAgent（unity 的寻路组件），以此组件为核心判断和玩家之间的距离以来影响状态机的决策
   2. 另外 玩家的实体即为ActorManger，所以Brain 只需要访问AM 的一些对外信息（比如StateController 带有的状态信息等）就可以知道敌对方的状态，以此来影响决策

#### FSMBehaviours

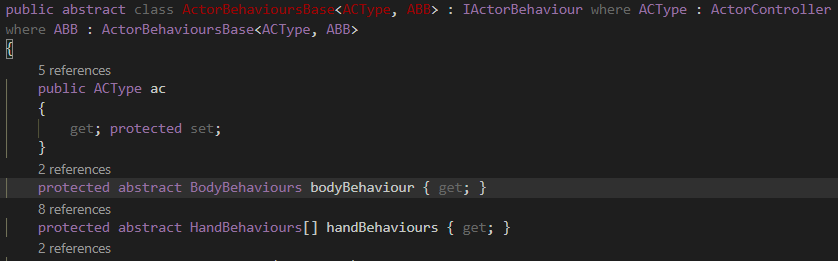
FSMBehaviour是配合 玩家的动画状态机，来在进入对应的动画的时候设置玩家的状态（动画状态机如下）：  


举例对于Ground状态(其脚本参数的设置如下，ba即为希望的BodyAction，ha即为希望的handAction，可以看到对于GROUND 状态 则希望 ba=Ground ha = None)：



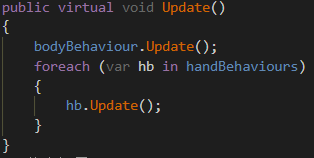
将状态的设置和 动画状态机绑定的好处就在于可以统一动画状态机和玩家的状态，让玩家的状态变换结构清晰，比起在代码中进行状态的变化更易于维护。

#### ActorBehaviours

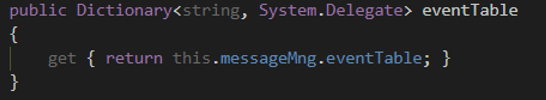
ActorBehaviours 用于定义角色的不同部件在不同的状态下持续调用的函数，由于不同类型的角色将会有不同的更新行为，并且行为将会和对应的ActorController 的实现对应所以ActorBehaviours类需要使用泛型来应对不同的实现（如下为ActorBehaviours的抽象基类）：  
  
ActorBehaviours的子类 将会组装特定的 BodyBehaviours的子类 和 HandBehaviours的子类来作为身体和手部部位的具体更新实现。

1. 举例 BodyBehaviour 的更新周期：

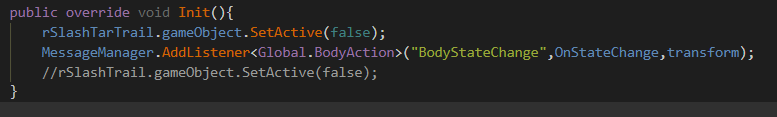
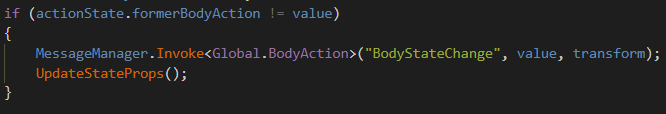


1. ActorBehaviours的Update 本体  
     
   actorBehaviour 的更新函数做的事情很简单即 调用被组装在内的 Body和HandBehaviour 来进行更新。并且此更新函数将被其隶属的ActorController(为MonoBehaviour 可以利用Unity 的生命周期)调用以此达到更新目的
2. 之所以将Behaviour从Controller 中独立出来是为了让使用同一种ActorController的角色在具体实行动作的过程中可以有不同的行为
   1. 比如 使用IK来为微调动作的走向
   2. 使用 不同的 动画层来调整动作
   3. 在动作执行过程中 鉴于其它因素强制转换状态(比如拾取物品的时候物品被打飞，此时角色应该直接退出拾取状态)

### SubManager

先阶段SubManager没有具体实现而是使用全局的 MessageManager，但是其基本实现的内容都是一致的，就是使用C#的委托机制配合Dictionary结构来在全局注册和触发事件降低代码的耦合性（因为通常的委托方式需要持有存储委托的对象才能进行注册，这在一定程度上将会增加代码的耦合性所以需要改进），而ActorManager中所示的evnetTable  


即为委托的字典，通过使用string来标识特定的委托。其具体的使用是注册，调用可能的话还会有删除。此外 MessageManger 的具体实现使用了模板 来让函数可以获得不同类型的参数 具体实例如下：

1. 在SubEffectManager.cs中：  
   
   1. 如上通过使用 AddListener来进行事件的注册 具体的操作就是将指定的委托 函数 OnStateChange 注册到 transform指定的 actorManager 的eventTable下
2. 在StateController.cs中：  
   
   1. 通过使用 Invoke 来调用指定的委托函数，让其它脚本对象获取消息