实验报告

项目名称：USTC SOUL TALE

（2D 动作类游戏）

院系：计算机系

专业：计算机科学与技术

学生姓名：李牧龙、徐航宇

2022 年 6 月 8 日

目录

一、项目介绍：

1. 概述：

本游戏是基于 unity 和 C#语言开发的在 Windows 系统下运行的横版 2D 战斗类游戏。游戏难度不大，操作简便，同时植入了一定的科大元素，在锻炼编程能力的同时有利于增加对科大的认同感。

2. 功能分析：

(1).单人游戏：由于时间限制，只完成了单人游戏的开发工作。

(2).画面设计：游戏美术素材全部来自于网络素材库中的免费素材，部分素材经过开发者的重新加工，大部分动画为开发者自制。

(3).游戏音乐：音乐取自其它游戏与免费素材，基本能配合游戏的进行。

(4).战斗设计：游戏具有较为完整的战斗系统，可以实现攻击、受击、技能释放等功能。

(5).游戏存档：游戏提供存档功能。

(6).任务系统：游戏提供任务系统。

(7).背包系统：游戏提供背包系统。

(8).对话系统：游戏包含一段较为简单的原创剧情与人物对话功能。

3. 游戏性能：

条件有限，未能进行精细的测试。目前已知有：

(1).游戏在搭载 Intel Core i7-11800H @ 2.30GHz CPU 和 16GB 内存的测试环境下画面运行流畅，无明显卡顿。

(2).游戏中各种操作的实现经过多次测试，运行稳定，无明显漏洞。

二、环境依赖：

OS：Windows

三、人员分工：

徐航宇（PB21050996）:负责代码（主要负责动画制作、数据库与存档以及 UI 等）、素材收集和报告撰写

李牧龙（PB21111639）：负责代码（主要负责人物与 NPC 行为等）和报告撰写

陈明哲（PB21111621）：参与素材收集

贾翰锐（PB21111620）：参与报告撰写

刘丰毅（PB21111622）：负责剧情设计

四、核心功能：

1、人物与 NPC 行为：

（1）人物移动：

人物移动主要通过 Unity 提供的输入管理器、Input 类，以及 Player 物体上的 Transform 组件和 Rigidbody2D 组件实现。Input 类中的 GetAxis()和 GetButtonDown()方法用于读入输入管理器中设置的对应按键。通过改变 Transform 中的 position 值（坐标）实现前后移动，使用 Rigidbody2D 提供的 Addforce(Vector2 Force,ForceMode2D mode)方法实现人物的跳跃。

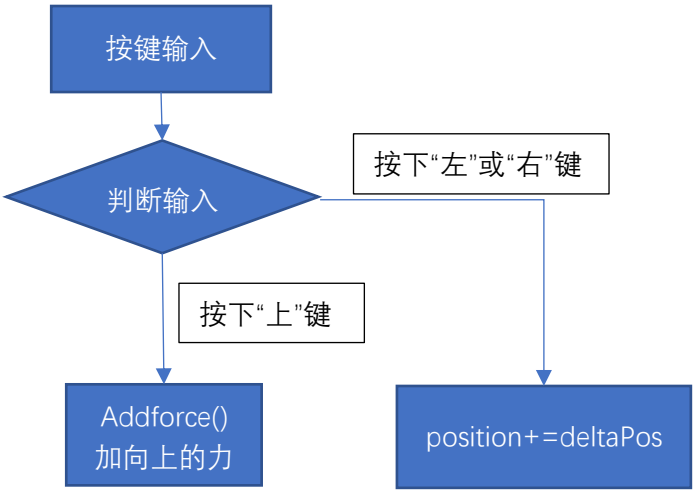


图 1 人物移动程序流程图

（2）人物的多段普攻：

多段普攻通过动画器逻辑实现。每次按下攻击键，SetTrigger(string name)方法都将被调用一次，触发动画器中用于过渡到攻击动画的 Trigger。在当前攻击动画结束前，若 Trigger 被触发，则过渡到下一段攻击动画，否则过渡到人物移动或静止动画，在下一次触发时重新过渡到第一段攻击。动画过渡的逻辑将在下面提及。

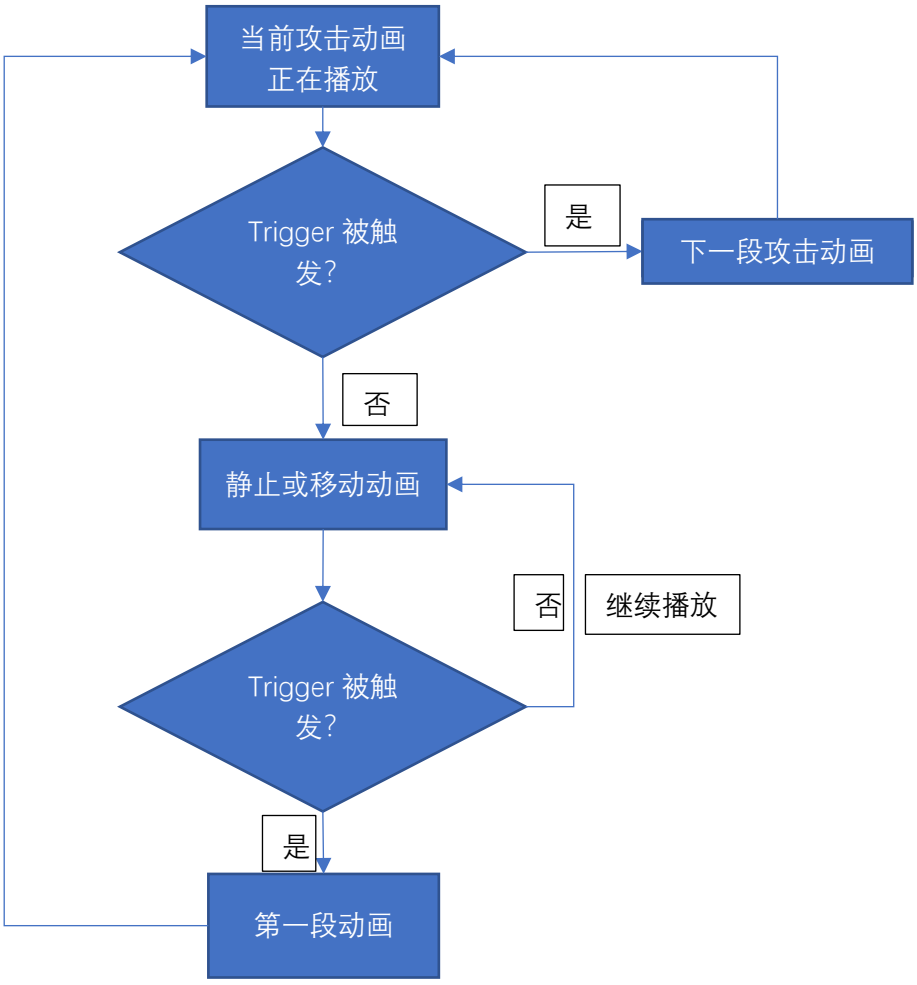


图 2 人物的多段普攻程序流程图

（3）敌人的生成：

敌人的生成通过每个敌人对应的 SpawnPoint 物体实现。SpawnPoint 每帧对玩家坐标进行检测，当玩家进入到生成范围时，即调用 Instantiate(GameObject original,Vector3 position,Quaternion rotation)方法生成对应敌人的预制体。

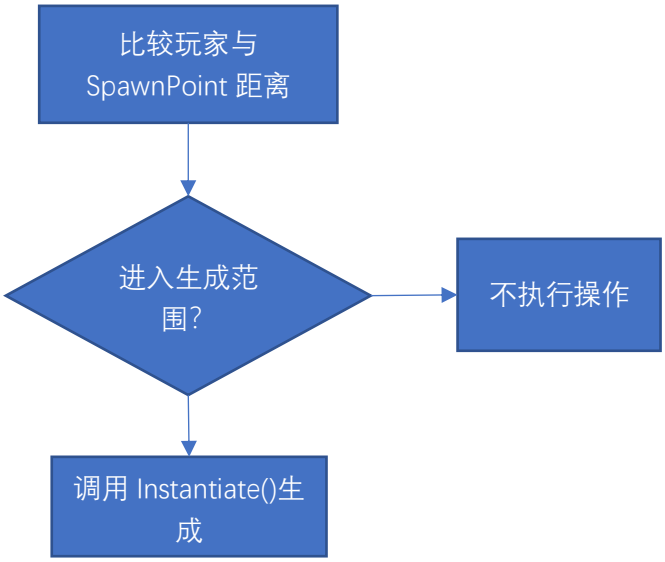


图 3 敌人的生成程序流程图

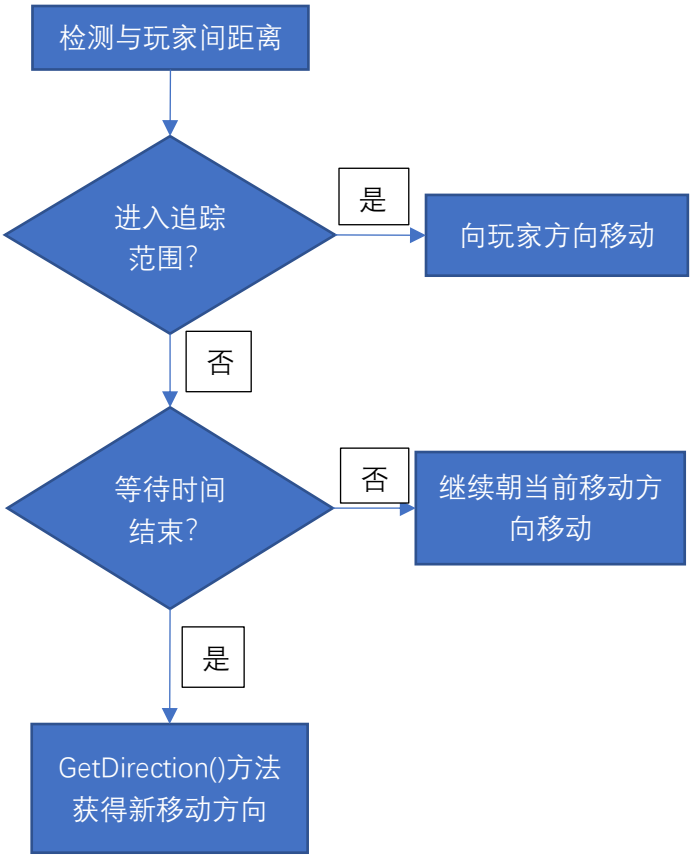
（4）敌人的简单 AI：

不同类型敌人的行动逻辑略有不同，但主要分为漫游、追踪玩家和进行攻击三部分。

(i).漫游：挂载在敌人物体上的漫游脚本每一帧都检测一次自己与玩家间的距离。当玩家不在追踪范围内时，每隔一段固定的等待时间调用一次 GetDirection()方法获取移动方向，在两次 GetDirection()之间通过每帧刷新的 Update()方法改变 Wizard 物体的坐标，向对应方向移动。

(ii).追踪玩家：当玩家进入追踪范围时，不再调用 GetDirection()方法，而是通过改变坐标值直接朝玩家方向移动。

(iii).攻击：敌人的攻击有一定等待时间。等待时间结束后，敌人将根据当前攻击的类型选择在原地停留发动远程攻击或通过追踪移动到玩家位置后进行近程攻击。两种攻击类型具体的实现机制将在技能部分详细叙述。



（该过程每帧进行一次）

图 4　敌人的简单 AI 程序流程图

（5）技能：

玩家和敌人的技能主要分为远程和近程两类。

(i).远程：远程攻击通过生成对应攻击特效的预制体实现。当玩家或敌人进行远程攻击时，将调用 Instantiate(GameObject original,Vector3 position,Quaternion rotation)方法向对应方向生成以一定速度移动的预制体，或直接在攻击对象的坐标上生成预制体。随后预制体按上面提到的攻击判定过程进行攻击，并在完成攻击或过一段时间后销毁。

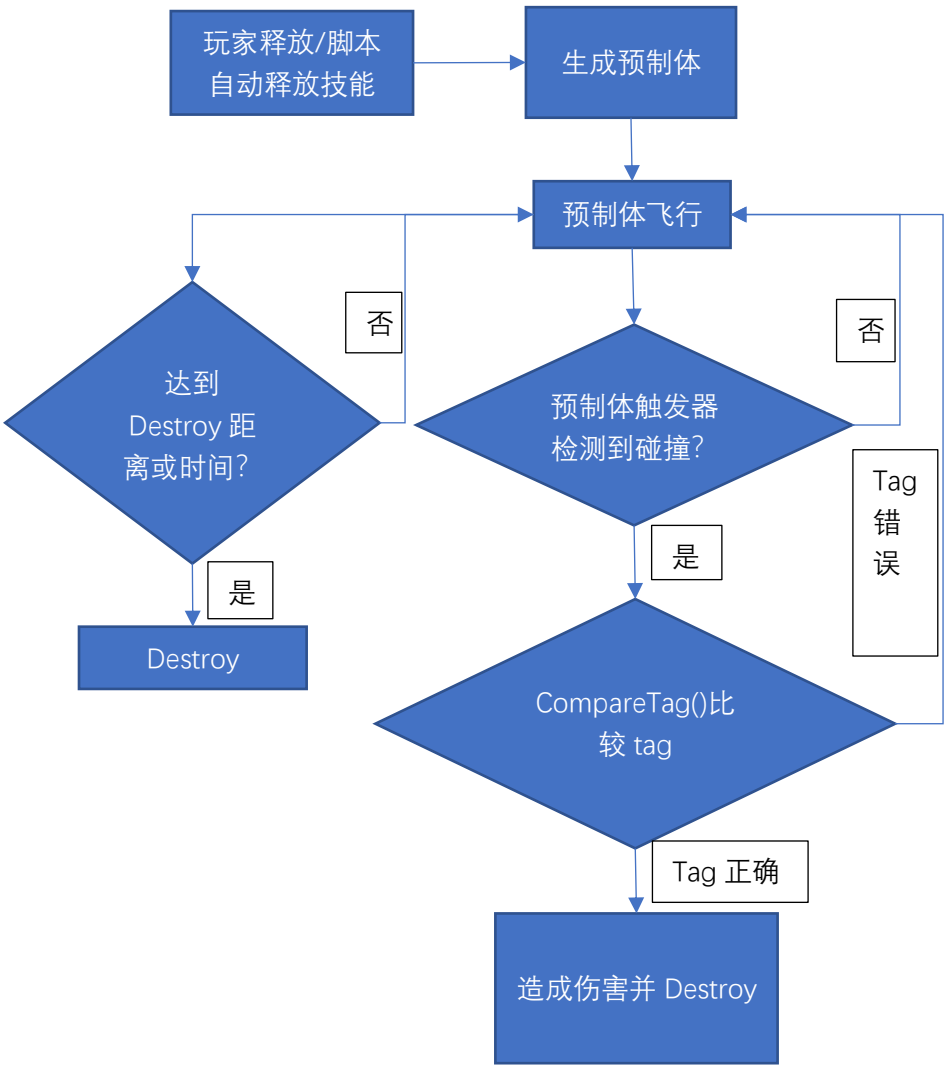


图 5 远程技能程序流程图

（注：上述过程中对距离或时间的检测由每帧刷新的 Update()方法中的语句进行判断，而对碰撞的检测由 Unity 提供的 OnTriggerEnter2D()方法完成）

(ii).近程：近程攻击通过发起攻击的物体上的碰撞器组件实现。进行近程攻击时，作为触发器的碰撞器组件被 enable，并按上面的攻击判定过程进行攻击，在攻击动作完成后 disable。

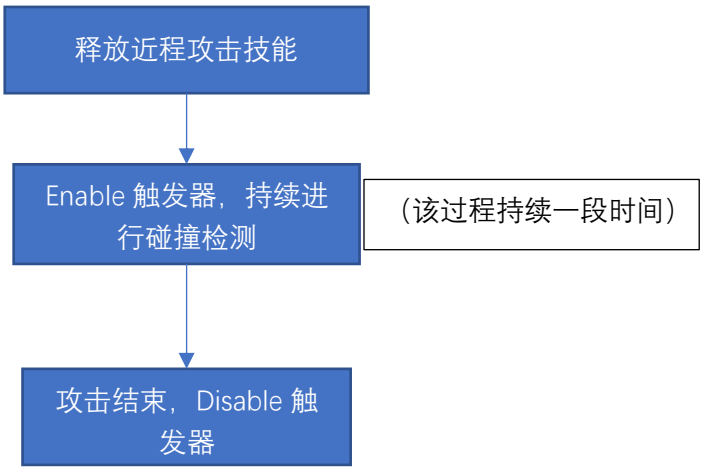


图 6　近程技能程序流程图

同时，在使用技能时，动画器中对应技能动画的 Trigger 将被触发，使该物体的动画过渡到技能动画。上述实现中的部分功能，如生成预制体，enable 触发器等，将通过技能动画中的帧事件实现（即在播放到对应帧时调用脚本中预先设置的方法）。

敌人的技能释放由脚本自动控制，每隔一段时间按预定顺序触发技能。玩家的技能使则通过按下对应按键实现，这一过程与玩家的移动类似，同样通过 unity 的输入管理器和 Input 类中的方法实现。在此不再赘述。

2、人物与动画：

（1）人物与动画制作：

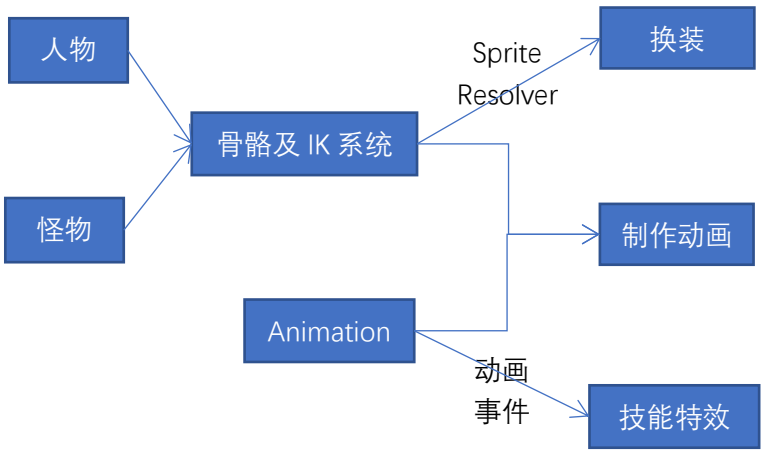


图 7　人物与动画制作逻辑图

为实现换装与武器的切换，将人物的图片素材分层并添加了骨骼和 IK 系统。对不同人物服装的图片添加完全相同的骨骼，存储于 Sprite Library Asset 对象中，并对应地为人物身体各部分添加了 spriteresolver 组件以实现换装效果。

同时，借助创建的骨骼，使用 animator 中的录制方法，制作了人物从跑动到攻击的各类动画。此外，在动画中通过对编写的各种动画事件的调用，实现了各种特效。

此外，将服装也创建为一个 scriptable 类型的类 Costume，存储服装名字，在切换服装时，获取 Player 下的所有 spriteresolver 组件，通过从 Costume 中获取的服装名字修改 spriteresolver 的图片。

游戏中的两个 boss 也做了相似的处理，从而设计出移动与攻击动画，在此不做赘述。

（2）动画过渡

游戏中动画过渡依靠 unity 提供的动画器。在需要动画过渡时，脚本中将调用 SetInteger(string name,int value)，SetBool(string name,bool value)等方法改变动画器中对应的参数。动画器则实时进行检测，当参数满足过渡条件时即过渡到对应动画。

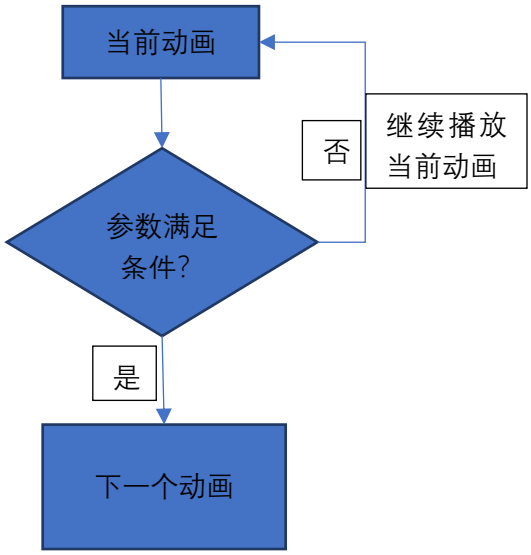


图 8　动画过渡程序流程图

3、数据库与存档：

（1）数据库

采用 scriptableobject 存储开发数据，主要包括武器库（用于提供攻击模式、武器外观、武器属性等多种数据），任务库（用于提供任务的完成条件、奖励以及任务之间的牵连关系等）、敌人库（为敌人提供数值等信息）等等。由 scriptableobject 派生出来的类无需依托于 inspecter 上的物体，可以独立存在，并且可以可视化地调整和配置各种游戏数据。采取这种方法存储开发数据，可以分隔开游戏数据与代码，使得游戏开发更有序、更方便。

（2）存档与读档:

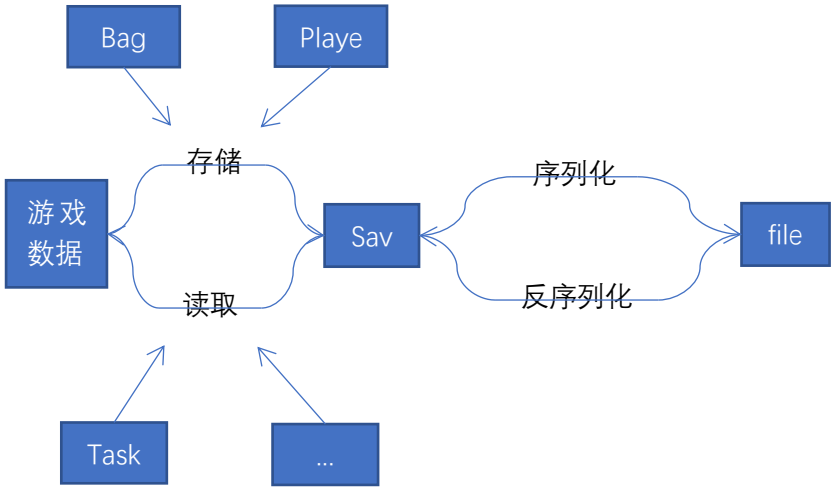


图 9　存档与读档逻辑图

创建 Save 类，用于存储各类数据。在 SaveFunction 脚本中写了 savegame()和 loadgame()两个函数作为对外的存档与读档的接口。在所需存储的数据的对应脚本（玩家信息，背包信息，任务信息）中添加了对应的存档与读档方法，并封装于 SaveFunction 中。整个过程的信息依托于所创建的 Save 类的变量 save，在 savegame 中对其进行修改和序列化存入二进制文件，并在 loadgame 中从文件中反序列化得到 save 变量，用于各类数据的读取，具体的存取方法会在后续的对应部分展开。

为了方便起见，存档的时机选在了每一次加载另一场景中，而读档则选在加载本场景时。前者通过在 loading 脚本（后续会展开）中调用，后者则在 Player 脚本的 Start 方法中调用。值得一提的是，由于加载的顺序问题，这里存在一个非常严重的难以获取到游戏对象的问题，同样在后续展开。

4、UI:

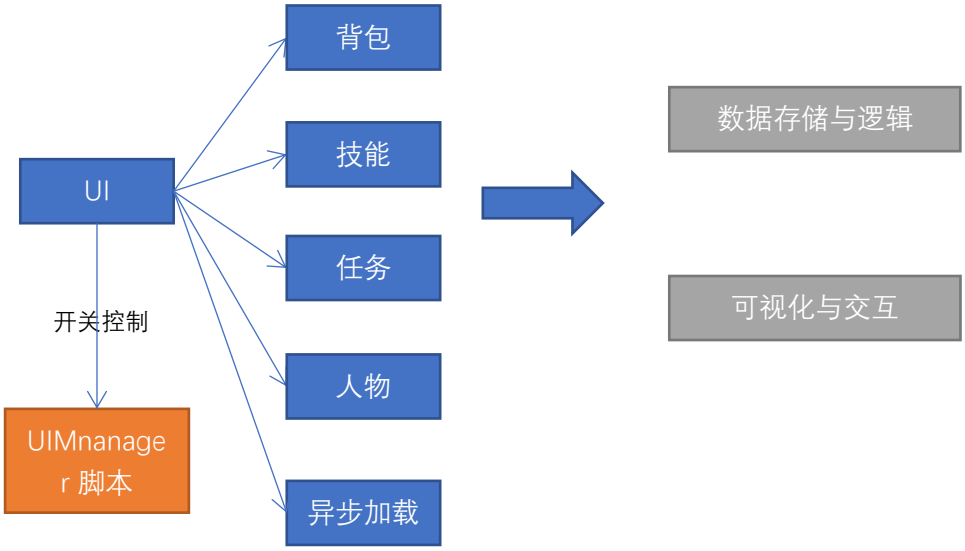


图 10　UI 总逻辑图

（1）UIPanel 的开关控制：

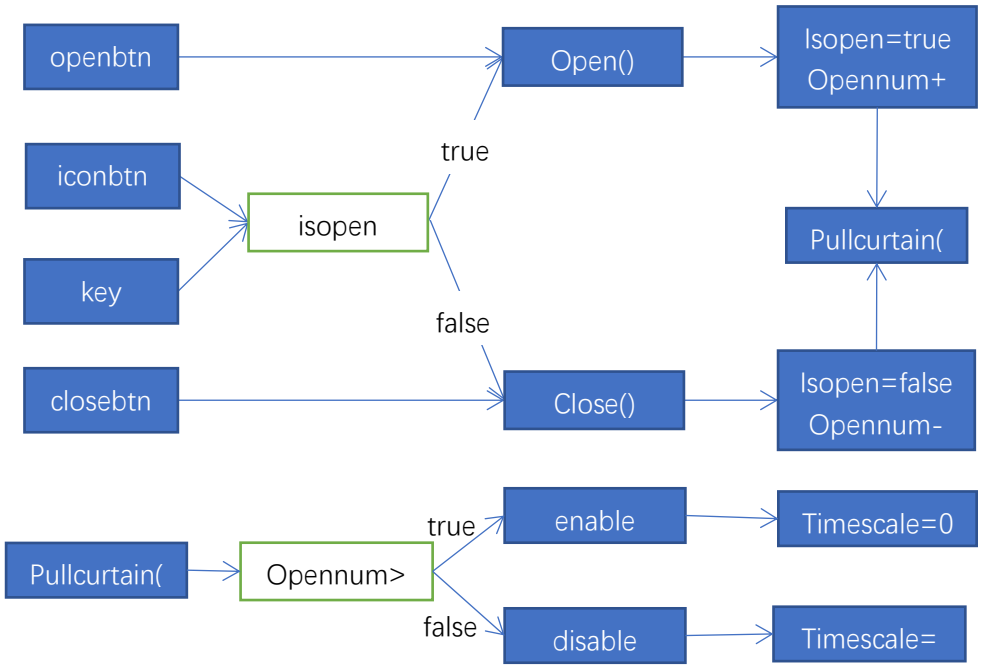


图 11　UIPanel 的开关控制逻辑图

我们采用统一的脚本来控制 UI Panel 的开关。我们写了 UI_Manager 脚本并拖拽到了 canvas 上，作为总的控制器，并为每个 Panel 写了 UI_1 的脚本（意为第一种模式的 UI），在其中存储对应 Panel 的按钮信息。在 UI_Manager 中，我们创建了一个 List<GameObject>，Add 各个 Panel，然后使用循环来为从 Panel 的 UI_1 脚本组件中获

取的各个按钮添加监听事件。另外，由于不同 Panel 的按钮情况不一定相同，我们添加了对按钮是否为空的判断，防止出错。

监听事件根据功能分为了三类，open、close、icon、key 为了使得 icon 正常使用，我们为每个 Panel 额外添加了一个 bool 变量 isopen，在使用按钮时，对其进行修改，并根据其值选择 icon 按钮和 key 的行为。

此外，我们希望在玩家打开这些 Panel 时，能够暂停游戏进程，并且使游戏画面略被遮挡（我们称之为 curtain）。允许玩家同时打开多个 Panel，如果简单地将 Curtain 的开关控制放在 Panel 的开关监听事件中，势必会引起混乱。例如：打开 Panel1 后，再打开 Panel2，这时，如果关闭 Panel2，Curtain 就会关闭，但这是不合理的，因为 Panel1 仍处于打开状态。

（2）背包：

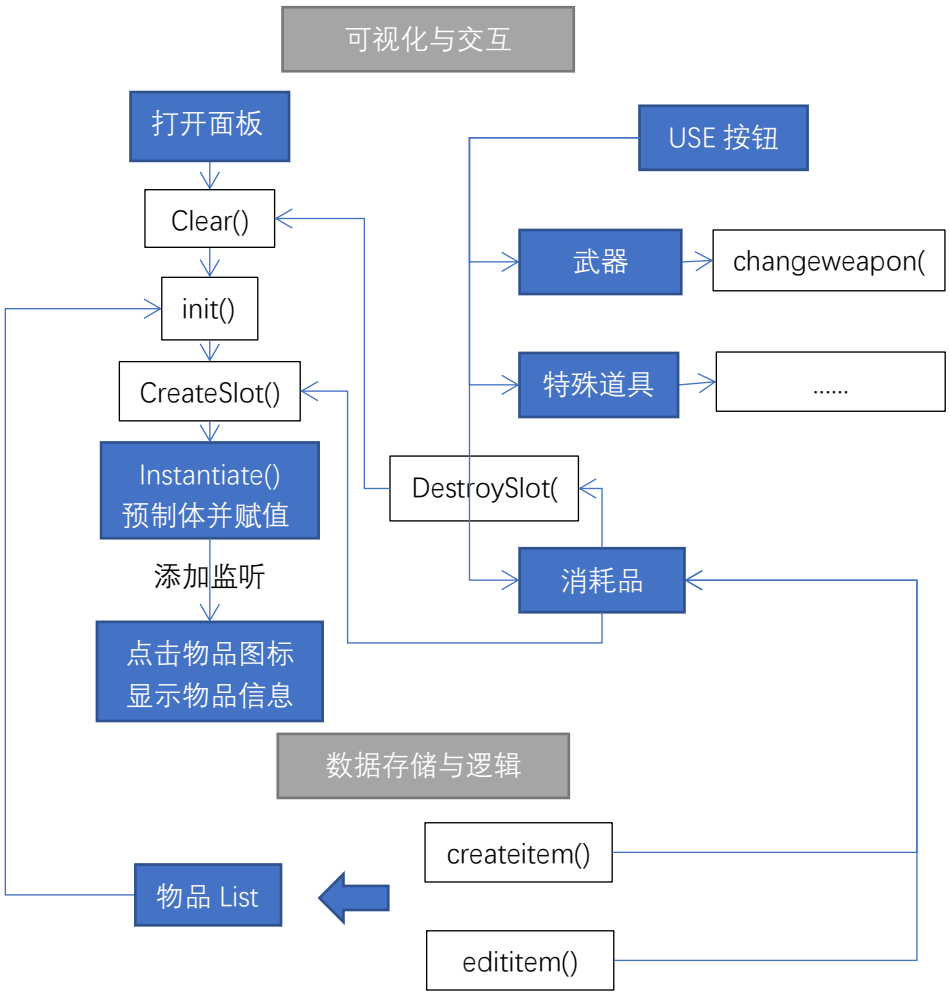


图 12　背包的逻辑图

在人物下挂载 MyBag 脚本以存储人物所持有的物品信息。在其中分别写了 createitem()与 edititem()两个函数，分别用于创建新物体和改变物品数量（增加或减少）。

通过 instantiate 背包格子的预制体，并将其作为带有 Grid Layout Group 组件的背包方格区的子物体，使其自动排版。对生成的背包格子进行赋值，并对其上的按钮添加了监听，使得其在点击后更改选中物品并显示物品信息。在背包打开时调用 clear()，并调用 init()初始化。

对于 Use 按钮，通过所选物品的类型（存储于 Item 数据中），选择使用的模式。消耗品调用物品增减的函数，并调用其他函数来实现其功能；武器则通过调用 Player 脚本中编写好的切换武器的函数，切换武器图像并修改数值，改变模式。

（3）人物数据显示：

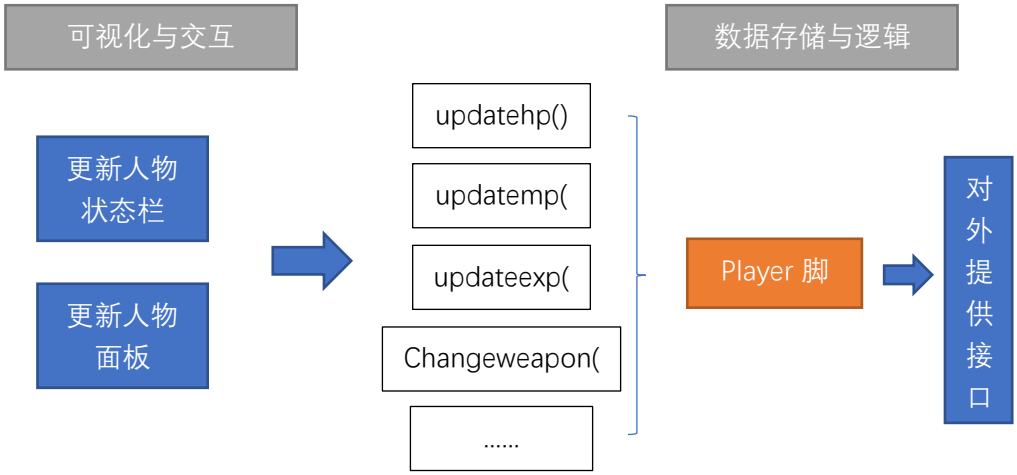


图 13 人物数据的显示逻辑图

考虑到人物的数据并不是时时都要更新，故为节省运行资源，在 Player 脚本（人物的信息存储中心）中写了若干个更新各项数据的方法（如 updatehp()，updatemp()等），调用于需要改变人物数据的函数（如受到伤害等）中。这些更新数据的方法封装了包括改变人物面板中的数据显示，人物血条蓝条等的显示在内的多种在其他脚本中编写的方法。这一封装很好的简化了对于与这些人物信息相关的 ui 的更新操作，并且方便了其他开发人员对人物数据的修改行为。

（4）技能：

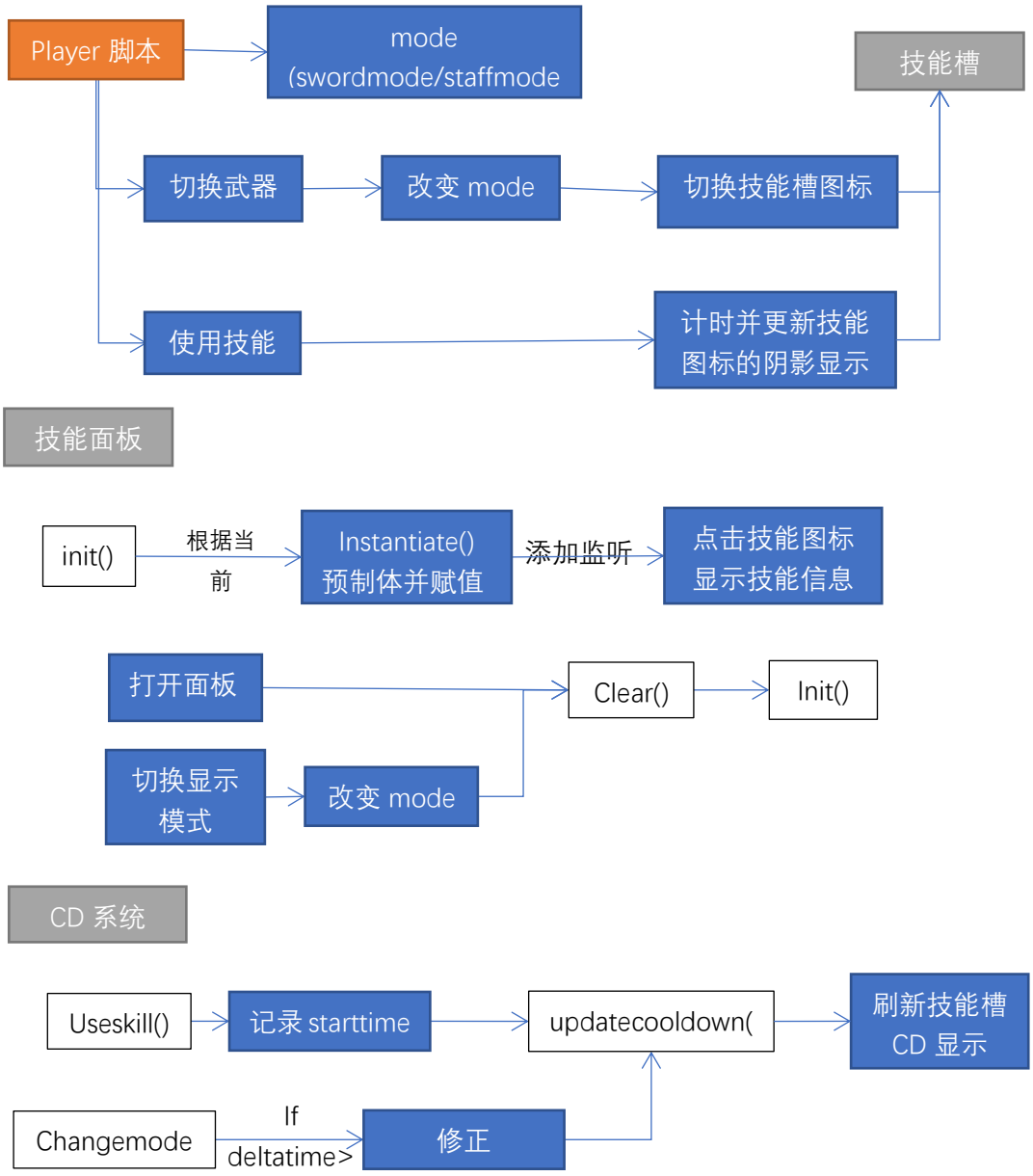


图 14 技能逻辑图

由于人物有 sword 与 staff 两套攻击模式，但其攻击与技能公用一套按键，故在 Player 中用枚举变量 mode 加以标识。

技能面板主要也是采用 instantiate 预制体并赋值、添加监听事件的方法制作的。打开面板时，会根据当前的 mode 显示对应的技能，并默认显示第一个技能的信息。同时，可以通过点击 sword 与 staff 按钮来切换显示的技能的模式，也可以通过点击技能图标切换展示的技能。

左下角的技能槽则为了节约资源，将其图标切换封装为 changmode()方法，并在人物通过更换武器切换攻击模式时调用。

此外，我们加入了 CD 系统，在使用技能后，将记录技能释放时间 starttime 并计算剩余时间 deltatime，通过 deltatime 与 0 的比较来判断冷却是否结束，并不断更新技能图标的 fill 属性值来制造出 CD 图标效果。为了使得切换模式后 CD 系统仍能正常显示，我们也采取了一些小技巧，即：对每个技能单独计时，在切换技能后，用公式 starttime = Time.time + deltatime - cooldowntime 来修正等价的开始时间，以保证技能之间的 CD 不互相

（5）任务系统：

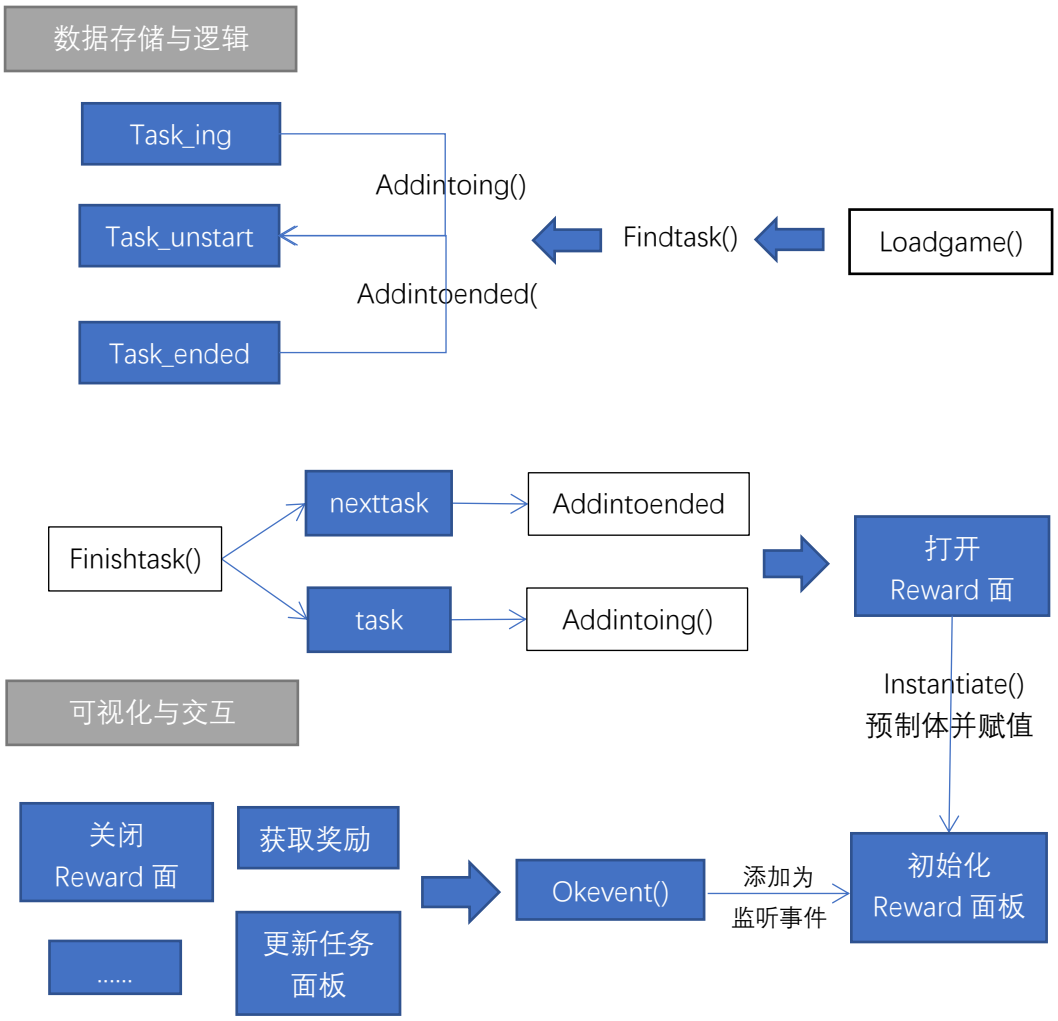


图 15 任务系统逻辑图

在脚本 TaskManager 中，将数据库中的任务分为三类存储于 List 中：进行中、未完成、已完成(task_ing，task_unstart，task_ended)，其中，task_ing 因为需要包含任务是否完成的信息，故额外创建了一个结构来进行封装。

初始化游戏时将任务全部放入 task_unstart，并写了 addintoing(),addintoended()两种方法（接受 int index 参数），分别用于将任务从 task_unstart 中转移至 task_ing 和 task_ended 两个数组中，通过这两种方法，实现了对存档中 task 部分的信息的读取。

任务面板的显示通过 init()和 clear()两个方法来实现：init()循环地读取 task_ing 中的任务，调用 instantiate()方法在任务槽上生成任务的预制体（taskprefab），并对该预制体的各项数

据进行赋值，为其按钮组件添加显示任务信息的监听事件。Clear()则是 destory 任务槽中的所有对象。在每次打开面板时分别调用 clear()与 init()方法，实现任务面板的刷新。

此外，我们写了方法 finishtask()用于实现任务奖励的获取、后续事件的触发等多项功能。为 Reward 按钮添加监听事件，在当前任务完成的前提下，打开 reward 面板并展示奖励（同样通过 instantiate 预制体并赋值的方式），在 okevent()中调用 finishtask()并关闭 reward 面板并获取奖励，添加为 ok 按钮的监听事件。

（6）异步场景加载：

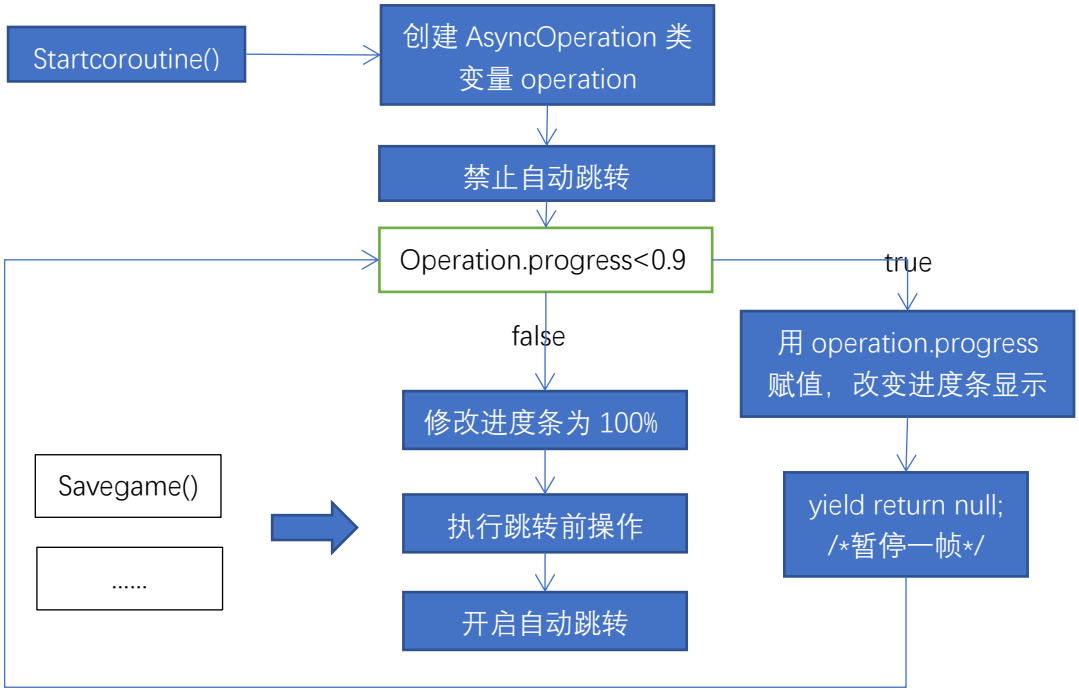


图 16 异步场景加载程序流程图

采用异步加载和协程的方法加载场景，创建 AsyncOperation 类变量 operation，在加载过程中用携程将 operation.progress 的值传给进度条的 slider 组件，以实现进度条的移动。

为了防止加载完下一个场景后立即跳转，我们在开始加载场景时就将 operation.allowSceneActivation 设置为 false。但是，后果是如果不将其设置为 true，operation.progress 的值会停在 0.9f 不动，即无法检测到场景是否加载完成。采取的解决方法是，当 operation.progress 到达 0.9f 后就退出更新的循环，直接设置 slider 的值为 100%，并且在执行完场景跳转前的操作后，将 operation.allowSceneActivation 设置为 true。

场景跳转前的操作中，值得一提的是存档，由于在任意情况下加载到下一场景时都调用 savegame()，会导致在由主菜单加载到游戏场景前调用 savegame()，覆盖了原有存档。故设置了静态 bool 变量 ismainmenu 加以标识，并在 loading 脚本执行 savegame()前加以判断。

（7）对话系统

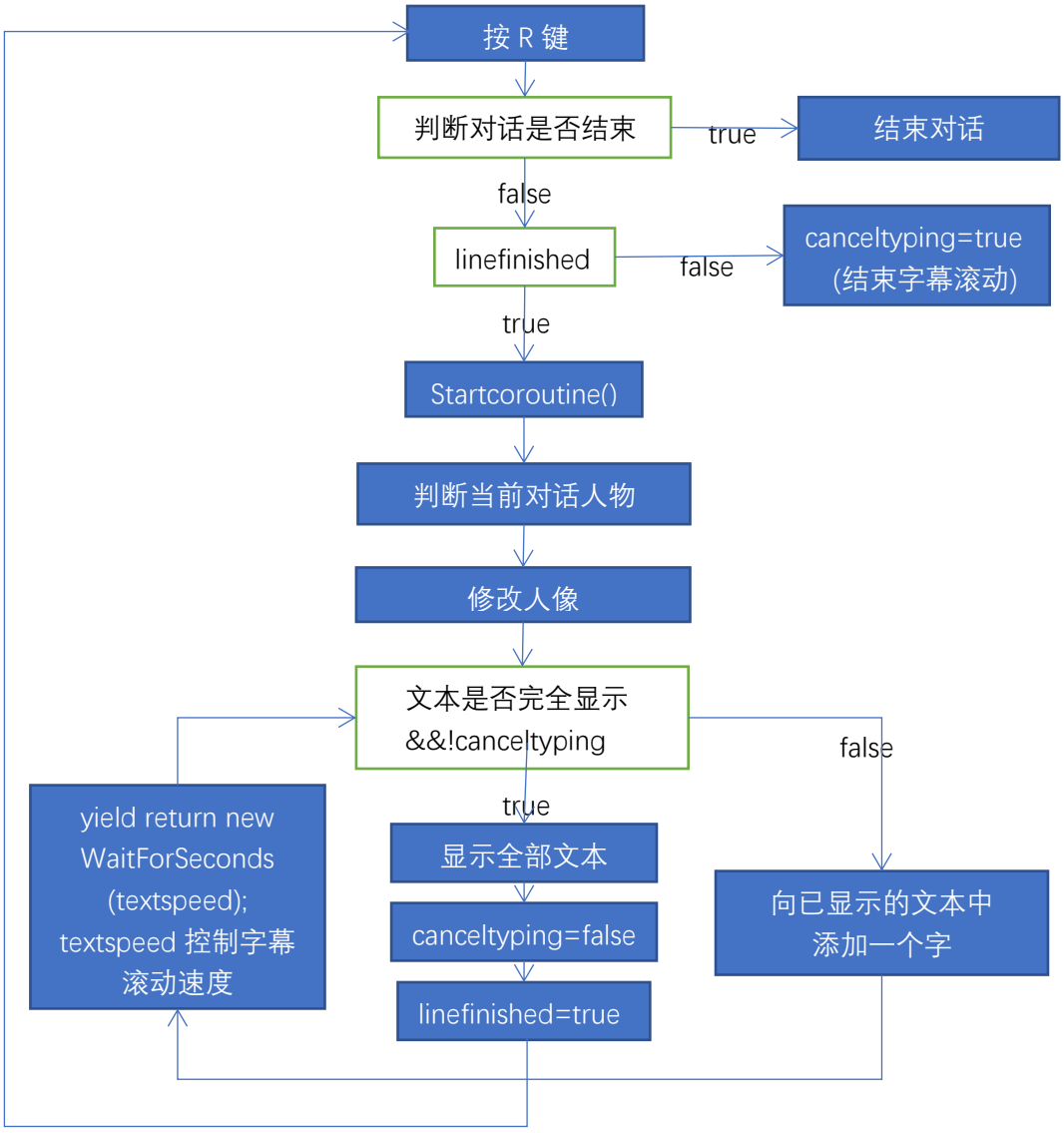


图 17 对话系统程序流程图

用 linefinished 和 canceltyping 两个 bool 值分别判断 本行是否显示完毕 和 是否结束字幕跳动，直接显示文本。按下 R 键，若尚未显示全部文本，则结束字幕跳动，否则，跳转至下一行对话。字幕跳动通过异步实现，每次显示一个字，通过 yield return 等待 textspeed 时间，再继续显示。

文本格式为：一行 对话人物 接一行 对话内容 ，可以方便地在每次显示对话内容前判断当前对话人物，以进行更换人像等操作。

6、摄像机跟随

玩家的视角由 main camara 控制，因此改变 main camara 的位置即可实现跟随。在游戏中，camara 的脚本每帧调用一次 Lerp(Vector3 a,Vector3 b,float t)对 Player 物体和 camara 的坐标进行插值，得到一个在玩家坐标和 main camara 当前坐标之间的一个新坐标。随后将得到的新坐标赋给 main camara，这样就实现了略有延迟的 camara 平滑跟随效果。（该过程较为简单，故图略）

五、效果展示：

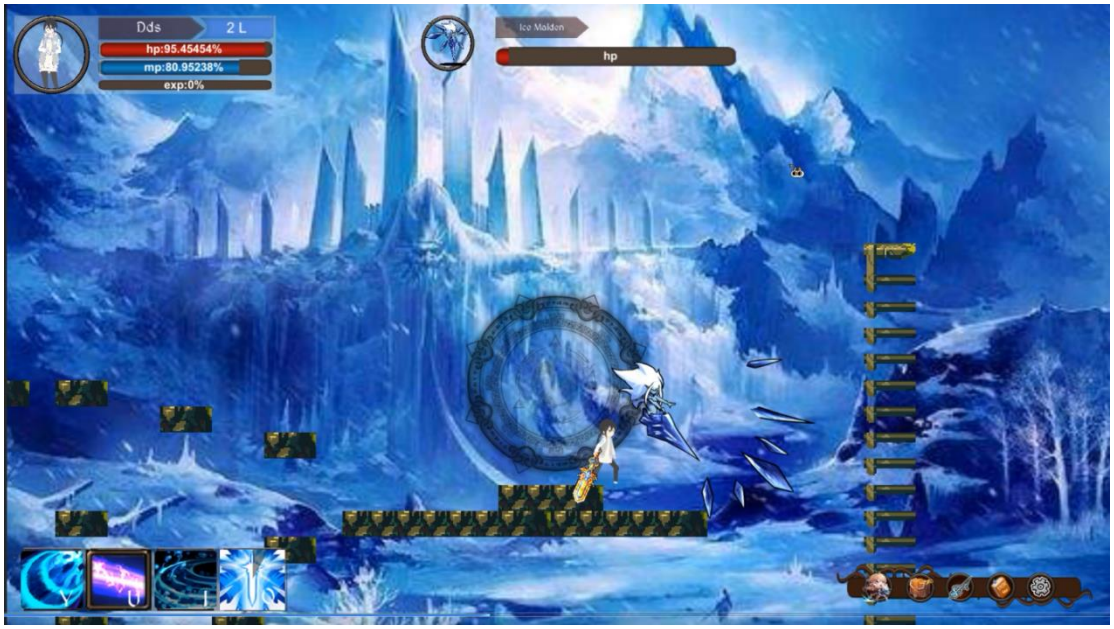


图 18 战斗场景 1

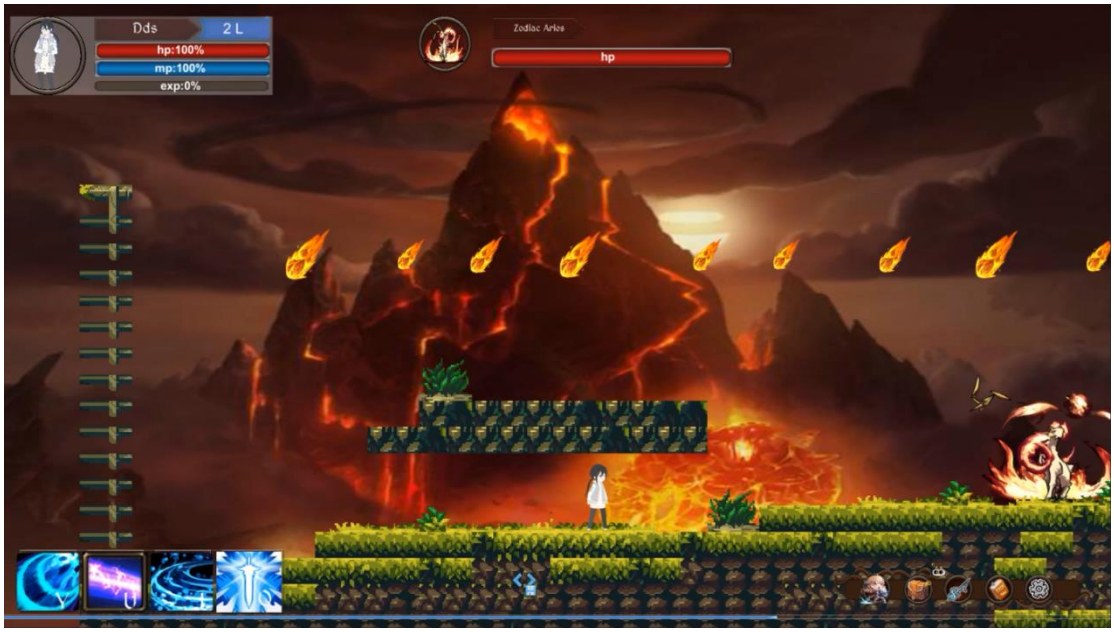


图 19 战斗场景 2

图 20 战斗场景 3

图 21 加载场景

图 22 对话

图 23 背包界面

图 24 任务界面

图 25 人物界面

更多游戏截图请见压缩包中的“游戏截图”文件夹。另外，压缩包中还附带了一个宣传视频，若感兴趣可自行观看。

六、问题解决：

1、存档中的问题：

无法将 Item、Task 等由 ScriptableObject 派生而来的类的变量进行序列化。因为 unity 会自动将这个继承 ScriptableObject 的类序列化，所以这些类的变量无法被再次序列化。而

由于这些变量在真机上无法被修改，故也无法用于存储。综上考虑，我们最终采取这样的替代方案，即：在 Save 中用 string 类变量存储对应 name（或用 int 变量存储 Index），并在对应的脚本中写一个按 name 或 index 寻物的方法。

另一方面，一些存储数据应同时绑定官方的数据与玩家的数据两种信息，如，task_ing 由 Task 与标识是否完成的 bool 值两类数据组成，Iteminbag 由 Item 与标识拥有数量的 int 组成。由于在 C#中，不同类中声明的结构不互通：不能强制转换（即使完全相同），也不能声明在另一个类中定义的结构，最终采用了用函数在不同类间传递信息的方法，即：将所要存储的结构中的信息作为参数传递给 Save 类中定义的创建 Save 中对应结构变量的函数，有效地解决了问题。

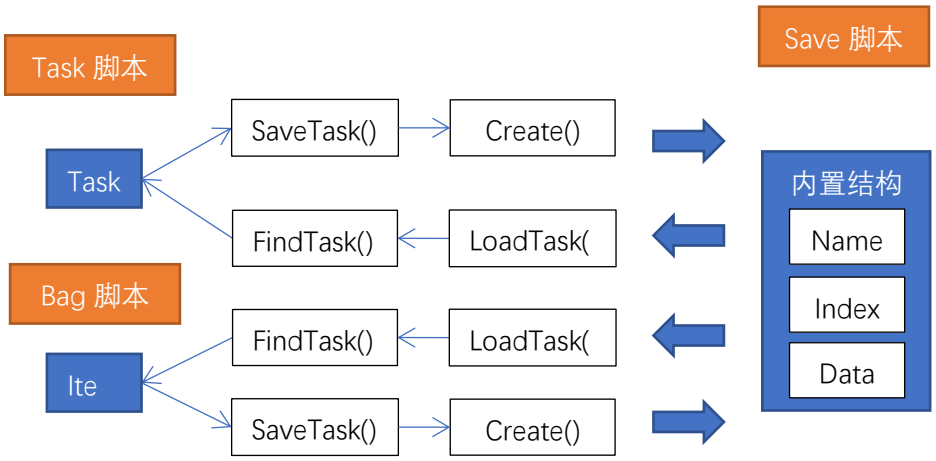


图 26 存档问题的解决

2、加载：

Unity 中的各种游戏对象和行为存在一个加载顺序，当工程不大时，这一特点并不会明显的展现出来；但当工程量增大，尤其是各个脚本间存在频繁的相互调用时，会产生很大的麻烦：即在调用另一对象时，可能会因为其尚未加载而无法获取到对象，导致游戏出错。各类函数在每一帧的执行顺序如下图：

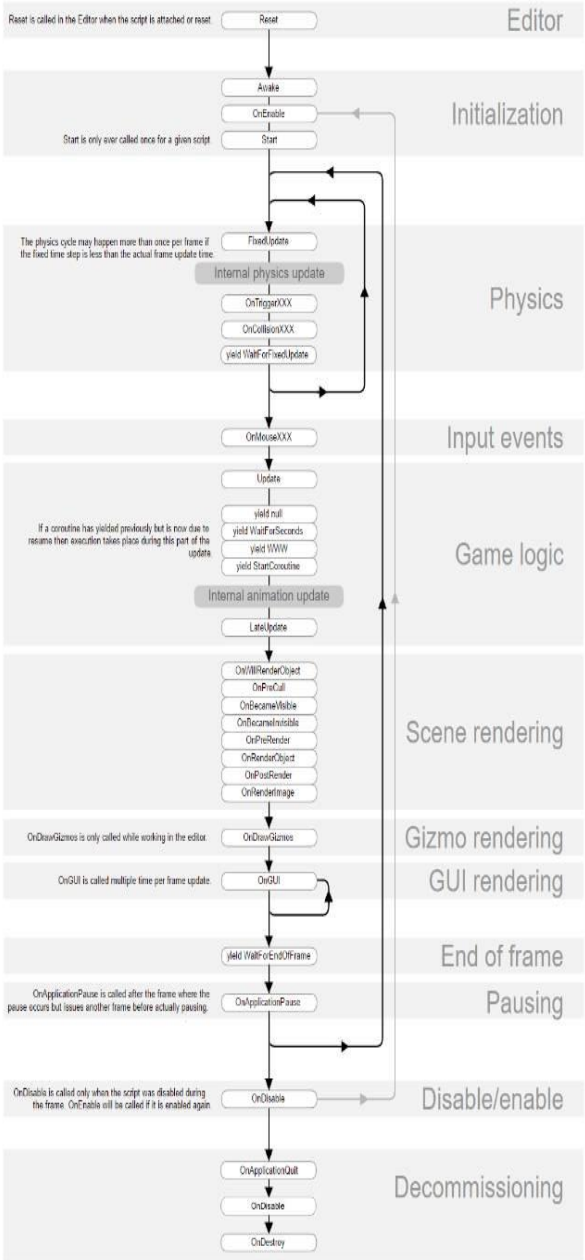


图 27 Unity 加载顺序

为了解决复杂的获取不到对象的问题，我们一方面尽量规范地书写代码，即尽量将对其他对象的调用写进 Start()函数中，将自身脚本的实例化（大多为为 Instance 单例赋值，用于替代声明 static 的变量和方法）写入 Awake()函数中。但这并不是万全之策，毕竟游戏对象的加载顺序难以预测，所以我们采用协程的方法作为保护机制，主要的逻辑如下：

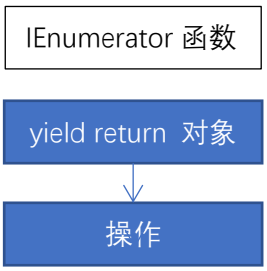


图 28 Unity 异步原理

其中，“yield return 对象”，会在对象启用后执行其后的代码，这样就很好地避免了获取不到对象的问题。当然，这也可以通过循环地开启协程的方法（事实上我们一开始就是这样做的），在此不过多展开。

3、delegate 的使用：

在为按钮添加监听事件时，总是绕不开 delegate 方法。但当我们尝试着循环地使用 delegate 对按钮添加监听时，却惊讶的发现，每个按钮都添加的是以循环的数组中最后一个元素为参数的事件。经过查阅资料，我们发现，对于 delegate 方法委托执行的代码块中，循环变量是不会被替换的。最终的解决方案是：在循环里创建另一临时变量并用循环变量赋值，这样就可以为各个按钮添加正确的监听事件了。

4、敌人无故消失：

在设置出生点时不慎将 z 轴坐标设置为了一个负数（即在 scene 的后方），导致敌人出生点已生成敌人的预制体，敌人可以与玩家发生正常的攻击、碰撞行为，但是在游戏场景中看不见敌人。

这是因为 unity 本身是为 3D 项目开发设计的引擎，因此 2D 项目实际上是被放在 3D 场景中的一个 2D 平面。同时 Z 轴上位于 scene 后方的物体会被遮挡，但物体间交互时却不考虑 z 轴坐标——这就导致场景中不显示该物体，但 object 间的交互正常。最终我们将出生点 z 轴坐标设为 0，解决了这一问题。

5、多段普攻的实现：

在进行玩家的多段普攻时，角色播放完第一段攻击动画后总是直接跳回静止/移动动画，无法进入下一段普攻动画。

经过排查后我们发现，在设置从攻击动画到静止/移动动画的过渡时，只设置了判断当前是否在移动/跳跃状态下的条件，而未设置当前是否在进行攻击的判断。这导致每次播放攻击动画时，过渡到静止/移动动画的条件从一开始就总是满足的。因此即使玩家即使在攻击动画结束前按下攻击键，动画也会因先满足过渡到静止/移动动画的条件而不过渡到下一段攻击动画。

最终，我们设置了一个“isAttack”参数，在攻击动画正在播放时将其设为“true”，在攻击动画的最后一帧通过帧事件将其设置为“false”。并将“isAttack”为“false”设置为过渡到静止/移动动画的条件。这样就可以确保攻击动画优先过渡到下一段攻击。

六、总结

经过组内成员的通力协作，开发者已基本完成 USTC Soul Tales 的开发。游戏已经具有了菜单界面，存档系统，人物行为，敌人 AI，剧情等功能。虽不算精致，但已经作为一个游戏的完整框架和一定的可玩性。

另外，开发者在设计游戏代码时，大多采用可复用的设计理念，通过复用性较强的代码，搭建了一个易于拓展的游戏框架。

在开发过程中，开发者们深深体会到了开发大型软件与编写简单程序的不同，也感受到了良好的编程习惯和清晰的算法逻辑的影响力。对每一个模块的编写和测试也大大拓宽了我们的视野，使我们的计算思维有了进一步提升。编写游戏的过程中各类 bug 的出现也反映出了我们自身的很多不足，并使我们在解决 bug 的过程中得以弥补。开发者们经过研究与开发，提升了能力，丰富了认知，改正了不足。

总而言之，本次开发过程对我们而言是一个较大的挑战。通过这一次作业，我们很好地巩固和提升了能力，应用了自己的学习成果。在未来的学习中，我们会继续提升自己，完善程序设计的思维和能力。